

VulProbe: Probe the Hidden Representation to Localize the Vulnerability

Xiangping Chen

Sun Yat-sen University

Yuxuan Zhang

Sun Yat-sen University

Mingxuan Chen

Sun Yat-sen University

Yuan Huang

`huangyuan5@mail.sysu.edu.cn`

Sun Yat-sen University

Research Article

Keywords: Vulnerability localization, Probing, Hidden representation, Abstract Syntax Tree, Mask-based weighting

Posted Date: October 5th, 2026

DOI: <https://doi.org/10.21203/rs.3.rs-10677302/v1>

License: © ⓘ This work is licensed under a Creative Commons Attribution 4.0 International License.

[Read Full License](#)

Additional Declarations: No competing interests reported.

VulProbe: Probe the Hidden Representation to Localize the Vulnerability

Xiangping Chen^{1,2}, Yuxuan Zhang¹, Mingxuan Chen¹,
Yuan Huang^{1*}

¹School of Software Engineering, Sun Yat-sen University, Zhuhai, China.

²School of Journalism and Communication, Sun Yat-sen University,
Zhuhai, China.

*Corresponding author(s). E-mail(s): huangyuan5@mail.sysu.edu.cn;

Contributing authors: chenxp8@mail.sysu.edu.cn;

zhangyx568@mail2.sysu.edu.cn; chenmx223@mail2.sysu.edu.cn;

Abstract

Vulnerability localization aims to identify the specific lines of code that contain vulnerabilities, thereby enabling developers to address security issues more efficiently and promptly. While recent methods have employed attention weights (e.g., LineVul) or feature attribution techniques (e.g., Line-DP) to guide vulnerability localization based on detection outcomes, these methods may rely on signals that are not faithfully reflective of the model’s decision process. In this paper, we propose VulProbe, a novel method that probes the hidden representations of a vulnerability detection model to achieve line-level vulnerability localization. The hidden representations that are input to discriminate vulnerabilities encode rich information about the source code. By probing these representations, VulProbe identifies syntactic structures that are highly associated with the model’s predictive decisions and subsequently localizes the vulnerable lines. Specifically, we first fine-tune CodeBERT on a vulnerability detection task to ensure that the model learns vulnerability-related features. Next, we probe the syntactic information embedded in the hidden representations using a mask-based weighting mechanism that emphasizes vulnerability-related token pairs during probe training, and transform the probed information into Abstract Syntax Trees (ASTs). By comparing the predicted AST (P-AST) with the original AST (O-AST) derived from the input source code, we identify key syntactic structures associated with vulnerabilities and leverage these insights to localize vulnerable lines. To evaluate the performance of our VulProbe, we compare it with eight baseline methods using six evaluation metrics. VulProbe, at the median value, achieves Top-1 Accuracy of 0.2586, Top-3 Accuracy of

0.4224, Top-5 Accuracy of 0.4914, IFA of 6.03, Recall@1%Effort of 0.422414, and Effort@20%Recall of 0.003951. It improves the baselines by 108.21% – 660.03%, 52.47% – 888.07%, 25.76% – 723.47%, 17.15% – 57.59%, 62.58% – 9950.26%, and 51.28% – 96.78%, respectively. Statistical analysis further confirms that the improvements achieved by VulProbe over the baseline methods are significant.

Keywords: Vulnerability localization, Probing, Hidden representation, Abstract Syntax Tree, Mask-based weighting

1 Introduction

Software systems have become the cornerstone of modern society, and vulnerabilities in code pose a serious potential risk [Jang-Jaccard and Nepal \(2014\)](#); [National Vulnerability Database \(2026\)](#). To guarantee the security of software systems, researchers have proposed many approaches for detecting vulnerabilities in code. Existing vulnerability detection approaches can be broadly classified into two categories: program analysis-based (PA-based) approaches [Cppcheck Team \(2026\)](#); [Ferschke et al. \(2012\)](#); [Secure Software, Inc. \(2026\)](#) and machine learning-based (ML-based) approaches [Fu and Tantithamthavorn \(2022\)](#); [Guo et al. \(2022\)](#); [Li et al. \(2021, 2018\)](#); [Pornprasit and Tantithamthavorn \(2022\)](#); [Wattanakriengkrai et al. \(2020\)](#); [Wu et al. \(2022\)](#); [Zhang et al. \(2023\)](#); [Zhou et al. \(2019\)](#).

The PA-based approaches rely on manually defined rules crafted by security experts to identify vulnerabilities. In contrast, ML-based approaches leverage large-scale data to achieve better generalization and detection performance. However, current ML-based vulnerability detection approaches are commonly limited to coarse-grained vulnerability detection.

Many works [Li et al. \(2018\)](#); [Wu et al. \(2022\)](#); [Zhang et al. \(2023\)](#); [Zhou et al. \(2019\)](#) have implemented vulnerability detection approaches that demonstrate promising performance. Nevertheless, these methods still require developers to invest significant time and effort to further localize vulnerabilities to specific lines of code. According to our analysis of the BigVul dataset [Fan et al. \(2020\)](#), vulnerable lines constitute only about 5% of the total number of code lines within the functions labeled as vulnerable. Accurately localizing vulnerable lines would significantly reduce the workload associated with vulnerability identification and enable developers to address vulnerabilities more efficiently and in a timely manner.

