

Weakly Supervised Vulnerability Localization via Multiple Instance Learning

WENCHAO GU, The Chinese University of Hong Kong, China

YUPAN CHEN, Harbin Institute of Technology, Shenzhen, China

YANLIN WANG, Sun Yat-sen University (Zhuhai), China

HONGYU ZHANG, Chongqing University, China

CUIYUN GAO*, Harbin Institute of Technology, Shenzhen, China

MICHAEL R. LYU, The Chinese University of Hong Kong, China

Software vulnerability detection has emerged as a significant concern in the field of software security recently, capturing the attention of numerous researchers and developers. Most previous approaches focus on coarse-grained vulnerability detection, such as at the function or file level. However, the developers would still encounter the challenge of manually inspecting a large volume of code inside the vulnerable function to identify the specific vulnerable statements for modification, indicating the importance of vulnerability localization. Training the model for vulnerability localization usually requires ground-truth labels at the statement-level, and labeling vulnerable statements demands expert knowledge, which incurs high costs. Hence, the demand for an approach that eliminates the need for additional labeling at the statement-level is on the rise. To tackle this problem, we propose a novel approach called WAVES for **We**Akly supervised **V**ulnerability **L**ocalization via **m**ultiple **I**n**S**tance learning, which does not need the additional statement-level labels during the training. WAVES has the capability to determine whether a function is vulnerable (i.e., vulnerability detection) and pinpoint the vulnerable statements (i.e., vulnerability localization). Specifically, inspired by the concept of multiple instance learning, WAVES converts the ground-truth label at the function-level into pseudo labels for individual statements, eliminating the need for additional statement-level labeling. These pseudo labels are utilized to train the classifiers for the function-level representation vectors. Extensive experimentation on three popular benchmark datasets demonstrates that, in comparison to previous baselines, our approach achieves comparable performance in vulnerability detection and state-of-the-art performance in statement-level vulnerability localization.

CCS Concepts: • **Software and its engineering**; • **Security and privacy** → **Software security engineering**; • **Computing methodologies** → **Knowledge representation and reasoning**;

Additional Key Words and Phrases: vulnerability detection, neural networks, vulnerability localization, multiple instance learning

ACM Reference Format:

Wenchao GU, Yupan Chen, Yanlin Wang, Hongyu Zhang, Cuiyun Gao, and Michael R. Lyu. 2023. Weakly Supervised Vulnerability Localization via Multiple Instance Learning. *J. ACM* 37, 4, Article 111 (August 2023), 32 pages. <https://doi.org/XXXXXXX.XXXXXXX>

*Corresponding author.

Authors' addresses: Wenchao GU, The Chinese University of Hong Kong, Hong Kong, China, wengu@cse.cuhk.edu.hk; Yupan Chen, Harbin Institute of Technology, Shenzhen, Guangdong, China, cyp36889@gmail.com; Yanlin Wang, Sun Yat-sen University (Zhuhai), Zhuhai, Guangdong, China, wangylin36@mail.sysu.edu.cn; Hongyu Zhang, Chongqing University, Chongqing, China, hongyujohn@gmail.com; Cuiyun Gao, Harbin Institute of Technology, Shenzhen, Guangdong, China, gaocuiyun@hit.edu.cn; Michael R. Lyu, The Chinese University of Hong Kong, Hong Kong, China, lyu@cse.cuhk.edu.hk.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2023 Association for Computing Machinery.

Manuscript submitted to ACM

Manuscript submitted to ACM

1 INTRODUCTION

Software vulnerabilities are flaws in the logical design of software or operating systems that can be exploited maliciously by attackers. By exploiting these vulnerabilities, attackers can implant Trojan horses and viruses over networks, extract crucial user information, and even inflict severe damage to the system[17]. The detection of software vulnerabilities has emerged as a crucial issue in the realm of software security, garnering considerable interest from researchers and developers in recent decades.

Most traditional vulnerability detection tools [13, 22, 38, 43, 45], such as Flawfinder [45], employ static analysis techniques to identify vulnerabilities in programs. These tools typically rely on predefined vulnerability patterns to determine whether the target programs are vulnerable. Although these tools can effectively detect well-defined vulnerabilities such as use-after-free issues, they often struggle to identify vulnerabilities that are not easily defined. Furthermore, the manual definition of vulnerability patterns is a time-consuming process that can hinder the efficiency of these methods. Moreover, these tools often generate a large number of false positives/negatives in their reported vulnerabilities [7], further diminishing their utility.

With the advancement of deep learning techniques, there has been a growing interest in using these methods for vulnerability detection in recent years. Various deep learning-based approaches have been proposed by researchers, leveraging neural networks like Convolutional Neural Networks (CNNs) [36], Recurrent Neural Networks (RNNs) [21], and Graph Neural Networks (GNNs) [6, 52]. These approaches enable automatic acquisition of vulnerability features or patterns from training data. Notably, these techniques have demonstrated their effectiveness in detecting unreported or unknown vulnerabilities [27].

However, most current deep learning-based approaches for vulnerability detection only offer predictions at the function level. This falls short of developers' needs because the majority of vulnerable statements within the code tend to be relatively concealed and challenging to uncover. Even with function-level predictions, developers still face the time-consuming task of locating these vulnerable statements. Some previous approaches [16, 26] have aimed to enhance the interpretability of models to help developers identify vulnerable statements. However, these methods [16, 26] have struggled to achieve accurate localization, as they do not prioritize the localization problem during training and solely rely on attention scores or GNN explainers [49] to explain the model's behavior after training. Automatically predicting statement-level vulnerabilities in a supervised manner poses difficulties, as it necessitates labeled data for model learning. Therefore, there is an urgent demand for unsupervised or weakly supervised approaches for statement-level vulnerability localization.

Multiple instance learning (MIL) is a form of weakly supervised learning that finds extensive use across diverse tasks, including drug activity prediction [3], image retrieval [2, 35], and text classification [23]. MIL handles training data arranged in sets called bags, where each bag contains multiple instances. In MIL, only the label for the entire bag is provided, while the labels for individual instances are unknown. MIL allows us to convert the bag's label into pseudo-labels for each instance, thus addressing the missing label problem during instance classification training. This scenario is similar to vulnerability localization, where only the vulnerability label for the entire function is given, and the labels for each statement are unavailable. And we discovered that we can effectively reframe the problem of vulnerability localization as a Multiple Instance Learning (MIL) problem. In this approach, each statement within the target function is treated as an instance, and the entire function represents a bag. Since each bag (function) contains multiple instances (statements), it follows that a vulnerable function must include at least one vulnerable statement. Conversely, if a function is not vulnerable, all its included statements are considered non-vulnerable. This logic parallels

the relationship between the label of a bag (function) and the labels of its instances (statements) in MIL. However, applying MIL to this task faces two primary challenges. Firstly, when employing deep learning models for code analysis, the smallest unit of input is typically variable names or subtokens, rather than whole statements. This complicates the generation of statement-level representation vectors that seamlessly integrate with the MIL framework. Secondly, traditional MIL approaches assume independence among instances within the same bag, implying no interaction across different instances [5]. However, in vulnerability detection, a single statement often does not raise vulnerability concerns; vulnerabilities typically arise from specific contextual statements within the program. To tackle these challenges, we propose a Transformer-based model within the framework of multiple instance learning. This model can effectively capture local and global vulnerability information for each statement and allows statements within the same function to interact during the training. Additionally, it retains the core concept of generating pseudo instance labels from the bag label. By adopting this approach, we can construct pseudo-labeled training instances, reducing the labor-intensive task of manually labeling vulnerabilities at the statement level.

In this paper, we propose a novel approach named WAVES for function-level vulnerability detection with statement-level localization. WAVES first converts an input code snippet into a token sequence and feeds it into a Transformer-based encoder. During the encoding process, tokens from the same or different statements interact freely, enabling the model to learn contextual information for each statement. There are two channels aiming to capture local and global features separately. The statement-level classifier for each channel is then trained individually to determine whether the statement-level representation vectors are vulnerable or not. The results from these two classifiers are combined to produce a single prediction for a single statement. The evaluation of WAVES is conducted using three widely used datasets, and extensive experimental findings showcase that WAVES achieves comparable performance in detecting vulnerabilities at the function level compared to previous models. Furthermore, its ability to localize vulnerabilities surpasses that of the previous models.

We summarize the main contributions of this paper as follows:

- We introduce WAVES, a novel approach that achieves vulnerability localization without requiring additional statement-level labels during model training. Furthermore, the statement-level predictions from WAVES can be leveraged for function-level vulnerability detection. To our best knowledge, WAVES is the first approach to adopt multiple instance learning for localizing vulnerabilities at the statement level, all without requiring additional vulnerability labeling at the statement level.
- We integrate various pooling modules capable of capturing code features specific to vulnerabilities. We also validate the effectiveness of each pooling module on the overall performance.
- We have performed comprehensive experiments on public benchmarks, and the results indicate that WAVES achieves comparable performance in function-level vulnerability detection and outperforms previous models in statement-level vulnerability localization, showcasing state-of-the-art performance.

The remainder of this paper is structured as follows: Section 2 provides an overview of the architecture of our proposed WAVES, along with its design details. Section 3 describes our experimental setup, including the datasets used, evaluation metrics, and implementation specifics. In Section 4, we present the experimental results and provide our analysis. In Section 6, we discuss the threats to the validity of our experiments. Section 7 discusses the related work on vulnerability detection and multiple instance learning, while Section 8 concludes the paper.

2 METHODOLOGY

In this section, we propose a novel deep learning-based vulnerability detection approach that can achieve function-level detection and statement-level localization simultaneously with weakly supervised learning. Specifically, we present the overview and detailed design of the proposed approach WAVES, including model design, model training strategy, and inference strategy.

2.1 Overview

Figure 1 illustrates an overview of the proposed approach WAVES. Our approach consists of three steps: code encoding, multiple instance learning-based training strategy, and model inference. In the step of code encoding, the Transformer-based encoder learns to generate representation vectors for each statement within the given function. Two linear classifiers are trained to classify whether these statement-level representation vectors indicate vulnerability or not. In the multiple instance learning-based training strategy, we convert function-level ground-truth labels for vulnerability detection into statement-level pseudo labels and utilize these pseudo labels for the model training. During the inference step, the model also generates representation vectors for each statement and employs the previous two trained linear classifiers to determine their vulnerability status. Additionally, the overall vulnerability prediction for the entire function is determined by considering the vulnerability predictions of each statement within the function. Further details about these three steps will be presented in the following sections.

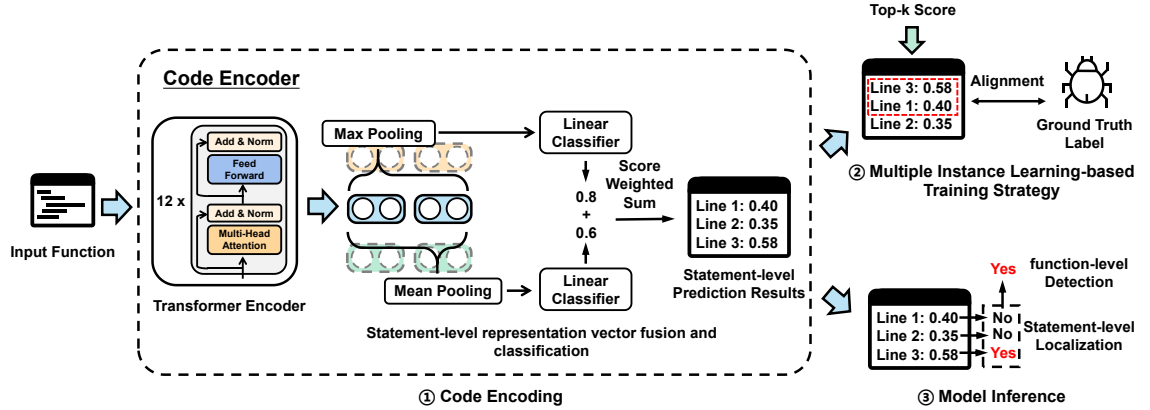


Fig. 1. An overview architecture of WAVES, containing three main steps. ① Code Encoding: The target function will first be transformed into a token sequence and then fed into a Transformer-based Encoder. Subsequently, the token vectors will be integrated into statement-level vectors, and two linear classifiers will be employed to classify them. ② Multiple Instance Learning-based Training Strategy: The statements will be ranked in descending order based on the statement-level predicted results. The top-k statements will then be assigned pseudo labels identical to the function label for model training purposes. ③ Model Inference: The results obtained in Step 1 will be utilized to predict the vulnerability of each statement. The prediction results from all the statements will then be used to determine the vulnerability of the entire function.

