LineVul: A Transformer-based Line-Level Vulnerability Prediction

Michael Fu
michael.fu@monash.edu
Monash University
Australia

Chakkrit Tantithamthavorn*
chakkrit@monash.edu
Monash University
Australia

ABSTRACT

Software vulnerabilities are prevalent in software systems, causing a variety of problems including deadlock, information loss, or system failures. Thus, early predictions of software vulnerabilities are critically important in safety-critical software systems. Various ML/DL-based approaches have been proposed to predict vulnerabilities at the file/function/method level. Recently, IVDetect (a graph-based neural network) is proposed to predict vulnerabilities at the function level. Yet, the IVDetect approach is still inaccurate and coarse-grained. In this paper, we propose LineVul, a Transformer-based line-level vulnerability prediction approach in order to address several limitations of the state-of-the-art IVDetect approach. Through an empirical evaluation of a large-scale real-world dataset with 188k+ C/C++ functions, we show that LineVul achieves (1) 160%-379% higher F1-measure for function-level predictions; (2) 12%-25% higher Top-10 Accuracy for line-level predictions; and (3) 29%-53% less Effort@20%Recall than the baseline approaches, highlighting the significant advancement of LineVul towards more accurate and more cost-effective line-level vulnerability predictions. Our additional analysis also shows that our LineVul is also very accurate (75%-100%) for predicting vulnerable functions affected by the Top-25 most dangerous CWEs, highlighting the potential impact of our LineVul in real-world usage scenarios.

ACM Reference Format:
Michael Fu and Chakkrit Tantithamthavorn. 2022. LineVul: A Transformer-based Line-Level Vulnerability Prediction. In *19th International Conference on Mining Software Repositories (MSR '22), May 23–24, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3524842.3528452

1 INTRODUCTION

Software vulnerabilities are weaknesses in an information system, security procedures, internal controls, or implementations that could be exploited or triggered by a threat source [26]. Those unresolved weaknesses associated with software systems may result in extreme security or privacy risks. For instance, in 2021, a criminal group leveraged the ProxyLogon flaw [8] to access highly confidential data of PC-maker Acer and issued an opening ransom demand of $50 million USD [7]. The vulnerability was on Microsoft Exchange Server, that allowed an attacker to bypass the authentication and impersonating. Therefore, an unauthenticated attacker could execute arbitrary commands on the server. Cybersecurity Ventures expects global cybercrime costs to reach $10.5 trillion USD by 2025, up from $3 trillion USD in 2015 [4]. As cyberattacks become the main contributing factors to revenue loss of some businesses [10], ensuring the safety of software systems becomes one of the critical challenges of private and public sectors.

To mitigate this challenge, program analysis (PA) tools [1, 2, 5, 9] have been introduced to analyze source code using predefined vulnerability patterns. For instance, Gupta *et al.* [19] found that there are static analysis approaches that were used to detect SQL injection and cross-site scripting vulnerabilities. Both PA-based and ML/DL-based approaches fall short of the capability to detect fine-grained vulnerabilities." – PA tools, including ones cited in the paper, highlight the specific lines in code that contain the detected vulnerabilities.

On the other hand, Machine Learning (ML) / Deep Learning (DL) approaches have been proposed. Specifically, these ML/DL-based approaches [12, 32, 33, 43, 65] first generate a representation of source code in order to learn vulnerability patterns. Finally, such approaches will learn the relationship between the representation of source code and the ground-truth (i.e., whether a given piece of code is vulnerable). Despite the advantages of dynamically learning the vulnerability patterns without manual predefined vulnerability patterns, previous ML/DL-based approaches still focus on coarse-grained vulnerability prediction where models only point out vulnerabilities at the file level or the function level—which is still coarse-grained.

Recently, Li *et al.* [30] proposed an IVDetect approach to address the need of fine-grained vulnerability prediction. IVDetect leverages a FA-GCN (i.e., Feature-attention Graph Convolution Network) approach to predict function-level vulnerabilities and a GNNExplainer to locate the fine-grained location of vulnerabilities. Li *et al.* [30] found that the IVDetect approach achieves a F-measure of 0.35, which outperforms the state-of-the-art approaches. However, IVDetect has the following three limitations.

- **First, the training process of IVDetect is limited to project-specific dataset.** Due to the limited amount of training data used by IVDetect, the language models may not be able to capture the most accurate relationship between tokens and their surrounding tokens. Thus, the vectors representation of source code by IVDetect are still suboptimal.
- **Second, the RNN-based architecture of the IVDetect approach is still not effective to capture the meaningful long-term dependencies and semantics of source code.** IVDetect relies on RNN-based models to generate vector representations to be used by its graph model during the

*Corresponding Author.

MSR '22, May 23–24, 2022, Pittsburgh, PA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9303-4/22/05…$15.00
https://doi.org/10.1145/3524842.3528452

prediction step. However, RNN-based models often have difficulties in learning the long sequence of source code. Such the limitation could make the generated vector representations less meaningful, resulting in inaccurate predictions.

- **Third, the sub-graph interpretation of IVDetect is still coarse-grained.** IVDetect leveraged a GNNExplainer to identify which sub-graph contributed the most to the predictions. Although such sub-graph interpretation can help developers narrow down to locate vulnerable lines, such sub-graphs still contain many lines of code. Thus, security analysts still need to manually locate which lines of these sub-graphs are actually vulnerable.

In this paper, we propose LineVul, a Transformer-based fine-grained vulnerability prediction approach to address the three important limitations of IVDetect. First, instead of using RNN-based models to generate representation of code, we leverage a BERT architecture [15] with self-attention layers that are capable of capturing long term dependencies within a long sequence using dot-product operations. Second, instead of using project-specific training data, we leverage a CodeBERT pre-trained language model to generate vector representation of source code. Third, instead of using a GNNExplainer to identify sub-graphs that contribute to the predictions, we leverage the attention mechanism of the BERT architecture to locate vulnerable lines, which is finer-grained than the IVDetect approach. Finally, we conduct an experiment to compare our LineVul approach with seven baseline approaches (i.e., IVDetect [30], Reveal [12], SySeVR [32], Devign [65], Russell *et al.* [43], VulDeePecker [33], and BoW+RF), and evaluated on both coarse-grained (i.e., function-level) and fine-grained (i.e., line-level) vulnerability prediction scenarios. Through an extensive evaluation of our approach on 188k+ C functions including 91 different types of CWEs (Common Weakness Enumeration), we answer the following three research questions:

(RQ1) How accurate is our LineVul for function-level vulnerability predictions?
Results. Our LineVul achieves an F-measure of 0.91, which is 160%-379% better than the state-of-the-art approaches with a median improvement of 250%. Similarly, our LineVul achieves a Precision of 0.97 and a Recall of 0.86, which outperform the baseline approaches by 322% and 19%, respectively.

(RQ2) How accurate is our LineVul for line-level vulnerability localization?
Results. Our LineVul achieves a Top-10 Accuracy of 0.65, which is 12%-25% more accurate than the other baseline approaches. In addition, LineVul achieves the lowest median IFA of 1, while the baseline approaches achieve a median IFA of 3-4.

(RQ3) What is the cost-effectiveness of our LineVul for line-level vulnerability localization?
Results. Our LineVul achieves the lowest Effort@20%Recall of 0.75, which is 29%-53% less than other baseline approaches. In addition, LineVul achieves the highest Recall@1%LOC of 0.24, which is 26%-85% higher than other baseline approaches.

These results lead us to conclude that LineVul is more accurate, more cost-effective, and more fine-grained than existing vulnerability prediction approaches. Thus, we expect that our LineVul may help security analysts to find vulnerable lines in a cost-effective manner. In addition, we recommend the attention mechanism be used in future research to improve the explainability of the Transformer-based models, since this paper has shown the substantial benefits of using the attention mechanism for line-level vulnerability localization, which outperforms other model-agnostic techniques (e.g., DeepLift, LIG, and SHAP).

Novelty & Contributions. To the best of our knowledge, the main contribution of this paper is as follows: (1) our LineVul, a Transformer-based line-level vulnerability prediction approach, which address various limitations of existing vulnerability prediction approaches; and (2) the experimental results confirm that our LineVul is more accurate, more cost-effective, and more fine-grained than existing vulnerability prediction approaches.

Open Science. To support the open science community, we publish the studied dataset, scripts (i.e., data processing, model training, and model evaluation), and experimental results in GitHub (https://github.com/awsm-research/LineVul).

