Low Level Source Code Vulnerability Detection Using Advanced BERT Language Model

Mansour Alqarni†, Akramul Azim†
† Ontario Tech University

Abstract
In software security and reliability, automated vulnerability detection is an essential and compulsory task. Software needs to be tested and checked before it goes to the client for production. As technology changes rapidly, source code is also becoming massive. Thus the adequate accuracy of automated vulnerability detection has become very important to produce secure software and remove security concerns. According to previous research, a deep and recurrent neural network model can not satisfactorily test accuracy to detect all vulnerabilities. In this paper, we introduce experimental research on Bidirectional Encoder Representations Transformers (BERT), a state-of-the-art natural language processing model aimed to improve test accuracy, contributing to updates to the development of deep layers of the BERT model. As well, we balance and fine-tune the dataset of the model with improved parameters. This combination of changes achieves new levels of accuracy for the BERT model, with 99.30% test accuracy in detecting source code vulnerabilities. We have made our balanced dataset and advanced model publicly available for any research purposes.

Keywords: Machine Learning, Neural Network, Advance Deep Learning, Program Analysis, Software Security, Balanced Dataset.

1. Introduction
Recently, production using Continuous Integration (CI) and Continuous Delivery (CD) pipelines have been prioritized in the software development industry. This pipeline also includes automated tests. Static analyzer software like SonarQube [1] and Brakemen [2] are used to detect software vulnerabilities. Still, these automated processes cannot accurately detect vulnerabilities due to a preponderance of false-positive and false-negative results. As security vulnerability detection is a sensitive task, the software’s accuracy needs to be paramount; otherwise, they can be hacked, cracked, or stopped due to security bugs, breaches, unhandled exceptions, memory leaks, buffer overflow, etc. Previously, researchers used traditional algorithms to detect vulnerabilities [3], but with the increased availability of computing resources, machine learning [4] and deep learning [5] have been used to improve the accuracy of vulnerability detection. Even so, accuracy has not reached the level at which it needs to be. Recently, advanced models of Natural Language Processing (NLP) have been introduced, which are called transformers [6] [7] [8], able to extract all important features that were not possible previously with common machine learning algorithms. Transformers use a sequence architecture generated from layers to offer state-of-the-art accuracy.

Though these transformer models are being used for human language taxonomy classification, translation, answering questions, grammar correction etc. In this paper, we have used the transformers for vulnerability detection within programming languages. We have proposed a bi-modal model that can understand the difference between natural language and programming language. Low level programming languages in the past have multi vulnerabilities causing Cyber Hijacking, Buffer Overflow, Firewall system, Memory corruption attacks and more [9] which can not be detected during the coding process. To improve the security and integrity of the application and embedded software we propose our advanced BERT model with a higher accuracy for detection weakness during the coding process. Our main contributions are listed below:

*Mansour.Alqarni@ontariotechu.net

This article is © 2022 by author(s) as listed above. The article is licensed under a Creative Commons Attribution (CC BY 4.0) International license (https://creativecommons.org/licenses/by/4.0/legalcode), except where otherwise indicated with respect to particular material included in the article. The article should be attributed to the author(s) identified above.
We have balanced the existing dataset using sampling algorithms as previous results from existing research papers were biased due to data imbalances.

We have proposed to use pre-trained Bidirectional Encoder Representations Transformers (BERT), and have updated the BERT model by adding additional scope for vulnerability detection. Also developed an architecture for deep layers so that our updated model can outperform machine learning and deep learning models from previous researchers’ work.

Finally, we have fine-tuned the model by finding the most effective parameters for state-of-the-art accuracy.

The remainder of the paper is organized in the following way. In section II, previous related works of researchers are highlighted. In section III, the dataset of vulnerability detection is described, as well as the techniques we use to remove data imbalance problems. In section IV, we discuss methods, transformer model architectures, and model parameters. Experimental results and performance comparison are offered in section V, followed by a conclusion in section VI.