2.2 Code Encoding

Following previous work [16], we adopt the Transformer as the base model for code encoding. The given code snippet will be converted into a token sequence and each token will be split into sub words by the tokenizer. We adopt the

Byte Pair Encoding (BPE) approach [37] from Roberta as our tokenizer to tokenize the word. To incorporate the positional relationships between the subwords into the model, positional embedding vectors are encoded to represent the token's position within the token sequence. The token embedding vector and the positional embedding vector are then combined into a unified representation vector, which represents the corresponding token within the input sequence.

It is necessary to record the location information of each token for the generation of statement-level representation vectors in subsequent stages. In order to capture this information, we create a binary statement indicative matrix. The matrix, denoted as S , is defined as follows:

$$S = \{s_{11}, \dots, s_{1n}, \dots, s_{m1}, \dots, s_{mn}\} \quad (1)$$

where n is the token number, m is the statement number, and s_{ij} indicates whether the i -th token belongs to the j -th statement in the given function. The value of s_{ij} will be 1 if the i -th token belongs to the j -th statement in the given function; otherwise, s_{ij} will be zero.

In the subsequent sections, we will introduce the design of the code encoder, which takes both the token embedding sequence and the binary statement indicative matrix as inputs and generates statement-level prediction scores.

2.3 The Design of Code Encoder

In this subsection, we present the code encoder of our proposed approach WAVES. The code encoder consists of a Transformer-based encoder, as well as linear classifiers for both the max pooling channel and the mean pooling channel.

2.3.1 Transformer Encoder with Self-attention. In our approach WAVES, we utilize a Transformer-based encoder. The encoder comprises 12 stacked Transformer blocks, each consisting of a multi-head self-attention layer and a fully-connected feed-forward neural network. The multi-head self-attention layer's purpose is to generate the attention vector based on the attention score assigned to each code token. To accomplish this, the dot product between the query vector of the current code token and the key vectors of the other tokens is computed. Subsequently, the dot product is normalized to probabilities via the Softmax function. Finally, the attention vector is obtained by taking dot product between the value vectors and previous normalized probabilities. The equation for calculating the attention score is provided below:

$$Attention(Q, K, V) = \text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V, \quad (2)$$

where Q, K, V is the query vector, key vector, and value vector, respectively.

The multi-head mechanism enables the model to create several subspaces, each dedicated to different aspects of the input sequence. This allows the model to effectively capture diverse semantic information from the input. Initially, the multi-head mechanism divides the input vectors into h heads, with each head having a dimension of $\frac{d}{h}$. Following the self-attention operation on each head, these heads are then concatenated back together as:

$$MultiHead(Q, K, V) = \text{Concat}(head_1, \dots, head_h)W^O, \quad (3)$$

where $head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$ and W^O is the projection matrix for the concatenated vectors.

Finally, the concatenated vectors will be fed into a fully-connected feed-forward neural network. This neural network comprises two linear layers, with a ReLU activation function sandwiched between them. As demonstrated by numerous previous studies [15, 18, 19], Transformers have advantages over RNNs in capturing long-term dependencies. They are

more effective at capturing contextual information among different statements, even with variations in edit distance. This is why we chose Transformers for encoding the code.

2.3.2 Statement-level Representation Vector Fusion. After the encoding of the input token sequence from the previous Transformer-based encoder, we obtained the representation vector for each token from the last hidden states in the model. Since previous Transformer-based approaches focus solely on function-level vulnerability detection, they do not require generating statement-level representation vectors. In contrast, our approach [16] aims to achieve statement-level vulnerability localization, making it crucial to generate these vectors. To accomplish this, we need to merge the embedding vectors of tokens within the same statement into a single statement-level representation vector. We propose two methods for efficiently fusing these vectors: max pooling and mean pooling. The max pooling method captures local suspicious information within a statement, while the mean pooling method captures suspicious information from a broader, statement-wide perspective.

Max Pooling Channel: Max pooling is a down-sampling technique frequently used in deep learning to efficiently preserve local features from the original data. This property enables the model to capture local information, such as variable or API misuse, which is crucial for software vulnerability detection. The operation of max pooling is illustrated as follows:

$$v_{max_j} = \max(h_1 \cdot s_{1j}, \dots, h_n \cdot s_{nj}) \quad (4)$$

where v_{max_j} is the representation vector for the locality information in the j -th statement, h_i is the hidden vector for the i -th token from the encoder, and s_{ij} is the indicator to show whether the i -th token belongs to the j -th statement. $h_i \cdot s_{ij}$ can remove the irrelevant token vectors during the operation of max pooling.

Mean Pooling Channel: Mean pooling is a down-sampling technique frequently used in deep learning to efficiently preserve global features from the original input. This technique assists the model in determining if the execution of a single statement might raise a vulnerability issue. The operation of mean pooling is illustrated as follows:

$$v_{mean_j} = \frac{\sum_{i=1}^n h_i \cdot s_{ij}}{\sum_{i=1}^n s_{ij}} \quad (5)$$

where v_{mean_j} is the representation vector for the global information in the j -th statement, h_i is the hidden vector for the i -th token from the encoder, and s_{ij} is the indicator to show whether the i -th token belongs to the j -th statement. $h_i \cdot s_{ij}$ can remove the irrelevant token vectors during the operation of mean pooling.

```

1  #define JAN 1
2  #define FEB 2
3  #define MAR 3
4
5  short getMonthlySales(int month) {...}
6
7  float calculateRevenueForQuarter(short quarterSold) {...}
8
9  int determineFirstQuarterRevenue() {
10
11     // Variable for sales revenue for the quarter
12     float quarterRevenue = 0.0f;
13
14     short JanSold = getMonthlySales(JAN); /* Get sales in January */
15     short FebSold = getMonthlySales(FEB); /* Get sales in February */
16     short MarSold = getMonthlySales(MAR); /* Get sales in March */

```

```

17
18 // Calculate quarterly total
19 short quarterSold = JanSold + FebSold + MarSold;
20
21 // Calculate the total revenue for the quarter
22 quarterRevenue = calculateRevenueForQuarter(quarterSold);
23
24 saveFirstQuarterRevenue(quarterRevenue);
25
26 return 0;
27 }

```

Listing 1. A motivating example for mean pooling.

Here is a motivating example from CWE-190, specifically "Integer Overflow" or "Wraparound," presented in Listing 1. In line 19, the values of the variables JanSold, FebSold, and MarSold are summed and assigned to the variable quarterSold. However, there is a potential risk of integer overflow if the sum exceeds the maximum value allowed for the short int data type. In this example, the max pooling channel can help the model focus on the data type of quarterSold being short, and the mean pooling channel can help the model notice the sum operation in this statement. By combining the information from these two channels, the model can detect a potential data overflow problem.

2.3.3 Representation Vector Classification and Fusion. Upon combining the original embedding vectors of the tokens within a statement, we generate two representation vectors that encompass both local and global information. Then we employ two linear classifiers to classify these two representation vectors as the binary classification task, respectively. Each classifier consists of a fully connected layer followed by a softmax activation function. The scores produced by these classifiers are then combined through a weighted sum, resulting in a single score that serves as the final prediction for the statement. This fusion enables the model to assess whether a given statement is vulnerable by considering both local and global perspectives, thereby enhancing the model's detection capability.

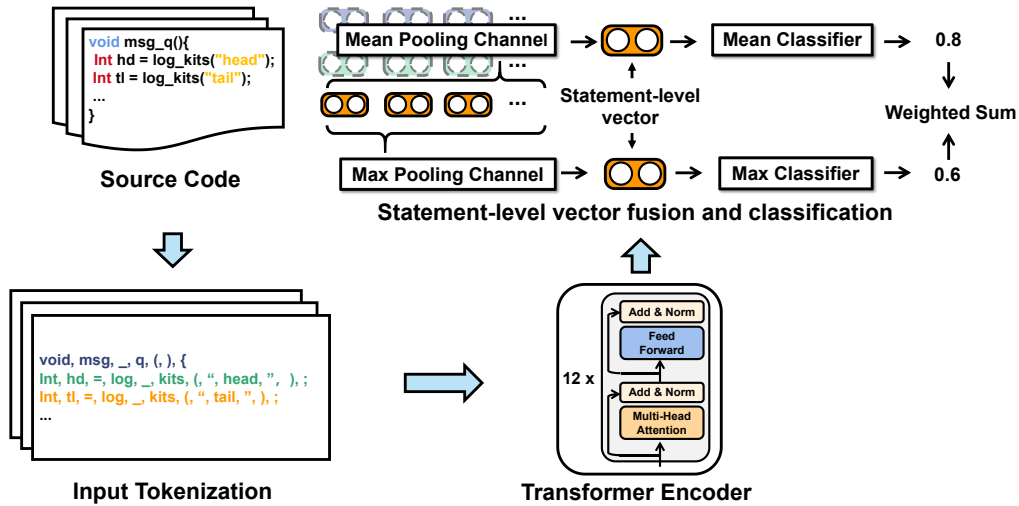


Fig. 2. An illustrative example of code encoder design

Figure 2 illustrates our code encoder design. Initially, the code is split into several statements, and each statement is tokenized into a sequence of tokens. These token sequences are then fed into a Transformer-based encoder, where each token is encoded into a vector. Next, the token vectors belonging to the same statement are fused into two statement-level vectors via mean pooling and max pooling channels, respectively. Finally, the statement vectors are fed into two classifiers, one for mean pooling and one for max pooling. Predictions from the local and global views are obtained and combined into a single prediction score.

2.4 Multiple Instance Learning-Based Training Strategy

Despite generating the representation vector and utilizing the classifier for individual statements, the absence of statement-level labels poses an ongoing challenge. Inspired by the concept from multiple instance learning, we convert the label of the entire function into pseudo labels for each statement within the target function to tackle this issue. These pseudo-labels serve as the supervised signal during training.

It is known that a vulnerable function must contain at least one vulnerable statement, whereas a non-vulnerable function has no vulnerable statements. In a given code snippet, the label y_i represents the vulnerability status of i -th statement, where y_i is 0 for non-vulnerable statements and 1 for vulnerable statements. The label Y indicates whether the code snippet as a whole is vulnerable or not. Thus, we can determine the label of the entire code snippet as follows:

$$Y = \max\{y_1, \dots, y_n\} \quad (6)$$

where n is the number of statements in the code snippet.

By referring to equation 6, it becomes evident that the label of every statement within a function labeled as 0 will also be 0. However, there are two challenges in determining the statement labels within a function labeled as 1. Firstly, the vulnerable function contains only a few vulnerable statements, while the majority of statements inside the function are non-vulnerable. This raises the issue of how to assign labels to these statements. Secondly, even if we solve the problem of label assignment for the vulnerable function, we encounter another challenge with the label ratio. In non-vulnerable functions, the statement labels will be predominantly 0, and the same applies to most of the statement labels in the vulnerable function. The ratio of pseudo-positive/negative labels becomes much smaller than the ratio of the original positive/negative labels in the dataset. The sample ratio imbalance will lead the model to favor predicting functions as non-vulnerable, impacting its overall performance [29].

We address the first issue based on the assumption that the non-vulnerable statements exhibit a distinct pattern that differs from the vulnerable statement. This pattern can be detected and distinguished by the classifier model. Specifically, we sort the statements within the function in descending order, based on their previous classification prediction scores. Next, we generate pseudo labels for the top k statements, assigning them the same label as the entire function for training purposes. The value of k can be determined as the average number of vulnerable statements, as indicated by the dataset statistics. It is important to note that the pseudo labels for the vulnerable statements are not accurate at the beginning of the training. However, the pseudo labels for the non-vulnerable statements must be accurate because all the statements in the non-vulnerable statements are non-vulnerable. As negative samples are incorporated into the training process, the prediction scores for statements that are semantically similar to non-vulnerable statements will progressively decrease. The pseudo labels for vulnerable statements will gradually become more relatively accurate as training progresses [5].

To address the second issue, we set the training objective exclusively for the top-k statement within the given function, regardless of whether the function is vulnerable or non-vulnerable. This operation can effectively address the issue of imbalanced sample ratios resulting from the conversion of pseudo labels. From the experiment results, we find that the selection of hyperparameter k will have a slight effect on the overall performance, which will be discussed in Section 4.2

In summary, the loss function used in WAVES is cross-entropy loss, which is defined as follows:

$$loss = \frac{1}{N \cdot k} \sum_i \sum_j^k -[Y_i \log(p_{ij}) + (1 - Y_i) \log(1 - p_{ij})] \quad (7)$$

where N is the number of the function, Y_i is the label for the function i , p_{ij} is the vulnerability probability for the j -th statement in the function i , and k is the pre-defined parameter.

