10-27-2021

# Auto-Detection of Programming Code Vulnerabilities With Natural Language Processing

Yubai Zhang
*Illinois State University*, superalexzhang123@outlook.com

# AUTO-DETECTION OF PROGRAMMING CODE VULNERABILITIES WITH NATURAL LANGUAGE PROCESSING

YUBAI ZHANG

27 Pages

Security vulnerabilities in source code are traditionally detected manually by software developers because there are no effective auto-detection tools. Current vulnerability detection tools require great human effort, and the results have flaws in many ways. However, deep learning models could be a solution to this problem for the following reasons: 1. Deep learning models are relatively accurate for text classification and text summarization for source code. 2. After being deployed on the cloud servers, the efficiency of deep learning based auto-detection could be much higher than human effort. Therefore, we developed two Natural Language Processing(NLP) models: the first one is a text-classification model that takes source code as input and outputs the classification of the security vulnerability of the input. The second one is a text-to-text model that takes source code as input and outputs a completely machine-generated summary about the security vulnerability of the input. Our evaluation shows that both models get impressive results.

KEYWORDS: Natural Language Processing; Deep Learning; Security Vulnerabilities; Auto-Detection, Source Code Testing

# AUTO-DETECTION OF PROGRAMMING CODE VULNERABILITIES

# WITH NATURAL LANGUAGE PROCESSING

YUBAI ZHANG

A Thesis Submitted in Partial
Fulfillment of the Requirements
for the Degree of

MASTER OF SCIENCE

School of Information Technology

ILLINOIS STATE UNIVERSITY

2021

# AUTO-DETECTION OF PROGRAMMING CODE VULNERABILITIES WITH NATURAL LANGUAGE PROCESSING

YUBAI ZHANG

COMMITTEE MEMBERS:

Shaoen Wu, Chair

Sumesh Philip

Yongning Tang

# ACKNOWLDGMENTS

I would like to express my deepest appreciation to Dr.Shaoen Wu and Noah Ziems who have been giving me a great amount of guidance and help. I also wish to thank my committee members for their time and patience. I am grateful for the School of Information Technology of Illinois State University for providing me with the computational resources for my research.

Y.Z.

## CONTENTS

# TABLES

Table Page

# FIGURES

# CHAPTER I: INTRODUCTION

In practice, manual detection of security vulnerabilities is costly in terms of time and human resources. However, this cost could be greatly reduced if certain vulnerabilities are detected before deployment instead of after. Auto-detection of software vulnerabilities can save the time of fixing the problems after deployment. Over the years, machine learning models in Natural Language Processing(NLP) have significantly improved because more techniques, such as pre-training, modifications in the dataset, and fine-tuning(Howard and Ruder, 2018) have been broadly used in NLP. Moreover, recent work in the deep learning field has shown tremendous promise of enabling auto-detection of security vulnerabilities. However, most of those works are limited in their own ways, such as the structure of the source code files or the capability of their models. This work presents two very different models for solving the same problem: the auto-detection of security vulnerabilities. The first is a text-classification model, which shall be addressed as the Classifier for the remainder of the paper. The Classifier takes source code files as input, then picks out the right security vulnerability class for the input and outputs the description of the class. The second model is a text-to-text model, which shall be addressed as the Summarizer for the remainder of the paper. The Summarizer also takes source code files as input, but it outputs an entirely machine-generated summary of the input. Furthermore, the Classifier is a more traditional RNN model, and the Summarizer uses a more state-of-the-art Transformer structure. The reason for presenting two very different NLP models is that this research aim for proving auto-detection for security vulnerabilities in source code files is achievable. Both RNN and Transformers are excellent for this task, and there should be no specific rules regarding the source code files because both models use the same dataset, and both models get excellent results. Furthermore, this research hopes to prove that using NLP models for auto-detection of security vulnerabilities is the future direction of source code security testing.

## CHAPTER II: RELATED WORK

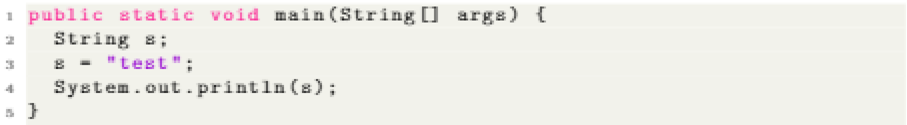
### Security Vulnerability Detection

There are a variety of traditional ways of attempting to detect security vulnerabilities in source code. Nevertheless, most of them fall into two categories: Static Code Analysis and Dynamic Code Analysis. Furthermore, researchers have been trying to achieve auto-detection of security vulnerabilities for years and have made decent results within the field of Natural Language Processing and other approaches.

### Static Code Analysis

Static Code Analysis(Gomes et al., 2009) is a traditional technique that examines the program's codes with hard-coded rules developed by programmers(Louridas, 2006). According to Xueqi et al. (Yang et al., 2020), Static Code Analysis looks for duplicated or unused code that breaks the hard-coded rules, and this approach does not run the program itself. For example: as the code is shown in figure 1, the program declares a String variable s, then changes the value of s into ”test,” and outputs the value of s. Technically, this program does not have any problem because output s will have the value of ”test” instead of null by the time of output. But, it is commonly considered a lousy code to declare a String without initializing its value. So, Static Code Analysis will detect this problem and put that in the report. However, there are problems with Static Code Analysis; First of all, it has so many false positives because the rules are hard-coded, and sometimes the rules do not fit the situation.

