MatsVD: Boosting Statement-Level Vulnerability Detection via Dependency-Based Attention

Cheng Weng
National University of Defense Technology
Changsha, China
wengcheng@nudt.edu.cn

Yihao Qin
National University of Defense Technology
Changsha, China
yihaoqin@nudt.edu.cn

Bo Lin
National University of Defense Technology
Changsha, China
linbo19@nudt.edu.cn

Pei Liu
National University of Defense Technology
Changsha, China
liupei@nudt.edu.cn

Liqian Chen*
National University of Defense Technology
Changsha, China
lqchen@nudt.edu.cn

ABSTRACT

Software vulnerabilities inevitably arise during software development and may leave behind huge security risks. In order to detect and mitigate vulnerabilities before they can be exploited, various fine-grained deep learning (DL)-based vulnerablity detection (VD) approaches have been proposed to locate vulnerable statements, among which the Transformer-based methods have shown promising performances. However, existing Transformer-based statement-level approaches still suffer from a crucial limitation: they ignore the intrinsic data/control dependency relations between the statements. In this work, we propose a novel Transformer-based model MatsVD, which aims to address the above challenge from two aspects: Firstly, inspired by the hierarchical structure of code (i.e., tokens, statements, and functions), MatsVD comprises three different Transformer-based layers (i.e., statement embedding layer, statement representation layer, and function representation layer) to gradually aggregate the basic code tokens into meaningful statement/function representations; Secondly, to further exploit the data/control dependencies between statements, we replace the original attention mechanism of the Transformer with a novel dependency-based attention by masking irrelevant attention scores according to the program dependency graph. We comprehensively evaluate MatsVD on the widely used C/C++ vulnerability dataset Big-Vul. The results show that MatsVD significantly outperforms 6 other statement-level methods on both binary classification and ranking metrics. In particular, MatsVD obtains an F1 score of 86% and a Top-1 Accuracy of 93% on statement-le, which improves by respectively 22.97% and 7.76% compared to the state-of-the-art method VELVET.

*Liqian Chen is the corresponding author.

Internetware 2024, July 24–26, 2024, Macau, Macao

ACM ISBN 979-8-4007-0705-6/24/07
https://doi.org/10.1145/3671016.3674807

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

KEYWORDS

Software Vulnerability Detection, Deep Learning, Transformer

ACM Reference Format:
Cheng Weng, Yihao Qin, Bo Lin, Pei Liu, and Liqian Chen. 2024. MatsVD: Boosting Statement-Level Vulnerability Detection via Dependency-Based Attention. In *15th Asia-Pacific Symposium on Internetware (Internetware 2024), July 24–26, 2024, Macau, Macao.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3671016.3674807

1 INTRODUCTION

Software vulnerabilities refer to accessible and exploitable security issues in software systems due to insecure coding practices. NVD disclosed 26,447 vulnerabilities in 2023, compared to only 18,938 reported in 2019, indicating that the number of software vulnerabilities is growing rapidly [24]. In particular, over 176,000 vulnerabilities are recorded in the National Vulnerability Database (NVD) [7] and the Common Vulnerabilities and Exposures (CVE) [6]. The insidious and harmful nature of the vulnerabilities could pose significant financial and social risks.

To mitigate this challenge, researchers have proposed various techniques for software vulnerability detection (VD), which can mainly be classified into two categories: (1) program analysis-based approaches, and (2) Deep Learning (DL)-based approaches. Program analysis-based approaches [2, 4, 12] use manual predefined vulnerability templates to detect specific types of vulnerabilities, such as Out-of-bounds Write, Cross-site Scripting, and Use After Free. However, due to the diversity and complexity of software vulnerabilities, it is unaffordable to define an accurate and comprehensive template for every type of vulnerability. Consequently, DL-based approaches [1, 19, 35] are increasingly being proposed to identify a broader range of vulnerabilities without human labor. Typically, DL-based approaches first utilize code representation learning to implicitly capture the vulnerability patterns in the source code, and then detect vulnerabilities through a classifier model.

Recently, to provide developers with more precise detection results, statement-level DL-based VD approaches have been proposed

to identify vulnerable statements within a function. Specifically, some of these approaches [14, 17] represent source code as graphs in multiple views such as data flow graph (DFG), code property graph (CPG) [32], and program dependency graph (PDG) [11], and then predict suspicious nodes that correspond to the vulnerable statements with various GNNs. Other approaches [8, 13, 21], however, directly transform the source code to token sequences and capture the syntactic features and semantic information of the statements with sequence-based DL models such as RNN and Transformer.

Although Transformer has been shown to be superior at capturing syntactic and semantic features of source code [13], existing Transformer-based approaches suffer from a limitation: they simply consider the source code as a set of sequential code statements that are fed directly into Transformer. Consequently, the self-attention mechanism [26] in Transformer treats code statements as fully connected relationships, which ignores the intrinsic structural information of the source code (e.g., data or control dependencies). This oversight may result in a gap between the vulnerability features learned by the model and the underlying semantics of the code, potentially compromising the performance of VD.

To alleviate the above limitation, we introduce a novel Transformer-based approach for statement-level VD. Intuitively, we notice that the source code possesses an intrinsic three-level hierarchical structure: tokens, statements that are connected through data and control dependency relationships, and functions. Therefore, we propose to learn code representations with structural information through a three-layer Transformer-based model: (1) **Statement embedding layer**. First, to obtain more meaningful source code embeddings, we employ a Transformer-based pre-trained model [28] to generate token embeddings and aggregate them into statement embeddings. (2) **Dependency-based statement representation layer**. Second, rather than directly feed the statement embeddings into the Transformer model, we equip the model with the ability to exploit the data/control dependencies between statements. Particularly, we transform source code into PDGs, and then selectively mask partial attention in the Transformer based on the connectivity of the nodes in the PDGs. The masked attention can reduce the interference of information from irrelevant statements and thus generate statement representations with inter-statement dependency information. (3) **Function representation layer**. Finally, we aggregate all statement representations into a function representation using the attention mechanism for obtaining semantic information of the whole function. After the generation of statement and function representations, two MLP networks are trained as classifiers to identify vulnerable statements and functions, respectively. We name our model as MatsVD (**M**asked-**a**ttention **T**ransformer-based **s**tatement-level **V**ulnerability **D**etection approach).

We compared MatsVD against six DL-based statement-level vulnerability detection approaches (i.e., TextCNN [16], ICVH [21], IVDetect [17], LineVD [14], LineVul [13] and VELVET [8]) on the Big-Vul [9], a large real-world dataset with line-level labels. Evaluation results show that our approach significantly outperforms the existing VD techniques. Specifically, MatsVD obtains an F1 score of 86% and a Top-1 Accuracy of 93% on the statement-level VD, with a relative improvement of 22.97% and 7.76% over the best baseline VELVET. In addition, we performed an ablation study to validate the contribution of each layer of MatsVD in statement-level vulnerability detection. Finally, We conduct quantitative and qualitative studies to explore how masked attention and Transformer-based embedding aggregation improve the performance of detecting vulnerable statements.

