Using Deep Learning Natural Language Processing to Predict Software Vulnerabilities

Presented to the Faculty of the School of Engineering and Applied Science

University of Virginia • Charlottesville, Virginia

In Fulfillment of the Requirements for the Degree

Bachelor of Science in Computer Science, School of Engineering

Ethan M. Gumabay

Spring 2020

Department of Computer Science

Signed: _____

Approved: _____ Date _____

Yuan Tian, Assistant Professor, Department of Computer Science

1 Abstract

As technological advances propel modern society towards a seemingly utopian future, an increasing number of software vulnerabilities are discovered every year, whether reported publicly or discovered internally. As of April 2019, C and C++ were the most common languages found in modern software [1]. The reason for this is that C and C++ were some of the first languages developed; unfortunately, both languages were vulnerable to attacks that were made infeasible in later programming languages such as python and java. In an effort to combat this problem, I leveraged a large dataset of C and C++ code containing known vulnerabilities in order to develop a vulnerability detection system using deep learning and natural language processing. I focused on classifying the most common vulnerabilities in C and C++ by grouping code snippets into six categories: CWE-119, CWE-120, CWE-469, CWE-476, CWE-other, and not vulnerable. The aim of my research is to accurately predict whether or not a never-before-seen code snippet contains a potential software vulnerability. Ideally, this system could be deployed on a large scale as a modern intrusion detection system. The result of my research is a deep ensemble classifier that consistently and accurately predicts whether a piece of code contains a software vulnerability with over 90% accuracy.

2 Introduction

Software products are used every day by millions of people all over the world, now more than ever before. The code that actually allows this software to function, however, is almost never shown to the user. In fact, the use of black-boxed software is often one of the primary goals of modern technology. This is rather unsurprising; users want software that allows them to simplify their life without having to think about how it actually works. A calculator, for example

is a tool that many people use every day all over the world to simplify the task of performing mental computation, but very few people know how a calculator actually works. The same idea applies to software, which had led technological development to prioritize usability. Even if the user can see the code that makes their software work though, it is very unlikely that an average user will understand why their software works or does not work. In the best case, an experienced software developer can look closely at the code behind the software they are using and truly understand how it works. Even in these cases, however, it would be extremely challenging for a software developer to detect a software vulnerability with the naked eye. It is often the case that a software vulnerability needs to be intentionally searched for and takes careful inspection to detect. Ultimately, software vulnerabilities are most often completely invisible to the user, regardless of expertise. For this reason, it would be extremely useful to develop a tool that automatically inspects software for potential vulnerabilities to protect users from otherwise undetectable threats.

There are many tools that attempt to detect potential software vulnerabilities, but too often are constrained by a specific set of predictable rules. Many viruses, for example, work by pushing something to the stack directly before a function return. Virus detectors, then, were able to thwart this by simply ensuring the line directly before a return was not a push [2]. This trivial method of detection is rather easy to circumvent, simply pushing and then adding a series of nop instructions will give an attacker the desired behavior. For this reason, modern vulnerability detection systems are insufficient for effectively finding and preventing latent software vulnerabilities.

One technique that has revolutionized many modern computing problems such as image classification, outlier resolution, and sentiment analysis is machine learning, specifically deep learning. Deep learning is subfield of machine learning that focuses on the use of artificial neural networks modeled after the neural networks found in the human brain [3]. Deep learning has the ability to detect and learn on features of the data that are completely invisible to the human eye. Deep natural language processing aims to build neural networks that are able to analyze and represent human language [4]. Although deep natural language has traditionally been used to perform sentiment analysis on news articles or tweets, it appears to have the groundwork needed to analyze and represent code in the same way it does for human language. Specifically, it should be able to interpret code the way it interprets human language and predict vulnerability as opposed to sentiment.

Similar to a deep natural language processing model, the model I built is only able to handle a few different classification labels. The classification labels I chose were from the Common Weakness Enumeration (CWE) database [5]. Specifically, I chose to analyze CWE-119 (out-of-bounds failure), CWE-120 (buffer overflow), CWE-469 (invalid pointer), CWE-476 (null pointer dereferences), and CWE-other as a catch-all. The four vulnerabilities combined with a catch-all are common enough to be a sufficient set of classification labels for a proof-of-concept.

