Current Topics in Software Engineering: Automating Software Engineering

Group F

Merlin Volkmer Ingrid Guza Ahmed Aboahmed

Contents

Introduction

For our submission, we have experimented with the big-vul dataset to achieve a generic classifier for detecting vulnerabilities in C++ on a per function and per line detection. For the first milestone, we have relied on the pretrained CodeBert tokenizer to generate the vector embeddings. Furthermore, we navigated through state of the art research approaches to facilitate our second milestone. This resulted in attempting three different approaches for the second milestone: First, a native CodeBert binary classifier. Second, a fine-tuned CodeBert model specifically for C++ code formulated the foundation for a binary sequence classifier. Third, we attempted a bidirectional long short term memory (BILSTM) neural network approach where we relied only on the fine tuned Codebert to obtain the embeddings which were fed to the network. Due to the severe requirements for resources on the BILSTM to achieve reasonable results, the third milestone built upon the second approach in the second milestone. A separate token classification model is trained on the same dataset with resampling in the training dataset. The token classification model is only triggered when the sequence classification highlights a function as vulnerable. We also investigated a sliding window mechanism as well as the use of the Longformer model and both options were deprecated due to lack of significant results and resource constraints.

Milestone 1: Embedding Extraction

Thanks to the precleaned dataset, We did not have to do manual clean up on the dataset to remove comments and other non syntax related additions. However, we deleted the unnecessary columns such as `bigvul_id` and initially `flaw_line_no` which was not necessary for the milestone. Furthermore, splitting the dataset into training, testing and evaluation sets. We used the huggingface trainer library for loading the tokenizer and the model.

Milestone 2: Per Function Classification

For Milestone 2, we have developed three different approaches:

1. Basic CodeBert Binary Classification
2. FineTuned CodeBert for C++ Binary Classification
3. BILSTM Neural Network Binary Classification with FineTuned CodeBert embeddings

Basic CodeBert Binary Classification

We use an already existing tokenizer to train a classifier to label a function as vulnerable or non-vulnerable. We decided to use the base coderbert model by microsoft[1] since it is well established as a simple and efficient baseline model. Upon looking at other model we could use we found a codebert model fine-tuned on C++[2] which led to marginal improvements so we decided to use it for milestone 3.

	precision	recall	f1-score	support
Non-vulnerable	0.99	1.00	0.99	21329
Vulnerable	0.94	0.80	0.86	1055
accuracy			0.99	22384
macro avg	0.96	0.90	0.93	22384
weighted avg	0.99	0.99	0.99	22384

Accuracy: 0.9880
MCC: 0.8591
ROC AUC: 0.9740

Figure 1: Results os using `neulab/codebert-cpp` to train a classifier

[1] https://huggingface.co/microsoft/codebert-base
[2] https://huggingface.co/neulab/codebert-cpp

BILSTM Neural Network Binary Classification

This approach is based on the paper "VulD-CodeBERT: CodeBERT-based Vulnerability Detection Model for C/C++ Code 2024" which involves the use of two models with a soft voting mechanism. The first model is a BILSTM model while the other is a multinomial binary classifier. The combination of the two classification models aims to overcome two issues:

1. Imbalanced datasets which is the case in our Big-Vul dataset with 95% non vulnerable snippents.
2. The need for huge amounts of data to have a good classification model. While the BILSTM requires significant amounts of data, the MultiNominal classifier requires significantly less data which can assist with false positives, false negatives when the difference between the characteristics of vulnerable and non-vulnerable are not obvious.

We implemented the BILSTM model which takes two forms of input:

1. Code Fragment as a whole using the special CLS embeddings as Pooler Output from CodeBert
2. Word Fragments as sequence output which is then fed into the BILSTM with attention mechanisms
3. Code fragment and the output of the BILSTM are combined to form the classifier which finally results into the binary classification

The BILSTIM Approach however also had significant issues which caused us to abandon the idea for Milestone 3.