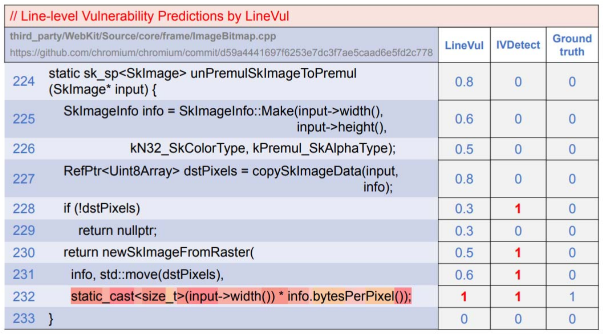
Paper Organization. Section 2 discusses the IVDetect approach and its limitations. Section 3 presents our LineVul approach. Section 4 presents the motivation of our three research questions, our studied datasets, and our experimental setup, while Section 5 presents the experimental results. Section 6 presents the ablation study of our LineVul. Section 8 discloses the threats to validity. Section 9 draws the conclusions.

2 BACKGROUND

A software vulnerability is a defect or a weakness in software implementation due to the way the software is designed or the way the software is coded. In computer security, such software vulnerabilities could lead to inner system crash or allow an attacker to gain control of a system by crossing privilege boundaries within a computer system. Thus, two types of automated vulnerability prediction approaches, Program Analysis (PA)-based and ML/DL-based, have been proposed to early predict software vulnerabilities.

PA-based techniques use pre-defined patterns to detect software vulnerabilities. Hence, PA-based methods only focus on specific types of vulnerabilities. For instance, the FlawFinder [5] supports common weakness enumeration (CWE), which takes C/C++ program as input and generates a list of vulnerabilities sorted by risk level. RATS [9] is another static program analysis tool, which can detect the vulnerabilities such as buffer overflows and TOCTOU (Time of Check, Time of Use) race conditions. Cppcheck [2] is a static analysis tool for C/C++ code using unique code analysis and focuses on detecting undefined behaviour and dangerous coding constructs. Checkmarx [1] provides automatic scans of uncompiled source code that can be integrated into DevOps, enabling developers to identify security vulnerabilities during development. However, such the pre-defined patterns for PA-based techniques need to be manually crafted by security experts, which are time-consuming.

ML/DL-based approaches leverage Machine Learning (ML) and Deep Learning (DL) techniques to automatically learn vulnerability patterns to detect software vulnerabilities. Specifically, code programs are transformed into vector representation for the model to learn the implicit patterns of vulnerabilities from prior vulnerable programs. Recently, several DL models have been applied to vulnerability prediction tasks. For instance, VulDeePecker [33] leverages symbolic representations on program slices, Devign [65] uses graph embedding on code property graphs (i.e., AST, CFG, DFG), SySeVR [32] relies on semantic information induced by data dependency, Reveal [12] adopted graph embedding with triplet loss function for representation learning, Russell *et al.* [43] leverages CNNs and RNNs to extract representation. Despite these DL models are able to generate better representation, they still focus on coarse-grained vulnerability prediction that provides vulnerability prediction at the file level or the function level.



// Line-level Vulnerability Predictions by LineVul

third_party/WebKit/Source/core/frame/ImageBitmap.cpp https://github.com/chromium/chromium/commit/d59a4441697f6253e7dc3f7ae5caad6e5fd2c778		LineVul	IVDetect	Ground truth
224	`static sk_sp<SkImage> unPremulSkImageToPremul (SkImage* input) {`	0.8	0	0
225	`SkImageInfo info = SkImageInfo::Make(input->width(), input->height(),`	0.6	0	0
226	`kN32_SkColorType, kPremul_SkAlphaType);`	0.5	0	0
227	`RefPtr<Uint8Array> dstPixels = copySkImageData(input, info);`	0.8	0	0
228	`if (!dstPixels)`	0.3	1	0
229	`return nullptr;`	0.3	0	0
230	`return newSkImageFromRaster(`	0.5	1	0
231	`info, std::move(dstPixels),`	0.6	1	0
232	`static_cast<size_t>(input->width()) * info.bytesPerPixel());`	1	1	1
233	`}`	0	0	0

Figure 1: A motivating example of our LineVul vs IVDetect.

2.1 IVDetect: A state-of-the-art fine-grained vulnerability prediction & Limitations

Both PA-based and ML/DL-based approaches fall short of the capability to detect fine-grained vulnerabilities. Therefore, developers would still need to inspect many lines of code to look for and fix the vulnerabilities in their code. Recently, Li *et al.* [30] proposed IVDetect—a fine-grained graph-based vulnerability prediction approach, which consists of three steps:

Step 1: Code Representation Learning. IVDetect leverages the GloVe word embedding (Global Vectors for Word Representation) to capture semantic similarity among tokens and a GRU model to summarize the sequence of vectors into one feature vector. IVDetect generates four feature vectors from the given code statement: 1) a sequence of sub-tokens to capture lexical information, 2) variable names and types to be used as node information in graph model, 3) data dependency context, and 4) control dependency context. In addition, a Tree-LSTM is used to generate the representation for AST trees. After obtaining all of the five feature vectors (i.e., $F_1, ..., F_5$), IVDetect uses a Bi-GRU and an attention layer to learn the weight vector W_i for each feature vector F_i. Finally, each feature vector is multiplied by the computed weight vector: $F_i' = W_i F_i$.

Step 2: Vulnerability Prediction with FA-GCN. Given an input method m, IVDetect first processes m into a program dependency graph (PDG) consisting of many statements, for each statement, the five feature vectors will be generated through the process discussed in **Step 1**. FA-GCN performs sliding a small window along all the nodes (statements) of the PDG and leverages a joint layer to link all generated feature vectors into a feature matrix $\mathcal{F}_m$ where each row corresponds to a small window in PDG. Then, the symmetric normalized Laplacian matrix [29] $\mathcal{L}_m$ is calculated and combined with feature matrix $\mathcal{F}_m$ through the convolution to generate the representation matrix $\mathcal{M}_m$ for the method m. Finally, FA-GCN uses a spatial pyramid pooling layer for normalization purpose followed by a fully connected layer to transform matrix $\mathcal{M}_m$ into vector V_m and performs classification using two hidden layers and a softmax function to produce a prediction score for m. The scores are used as vulnerability scores to rank the functions.

Step 3: Fine-grained Vulnerability Prediction by GNNExplainer. IVDetect leverages GNNExplainer [64] with a masking technique to explain which sub-graphs contribute the most to the vulnerability predictions from the FA-GCN model. The goal of GNNExplainer is to find out a sub-graph $\mathcal{G}_m$ from the whole PDG G_m of the method m that minimizes the difference in the prediction scores between using the entire graph G_m and using the minimal graph $\mathcal{G}_m$. To do so, GNNExplainer learns the set of edge-mask EM that derives the minimal graph $\mathcal{G}_m$. As an EM is applied, GNNExplainer checks if the FA-GCN model produces the same result (i.e., predicted as vulnerable). If yes, the edge in the edge-mask is not important and not included in $\mathcal{G}_m$. Otherwise, the edge is important and included in $\mathcal{G}_m$. IVDetect then utilizes the best sub-graph $\mathcal{G}_m$ learnt from GNNExplainer as an interpretation for the given function to detect fine-grained vulnerabilities. However, there exist the following limitations.

Limitation ①: The training process of IVDetect is limited to project-specific dataset. The quality of vector representation heavily relies on the language models of code being used. For the IVDetect approach, Li *et al.* [30] leveraged a GloVE (see Step 1 of IVDetect), which is an unsupervised learning algorithm for obtaining vector representations of words. However, their Glove language models are trained on the project-specific dataset without pre-training on large code base, which may not be able to generate the most meaningful code representation. Thus, the suboptimal vector representation of source code by IVDetect may lead to inaccurate predictions.

Limitation ②: The RNN-based architecture of the IVDetect approach is still not effective to capture the meaningful long-term dependencies and semantics of source code. IVDetect relies on RNN-based architectures to generate code representation in Step 1, which will encounter problems when processing a long sequence. RNN-based models process a sequence token by token, where the models consider a context vector and a hidden vector of the last token when processing each token. The hidden vector is used to capture short-term dependencies between tokens while the context vector is used for long-term dependencies. However, the context vector has problems capturing adequate long-term dependencies given a long sequence (e.g., a sequence of 500 tokens) due to its limited memory. Thus, this limitation could make the generated feature representations less meaningful, which further negatively impact the accuracy of the vulnerability prediction models.

Limitation ③: The sub-graph interpretation of IVDetect is still coarse-grained. Third, in Step 3, IVDetect leverages GNNExplainer to generate PDG sub-graph interpretations as fine-grained vulnerability predictions. Such sub-graph interpretations could consist of multiple lines of code, and they are not fine enough to effectively reduce the manual code inspection effort. For instance in Figure 1, the vulnerable function *unPremulSkImageToPremul* contains a vulnerable line (i.e., the ninth line) in which the variable type was wrongly defined and further caused a vulnerable type of CWE-787 [3]. IVDetect generated a PDG sub-graph using IVDetect, which pointed out the fifth, seventh, eighth, and ninth line as a vulnerable pattern. On the other hand, our LineVul generated a line-level interpretation, which directly pinpoint the actual vulnerable line.

3 LINEVUL: A LINE-LEVEL VULNERABILITY PREDICTION APPROACH

In this section, we present the design rationale and the architecture of our LineVul approach.