2.5 Model Inference

During the inference stage, WAVES does not provide a direct prediction of the vulnerability of the entire function. Instead, WAVES focuses on predicting vulnerable conditions for each statement within the function. The determination of the function-level prediction relies on the results obtained at the statement level. If at least one statement within the function is predicted as vulnerable, the function is considered vulnerable as well. Conversely, if no statements are predicted as vulnerable, the function is deemed non-vulnerable.

Two methods for localizing vulnerabilities are employed: absolute label prediction and relative scores ranking. In the absolute label prediction method, the model identifies and reports only the statements it predicts as vulnerable. On the other hand, the relative scores ranking method involves evaluating the statements within the function based on their prediction scores and sorting them in descending order. The top k statements are then selected as potential candidates for vulnerable statements, which are presented to the users. However, it is important to note that the absolute label prediction might not be accurate enough since there is no ground-truth label provided for training. Therefore, we recommend utilizing the relative scores ranking method as the preferred approach for vulnerability localization. The performance of both methods will be further discussed in the following section.

3 EXPERIMENTAL SETUP

In this section, we provide an overview of the statistics information for the dataset used in our study, the steps taken for data pre-processing, the baseline models employed, the evaluation metrics utilized, and the implementation details concerning both our proposed tool and the other baseline models included in our experiment.

3.1 Data Pre-processing

In our experiment, we evaluate the performance of vulnerability detection and localization using four datasets: FFM-Peg+Qemu [52], Reveal[6], Fan et al. [14], and CVEfixes [4]. The FFM-Peg+Qemu dataset, collected by Devign, comprises data from two open-source C projects and has been labeled by experts. It consists of approximately 10,000 vulnerable functions and 12,000 non-vulnerable functions. The Reveal dataset is obtained from Linux Debian Kernel and Chromium, containing about 2,000 vulnerable functions and 20,000 non-vulnerable functions. Fan et al. dataset, a C/C++ dataset, is gathered from over 300 open-source GitHub projects, covering 91 different Common Vulnerabilities and Exposures (CVE) databases from 2002 to 2019. This dataset includes around 10,000 vulnerable functions and 177,000 non-vulnerable functions. The CVEfixes dataset compiles CVE records from the National Vulnerability Database (NVD). Its initial

Table 1. Statistics of dataset.

Dataset		Vul function	Non-vul function	Avg stat num	Avg vul stat num
Fan <i>et al.</i>	Training	4,993	142,188	20.12	3.03
	Validation	624	17,774	20.69	3.27
	Testing	626	17,774	20.41	3.28
CVEfixes	Training	4,437	7,699	72.14	7.69
	Validation	554	964	73.14	7.60
	Testing	588	960	71.79	5.97
Reveal	Training	801	10,371	N/A	N/A
	Validation	98	1,256	N/A	N/A
	Testing	104	1,296	N/A	N/A
FFMPeg+Qemu	Training	7,078	8,526	N/A	N/A
	Validation	887	1,058	N/A	N/A
	Testing	879	1,024	N/A	N/A

release covers 5,365 CVEs from 1,754 projects as of June 9, 2021. For our experiments, we extract C/C++-related CVEs from this dataset.

The FFMpeg+Qemu and Reveal datasets only provide labels indicating whether a given function is vulnerable or not. Therefore, we solely assess the performance of function-level vulnerability detection using these two datasets. In contrast, Fan et al. [14] and CVEfixes [4] not only offer function-level labels but also provide the fixed version of the vulnerable function. This allows us to pinpoint the vulnerable statements by comparing the function before and after fixing. Therefore, we can evaluate both function-level vulnerability detection and statement-level vulnerability localization on this dataset.

We have imposed a length limitation on the input token sequence in order to accommodate the fixed-length input requirement of Transformer. Any token exceeding the maximum input length is discarded. However, the vulnerable statements in the code snippets may be contained in the discarded tokens, which means that the input statements are not vulnerable although the label for the entire function is vulnerable. To address this label conflict, we remove code snippets whose function-level label is vulnerable, but do not contain any vulnerable statements in the input to our model. Additionally, different baseline models used in our experiments necessitate different data pre-processing tools. Some of the data in our dataset cannot be processed correctly by all of these tools. In the interest of experimental fairness, we only retain the data that can be processed by all data pre-processing tools. To maximize the length of code that can be processed by the model, we removed all comments, ensuring that the input consists solely of executable code.

Within Table 1, "Vul function" denotes the number of vulnerable functions in the dataset, while "Non-vul function" represents the number of non-vulnerable functions. "Avg stat num" indicates the average number of statements in a single function within the dataset, and "Avg vul stat num" signifies the average number of vulnerable statements in a single function. As previously mentioned, Reveal and FFMpeg+Qemu do not provide information regarding vulnerable statements, thus preventing us from offering statistics on the average statement number and average vulnerable statement number for these two datasets.

3.2 Implementation Details

In our proposed WAVES, we set the number of encoder layers to 12, the number of attention headers to 12, and the hidden size to 768. The batch size and learning rate were set to 16 and $2e-5$, respectively. Our model supports a maximum input token length of 512. As the hyperparameter top-k is sensitive to the average number of vulnerable statements in the target dataset, which can vary between datasets, we experimented with values of 1, 3, and 5 for top-k and selected the best performance for each dataset. For optimization, we utilized the AdamW [30] optimizer. We set a maximum of 50 epochs for the training with 10-steps patience for early stopping.

We replicated all the baselines, except for Devign, using publicly released source code and adopted the same hyperparameter settings as described in their original paper. For Devign, since they did not make their code public, we reproduced it based on the code provided by Chakraborty et al. [6]. In the case of the baseline model called IVDetect, it requires the training dataset to have an equal ratio of vulnerable and non-vulnerable functions. Since none of the datasets we used had this ratio, we retained all the vulnerable functions and randomly selected an equal number of non-vulnerable functions from each training dataset to create a new training dataset for IVDetect. There are two version of the baseline model named LineVul, which are LineVul with pre-training and LineVul without pre-training. We observed that LineVul with pre-training performed exceptionally well on the dataset of Fan et al, while there was no significant difference between LineVul with and without pre-training on the other two datasets. This led us to suspect the presence of a data leakage problem specifically in the Fan et al dataset so that we exclude the pre-training model from our experiments. All the models were trained on a server equipped with NVIDIA A100-SXM4. The training process for WAVES consumed approximately 10 GPU hours.

3.3 Baselines

We compare WAVES with six state-of-the-art vulnerability detection methods, including two token-based methods [27, 28], three structure-based methods [6, 26, 52], and one unsupervised statement-level detection method [16]. Here, we provide a brief description of these baseline methods:

(1) **VulDeePecker** [28]: VulDeePecker extracts code gadgets from the given code snippet, which are several lines of code that are semantically related to each other, and adopted the bidirectional LSTM-based neural network with an attention mechanism for vulnerability detection.

(2) **SySeVR** [27]: SySeVR extracts the vulnerability syntax characteristics (SyVCs) from the given function at first and then transforms these SyVCs into semantics-based vulnerability candidates (SeVCs) which contain the statements related to the given SyVCs via the program slicing technique. Finally, a bidirectional recurrent neural network is employed to encode these SeVCs into vectors, facilitating the detection of vulnerable code snippets.

(3) **Devign** [52]: Devign extracts the information of abstract syntax tree (AST), control flow graph (CFG), data flow graph (DFG), and code token sequence from the given function to construct the graph which can represent the given functions and generate the embedding vector for each node inside the graph. Subsequently, the graph is inputted into a Gated Graph Neural Network (GGNN) for classification training.

(4) **Reveal** [6]: Reveal extracts the information of Code Property Graph (CPG) from the given function to construct the representation graph and adopts the technique of Word2Vec to generate the embedding vector for each node. The resulting graph is then inputted into GGNN, where all the vectors from the representation graph are combined into a single vector. This fused vector serves as the representation for the entire graph, facilitating vulnerability detection.

(5) IVDetect [26]: IVDetect is a code analysis tool that divides the code into multiple statements and extracts various features from each statement. These features encompass sub-token sequences, AST sub-trees, variable names, variable types, data dependency context, and control dependency context. To capture these features, they are embedded into representation vectors. Subsequently, these vectors are combined into a unified representation vector for each statement using an attention-based bidirectional GRU. These statement-level representation vectors serve as node embedding features, which are then fed into a Graph Convolutional Network (GCN) to acquire a comprehensive graph representation for detection purposes.

(6) LineVul [16]: LineVul adopts the Byte Pair Encoder (BPE) technique to tokenize the given code into a sub-token sequence and utilize a 12-layer Transformer based model for vulnerability detection. Not only predicting the function-level vulnerability, LineVul can also localize the vulnerable statement by calculating the attention scores for each sub-token.

3.4 Evaluation Metrics

In our experiment, we adopts four metrics, which are ACC , P , R , and $F1$, to evaluate the performance of all the models in function-level vulnerability detection. Additionally, we utilized nine metric, which are $Top - 1$, $Top - 5$, $Top - 10$, MFR , MAR , IFA , P , R , $F1$, to evaluate the performance of all the models in statement-level vulnerability localization.

The metric employed to determine the accuracy of the model is denoted as Acc . It quantifies the ratio of accurately classified samples to the total number of samples. The definition of Acc is presented below:

$$Acc = \frac{S_c}{S} \quad (8)$$

where S_c represents the number of samples correctly labeled by the model, while S denotes the total number of samples. A higher value of Acc indicates a better performance of the model.

P is the metric used to assess the accuracy of the model's detection of vulnerable samples. The definition of P is provided below:

$$P = \frac{TP}{TP + FP} \quad (9)$$

where TP represents the count of samples where both the label and the model's prediction are true, while FP refers to the count of samples where the label is true but the model's prediction is incorrect. A higher value for P signifies improved performance of the model.

R is a metric used to assess the percentage of vulnerable samples correctly detected out of all the vulnerable samples predicted by the model. The specific definition of this metric is provided below:

$$R = \frac{TP}{TP + FN} \quad (10)$$

where TP represents the count of samples with true labels that the model correctly predicts, while FN represents the count of samples with false labels that the model incorrectly predicts. A higher value of R signifies a superior performance of the model.

$F1$ is a metric that represents the harmonic mean of precision and recall. It is commonly employed to assess a model's performance by taking into account both precision and recall. The formula for calculating $F1$ is as follows:

$$F1 = 2 \times \frac{P \times R}{P + R} \quad (11)$$

where P represents the precision of the model, and R denotes the number of samples for which the label is false, but the model provides an incorrect prediction. A higher $F1$ score indicates superior performance of the model.

$Top - k$ is a metric used to assess the model's ability to identify vulnerable statements among the top k results it returns. The definition of $Top - k$ is as follows:

$$Top - k = \frac{1}{|S_v|} \sum_{s=1}^{S_v} \delta(FRank_s \leq k) \quad (12)$$

where S_v represents the count of vulnerable functions, and $FRank_s$ denote the ranking assigned to the first vulnerable statement in the statement set. A higher value for $Top - k$ signifies improved performance in vulnerability localization.

MFR (Mean First Ranking) is calculated as the average of the rankings assigned to the first vulnerable statement among the returned statements. The formula for calculating MFR is provided below:

$$MFR = \frac{1}{|S_v|} \sum_{s=1}^{S_v} FRank_s \quad (13)$$

A lower value of MFR indicates superior performance in vulnerability localization.

MAR (Mean Average Ranking) is calculated as the average ranking across all vulnerable statements present in the returned statements. The formula for MAR is provided below:

$$MAR = \frac{1}{|S_v|} \sum_{s=1}^{S_v} \frac{1}{|N|} \sum_{i=1}^N Rank_{si} \quad (14)$$

where $Rank_{si}$ represents the ranking of the i -th vulnerable statement within the returned statements of the s -th vulnerable function. A lower MAR value indicates a superior performance in terms of vulnerability localization.

IFA (Initial False Alarm) is a metric that quantifies the number of statements that are erroneously predicted as vulnerable by the models before correctly identifying the first vulnerable statement. The definition of this metric is provided below:

$$IFA = \frac{1}{|S_v|} \sum_{s=1}^{S_v} (FRank_s - 1) \quad (15)$$

A lower value of IFA indicates superior performance in vulnerability localization.

4 EXPERIMENTAL RESULTS

In this section, we firstly presents the experimental results and assessing the performance of WAVES in terms of function-level vulnerability detection and statement-level vulnerability localization. Secondly, we evaluate the impact of Top-K statement selection on the overall performance. Thirdly, we investigate the contribution of each channel to the overall performance. Fourthly, we examine the influence of data size on both function-level vulnerability detection and statement-level vulnerability localization abilities. Fifthly, we evaluate the ability of WAVES to detect different types of vulnerability. Lastly, we conduct a case study examining both successful and unsuccessful cases.

