CS230

Identifier Aware Vulnerability Detection using Deep Learning (Natural Language Processing, Programming Languages)

Neelima Mukiri
neelimam@stanford.edu

Abstract

Detecting software vulnerabilities from source code is an important problem and has been actively researched in the past years. There are many source code analysis tools that use rule based systems to detect common mistakes in code that lead to vulnerable software. This paper combines the use of intermediate representation of source code or code gadgets with transformer based models, specifically comparing identifier aware models with models trained on natural language alone and compares the performance of the models in detecting and identifying software vulnerabilities. The results show comparable results with both the models and show a macro F1 score of 96.58% showing that transformer models are well suited to identifying software vulnerabilities, even though the changes between vulnerable and non-vulnerable code is sometimes minimal. Use of code gadgets allows modeling dependencies across various source code libraries strongly to make automatic detection feasible. This is a 3 percent improvement over both [1] and [2] which use Code Gadgets with LSTMs or plain source code with BERT + LSTM.

1 Introduction

We work on a set of products based on Linux and Kubernetes which include many open source components. Our team spends a significant amount of time identifying, tracking and resolving security vulnerabilities, so that our customers have a secure production grade environment . As [9] and [10] show the number of vulnerabilities in Kubernetes and Linux over the years is large and growing. Also the cost of an undetected software vulnerability in production is high and opens up customers to data loss, data corruption and vulnerable to hacking. So we want to automate the detection of software vulnerabilities to benefit our product development cycle as well as the larger community.

In this paper, I have focused on combining intermediate code representation with transformer based models to detect software vulnerabilities. I have evaluated the performance on encoder only models(BERT) and compared to identifier aware encoder-decoder CodeT5 models.

CS230: Deep Learning, Winter 2018, Stanford University, CA. (LateX template borrowed from NIPS 2017.)

2 Related Work

2.1 VulDeePecker (Code Gadgets and Bidirectional LSTM)

[2] introduces 3 guiding principles: code can be transformed into an intermediate representation, the optimal granularity for detecting source code vulnerabilities is using code gadgets (CGD), and detecting vulnerabilities in source code is dependent on the context of the source code. It uses a Bidirectional Long Short-Term Memory cells to encode context of code before and after a given token. This paper also introduces the VulDeePecker dataset that I have used. It reports an F1 score of 86.6 on Buffer Error Vulnerabilities(CWE-119) and 95 on Resource Management Vulnerabilities(CWE-399). The tokenization mechanism used in this paper is to vector encode each code gadget based on all the unique tokens in the dataset. So this essentially ignores and semantic relation between different tokens.

2.2 Security Vulnerability Detection Using Deep Learning Natural Language Processing (Transformers)

[1] has compared using Transformer models to Bidirectional LSTM and combining the two to detect source code vulnerabilities. The primary focus of this paper has been to identify mechanisms that can predict a vulnerability fast, so that it can be used to augment a source code editor. Hence they forgo the use of an intermediate representation and work on source code directly. Source code is split into segments and fed to the transformer model (BERT). To encapsulate the context across different segments, they then feed the output of the transformer model to a Bidirectional LSTM. The paper reports an accuracy of 93.49 using the combination of BERT + LSTM.

2.3 Natural language vs Programming Language representations

Existing research has primarily treated code vulnerability detection either by going into generated intermediate representations of code [4] or by treating code as natural language([1],[2],[3]) . The approach in [4] completely discards syntactic information and the other approaches completely ignore code structure and identifier information. To find a middle ground, I have used Code Gadgets which allow encoding source code dependencies across function calls and Code-T5 tokenizer, which is an identifier aware model trained on source code. [1] has primarily focused on identifying vulnerabilities in functions. This is not easily generalized to any source code, as it expects input to be localized as a function. This is solved by the use of code gadgets in ([2] and [3]).

3 Dataset

VulDeePecker vulnerability data set corresponding to two sets of vulnerabilities - Buffer Error Vulnerabilities(CWE-119) and Resource Mangement Vulnerabilities(CWE-399). The data is pre-processed and labeled to give code gadgets - which are one dependency control flow or data flow graph for some selected source code.

4 Preprocessing

As seen in the above examples, existing data set has vulnerability information embedded in the code - comment, function names etc. I've removed text that is indicative of the code health from the training data. Some functions have Common Weakness Enumeration or Common Vulnerabilities and Exposures names and identifiers in them, which I have removed.