In other circumstances, Static Code Analysis can not detect actual problems because those problems are not in the hard-coded rules. Another flaw of Static Code Analysis is this approach takes a tremendous amount of time and human effort, which is the very problem this work is focused on. There are multiple static code analysis tools(Mantere et al., 2009)(Zhioua et al., 2014) that are widely used in software development, such as FindBugs or PMD(Mahmood and Mahmoud, 2018). However, they all have the same problem: they have many false positives and false negatives (Kaur and Nayyar, 2020). Furthermore, static code analysis tools do not perform well for certain common security weaknesses(Goseva-Popstojanova and Perhinschi, 2015).



```
public static void main(String[] args) {
  String s;
  s = "test";
  System.out.println(s);
}
```

*Figure 1: Dead Loop*

### Dynamic Code Analysis

Dynamic Code Analysis finds issues in the code by running the program with extreme parameters or malicious inputs(Bayer et al., 2006) and looking for errors or discrepancies in the outputs. Typically, this is in the form of unit tests(Kim et al., 2007). This approach still requires a tremendous amount of time because dynamic code analysis needs programmers to write test programs such as unit tests and integration tests, which are called test cases(Kim et al., 1999). Although the test case generating process has been automatic for years(Meudec, 1998)(Shanthi and Kumar, 2011), it still requires time and human effort to verify the integrity of the test cases.

Furthermore, dynamic code analysis can only test issues that the programmer has anticipated, which is not ideal for the detection of security vulnerabilities since most problems can cause financial problems or personal information loss(McGraw, 2004)(Ghosh and Swaminatha, 2001).

**Natural Language Processing(NLP)**

It is common knowledge that NLP models that use pre-training and fine-tuning have shown excellent results on a wide variety of tasks.(Howard and Ruder, 2018) In NLP, the pre-training phase is akin to training an ImageNet (Russakovsky et al., 2015) whereas the fine-tuning phase is similar to training the model on a new task or dataset. There has been previous work done in NLP models to extract useful features from programming languages. Karpathy et al. (Karpathy et al., 2015) have trained a Recurrent Neural Network (RNN) to predict the next character in each sequence of C code, then analyze the remote activation. Lukasz et al.(Kaiser et al., 2017) have used more traditional NLP models to detect security vulnerabilities in software code. Baptiste et al. (Roziere et al., 2021) have done great work about deobfuscation in the pre-training models for programming codes.Furthermore, Lowik et al. (Lachaux et al., 2020) have done impressive research about building a transpiler for translating functions/methods between different programming languages such as C++, Java, and Python with high accuracy. Harer et al.(Harer et al., 2018) have trained source code-based model and feature-based model to achieve auto-detection of programming source files. Zhuang et al.(Zhuang et al., 2020) built a graph neural network(GNN) model for auto-detection of security vulnerabilities. They generate a graph based on the functions' control and data flow, then normalize the graph before using it as input.

## Other Approaches for Code Vulnerabilities Auto Detection

Other approaches besides Natural Language Processing have made progress toward auto-detection of security vulnerabilities. Bau et al.(Bau et al., 2010) use Black box Scanners to detect weaknesses in web applications. Wagner et al.(Wagner et al., 2000) made a prototype that detects buffer overrun vulnerabilities in C code. Lin et al.(Lin et al., 2017) built a deep learning model that uses ”CodeSensor(Yamaguchi et al., 2012)” to detect function’s structure, which is called AST, and convert them to vectors, and then use vectors as inputs to achieve auto-detection of vulnerabilities in functions. Meng et al. (Meng et al., 2016) have done excellent research on auto-detection of security vulnerabilities through similar functions. They use a vulnerable function as a template and divide source code into substructures, and detect similar structures in the template.

## CHAPTER III: DATASET

### Dataset Source

This work uses the Common Weaknesses Enumeration(CWE)(Martin, 2007) dataset because the CWE dataset represents common security vulnerabilities in software which is perfect for this work. Both datasets for the two models that this work presents come from the same CWE dataset as the original dataset. The only difference is how we process them in two different ways. Here is the dataset URL:

https://samate.nist.gov/SRD/testsuites/juliet/Juliet_Test_Suite_v1. 3_for_Java.zip

### Source Code Dataset for the Pre-training Model of Classifier

This dataset is used in the pre-training model for the Classifier. The pre-training model is a language model that could predict the next word of input text. Unlike the dataset used for training the Classifier, the files in this dataset are collected from GitHub since all the codes on GitHub are open-source and free. Below is a table of the URLs for the source code files.