Design Rationale. To address the three key limitations of IVDetect, we propose the architecture of our LineVul, a Transformer-based line-level vulnerability prediction approach. First, instead of using RNN-based models to generate representation of code, we leverage BERT architecture [15] with self-attention layers that are capable of capturing long term dependencies within a long sequence using dot-product operations. Second, instead of using project-specific training data, we leverage a CodeBERT pre-trained language model to generate vector representation of source code. The pre-trained CodeBERT language model was pre-trained on 20GB of code corpus (i.e., CodeSearchNet) using a Robustly Optimized BERT pre-training approach [34]. Therefore, our approach is able to capture more lexical and logical semantics for the given code input and generate a more meaningful vector representation. Third, instead of using a GNNExplainer to identify sub-graphs that contribute to the predictions, we leverage the attention mechanism of the BERT architecture to locate vulnerable lines, which is finer-grained than the IVDetect approach.

We design our LineVul approach as a two-step approach: to predict vulnerable functions and to locate vulnerable lines. Figure 2 presents an overview architecture of our LineVul approach.

3.1 Function-level Vulnerability Prediction

The function-level vulnerability prediction consists of 2 main steps:

① **BPE Subword Tokenization.** In Step ①, we leverage the Byte Pair Encoding (BPE) approach [44] to build our tokenizer with two main steps. (1a) generating merge operations to determine how a word should be split, and (1b) applying merge operations based on the subword vocabularies. Specifically, BPE will split all words into sequences of characters and identify the most frequent symbol pair (e.g., the pair of two consecutive characters) that should be merged into a new symbol. BPE is an algorithm that will split rare words into meaningful subwords and preserve the common words (i.e., will not split the common words into smaller subwords) at the same time. For instance in Figure 1, the function name, *unPremulSkImageToPremul*, will be split into a list of subwords, i.e., ["un", "Prem", "ul", "Sk", "Image", "To", "Prem", "ul"]. The common word "Image" was preserved and other rare words were split. The use of BPE subword tokenization will help reduce the vocabulary size when tokenizing various of function names because it will split rare function name into multiple subcomponents instead of adding the full function name into dictionary directly. In this paper, we apply BPE approach on the CodeSearchNet [20] corpus to produce a subword tokenizer that is suitable for a pre-trained language model of source code corpus.

② **LineVul Model Building.** In Step ②, we build a LineVul model based on the BERT architecture and leverage the initial weights pre-trained by Feng *et al.* [17]. In Step (2a), LineVul performs a word & positional encoding for the subword-tokenized function in order to generate an embedding vector of each word and its position in the function. Then, in Step (2b), the vector is fed into the BERT architecture, which is a stack of 12 Transformer encoder blocks. Each encoder consists of a multi-head self-attention layer and a fully connected feed-forward neural network. Finally, in Step (2c), the output vector is fed into a single linear layer in order to perform binary classification for the given function. We describe each step below.

(2a) **Word & Positional Encoding.** Source code consists of multiple tokens where the meaning of each token heavily relies on the context (i.e., surrounding tokens) and its position of each token in a function. Therefore, it is important to capture the code context and its position within the function, especially, for function-level vulnerability predictions. The purpose of this step is to generate encoding vectors that capture the semantic meaning of code tokens and their positions in the input sequence. To do so, for each subword-tokenized token, we generate two vectors: (1) a word encoding vector to represent the meaningful relationship between a given code token and the other code tokens and (2) the positional encoding vector to represent the position of a given token in the input sequence. The token encoding vectors are generated according to the word embedding matrix $W_{te}^{|V| \times d}$ where $|V|$ is the vocabulary size and d is an embedding size. The positional encoding vectors are generated according to the positional embedding matrix $W_{pe}^{c \times d}$ where c is the context size and d is the embedding size. Finally, both the word encoding vector and the positional encoding vector are concatenated in order to produce input vectors of the Transformer encoder blocks.

(2b) **A Stack of 12 Transformer Encoders with Bidirectional Self-attention.** In this Step, the encoding vectors are fed into a stack of 12 encoder-only Transformer blocks (i.e., the BERT architecture [15]). Each encoder block consists of two components, i.e., a bidirectional multi-head self-attention [15] layer and a fully-connected feed-forward neural network. Below, we briefly describe the multi-head self-attention and the feed-forward neural network.

The multi-head self-attention layer is used to compute an attention weight of each code token, that produces an attention vector. The use of bidirectional self-attention allows every token to attend context to its left and right. The generated attention weights are used to indicate which code statements the Transformer model should pay attention to. Generally, the self-attention mechanism is used to obtain global dependencies where the attention weights represent how each code token in the sequence is influenced by all the other words in the sequence, allowing our LineVul approach to

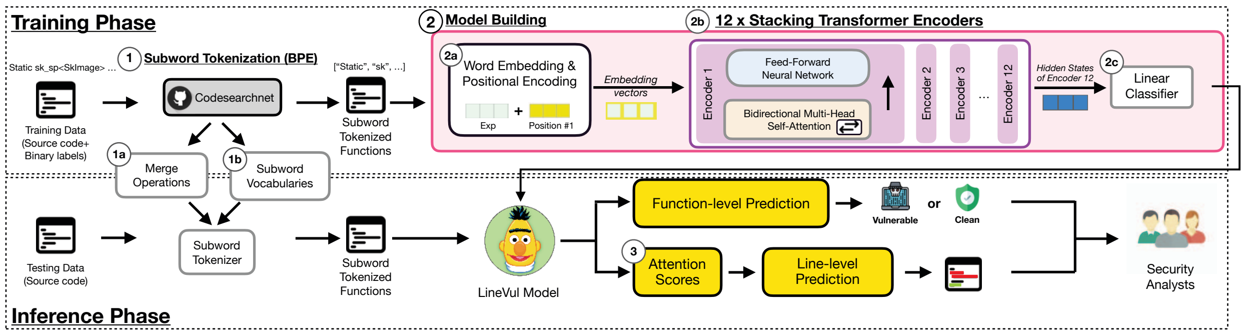

Training Phase
2 Model Building
2b 12 x Stacking Transformer Encoders
Static sk_sp<SkImage> ...
1 Subword Tokenization (BPE)
["Static", "sk", ...]
Codesearchnet
Training Data (Source code+ Binary labels)
Subword Tokenized Functions
2a Word Embedding & Positional Encoding
Exp
Position #1
Embedding vectors
Encoder 1
Feed-Forward Neural Network
Bidirectional Multi-Head Self-Attention
Encoder 2
Encoder 3
Encoder 12
Hidden States of Encoder 12
2c Linear Classifier
1a Merge Operations
1b Subword Vocabularies
Function-level Prediction
Vulnerable
or
Clean
Testing Data (Source code)
Subword Tokenizer
Subword Tokenized Functions
LineVul Model
3 Attention Scores
Line-level Prediction
Security Analysts
Inference Phase

Figure 2: An overview architecture of our LineVul.

capture dependencies between every code token to generate more meaningful representation.

The self-attention mechanism [59] employs a concept of information retrieval, that computes the relevant scores of each code token using the dot product operation where each token interacts with other token once. The self-attention mechanism relies on three main components, Query (Q), Key (K), and Value (V). The Query is a representation of the current code token used to score against all the other tokens based on their keys stored in the Key vectors. The attention scores of each token are obtained by taking the dot product between Query vectors and Key vectors. The attention scores is then normalized to probabilities using the Softmax function in order to get the attention weights. Finally, the Value vectors can be updated by taking dot product between the Value vectors and the attention weight vectors. The self-attention used in our LineVul is a scaled dot-product self-attention, in which the attention scores are divide by $\sqrt{d_k}$. The self-attention mechanism we adopted can be summarized by the following equation:

$$Attention(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V.$$

To capture richer semantic meanings of the input sequence, we used multi-head mechanism to realize the self-attention, which allows the model to jointly attend to information from different code representation subspaces at different positions. For d-dimension Q, K, and V, we split those vectors into h heads where each head has $\frac{d}{h}$-dimension. After all of the self-attention operation, each head will then be concatenated back again to feed into a fully-connected feed-forward neural network including two linear transformations with a ReLU activation in between. The multi-head mechanism can be summarized by the following equation:

$$MultiHead(Q, K, V) = Concat(\text{head}_1, ..., \text{head}_h)W^O,$$

where $head_i$ = Attention(QW_i^Q, KW_i^K, VW_i^V) and W^O is used to linearly project to the expected dimension after concatenation.

(2c) **Single Linear Layer.** Multiple linear transformations with non-linear activation functions (i.e., ReLU) are built in the stacking Transformer encoder blocks, therefore, the representation output by the final encoder is meaningful. We only need a single linear layer to map the code representation into binary label, i.e., $\hat{y} = W^T X + b$.