The main contributions of this paper are as follows:

- **Methodology**. We propose to incorporate inter-statement data and control dependency information into the Transformer via *masked attention*, which reduces the interference of irrelevant attention computation for generating better code representation.
- **Tool**. We propose MatsVD, a novel DL-based approach for statement-level vulnerability detection. MatsVD comprises three Transformer layers, corresponding to the intrinsic three-level hierarchical structure of code: tokens, statements, and functions. MatsVD is publicly available at https://github.com/MatsVD/MatsVD.
- **Experiment**. We extensively evaluate the effectiveness of MatsVD on the large real-world vulnerability dataset Big-Vul [9]. The experimental results show that MatsVD significantly outperforms other statement-level baselines in terms of both binary classification and ranking metrics.

2 BACKGROUND

2.1 Transformer

The Transformer [26] model consists of two parts: encoder and decoder. For vulnerability detection, only the encoder is used to convert the source code into vector representations. The encoder consists of N identical layers and each layer consists of two components: a multi-head self-attention layer and a fully connected feed-forward network. Let $X^v = [x_1^v, x_2^v, \ldots, x_n^v] \in \mathsf{R}^{n \times d}$ represents the input vectors, where n is the sequence length and d is the embedding size. The input matrix X^v is firstly respectively multiplied by three different weight matrices W^Q, W^K and W^V to obtain three main components, namely, Query (Q), Key (K) and Value (V). Then, the scaled dot products of the query with all keys are calculated to obtain the attention score matrix $A \in \mathsf{R}^{n \times n}$. The element a_{ij} of matrix A represents the similarity between the embedding vector x_i^v and x_j^v. Finally, attention scores are normalized to attention weights that are multiplied by corresponding value vectors to obtain attention vectors. After all the self-attentions have been calculated, they are concatenated to obtain the multi-head attention. The calculation of the self-attention and the vector representations X^t is shown below:

$$Q = X^v W^Q,\ K = X^v W^K,\ V = X^v W^V, \tag{1}$$

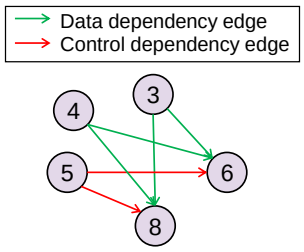
$$A = \frac{QK^\top}{\sqrt{d_k}}, \quad Attention(Q, K, V) = \text{softmax}(A)V, \tag{2}$$

$$\begin{aligned} M^t &= LN(X^{t-1} + MultiHead(X^{t-1})), \\ X^t &= LN(M^t + FFN(M^t)), \end{aligned} \tag{3}$$

where $X^0 = X^v$, $t \in [1, \ldots, N]$, M^t is the attention output after residual connection and layer normalization, $MultiHead$ is a multi-head self-attention layer, FFN is a fully connected feed-forward network, and LN denotes layer normalization.

```
1  void foo( )
2  {
3    int x = getData( );
4    int y = 0;
5    if (x < 0)
6        y = x + y;
7    else
8        y = x - y;
9  }
```

(a) Exemplary code sample



(b) Program dependence graph (PDG)

Figure 1: Program Dependence Graph for an exemplary code.

2.2 Program Dependence Graph

Program Dependence Graph (PDG) is an intermediate program representation [11] for program analysis. Figure 1 shows a simple code example. The PDG on the right side represents the code on the left side as a graph $G = (V, E)$, where V is a set of nodes corresponding to the code statements, and E is a set of data and control dependence edges that labeled with green and red arrows, respectively. The data dependence edges reflect the dataflow relations between statements. For example, the variable `x` declared in line 3 is used in statement `y = x + y;` in line 6, which is represented as an edge from node 3 to node 6 in the PDG. The control dependence edges represent the control relationships between statements, for example, the `if` condition at line 5 decides the execution of statements in line 6 and line 8, which induce two edges from node 5 to node 6 and 8, respectively.

3 APPROACH

3.1 Problem definition

In real-world scenarios, developers typically first determine whether a given function is vulnerable, and then further identify vulnerable statements within the vulnerable function. This practice can prevent the time wasted on trying to locate vulnerable statements from benign functions. Therefore, we formulate statement-level VD as a two-phase binary classification problem in this work. In the first phase, we predict whether a given function is vulnerable or benign. If the function is identified as vulnerable, the second phase is then activated to predict which statements are vulnerable in all the code statements.

We formalize the dataset as $\{(X_i, Y_i, Z_i)|X_i \in \mathcal{X}, Y_i \in \mathcal{Y}, Z_i \in \mathcal{Z}, i \in \{1, 2, \ldots, N\}\}$, where N is the number of functions, $\mathcal{X}$ is a set of code functions, $\mathcal{Y} = \{0, 1\}$ is the function-level label set with 1 denotes vulnerable and 0 denotes benign, $\mathcal{Z} = \{(z_1, z_2, \ldots, z_n)|z_i \in \{0, 1\}, \forall i = 1, 2, \ldots, n\}$ is a flat binary vector set with 1 indicates vulnerable statement and 0 otherwise, and n is the maximum number of statements we consider in a function.

3.2 MatsVD Framework

3.2.1 Overview. As shown in Figure 2, MatsVD consists of three main components: (1) statement embedding layer that is used to transform textual code statements into numerical embedding vectors, (2) statement representation layer that is used to capture data and control dependencies between statements to generate statement representations through masked attention, and (3) function representation layer that is used to aggregate statement representations into function representations. After obtaining statement representations and function representations, we first feed the function representations into a function-level classifier to determine whether the given function is vulnerable. If it is predicted to be vulnerable, the statement representations will be fed into a statement-level classifier to further identify vulnerable statements.

3.2.2 Statement Embedding Layer. The statement embedding layer is the fundamental layer to transform the textual code statements into numerical vector embeddings that can be processed by deep learning models. Previous approaches [8, 30] adopted a typically primitive way to obtain statement embeddings. They first generate an embedding vector for each token by an embedding model (e.g., Word2Vec [20]) and then aggregate the token embeddings of a statement to get the corresponding statement embedding using *max pooling* or *average pooling*. However, the existing statement embedding generation approaches may suffer from several potential performance losses. On the one hand, the max pooling strategy extracts the maximum token embedding of each statement, which may result in information loss. On the other hand, the average pooling treats all the tokens equally, which can potentially dilute the impact of prominent tokens.