*Table 1: Github URL List*

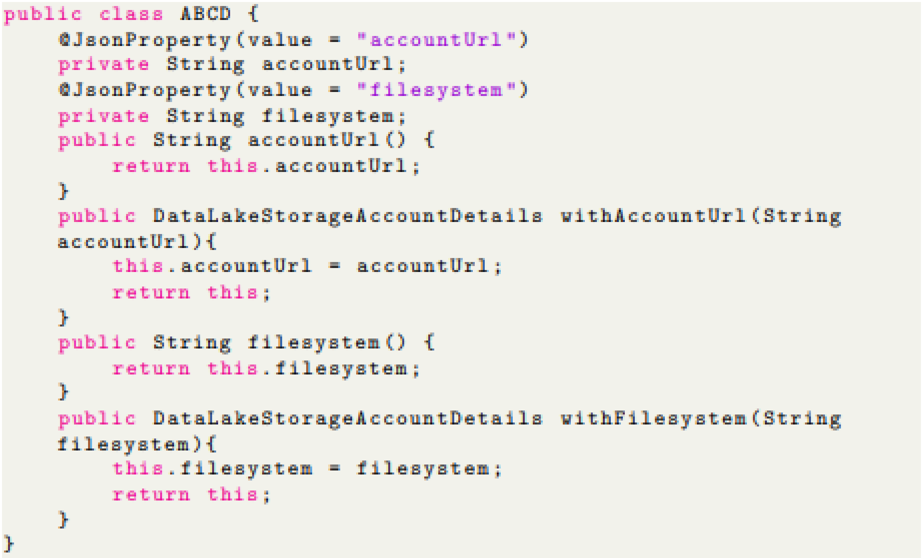
| Github URL List |
|---|
| https://github.com/gz-yami/mall4j |
| https://github.com/muyinchen/migoShop |
| https://github.com/coderbruis/JavaSourceCodeLearning |
| https://github.com/wistbean/manong-ssm |
| https://github.com/thrsky/ThrskyShop |
| https://github.com/zhulinn/SpringBoot-Angular7-Online-Shopping-Store |

Table Continues

Table Continued

| Github URL List |
|---|
| https://github.com/StevenWash/xxshop |
| https://github.com/Vampx/netcai |
| https://github.com/jbloch/effective-java-3e-source-code |
| https://github.com/5zhu/ssm |
| https://github.com/lkmc2/AwesomeMall |
| https://github.com/jbloch/effective-java-3e-source-code |
| https://github.com/ITpandaffm/LexianManager |
| https://github.com/Wasabi1234/shopping-mmall |
| https://github.com/w312033591/netcai-doc |
| https://github.com/muxin-4/foodie |
| https://github.com/leifchen/mmall |
| https://github.com/awangdev/LintCode |
| https://github.com/candyadmin/likejd |
| https://github.com/winnerczr/baishop |
| https://github.com/muxin-4/foodie |
| https://github.com/leifchen/mmall |
| https://github.com/candyadmin/likejd |
| https://github.com/winnerczr/baishop |

After the successful collection of the source code files, the next step is processing them. It is commonly known that in every programming language where comment blocks are needed to make codes human readable. However, comment blocks and other necessary statements for the program, such as import statements or package statements, only add unnecessary information to the dataset, which is commonly called noise. Because import statements or package statements do not affect the codes' result, the language model does not need to learn them or predict them. Therefore, the first step of processing the dataset is getting rid of all the comment blocks, package statements, and import statements. The second step of processing the dataset is replacing the class names with placeholders. The reason for this action is that class names are completely random, humans created them, and they have no actual meaning, which means they do not affect the outcome of the code. Thus, they play a role of distraction in the dataset, just like package statements and import statements, and the model's accuracy will improve by getting rid of them. The third step of processing the data is dividing all data samples into batches of data. Below is a picture of one of the data samples we use.

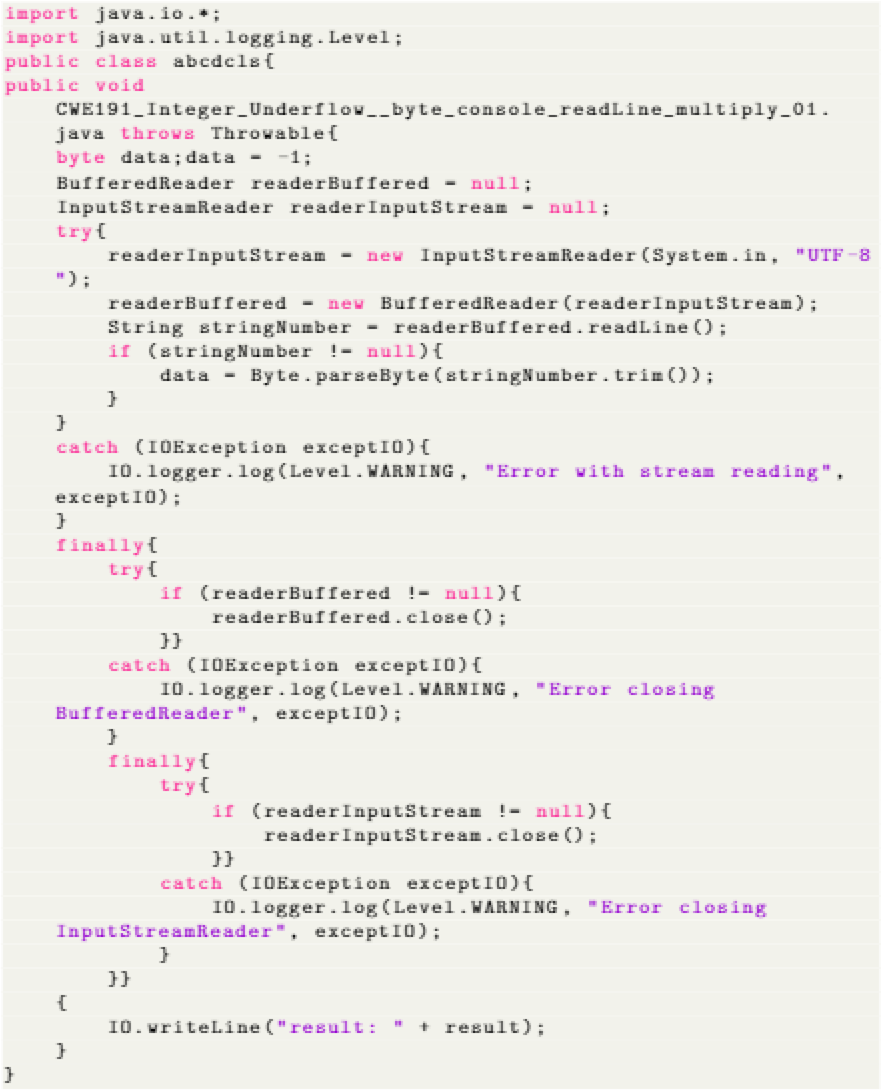


```
public class ABCD {
    @JsonProperty(value = "accountUrl")
    private String accountUrl;
    @JsonProperty(value = "filesystem")
    private String filesystem;
    public String accountUrl() {
        return this.accountUrl;
    }
    public DataLakeStorageAccountDetails withAccountUrl(String
    accountUrl){
        this.accountUrl = accountUrl;
        return this;
    }
    public String filesystem() {
        return this.filesystem;
    }
    public DataLakeStorageAccountDetails withFilesystem(String
    filesystem){
        this.filesystem = filesystem;
        return this;
    }
}
```