4.1 Comparison on function-level vulnerability detection and statement-level vulnerability localization

Table 2 illustrates the comparison results of function-level vulnerability detection performance. It is worth noting that WAVES_{Select} represents a variant of our proposed approach, and the discussion regarding it will be included in Section 5.

Table 2. Comparison results on function-level vulnerability. The best results are highlighted in **bold** font.

Model	Fan <i>et al.</i>				Reveal				FFMPeg+Qemu				CVEfixes			
	Acc	P	R	F1	Acc	P	R	F1	Acc	P	R	F1	Acc	P	R	F1
VulDeePecker	0.913	0.155	0.146	0.150	0.763	0.211	0.131	0.162	0.496	0.461	0.326	0.381	0.598	0.330	0.096	0.148
SySeVR	0.904	0.129	0.194	0.155	0.743	0.401	0.249	0.307	0.479	0.461	0.588	0.517	0.582	0.361	0.127	0.188
Devign	0.957	0.257	0.143	0.184	0.875	0.316	0.367	0.339	0.569	0.525	0.647	0.580	0.615	0.464	0.076	0.131
Reveal	0.928	0.270	0.661	0.383	0.818	0.316	0.611	0.416	0.611	0.555	0.707	0.622	0.513	0.412	0.656	0.506
IVDetect	0.696	0.073	0.600	0.130	0.808	0.276	0.556	0.369	0.573	0.524	0.576	0.548	0.601	0.352	0.085	0.137
LineVul	0.972	0.632	0.436	0.516	0.847	0.248	0.519	0.335	0.541	0.496	0.909	0.642	0.496	0.402	0.672	0.503
WAVES	0.977	0.724	0.522	0.607	0.922	0.471	0.394	0.429	0.589	0.530	0.812	0.641	0.461	0.397	0.801	0.530
WAVES _{Select}	0.975	0.678	0.474	0.558	0.869	0.294	0.549	0.383	0.570	0.520	0.652	0.578	0.488	0.391	0.621	0.480

Table 3. Comparison results on statement-level vulnerability localization. The best results are highlighted in **bold** font.

Dataset	Model	Acc	P	R	F1	MAR	MFR	IFA	Top-1	Top-3	Top-5
Fan <i>et al.</i>	LineVul	N/A	N/A	N/A	N/A	9.49	7.17	6.17	0.005	0.252	0.375
	WAVES	0.983	0.183	0.338	0.237	9.08	6.46	5.46	0.283	0.484	0.609
	WAVES _{Select}	0.981	0.185	0.404	0.253	9.15	6.63	5.63	0.243	0.452	0.572
CVEfixes	LineVul	N/A	N/A	N/A	N/A	9.26	5.62	4.62	0.221	0.469	0.585
	WAVES	0.614	0.078	0.407	0.131	9.13	5.40	4.40	0.322	0.556	0.650
	WAVES _{Select}	0.647	0.076	0.352	0.125	9.08	5.58	4.58	0.335	0.551	0.669

In the datasets of Fan et al. and Reveal, where there is an imbalance in the proportion of positive and negative examples, the F1 metric holds more significance compared to other metrics. The results reveal that WAVES outperforms other approaches and achieves state-of-the-art performance in terms of F1 on all the Fan et al., Reveal and CVEfixes datasets. Notably, WAVES demonstrates a relative improvement of 17.6%, 4.7%, and 3.1% in F1 on the Fan et al., CVEfixes, and Reveal datasets, respectively. Regarding the FFMPeg+Qemu dataset, while WAVES does not surpass all the baselines, its performance is closely aligned with the state-of-the-art baseline, with only a 0.2% difference in F1. These findings demonstrate that WAVES can attain performance comparable to the current state-of-the-art baselines for function-level vulnerability detection.

Table 3 presents the results of statement-level vulnerability localization performance for our proposed approach and the baselines. Due to the availability of sentence-level annotation labels in the dataset by Fan et al. and CVEfixes, we display the experimental results for these two datasets. To evaluate accuracy, precision, recall, and F1, we employ the first method outlined in Section 2.5, which utilizes absolute label prediction. For the remaining metrics, we utilize the second method introduced in Section 2.5, employing relative scores to ensure a fair comparison between WAVES and the baseline.

LineVul employs the attention score for each statement to estimate the likelihood of a statement being vulnerable, rather than directly determining its vulnerability. The metrics such as accuracy, precision, recall, and F1 are not applicable to this baseline. Instead, we present the results of our proposed WAVES. A comparison between the results in Table 2 and Table 3 reveals that the ability of WAVES to detect vulnerabilities at the statement level is inferior to its ability to detect vulnerabilities at the function level. Therefore, relying solely on the statement-level predictions from WAVES may not be advisable. It can be easily understood that it is quite hard to localize the vulnerable statement since there is no explicit ground-truth label for the training. Moreover, the recall of WAVES is significantly higher than its

Table 4. Results of the function-level vulnerability detection performance comparison with different Top-K selection. The best results among the three variants of WAVES are highlighted in **bold** font.

Model	Fan <i>et al.</i>				Reveal				FFMPeg+Qemu				CVEfixes			
	Acc	P	R	F1	Acc	P	R	F1	Acc	P	R	F1	Acc	P	R	F1
WAVES _{top-1}	0.976	0.724	0.481	0.578	0.913	0.415	0.423	0.419	0.589	0.530	0.812	0.641	0.515	0.396	0.527	0.452
WAVES _{top-3}	0.977	0.724	0.522	0.607	0.894	0.365	0.471	0.397	0.575	0.519	0.824	0.637	0.461	0.397	0.801	0.530
WAVES _{top-5}	0.974	0.653	0.524	0.582	0.922	0.471	0.394	0.429	0.576	0.520	0.824	0.638	0.446	0.391	0.827	0.531

Table 5. Results of the statement-level vulnerability localization performance comparison with different Top-K selection. The best results among the three variants of WAVES are highlighted in **bold** font.

Dataset	Model	Acc	P	R	F1	MAR	MFR	IFA	Top-1	Top-3	Top-5
Fan <i>et al.</i>	WAVES _{top-1}	0.987	0.155	0.142	0.148	9.24	6.53	5.53	0.219	0.446	0.577
	WAVES _{top-3}	0.983	0.183	0.338	0.237	9.08	6.46	5.46	0.283	0.484	0.609
	WAVES _{top-5}	0.979	0.164	0.380	0.229	9.02	6.43	5.43	0.294	0.497	0.605
CVEfixes	WAVES _{top-1}	0.863	0.115	0.137	0.125	8.99	4.34	3.34	0.406	0.631	0.733
	WAVES _{top-3}	0.614	0.078	0.407	0.131	9.13	5.40	4.40	0.322	0.556	0.650
	WAVES _{top-5}	0.478	0.073	0.534	0.128	9.48	6.02	5.02	0.303	0.488	0.621

precision according to the results from Table 3, indicating that WAVES tends to predict statements as vulnerable at the cost of a higher false alarm rate. Comparing the performance of WAVES and the baselines on the other metrics in Table 3, we observe that our proposed WAVES achieves state-of-the-art performance in statement-level vulnerability localization across all metrics. Notably, there is a substantial improvement in the Top-1 metric and significant improvements in the Top-3 and Top-5 metrics. Unlike previous approaches that lack supervised signals in the attention score, our mechanism enhances vulnerability localization by providing pseudo labels during training, resulting in improved localization ability. Furthermore, the accuracy of the top 5 predictions from WAVES is considerably higher than the F1 score, suggesting that our proposed WAVES can effectively notify users about vulnerable statements by ranking their relative scores when the target function is predicted to be vulnerable.

In conclusion, the comprehensive experiment results shows that WAVES can achieve a comparable function-level vulnerability detection performance and the state-of-the-art statement-level vulnerability localization performance compared to previous baselines, which demonstrates the effectiveness of our proposed WAVES.

4.2 Impact of the top-k statement selection on the performance of WAVES

Table 4 presents the impact of the Top-K hyperparameter on the performance of function-level vulnerability detection models. It is evident that there is no universally optimal fixed value for top-k that guarantees optimal performance across all datasets. This is primarily due to the variation in the average number of vulnerable statements within functions in each dataset. Our proposed WAVES employs pseudo statement-level labels as supervised signals for vulnerability localization during training. However, if the predefined top-k hyperparameter does not align with the actual number of vulnerable statements in a function, incorrect pseudo labels may be assigned. Setting a larger or smaller value of k can result in misclassifying non-vulnerable statements as vulnerable or missing some vulnerable statements, introducing noise into the model and adversely affecting performance. The results from Section 1 reveals that the average number of vulnerable statements in the Fan *et al.* dataset is approximately 3, which explains why the best performance is achieved

Table 6. Results of the function-level vulnerability detection performance comparison with different channels. The best results are highlighted in **bold** font.

Model	Fan <i>et al.</i>				Reveal				FFMPeg+Qemu				CVEfixes			
	Acc	P	R	F1	Acc	P	R	F1	Acc	P	R	F1	Acc	P	R	F1
WAVES _{max}	0.976	0.721	0.500	0.591	0.933	0.574	0.375	0.453	0.567	0.513	0.816	0.630	0.476	0.397	0.735	0.516
WAVES _{mean}	0.973	0.660	0.446	0.532	0.922	0.470	0.385	0.423	0.560	0.509	0.752	0.608	0.475	0.395	0.718	0.509
WAVES	0.977	0.724	0.522	0.607	0.922	0.471	0.394	0.429	0.589	0.530	0.812	0.641	0.461	0.397	0.801	0.530

Table 7. Results of the statement-level vulnerability localization performance comparison with different channels. The best results are highlighted in **bold** font.

Dataset	Model	Acc	P	R	F1	MAR	MFR	IFA	Top-1	Top-3	Top-5
Fan <i>et al.</i>	WAVES _{max}	0.984	0.188	0.299	0.231	9.17	6.66	5.66	0.268	0.481	0.617
	WAVES _{mean}	0.977	0.155	0.416	0.226	9.78	7.27	6.27	0.224	0.443	0.586
	WAVES	0.983	0.183	0.338	0.237	9.08	6.46	5.46	0.283	0.484	0.609
CVEfixes	WAVES _{max}	0.688	0.076	0.303	0.122	9.88	6.41	5.41	0.291	0.478	0.579
	WAVES _{mean}	0.641	0.078	0.368	0.128	9.02	5.44	4.44	0.326	0.545	0.682
	WAVES	0.614	0.078	0.407	0.131	9.13	5.40	4.40	0.322	0.556	0.650

when k is set to 3 in this dataset. Similarly, the model achieves the best performance when k is set to 5 on the CVEfixes dataset, where the average number of vulnerable statements exceeds 5.

Table 5 illustrates how the hyperparameter of Top-K influences the model performance of statement-level vulnerability localization. Before analyzing the experimental results, we must clarify that the accuracy metric in this table is meaningless due to data imbalance. All statements within non-vulnerable functions are non-vulnerable, and only a few statements within functions are vulnerable, with the rest also being non-vulnerable. Consequently, the number of non-vulnerable statements is much larger than the number of vulnerable ones, making the accuracy metric unreliable, as the model can achieve high accuracy simply by classifying all statements as non-vulnerable. We observed an interesting phenomenon: different hyperparameter values for top K lead to varying tendencies in metrics for absolute label prediction (accuracy, precision, recall, and F1) and relative scoring (MFR, MAR, IFA, Top-1, Top-3, and Top-5), as described in Section 2.5. For absolute label prediction, the recall ratio increases as the k value increases. A higher k means that more statements are labeled as vulnerable during training, making the model more inclined to predict statements as vulnerable, thereby improving recall. However, a higher k negatively impacts precision, creating a trade-off when selecting an appropriate k value for absolute label prediction.

In contrast, the impact of k on relative scoring is more complex, as the performance trend varies across different datasets. Specifically, the model performs best on the Fan *et al.* dataset when k is set to 5, while it achieves better results on the CVEfixes dataset when k is set to 1. This discrepancy arises because statement-level labels are derived from function-level labels, and the accuracy of these generated pseudo labels is not guaranteed. Consequently, this uncertainty affects the final performance of relative score prediction.

In conclusion, the comprehensive experimental results demonstrate that the choice of top-k significantly impacts the performance of WAVES in both function-level vulnerability detection and statement-level vulnerability localization.