Here is an example of a sample data after pre-processing, before tokenization.

```
char *psz_fileName = calloc( ZIP_FILENAME_LEN, 1 ); ZIP_FILENAME_LEN, NULL, 0, NULL, 0 )
if( unzGetCurrentFileInfo( file, p_fileInfo, psz_fileName, vlc_array_append( p_filenames, strdup( psz_fileName ) );
free( psz_fileName );
```

The original dataset consists of 61638 samples. During error analysis, I found that there were may samples with duplicates and samples which had both a Vulnerability and a No-Vulnerability label, leading to a higher error rate. After removing all the duplicates and the samples with conflicting labels, I found a large data imbalance as shown in Figure 3. There was a 6x representation of

```c
#define BUFSIZE 256

int main(int argc, char **argv)
{
  char *buf;
  buf = (char *)malloc(BUFSIZE);
  if (buf == NULL)
        {printf("Memory allocation problem"); return 1;}

  if (argc > 1 && strlen(argv[1]) < BUFSIZE)          /* FIX
  {
    strcpy(buf, argv[1]);
    printf("buf = %s\n", buf);
  }
  free(buf);
  return 0;
}
```

Figure 1: Non-Vulnerable Code

```c
#define BUFSIZE 256

int main(int argc, char **argv)
{
  char *buf;
  buf = (char *)malloc(BUFSIZE);
  if (buf == NULL)
        {printf("Memory allocation problem"); return 1;}

  if (argc > 1)
  {
    strcpy(buf, argv[1]);              /* FLAW */
    printf("buf = %s\n", buf);
  }
  free(buf);
  return 0;
}
```

Figure 2: Vulnerable Code

Dataset	Original	Pre-Processed (Remove Duplicates, Drop conflicts)	Data Augmentation (Over sampling)
No Vulnerability	43913	17031	17031
CWE-119	10440	7485	14970
CWE-399	7285	2754	16524

Figure 3: Data Processing

non-vulnerable code vs code with resource management errors(CWE-399). I used oversampling to bring the data samples to a similar count ending up with 48525 total samples.

5 Learning Method

I used BERT and CodeT5 transformer models as the base and fine tuned them with the pre-processed VulDeePecker dataset. As BERT is an encoder-only model, fine tuning was simply changing the final layer to use a 3 class soft-max classifier as shown in Figure 4. For the Code-T5 model, Figure 5 shows the option with an additional linear layer added after the last hidden layer followed by a 3 class softmax classifier.

The loss function used was cross entropy loss and the optimization was done using Adam with weight decay. Cross Entropy Loss

$$\sum_{c=1}^{M} y_{o,c} \log(p_{o,c})$$

Code-T5 tokenization with labels added was used for the Code-T5 model. Fine-tuning experiments have been done primarily on an Nvidia A100 GPU.

6 Experiments

6.1 Neural Architecture Search

I fine-tuned the following models and compared their performance:

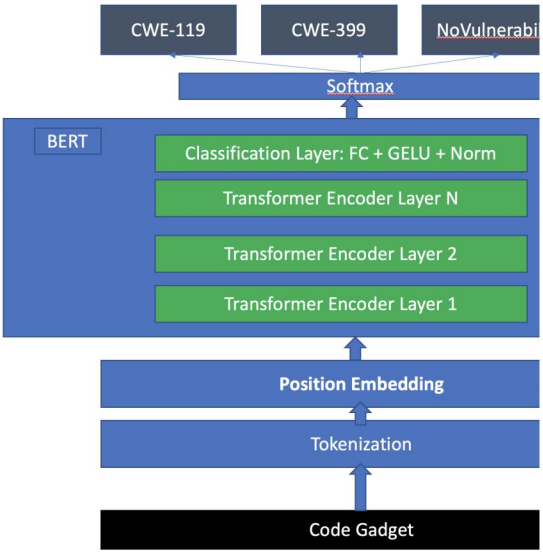

Figure 4: BERT Fine-tuning

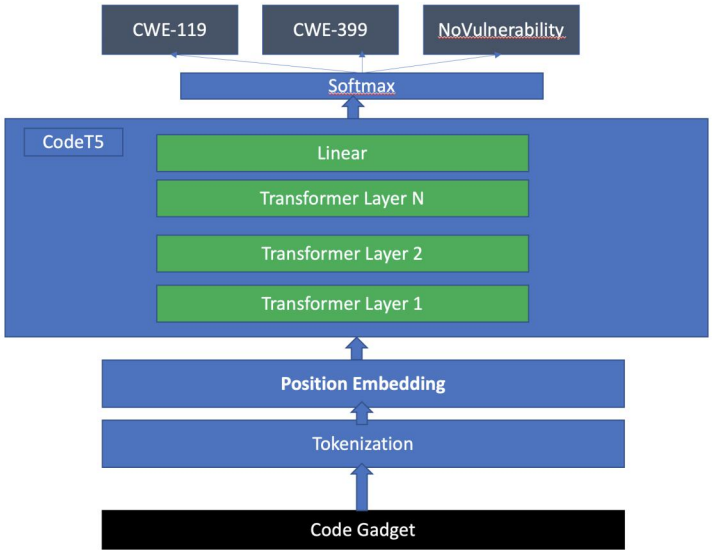

Figure 5: CodeT5 fine tuning

- BERT (bert-base-uncased without data augmentation): This was the base line model I started with. I changed the final layer of the model to a 3-class softmax function and used cross entropy loss. Though the performance was pretty good(96.65), it showed a large variability in performance across the different classes (90.5 vs 97.06).
- BERT (bert-base-uncased with data augmentation): I then used over sampling to reduce the imbalance in the dataset across different classes. This reduced the variability across classes (95.0 to 97.89) and improved the overall performance to 96.58.
- CodeT5 (codet5-small with data augmentation): As CodeT5 is an encoder-decode model, I tried two experiments for fine tuning. One was to use in as a sequence-to-sequence model, but to train it to output the vulnerability class or absence of vulnerability as the output sequence. To enable this I added CWE-119, CWE-399, and NoVulnerabilty labels to the CodeT5 tokenizer and updated the model to use the new tokens. That allowed it to correctly classify the vulnerabilities with a F1 score of 96.21. The second approach was to take the last hidden layer output and use that as the input to a linear layer and then use a softmax classifier to identify the vulnerability details which gave a similar performance.

6.2 Hyper-parameter Tuning

I used Adam with Weight Decay as the optimization algorithm for fine tuning. I experimented with batch sizes from 128 to 16. Anything above 64 was hitting the GPU memory limits while using the bert-base-uncased or codet5 models and hence I used a batch size of 48 for most of the experiments. I used a learning rate of 5e-5 for CodeT5 and 2e-5 for BERT models as this was recommended for the BERT base model. Below is the example of how fast the training converged for the BERT model. Layer selection information is detailed in the above section.

7 Results and Discussion

I evaluated the results using per-class F1 score and macro F1-score as a single metric across the different classes. Using Code-Gadgets as an intermediate code representation gave the best performance while using Transformer models as compared with [1]. As shown below, BERT with data augmentation performed the best of the models evaluated.

This was surprising to me as I expected CodeT5 to perform significantly better. One of the reasons for this might be the data pre-processing that is done. Using Code Gadgets instead of source code removes some of the code structure from the dataset. Also, the original data had variables and