*Figure 2: Language Model Data Sample*

### Labeled Source Code Dataset for the Classifier

Java source code dataset for classifier has over 70000 data samples, and it is used as input data for the Classifier model. The Java files in the dataset have labels of specific security vulnerabilities, and the dataset has 92 different labels in total. After successfully collecting the dataset, the next step is putting all the data in the right column. Each data sample has four columns: code, problem description, binary label column for identifying if this data sample has any problem, a multi-class label that put a specific problem code to the row, and if this data sample does not have any problem, then it will be a placeholder for the non-problematic data sample, like the dataset used in the pre-training language model. The second dataset has comment blocks, package statements, and random class names, all of which do not influence the result of the code. So, an essential step of dataset processing is to trim down the dataset to make it more straightforward for the model. The final step is dividing the dataset into batches to save the computational resources. Below is a data sample in the dataset; as shown in the code, the class name has been replaced with a placeholder, and all the comment blocks and package statements have been removed.



```
import java.io.*;
import java.util.logging.Level;
public class abcdcls{
public void
    CWE191_Integer_Underflow__byte_console_readLine_multiply_01.
    java throws Throwable{
    byte data;data = -1;
    BufferedReader readerBuffered = null;
    InputStreamReader readerInputStream = null;
    try{
        readerInputStream = new InputStreamReader(System.in, "UTF-8
    ");
        readerBuffered = new BufferedReader(readerInputStream);
        String stringNumber = readerBuffered.readLine();
        if (stringNumber != null){
            data = Byte.parseByte(stringNumber.trim());
        }
    }
    catch (IOException exceptIO){
        IO.logger.log(Level.WARNING, "Error with stream reading",
    exceptIO);
    }
    finally{
        try{
            if (readerBuffered != null){
                readerBuffered.close();
            }}
        catch (IOException exceptIO){
            IO.logger.log(Level.WARNING, "Error closing
    BufferedReader", exceptIO);
        }
        finally{
            try{
                if (readerInputStream != null){
                    readerInputStream.close();
                }}
            catch (IOException exceptIO){
                IO.logger.log(Level.WARNING, "Error closing
    InputStreamReader", exceptIO);
            }
        }}
    {
        IO.writeLine("result: " + result);
    }
}
```

*Figure 3: Labeled Data Sample for the Classifier*

### T5 Format Dataset for the Summarizer

Java code dataset for Summarizer: The data for the Summarizer is the same data this work uses for the text classifier model. The only difference is how we process it differently. Each data sample has one problematic method and one problem-free method. The dataset for the T5 model has six columns: the first column is called flag, which is the problem of the bad method or the Fixing measure of the good method. For example, the flag is ”FLAW: Read data using an outbound TCP connection." Then it means that the flawed statement, which is a comment block, is in the middle of the entire method. The second column is the actual code of the data sample. The third column is the CWE(Wu et al., 2015) label which is the number of the problem. The fourth column is the description of the problem of the code. For example, CWE: 113 HTTP Response Splitting. The last column is the file name of the data sample. This dataset also gets rid of the unnecessary comment blocks, package statements, and import statements for the same reason written earlier.

# CHAPTER IV: MODELS

The security vulnerability detection system we propose can be broken into two parts: the Classifier and the Summarizer. Figure 3: Dataset-Model-Output shows the structure of this work. Both the dataset for Classifier and the dataset for Summarizer is generated from the CWE dataset. The only difference between them is how we process them. After processing the datasets, each model uses the corresponding dataset and outputs the result.