4.3 Impact of different channels on the performance of WAVES

In this experiment, we conducted an analysis of the channels utilized in our model to determine their contribution to the performance of WAVES. The results of the function-level vulnerability detection experiment using different channels are presented in Table 6. It is observed that the model combining max pooling and mean pooling achieves the best performance across all datasets, except for the Reveal dataset. This outcome highlights the effectiveness of fusing max pooling and mean pooling. There are two potential explanations for why the model with only max pooling performs best in terms of the F1 metric in the dataset of Reveal. Firstly, the size of the training data could be a factor. Since the Reveal training dataset contains only around 800 vulnerable functions, the training of the model becomes unstable. As WAVES has a more complex structure compared to WAVES_{max}, overfitting occurs. Secondly, the specific vulnerability type prevalent in the Reveal dataset might play a role. As explained in Section 2.3, the design of max pooling and mean pooling aims to capture features associated with different vulnerability types. It is possible that most vulnerabilities in the Reveal dataset align closely with the vulnerability type effectively captured by max pooling. Consequently, mean pooling may have limited contribution or even adversely affect the overall model performance. Another noteworthy finding is that WAVES_{max} performs very similarly to WAVES, while WAVES_{mean} exhibits considerably worse performance than WAVES_{max}. This discrepancy may be attributed to the fact that many vulnerabilities are related to specific keywords within statements, and these features are effectively captured by the max pooling mechanism. Listing 2 presents an illustrative example. WAVES effectively detects and localizes use-after-free vulnerabilities by identifying the repeated occurrence of keywords like ‘buffer,’ which has already been deleted in a previous statement. In contrast, mean pooling averages the high-dimensional representation vectors for each token in the statement, potentially diluting these keyword features and reducing the model’s performance.

```

1  #include <iostream>
2  #include <fstream>
3
4  void processFile(const std::string& filename) {
5      char* buffer = new char[1024];
6      std::ifstream file(filename, std::ios::binary);
7
8      if (!file) {
9          std::cerr << "Failed to open the file." << std::endl;
10         delete[] buffer;
11         return;
12     }
13
14     file.read(buffer, 1024);
15     std::cout << "File content read successfully." << std::endl;
16
17     delete[] buffer;
18     std::cout << "Buffer memory freed." << std::endl;
19
20     std::cout << "First byte of file content (post-delete): " << buffer[0] << std::endl;
21 }

```

Listing 2. An example for vulnerability related to specific keywords within statements.

Table 7 presents the experimental results for statement-level vulnerability localization using different channels. The results demonstrate that our WAVES performs the best across most metrics, showcasing the effectiveness of the fusion

Table 8. Comparison results on function-level vulnerability detection and statement-level vulnerability localization with different sizes of training data in the dataset of Fan et al.. The best results are highlighted in **bold** font.

Dataset	Model	Function-level Detection				Statement-level Localization									
		Acc	P	R	F1	Acc	P	R	F1	MAR	MFR	IFA	Top-1	Top-3	Top-5
Fan et. al	WAVES _{10%}	0.969	0.580	0.265	0.364	0.991	0.074	0.014	0.024	9.33	6.73	5.73	0.235	0.444	0.585
	WAVES _{20%}	0.971	0.610	0.398	0.482	0.987	0.137	0.118	0.127	9.07	6.48	5.48	0.227	0.431	0.583
	WAVES _{50%}	0.971	0.591	0.521	0.554	0.985	0.170	0.217	0.191	9.33	6.64	5.64	0.260	0.460	0.593
	WAVES	0.977	0.724	0.522	0.607	0.983	0.183	0.338	0.237	9.08	6.46	5.46	0.283	0.484	0.609
CVEfixes	WAVES _{10%}	0.596	0.422	0.170	0.242	0.920	0.075	0.011	0.018	9.80	8.51	7.51	0.179	0.358	0.471
	WAVES _{20%}	0.534	0.380	0.357	0.368	0.872	0.079	0.073	0.076	9.72	7.66	6.66	0.191	0.389	0.520
	WAVES _{50%}	0.543	0.400	0.403	0.401	0.851	0.102	0.139	0.118	9.13	5.43	4.43	0.328	0.568	0.661
	WAVES	0.461	0.397	0.801	0.530	0.614	0.078	0.407	0.131	9.13	5.40	4.40	0.322	0.556	0.650

of max pooling and mean pooling. Unlike the conclusion drawn from Table 6, the performance of max channel and mean channel in statement-level vulnerability localization varies across different datasets. This discrepancy may be attributed to the differences in CVE types among the datasets.

In conclusion, the comprehensive experimental results validate the effectiveness of both max pooling and mean pooling in the proposed WAVES. This combination significantly enhances the ability of function-level vulnerability detection and statement-level vulnerability localization.

4.4 The influence of the training data size to the performance of WAVES

In this experiment, we assess the impact of training data size on the performance of our proposed WAVES for function-level vulnerability detection and statement-level vulnerability localization. Since only the dataset of Fan et al. includes statement-level labels, we exclusively evaluate our WAVES using this dataset. To thoroughly examine the influence of training data size on model performance, we randomly select 10%, 20%, 50%, and 100% of the data from the training dataset to construct new training datasets. Table 8 presents the experimental results for function-level vulnerability detection and statement-level vulnerability localization across different training data sizes. As expected, the model's performance in function-level vulnerability detection consistently improves as the training data size increases. Interestingly, even though the data lacks statement-level labels, the model's performance in statement-level vulnerability localization also improves as the training data size increases. These findings demonstrate that our proposed WAVES becomes more proficient at locating vulnerability statements as the training data size grows, even without explicit annotations. However, the performance improvement of WAVES on absolute label prediction and relative scores, as introduced in Section 2.5, varies with the increase in training data size. Specifically, the performance improvement in absolute label prediction, measured by accuracy, precision, recall, and F1, is substantial with larger training data sizes. Conversely, WAVES maintains most of its performance in relative scores metrics even with limited training data, and the performance improvement in relative scores is not as rapid as the improvement in absolute label prediction with increasing training data size. In the CVEfixes dataset, the performance of relative score predictions using 50% of the data is nearly identical to that achieved with 100% of the data.

In summary, increasing the size of the training data can enhance the performance of function-level vulnerability detection and statement-level vulnerability localization, even without any additional information about the vulnerable statements in the data.

Table 9. Detection results for different CWE vulnerabilities with our proposed WAVES.

CWE-ID	Description	Rank	TPR	Proportion
CWE-787	Out-of-bounds Write	1	50.0%	7/14
CWE-416	Use After Free	4	61.5%	8/13
CWE-20	Improper Input Validation	6	56.6%	61/108
CWE-125	Out-of-bounds Read	7	54.2%	13/24
CWE-476	NULL Pointer Dereference	12	55.6%	5/9
CWE-190	Integer Overflow or Wraparound	14	77.8%	14/18
CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	17	56.0%	70/125
CWE-362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	21	52.4%	11/21
CWE-284	Improper Access Control	N/A	37.5%	3/8
CWE-189	Numeric Errors	N/A	52.6%	10/19
CWE-732	Incorrect Permission Assignment for Critical Resource	N/A	57.1%	4/7
CWE-254	7PK - Security Features	N/A	44.4%	4/9
CWE-200	Exposure of Sensitive Information to an Unauthorized Actor	N/A	42.9%	12/28
CWE-415	Double Free	N/A	71.4%	5/7
CWE-399	Resource Management Errors	N/A	48.7%	19/39
Total			54.8%	246/449

4.5 The detection ability of WAVES for different types of CWE vulnerabilities

Table 9 presents the detection results of our proposed WAVES for different types of vulnerabilities from CWE. To ensure meaningful data analysis, we filtered out vulnerability types with a small number of occurrences and retained only those with at least five instances from the same test dataset, which was also used in previous experiments. The CWE has released the 2023 Top 25 Most Dangerous Software Weaknesses on their website [1]. The rank metric in the table indicates the CWE's ranking in their list. In cases where the vulnerability types in our test dataset are not included in the Top 25 list, we denote them as "N/A" in the rank metric. Additionally, the TPR metric in the table represents the true positive rate, indicating the percentage of vulnerabilities successfully detected by the model.

The results reveal an interesting finding: the detection ability of WAVES does not vary significantly between different types of CWE vulnerability, with a TPR of approximately 56% for most CWE types. However, certain vulnerabilities, such as CWE-415 and CWE-284, exhibit notably high or low TPRs. It is important to note that the limited number of vulnerabilities makes it difficult to conclusively determine whether WAVES performs well or poorly in these specific types. Nevertheless, some exceptions stand out. Notably, WAVES demonstrates a strong detection ability for vulnerabilities classified as CWE-190, successfully identifying almost all instances of this type. Conversely, WAVES displays a poor detection ability for vulnerabilities categorized as CWE-200 and CWE-399, as it only identifies less than half of the vulnerabilities within these types. We believe that the accuracy of detecting different types of vulnerabilities may be influenced by the difficulty of identifying their patterns. Our findings indicate that the detection accuracy for use-after-free, integer overflow or wraparound, and double-free vulnerabilities is higher than for other types. These vulnerabilities have relatively fixed patterns, making them easier to detect by conventional static analysis methods. And our proposed approach is likely to detect these patterns more effectively as well.

In conclusion, WAVES demonstrates comparable detection abilities for most types of vulnerabilities. Nevertheless, it exhibits superior performance in detecting vulnerabilities related to Integer Overflow or Wraparound, while its effectiveness is relatively weaker in identifying vulnerabilities associated with Exposure of Sensitive Information to an Unauthorized Actor and Resource Management Errors.

4.6 Case Study

Listing 3 shows a successfully detected vulnerability associated with CWE-20, which is related to Improper Input Validation. In this code snippet, the `memcpy` function is invoked on lines 6 and 8. However, the input length for this function is not validated. If the length of the data pointed to by the `smac` or `alt_smac` pointer exceeds the size of `av_smac_size` or `alt_av_smac_size`, the `memcpy` function will copy data beyond the boundary of the target array, resulting in a buffer overflow. In this case, the lack of comprehensive verification of the input data's validity may lead to unpredictable system behavior or a system crash, especially when processing network packets. This case indicates that WAVES can capture the feature of missing input validation before the execution of the `memcpy` function during contextual learning and issue warnings about these two statements.

```

1  int ib_update_cm_av(struct ib_cm_id *id, const u8 *smac, const u8 *alt_smac)
2  {
3      struct cm_id_private *cm_id_priv;
4      cm_id_priv = container_of(id, struct cm_id_private, id);
5      if (smac != NULL)
6          memcpy(cm_id_priv->av.smac, smac, sizeof(cm_id_priv->av.smac));
7      if (alt_smac != NULL)
8          memcpy(cm_id_priv->alt_av.smac, alt_smac,
9                 sizeof(cm_id_priv->alt_av.smac));
10     return 0;
11 }

```

Listing 3. Successfully detected vulnerability associated with CWE-20

Listing 4 shows another successfully detected vulnerability associated with CWE-200, which relates to the exposure of sensitive information to unauthorized actors. In this code snippet, line 26 logs error information when `valid_path` does not exist. However, the user email address is included in the `valid_path` variable on lines 17 and 18, so user information will also be logged. This recording can expose sensitive user information, especially if the logs are accessible to unauthorized personnel. In this case, WAVES successfully captures sensitive tokens like the user email and detects the sensitive information logging behavior, issuing a warning about this vulnerability.

```

1  void WallpaperManagerBase::GetCustomWallpaperInternal(
2      const AccountId& account_id,
3      const WallpaperInfo& info,
4      const base::FilePath& wallpaper_path,
5      bool update_wallpaper,
6      const scoped_refptr<base::SingleThreadTaskRunner>& reply_task_runner,
7      MovableOnDestroyCallbackHolder on_finish,
8      base::WeakPtr<WallpaperManagerBase> weak_ptr) {
9      base::FilePath valid_path = wallpaper_path;
10     if (!base::PathExists(wallpaper_path)) {
11         valid_path = GetCustomWallpaperDir(kOriginalWallpaperSubDir);
12         valid_path = valid_path.Append(info.location);

```

```

13 }
14
15 if (!base::PathExists(valid_path)) {
16     LOG(ERROR) << "Failed to load custom wallpaper from its original fallback "
17         "file path: " << valid_path.value();
18     const std::string& old_path = account_id.GetUserEmail(); // Migrated
19     valid_path = GetCustomWallpaperPath(kOriginalWallpaperSubDir,
20         WallpaperFilesId::FromString(old_path),
21         info.location);
22 }
23
24 if (!base::PathExists(valid_path)) {
25     LOG(ERROR) << "Failed to load previously selected custom wallpaper. "
26         << "Fallback to default wallpaper. Expected wallpaper path: "
27         << wallpaper_path.value();
28     reply_task_runner->PostTask(
29         FROM_HERE,
30         base::Bind(&WallpaperManagerBase::DoSetDefaultWallpaper, weak_ptr,
31             account_id, base::Passed(std::move(on_finish))));
32 } else {
33     reply_task_runner->PostTask(
34         FROM_HERE, base::Bind(&WallpaperManagerBase::StartLoad, weak_ptr,
35             account_id, info, update_wallpaper, valid_path,
36             base::Passed(std::move(on_finish))));
37 }
38 }

```

Listing 4. Successfully detected vulnerability associated with CWE-200

4.7 Error Analysis

Despite WAVES's improved performance over previous vulnerability localization methods, certain cases remain undetected or mislocalized. We have selected two such cases for error analysis.