2. Related Works

Many researchers have contributed to source code vulnerability detection as the number of security issues has increased rapidly in the software industry. In the early stages of this effort, static analyzer techniques were used to detect vulnerabilities [10] and tools like Clang were developed [11]. But after the evolution of highly configured cloud computing resources, machine learning and deep learning models have been used to detect vulnerabilities by researchers in recent years. Using machine learning algorithms smaller to naive-bayes and support vector machine (SVM) [12] and tokenization to use bags of words [13] have been used. Tools like CppCheck [14] and Flawfinder [15] have been created. But these algorithms have failed to provide high levels of accuracy. After the deep learning evolution, some researchers have applied convolutional neural network (CNN), recurrent network (RNN), and long short term memory (LSTM) options to detect different kinds of vulnerabilities [16] [17]. These deep learning models can provide an advanced performance than common machine learning algorithms and the accuracy of experimental results are also better than previous traditional machine learning algorithms. But the memorization techniques of deep learning models have some drawbacks as important features and information can be missed. To solve these problems, Google introduced BERT [18] as a natural language processing transformer model to increase positive results on natural language processing tasks. But this model was mainly designed for human language related tasks like sentiment analysis [19], question-answering [20], and named entity recognition. Using similar techniques that has been used by other researchers [21] on the other hand we used an advanced ML model to attain better results. Our research aims to make the BERT model compatible with programming languages and achieve new levels of accuracy.

3. Dataset

Datasets are the main raw materials for natural language processing models. Experimental results are not only dependent on best models but also dependent on preprocessed and unbiased datasets. We have selected the latest and updated code repository Github [22] where developers push source code from different parts of the world. The Github dataset is comparatively the most unbiased, real-world dataset as it is a worldwide code repository expanded and updated every day. It is assumed that using this dataset, experimental results will be as reliable as possible and detect new vulnerabilities. This dataset has 119 different kinds of Common Weakness Enumeration (CWE) for vulnerability detection. Table 1, below, shows the Github dataset overview and Figure 1 shows primary dataset visualization.

<table>
<thead>
<tr>
<th>Table 1. Overview of Github Dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Code</td>
</tr>
<tr>
<td>Vulnerable Source Code</td>
</tr>
<tr>
<td>Non Vulnerable Source Code</td>
</tr>
</tbody>
</table>
To understand more about the dataset, we compared a vulnerable source code and non-vulnerable source code from this dataset. The source code reveals a vulnerability for `strlen()` built in function used in `memcpy()` function, revealed in Figure 2. In Figure 3 the correct function `sizeof()` is used to remove the vulnerability.

![Figure 2. vulnerable source code](image1)

```
void vulnerable(char* S)
{
    char text[256];
    memcpy(text, s, strlen(s));
    printf("%s\n", text);
}
```

![Figure 3. non-vulnerable source code](image2)

```
void nonvulnerable(char* S)
{
    char text[256];
    memcpy(text, s, sizeof(s));
    printf("%s\n", text);
}
```

The dataset has two columns. The first column is the source code of different functions and the second column is the vulnerability status (CWE). Figure 4, below, shows the dataset contents. Table 2 shows the statics of vulnerability types which are present in this dataset.

![Figure 1. Visualization of Github Dataset](image3)

```
<table>
<thead>
<tr>
<th>Description</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer Overflow Problems</td>
<td>38.2%</td>
</tr>
<tr>
<td>Memory Leaks</td>
<td>18.9%</td>
</tr>
<tr>
<td>NULL Pointers</td>
<td>9.5%</td>
</tr>
<tr>
<td>Pointer Subtraction Problems</td>
<td>2.0%</td>
</tr>
<tr>
<td>Incorrect Length of Buffer Access</td>
<td>31.4%</td>
</tr>
</tbody>
</table>
```

**Improvement of Dataset:** After observing the dataset, we found that it was not balanced. As an imbalanced dataset can give wrong experimental results, we used over-sampling algorithms via the Synthetic Minority Oversampling Technique (SMOTE) to
balance the existing dataset, randomly duplicating minority class samples. The SMOTE technique selects close samples from the feature space, and this technique will calculate the distances between two corresponding samples. After selecting the close samples, more related samples are duplicated to balance the dataset.

Table 3 shows the updated dataset overview and Figure 5 shows the balanced dataset visualization.