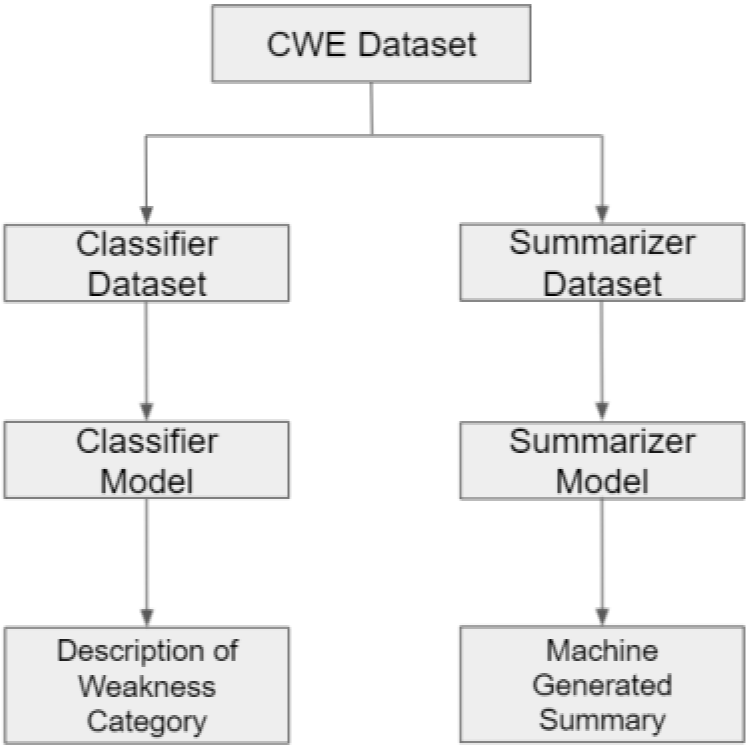


*Figure 4: Research Structure*

## Classifier

The text classification model, which will be referred to as the Classifier for the remainder of this paper, is a text classifier model to detect vulnerabilities in the source code and output the category of the vulnerability, which is also the label of the data sample. The Classifier does not generate a summary of the data. For example, as shown in Figure 2: Data sample, the source code produces a problem: an uninitialized string, which is also the label of this data sample. Technically the above code does not produce any problem, but in real life, uninitialized variables (especially string) could become serious security loopholes in the program. The classifier takes code above as the input and outputs ”CWE563: Unused-Variable-unused-unit-variable-String”. In order to improve the performance of the Classifier, the Classifier uses a language model as the pre-training model. The purpose of this is that after training the language model, its weights get better at predicting source code. The pre-trained language model uses AWDLSTM as its pre-training model. The Classifier’s inside is relatively similar to a standard LSTM model since the AWD-LSTM model is just an LSTM model with other techniques.

### Pre-training Model for the Classifier

This research uses AWD-LSTM (Stephen et al., 2017)(Merity et al., 2017) as the pre-training model for the language model in the text classification approach. AWD-LSTM model is one of the top models in NLP models. As shown in Figure 3, the LSTM model has two different states: cell state, which is what the letter C stands for, and hidden state, which is what the letter h stands for. The state of C changes slowly because the cell state has to go through a sigmoid layer called the ”forgetting gate layer.” The forgetting gate layer outputs 0 and 1 for each number in the previous cell state (Ct-1), where 0 means forget it entirely, and one means keep it. As the training goes by, the cell state remembers information we through each layer and forget the information we do not need, and then we get our long-term memory part of the LSTM. Unlike cell state, the Hidden state changes rapidly because of the similar mechanics but with a tanh function. The hidden state is also known as the short memory part of LSTM. Those two different paces of changing state make LSTM models excellent models for solving NLP problems where the data samples are consist of a significant number of words.AWD-LSTM model uses additional regularization, NT-ASGD optimization, and DropConnect techniques to improve the model's performance. This research uses AWD-LSTM as the pre-training model of the language model, which is used as the pre-training model for the Classifier.