3.2 Line-level Vulnerability Localization

Given a function predicted as vulnerable by LineVul, we perform a line-level vulnerability localization by leveraging the self-attention mechanism inside the Transformer architecture to locate the vulnerable lines. The intuition is that tokens that are most contributed to the predictions are likely to be vulnerable tokens.

For each subword token in the function, in Step (3), we summarize the self-attention scores from each of the 12 Transformer encoder blocks. After obtaining the attention subword-token scores, we then integrate those scores into line scores. We split a whole function into many lists of tokens (each list of tokens represents a line) by the Newline control character (i.e., \n). Finally, for each list of token scores, we summarize it into one attention line score and rank line scores in a descending order.

4 EXPERIMENTAL DESIGN

In this section, we present the motivation of our three research questions, our studied dataset, and our experimental setup.

4.1 Research Questions

To evaluate our LineVul approach, we formulate the following three research questions.

(RQ1) How accurate is our LineVul for function-level vulnerability predictions? Recently, Li *et al.*[30] proposed IVDetect, a state-of-the-art fine-grained vulnerability predictions approach. However, as mentioned in Section 2.1, IVDetect has three key limitations, leading to inaccurate and coarse-grained predictions. Therefore, we propose our LineVul approach to address these challenges. Thus, we investigate if the accuracy of our LineVul outperforms the state-of-the-art function-level vulnerability prediction approaches.

(RQ2) How accurate is our LineVul for line-level vulnerability localization? Line-level vulnerability prediction is needed to help developers identify the fine-grained locations of vulnerable lines, instead of wasting their time inspecting non-vulnerable lines. Although IVDetect can identify the sub-graphs of a vulnerable function, such sub-graphs still consist of many lines of code that security analysts need to inspect, which is still coarse-grained.

Thus, we investigate the accuracy of our LineVul for line-level vulnerability predictions.

(RQ3) What is the cost-effectiveness of our LineVul for line-level vulnerability localization? One of the primary objectives of vulnerability prediction is to help security analysts locate vulnerable lines in a cost-effective manner by uncovering the maximum number of vulnerabilities with the least amount of effort. Thus, the amount of effort requires to inspect source code becomes a critical concern when security analysts decide to deploy advanced approaches in practice. Thus, we investigate the cost-effectiveness of our LineVul for line-level vulnerability predictions.

4.2 Studied Dataset

We use the benchmark dataset provided by Fan *et al.* [16] due to the following reasons. First is to establish a fair comparison with the IVDetect approach. Seconds is to evaluate our line-level vulnerability approach, since Fan *et al.* [16]'s dataset is the only vulnerability dataset that provides line-level ground-truths (i.e., which lines in a function are vulnerable). On the other hand, other existing vulnerability datasets (i.e., Devign [65], Reveal [12]) only provide the ground-truths at the function level, but not the line level—which are not suitable for our scope. The Fan *et al.*'s dataset is one of the largest vulnerability datasets that includes line-level ground-truths. The dataset is collected from 348 open-source Github projects, which includes 91 different CWEs from 2002 to 2019, 188,636 C/C++ functions with a ratio of vulnerability functions of 5.7% (i.e., 10,900 vulnerable functions), and 5,060,449 LOC with a ratio of vulnerable lines of 0.88% (i.e., 44,603 vulnerable lines). Among the 10,900 vulnerable functions, the ratio of vulnerable lines varies from 2.5% (1st quantile) - 20% (3rd quantile) with a median of 7%.

4.3 Experimental Setup

Data Splitting. Similar to Li *et al.* [30], we use the same data splitting approach, i.e., the whole dataset is split into 80% of training data, 10% of validation data, and 10% of testing data.

Function-level Model Implementation. To implement our LineVul approach for the function-level vulnerability prediction, we mainly use two Python libraries, i.e., Transformers [62] and Pytorch [13]. The Transformers library provides API access to the transformer-based model architectures and the pre-trained weight, while the PyTorch library supports the computation during the training process (e.g., back-propagation and parameter optimization). We download the CodeBERT tokenizer and CodeBERT model pre-trained by Feng *et al.* [17]. We use our training dataset to fine-tune the pre-trained model to get suitable weights for our vulnerability prediction task. The model was fine-tuned on an NVIDIA RTX 3090 graphic card and the training time was around 7 hours and 10 minutes. As shown in Equation 1, the Cross Entropy Loss was used to update the model and optimize between the function-level predicted values and the ground-truth labels, where x is the number of classes, p is the ground truth probability distribution (one-hot), and q is the predicted probability distribution. To retrieve the best fine-tuned weight, we used the validation set to monitor the training process by epoch, and the best model was selected based on the optimal F1-score against the validation set (not the testing set).

$$H(p,q) = -\sum_{x} p(x)\ log_q(x) \tag{1}$$

Line-level Model Implementation. To implement our LineVul approach for the line-level vulnerability prediction, we use the self-attention matrix returned by the fine-tuned model during inference. We summarize the dot product attention score for each token, hence, every token has one attention score. We integrate tokens into lines and summarized the score of each token to get the line scores. The line scores are then used to prioritize the lines where a higher line score represents that the line is likely to be a vulnerable line.

Hyper-Parameter Settings for Fine-tuning. For the model architecture of our LineVul approach, we use the default setting of CodeBERT, i.e., 12 Transformer Encoder blocks, 768 hidden size, and 12 attention heads. We follow the same fine-tuning strategy provided by Feng *et al.* [17]. During training, the learning rate is set to 2e-5 with a linear schedule where the learning rate decays linearly throughout the training process. We use backpropagation with AdamW optimizer [35] which is widely adopted to fine-tune Transformer-based models to update the model and minimize the loss function.

5 EXPERIMENTAL RESULTS

In this section, we present the experiment results with respect to our three research questions.

(RQ1) How accurate is our LineVul for function-level vulnerability predictions?

Approach. To answer this RQ, we focus on the function-level vulnerability predictions and compare our LineVul with the other seven baseline models described as follows:

1. IVDetect [30] leverages Feature-Attention Graph Convolutional Network (GCN) for vulnerability predictions using five types of feature representation (i.e., sequence of sub-tokens, AST sub-tree, variable names and types, data dependency context, and control dependency context) from the source code;
2. ReVeal [12] leverages a Gated Graph Neural Network (GGNN) in order to learn the graph properties of source code;
3. Devign [65] leverages a Gated Graph Neural Network (GGNN) to automatically learn the graph properties of source code (i.e., AST, CFG, DFG, and code sequences);
4. SySeVR [32] uses code statements, program dependencies, and program slicing as features for several RNN-based models (LR, MLP, DBN, CNN, LSTM, etc.) for classification;
5. VulDeePecker [33] leverages a Bidirectional LSTM network for statement-level vulnerability predictions;
6. Russell *et al.* [43] leverages an RNN-based model for vulnerability predictions.
7. BoW+RF (LineDP/JITLine) uses bag of words as features together with a Random Forest model for software defect predictions [38, 60].

Similar to Li *et al.* [30], we evaluate our LineVul with three binary classification measures i.e., Precision, Recall, and F1-score. Precision measures the proportion of the functions that are correctly predicted as vulnerable and the number of functions predicted as

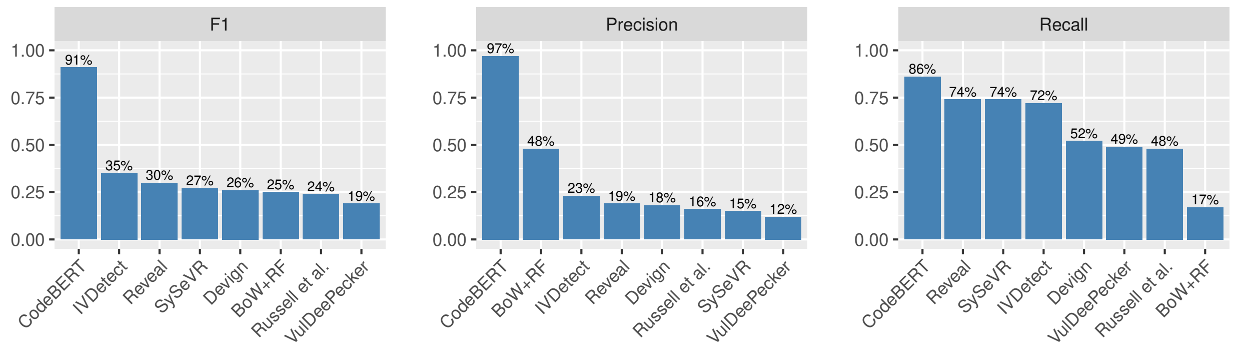


Figure 3: (RQ1) The experimental results of our LineVul and the seven baseline comparisons for function-level vulnerability prediction. (↗) Higher F1, Precision, Recall = Better.

vulnerable by the model, which is computed as $\frac{TP}{TP+FP}$. Recall measures the proportion of the functions that are correctly predicted as vulnerable and the number of actual vulnerable functions, which is computed as $\frac{TP}{(TP+FN)}$. F-measure is a harmonic mean of precision and recall, which is computed as $\frac{2\times\text{Precision}\times\text{Recall}}{\text{Precision}+\text{Recall}}$. We use the probability threshold of 0.5 as a cut-off value to indicate the prediction label. The values of these measures range from 0 to 1 where 1 indicates the highest accuracy.