Table 3. Overview of balanced Github dataset

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Code</td>
<td>247858</td>
</tr>
<tr>
<td>Vulnerable Source Code</td>
<td>123929</td>
</tr>
<tr>
<td>Non Vulnerable Source Code</td>
<td>123929</td>
</tr>
</tbody>
</table>

4. Proposed Methods

BERT is a language model that is based on transformers. The speciality of BERT is that it is fully bidirectional. Traditional models scan texts from left to right or right to left. But BERT can scan the text from both sides at the same time [18]. For example, previous models might assume "bank deposit" and "riverbank" as linked to the word "bank." But due to the bidirectional property of BERT, it is now possible to derive deeper meanings from cognates such as these. BERT is a pre-trained model that has been previously trained from unlabelled Wikipedia (2500M Words) and Book corpus
We can use transfer learning to train the BERT model with our vulnerability detection dataset for the programming language. To do this, we have used some pre-processing techniques to convert the source code into specific formats and then added deep layers to develop the deep transition architecture of the BERT training model. The pre-processing and feature extraction techniques are described below.

**Tokenization:** Computers do not understand any language. It mainly understands the numbers. To convert the language into numbers, the first step is tokenization. Tokenization mainly splits the sentences into meaningful words. So we have proposed to tokenize the source code according to BERT specific format. As BERT is bidirectional, it marks the beginning by [CLS] token and ending by [SEP] token. Then it starts tokenization. As an example, a simple source code is shown below that can be tokenized.

### Normal Code:

```plaintext
char text [256]; gets(text);
```

### Tokenized code:

```plaintext
[ 'char', 'text', '####', '###25', '###6', '###', '###;', 'gets', '### (', '###text', '###)', '###;']
```

**Padding:** Padding must be completed before training any dataset to a specific model. BERT only allows fixed length sentences as input. If the length of source code from the dataset is shorter than the fixed-length, some paddings will be added to fit in the fixed length. BERT provides an artificial token [PAD] to add necessary padding.

**Encoding:** The final step of pre-processing is token encoding. In this step, all the tokens will be converted into numbers. Every token will be converted into its corresponding unique id.

Here is the final encoding output of the tokenization example:

```plaintext
[ 25869, 3793, 29634, 17788, 2575, 29636, 29628, 4152, 29619, 18209, 29620, 29628]
```

### 4.1. Architecture of the BERT model and our additional model:

BERT is not trained on a vulnerability detection using the best appropriate dataset, we have proposed to update the model by adding an additional model with some layers and train the whole model with vulnerability detection GitHub dataset. At first, we discussed the main BERT architecture and after that we presented our additional models with updated layers and features.

#### Existing BERT Architecture:
BERT has mainly two different models. These are:

- BERT Base
- BERT Large

The BERT Base has 12 transformer blocks and BERT Large has 24 transformer blocks. These transformer blocks are called encoders. Under the hood, these encoders have large feed-forward networks (768 and 1024), respectively. Figure 6 shows the BERT model architecture.

In BERT, the input sentences are the sum of the token, segment, and position embeddings. After separating the sentences with a special token, learned embeddings indicated by E are added to check the belongings of the sentences. Figure 7 shows the input embeddings of the BERT model.

BERT uses transformer which is an advanced attention mechanism used to learn contextual comparison between words or sub-words in a text. Transformer has two different mechanisms in its basic form: an encoder that reads the text input and a decoder that generates a job prediction. However, encoder technique is required because BERT’s purpose is to construct a model language.

The Transformer encoder is described in detail in Figure 6. The input is a sequence of tokens that are embedded into multi-vectors at the beginning of the neural network processor. The result is a sequence of H-dimensional into multi-vectors, each of the vectors has a corresponding index linked to that input.

To train a BERT model, randomly, some percentage of the tokens are masked. This is called masked LM. Approximately about 15% there is a sequence in each word that is substituted with a [MASK token] before it is being processed into BERT. Based on
Figure 6. BERT model architecture [23]

Figure 7. All Embeddings of BERT

Figure 8. Masked LM of BERT Model [25]

the context that it is been provided by different studies, where most the sequence of words are non-masked and during each process the model will attempts to predict the original value of the masked words. The output of the encoder has a classification layer will be added to Transform the outcome vectors into the vocabulary dimension by using multiplying them into embedding matrix. Soft-max also been used for calculating the probability of each word [18]. Figure 8 shows the masked LM of BERT.

