Vulnerability Detection using Program Slicing and Transformer-based Model

Md Mahbubur Rahman, Jobayer Ahmmed
Md Rayhanul Islam, Samrajya Thapa,
Department of Computer Science, Iowa State University
Email: {mdrahman, jobayer, rayhanul, svthapa}@iastate.edu

Abstract

Current transformer-based software vulnerability detection approaches consider the function as input. However, one drawback of the transformer-based model is that it processes a certain number of input tokens and discards the remaining ones. The discarded tokens may contain the root cause of vulnerabilities, and the predictions may be inaccurate. In this paper, our proposed approach makes slices based on the program points of interest where a generated slice is smaller in length than the function and includes statements from different parts of the function depending on the program point of interest. Thus, it reduces the chance of discarding the root causes of vulnerability from the input. We classify a function as vulnerable or non-vulnerable based on the prediction of its slices. Experimental result shows that our slice-based approach performs better than the function-based approach, and gets a performance increase of 2.7%, 1%, 0.7%, and 0.4% in Recall, F1 score, Accuracy, and Precision, respectively.

1 Problem Statement and Importance

A software vulnerability is a weakness or flaw in a software application that can damage the software's security, usage, or stability. Software vulnerabilities can hinder the operations of software systems, resulting in financial losses for software companies. A recent National Institute of Standards and Technology study shows that the US economy loses about 60 billion dollars yearly in software redistribution, redeployment, and patching due to vulnerabilities [1]. Therefore, it is necessary to detect vulnerabilities before releasing software to real users. In recent years, deep learning-based techniques have been widely used in detecting software vulnerabilities, and transformer-based models, such as CodeBERT [2], VulBERTa [3], LineVul [4], etc., have achieved state-of-the-art performance in vulnerability detection.

Transformer-based approaches use the pre-trained model with the down-streaming data to solve different down-streaming tasks. It takes a function as a flat input sequence and tokenizes it using different techniques. After that, it only considers the first max_length of tokens to feed the transformer where the max_length is a hyper-parameter. However, if the source of the vulnerability exists after the max_length of tokens, it cannot learn the root causes of the vulnerability because it discards that part. To mitigate this issue, one obvious solution is to increase the max_length, but increasing the max_length may produce two problems- (i) it increases the model's complexity, and (ii) it creates unnecessary extra learnable parameters if more padding is added for smaller functions. Therefore, fixing this by increasing the max_length is not a feasible solution. That is why it is important to resolve this issue without increasing the max_length so that the transformer-based model can learn the root causes of vulnerabilities.

This project proposes a new approach to solve the above issues. Instead of discarding the continuous part of the input program, our approach makes slices based on the program points of interest, and a slice can have statements from any part of the program. It is necessary to mention that the generated slice length is usually smaller than that of the function. Additionally, we assume that the number of tokens of the generated slices will not exceed the max_length of the model. Finally, it sends these slices to the transformer-based model for training and testing. By doing so, it reduces the length of the input program and maximizes the probability of using the root causes of the vulnerability in the model input.

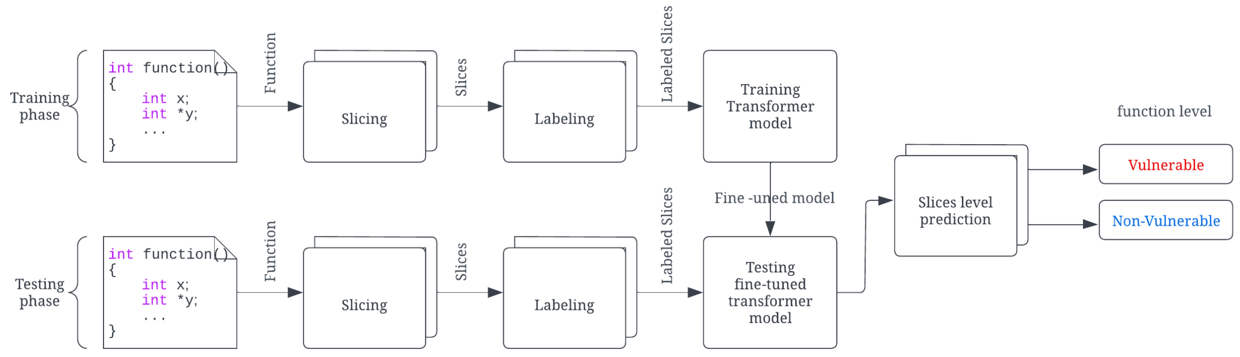

Figure 1: The architecture of our proposed model

2 Methodology

Our proposed algorithm is divided into two phases: Training and Testing. At first, it takes the whole function as a flat input sequence during the training phase. After getting the input, it divides it into multiple slices based on the program slicing method from the existing literature, MVD [5] and SySeVR [6]. Once the input function is sliced, each slice is labeled as vulnerable or non-vulnerable based on the commit analysis of a program. Then, a transformer-based model is fine-tuned with the labeled slices, where the model learns whether a slice is vulnerable or not. During testing, the model generates the slices and their labels using the same way as the training phase. After that, the trained transformer model makes predictions about all the slices of the input function. Finally, it classifies the function as vulnerable if any of that function's slices is predicted as vulnerable; otherwise, it considers the function non-vulnerable. The overall architecture of the proposed model is provided in Figure 1.