3 Related Work

There are many tools that are currently used to uncover common software vulnerabilities. Clang proposed a memory model for static analysis of programs which would not require direct execution in 2010 [6]. This model, however is a rule-based system that can be defeated simply by understanding the constraints the model checks. King proposed a different system that worked by

converting the input vectors into symbolic representations that could be analyzed over the control flow graph of a program [7]. This method, although functional for malware that alters the execution of a program, is extremely expensive since it must exhaustively model all possible execution paths. For this reason, it does not scale well for large, complex programs.

One extremely relevant work was published in 2018 by Li et al. [8]. Li et al using deep learning for vulnerability detection to achieve almost 94% accuracy on some vulnerabilities in a model called VulDeePecker. VulDeePecker employs deep natural language processing to detect vulnerabilities. One limitation, however, was the use of the Long-Short-Term Memory (LSTM) cells that allow the network to learn long term dependencies in the data. The limitation pointed out by Li et al is that sometimes the behavior of code depends on what codes comes before and after the present snippet, but LSTM's are unidirectional. This means that the context can only be passed from earlier cells to later cells, and might therefore be insufficient in performing vulnerability detection. Li et al instead suggest the use of BLSTM, or Bidirectional Long Short-Term memory cells.

Lastly, Russell et al published a model also using deep learning for vulnerability detection in November of 2018 [5]. The system developed by Russell et al used a massive codebase with over 12 million functions to be analyzed. The massive dataset yielded an accuracy great than 95%. The strategy to build this network was very similar to the typical strategy employed for deep natural language processing; construction of a language model with word embeddings used to build a separate classifier to predict labels. The research presented here, however, trained on an extremely large dataset with a majority of the code labeled as not vulnerable. This could have potentially led to a slight overfitting of the data; the model may have

assumed that everything was not vulnerable since over 90% of the training data was marked as such.

4 Methodology

The vulnerability detection system I built contains three distinct parts. In the first stage, data is passed to a language model in order to teach the model what the input vectors will look like. The language model tokenizes the data and usually creates word embeddings, though since the input was not English words in this case, the embedding scheme was based more around syntax. For example, the int token was likely close to the x token since int x is a fairly common declaration in C/C++ code. Once the language model tokenized the data and created the feature embeddings, the data could then be passed to the classifiers.

The vulnerability classifiers are the second part of the system I constructed. Russel et. al trained a single classifier to recognize all of the potential vulnerabilities, which may have possibly led to misclassification of particular vulnerabilities in some cases. To combat this, I created a separate classifier for each vulnerability. This way, I could use five distinct models to recognize the five distinct vulnerabilities. I passed a dataset to each of the five models containing 50% vulnerable samples and 50% benign samples. Theoretically, this should train each model as a binary classifier for vulnerable or not vulnerable to a specific known vulnerability.

The final part of my model was an ensemble learning method to combine the predictions of all five classifiers. Every instance would be passed through each of the five classifiers. The classifiers would then return a confidence score to represent its confidence that a snippet was vulnerable or not vulnerable with respect to the specific vulnerability that particular classifier was trained on. Once the confidence scores from each model were recorded in the final phase, the model classified each instance according to the model which recorded the highest confidence

score. If no model reported a confidence score of over 50% for its specific vulnerability, the model classifies the sample as benign.

5 Experiment

In order to build a vulnerability detection system using deep learning, data needed to be acquired, analyzed, preprocessed, and passed to a language model. Once the language model has been trained on the vocabulary of the dataset, it can then be used to help a text classifier to understand input and accurately predict vulnerability of code snippets. In order to combat the potential problem of overfitting, I built five separate classifiers for the five different labels. I then employed ensemble learning, passing each code snippet to all five models, each predicting vulnerable or not vulnerable in order to determine whether or not a particular code snippet was vulnerable at all.

5.1 Data Acquisition and Analysis

The data was acquired from https://osf.io/d45bw/, which contained an extremely large dataset of over 12 million code snippets separated into training, validation, and testing sets. Due to time and resource constraints, a dataset of this size was not feasible for this research project. I partitioned this massive dataset and used about 150,000 snippets separated into 120,000 in the train set, 15,000 in the validation set, and 15,000 in the test set.