4.2. Proposed advanced model architecture:

We have proposed adding some additional layers combined with a pre-trained BERT model. In the final updated model, there are three input layers. These are: input, word, ds, input, mask, and segment, ds.
In the next position, there is a pre-trained BERT layer. Then we have added three dense layers. The first dense layer has 64 filters with a dropout technique to reduce the overfitting problem. The middle dense layer has 32 filters with dropouts. The final dense layer has 2 filters for two classes with a dropout technique. We have used the Rectified linear unit (ReLU) as an activation function for the first two layers. But for final classification, the softmax function has been used. Equation 4.1 shows the probability distribution of the class labels. Here P is the probability, C is the context vector that is the final hidden state corresponding to the first token [CLS]. W is the weight and T is the numerical tokens.

\[ P = \text{softmax}(CW^T) \] (4.1)

We have used the categorical cross-entropy formula to calculate the loss of our proposed model. Figure 7 shows our proposed updated model architecture for vulnerability detection. After adding our proposed additional layers, we have successfully trained the source code vulnerability dataset and improved accuracy. Our updated model can be expressed by a summary of architecture also. Figure 9 shows the model summary.

Finally, we have fine-tuned our updated model by finding optimal parameters, revealed in Table 4 below.

Table 4. The parameters for training our advanced model

<table>
<thead>
<tr>
<th>Parameters Name</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Learning Rate</td>
<td>2e-5</td>
</tr>
<tr>
<td>Batch Size</td>
<td>64</td>
</tr>
<tr>
<td>Epoch</td>
<td>10</td>
</tr>
</tbody>
</table>
5. Experimental Setup and Result Analysis

Experimental environment: Training a BERT-based model is a very resource-consuming task, and it takes much time. The experimental environment has an Intel Core i9 processor, a Tesla k80 GPU, and 12 GB of RAM. As a result of this experimental environment, training time has been reduced while performance has improved.

Training, validation, and testing experiment: We have split the whole dataset to three types of experiment. These are: training, validation, and testing. Training dataset is for training the model; validation dataset mainly validates the accuracy before final testing, and testing dataset is used for testing the model accuracy. Test dataset is 0.2% of the whole dataset and the validation dataset is 0.2% of the training dataset. Thus we have updated the BERT model and fine tuned the model for better performance. Table 5 shows the distribution of training, validation, and testing dataset numbers.

Experimental Result Analysis: After 4 hours of training our proposed model, we achieved the new state-of-the-art accuracy for source code vulnerability detection. Table 6 shows the final result of our experiment for training, validation, and testing.

Figure 11, below, shows the training versus validation accuracy curve and Figure 10 shows the training versus validation loss curve.
After getting the result, we compared our result with previous researcher results. It is important to match the dataset also while comparing the result with previous research experiments. We have found two research papers for comparison. Table 7 shows the comparison of test results with previous experiments.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>81.4%</td>
<td>94.3%</td>
<td>99.30%</td>
</tr>
</tbody>
</table>

From table 7, we can easily understand that a pre-trained BERT based customized NLP model can outperform previous deep learning models like RNN, LSTM, and Bi-LSTM. As BERT is already pre-trained using millions of texts and can understand the meaning of different kinds of sentences, thus BERT has achieved the state of the art accuracy than other models.

6. Conclusion

In our research, we balanced the Github dataset and successfully developed a new BERT-based source code vulnerability detection model. BERT was not designed for programming language related tasks rather it was designed for human language related tasks previously but we have added an additional model with BERT and train the updated model again for vulnerability detection. As a result, we have achieved the new state of the art result which is 99.30% for some vulnerability detection. But due to some limitation of powerful computing resources, we have not added all vulnerabilities. In future, we are working on more vulnerabilities and programming languages also. Our updated model can be also used by other researchers and engineers for developing vulnerability detection software or further improvement.

References