1. Significantly slower training time when compared to binary classifiers built on CodeBert. Ten percent of the data on T4 Colab session requires 1.5 hours to complete.
2. Suffers severely from data imbalance resulting in false positives and false negatives during training as observed on the confusion matrix.
3. Fine Tuning the metrics for training is extremely time consuming and requires various iterations of training.

Milestone 3: Per Vulnerable Line Classification

Abandoned Approaches

Weighted Loss Mechanism

This approach relied on the basic mechanism of triggering the line detection model given that the function is vulnerable. The key distinctions in this approach are the use of a weight loss function and a sliding Window for context. The weighted loss function attempts to resolve the significant dataset imbalance by assigning a higher loss value to the minority category (Vulnerable), while the sliding window mechanism seeks to provide context between lines since a vulnerable line would likely be somehow connected to the previous or following

lines. In testing, the window would be defined as zero thus targeting each specific line. However, due to resource constraints, the colab session never managed to fulfill a full training session with even ten percent of the dataset.

Epoch	Training Loss	Validation Loss	Accuracy	Precision	Recall	F1	Mcc	Auc
1	0.042700	0.092390	0.983694	0.590206	0.147742	0.236326	0.289850	0.778382
2	0.034600	0.109631	0.983276	0.529091	0.187742	0.277143	0.308573	0.763311

Figure 2: Vulnerability detection at the function level using a weighted loss function after two epochs.

Longformer

While looking for options to overcome the 512 token sequence length barrier of CodeBert, we came across Longformer. Longformer is a transformer model for long documents. `longformer-base-4096` is a BERT-like model started from the RoBERTa checkpoint and pretrained for MLM on long documents. It supports sequences of length up to 4096. Longformer uses a combination of a sliding window local attention and global attention. Global attention is user-configured based on the task to allow the model to learn task-specific representations (Allenai n.d.). In theory, this model should spot cross-line patterns CodeBERT misses, so we started implementing this model in our baseline notebook to test this approach. The first change made to implement this model was to enable global-attention mask helper, which was needed by this model with the following snippet on the notebook:

```
def _with_global_attention(enc):
    if isinstance(enc["input_ids"][0], list):
        enc["global_attention_mask"] = [
            [1]+[0]*(len(ids)-1) for ids in enc["input_ids"]]
    elif isinstance(enc["input_ids"], torch.Tensor):
        g = torch.zeros_like(enc["input_ids"])
        g[..., 0] = 1
        enc["global_attention_mask"] = g
    else:
        enc["global_attention_mask"] = [1]+[0]*(len(enc["input_ids"])-1)
    return enc

def tokenize(batch):
    max_len = tokenizer_max_length if not use_tokenizer_max_length
        else tokenizer.model_max_length
    enc = tokenizer(batch["code"],
        padding="max_length", truncation=True, max_length=max_len)
    return _with_global_attention(enc)
```

We then ran this notebook with the following parameters:

```
epochs = 3

tokenizer_name        = "allenai/longformer-base-4096"
fn_level_model_name   = "allenai/longformer-base-4096"
line_level_model_name = "allenai/longformer-base-4096"

use_tokenizer_max_length = True

fn_level_trainer_args = TrainingArguments(
    per_device_train_batch_size = 150,
    per_device_eval_batch_size  = 150,
    gradient_accumulation_steps = 4,
    # other config ommited for brevity
)

line_level_trainer_args = TrainingArguments(
    per_device_train_batch_size = 150,
    per_device_eval_batch_size  = 150,
    gradient_accumulation_steps = 4,
    # other config ommited for brevity
)
```

According to Colab, training the model would take an estimated 20 hours, which was not viable given our constraints. Therefore, we started tweaking the approach and hyperparameters to make the model more lightweight for our conditions. Due to time constraints and the cooldown period of Google Colab, whereby free resources are available once every 24 hours, we also purchased some computing units from the paid services of Google Colab. After checking the cost of training with an A100 GPU on Colab, we found that running the baseline Longformer model would consume about seven compute units per hour, so we had to limit the amount of time we would run this baseline Longformer approach.