Results. Figure 3 presents the experimental results of our LineVul and the seven baseline approaches according to our three evaluation measures (i.e., F1, Precision, Recall).

Our LineVul achieves an F-measure of 0.91, which is 160%-379% better than the state-of-the-art approaches with a median improvement of 250%. In terms of F-measure, Figure 3 shows that LineVul achieves the highest F-measure of 0.91, while the state-of-the-art approaches achieve an F-measure of 0.19-0.35. This finding shows that LineVul substantially improves the state-of-the-art by 160%-379% with a median improvement of 250%. In terms of Precision, Figure 3 shows that LineVul achieves the highest Precision of 0.97, while the state-of-the-art approaches achieve a Precision of 0.12-0.48. This finding shows that LineVul substantially improves the state-of-the-art by 102%-708% with a median improvement of 439%. In terms of Recall, Figure 3 shows that LineVul achieves the highest Recall of 0.86, while the state-of-the-art approaches achieve a Recall of 0.17-0.86. This finding shows that LineVul substantially improves the state-of-the-art by 16%-406% with a median improvement of 65%.

In other words, our results demonstrate that **the use of semantic and syntactic features with a Transformer architecture outperforms existing work that use graph properties of source code.** Our finding is different from the findings of many recent studies who found that the use of graph properties (e.g., Data Dependency Graph, Abstract Syntax Tree, Control Flow Graph, and Data Flow Graph) often outperform the use of syntactic and semantic features for vulnerability predictions [12, 29, 65]. This is because the RNN-based approaches (e.g., RNN, LSTM, GRU) used as a baseline in prior studies (1) are trained on a project-specific dataset; and (2) suffer from capturing the long-term dependencies of source code, as discussed in Section 2.1 (cf. Limitations ①, ②). Different from prior studies, our results confirm that our LineVul approach is more accurate than the state-of-the-art approaches, highlighting the substantial benefits of the use of the CodeBERT pre-trained language model trained on million GitHub repositories and the use of Transformer architecture to capture the long-term dependencies of source code, leading to significant improvement of the vulnerability prediction approach at the function level.

(RQ2) How accurate is our LineVul for line-level vulnerability localization?

Approach. To answer this RQ, we focus on evaluating the accuracy of the line-level vulnerability localization. Thus, we start from the vulnerable functions that are correctly predicted by our approach. Then, we perform the Step ③ (see Section 3.2) to identify which lines are likely to be vulnerable for a given function predicted as vulnerable. Thus, each line in the given function will have their own score (i.e, line-level score). Since other approaches in RQ1 are not designed for the line-level localization, we do not compare our approach with them. Instead, we compare our LineVul approach with 5 other model-agnostic techniques that are commonly used for deep learning models as follows:

(1) Layer Integrated Gradient (LIG) [48] is an axiomatic path-attribution method that attributes an importance score to each input feature by approximating the integral of gradients of the model's output with respect to the inputs along the path (straight line) from given baselines to inputs.;
(2) Saliency [47] takes a first-order Taylor expansion of the network at the input, and the gradients are simply the coefficients of each feature in the linear representation of the model. The absolute value of these coefficients can be taken to represent feature importance.;

(3) DeepLift [11, 46] compares the activation of each neuron to its reference activation and assigns contribution scores according to the difference;
(4) DeepLiftSHAP [36] extends DeepLift algorithm and approximates SHAP values using DeepLift approach;
(5) GradientSHAP [36] approximates SHAP values by computing the expectations of gradients by randomly sampling from the distribution of baselines;
(6) CppCheck [2] is a static code analysis tool for the C and C++ programming languages.

To evaluate our LineVul approach for line-level vulnerability localization, we use the following two measures described below:

1) Top-10 Accuracy measures the percentage of vulnerable functions where at least one actual vulnerable lines appear in the top-10 ranking. The intuition is that security analysts may ignore line-level recommendations if they do not appear in the top-10 ranking, similar to any recommendation systems [37]. Thus, top-10 accuracy will help security analysts better understand the accuracy of the line-level vulnerability localization approaches.
2) Initial False Alarm (IFA) measures the number of incorrectly predicted lines (i.e., non-vulnerable lines incorrectly predicted as vulnerable or false alarms) that security analysts need to inspect until finding the first actual vulnerable line for a given function. IFA is calculated as the total number of false alarms that security analysts have to inspect until finding the first actual vulnerable line. A low IFA value indicates that security analysts only spend less amount of effort in inspecting false alarms.

Results. Figure 4 presents the experimental results of our LineVul and the five baseline approaches according to our two evaluation measures (i.e., Top-10 Accuracy and IFA).

Our LineVul achieves a Top-10 Accuracy of 0.65, which is 12%-25% more accurate than the other baseline approaches. In terms of Top-10 Accuracy, Figure 4 shows that LineVul achieves the highest Top-10 Accuracy of 0.65 while the state-of-the-art approaches achieve a Top-10 Accuracy of 0.52-0.58. In terms of IFA, Figure 4 shows that LineVul achieves the lowest median IFA of 1 while the baseline approaches achieve a median IFA of 3-4. This finding shows that LineVul substantially improves the baseline approaches by 67%-75% with a median improvement of 67%. These results confirm that our LineVul approach is more accurate than the baseline approaches for line-level vulnerability localization.

In other words, our results demonstrate that **the attention mechanism outperforms other model-agnostic techniques.** In the line-level defect prediction literature, prior studies [38, 60] often leverage a LIME model-agnostic technique [42] to explain the predictions of DL/ML-based defect prediction models. However, such model-agnostic techniques are considered as a extrinsic model-agnostic technique (i.e,. a model-agnostic is applied to explain a black-box DL/ML model after the model is trained), not an intrinsic model-agnostic technique (i.e., a DL/ML model that is interpretable by itself so extrinsic model-agnostic techniques are not needed to apply afterward). Although it is widely known that deep learning is complex and hardly interpretable, this paper is among the first attempt to leverage the attention mechanism to explain the prediction of Transformer-based models, highlighting the substantial benefits of the attention mechanism for line-level vulnerability localization.

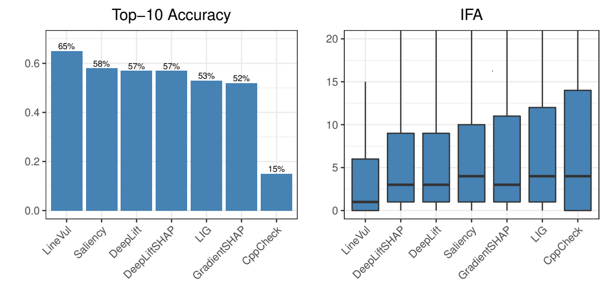


Figure 4: (RQ2) The Top-10 Accuracy and IFA of our self-attention approach and five other methods. (↗) Higher Top-10 Accuracy = Better, (↘) Lower IFA = Better.

(RQ3) What is the cost-effectiveness of our LineVul for line-level vulnerability localization?

Approach. To answer this RQ, we focus on evaluating the cost-effectiveness of our LineVul approach for line-level vulnerability localization. In the real-world scenario, the most cost-effective line-level vulnerability prediction approaches should help security analysts to find the highest number of actual vulnerable lines with the least amount of effort. Thus, let's assume that the 18,864 functions (i.e., 504,886 LOC) in the testing dataset are functions that security analysts have to inspect. To measure the cost-effectiveness of our approach, we first obtain the predictions from our LineVul approach. Then, we sort the predicted functions according to the predicted probability. Among the functions predicted as vulnerable, we sort the lines according to the line score obtained from Step ③ of our approach in descending order. Thus, lines with the highest score (i.e., likely to be vulnerable) will be ranked at the top. Then, we evaluate the cost-effectiveness using the following measures:

1) Effort@20%Recall measures the amount of effort (measured as LOC) that security analysts have to spend to find out the actual 20% vulnerable lines. It is computed as total LOC used to locate 20% of the actual vulnerable lines divided by total LOC in the testing set. A low value of Effort@20%Recall indicates that the security analysts may spend a smaller amount of effort to find the 20% actual vulnerable lines.
2) Recall@1%LOC measures the proportion of actual vulnerable lines that can be found (i.e., correctly predicted) given a fixed amount of effort (i.e., the top 1% of LOC of the given testing dataset). Recall@1%LOC is computed as the total number of correctly located vulnerable lines within the top 1% lines in the testing set divided by total number of actual vulnerable lines in the testing set. A high value of Recall@1%LOC indicates that an approach can rank many actual vulnerable lines at the top.