4.7.1 Missing Implementation of Invoked Function. Listing 5 is a failed detection example associated with CWE-787. This code snippet contains a potential Out-of-bounds Write issue, which allowed a remote attacker who had compromised the renderer process to perform an out-of-bounds memory write via a crafted HTML page. However, this vulnerability is related to the implementation of the invoked function `CopyMetafileDataToSharedMem`, and such implementation details are not included in the code snippet. The absence of this key information caused WAVES to fail in detecting this vulnerability.

```

1 bool PrintRenderFrameHelper::PrintPagesNative(blink::WebLocalFrame* frame,
2                                             int page_count) {
3     const PrintMsg_PrintPages_Params& params = *print_pages_params_;
4     const PrintMsg_Print_Params& print_params = params.params;
5
6     std::vector<int> printed_pages = GetPrintedPages(params, page_count);
7     if (printed_pages.empty())
8         return false;
9
10    PdfMetafileSkia metafile(print_params.printed_doc_type);
11    CHECK(metafile.Init());

```

```

12
13 PrintHostMsg_DidPrintDocument_Params page_params;
14 PrintPageInternal(print_params, printed_pages[0], page_count, frame,
15                  &metafile, &page_params.page_size,
16                  &page_params.content_area);
17 for (size_t i = 1; i < printed_pages.size(); ++i) {
18     PrintPageInternal(print_params, printed_pages[i], page_count, frame,
19                     &metafile, nullptr, nullptr);
20 }
21
22 FinishFramePrinting();
23
24 metafile.FinishDocument();
25
26 if (!CopyMetafileDataToSharedMem(metafile,
27                                  &page_params.metafile_data_handle)) {
28     return false;
29 }
30
31 page_params.data_size = metafile.GetDataSize();
32 page_params.document_cookie = print_params.document_cookie;
33 #if defined(OS_WIN)
34 page_params.physical_offsets = printer_printable_area_.origin();
35 #endif
36 Send(new PrintHostMsg_DidPrintDocument(routing_id(), page_params));
37 return true;
38 }

```

Listing 5. Case of failed vulnerability detection associated with CWE-787

4.7.2 Missing External Knowledge. Listing 6 illustrates another type of detection failure associated with CWE-241. CWE-241 pertains to the improper handling of unexpected data types, where a product does not correctly handle an element that is not of the expected type. In this code snippet, `raw_headers.push_back('\0')` is used to terminate each line, with `\0\0` appended as the terminator at the end. This approach does not comply with the HTTP protocol specification and can cause the code parsing HTTP header information to misinterpret and misprocess the headers. Sometimes, HTTP header information may include user input data. If this data contains the `\0` character, using `\0` as the terminator might lead to injection attacks, as the parser may prematurely terminate parsing, resulting in incomplete or incorrect parsing outcomes. However, without the specific knowledge about the HTTP protocol, WAVES failed to recognize this as a vulnerability, leading to the detection failure.

```

1 std::string HttpUtil::AssembleRawHeaders(const char* input_begin,
2                                          int input_len) {
3     std::string raw_headers;
4     raw_headers.reserve(input_len);
5
6     const char* input_end = input_begin + input_len;
7
8     int status_begin_offset = LocateStartOfStatusLine(input_begin, input_len);
9     if (status_begin_offset != -1)
10         input_begin += status_begin_offset;
11

```

```

12  const char* status_line_end = FindStatusLineEnd(input_begin, input_end);
13  raw_headers.append(input_begin, status_line_end);
14
15
16  CStringTokenizer lines(status_line_end, input_end, "\r\n");
17
18  bool prev_line_continuable = false;
19
20  while (lines.GetNext()) {
21      const char* line_begin = lines.token_begin();
22      const char* line_end = lines.token_end();
23
24      if (prev_line_continuable && IsLWS(*line_begin)) {
25          raw_headers.push_back(' ');
26          raw_headers.append(FindFirstNonLWS(line_begin, line_end), line_end);
27      } else {
28          raw_headers.push_back('\0');
29
30          raw_headers.append(line_begin, line_end);
31
32          prev_line_continuable = IsLineSegmentContinuable(line_begin, line_end);
33      }
34  }
35
36  raw_headers.append("\0\0", 2);
37  return raw_headers;
38  }

```

Listing 6. Case of failed vulnerability detection associated with CWE-241

5 DISCUSSION

In this section, we first explore the impact of different channel fusion strategies on model performance. Then, we examine software developers' attitudes toward vulnerability detection and localization tools.

5.1 Performance of using different channel fusion strategies

In this subsection, we explore the impact of employing various channel fusion strategies on model performance. In our proposed method, we combine the scores from two linear classifiers using a weighted sum to create a single prediction score. We compare this approach with another strategy, denoted as $WAVES_{Select}$, where we select the maximum score from the two linear classifiers as the prediction score. Analyzing the results presented in Table 2, we observe that the strategy that fuses prediction scores from both max pooling and mean pooling channels significantly outperforms the strategy of selecting the maximum score from the two channels. This finding suggests that focusing solely on either max pooling or mean pooling features is insufficient for effective vulnerability detection; it is essential to consider prediction results from both channels simultaneously.

The results of statement-level vulnerability localization, presented in Table 3, exhibit a different trend compared to those in Table 7. The effectiveness of different strategies varies across datasets. Specifically, on the dataset from Fan et al., $WAVES_{Select}$ outperforms WAVES in absolute label prediction (Accuracy, Precision, Recall, and F1) but underperforms

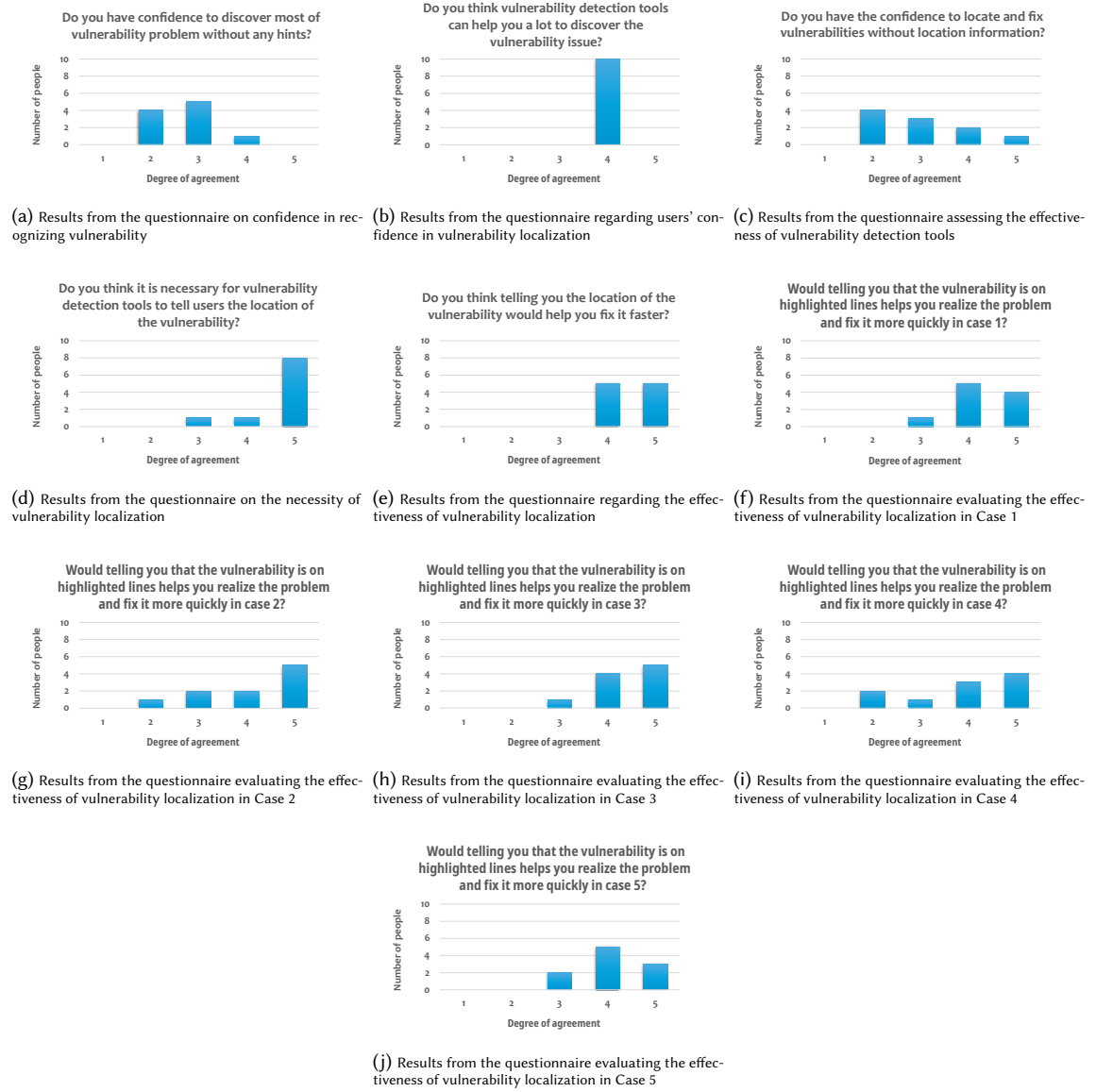


Fig. 3. Results from the questionnaire about users' perspectives on vulnerability localization

in relative score ranking (MAR, MFR, IFA, Top-1, Top-3, Top-5). In contrast, on the CVEfixes dataset, WAVES_{Select} performs worse in absolute label prediction but achieves comparable performance to WAVES in relative score ranking.

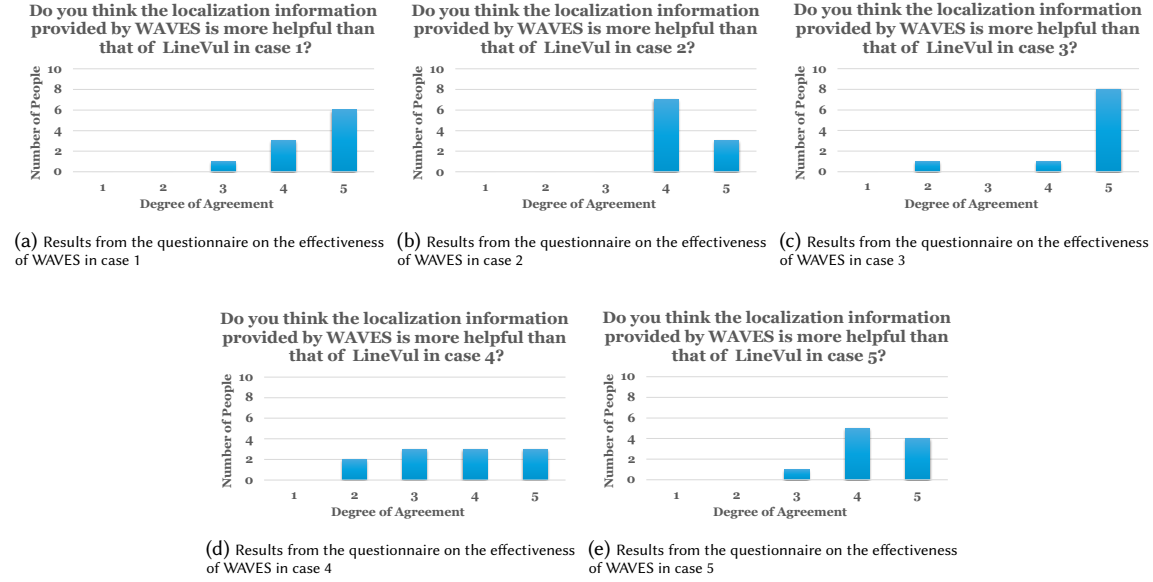


Fig. 4. Results from the questionnaire about the effectiveness of WAVES compared to LinVul on vulnerability localization

5.2 Human assessment of the effectiveness of vulnerability localization

5.2.1 Study on the effectiveness of vulnerability localization tools for developers in real-world scenarios. To understand the effectiveness of vulnerability localization tools for developers in real-world scenarios, we conducted a study with 10 participants: 3 data analysts and 7 developers. Among them, three have 5 to 10 years of programming experience, five have 3 to 5 years, and the remaining two have 1 to 3 years. All participants frequently use Python; nine regularly use C/C++, three often use Java, and one regularly uses C#. In the study, we first asked the participants some general questions about their experience with vulnerabilities. Then, we provided five specific vulnerability examples successfully localized by our tools and evaluated whether our tools helped users identify and fix the vulnerabilities in these examples.