Transformer-based Embedding Aggregation. To enhance the quality of the initialized statement embeddings, MatsVD proposes to utilize a code pre-trained model CodeT5 [28] to generate an embedding for each token in a statement, then employ the self-attention mechanism [26] to naturally aggregate the token embeddings into statement embeddings. We expect to obtain better-initialised token embeddings With the pre-trained model, which could get more meaningful code representations and improve model capabilities. Moreover, different from existing statement embedding approaches that adopt average or max pooling strategies for token embedding aggregation, MatsVD adds an extra special token at the beginning of the token sequence of a statement, and takes the corresponding output embedding of the special token as the statement embedding. Inspired by the concept of the *classification token* in some Transformer-based models for downstream classification tasks [18], we argue that the special token could naturally guide the Transformer model to aggregate the information through self-attention computation. The overall process of the statement embedding layer can be formalized as follows:

Given a function consisting of n code statements $X_i = [x_1, \ldots, x_n]$, we first use byte pair encoding (BPE) [23] to tokenize each statement $x_j (j \in \{1, 2, \ldots, n\})$ to a sequence of tokens denoted as $[t_1, \ldots, t_r]$, where r is the max number of tokens we consider in a statement. Then, we insert a special token t_0 = "<s>" at the beginning of each sequence and feed it into the CodeT5 encoder, which results in the vector representations of all tokens from the output of the last encoder layer. Finally, we retain the vector representation of the first token (i.e., the vector corresponds to the special token) as the statement embedding for x_j and name it as $x_j^v \in \mathsf{R}^d$. In summary, given a function X_i, the statement embedding layer produces the statement embedding for each code statement, denoted as $X_i^v = [x_1^v, x_2^v, \ldots, x_n^v] \in \mathsf{R}^{n \times d}$.

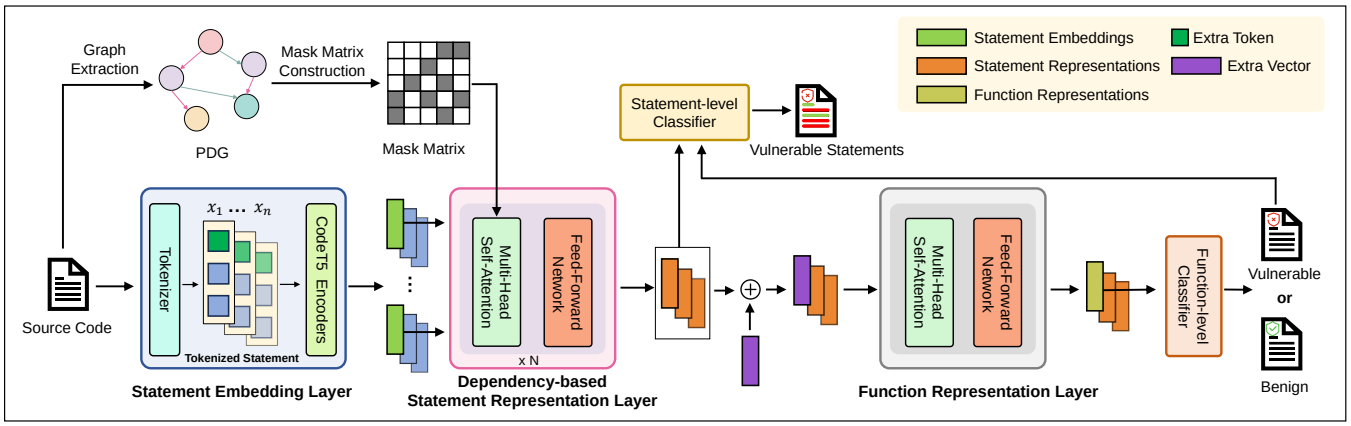


Figure 2: An overview architecture of MatsVD.

3.2.3 Dependency-based Statement Representation Layer. Intuitively, vulnerabilities may caused by the vulnerable statements interacting with other dependent statements. Consequently, to identify whether a statement is vulnerable, it is not only necessary to consider its own syntactic and semantic information (we have achieved it in the aforementioned statement embedding layer), but also crucial to capture valuable features from the relevant contextual information (i.e., the other dependent statements). However, although previous work [13] has shown the Transformer model's capability to capture long-term dependencies between source code tokens, the primitive self-attention computation (mentioned in Section 2.1) treats each token equally, which failed to employ the inter-statement data or control dependencies. In this layer, MatsVD aims to utilize the data and control dependencies between statements for generating more meaningful statement representations. Specifically, to guide the Transformer model capture dependencies between statements more efficiently, we encode the dependencies within an input function as a mask matrix and then apply it to the attention score matrix during the self-attention computation.

Mask Matrix Construction. Formally, we first obtain the PDG $G = (V, E)$ of a given function through program analysis [31], where V is a set of nodes representing code statements and E is a set of data and control dependency edges. Let (v_i, v_j) denotes a data or control dependency edge between statements x_i and x_j, where $i, j \in \{1, 2, \dots, n\}$. Note that we do not explicitly distinguish the different edge types in the constructed PDG, because the alone data/control relationship between statements may be too sparse. According to the PDG, we then construct a mask matrix $M \in \mathbb{R}^{n \times n}$ consisting of zero and minus infinity. We define m_{ij} as the (i, j)-element of the mask matrix M. As shown in Equation 4, if there is no data or control dependency between code statements x_i and x_j, we set m_{ij} as minus infinity. Otherwise, we set it as zero. Note that our intuition is that each statement is naturally most relevant to itself, even if it is not represented as data or control dependency. Thus, we do not mask the attention score at the position (i, j) when $i = j$. Figure 3 shows an example of the construction process of a mask matrix. For example, the edge from node 3 to node 6 is derived from the fact that the variable x declared at line 3 is used at line 6. Therefore, the element m_{36} and m_{63} in the mask matrix M are set to zero, which means there is a data dependency between code statements x_3 and x_6. On the contrary, since there is no data or control dependency between statements 3 and 4, the elements m_{34} and m_{43} in the mask matrix M are set to minus infinity.

$$m_{ij} = \begin{cases} 0, & (v_i, v_j) \in E \text{ or } v_i = v_j \\ -\infty, & otherwise \end{cases} \tag{4}$$

Dependency-based Attention. After obtaining the mask matrix M, we then utilize it to modify the self-attention computation in the Transformer model. Intuitively, we argue that although the original self-attention mechanism can adaptively focus on important inputs during the training process, the fully-connected manner may still cause interference of information from irrelevant inputs. Consequently, we employ the mask matrix M to explicitly guide the model to concentrate on specified inputs, i.e., other statement embeddings that possess data or control dependencies with the current statement embedding. Let a_{ij} denote the (i, j)-element of the attention score matrix A in equation 2, we modify it by adding m_{ij} as below:

$$a_{ij} = \frac{x_i^v W^Q \quad x_j^v W^K \quad ^T}{\sqrt{d_k}} + m_{ij}. \tag{5}$$

We then normalize the attention scores using the softmax function to get the attention weights and multiply them by the value vectors to obtain the attention vector as below:

$$w_{ij} = \text{Softmax}\ a_{ij} = \frac{\exp(a_{ij})}{\sum_{j=1}^{n} \exp\ a_{ij}}, \tag{6}$$

$$c_i = \sum_{j=1}^{n} w_{ij}(x_j^v W^V), \tag{7}$$

where c_i is the output vector of the self-attention calculation which corresponds to statement x_i, and w_{ij} is the attention weight that indicates the contribution degree of statement x_j in generating the statement representation of x_i. Considering the modified self-attention, when statements x_i and x_j have no dependency relationship, we add an infinitely negative value to the corresponding attention scores. In this way, the attention weights outputted by

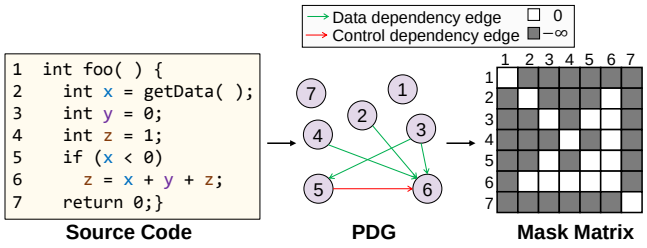


Figure 3: An example of mask matrix construction.

the softmax function are zero, which means that the embedding of x_i has no contribution to the representation computation of x_j, and vice versa. In other words, dependency-based self-attention guides the model to generate the statement representations by only considering the statements that are dependent on them, thus avoiding the influence of irrelevant information.

