Identifying Vulnerable C Code using Machine Learning Techniques

Sid Phatak

June 2024

Abstract

This report presents the development and evaluation of a machine learning model for identifying vulnerable C code. Using an AI-generated dataset of both vulnerable and non-vulnerable C code snippets, we explore various methodologies including Bag of Words (BOW), Logistic Regression, word embeddings, and Recurrent Neural Networks (RNNs) to build an effective classification model.

1 Introduction

1.1 Background

Identifying vulnerabilities in C code is very necessary for several reasons:
Security Risks: Vulnerabilities can be exploited to gain unauthorized access, disrupt system operations, or even execute arbitrary code which can lead to data breaches, system crashes, or other negative results.
Software Quality: Vulnerabilities often show deeper issues with code quality. Addressing these can lead to more reliable and secure software.
Cost Efficiency: Detecting and fixing vulnerabilities in the early development stages is usually much cheaper and less time-consuming to deal with than if there was a massive security incident much later on when there is alot more things to deal with.
Artificial intelligence and machine learning are showing very positive results in automating vulnerability detection. Traditional methods, such as manually reviewing code line by line or static analysis tools can be extremely time consuming and may not catch every vulnerability. AI and ML models can be trained to learn from previous vulnerabilities and could identify complex patterns and vulnerabilities that most traditional methods might miss. They can also analyze large codebases quickly, displaying its scalability in vulnerability detection.

1.2 Objective

The main objective of this project is to develop as well as evaluate ML (machine learning) models that are able to distinguish between nonvulnerable and

vulnerable C code. This involves training the models using the FormAI Dataset of C code snippets labeled as vulnerable or nonvulnerable. The way the models perform are evaluated on its ability to correctly classify the code snippets. The goal is to create a tool that assists in software development by automatically identifying potential vulnerabilities.

1.3 Organization

1. Dataset Generation
   This section describes how the dataset is created, such as the collection and preprocessing of data.
2. Methodology
   This section outlines the different approaches, models, and reasoning used in the project, such as the BOW Model, Logistic regression, word embeddings, and RNNs.
3. Model Training and Evaluation
   This section covers the training process and the evaluation metrics of the models.
4. Discussion
   This section analyzes the results and discusses the challenges faced as well as potential work in the future.
5. Conclusion
   Summarizes the key findings of the project, as well as the importance of ML for C code. Reflects on the project and its potential implications for software security.
6. References
   References used for the project.

2 Dataset Generation

2.1 Data Collection

The FormAI dataset consists of 3 main files -
The main 112,000 C sample files, a CSV detailing vulnerabilities and corresponding C code across 246,550 rows, and a streamlined human-readable CSV version excluding the C code. Each C file is programmatically generated and classified with information including file names, vulnerability types, and specific lines of code where these vulnerabilities occur.
The labeling of code snippets as either vulnerable or nonvulnerable come from the "Vulnerability Type" column within the dataset file. It is processed in two different ways -
If the preprocessed vectorized data is already present, the system loads this data along with the original set, with the "Vulnerability Type" column being extracted. If preprocessed data does not exist, then the original dataset is loaded, and a filtered version is created with only the columns "Vulnerability

Type" and "Source Code" columns present. These columns are transformed into a matrix of word embeddings. It is then utilized in model training and the evaluation phase, where it is then binarized, with "vulnerable" being set to true and everything else set to false.

2.2 Data Preprocessing

The dataset is first loaded through a CSV file, resulting in a dataframe in which each row corresponds to a code snippet, and each column corresponding to a feature of the snippet. the dataframe then gets filtered to only include the "Vulnerability Type" (labeled y) and "Source Code" columns (labeled x). The inputted data (x) is converted to a matrix of word embeddings, which is done by tokenizing each code snippet and converting each token to a vector representation, resulting in a list of lists, where each inner list corresponds to a code snippet and contains the vector representations of the tokens in the code snippet.

3 Methodology

3.1 Bag of Words (BOW) Model

The Bag of Words Model is used in natural language processing and for text classification. It represents a set of words, like a document or sentence, as a bag, keeping track of the frequency of each word. In the case of our project, each C code snippet was treated as a document. The BOW model was used to convert each of the code snippets into vectors that could be used as input for the models. The BOW model is incredibly simple to implement and is very effective at the task, allowing models to see the frequency of individual words used within the document,

3.2 Logistic Regression

Logistic Regression was used to help classify the accuracy of the model in determining a snippet to be vulnerable or nonvulnerable. With logistic regression, it is relatively simple in application and it is fast to train.

3.3 Word Embeddings

Word embeddings are word representations that lets words with similar meanings to have a similar representation. In this project, the word embeddings are generated from the spaCy and are used to represent the code snippets in a continuous vector space, where the position of each word is learned based on its context. The en core web md pre-trained model from spaCy is used in this project, to help convert each token in each snippet into vectors which are then used as input to the ML models.