2.1 Program Slicing

To perform program slicing, we adopt the technique used in the two existing works, SySeVR and MVD. They first use Joern to parse source code and construct the program dependencies graph (PDG). Then they perform forward and backward slicing from four program points of interest. The program points of interest are-

1. Library/API function call (FC): SySeVR authors published a list of 811 library/API function calls that correspond to the 106 CWE Ids [7]. Any statement with a function call from that list is considered a program point of interest.

2. Array usage (AU): Any statement where the array element is declared or accessed is considered a program point of interest.

3. Pointer usage (PU): Any statement where the pointer variable is declared or accessed is regarded as a program point of interest.

4. Arithmetic expression (AE): Any statement with an arithmetic expression or having '=' is considered a program point of interest.

It is worth mentioning that starting from the program point of interest, they perform backward slicing according to both control and data dependencies but perform forward slicing based on only data dependencies. They do not use control dependencies on forward slicing because consideration of control dependencies on the program point of interest would add many extraneous statements which have nothing to do with vulnerabilities.

2.2 Labeling

To label the slices, we leverage the `diff` files generated from the current and next commits of the input function/program. We label a slice vulnerable if and only if it is extracted from a vulnerable function and at least one of the lines of that slice is removed/altered in the next commit (starting with '-' in `diff` files). If any slice extracted from a vulnerable function does not have any line removed/altered in the next commit, we don't consider that slice. On the other hand, we label a slice non-vulnerable if and only if it is extracted from a non-vulnerable function.

2.3 Transformer Model for Vulnerability Detection

We use the CodeBERT [2] transformer model as the detection model. CodeBERT is a pre-trained programming language model which can be used for many down streaming tasks.

At the very first step, CodeBERT takes a source code as input and tokenizes the input code. During tokenization, it adds two extra tokens; one is the [CLS] token which is added at the beginning of the sequence, and another is the [SEP] or [S] token, which is added at the end of the sequence. It then represents each token as a vector of embed_dim size. If the input text has T tokens, it gets a matrix of size (T $\times$ embed_dim) after converting each token into a vector of size embed_dim. Then the token embedding goes through 12 encoder blocks sequentially. The dimension of the input and output of each encoder block is the same, and the output of one encoder is the input of the next encoder. A specific encoder block is responsible for finding relationships among the input embeddings and encoding them in its output representations. The initial block learns the basic relationship, but the more it goes through the encoder block, the more complex relations it learns.

In each encoder block, the input is passed through a multi-head attention layer. The multi-head attention layer computes the attention multiple times in parallel with different weights and then concatenates these attentions together. The result of each of those parallel computations of attention is called a head. After normalizing the output of the multi-head attention, the result is sent to a feed-forward network. The normalized output is the output of the encoder block.

After getting the output of the final encoder block, the embedding of the [CLS] token is sent to a classification layer (multi-layer perceptron). Finally, the classification layer decides whether the input source code is vulnerable or not.

3 Experimental Setup

3.1 Research Questions

To evaluate our model, we aim to answer the following research questions.

RQ1: How much performance improves when the slices are fed to the transformer model instead of functions?

RQ2: Does our approach solve the root cause's excluding issue?

3.2 Dataset

We construct a new vulnerability dataset from Devign [8] and MSR [9]. In Devign and MSR datasets, functions are labeled as vulnerable or non-vulnerable. For each example, at first, we generate the `diff` information. If the `diff` information is empty for an example, we do not consider that example. To make our dataset balanced, we remove all the non-vulnerable data with less than three slices. Then, we extract several slices from the remaining examples and label them according to our approach. Finally, we build two datasets: (i) function-level and (ii) slice-level dataset.

Level / Split	Train	Test	Validation
Functions	13796	1724	1724
Vulnerable	6927	865	865
Non Vulnerable	6869	859	859
Slices	35143	4425	4394
Vulnerable	16935	2128	2102
Non Vulnerable	18208	2297	2292

Table 1: Function-level and slice-level data distribution over different splits.