After getting the vectors for each statement, the output of the current encoder will be fed into the next encoder according to Eq. 3. We utilize the output vectors of the last encoder as statement representations. All the statement representations are denoted as $S_i^v = [s_1^v, s_2^v, \ldots, s_n^v] \in \mathsf{R}^{n \times d}$, where $s_i^v \in \mathsf{R}^d$ represents the statement representation of x_i.

3.2.4 Function Representation Layer. In this layer, we aim to aggregate the statement representations we have obtained into the function representation for function-level vulnerability detection. Similar to the aggregation of the token embeddings in the statement embedding layer, we argue that some code statements may contribute more to identifying function-level vulnerabilities than others. Therefore, we utilize the intrinsic self-attention mechanism of the Transformer to allow the model to adaptively concentrate on the statements that are more important and aggregate statement representations to form the function representation.

Specifically, given a sequence of statement representations $S_i^v = [s_1^v, s_2^v, \ldots, s_n^v] \in \mathsf{R}^{n \times d}$, we first insert an extra special vector s_0^v at the beginning position. The vector is regarded as a statement-level context vector used to measure the importance of the statement based on the similarity between s_0^v and s_i^v. Then we input the sequence into Transformer to derive $n+1$ vector representations. During the computation process, the vector representation of the beginning position is computed as a weighted sum of all the statement representations based on their importance. Consequently, we preserve the vector at the beginning position as the function representation for function-level detection, denoted as $M_i^v \in \mathsf{R}^d$.

3.2.5 Linear Classifiers. In the training phase, given a function $X_i = [x_1, \ldots, x_n]$ and the corresponding labels Y_i and Z_i, we aim to learn two mapping functions to precisely detect the vulnerable statements in a function. One map function $f : \mathcal{X} \rightarrow \mathcal{Y}$ is designed to identify whether a given function is vulnerable or not, and another map function $g : \mathcal{X} \rightarrow \mathcal{Z}$ is employed to further predict the problematic statements within the function. Specifically, two multi-layer perceptron (MLP) networks are trained as classifiers to perform two-stage vulnerability detection. We denote the cross entropy loss function as $\mathcal{L}(.)$ and optimize the model parameters by minimizing the following loss function:

$$\begin{aligned} \min \frac{1}{N} \sum_{i=1}^{\tilde{N}} & \mathcal{L}_{\text{func}}^i + \mathcal{L}_{\text{statement}}^i \ , \\ s.t. \quad & \mathcal{L}_{\text{func}}^i = \mathcal{L}(f(X_i, Y_i \mid X_i)), \\ & \mathcal{L}_{\text{statement}}^i = \mathcal{L}(g(X_i, Z_i \mid X_i)). \end{aligned} \tag{8}$$

In the inference phase, a function X_i is firstly transformed into one function representation M_i^v and a set of statement representations S_i^v through the three-layer MatsVD encoding model. Then, we employ the trained classifiers to convert the representations into the function-level probability $\hat{Y}_i \in [0, 1]$ and the statement-level probabilities $\hat{Z}_i = [\hat{z_i}^1, \ldots, \hat{z_i}^n] \in [0, 1]^n$, respectively. We predict function X_i as vulnerable if function-level probability $\hat{Y}_i$ is greater than a predefined threshold t_f. If X_i is predicted as vulnerable, we further predict which statements in X_i are vulnerable by comparing the probabilities in $\hat{Z}_i$ with another threshold t_s. Note that if a function is predicted as benign, we mute the statement-level classifier to directly predict all statements in this function as benign.

4 EXPERIMENTAL DESIGN

4.1 Research Questions

To evaluate the effectiveness of MatsVD in detecting vulnerable statements, we aim to investigate the following three questions:

RQ1: How effective is MatsVD for statement-level vulnerability detection?

RQ2: How does each component affect the performance of MatsVD at the statement level?

RQ3: How do masked attention and Transformer-based aggregation help locate vulnerable statements more precisely?

4.2 Datasets

In order to fully evaluate the effectiveness of our method at the statement level, we choose the large real-world dataset Big-Vul [9]. Extracted from 348 Github projects, the Big-Vul dataset comprises 188,000 C/C++ functions, which encompasses 3,754 code vulnerabilities across 91 distinct vulnerability types. We select this dataset for our evaluation for the following reasons: (1) Big-Vul is one of the largest vulnerability datasets with line-level ground truths, which is indispensable to training a statement-level VD model such as MatsVD. (2) Big-Vul comes from real-world vulnerabilities. Previous study [1] has shown that model performance can be more realistically evaluated on real-world vulnerability projects. (3) Big-Vul is widely used, which has been utilized by three state-of-the-art approaches (i.e., IVDetect [17], LineVD [14] and LineVul [13]) to assess the performance of statement-level VD.

To the best of our knowledge, there are other datasets with statement-level labels that have been used in other studies [8], such as Juliet Test Suite [25] and D2A [34]. However, we did not evaluate MatsVD on these datasets in this work for the following reasons: (1) Juliet Test Suite is an artificially produced synthetic dataset, which has been demonstrated to fall short in reflecting real-world vulnerability patterns [1]. (2) The statement labels in D2A are obtained from the results of static analyzers, which may carry many false positives and false negatives and thus affect the performance of the model [5].

4.3 Baselines

We compare MatsVD with six statement-level deep learning-based vulnerability detection methods.

CNN-based and RNN-based Methods. TextCNN [16] uses a CNN-based model for sentence-level classification tasks. ICVH [21] leverages two bi-RNNs to detect statement-level vulnerabilities by maximizing mutual information.

GNN-based Methods. IVDetect [17] leverages a graph convolution network [15] to predict function-level vulnerabilities and a GNNExplainer [33] to further locate vulnerable statements. LineVD [14] represents source code as PDGs and leverages a graph attention network [27] for statement-level VD.

Transformer-based Methods. LineVul [13] leverages the principle of model interpretability to find the most prominent lines that are interesting to the Transformer model (i.e., the lines with the highest attention scores). VELVET [8] combines a Transformer-based model and a GNN model to capture the semantics of statements for statement-level VD.

4.4 Evaluation Metrics

In addition to the binary classification problem as defined in Section 3.1, statement-level VD can also be considered as a ranking problem where models rank statements by suspicious scores in a vulnerable function. Therefore, We measure the performance of the statement-level approaches on both binary classification and ranking metrics.