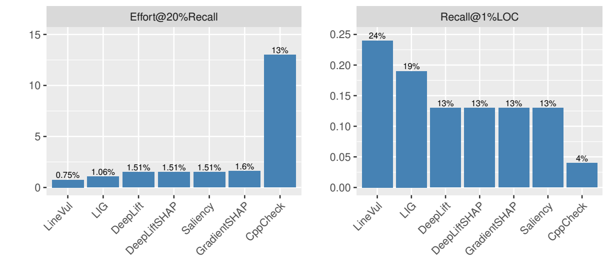


Figure 5: (RQ3) The Effort@20%Recall and Recall@1%LOC of our self-attention approach and five other methods. (↘) Lower Effort@20%Recall = Better, (↗) Higher Recall@1%LOC = Better.

<u>**Results.**</u> Figure 5 presents the experimental results of our LineVul and the five baseline approaches according to our two evaluation measures (i.e., Effort@20%Recall and Recall@1%LOC).

Our LineVul achieves an Effort@20%Recall of 0.75, which is 29%-53% less than other baseline approaches. In terms of Effort@20%Recall, Figure 5 shows that LineVul achieves the lowest Effort@20%Recall of 0.75% while the baseline approaches achieve an Effort@20%Recall of 1.06%-1.60%. It means that security analysts may spend effort to inspect 0.75% of the total LOC in the testing dataset (0.75%*504,886=3,786 LOC) in order to find 20% of the actual vulnerable lines (i.e., 20%Recall). Thus, this finding indicates that our approach may help security analysts spend less amount of effort in order to find the same amount of actual vulnerable lines.

In terms of Recall@1%LOC, Figure 5 shows that LineVul achieves the highest Recall@1%LOC of 0.24, while the baseline approaches achieve a Recall@1%LOC of 0.13-0.19, indicating that LineVul substantially improves the baseline approaches by 26%-85% with a median improvement of 85%. This finding implies that our approach may help security analysts to find more actual vulnerable lines than other baseline approaches given the same amount of effort that they have to inspect.

6 DISCUSSION

6.1 Why LineVul performs so well for vulnerability predictions?

We conduct an ablation study on function-level vulnerability prediction to quantify the contributions of the components of our LineVul approach. Generally, our LineVul approach consists of 3 components: BPE+Pretraining$_{\text{Code}}$+BERT. To understand the contribution of each component, we alter each of the components as follow (highlighted as underline):

- <u>*Word-Level*</u>*+Pretraining*$_{\text{Code}}$*+BERT:* Remove the BPE subword tokenization, but use a word-level tokenization instead.
- *BPE+*<u>*No Pretraining*</u>*+BERT:* Remove the pretraining, but use non-pretrained weights to initialize BERT instead.
- <u>*Word-Level*</u>*+*<u>*No Pretraining*</u>*+BERT:* Remove the BPE and the pre-training on source code components, but use Word-Level tokenization and non-pretrained weights to initialize BERT.

Table 1: The contribution of each component of LineVul for function-level vulnerability predictions.

Model	F1	Precision	Recall
LineVul (BPE+Pretraining$_{\text{Code}}$+BERT)	**0.91**	**0.97**	**0.86**
BPE+No Pretraining+BERT	0.80	0.86	0.75
Word-Level+Pretraining$_{\text{Code}}$+BERT	0.42	0.55	0.34
Word-Level+No Pretraining+BERT	0.39	0.43	0.36
IVDetect	0.35	0.23	0.72

We find that **the BPE component of our LineVul is the most important.** Within our LineVul, the BPE component contributes to 53.8% of the F-measure. When comparing between (BPE+Pre+BERT and Word-level+Pre+BERT) where the BPE component is changed to a word-level tokenizer, we observe a performance decrease from 0.91 to 0.42, accounting for 53.8%. This finding indicates that BPE subword-level tokenization is very beneficial for source code preprocessing than word-level tokenization. We suspect that source code often contains uncommon keywords (e.g., variable names, identifies) than the natural languages (e.g., English). Thus, language models of code may not be to generate the most meaningful vector representation for such uncommon keywords in the source code when using word-level tokenization. Instead, when using BPE subword-level tokenization, such uncommon words (e.g., *['unPremulSkImageToPremul']*) are broken into common subwords (e.g., *['un', 'Prem', 'ul', 'Sk', 'Image', 'To', 'Prem', 'ul']*). Karampatsis *et al.* [27] conduct a study of how different modelling choices affects the model performance of language models of source code and find the advancement of leveraging BPE model for language modeling of source code. Similar to their results, we find that the use of BPE will not only reduce the size of the unique vocabulary but also help the language models of code to better understand the relationship between a given subword and their surrounding subwords in order to generate more meaningful vector representation, achieving higher accuracy.

Within our LineVul, the pre-training component contributes to 12.1% of the F-measure. When comparing between (BPE+Pre+BERT and BPE+No Pre+BERT) where the pre-training component is eliminated, we observe a performance decrease from 0.91 to 0.80, accounting for 12.1%. This is because the pre-training language model of code has learnt the relationship of tokens from million GitHub repositories, therefore, generating more meaningful vector representation than the approach without pre-training. This finding confirms that the use of pre-training language model of code (i.e., CodeBERT) to generate vector representation outperforms an approach that is only trained on project-specific dataset.

Within our LineVul, BPE component combined with the pre-training component contribute to 57.1% of the F-measure. When comparing between (BPE+Pre+BERT and Word-level+No Pre+BERT) where both BPE and Pre-training are altered, we observe a performance decrease from 0.91 to 0.39, accounting for 57.1%.

Last but not least, we find that **our LineVul that leverages both BPE and pre-training provides the best F-measure among all variants**, which is 160% better than IVDetect, highlighting the

Table 2: (Discussion) The Accuracy of our LineVul for the Top-25 Most Dangerous CWEs (https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html).

Rank	CWE Type	Name	TPR	Proportion
1	CWE-787	Out-of-bounds Write	75%	18/24
4	CWE-20	Improper Input Validation	86%	98/114
8	CWE-22	Path Traversal	100%	4/4
12	CWE-190	Integer Overflow	90%	27/30
17	CWE-119	Improper Restriction	88%	173/197
20	CWE-200	Exposure of Sensitive Info	85%	45/53
25	CWE-77	Improper Neutralization	100%	2/2
		TOTAL	87%	367/424

significant advancement of LineVul for line-level vulnerability predictions.

6.2 How accurate is our LineVul for predicting the Top-25 Most Dangerous CWEs?

Our LineVul can correctly predict 87% of the vulnerable functions affected by the Top-25 most dangerous CWEs. CWE (Common Weakness Enumeration) is a list of vulnerability weaknesses in software that can lead to security issues with its severity of risk, providing guidance to organizations and security analysts to best secure their software systems. To better understand the significance of our LineVul on the practical usage scenarios, we perform a further investigation to better understand our accuracy for the Top-25 most dangerous CWEs. The Top-25 most dangerous CWEs are the most common and impactful issues experienced over the previous two calendar years. Such weaknesses are dangerous because they are often easy to find, exploit, and can allow adversaries to completely take over a system, steal data, or prevent an application from working. Since not all of the Top-25 most dangerous CWEs are included in the studied datasets, Table 2 presents the results for the ones that are included in the dataset. We compute the accuracy as the True Positive Rate (TPR) to focus on the vulnerable functions that are correctly predicted by our LineVul. We find that LineVul achieves an accuracy of 75% (CWE-787 Out-of-bounds Write) to 100% (CWE-22 Path Traversal, CWE-77 Improper Neutralization), depending on the CWE types in the dataset.

7 RELATED WORK

7.1 DL-based Vulnerability Prediction

Traditionally, ML-based vulnerability prediction approaches are proposed by using software metrics as features (e.g., code complexity) [45, 66]. The use of software metrics is also widely use in the defect prediction literature [24, 25, 49, 50, 53–57, 63]. However, the collection of such software metrics is manual and time-consuming. Thus, multiple DL-based approaches have been proposed to automatically learn the vulnerability patterns from historical data [12, 30–33, 39, 43, 65].

Therefore, an RNN-based architecture (i.e., LSTM) is used to automatically learn the semantic and syntactic features of source code [14, 43]. For example, Russell *et al.* [43] proposed an RNN-based architecture to automatically extract feature of source code for vulnerability prediction. Dam *et al.* [14] proposed a LSTM-based architecture to automatically learn the semantic and syntactic features of source code. However, the RNN-based approaches often assume that source code is a sequence of tokens without considering the graph structure of source code (e.g., Abstract Syntax Trees), leading to inaccurate predictions.

Therefore, Li *et al.* [33] proposed VulDeePecker which is an RNN-based model that is learnt from different types of graph properties of source code (e.g., Data Dependency Graph). However, the VulDeePecker approach still learns the graph properties in a sequential fashion, without leveraging the graph neural network. Therefore, a Graph Neural Network has been recently used to learn the graph properties of source code for vulnerability predictions. For example, Zhou *et al.* [65] leveraged a Graph Neural Network to learn four types of graph properties of source code, i.e., Abstract Syntax Tree, Control Flow Graph, Data Flow Graph, and syntactic features. Chakraborty *et al.* [12] proposed Reveal, which is a Gated Graph Neural Network (GGNN) that learns the graph properties of source code.