For the function level dataset, we randomly select 80% of the functions as the training set, 10% of the programs as the testing set, and the remaining 10% as the validation set. For the slice-level dataset, we consider the slices extracted from the training, testing and validation sets of the function-level dataset as the training, testing and validation sets, respectively. Table 1 shows the details data distribution over the train, test, and validation data splits.

3.3 Hyper parameters

We use a max_length of 400 to train both of the models. We train the models up to 10 epochs with a batch size of 256 and an initial learning rate of 0.005. We save the best model checkpoint that achieves the highest accuracy on the validation set, and use that saved weight during testing.

3.4 Evaluation Metrics

To evaluate the effectiveness of our model, we used the following evaluation metrics:
Accuracy is the fraction of correctly predicted vulnerable/non-vulnerable functions out of all the functions [10] and is calculated as -

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

Here TP, TN, FP, and FN denote true positive, true negative, false positive, and false negative, respectively. Accuracy gives a general overview of how well a model performs. If there is an imbalance in the number of vulnerable and non-vulnerable functions in the dataset, performance can be skewed by the majority class.
Precision indicates the fraction of correctly predicted vulnerable functions out of all the predicted vulnerable functions. [10]. It is computed as -

$$Precision = \frac{TP}{TP + FP}$$

Recall indicates the fraction of correctly predicted vulnerable functions out of all the actual vulnerable functions. The formula is -

$$Recall = \frac{TP}{TP + FN}$$

F1 score is a way to combine both `precision` and `recall` into a single measure by giving equal weights to both of them. It is generally described as the harmonic mean of `precision` and `recall` [5]. It can be formulated as-

$$F1\ score = 2 * \frac{Precision * Recall}{Precision + Recall}$$

F1 score is preferred over Accuracy when there is an imbalance in the classes, but we crafted our dataset to eliminate the imbalance of vulnerable/non-vulnerable classes. So, we utilize both Accuracy and F1 Score to evaluate our model.

3.5 Experimental Design of Research Questions

RQ1. For RQ1, we train the CodeBERT [2] model using our two datasets: slice-level and function-level. We use our proposed approach to train and test the model when the slice-level dataset is used. We feed the slices of an input function to the model, get the prediction of these slices and classify the function as vulnerable if at least one of its slices is vulnerable. For the function-level dataset, we directly feed the functions as the input to the transformer model and the output of the transformer model is the prediction of the input function.
RQ2. For RQ2, we search for examples whose ground truths are vulnerable and meet the following two criteria:

1. The function is predicted as non-vulnerable with the function-level approach, but the root cause is discarded because of the max_length hyperparameter.

2. The function is predicted as vulnerable with the slice-level approach; this time, the root cause is included in the input slices.

If these kinds of examples are found, then we can make a hypothesis that our proposed approach works to resolve the root cause's excluding issue.

4 Experimental Result

RQ1: How much performance improves when the slices are fed to the transformer model instead of functions?
Table 2 shows the performance of the CodeBERT model for the two approaches. Overall, our proposed model achieves better performance than the function-level approach in all four metrics. It gets 0.731 Accuracy, 0.765 Precision, 0.67 Recall, and 0.715 F1 scores. We can see that the recall score improves from 0.657 to 0.670, which means that when our approach is used, the number of false negatives decreases and the number of true positives increases. To get more information, we did some analysis on the predictions and found that 9 of the vulnerable examples which are predicted as non-vulnerable in the function-level approach are predicted as vulnerable in our approach. This implies that our approach performs better than the function-level approach.

Dataset Granularity	Accuracy	F1 Score	Precision	Recall
Function	0.724	0.705	0.761	0.657
Slice	**0.731**	**0.715**	**0.765**	**0.670**

Table 2: Results

RQ2: Does our approach solve the root cause's excluding issue?
In RQ1, we found nine vulnerable(ground truth) examples whose predictions are non-vulnerable in the function-level approach and vulnerable using our approach. We inspected these examples to answer RQ2. We found four examples that are big in size, and the root cause is not included within the max_length tokens in the function-level approach. On the other hand, our approach generates slices from these functions where the slices are very smaller than the actual functions, and the root cause is included within the max_length tokens. We publish these four examples at shorturl.at/twEH7.

```c
static int sp5x_decode_frame(AVCodecContext *avctx,
                             void *data, int *data_size,
                             AVPacket *avpkt)
{
    // ...30 lines
    AV_WB16(recoded+j+7, avctx->coded_width);
    // ...20 lines
    avctx->flags &= ~CODEC_FLAG_EMU_EDGE; // 56th line
    av_init_packet(&avpkt_recoded);
    avpkt_recoded.data = recoded;
    avpkt_recoded.size = j;
    i = ff_mjpeg_decode_frame(avctx, data, data_size, &avpkt_recoded);
    av_free(recoded);
    return i;
}
```

Listing 1: A vulnerable function which is predicted as vulnerable in function-level approach and predicted as non-vulnerable in our approach. The highlighted line is the root cause of the vulnerability.