Figure 3 presents the responses of our participants to our questionnaire. Using a scale from 1 to 5, we measured participants' agreement with our queries, where 1 indicated strong disagreement and 5 represented strong agreement. From Figure 3 (a) and (b), we found that recognizing vulnerabilities is challenging for human developers. Most participants have limited confidence in discovering vulnerability issues independently. Even when informed that a given function is vulnerable, they still lack confidence in localizing and fixing the vulnerable statements. Figures 3 (c) to (e) indicate that vulnerability detection tools assist users in identifying potential vulnerabilities and highlight a strong need for automatic vulnerability localization.

We randomly selected five vulnerable functions that can be correctly detected and localized by our proposed approach. We then asked participants if the localization information provided by our approach helped them recognize the vulnerability and fix it more quickly. According to the results shown in Figure 3 (f) to (j), over 70% of participants agreed that our proposed approach helped them identify and resolve the vulnerabilities in all the sample functions. Additionally, an average of 4.2 participants strongly agreed that our approach enabled them to address the problem more quickly in each sample vulnerable function, demonstrating the effectiveness of our proposed approach.

5.2.2 Study on WAVES’s effectiveness compared to LineVul in vulnerability localization. To evaluate the effectiveness of the vulnerability localization information provided by WAVES compared to the baseline method LineVul, we conducted a study with 10 participants: 5 software developers, 4 software testing engineers, and 1 assistant professor in computer science. Among them, 2 participants have 3 to 5 years of programming experience, 4 have 5 to 10 years, and the remaining 4 have over 10 years of experience. All participants frequently use Python; 6 regularly work with C/C++, 7 often use Java, 2 frequently use JavaScript/TypeScript, and 1 regularly works with C#.

In Section 4.1, we have already demonstrated that WAVES provides more accurate vulnerability localization information than LineVul. In this study, we aim to further validate whether these correct predictions from WAVES actually help developers better recognize vulnerability issues compared to the incorrect predictions from LineVul. Specifically, we allow both WAVES and LineVul to predict the most likely vulnerable statement and randomly select five test cases where WAVES provides correct predictions while LineVul fails. Participants are then asked to evaluate how much the localization information provided by WAVES improves their understanding of the vulnerability issue compared to LineVul, using a set of five predefined response options.

Figure 4 presents the results of our user study. We observe that the localization information provided by WAVES is generally more helpful than that from LineVul in assisting participants to identify vulnerability issues, demonstrating the effectiveness of WAVES in vulnerability localization. However, its relative effectiveness varies across different test cases. Most participants agree that WAVES’s information is more useful in test cases 1, 2, 3, and 5, while in test case 4, half of the participants found its localization information to be as helpful as—or even less helpful than—that provided by LineVul, despite LineVul highlighting incorrect information. This discrepancy is likely due to the intrinsic difficulty of the vulnerability in test case 4; certain vulnerabilities are inherently hard to recognize, and even if the vulnerable statement is explicitly emphasized, participants may still fail to understand the issue.

6 THREATS TO VALIDITY

After careful analysis, we have identified several potential threats to the validity of our study:

Threats to External Validity. Although we utilized three datasets for evaluation, we only assessed vulnerability localization performance on the dataset provided by Fan et al. The reason for not evaluating localization on the other two datasets is the lack of repaired data for vulnerable functions, which prevents us from labeling the vulnerable statements in these datasets. This challenge highlights the need for our proposed approach, as obtaining statement-level labels is often difficult in real-world scenarios. To effectively evaluate vulnerability localization, a dataset must include both the vulnerable code snippets and their corresponding fixes. While the D2A dataset [51] does contain such fixing information, Croft et al. [10] reported that over 60% of samples in D2A are mislabeled, rendering it unreliable. Therefore, we chose the dataset by Fan et al. as the testing dataset for vulnerability localization. While we have chosen three commonly used datasets to assess the effectiveness of our vulnerability detection approach, it is important to note that these datasets have limited size. Consequently, the results obtained from these datasets may not accurately reflect the performance of our approach in real-world scenarios.

Threats to Internal Validity. In this paper, we adopt the hyperparameters from LineVul [16] to maintain consistency. While we acknowledge the potential impact of hyperparameters on the performance of our proposed WAVES, we did not investigate their influence due to the considerable cost associated with model training. However, it is important to note that different hyperparameter settings may indeed affect WAVES’s performance. Among these settings, the maximum input token length holds particular significance. Currently, in WAVES, we have set the maximum input token

length to 512, discarding any additional tokens. The performance of WAVES for longer code samples has not been thoroughly explored under this constraint.

Moreover, it is worth mentioning that certain vulnerabilities may be closely tied to the context of the code, such as use-after-free vulnerabilities. In some cases, the division of code segments could lead to the disappearance of existing vulnerabilities or even the emergence of new ones, potentially altering the label of the target function. Therefore, we need to carefully consider the implications of code segmentation on WAVES’s effectiveness in identifying such vulnerabilities. Further investigations into the impact of these factors are necessary for a comprehensive understanding of WAVES’s performance.

Furthermore, the vulnerability labels at the statement level are determined by checking whether the statements have been modified in the commit. Consequently, all the modified statements are considered vulnerable. However, simply altering a statement does not guarantee the presence of an actual vulnerability. This data pre-processing approach could potentially introduce biases into the vulnerable statement labels.

7 RELATED WORK

7.1 Deep Learning-based Vulnerability Detection

With the advent of deep learning technology, significant advancements have been made in various tasks, such as code retrieval and code generation. Consequently, researchers in the vulnerability detection field have also taken notice. The integration of deep learning into vulnerability detection approaches has resulted in a substantial improvement in performance compared to conventional methods. Existing studies in this area can be broadly categorized into token-based approaches and graph-based approaches, each utilizing different representations of the source code. In the following subsections, we provide a brief overview of these two types of approaches.

7.1.1 Token-based Vulnerability Detection Approaches. Several works [11, 27, 28] approach source code as flat sequences and adopt natural language processing techniques to represent input code and initialize tokenized code tokens with Word2Vec [31]. Li et al. [28] initiate the study of using deep learning for vulnerability detection. They transform programs into code gadgets consisting of multiple lines of code statements that are semantically related and propose VulDeePecker, a Bidirectional Long Short Time Memory (BLSTM) neural network with a dense layer to learn representations. To further represent programs into vectors that accommodate the syntax and semantic information suitable for vulnerability detection, Li et al. [27] extract code slices according to data dependency and control dependency and also utilize BLSTM to obtain representation for detection. Another work [11] leverages Convolutional Neural Networks (CNN) and Recurrent Neural Networks (RNNs) to directly learn features from source code.

7.1.2 Graph-based Vulnerability Detection Approaches. Source code is inherently structured and logical and has heterogeneous aspects of representation, such as Abstract Syntax Tree (AST), Control Flow Graph (CFG), Data Flow Graph (DFG), and Program Dependency Graph (PDG). Many works [6, 7, 25, 26, 32, 44, 46, 52] represent source code into single code graphs or composite graphs to improve the syntactic and semantic information. Zhou et al. [52] construct a heterogeneous joint graph consisting of AST, CFG, and DFG following [47]. They connect the neighboring leaf nodes of the AST to preserve the natural sequential order of the source code and utilize Gated Graph Neural Network (GGNN) with the convolution module for graph-level classification. Chakraborty et al. [6] extract the information of Code Property Graph (CPG) [47] from the given function and adopts the technique of Word2Vec [31] to initialize the embedding vector. They also utilize the GGNN model to learn graph representation and focus on solving the problem of

dataset imbalance. Li et al. [26] also extract multiple types of graphs, but unlike the above two methods which use GNN to directly learn the representation of joint graphs, they leverage Gate Recurrent Unit (GRU) [9] or Tree-LSTM [39] to learn a single type of graph respectively and aggregate these vectors through convolution operation. Cheng et al. [7] and Wu et al. [46] both distill the function semantic information into a PDG which contains control-flow and data-flow details of source code, and they train a GNN model and a Convolutional Neural Network (CNN) [24] model to detect the vulnerability, respectively.

7.2 Deep Learning-based Statement-Level Vulnerability Detection and Localization

Although deep learning-based approaches for function-level vulnerability detection have attracted a large number of researchers and achieved great progress, there are still limitations in practical applications. Even if the function-level vulnerability can be successfully detected, it still requires a large effort to locate the vulnerable statements inside the function if the vulnerable function contains many statements.

Li et al. [26] leveraged GCN for function-level predictions and GNNExplainer [49] to explore the subgraphs that contribute most to the predictions. However, this still cannot provide specific vulnerable statements. Many statement-level vulnerability detection approaches [12, 16, 20] have been proposed. Ding et al. [12] and Hin et al. [20] both provided labels for vulnerable statements in the training phase. Ding et al. [12] propose an ensemble learning approach named VELVET which combines GGNN and Transformer [41] to capture the local and global context of the source code. VELVET shows good performance on both vulnerability classification and localization. Hin et al. [20] formulates statement-level vulnerability detection as a node classification task. They use CodeBERT [15] to initialize the embedding of each statement in the function and further leverage Graph Attention Network (GAT) [42] to update these statement embeddings according to the control and data dependency between statements. The performance of these supervised approaches is sensitive to the quality and quantity of the annotated data. To solve this issue, Fu and Tantithamthavorn [16] propose an unsupervised approach name LineVul that leverages the attention mechanism of CodeBERT [15] to find the most likely vulnerable statements. LineVul uses CodeBERT to capture the long-term dependencies and semantic context of the statements and aggregate the attention score of each statement for statement-level vulnerability detection. The experiment results indicate that unsupervised statement-level vulnerability detection is feasible and has room for further improvement.

7.3 Multiple Instance Learning

With the development of deep learning technology, people start to understand the importance of the data scale to deep learning. However, collecting fine-grained labeled data from the real world remains a big problem. The data we acquire from the real world are usually unlabelled and it will cost a lot of time and huge manpower to label these data. To alleviate the need for labeled data, the deep learning methods that require fewer data, including unsupervised learning and weakly supervised learning, have become a research hotspot. Multiple instance learning is a common method in weakly supervised learning. Under the framework of multiple instance learning, the training data are arranged into sets, which are called bags. Only the label of the bags will be provided and the label of every single data is not available during the training. The model with multiple instance learning can learn to predict either the instance label or the bag label by learning the labels of bags and distinguishing the positive instances and negative instances inside the bags. The feature of less demand for labeled data makes multiple instance learning widely used in different scenarios. We briefly introduce its application in a few areas.

In Computer Vision, Pinheiro et al. [34] convert the task of object segmentation to the task of inferring the pixels in the given image which belongs to the class of the object and propose a Convolutional Neural Network-based model with multiple instance learning framework for such a task. In the task of histopathology image analysis, the duration of pixel-level annotations is laborious and time-consuming. To address this problem, Chikontwe et al. [8] propose a MIL-based approach that can jointly learn the embedding vectors at both bag-level and instance-level to diagnose cancer. In the task of retinal image classification, Tu et al. [40] first attempt to combine the Graph Neural Networks with multiple instance learning and achieve state-of-art performance on the public datasets. Inspired by the dynamic routing in capsule networks, Yan et al. [48] adopt a dynamic pooling function to replace the conventional max pooling or mean pooling function for multiple instance learning and achieve the state-of-art performance in the task of animal detection.

In the task of sentiment analysis for texts, Pappas et al. [33] encode each sentence or paragraph into a feature vector and adopt multiple instance regression (MIR) to assign importance weights to each of the sentences or paragraphs of a text to uncover their contribution to the aspect ratings. In the task of contextual advertisement, advertisers wish to avoid some specific content on web pages but it is difficult since most training pages are multi-topic and need people to label at the sub-document level. To address this challenge, Zhang et al. [50] adopt multiple instance learning to detect and avoid content that is related to war, violence, and pornography or the negative opinion about their product.

8 CONCLUSION

In this paper, we present a novel approach named WAVES for vulnerability detection. WAVES incorporates the multiple instance learning framework to predict whether a given function is vulnerable or not, while also offering precise localization information about the vulnerable statements within the function. Through experiments conducted on public datasets, we have demonstrated that WAVES achieves comparable performance to previous baselines in function-level vulnerability detection, and outperforms state-of-the-art baselines in statement-level vulnerability localization.

In the future, we plan to investigate the dynamic assignment of distinct pseudo-labels at the statement level across different functions, aiming to further improve the accuracy of vulnerability localization. In addition, we will expand our dataset to include programs with longer code segments in order to systematically analyze how code length influences the performance of the proposed method.

9 ACKNOWLEDGMENT

This paper is supported by the Research Grants Council Grant for Theme-based Research Scheme Project (RGC Ref No. T43-513/23-N), National Natural Science Foundation of China under project (No. 62472126), and Natural Science Foundation of Guangdong Province (Project No. 2023A1515011959).