Classification Metrics. On the binary classification problem, we report the Matthews Correlation Coefficient (MCC), Precision (Pre), Recall (Re), and F1 Score (F1) metrics of each method:

$$\begin{aligned} MCC &= \frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}, \\ Pre &= \frac{TP}{TP+FP}, Re = \frac{TP}{TP+FN}, F1 = \frac{2 \times Pre \times Re}{Pre + Re}, \end{aligned} \tag{9}$$

where TP, FP, TN, and FN is the number of true positives, false positives, true negatives, and false negatives, respectively. MCC is particularly suitable for dealing with unbalanced datasets. It takes values between -1 and +1, with higher values indicating better model performance.

Ranking Metrics. We use Mean First Ranking (MFR), Mean Average Ranking (MAR), and Top-k Accuracy (Top-k).

MFR measures the average ranking of the first vulnerable statement, and MAR measures the average ranking of all the vulnerable statements. They are defined as:

$$MFR = \frac{\sum_{i=1}^{N} FR(i)}{N}, MAR = \frac{\sum_{i=1}^{N} AR(i)}{N}, \tag{10}$$

where N is the number of vulnerable functions, $FR(i)$ denotes the rank of the first vulnerable statement among the i-th vulnerable function, and $AR(i)$ denotes the average rank of all the vulnerable statements on the i-th vulnerable function. Lower MAR and MFR indicate higher accuracy of vulnerability localization.

Top-k Accuracy measures the percentage of vulnerable functions where at least one actual vulnerable statement is ranked in the Top-k, over all the vulnerable functions. Higher values of Top-k Accuracy indicate better detection performance.

4.4.1 Evaluation Scenario. To better evaluate the performance of the statement-level VD methods, we set two evaluation scenarios for binary classification: (1) **Two-Phase Detection.** In this scenario, only when the function is classified as vulnerable by the model, the vulnerable statements are further detected. The scenario is more consistent with vulnerability detection in practice and reflects the models' authentic detection capability. (2) **One-Phase Detection.** This scenario eliminates interference from function-level detection, which concentrates on reflecting the statement-level VD performance of the model. We use the same trained models and only input vulnerable functions for the models to detect vulnerable statements.

4.5 Experimental Setup

Data Preprocessing and Splitting. We generate the PDG for each function through Joern [31], which is a program analysis tool widely used by existing research [8, 14]. Due to the limitation of Joern and the incorrect syntax of some functions, we discarded some functions for which we failed to get their corresponding PDGs. Then, we delete all the blank lines in the remaining functions. Finally, We follow existing works [14, 17] to obtain line-level ground truth from the vulnerability-fixing commit: (1) Deleted or modified lines in the vulnerable version for fixing are treated as vulnerable. (2) If some lines are added to the vulnerable version for fixing, all lines that are are control or data dependent on the added lines in the vulnerable version are also considered vulnerable. The distribution of the processed Big-Vul dataset is similar to the real-world scenario[1], where the number of vulnerable and benign functions are 7977 and 177,982 respectively, with an approximate proportion of 1:20. Following the previous work [13], we randomly split the entire dataset into training, validation, and test sets by the ratio of 8:1:1.

Implementation Details. Based on the Transformer model, MatsVD requires all input sequences to be of equal length. In order to avoid the loss of valuable information as much as possible, we investigate the code length distribution of the dataset, which shows that approximately 95% of functions contain less than 155 lines and about 95% of lines contain less than 20 tokens. Therefore, we define the max number of statements $n = 155$ in a function and the max number of tokens $r = 20$ in a statement for MatsVD and all the baselines. Note that we treat each line in the source code as a code statement in this work. For functions or statements that exceed or fall short of their corresponding thresholds, we either pad or truncate them as necessary.

Hyper-Parameters Settings. For the pre-trained model CodeT5 [28] used in the statement embedding layer, we use its default settings, i.e., 12 Transformer encoder layers, 768 hidden sizes, and 12 attention headers. We use 4 Transformer encoder layers for generating statement representations and 1 layer for generating function representations, respectively. We set the training batch size to 16, and the epoch size to 20. For classifier models, we set both the threshold of the function and the statement-level VD to 0.5. The learning rate is set to 5e-5. We use Adam as the training optimizer. The model is trained on the Ubuntu20.04 system with 80GB RAM, AMD EPYC 7543 CPU, and A40 graphics card.

Table 1: Results of VD methods on the classification metrics.

Method	Two-Phase Detection								One-Phase Detection			
	Function Level				Statement Level				Statement Level			
	Mcc	Pre	Re	F1	Mcc	Pre	Re	F1	Mcc	Pre	Re	F1
TextCNN	0.3849	61.75	26.20	36.79	0.1404	21.10	30.15	24.83	0.2206	26.24	40.27	31.78
ICVH	0.4556	65.73	33.94	44.76	0.1804	21.39	43.85	28.75	0.2656	27.34	50.76	35.54
IVDetect	0.6171	72.28	55.23	62.61	0.3014	28.36	58.46	38.19	0.3564	31.52	65.06	42.47
LineVD	0.7537	82.47	70.76	76.17	0.3899	35.52	63.86	45.65	0.4506	40.94	67.38	50.93
LineVul	0.8224	87.65	78.58	82.87	0.4951	62.10	47.45	53.79	0.5299	62.38	53.40	57.54
VELVET	0.8973	95.41	85.19	90.01	0.6966	68.74	71.23	69.96	0.6781	59.87	**86.25**	70.68
MatsVD (ours)	**0.9361**	**97.53**	**90.37**	**93.81**	**0.8606**	**91.57**	**81.13**	**86.03**	**0.8585**	**89.90**	84.86	**87.31**

Table 2: Results of VD methods on the ranking metrics.

Method	Top-1	Top-3	Top-5	MFR	MAR
TextCNN	27.41	34.79	46.01	13.16	18.69
ICVH	35.37	43.08	55.83	11.80	16.24
IVDetect	46.69	54.27	62.97	8.73	12.63
LineVD	60.52	73.76	80.74	4.63	10.21
LineVul	24.54	48.49	60.64	9.32	13.94
VELVET	86.09	92.41	93.86	3.73	7.81
MatsVD (ours)	**92.77**	**94.46**	**94.82**	**2.46**	**6.34**

5 EXPERIMENTAL RESULTS

5.1 RQ1: Effectiveness of MatsVD

Classification Metrics. The experimental results of MatsVD and other baselines on classification metrics are shown in Table 1. Under the Two-Phase Detection scenario, MatsVD significantly outperforms all baselines on all metrics in terms of both the function level and the statement level. Specifically, MatsVD achieves 0.8606 in MCC and 86.03% in F1 on the statement level, respectively, which gains an improvement of 23.54% and 22.97% compared to the state-of-the-art baseline VELVET.

Considering the model architecture, we observe that Transformer-based VD methods tend to achieve better performance than methods based on other models. For example, the F1 score of the Transformer-based method LineVul on the function (statement) level is 82.87% (53.79%), which is consistently better than the best GNN-based method LineVD 76.17% (45.65%) and the RNN-based ICVH 44.76% (28.75%). The result is consistent with the findings of Fu et al. [13] that the Transformer architecture better captures long-term dependencies in the source code, leading to better performance. Particularly, even though the Transformer-based baselines have achieved relatively high performance, MatsVD still obtains dramatic performance improvement. We attribute this to the designs of the three Transformer-based layers and the dependency-based attention in MatsVD. The layer-by-layer learning manner from fine to coarse granularity helps the model capture more subtle vulnerability features, while the extra dependency information prevents the model from focusing on useless details.