While these studies focus on the vulnerability predictions at the file/function level, our LineVul focuses on the line-level vulnerability prediction problem—which still remains largely unexplored.

7.2 Line-Level Vulnerability Prediction

Although various vulnerability prediction approaches are proposed, they mainly focus on the granularity of file, function, method levels—which is still coarse-grained. Thus, Li *et al.* [31, 32] proposed VulDeeLocator, which is based on a program slicing technique to narrow down the scope of vulnerability localization. In addition, Li *et al.* [30] also proposed IVDetect, which leverages a Graph Neural Network (GNN) for function-level predictions and a GNNExplainer to identify which sub-graph contributes the most to the predictions. Yet, security analysts still need to manually locate which lines in the sub-graph are actually vulnerable.

Similarly, line-level defect prediction has recently received high attention from the research community [38, 39, 60]. For example, Pornprasit and Tantithamthavorn [38] and Wattanakriengkrai *et al.* [60] proposed a machine learning-based approach with LIME model-agnostic technique (BoW+RF+LIME) to predict which lines are likely to be defective in the future. However, such approaches only learn the frequency of the appearance of code tokens in a file (i.e., Bag-of-Word), without considering the lexical and semantic of source code (i.e., the sequence of code tokens).

To the best of our knowledge, this paper is among the first to leverage the attention mechanism inside the BERT architecture for line-level vulnerability predictions.

7.3 Explainable AI for SE

The explainability of AI models in SE becomes one of the research grand challenges [51] (see http://xai4se.github.io), since practitioners often do not trust the predictions [52], hindering the adoption of AI-powered software development tools in practices. Recently, Explainable AI has been actively investigated in the domain of defect prediction [52, 58]. For example, recent works have shown some successful case studies to make defect prediction models more practical [38, 60], explainable [22, 28], and actionable [40, 41]. However, these studies only focus on explaining the traditional machine learning models, not the complex black-box deep learning models.

Recently, researchers start to explore the explainability of AI models in various SE tasks (i.e., leveraging the attention weights to provide meaningful 'explanations' for predictions). For example, Fu and Tantithamthavorn [18] proposed a GPT-2 based Agile story point estimation, by leveraging the Integrated Gradient attention to interpret the GPT-2 model and understand what words in a JIRA issue report contributed to the estimation of Agile story points. Similarly, Pornprasit and Tantithamthavorn [39] proposed a Hierarchical Attention Network (HAN) architecture for line-level defect prediction, by leveraging the attention mechanism of the HAN architecture to understand what code tokens in a source code contributed to the prediction of defective files. However, Jain *et al.* [21] argued that the learned attention weights are frequently uncorrelated with gradient-based measures of feature importance, while Wiegreffe *et al.* [61] argued that the accuracy/reliability of such attention weights could provide meaningful explanations, depending on the definition and the rigor of experimental design.

To the best of our knowledge, this paper is among the first to leverage the attention mechanism of BERT architecture for line-level vulnerability predictions. This concept is directly aligned with findings from AI discipline by Wiegreffe *et al.* [61]. However, this concept is still novel for line-level vulnerability predictions, since prior works [18, 39] only focused on explaining Agile story point estimations and defect predictions—*not CodeBERT-based line-level vulnerability predictions*. Our results in RQ2 and RQ3 confirm that the use of self-attention mechanism outperforms other model-agnostic techniques for line-level vulnerability predictions.

8 THREATS TO VALIDITY

Threats to the construct validity relate to the dataset selection. We use Fan *et al.* [16] dataset when conducting our experiments. Other datasets [12, 65] are not selected because they only provide ground-truths at the function level, which is not suitable for our research scope. Thus, we use the same dataset that was used by IVDetect for a fair comparison.

Moreover, our line-level prediction results (RQ2 and RQ3) are not compared with IVDetect [30], since their interpretation is based on subgraphs, not lines. The different granularity of local interpretation makes the comparison infeasible. In addition, LIME [42] that is widely used in prior defect prediction studies [22, 23, 38, 40, 41] is not used since it is best designed for ML techniques, not complex deep neural networks. To mitigate this threat, we compare our approach with many other advanced model-agnostic techniques that are suitable for deep neural networks, e.g., Layer Integrated Gradient [48], Saliency [47], DeepLift [46], and SHAP-based techniques [36]. Our experiment results still confirm that the self-attention used by our LineVul outperforms other baseline approaches.

Threats to the internal validity relate to hyperparameter settings when fine-tuning our LineVul model. We use the default hyperparameter settings as specified by Feng *et al.* [17]. As hyperparameter tuning is extremely expensive for a BERT-based model that consists of millions of parameters, we only tune the learning rate and set it to $2e^{-5}$ in the end.

The window size of our approach is limited to 512 tokens. Thus, our approach may not be able to fully learn any function that is longer than 512 tokens. Nevertheless, to maximize the benefit of using the CodeBERT pre-trained model, it is best to not extend the window size. Thus, extending the window size of our approach can be explored in future work.

In RQ1, the IVDetect result is reused from Li *et al.* [30], which cannot verified by us. While Li *et al.* [30] published a replication package for future research, we attempted our best to reproduce the results of the IVDetect approach. Unfortunately, we are not able to rerun their experimental scripts. This concern is also shared by others [6]. To mitigate this threat, we have to reuse the IVDetect result for RQ1 in order to ensure that there exists no potential bias in the re-implementation of the baseline approaches. Nevertheless, we strictly followed the experimental setup of IVDetect to ensure that we used the same data splitting strategy for a fair comparison (80% for training, 10% for validation, and 10% for testing). In addition, time-wise evaluation scenarios are not considered in this paper, since the dataset is at the function level (not the commit level). To mitigate these threats, we publish our replication package to improve the transparency of our work.

In RQ2, Top-10 accuracy is only meaningful for the function size of greater than 10 lines. We observed that there are 16% of vulnerable functions (1,840/10,900) that have less than 10 lines. Thus, Top-10 accuracy is still sensible for the majority of the vulnerable functions (84%). To mitigate this threat, we experiment with other k values (i.e., Top-3 and Top-5 Accuracy). We found that the attention mechanism used in our approach is still top-performing (see in Appendix in our replication package). Thus, the k value does not pose any threats to the validity of our results.

Threats to the external validity relate to the generalizability of our LineVul approach. We conduct our experiment using a large-scale line-level vulnerability dataset (i.e., Fan *et al.* dataset) to ensure a fair comparison with the IVDetect approach [30]. Thus, other line-level vulnerability datasets can be explored in future work.

9 CONCLUSION

In this paper, we propose LineVul, a Transformer-based vulnerability prediction approach to predict which functions will be vulnerable and which lines are vulnerable. Through an empirical evaluation of a large-scale real-world dataset with 188k+ C/C++ functions, we show that LineVul achieves (1) 160%-379% higher F1-measure for function-level predictions; (2) 12%-25% higher Top-10 Accuracy for line-level predictions; and (3) 29%-53% less Effort@20%Recall than the state-of-the-art approaches. Our results confirm that LineVul is more accurate and more fine-grained than existing vulnerability prediction approaches. Thus, we expect that our LineVul may help security analysts to find vulnerable lines in a cost-effective manner.

Acknowledgement. Chakkrit Tantithamthavorn is partially supported by the Australian Research Council's Discovery Early Career Researcher Award (DECRA) funding scheme (DE200100941).

REFERENCES

[1] [n.d.]. Checkmarx. https://checkmarx.com/.

[2] [n.d.]. Cppcheck. https://cppcheck.sourceforge.io/.

[3] [n.d.]. CWE-787. https://cwe.mitre.org/data/definitions/787.html.

[4] [n.d.]. Cybercrime To Cost The World $10.5 Trillion Annually By 2025. https://cybersecurityventures.com/hackerpocalypse-cybercrime-report-2016/.

[5] [n.d.]. Flawfinder. https://dwheeler.com/flawfinder/.

[6] [n.d.]. IVDectect Replication Package Issue #1: Cannot reproduce. https://github.com/vulnerabilitydetection/VulnerabilityDetectionResearch/issues/1.

[7] [n.d.]. Microsoft Exchange Flaw: Attacks Surge After Code Published. https://www.bankinfosecurity.com/ms-exchange-flaw-causes-spike-in-trdownloader-gen-trojans-a-16236.

[8] [n.d.]. ProxyLogon Flaw. https://proxylogon.com/.

[9] [n.d.]. RATS. https://code.google.com/archive/p/rough-auditing-tool-for-security/.

[10] [n.d.]. THE COST OF CYBERCRIME. https://www.accenture.com/_acnmedia/pdf-96/accenture-2019-cost-of-cybercrime-study-final.pdf.