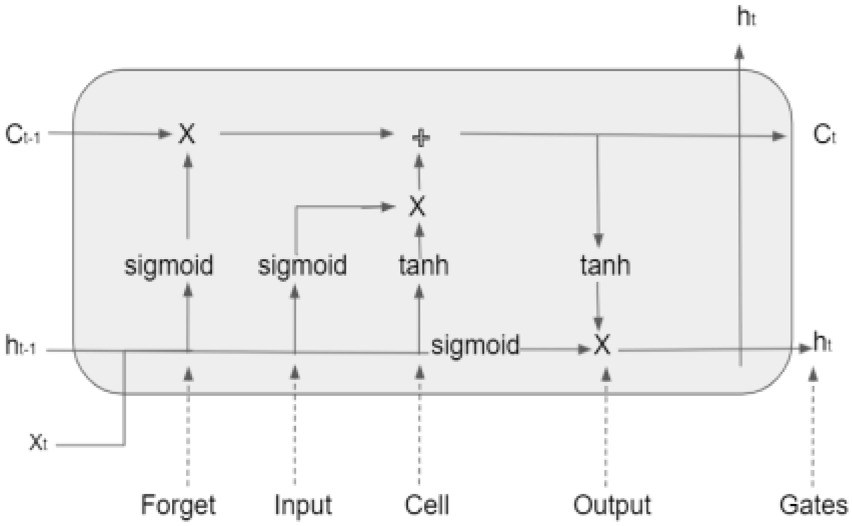


*Figure 5: AWD-LSTM Diagram*

**Summarizer**

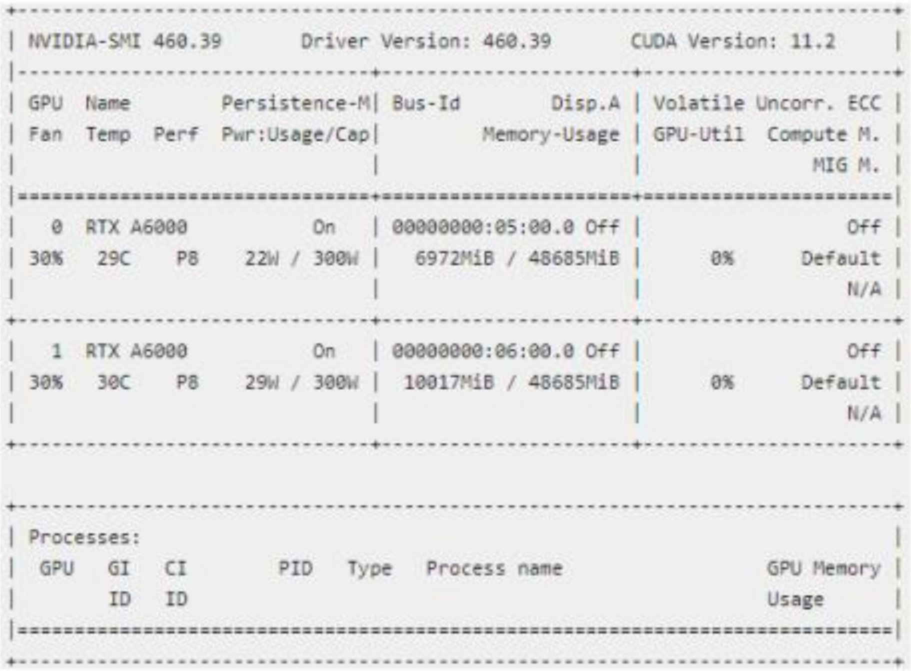
The vulnerability summarization model, referred to as the Summarizer for the remainder of this paper, is a text-to-text model used to describe vulnerabilities detected by the Classifier. The Classifier's role is to detect a vulnerability in the source code. The Summarizer's role is to describe what may be causing the vulnerability, often even suggesting what should be fixed. The Summarizer is trained to be as human-readable as possible, meaning the end-user isn't required to decipher cryptic CWE codes. The main difference between the output of the Classifier and the output of the Summarizer is that the output of the Classifier is either right or wrong because the Classifier outputs the category of the problem of the input and the all the categories are human-generated, the Classifier just recognize the category of the input and output the right label for that. Meanwhile, the Summarizer outputs the summary of the input, and the result is completely machine-generated. The Summarizer uses T5 as the pretraining model to improve the performance of the model. T5 model is based on the Transformer architecture(Vaswani et al., 2017), and it has not made many changes to the original architecture.

## Pre-training Model for the Summarizer

This research uses the T5 (Text-to-Text Transfer Transformer) model (Colin et al., 2020) (Raffel et al., 2020) as the pre-training model for the text generation approach. The input of the T5 model has a particular format which is ”task: text.” For example, if one needs to translate ”what a beautiful day” from English to German, the data sample needs to be formatted: ”translate English to Germany: what a beautiful day.” The model is supposed to be trained to output ”was f¨ur ein sch¨oner Tag.” The T5 model has been pre-trained on wide variety of NLP tasks including classification, translation, summarization, question-answering, etc.(Raffel et al., 2020) Unlike most other pre-training forms such as language modeling, all tasks are done in a Text-to-Text manner, meaning the same model, loss function, and hyperparameters are used to train the model across all tasks. To achieve this, each sequence of input text is prepended with a description of the task followed by a colon. For example, an input of ”translate English to German: That is good.” will output the text sequence ”Das ist gut.” Another example of the T5 task would be text classification tasks: the T5 model predicts a single word regarding the label. For example, on the MNLI benchmark(Williams et al., 2018) the goal is to distinct a premise implies(entailment), contradicts(contradiction) or neither(neutral). For this task, the input of the model becomes ”mnli premis: I hate pigeons. Hypothesis: My feelings towards pigeons are filled with animosity”. The model outputs the word ”entailment” as the classification of the input.

## CHAPTER V: PERFORMANCE EVALUATION

### Experiment Settings



```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 460.39       Driver Version: 460.39       CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  RTX A6000           On   | 00000000:05:00.0 Off |                  Off |
| 30%   29C    P8    22W / 300W |   6972MiB / 48685MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   1  RTX A6000           On   | 00000000:06:00.0 Off |                  Off |
| 30%   30C    P8    29W / 300W |  10017MiB / 48685MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
+-----------------------------------------------------------------------------+
```

*Figure 6: Computational Resources*

This work is done on the Lamda Server of Illinois State University. The server’s hardware configuration is shown in Figure 6. The Lamda Server has two Nvdia RTX A6000. Each GPU has 48GB of memory, for a total of 96GB. This advanced hardware set allows for much larger batch size and thus faster training. This work uses two different python libraries to build the models. One is Pytorch(Paszke et al., 2019) which is an easy-to-use, high-performance deep learning library that has been widely used in the whole world. This work uses Pytorch to build the Summarizer Model. The other is FastAI(Howard and Gugger, 2020), a deep learning library that uses Pytorch as its foundation. This work uses FastAI to build the Classifier Model.

### Performance Metrics

This work presents two NLP models: The Classifier and the Summarizer. Both models use a pre-training model to improve the performance, and the result is excellent. Both models generate high-quality output for auto-detection of source code vulnerabilities. This section will discuss the result of Classifier first from three different angles: the model it uses and the accuracy and the size of the model. Then this section will talk about the BLEU score and ROUGE score in general because this work uses the BLEU score and ROUGE score as the criteria for performance evaluation of the Summarizer. Below is a table of the result from the Classifier and a sampler model, which is just a text classification model using AWD-LSTM as the pre-training model.