3.4 Recurrent Neural Networks (RNNs)

Recurrent neural networks are a type of neural network that are made to identify patterns in sequences of data, making them extremely suitable for analyzing code, as it can be represented as a sequence of tokens. The neural network used was a sequential model implemented through Keras. It was configured with a variable number of layers, and a variable number of neurons, with the first layer having a different number of neurons, and needing the input dimension to be specified. Each layer uses the ReLU activation function, except for the final layer which uses the Sigmoid activation function. Several models are created, trained, and evaluated. The configurations differ in the number of layers, the number of neurons in the first layer, and the number of neurons in the other layers.

4 Model Training and Evaluation

4.1 Training Process

The training process for each model is carried out through one function with the purpose being to test the neural network with the optimization being handled by the Adam optimizer. The function iterates over many different configurations of hyperparameters, such as the number of layers, the number of neurons in the layers. For each configuration, a new model is made that is built using another function, where it is then trained using the fit method (X Train and Y Train). the predictions are evaluated using a custom accuracy function, where which the prediction accuracy is then displayed.

4.2 Evaluation Metrics

A neural network model was used, and the performance was evaluated using accuracy as the metric. The models would be built based upon three factors, those being the number of layers (1, 3, 5, or 10 layers), the number of hidden neurons (8, 32, 124, or 512 hidden), and the number of neurons in the first layer (512 or 1024). Every possible configuration was tested from these numbers, with the accuracy being documented.

4.3 Results

The results from the model are sorted from the highest accuracy to the lowest accuracy.

Layers	First Neurons	Hidden Neurons	Accuracy
1	1024	124	0.9419590346785642
3	512	512	0.9414723179882377
1	1024	512	0.9405394443317785
3	1024	32	0.9406408436422632
1	1024	32	0.9408030825390388
3	512	32	0.9403974852971
3	1024	512	0.940417765159197
1	512	124	0.9404583248833908
3	1024	124	0.9400527276414521
1	512	32	0.9398904887446765
5	512	512	0.9398904887446765
10	512	32	0.9395457310890286
3	512	124	0.9396268505374163
1	512	512	0.9393023727438653
5	512	124	0.93940377205435
10	1024	124	0.938775096329345
5	1024	512	0.9388764956398297
5	512	32	0.9382275400527277
5	1024	32	0.938105860880146
5	512	8	0.9380044615696613
10	1024	8	0.9373352261204624
10	512	8	0.9370918677752992
1	1024	8	0.9379030622591766
10	1024	32	0.9374771851551409
1	512	8	0.933887649563983
5	1024	8	0.9339687690123707
10	512	124	0.9353477996349625
3	1024	8	0.9354289190833502
5	1024	124	0.9366254309470695
10	1024	512	0.9368687892922328
3	512	8	0.9320827418373555

5 Discussion

5.1 Analysis of Results

Many approaches were used during this project, such as logistic regression, but ultimately, neural networks were used as the main approach. While neural networks can handle and model complex datasets with high inputs, it requires a lot of data to train effectively. It is also computationally expensive, which caused many problems down the line while gathering data. Important findings to note include that configurations using 1024 first neurons generally perform

much better than the other configurations, that 124 or 512 hidden neurons correlate with higher accuracy, that 1 and 3 layers perform better than 5 and 10 layers, and finally, that configurations with fewer hidden neurons tended to have lower accuracy.

5.2 Challenges

Many challenges were encountered throughout this project, with the most prevalent being computer processing power and too little memory. In some instances, when testing different approaches or other methods, the computer would take an unreasonable amount of time or not work entirely as the memory required was extremely high.

5.3 Future Work

Potential improvements and future research directions include:

- **Exploring other Machine Learning Techniques**
  - While neural networks are powerful, there are still many other algorithms I could potentially use to get a better result for my data. For instance, decision trees, random forests, support vector machines, and gradient-boosting algorithms could be used. They each have their own sets of strengths and weaknesses and might perform better.
- **Expanding the Dataset**
  - The quality and quantity of data used to train the model has significant impact on the performance. By using a larger dataset or artificially increasing the dataset size through data augmentation techniques, it could be used to further train the model which would result in outputting even more accurate data.

6 Conclusion

The use of neural networks with varying layers and neurons demonstrated a range of accuracies, with the highest reaching about 94.2%. This indicates that machine learning models can effectively learn patterns associated with vulnerabilities in C code and predict them accurately. As software become more complex, manually identifying vulnerabilities could take an incredibly long time. Machine learning offers an entirely new, automated and scalable solution that is able to keep up with this more complex software. By identifying vulnerabilities in C code, we can address potential security threats before they are exploited, strengthening the overall security of software systems. The implications of this project extend past just software security. It shows how machine learning can be used to automate complex tasks in many other areas of software security, demonstrating the incredible potential to increase efficiency.

7 References

[1] N. Tihanyi et al., "The formai dataset: Generative AI in software security through the lens of formal verification," arXiv.org, https://arxiv.org/abs/2307.02192 (accessed Jun. 12, 2024).