In listing 1, we show a vulnerable example where the root cause is at the 56th line. The function level approach predicts this example as non-vulnerable. In listing 2, we show a slice generated from the function of listing 1. We can see that the slice is smaller than the actual function, and the root cause is also included in the slice. Our proposed slice-level approach predicts this slice as vulnerable. To sum up, our approach solves the root cause's excluding issue to some extent.

5 Related Work

Several transformer-based approaches have been proposed for vulnerability detection in recent years. CodeBERT [2] and VulBERTa [3] are two transformer-based models that predict software vulnerability at function-level granularity. They use the same architecture, RoBERTa [11], but use different tokenization methods. LineVul [4]

```
1   static int sp5x_decode_frame(AVCodecContext *avctx,
2                              void *data, int *data_size,
3                              AVPacket *avpkt)
4   const uint8_t *buf = avpkt->data;
5   int buf_size = avpkt->size;
6   if (!avctx->width || !avctx->height)
7   buf_ptr = buf;
8   AV_WB16(recoded+j+5, avctx->coded_height);
9   AV_WB16(recoded+j+7, avctx->coded_width);
10  if(avctx->codec_id==CODEC_ID_AMV)
11  avctx->flags &= ~CODEC_FLAG_EMU_EDGE; // 56th line
12  i = ff_mjpeg_decode_frame(avctx, data, data_size, &avpkt_recoded);
13  return i;
```

Listing 2: A slice produced from the listing 1. The root cause is highlighted at line 11.

works on line-level vulnerability detection, which depends on the function level transformer, CodeBERT, and uses attention scores to detect vulnerable statements. In these transformer-based approaches, a fixed number (max_length) of tokens are passed to the models, and the remaining are discarded. Thus, for functions longer than the max_length, the causes of vulnerability may remain unused in the training.

VulDeePecker [12] and SySeVR [6] are approaches that identify vulnerabilities at the slice level. They use RNN (e.g., LSTM and BGRU) to train the detection models. MVD [5] is another slice-level approach that detects memory-related vulnerabilities. It uses flow-sensitive graph neural networks (FS-GNN) for model training.

To the best of our knowledge, our work offers the first approach that predicts function-level vulnerability using the slice-level predictions of the transformer-based model.

6 Conclusion and Future Work

In this work, we propose a transformer-based model to detect software source code vulnerability at the function level using program slices. Our approach generates slices using the techniques defined in SySeVR and MVD and uses these slices to train and test the model. As the transformer-based model discards the statements contributing to the vulnerabilities after max_length, our approach reduces the possibility of excluding the root cause to minimal for long input functions.

In the future, We only use the CodeBERT transformer model for our study. In future, we plan to train the model with different transformer-based models. We also plan to train our model with other datasets. In addition, we want to use our approach in inter-procedural programs, and for that, we aim to extract program slices with additional information, i.e., call relations.

References

[1] A. Anwar, A. Khormali, J. Choi, H. Alasmary, S. Choi, S. Salem, D. Nyang, and D. Mohaisen, "Measuring the cost of software vulnerabilities," *EAI Endorsed Transactions on Security and Safety*, vol. 7, no. 23, 2020.

[2] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[3] H. Hanif and S. Maffeis, "Vulberta: Simplified source code pre-training for vulnerability detection," *arXiv preprint arXiv:2205.12424*, 2022.

[4] M. Fu and C. Tantithamthavorn, "Linevul: A transformer-based line-level vulnerability prediction," 2022.

[5] S. Cao, X. Sun, L. Bo, R. Wu, B. Li, and C. Tao, "Mvd: Memory-related vulnerability detection based on flow-sensitive graph neural networks," *arXiv preprint arXiv:2203.02660*, 2022.

[6] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "Sysevr: A framework for using deep learning to detect software vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, 2021.

[7] "Common Weakness Enumeration," https://cwe.mitre.org/, 2008, [Online; accessed 12-Dec-2022].

[8] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *Advances in neural information processing systems*, vol. 32, 2019.

[9] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "Ac/c++ code vulnerability dataset with code changes and cve summaries," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 508–512.

[10] "Accuracy, precision, recall f1 score: Interpretation of performance measures," https://blog.exsilio.com/all/accuracy-precision-recall-f1-score-interpretation-of-performance-measures/, [Online; accessed 12-Dec-2022].

[11] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," 2019. [Online]. Available: https://arxiv.org/abs/1907.11692

[12] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, "Vuldeepecker: A deep learning-based system for multiclass vulnerability detection," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 5, pp. 2224–2236, 2019.