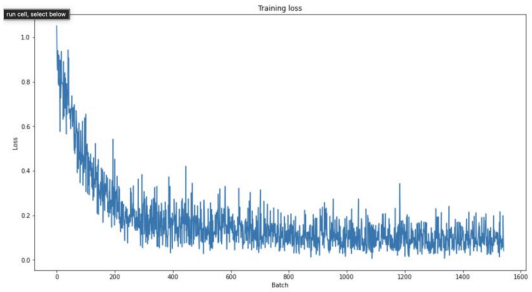

Figure 6: BERT Fine-tuning

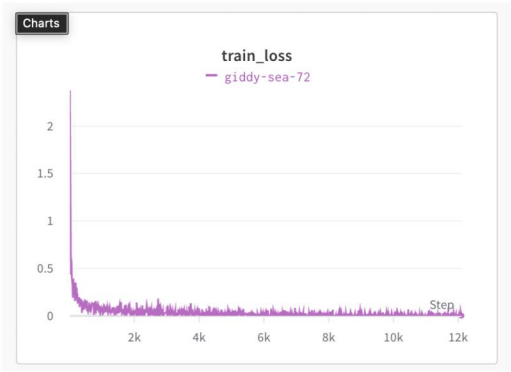

Figure 7: CodeT5 fine tuning

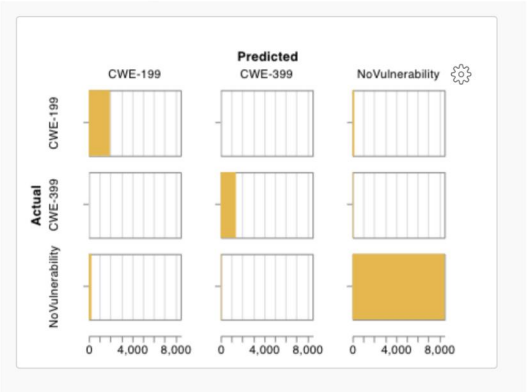

Figure 8: BERT baseline

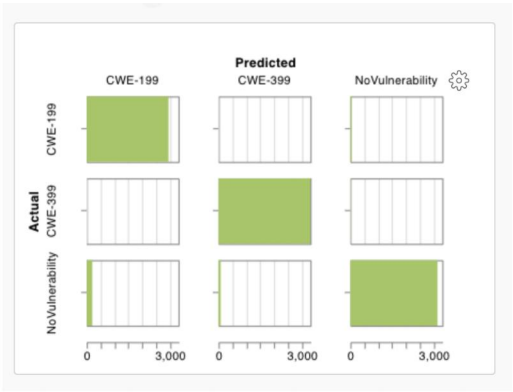

Figure 9: BERT data-augmentation

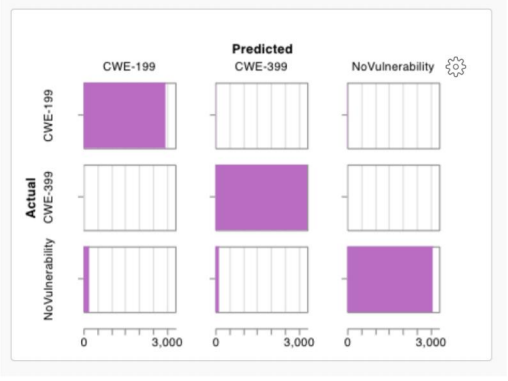

Figure 10: CodeT5 small

Model Architecture/ Experiment	F1 Score (macro)	F1 Score No Vulnerability	F1 Score CWE-119	F1 Score CWE-399
bert-base-uncased	94.65	97.06	90.5	96.3
bert-base-uncased (Data augmentation)	96.58	95.0	96.0	98.74
CodeT5-small(Data augmentation)	96.21	94.59	96.16	97.89

Figure 11: F1 scores

functions named 'good*' or 'bad*' to represent the presence of a vulnerability. I replaced all of them with 'something' thus nullifying the value of any information the identifiers were bringing. Code identifier information is essential to tasks like code generation or auto-completion where we are trying to emulate human readable code generation. A vulnerability detector however needs to be immune to intentionally introduced tokens like 'good' and 'bad' and instead focus on the cause of the vulnerability. Hence the context of the source code seems to be the most relevant factor for detecting vulnerabilities.

8 Acknowledgements

I would like to thank Meenakshi Kaushik for collaborating on the project proposal.

9 References

[1] Noah Ziems and Shaoen Wu. Security Vulnerability Detection Using Deep Learning Natural Language Processing, 2021; arXiv:2105.02388.

[2] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng and Yuyi Zhong. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection, 2018; arXiv:1801.01681. DOI: 10.14722/ndss.2018.23158.

[3] Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li and Hai Jin. VulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection, 2020; arXiv:2001.02334. DOI: 10.1109/TDSC.2019.2942930.

[4] Mingyue Yang. Using Machine Learning to Detect Software Vulnerabilities, 2020;

[5] Yue Wang, Weishi Wang, Shafiq Joty and Steven C. H. Hoi. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation, 2021; arXiv:2109.00859.

[6] NIST software assurance reference dataset project. https://samate.nist.gov/SARD/.

[7] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser and Illia Polosukhin. Attention Is All You Need, 2017; arXiv:1706.03762.

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, 2018; arXiv:1810.04805.

[9] Magno Logan Minding the Gaps: The State of Vulnerabilities in Cloud Native Applications https://www.trendmicro.com/vinfo/us/security/news/virtualization-and-cloud/minding-the-gaps-the-state-of-vulnerabilities-in-cloud-native-applications

[10] CVSS Severity Distribution Over Time https://nvd.nist.gov/general/visualizations/vulnerability-visualizations/cvss-severity-distribution-over-time

[11] https://kubernetes.io/

[12] Denis Rothman, Transformers for Natural Language Processing,