### Compare Classifier with Sampler Model

| Model | Accuracy | #Parameters |
|---|---|---|
| AWD-LSTM | 85.73% | 67K |
| Classifier | 87.77% | 67K |

*Table 2: Result of Pre-training*

This work uses pre-training and fine-tuning to improve the performance of the Classifier. Hence, this work compares the differences between the result of plain AWD-LSTM, which will be called the sampler model in the following, and the results of the Classifier of our work. By comparing the result of two models, one can observe whether the techniques of our work can make a difference. The sampler model and the Classifier have the same number of parameters, which means they both are the same size which means that both the sampler model and the Classifier have the same number of parameters because they are both based on AWD-LSTM models. The same number of parameters means that with the fixed size of models, one can conclude that the accuracy improves because of the transfer learning and the fine-tuning. This is another angle to explain how the pre-training and the fine-tuning work well. As shown above, the accuracy of plain AWD-LSTM is 85.73%, which is an acceptable accuracy for source code classification. However, after fine-tuning and using the transfer learning technique, the accuracy has jumped to 87.77%. Because of the pretraining language model, the parameters in the classifier has been adjusted to learn source code files, and after the fine-tuning process such as adjusting the hyperparameters. The accuracy has improved. This work believes that 87.77% accuracy over 92 different labels is an acceptable performance for source code classification tasks since the randomness of the programming code and the uncertainty of source code files.

### BLEU Score for Summarizer

This work uses the T5 model as the pre-training model for the Summarizer. As for the result of the Summarizer, this work uses BLEU as the main method to measure the model’s performance. The goal of BLEU score (Papineni et al., 2002a) is to calculate how many words in the model generated words appeared in the human-generated summaries, and The Summarizer gets a BLEU score of 47.18. It is well known that predictions with BLEU score above 40 are quality summarization. The formula for calculating the BLEU score is shown below. In general, the precision of an n-gram, which is Pn (unigram would be P1, bigram would be P2), is calculated by add the number of n-grams matches with the actual result, then we add the clipped n-gram counts for all the candidate translations and then divide by the number of candidate n-grams in the actual translations to get our Pn (Papineni et al., 2002b).

$$p_n = \frac{\sum_{C \in \{candidates\}} \sum_{n-gram \in C} Count_{clip}\,(n - gram)}{\sum_{C' \in \{candidates\}} \sum_{n-gram \in C'} Count\,(n - gram')}$$

*Figure 7: BLEU Score Pn Formula*

After that, one can get the BLEU score for the translation based on the following formula.

$$BLEU = BP \cdot exp\left(\sum_{n=1}^{N} W_n log p_n\right)$$

*Figure 8: Final BLEU Score Formula*

The output predictions are shown below. This work calculates BLEU score by using the formula with the Actual Summary, which is the label for the data sample in the dataset, and the Prediction, which is the generated summary of the Summarizer model.

| Generated Prediction | Actual Summary |
|---|---|
| Use the maximum size of the data type | FLAW: Use the maximum size of the data type |

*Table 3: Summarizer Predictions*

**ROGUE Score for Summarizer**

The purpose of the ROUGE score (Lin, 2004a) is to calculate how many words in the human-generated words appeared in the model-generated summaries. The Summarizer gets a ROUGE score of 59.42, which is an excellent score because any summarization with a ROUGE score above 50 is useful and human readable (Lin, 2004b). Below is the mathematical formula of how the ROUGE score is calculated.

$$ROUGE - N = \frac{\sum_{S \in \{ReferemceSummaries\}} \sum_{gram_n \in S} Count_{match}(gram_n)}{\sum_{S \in \{ReferemceSummaries\}} \sum_{gram_n \in S} Count(gram_n)}$$

*Figure 9: ROUGE Score Formula*

Unlike the BLEU score, the ROUGE score focuses on recall. Because the denominator of the formula is the total sum of the number of n-grams occurring at the reference summary, the Rouge score does an exceptional job at calculating the recall of the summaries, which means how much the words (n-grams) in the human reference summaries appeared in the machine-generated summaries. The Summarizer of this work got a 59.42 ROUGE score, which means more than half of the words in the human references appear in the machine-generated summaries, which means quality summaries that are capable of using for practical work.

# CHAPTER VI: CONCLUSION

This research uses two different NLP models with two different deep learning frameworks using the same original dataset. Both models get impressive performance in the auto-detection of security vulnerabilities in source code files. If this research gets implemented on the cloud server, it could be a revolutionary change in the software testing industry. However, Although, we have proved in our research that NLP models are a very promising direction for source code auto-detection, future work is still needed. There are several directions to develop the depth of this research: the first direction is multi-language models that can detect security vulnerabilities in more than one programming language.  The second direction is expanding the dataset's size significantly. The dataset this research uses is still limited for applying the models in real life. A limited dataset means that models will have a higher chance of producing false negatives. The third direction is to pinpoint the location of harmful code in the source code files to make the models more functional.

# REFERENCES

Bau, J., Bursztein, E., Gupta, D., & Mitchell, J. (2010, May). State of the art: Automated black-box web application vulnerability testing. In *2010 IEEE symposium on security and privacy* (pp. 332-345).IEEE.

Bayer, U., Moser, A., Kruegel, C., & Kirda, E. (2006). Dynamic analysis of malicious code. *Journal in Computer Virology*, *2*(1), 67-77.

Ghosh, A. K., & Swaminatha, T. M. (2001). Software security and privacy risks in mobile e-commerce. *Communications of the ACM*, *44*(2), 51-57.

Gomes, I., Morgado, P., Gomes, T., & Moreira, R. (2009). An overview on the static code analysis approach in software development. *Faculdade de Engenharia da Universidade do Porto, Portugal*.