After partitioning the original dataset, I ordinally encoded the data so that vulnerable was represented by a 1 and non-vulnerable was represented by a 0. I then analyzed the dataset for frequency of each vulnerability. I found that the CWE-476 vulnerability was underrepresented with only about 1000 samples of vulnerable code. Although I considered truncating the other vulnerability datasets to this same size, I ultimately concluded that the underrepresentation of the CWE-476 vulnerability is consistent with the vulnerability's frequency in modern software. For

this reason, I trained each classifier with the proportion of data on that vulnerability within my dataset. The composition of my dataset is shown below in table 1.

Vulnerability	Snippets	Percentage (%)
CWE-119	2453	1.9
CWE-120	4891	3.9
CWE-469	3128	2.5
CWE-476	1192	0.9
CWE-other	3490	2.7
Benign	112266	88.1
Total	127419	100

Table 1: Dataset

The vulnerabilities here make up a very small part of the dataset, which led me to build individual classifiers for each vulnerability. Training a single model may have failed to detect subtleties between the various vulnerabilities.

5.2 Language Model Development

The first step in building a successful natural language processing system is the construction of a language model. Language models are used to convert the human-readable input to machine-readable vectors so a classifier can effectively predict a label. Language models handle word embeddings and tokenization of data so that the input data can be understood and processed. I used the fasta.ai's language_model_learner to build the language model. The language_model_learner employed by fast.ai is a transfer learner wrapper that creates an instance of a sequential recurrent neural network. Since the language model is not used to predict actual vulnerability but simply to understand the language (in this case C/C++ code snippets), the overall prediction accuracy of the model is inconsequential. As long as the model learns the

language at all and can form coherent tokens, it will be sufficient in helping the vulnerability classifier.

5.3 Individual Classifier Development

One of the possible limitations of the work published by Russel et al is that one classifier is expected to recognize all five sets of vulnerabilities [5]. In an effort to solve this problem, I built five individual classifiers for each of the vulnerabilities I was attempting to detect.

I extracted the vulnerable snippets for each vulnerability and created a smaller dataset to pass to each vulnerability-specific dataset. Each of these datasets is comprised of half benign and half vulnerable instances of the given vulnerability. The breakdown of these datasets is shown in table 2.

Vulnerability	Train set	Validation Set	Test Set	Total
CWE-119	3923	491	491	4905
CWE-120	7825	978	978	9781
CWE-469	5006	625	625	6256
CWE-476	1907	238	238	2383
CWE-other	5584	698	698	6980

Table 2: Vulnerability Datasets

The datasets are different sizes, so it is possible that some classifiers will learn better than others as a result of the amount of data it is given, but the breakdown of vulnerabilities between datasets here is a rough approximation of its true frequency in practice.

Once the data had been partitioned into subsets to represent each vulnerability, I passed each classifier the dataset for one specific vulnerability as well as the language model vocabulary. The language model was trained on the larger dataset, which handled feature embeddings and tokenization so each classifier had sufficient information about the input data to

make an informed classification on each of the instances. I trained each of the classifiers for 5 epochs with a learning rate of 0.1. I then employed fasta.ai's find_lr() method to determine the optimal learning rate for each model. Once I adjusted the learning rate for each model, I trained each classifier for 5 more epochs. The final prediction accuracy of each binary classifier is shown in table 3.

Vulnerability Classifier	Prediction Accuracy (%)
CWE-119	57.1
CWE-120	60.2
CWE-469	55.3
CWE-476	56.4
CWE-other	53.2

Table 3: Classifier Accuracy

Each of these classifiers was binary, so the expected accuracy of a random guess would be 50%. From the table above, it is clear that all of the classifiers exceeded this mark which indicates that they are learning to recognize vulnerabilities better than simple random accuracy. It is likely that further fine running of hyperparameters for each model would result in an accuracy greater than 65%. A larger dataset would also help each classifier to learn better as long as the original proportions are maintained.

5.4 Ensemble Learning

Once I trained and fine-tuned the five individual classifiers, I set up a pipeline to pass test data to that would generate a confidence score for every potential vulnerability. Each instance would be passed into the final model, which would then pass the instance through each of the five classifiers. Each classifier would generate a confidence tuple containing P(vulnerable) and

P(benign). Whichever one of these probabilities was greater was taken to be the predicted label for each classifier. All five of the predictions were returned to the ensemble classifier. The ensemble classifier then labeled the instance according to the highest confidence score generated by any of the five models. This label was compared to the ground truth label for each instance in order to generate overall accuracy of the system.