[11] Marco Ancona, Enea Ceolini, Cengiz Öztireli, and Markus Gross. 2018. Towards better understanding of gradient-based attribution methods for Deep Neural Networks. In *6th International Conference on Learning Representations (ICLR)*. Arxiv-Computer Science, 0–0.

[12] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering* (2021).

[13] R. Collobert, K. Kavukcuoglu, and C. Farabet. 2011. Torch7: A Matlab-like Environment for Machine Learning. In *BigLearn, NIPS Workshop*.

[14] Hoa Khanh Dam, Truyen Tran, Trang Pham, Shien Wee Ng, John Grundy, and Aditya Ghose. 2017. Automatic feature learning for vulnerability prediction. *arXiv preprint arXiv:1708.02368* (2017).

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. 4171–4186.

[16] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. AC/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 508–512.

[17] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. 1536–1547.

[18] Michael Fu and Chakkrit Tantithamthavorn. 2022. GPT2SP: A Transformer-Based Agile Story Point Estimation Approach. *IEEE Transactions on Software Engineering* (2022).

[19] Mukesh Kumar Gupta, MC Govil, and Girdhari Singh. 2014. Static analysis approaches to detect SQL injection and cross site scripting vulnerabilities in web applications: A survey. In *International Conference on Recent Advances and Innovations in Engineering (ICRAIE-2014)*. IEEE, 1–5.

[20] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).

[21] Sarthak Jain and Byron C Wallace. 2019. Attention is not Explanation. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. 3543–3556.

[22] Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, Hoa Khanh Dam, and John Grundy. 2020. An Empirical Study of Model-Agnostic Techniques for Defect Prediction Models. *IEEE Transactions on Software Engineering (TSE)* (2020), To Appear.

[23] Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, and John Grundy. 2021. Practitioners' Perceptions of the Goals and Visual Explanations of Defect Prediction Models. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*. To Appear.

[24] Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, and Ahmed E Hassan. 2021. The Impact of Correlated Metrics on Defect Models. *IEEE Transactions on Software Engineering* (2021).

[25] Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, and Christoph Treude. 2018. AutoSpearman: Automatically Mitigating Correlated Software Metrics for Interpreting Defect Models. In *ICSME*. 92–103.

[26] Arnold Johnson, Kelley Dempsey, Ron Ross, Sarbari Gupta, Dennis Bailey, et al. 2011. Guide for security-focused configuration management of information systems. *NIST special publication* 800, 128 (2011), 16–16.

[27] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big code!= big vocabulary: Open-vocabulary models for source code. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 1073–1085.

[28] Chaiyakarn Khanan, Worawit Luewichana, Krissakorn Pruktharathikoon, Jirayus Jiarpakdee, Chakkrit Tantithamthavorn, Morakot Choetkiertikul, Chaiyong Ragkhitwetsagul, and Thanwadee Sunetnanta. 2020. JITBot: An Explainable Just-In-Time Defect Prediction Bot. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1336–1339.

[29] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[30] Yi Li, Shaohua Wang, and Tien N Nguyen. 2021. Vulnerability detection with fine-grained interpretations. In *29th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2021*. Association for Computing Machinery, Inc, 292–303.

[31] Zhen Li, Deqing Zou, Shouhuai Xu, Zhaoxuan Chen, Yawei Zhu, and Hai Jin. 2021. Vuldeelocator: a deep learning-based fine-grained vulnerability detector. *IEEE Transactions on Dependable and Secure Computing* (2021).

[32] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2021. SySeVR: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* (2021).

[33] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. *arXiv e-prints* (2018), arXiv–1801.

[34] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[35] Ilya Loshchilov and Frank Hutter. 2018. Decoupled Weight Decay Regularization. In *International Conference on Learning Representations*.

[36] Scott M Lundberg and Su-In Lee. 2017. A unified approach to interpreting model predictions. In *Proceedings of the 31st international conference on neural information processing systems*. 4768–4777.

[37] Chris Parnin and Alessandro Orso. 2011. Are automated debugging techniques actually helping programmers?. In *Proceedings of the 2011 international symposium on software testing and analysis*. 199–209.

[38] Chanathip Pornprasit and Chakkrit Tantithamthavorn. 2021. JITLine: A Simpler, Better, Faster, Finer-grained Just-In-Time Defect Prediction. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*.

[39] Chanathip Pornprasit and Chakkrit Tantithamthavorn. 2022. DeepLineDP: Towards a Deep Learning Approach for Line-Level Defect Prediction. *IEEE Transactions on Software Engineering* (2022).

[40] Chanathip Pornprasit, Chakkrit Tantithamthavorn, Jirayus Jiarpakdee, Michael Fu, and Patanamon Thongtanunam. 2021. PyExplainer: Explaining the Predictions of Just-In-Time Defect Models. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 407–418.

[41] Dilini Rajapaksha, Chakkrit Tantithamthavorn, Christoph Bergmeir, Wray Buntine, Jirayus Jiarpakdee, and John Grundy. 2021. SQAPlanner: Generating data-informed software quality improvement plans. *IEEE Transactions on Software Engineering* (2021).

[42] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. " Why should i trust you?" Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. 1135–1144.

[43] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. 2018. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE international conference on machine learning and applications (ICMLA)*. IEEE, 757–762.

[44] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural Machine Translation of Rare Words with Subword Units. In *54th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics (ACL), 1715–1725.

[45] Yonghee Shin and Laurie Williams. 2008. An empirical model to predict security vulnerabilities using code complexity metrics. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*. 315–317.

[46] Avanti Shrikumar, Peyton Greenside, and Anshul Kundaje. 2017. Learning important features through propagating activation differences. In *International Conference on Machine Learning*. PMLR, 3145–3153.

[47] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. 2013. Deep inside convolutional networks: Visualising image classification models and saliency maps. *arXiv preprint arXiv:1312.6034* (2013).

[48] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. 2017. Axiomatic attribution for deep networks. In *International Conference on Machine Learning*. PMLR, 3319–3328.

[49] Chakkrit Tantithamthavorn. 2016. Towards a Better Understanding of the Impact of Experimental Components on Defect Prediction Modelling. In *Companion Proceeding of the International Conference on Software Engineering (ICSE)*. 867—-870.

[50] Chakkrit Tantithamthavorn, Ahmed E Hassan, and Kenichi Matsumoto. 2018. The impact of class rebalancing techniques on the performance and interpretation

of defect prediction models. *IEEE Transactions on Software Engineering* 46, 11 (2018), 1200–1219.

[51] Chakkrit Tantithamthavorn and Jirayus Jiarpakdee. 2021. Explainable AI for Software Engineering. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1–2.

[52] Chakkrit Tantithamthavorn, Jirayus Jiarpakdee, and John Grundy. 2021. Actionable Analytics: Stop Telling Me What It Is; Please Tell Me What To Do. *IEEE Software* 38, 4 (2021), 115–120.

[53] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, Akinori Ihara, and Kenichi Matsumoto. 2015. The Impact of Mislabelling on the Performance and Interpretation of Defect Prediction Models. In *ICSE*. 812–823.

[54] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. 2016. Automated Parameter Optimization of Classification Techniques for Defect Prediction Models. In *ICSE*. 321–332.

[55] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. 2016. Comments on "Researcher Bias: The Use of Machine Learning in Software Defect Prediction". *TSE* 42, 11 (2016), 1092–1094.

[56] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. 2017. An Empirical Comparison of Model Validation Techniques for Defect Prediction Models. *TSE* (2017), 1–18.

[57] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, and Kenichi Matsumoto. 2019. The Impact of Automated Parameter Optimization on Defect Prediction Models. *TSE* (2019).

[58] Chakkrit Kla Tantithamthavorn and Jirayus Jiarpakdee. 2021. Explainable ai for software engineering. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1–2.

[59] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems (NeurIPS)*. 5998–6008.

[60] Supatsara Wattanakriengkrai, Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Hideaki Hata, and Kenichi Matsumoto. 2020. Predicting defective lines using a model-agnostic technique. *IEEE Transactions on Software Engineering (TSE)* (2020).

[61] Sarah Wiegreffe and Yuval Pinter. 2019. Attention is not not Explanation. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. 11–20.

[62] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2019. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771* (2019).

[63] Suraj Yatish, Jirayus Jiarpakdee, Patanamon Thongtanunam, and Chakkrit Tantithamthavorn. 2019. Mining Software Defects: Should We Consider Affected Releases?. In *ICSE*. 654–665.

[64] Rex Ying, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. 2019. Gnnexplainer: Generating explanations for graph neural networks. *Advances in neural information processing systems (NeurIPS)* 32 (2019), 9240.

[65] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. 10197–10207.

[66] Thomas Zimmermann, Nachiappan Nagappan, and Laurie Williams. 2010. Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In *2010 Third international conference on software testing, verification and validation*. IEEE, 421–428.