As we mentioned in Section 4.4.1, the statement-level results of the two-phase detection scenario depend on the function-level performance. To better investigate the authentic statement-level performance of each method, we also present the results under the one-phase detection scenario, where all of the methods directly detect vulnerable statements on the actual vulnerable functions. From the One-Phase Detection column in Table 1, we observe that MatsVD still holds its advantages compared to other baselines after we eliminate the interference of the function-level VD phase. Specifically, the Mcc and F1-score metrics of MatsVD significantly increase by respectively 26.6% and 23.5% compared to the second best method VELVET. The above results further confirm the superiority of MatsVD in the statement-level VD task.

Ranking Metrics. By considering the VD task as a ranking problem, the ranking metrics are capable of more comprehensively evaluating the detection capability of the model. Intuitively, the approach that ranks the defective statements more forward better captures the vulnerability characteristics. The experimental results of MatsVD and the baselines on classification metrics are shown in Table 2. The results show that MatsVD consistently outperforms other baselines on all ranking metrics. Specifically, MatsVD achieves 2.46 and 6.34 in MFR and MAR, respectively, compared to the second best method VELVET, which gains 3.73 and 7.81. When it comes to Top-k metrics, we found that VELVET and MatsVD both reach relatively high Top-5 (93.86% versus 94.82%). However, MatsVD significantly surpasses VELVET on Top-1 with 6.68% more statements are ranked at Top-1. This phenomenon reflects MatsVD's robustness in identifying vulnerable statements, especially in distinguishing the subtle differences in semantics between statements.

Answer to RQ1: MatsVD significantly outperforms all baselines on binary classification and ranking metrics at the statement level, which improves by respectively 22.97% and 7.76% in terms of the F1 and Top-1 Accuracy over the SOTA methods.

5.2 RQ2: Ablation Study

MatsVD comprises three components: statement embedding layer (SE), dependency-based statement representation layer (SR), and function representation layer (FR). We assess the effectiveness of each component of MatsVD by respectively modifying each layer. For SE, we replace the pre-trained model CodeT5 with three different variants: **SE-CodeBERT**, **SE-AvgPooling**, and **SE-RNN**. Specifically, the first variant replaces CodeT5 with another pre-trained model CodeBERT [10] to generate statement embeddings. The latter two variants use only the embedding layer of the pre-trained CodeT5 to initialize the embedding vector for each token and then aggregate them into statement embeddings using average

Table 3: Ablation study result of MatsVD on the classification metrics.

Method	Two-Phase Detection								One-Phase Detection			
	Function Level				Statement Level				Statement Level			
	Mcc	Pre	Re	F1	Mcc	Pre	Re	F1	Mcc	Pre	Re	F1
SE-CodeBERT	0.9270	96.38	89.77	92.95	0.8092	85.32	77.09	81.00	0.7956	81.01	82.61	81.80
SE-AvgPooling	0.8983	94.70	86.04	90.16	0.7518	80.66	70.51	75.25	0.8017	84.85	79.76	82.23
SE-RNN	0.9010	93.79	87.36	90.46	0.7741	79.91	75.42	77.60	0.7520	72.80	83.72	77.89
SR-NoMask	0.9288	96.49	89.53	92.88	0.8052	85.54	76.14	80.57	0.7734	76.89	83.03	79.84
FR-AvgPooling	0.9290	96.39	90.13	93.15	0.8101	82.40	80.01	81.19	0.7205	64.88	87.81	74.62
FR-RNN	0.9263	96.49	89.53	92.88	0.8023	81.61	79.25	80.41	0.7105	63.23	**88.10**	73.63
MatsVD (ours)	**0.9361**	**97.53**	**90.37**	**93.81**	**0.8606**	**91.57**	**81.13**	**86.03**	**0.8585**	**89.90**	84.86	**87.31**

Table 4: Ablation study result of MatsVD on the ranking metrics.

Method	Top-1	Top-3	Top-5	MFR	MAR
SE-CodeBERT	89.53	93.38	94.36	2.63	6.62
SE-AvgPooling	88.68	92.90	94.08	2.96	6.71
SE-RNN	89.29	93.26	94.34	3.04	7.15
SR-NoMask	90.49	91.81	92.17	3.17	6.85
FR-AvgPooling	90.85	93.50	94.05	2.94	6.69
FR-RNN	90.49	92.90	93.98	2.66	6.75
MatsVD (ours)	**92.77**	**94.46**	**94.82**	**2.46**	**6.34**

pooling or an RNN. For SR, we remove the masked-attention mechanism in MatsVD and denote this variant as **SR-NoMask**. For FR, we compare it with two variants: **FR-AvgPooling** and **FR-RNN**, which use average pooling and an RNN to aggregate the statement representations into the function representations, respectively.

Classification Metrics. Table 3 presents the performance of MatsVD and its variants on the classification metrics. The results show that MatsVD outperforms six variants on the most metrics except for Recall of one phase detection. In terms of statement embedding approaches, the statement-level F1 score of the SE-CodeBERT decreases by 6.21% and 6.74% under the two scenarios, respectively. This suggests that the pre-trained model CodeT5 in MatsVD contributes to converting textual source code into more meaningful vector representations compared to CodeBERT. Meanwhile, MatsVD achieves 10.78%/5.08% and 8.43%/9.42% absolute improvement on statement-level F1-score under the two scenarios compared to SE-AvgPooling/RNN. This trend is similar to the comparison with the FR-AvgPooling/RNN, which indicates that the Transformer-based aggregation we proposed is more effective than average pooling or RNN. In addition, the performance of SR-NoMask decreases sharply in both two detection scenarios compared with MatsVD, with the F1 score dropping by 6.78% and 9.36%, respectively. It's MCC also similarly decreases by 7% and 11%, respectively. This result indicates that masked attention plays an important role in statement-level VD.

On the function level, we observe that MatsVD consistently outperforms all its variants. However, the performance loss of each variant is marginal compared to the situation at the statement level, with the absolute reduction of F1 score not more than 4%. To explain this, we notice that compared to the statement-level VD which concentrates on more local features within one statement, the function-level VD considers the global features within the whole function. Therefore, it is easier for the function-level VD to capture more coarse vulnerable features compared to the statement-level VD, which is more difficult to identify fine-grained features.

Ranking Metrics. Table 4 presents the performance of MatsVD and its variants on the ranking metrics. We can observe from the table that all of the variants fall short in prioritizing the vulnerable statements, especially on Top-1 and Top-3. Specifically, MatsVD improves the best performance of different variants by 2.1%, 6.9%, 4.4% on Top-1, MFR, and MAR, respectively. This phenomenon validates the importance of each component in MatsVD for more precisely detecting problematic statements.

Answer to RQ2: All three layers of MatsVD play important roles in vulnerability detection, where the statement level is more affected than the function level.