REFERENCES

- [1] 20123. 2023 CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html.
- [2] Stuart Andrews, Ioannis Tsochantaridis, and Thomas Hofmann. 2002. Support Vector Machines for Multiple-Instance Learning. In *Advances in Neural Information Processing Systems 15 [Neural Information Processing Systems, NIPS 2002, December 9-14, 2002, Vancouver, British Columbia, Canada]*, Suzanna Becker, Sebastian Thrun, and Klaus Obermayer (Eds.). MIT Press, 561–568. <https://proceedings.neurips.cc/paper/2002/hash/3e6260b81898beacda3d16db379ed329-Abstract.html>
- [3] Charles Bergeron, Gregory M. Moore, Jed Zaretski, Curt M. Breneman, and Kristin P. Bennett. 2012. Fast Bundle Algorithm for Multiple-Instance Learning. *IEEE Trans. Pattern Anal. Mach. Intell.* 34, 6 (2012), 1068–1079. <https://doi.org/10.1109/TPAMI.2011.194>
- [4] Guru Prasad Bhandari, Amara Naseer, and Leon Moonen. 2021. CVEfixes: automated collection of vulnerabilities and their fixes from open-source software. In *PROMISE '21: 17th International Conference on Predictive Models and Data Analytics in Software Engineering, Athens Greece, August 19-20, 2021*, Shane McIntosh, Xin Xia, and Sousuke Amasaki (Eds.). ACM, 30–39. <https://doi.org/10.1145/3475960.3475985>

- [5] Marc-André Carboneau, Veronika Cheplygina, Eric Granger, and Ghyslain Gagnon. 2018. Multiple instance learning: A survey of problem characteristics and applications. *Pattern Recognit.* 77 (2018), 329–353. <https://doi.org/10.1016/j.patcog.2017.10.009>
- [6] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2022. Deep Learning Based Vulnerability Detection: Are We There Yet? *IEEE Trans. Software Eng.* 48, 9 (2022), 3280–3296.
- [7] Xiao Cheng, Haoyu Wang, Jiayi Hua, Guoai Xu, and Yulei Sui. 2021. DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network. *ACM Trans. Softw. Eng. Methodol.* 30, 3 (2021), 38:1–38:33.
- [8] Philip Chikontwe, Meejeong Kim, Soo Jeong Nam, Heounjeong Go, and Sang Hyun Park. 2020. Multiple Instance Learning with Center Embeddings for Histopathology Classification. In *Medical Image Computing and Computer Assisted Intervention - MICCAI 2020 - 23rd International Conference, Lima, Peru, October 4-8, 2020, Proceedings, Part V (Lecture Notes in Computer Science, Vol. 12265)*. Anne L. Martel, Purang Abolmaesumi, Danail Stoyanov, Diana Mateus, Maria A. Zuluaga, S. Kevin Zhou, Daniel Racoceanu, and Leo Joskowicz (Eds.). Springer, 519–528. https://doi.org/10.1007/978-3-030-59722-1_50
- [9] Junyoung Chung, Çağlar Gülçehre, KyungHyun Cho, and Yoshua Bengio. 2014. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. *CoRR* abs/1412.3555 (2014).
- [10] Roland Croft, Muhammad Ali Babar, and M. Mehdi Kholoosi. 2023. Data Quality for Software Vulnerability Datasets. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 121–133. <https://doi.org/10.1109/ICSE48619.2023.00022>
- [11] Hoa Khanh Dam, Truyen Tran, Trang Pham, Shien Wee Ng, John Grundy, and Aditya Ghose. 2017. Automatic feature learning for vulnerability prediction. *CoRR* abs/1708.02368 (2017). [arXiv:1708.02368](http://arxiv.org/abs/1708.02368) <http://arxiv.org/abs/1708.02368>
- [12] Yangruibo Ding, Sahil Suneja, Yunhui Zheng, Jim Laredo, Alessandro Morari, Gail E. Kaiser, and Baishakhi Ray. 2022. VELVET: a noVel Ensemble Learning approach to automatically locate Vulnerable Statements. In *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022, Honolulu, HI, USA, March 15-18, 2022*. IEEE, 959–970.
- [13] Facebook. 2021. Infer. <https://fbinfer.com/>.
- [14] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In *MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*. ACM, 508–512.
- [15] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020 (Findings of ACL, Vol. EMNLP 2020)*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, 1536–1547.
- [16] Michael Fu and Chakkrit Tantithamthavorn. 2022. LineVul: A Transformer-based Line-Level Vulnerability Prediction. In *19th IEEE/ACM International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022*. ACM, 608–620.
- [17] Dan Goodin. [n. d.]. An NSA-derived ransomware worm is shutting down computers worldwide (2017). <https://arstechnica.com/information-technology/2017/05/>
- [18] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (Eds.). Association for Computational Linguistics, 7212–7225. <https://doi.org/10.18653/v1/2022.acl-long.499>
- [19] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. <https://openreview.net/forum?id=jLoC4ez43PZ>
- [20] David Hin, Andrey Kan, Huaming Chen, and Muhammad Ali Babar. 2022. LineVD: Statement-level Vulnerability Detection using Graph Neural Networks. In *19th IEEE/ACM International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022*. ACM, 596–607.
- [21] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (1997), 1735–1780.
- [22] Israel. 2021. Checkmarx. <https://checkmarx.com/>.
- [23] Dimitrios Kotzias, Misha Denil, Nando de Freitas, and Padhraic Smyth. 2015. From Group to Individual Labels Using Deep Features. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Sydney, NSW, Australia, August 10-13, 2015*, Longbing Cao, Chengqi Zhang, Thorsten Joachims, Geoffrey I. Webb, Dragos D. Margineantu, and Graham Williams (Eds.). ACM, 597–606.
- [24] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*, Peter L. Bartlett, Fernando C. N. Pereira, Christopher J. C. Burges, Léon Bottou, and Kilian Q. Weinberger (Eds.). 1106–1114.
- [25] Min Li, Chunfang Li, Shuailou Li, Yanna Wu, Boyang Zhang, and Yu Wen. 2021. ACGVD: Vulnerability Detection Based on Comprehensive Graph via Graph Neural Network with Attention. In *Information and Communications Security - 23rd International Conference, ICICS 2021, Chongqing, China, November 19-21, 2021, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 12918)*, Debin Gao, Qi Li, Xiaohong Guan, and Xiaofeng Liao (Eds.). Springer, 243–259.
- [26] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. Vulnerability detection with fine-grained interpretations. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*. ACM, 292–303.

- [27] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2022. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *IEEE Trans. Dependable Secur. Comput.* 19, 4 (2022), 2244–2258.
- [28] 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 preprint arXiv:1801.01681* (2018).
- [29] Tsung-Yi Lin, Priya Goyal, Ross B. Girshick, Kaiming He, and Piotr Dollár. 2020. Focal Loss for Dense Object Detection. *IEEE Trans. Pattern Anal. Mach. Intell.* 42, 2 (2020), 318–327. <https://doi.org/10.1109/TPAMI.2018.2858826>
- [30] Ilya Loshchilov and Frank Hutter. 2017. Fixing Weight Decay Regularization in Adam. *CoRR* abs/1711.05101 (2017).
- [31] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *1st International Conference on Learning Representations, ICLR 2013*.
- [32] Van-Anh Nguyen, Dai Quoc Nguyen, Van Nguyen, Trung Le, Quan Hung Tran, and Dinh Phung. 2022. ReGVD: Revisiting Graph Neural Networks for Vulnerability Detection. In *44th IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2022, Pittsburgh, PA, USA, May 22-24, 2022*. ACM/IEEE, 178–182.
- [33] Nikolaos Pappas and Andrei Popescu-Belis. 2014. Explaining the Stars: Weighted Multiple-Instance Learning for Aspect-Based Sentiment Analysis. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, Alessandro Moschitti, Bo Pang, and Walter Daelemans (Eds.). ACL, 455–466. <https://doi.org/10.3115/v1/d14-1052>
- [34] Pedro H. O. Pinheiro and Ronan Collobert. 2015. From image-level to pixel-level labeling with Convolutional Networks. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*. IEEE Computer Society, 1713–1721. <https://doi.org/10.1109/CVPR.2015.7298780>
- [35] Rouhollah Rahmani and Sally A. Goldman. 2006. MISSL: multiple-instance semi-supervised learning. In *Machine Learning, Proceedings of the Twenty-Third International Conference (ICML 2006), Pittsburgh, Pennsylvania, USA, June 25-29, 2006 (ACM International Conference Proceeding Series, Vol. 148)*, William W. Cohen and Andrew W. Moore (Eds.). ACM, 705–712. <https://doi.org/10.1145/1143844.1143933>
- [36] Rebecca L. Russell, Louis Y. Kim, Lei H. Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul M. Ellingwood, and Marc W. McConley. 2018. Automated Vulnerability Detection in Source Code Using Deep Representation Learning. In *17th IEEE International Conference on Machine Learning and Applications, ICMLA 2018, Orlando, FL, USA, December 17-20, 2018*, M. Arif Wani, Mehmed M. Kantardzic, Moamar Sayed Mouchaweh, João Gama, and Edwin Lughofer (Eds.). IEEE, 757–762.
- [37] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural Machine Translation of Rare Words with Subword Units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics.
- [38] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, Ayal Zaks and Manuel V. Hermenegildo (Eds.). ACM, 265–266.
- [39] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing, ACL 2015, July 26-31, 2015, Beijing, China, Volume 1: Long Papers*. The Association for Computer Linguistics, 1556–1566.
- [40] Ming Tu, Jing Huang, Xiaodong He, and Bowen Zhou. 2019. Multiple instance learning with graph neural networks. *CoRR* abs/1906.04881 (2019). [arXiv:1906.04881](http://arxiv.org/abs/1906.04881) <http://arxiv.org/abs/1906.04881>
- [41] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 5998–6008.
- [42] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net.
- [43] John Viega, J. T. Bloch, Y. Kohn, and Gary McGraw. 2000. ITS4: A Static Vulnerability Scanner for C and C++ Code. In *16th Annual Computer Security Applications Conference (ACSAC 2000), 11-15 December 2000, New Orleans, Louisiana, USA*. IEEE Computer Society, 257.
- [44] Huaning Wang, Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Yansong Feng, Lizhong Bian, and Zheng Wang. 2021. Combining Graph-Based Learning With Automated Data Collection for Code Vulnerability Detection. *IEEE Trans. Inf. Forensics Secur.* 16 (2021), 1943–1958.
- [45] D. A. Wheeler. 2021. Flawfinder. <https://dwheeler.com/flawfinder/>, title = Flawfinder..
- [46] Yueming Wu, Deqing Zou, Shihan Dou, Wei Yang, Duo Xu, and Hai Jin. 2022. VulCNN: An Image-inspired Scalable Vulnerability Detection System. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2365–2376.
- [47] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *2014 IEEE Symposium on Security and Privacy, SP 2014*. IEEE Computer Society, 590–604.
- [48] Yongluan Yan, Xinggang Wang, Xiaojie Guo, Jiemin Fang, Wenyu Liu, and Junzhou Huang. 2018. Deep Multi-instance Learning with Dynamic Pooling. In *Proceedings of The 10th Asian Conference on Machine Learning, ACML 2018, Beijing, China, November 14-16, 2018 (Proceedings of Machine Learning Research, Vol. 80)*. PMLR, 104–114.

- Learning Research*, Vol. 95), Jun Zhu and Ichiro Takeuchi (Eds.). PMLR, 662–677. <http://proceedings.mlr.press/v95/yan18a.html>
- [49] Zhitao Ying, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. 2019. GNNExplainer: Generating Explanations for Graph Neural Networks. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 9240–9251.
 - [50] Yi Zhang, Arun C. Surendran, John C. Platt, and Mukund Narasimhan. 2008. Learning from multi-topic web documents for contextual advertisement. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Las Vegas, Nevada, USA, August 24-27, 2008*, Ying Li, Bing Liu, and Sunita Sarawagi (Eds.). ACM, 1051–1059. <https://doi.org/10.1145/1401890.1402015>
 - [51] Yunhui Zheng, Saurabh Pujar, Burn L. Lewis, Luca Buratti, Edward A. Epstein, Bo Yang, Jim Laredo, Alessandro Morari, and Zhong Su. 2021. D2A: A Dataset Built for AI-Based Vulnerability Detection Methods Using Differential Analysis. In *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2021, Madrid, Spain, May 25-28, 2021*. IEEE, 111–120. <https://doi.org/10.1109/ICSE-SEIP52600.2021.00020>
 - [52] Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019*. 10197–10207.