6 Results

The results of the overall system can be understood as the accuracy of the vulnerability classifier. Though the functionality of the ensemble classifier is heavily dependent on the accuracies of the binary vulnerability classifiers, the system itself is benchmarked by the accuracy of completely new instances.

It is challenging to compare my accuracy to those who have done similar work because I did not use the same dataset as any previous work used. Inconsistencies across datasets could cause changes in accuracy of up to 10-15%, so it is ultimately unreasonable to compare the results of my work to the results of previous works. In order to demonstrate the usefulness of my model, then, I compare my work to the expected value of a naïve classifier. My test set contained the following data. None of this data was passed to any of the classifiers during training.

Vulnerability	Snippets	Percentage (%)
CWE-119	2419	1.9
CWE-120	4750	3.7
CWE-469	252	2.5
CWE-476	1208	0.1
CWE-other	3579	2.8
Benign	115268	89.0
Total	127476	100

Table 4: Test Set

A naïve classifier would likely overfit the data due to the underrepresentation of the vulnerabilities, resulting in a system that treated every instance as benign. The accuracy of such a system would be 89% with 11% false negatives. Due to the nature of the problem, however, false negatives are far worse than false positives. Failing to detect a vulnerability when it is present is worse than detecting a vulnerability when one is not present. The classifier I built also tracked false positive (FP) as well as false negative (FN) for each vulnerability. The results are show in table 5 below.

Classifier	FP (%)	FN (%)	Overall Accuracy
DeepNLPEnsemble	3.4	2.9	93.7
Naïve	0.0	11.0	89.0

Table 5: Results

My classifier considered false negative to be a vulnerable sample of code classified as benign when the ground truth was vulnerable. Table 5 demonstrates that the overall accuracy of my classifier is 4.7% better than a naïve NLP classifier. More importantly, the false negative percentage generated by my model was 8.1% lower than a naïve classifier. This is likely a result of passing each test instance through every classifier. By doing so, the probability that a potentially vulnerable snippet of code is not detected decreases substantially.

7 Limitations

There are two primary limitations of my system that I defer to future work. The first limitation is on scalability. The feature that separates my model from previous deep learning approaches is the use of five separate classifiers for each of the five vulnerabilities. There are obviously far more than five vulnerabilities that can be used to exploit systems by clever

attackers. My system would require the training of a separate classifier on thousands of instances for every vulnerability. This does not scale well to support all vulnerabilities.

The second limitation is the size of the dataset. Training a deep neural network is computationally intensive and requires massive amounts of compute power. To train the language model for 5 epochs on an AWS Elastic Compute Cloud with 72 virtual CPUs still took over two hours for my dataset which only contained 150,000 elements. Scaling this system to larger datasets would be extremely challenging both computationally and financially.

8 References

[1] Tung, L. (2019, April 8). Programming language popularity: C bounces back at Python's expense. Retrieved from https://www.zdnet.com/article/programming-language-popularity-c-bounces-back-at-pythons-expense/

[2] Bloomfield, A. (2010, February 10). DADA: “Tricky Jump” technique. Retrieved from https://aaronbloomfield.github.io/dada/docs/tricky-jump.html

[3] Brownlee, J. (2019, December 19). Deep learning: An Introduction Retrieved from https://machinelearningmastery.com/what-is-deep-learning/

[4] Mitre. (2020, April 25). Common Weakness Enumeration. Retrieved from https://cwe.mitre.org/

[5] Harer, A., J., Kim, Y., L., Russell, L., R., … Tomo. (2018, August 2). Automated software vulnerability detection with machine learning. Retrieved from https://arxiv.org/abs/1803.04497

[6] XU, Z., Kremenek, T, Zhang, J. A memory model for static analysis of c programs. In International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (2010), Springer, pp. 535–548.

[7] C. King, “Symbolic execution and program testing,” Commun. ACM, vol. 19, pp. 385–394, July 1976.

[8] Li, Zhen, Zou, Deqing, Xu, Shouhuai, … Yuyi. (2018, January 5). VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. Retrieved from https://arxiv.org/abs/1801.01681