5.3 RQ3: Impact of Masked Attention and Transformer-Based Aggregation

To explore the impact of masked attention and Transformer-based aggregation on detecting vulnerable statements, we first conducted a quantitative study on the statement-level results under the two-phase detection scenario. Figure 4 presents the F1 score of the four models on functions with different lengths. The results show that MatsVD outperforms the other variants on all the intervals. In particular, compared to the other three intervals, MatsVD achieves the largest increase in F1 score for long functions with 120 to 155 lines, with an absolute significant improvement of 11.4%, 11.8%, and 12.7% over SE-AvgPooling, SR-NoMask, and FR-AvgPooling, respectively. On the one hand, these results indicate that the involvement of masked attention is able to exploit dependencies in the function to adjust the allocation of attention, which reduces the noise information of irrelevant statements (e.g., independent variable declarations or return statements) and highlights critical statements in a long context. On the other hand, compared to the FR-AvgPooling (SE-AvgPooling), the Transformer-based aggregation approach preserves the information of significant statements (tokens) and captures the long-term dependencies of sequences in generating function (statement) representation. This also enhances the capability of the model to detect long vulnerability functions.

To better understand how MatsVD detects vulnerable statements more precisely, we conducted a qualitative study to evaluate

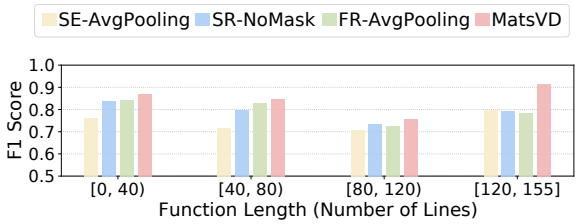


Figure 4: Statement-level F1 score on vulnerable functions with different lengths.

the advantage of MatsVD against other baselines. Specifically, we choose three baselines based on primitive self-attention including LineVul[13], LineVD[14], and VELVET[8]. Figure 5 shows a real-world vulnerable function from the Linux kernel (CVE-2013-4592), which contains a *Resource Management Errors* vulnerability that can lead to a memory leak. Statements at lines 109, 110, 111, and 114 (denoted as s109, s110, s111, and s114, respectively) are labeled as vulnerable in the Big-Vul dataset. For this example, we list the statement-level detection results of each method, where the vulnerable statements predicted by the VD methods are denoted with 1. Overall, MatsVD correctly detects all four vulnerable statements, while LineVul, LineVD, and VELVET failed to identify all of them.

LineVul performs the worst by only correctly identifying statements s110 and s114 and producing the most false positives. Intuitively, this may due to the fact that LineVul regards the source code as a sequence of tokens, which results in attention being spread across individual tokens and makes it difficult for the model to concentrate on important statements in the source code. Additionally, it also fails to leverage the dependency information between statements, and relies solely on the features of prominent tokens within the statements to indirectly predict vulnerable statements.

LineVD correctly identifies only s110 and s111 but fails to detect s109 and s114 that have data and control dependencies with them. Considering that LineVD utilizes a graph attention network (GAT) [27] model to learn data and control dependencies in the PDGs of source code, we argue that the unsatisfactory performance of LineVD may be caused by the intrinsic limitation of the GNNs. Many studies have proved that GNNs are facing the *over smoothing* [3] problem, which means that GNNs are better at capturing local features than learning global information.

VELVET achieves the second-best performance by correctly detecting three vulnerable statements except s111. Although VELVET combines Transformer and GNN to separately capture global and local semantic information for locating vulnerable statements, it suffers from the possible contradictory results of the two models [8]. This may be the reason why it does not detect s111 as vulnerable.

MatsVD correctly detects all the vulnerable statements with the highest accuracy. We attribute it to the following reasons: (1) MatsVD is implemented on top of the Transformer architecture. Due to the exceptional scalability of the Transformer, it can better capture dependency information for long sequences compared to GNN and RNN. This enables MatsVD to accurately and comprehensively understand vulnerability functions. (2) MatsVD adopts a unified three-layer Transformer model structure. Compared to the

File: virt/kvm/kvm_main.c
CVE Page: https://www.cvedetails.com/cve/CVE-2013-4592/

	LN	LV	LD	VT	MD	GT
1 int __kvm_set_memory_region(struct kvm *kvm, struct kvm_userspace_memory_region *mem, int user_alloc)	1	1	0	0	0	0
2 {	2	0	0	0	0	0
3 int r;	3	0	0	0	0	0
4 gfn_t base_gfn;	4	0	0	0	0	0
5 unsigned long npages;	5	0	0	0	0	0
6 r = check_memory_region_flags(mem);	6	1	0	1	0	0
...	...	0	0	0	0	0
102 rcu_assign_pointer(kvm->memslots, slots);	102	0	0	0	1	0
103 synchronize_srcu_expedited(&kvm->srcu);	103	0	1	0	0	0
104 kvm_arch_flush_shadow_memslot(kvm, slot);	104	0	0	0	0	0
105 kfree(old_memslots);	105	0	0	0	0	0
106 r = kvm_arch_prepare_memory_region(kvm, &new, old, mem, user_alloc);	106	1	0	1	0	0
107 if (r)	107	0	0	0	0	0
108 goto out_free;	108	0	0	0	0	0
109 if (npages) {	109	0	0	1	1	1
110 r = kvm_iommu_map_pages(kvm, &new);	110	1	1	1	1	1
111 if (r)	111	0	1	0	1	1
112 goto out_free;	112	0	0	0	0	0
113 } else	113	0	0	0	0	0
114 kvm_iommu_unmap_pages(kvm, &old);	114	1	0	1	1	1
115 r = -ENOMEM;	115	1	0	0	0	0
...	...	0	0	0	0	0
128 }	128	0	0	0	0	0

LN: LineNo, **LV**: LineVul, **LD**: LineVD, **VT**: VELVET, **MD**: MatsVD, **GT**: Ground Truth

Figure 5: Qualitative study of MatsVD compared to baselines.

state-of-the-art baseline VELVET which is based on hybrid models, MatsVD avoids compatibility issues between different model architectures to produce more stable results. (3) MatsVD refines the original attention computation of the Transformer by selectively masking partial attention based on data and control dependencies between statements. The dependency-based attention guides the model to focus more on prominent statements in the function and leverage dependency information in the source code, which helps the model to identify vulnerable statements more effectively.

Answer to RQ3: Masked attention and Transformer-based aggregation work together to effectively capture data/control dependencies between statements within the function to help better identify vulnerable statements.

6 THREATS TO VALIDITY

Threats to internal validity. The main internal threat is related to the process of PDG generation. To mitigate this threat, we used a popular code analysis tool Joern [31] to generate PDGs and discard functions not successfully parsed by Joern from our experiments. We manually checked 300 generated PDGs to ensure the correctness of the extracted data and control dependencies in the function.

Threats to external validity. The main external threat derives from the quality of the dataset [5]. To mitigate this threat, we selected the widely used practical vulnerability dataset Big-Vul and obtain statement-level ground truth using heuristics following existing works [14, 17]. Moreover, all the projects in the dataset are developed in C/C++ programming language. The effectiveness of our approach on other programming languages such as Java and Python remains to be explored. We will collect and explore more diverse datasets in future work.