To reduce the execution time and save on the computing units, here are the approaches we followed:

1. Gradient checkpointing
2. Used bf16 precision instead of fp16
3. Used LoRA adapters and froze the backbone
4. Used 2048 token length and trained on 50% of the full dataset

The result of these adjustments can be seen in Figure 3.

After consultation and comparison to the other approaches we could run using the free tier of Google Colab, we decided to stop pursuing this approach as it would be too computationally expensive for our course budget while providing little or no metric gain on Big-Vul. Even though the model we used was heavily redacted and pruned as we froze most of the layers, the metrics received would not justify the cost to use Longformer for our use case.

Epoch	Training Loss	Validation Loss	Accuracy	Precision	Recall	F1	Mcc	Auc
1	No log	0.654157	0.870410	0.130769	0.309795	0.183908	0.139917	0.719474
2	No log	0.647741	0.873631	0.133929	0.307517	0.186593	0.142676	0.725323
3	No log	0.646134	0.877174	0.136223	0.300683	0.187500	0.143250	0.726995

Figure 3: Longformer results after improving training time

Baseline CodeBert

This approach represents a basic two model setup with the unmodified dataset (See Figures 4 and 5).

	accuracy	precision	recall	f1	mcc	auc
0	0.987904	0.940647	0.793627	0.860905	0.858004	0.976235

	precision	recall	f1-score	support
Non-vulnerable	0.99	1.00	0.99	26624
Vulnerable	0.94	0.79	0.86	1318
accuracy			0.99	27942
macro avg	0.97	0.90	0.93	27942
weighted avg	0.99	0.99	0.99	27942

Figure 4: Baseline function level vulnerability detection

- The accuracy is inflated given that the imbalanced dataset has almost every line as non vulnerable. Therefore, this is a weak indicative and should not be considered accurate.
- Precision in cause of Vulnerable lines is 70% which is a decent value in flagging vulnerability.
- Recall is not optimal at 45%, basically the model is able to highlight the rough spot of the vulnerability but not the exact line. Furthermore, it also influences negatively the F1 value
- MCC does not achieve the optimal 70% threshold. However, at 55% the model is considered effective at detecting vulnerabilities.
- AUC at 95% highlights great separation capabilities and it also suggests that given the other measures, solving the problem of line-vulnerability could potentially be more than a basic binary classification approach and should incorporate other mechanisms.

	accuracy	precision	recall	f1	mcc	auc
0	0.990238	0.695432	0.450605	0.546867	0.55525	0.957564

	precision	recall	f1-score	support
Non-vulnerable	0.99	1.00	1.00	14091704
Vulnerable	0.70	0.45	0.55	186658
accuracy			0.99	14278362
macro avg	0.84	0.72	0.77	14278362
weighted avg	0.99	0.99	0.99	14278362

Figure 5: Baseline line level vulnerability detection

Balancing the dataset

Balancing the dataset by reducing the non-vulnerable functions by 50 percent and then oversampling the vulnerable ones until the ratio of vulnerable to non-vulnerable functions is one to one resulted in significantly improved results.

	accuracy	precision	recall	f1	mcc	auc
0	0.988047	0.938503	0.798938	0.863115	0.859944	0.974976

	precision	recall	f1-score	support
Non-vulnerable	0.99	1.00	0.99	26624
Vulnerable	0.94	0.80	0.86	1318
accuracy			0.99	27942
macro avg	0.96	0.90	0.93	27942
weighted avg	0.99	0.99	0.99	27942

Figure 6: Function level vulnerability detection after balacing the dataset

	accuracy	precision	recall	f1	mcc	auc
0	0.988307	0.542993	0.666652	0.598502	0.595846	0.954425

	precision	recall	f1-score	support
Non-vulnerable	1.00	0.99	0.99	14091704
Vulnerable	0.54	0.67	0.60	186658
accuracy			0.99	14278362
macro avg	0.77	0.83	0.80	14278362
weighted avg	0.99	0.99	0.99	14278362

Figure 7: Line level vulnerability detection after balacing the dataset

References

Allenai. n.d. "Longformer Is a Transformer Model for Long Documents." Accessed June 21, 2025. https://huggingface.co/allenai/longformer-base-4096.