Goseva-Popstojanova, K., & Perhinschi, A. (2015). On the capability of static code analysis to detect security vulnerabilities. *Information and Software Technology*, *68*, 18-33.

Harer, J. A., Kim, L. Y., Russell, R. L., Ozdemir, O., Kosta, L. R., Rangamani, A., ... & Lazovich, T. (2018). Automated software vulnerability detection with machine learning. *arXiv preprint arXiv:1803.04497*.

Howard, J., & Gugger, S. (2020). Fastai: a layered API for deep learning. *Information*, *11*(2), 108.

Howard, J., & Ruder, S. (2018). Universal language model fine-tuning for text classification. *arXiv preprint arXiv:1801.06146*.

Kaiser, L., Gomez, A. N., Shazeer, N., Vaswani, A., Parmar, N., Jones, L., & Uszkoreit, J. (2017). One model to learn them all. *arXiv preprint arXiv:1706.05137*.

Karpathy, A., Johnson, J., & Fei-Fei, L. (2015). Visualizing and understanding recurrent networks. *arXiv preprint arXiv:1506.02078*.

Kaur, A., & Nayyar, R. (2020). A comparative study of static code analysis tools for vulnerability detection in c/c++ and java source code. *Procedia Computer Science*, *171*, 2023-2029.

Kim, H., Kang, S., Baik, J., & Ko, I. (2007, July). Test cases generation from UML activity diagrams. In *Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD 2007)* (Vol. 3, pp. 556-561). IEEE.

Lachaux, M. A., Roziere, B., Chanussot, L., & Lample, G. (2020). Unsupervised translation of programming languages. *arXiv preprint arXiv:2006.03511*.

Lin, C. Y. (2004, July). Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out* (pp. 74-81).

Lin, C. Y. (2004, July). Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out* (pp. 74-81).

Lin, G., Zhang, J., Luo, W., Pan, L., & Xiang, Y. (2017, October). POSTER: Vulnerability discovery with function representation learning from unlabeled projects. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (pp. 2539-2541).

Mahmood, R., & Mahmoud, Q. H. (2018). Evaluation of static analysis tools for finding vulnerabilities in Java and C/C++ source code. *arXiv preprint arXiv:1805.09040*.

Mantere, M., Uusitalo, I., & Roning, J. (2009, June). Comparison of static code analysis tools. In *2009 Third International Conference on Emerging Security Information, Systems and Technologies* (pp.15-22). IEEE.

Martin, R. A. (2007). Common weakness enumeration. *Mitre Corporation*.

McGraw, G. (2004). Software security. *IEEE Security & Privacy*, *2*(2), 80-83.

Meng, Q., Wen, S., Zhang, B., & Tang, C. (2016, August). Automatically discover vulnerability through similar functions. In *2016 Progress in Electromagnetic Research Symposium (PIERS)* (pp. 3657-3661). IEEE.

Merity, S., Keskar, N. S., & Socher, R. (2017). Regularizing and optimizing LSTM language models. *arXiv preprint arXiv:1708.02182*.

Meudec, C. (1998). *Automatic generation of software test cases from formal specifications* (Doctoral dissertation, Queen's University of Belfast).

Papineni, K., Roukos, S., Ward, T., & Zhu, W. J. (2002, July). Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics* (pp. 311-318).

Papineni, K., Roukos, S., Ward, T., & Zhu, W. J. (2002, July). Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics* (pp. 311-318).

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., ... & Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, *32*, 8026-8037.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., ... & Liu, P. J. (2019). Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683*.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., ... & Fei-Fei, L. (2015). Imagenet large scale visual recognition challenge. *International journal of computer vision, 115*(3), 211-252.

Shanthi, A. V. K., & Kumar, D. G. M. (2011). Automated test cases generation for object oriented software. *Indian Journal of Computer Science and Engineering*, *2*(4), 543-546.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention is all you need. In *Advances in neural information processing systems* (pp.5998-6008).

Wagner, D. A., Foster, J. S., Brewer, E. A., & Aiken, A. (2000, February). A first step towards automated detection of buffer overrun vulnerabilities. In *NDSS* (Vol. 20, No. 0, p. 0).

Williams, A., Nangia, N., & Bowman, S. R. (2017). A broad-coverage challenge corpus for sentence understanding through inference. *arXiv preprint arXiv:1704.05426*.

Wu, Y., Bojanova, I., & Yesha, Y. (2015). They know your weaknesses–do you?: Reintroducing common weakness enumeration. *CrossTalk*, *45*.

Yamaguchi, F., Lottmann, M., & Rieck, K. (2012, December). Generalized vulnerability extrapolation using abstract syntax trees. In *Proceedings of the 28th Annual Computer Security Applications Conference* (pp. 359-368).

Yang, X., Chen, J., Yedida, R., Yu, Z., & Menzies, T. (2021). Learning to recognize actionable static code warnings (is intrinsically easy). *Empirical Software Engineering*, *26*(3), 1-24.

Zhioua, Z., Short, S., & Roudier, Y. (2014, July). Static code analysis for software security verification: Problems and approaches. In *2014 IEEE 38th International Computer Software and Applications Conference Workshops* (pp. 102-109). IEEE.

Zhuang, Y., Liu, Z., Qian, P., Liu, Q., Wang, X., & He, Q. (2020). Smart Contract Vulnerability| Detection using Graph Neural Network. In *IJCAI* (pp. 3283-3290).