7 RELATED WORK

To obtain more fine-grained detection results, various statement-level VD approaches have been proposed recently. ICVH[21] leverages bidirectional RNNs to identify statements in functions that are

highly relevant to vulnerabilities through maximizing mutual information. Li et al. [17] proposed IVDetect, which employs a graph neural network for function-level detections, coupled with a GNNExplainer to pinpoint the most influential sub-graph contributing to these detections. Hin et al. [14] proposed LineVD, which represents the source code as a Program Dependency Graph (PDG) and employs a Graph Attention Network to learn the statement representations. LineVul [13] utilized Transformer to capture long-term dependencies in code sequence and regarded the statements corresponding to higher attention scores as vulnerable lines. VELVET employs an ensemble learning strategy that individually computes a vulnerability score for each statement with a Transformer and a GNN, and then takes the average of the two scores as the final vulnerability score for a statement.

In addition, there have been works adopting explainable AI models for line-level defect prediction. Pornprasit et al. [22] proposed to leverage the attention mechanism for line-level defect prediction. Thongtanunam et al. [29] proposed a machine learning-based approach with LIME model-agnostic technique to predict which lines are likely to be defective in the future. MatsVD differs from these methods in that we use line-level labels to guide the model in learning vulnerability features for detection rather than using model interpretability to get predicted lines.

8 CONCLUSION AND FUTURE WORK

In this work, we propose MatsVD, a novel Transformer-based approach for statement-level vulnerability detection. MatsVD selectively masks partial attention inside the Transformer based on the data and control dependencies between the statements to better capture the semantic information of the source code. The evaluation on a large real-world dataset shows that MatsVD outperforms the existing DL-based methods on statement-level vulnerability detection in terms of both binary classification and ranking metrics. Specifically, MatsVD achieves 22.97% higher F1 score and 7.76% higher Top-1 Accuracy than the state-of-the-art method VELVET.

Our future work will investigate how to integrate MatsVD into existing software development lifecycles to help developers find vulnerable lines in a time-saving way.

REFERENCES

[1] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2022. Deep Learning Based Vulnerability Detection: Are We There Yet? *IEEE Transactions on Software Engineering* 48, 9 (2022), 3280–3296.

[2] Checkmarx. 2024. [Online]. Available: https://checkmarx.com/.

[3] Deli Chen, Yankai Lin, Wei Li, Peng Li, Jie Zhou, and Xu Sun. 2020. Measuring and relieving the over-smoothing problem for graph neural networks from the topological view. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 34. 3438–3445.

[4] Cppcheck. 2024. [Online]. Available: https://cppcheck.sourceforge.io/.

[5] Roland Croft, M. Ali Babar, and M. Mehdi Kholoosi. 2023. Data Quality for Software Vulnerability Datasets. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 121–133.

[6] CVE. 2024. [Online]. Available: http://cve.mitre.org/.

[7] National Vulnerability Database. 2024. [Online]. Available: https://nvd.nist.gov/.

[8] Yangruibo Ding, Sahil Suneja, Yunhui Zheng, Jim Laredo, Alessandro Morari, Gail Kaiser, and Baishakhi Ray. 2022. VELVET: a noVel Ensemble Learning approach to automatically locate VulnErable sTatements. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 959–970.

[9] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR '20)*. Association for Computing Machinery, 508–512.

[10] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, and Linjun Shou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*.

[11] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The program dependence graph and its use in optimization. (1987).

[12] FlawFinder. 2024. [Online]. Available: http://www.dwheeler.com/FlawFinder.

[13] Michael Fu and Chakkrit Tantithamthavorn. 2022. LineVul: a transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories (MSR '22)*. 608–620.

[14] David Hin, Andrey Kan, Huaming Chen, and M. Ali Babar. 2022. LineVD: statement-level vulnerability detection using graph neural networks. In *Proceedings of the 19th International Conference on Mining Software Repositories (MSR '22)*. Association for Computing Machinery, 596–607.

[15] Bo Jiang, Ziyan Zhang, Doudou Lin, Jin Tang, and Bin Luo. 2019. Semi-Supervised Learning With Graph Learning-Convolutional Networks. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 11305–11312.

[16] Yoon Kim. 2014. Convolutional Neural Networks for Sentence Classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 1746–1751.

[17] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. Vulnerability detection with fine-grained interpretations *(ESEC/FSE 2021)*. 292–303.

[18] Zhuo Li, Huangzhao Zhang, and Zhi Jin. 2023. WELL: Applying Bug Detectors to Bug Localization via Weakly Supervised Learning. *ArXiv* abs/2305.17384 (2023).

[19] 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. In *Proceedings 2018 Network and Distributed System Security Symposium (NDSS 2018)*. Internet Society.

[20] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2 (NIPS'13)*. Curran Associates Inc., 3111–3119.

[21] Van Nguyen, Trung Le, Olivier De Vel, Paul Montague, John Grundy, and Dinh Phung. 2021. Information-theoretic Source Code Vulnerability Highlighting. In *2021 International Joint Conference on Neural Networks (IJCNN)*. 1–8.

[22] Chanathip Pornprasit and Chakkrit Kla Tantithamthavorn. 2023. DeepLineDP: Towards a Deep Learning Approach for Line-Level Defect Prediction. *IEEE Transactions on Software Engineering* 49, 1 (2023), 84–98.

[23] 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 (Volume 1: Long Papers)*. Association for Computational Linguistics, 1715–1725.

[24] Cyber Security Vulnerability Statistics. 2024. [Online]. https://www.comparitech.com/blog/information-security/cybersecurity-vulnerability-statistics/.

[25] Juliet test suite v1.3 NIST Software Assurance Reference Dataset. 2024. [Online]. Available: https://samate.nist.gov/SARD/test-suites/.

[26] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS'17)*. Curran Associates Inc., 6000–6010.

[27] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, et al. 2017. Graph Attention Networks. *ArXiv* abs/1710.10903 (2017).

[28] Yue Wang, Weishi Wang, Shafiq Joty, Steven C.H. Hoi, et al. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*.

[29] Supatsara Wattanakriengkrai, Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Hideaki Hata, and Kenichi Matsumoto. 2022. Predicting Defective Lines Using a Model-Agnostic Technique. *IEEE Transactions on Software Engineering* 48, 5 (2022), 1480–1496.

[30] Xin-Cheng Wen, Yupan Chen, Cuiyun Gao, Hongyu Zhang, Jie M. Zhang, and Qing Liao. 2023. Vulnerability Detection with Graph Simplification and Enhanced Graph Representation Learning. In *Proceedings of the 45th International Conference on Software Engineering (ICSE '23)*. IEEE Press, 2275–2286.

[31] Joern: The Bug Hunter's Workbench. 2024. [Online]. Available:https://joern.io/.

[32] 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*. 590–604.

[33] Rex Ying, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. 2019. *GNNExplainer: generating explanations for graph neural networks*.

[34] Yunhui Zheng, Saurabh Pujar, Burn Lewis, Luca Buratti, Edward 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 *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 111–120.

[35] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. *Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks*. Curran Associates Inc.