Building on the promising performance of existing vulnerability detection methods [Li et al. \(2018\)](#); [Wu et al. \(2022\)](#); [Zhang et al. \(2023\)](#); [Zhou et al. \(2019\)](#), studies [Fu and Tantithamthavorn \(2022\)](#); [Li et al. \(2021\)](#); [Wattanakriengkrai et al. \(2020\)](#) have attempted to utilize model interpretation approaches [Ribeiro et al. \(2016\)](#); [Lundberg and Lee \(2017\)](#); [Shrikumar et al. \(2017\)](#); [Vaswani et al. \(2017\)](#); [Ying et al. \(2019\)](#) such as the attention mechanism and the feature attribution techniques for vulnerability localization. While these methods offer insights into the model’s focus areas, they have inherent limitations. Previous studies [Bai et al. \(2021\)](#); [Jain and Wallace \(2019\)](#) have shown that attention scores can, to some extent, reflect the model’s focus

on input tokens and reveal their correlation with the final output. However, a high attention score does not necessarily imply a causal relationship between the corresponding token and the final decision. Similarly, feature attribution methods typically provide correlation rather than guaranteeing causality, making it difficult to identify the fundamental factors driving model decisions [Cheng et al. \(2024\)](#).

Motivated by these limitations, we focus on the model’s hidden representations for vulnerability localization. In typical deep learning-based approaches to vulnerability detection, the input source code is first encoded into high-dimensional vector representations, referred to as hidden representations. These representations capture a wide range of vulnerability-related features, including syntactic structures, semantic dependencies, and additional indicators that contribute to the model’s final predictions. The refined hidden representations are then passed to a classifier, which utilizes them to determine whether the given code contains vulnerabilities. As these representations are directly correlated with the model’s final predictions, analyzing their key components allows us to identify the evidence that the model relies on when making predictions, thereby enabling faithful post-hoc interpretation.

Building on this idea, we propose VulProbe, a post-hoc explanation framework that probes the hidden representations of a vulnerability detection model to localize vulnerabilities. Specifically, we first fine-tune CodeBERT [Feng et al. \(2020\)](#) on a vulnerability detection task to ensure that the model captures vulnerability-related features in its hidden representations. These refined representations are then leveraged by our probe model [Belinkov et al. \(2017\)](#); [López et al. \(2022\)](#); [Hewitt and Manning \(2019\)](#); [Karmakar and Robbes \(2021\)](#); [Troshin and Chirkova \(2022\)](#); [Wan et al. \(2022\)](#) to reconstruct Abstract Syntax Trees (ASTs) that reflect the syntactic structure encoded within them. To enhance the probe’s ability to capture vulnerability-related syntactic structures, we introduce a mask-based weighting mechanism during probe training that emphasizes token pairs involving vulnerable code lines, thereby guiding the probe to focus on learning critical syntactic patterns for vulnerability detection. As a result of fine-tuning, vulnerability-related features are amplified while irrelevant features are suppressed. These changes in syntactic features are manifested in the extracted P-AST, where key vulnerability-related structures are effectively retained. Subsequently, we compare the predicted AST (P-AST) with the original AST (O-AST) of the input source code to identify the key syntactic elements associated with vulnerabilities. By mapping these elements back to their corresponding code lines, we assign scores to each line and rank them accordingly, thereby enabling precise vulnerability localization.

We conduct experiments to evaluate our proposed approach, VulProbe, on the widely used BigVul dataset [Fan et al. \(2020\)](#). To assess its performance in the vulnerability localization task, we compare VulProbe against eight baseline approaches [Ferschke et al. \(2012\)](#); [Fu and Tantithamthavorn \(2022\)](#); [Hin et al. \(2022\)](#); [Hurst et al. \(2024\)](#); [Lundberg and Lee \(2017\)](#); [Secure Software, Inc. \(2026\)](#); [Shrikumar et al. \(2017\)](#); [Simonyan et al. \(2013\)](#). The experimental results demonstrate that VulProbe outperforms these baselines. Specifically, VulProbe, at the median value, achieves Top-1 Accuracy of 0.2586, Top-3 Accuracy of 0.4224, Top-5 Accuracy of 0.4914, IFA of 6.03, Recall@1%Effort of 0.422414, and Effort@20%Recall of 0.003951. Compared to

the baseline methods, VulProbe improves performance by 108.21% to 660.03% for Top-1 Accuracy, 52.47% to 888.07% for Top-3 Accuracy, 25.76% to 723.47% for Top-5 Accuracy, 17.15% to 57.59% for IFA, 62.58% to 9950.26% for Recall@1%Effort, and 51.28% to 96.78% for Effort@20%Recall. Statistical analysis further confirms that the improvements achieved by VulProbe over the baseline methods are significant.

To facilitate research and application, the replication package of our VulProbe is available at: <https://github.com/alfulort/VulProbe>.

Scope and interpretation of our approach. VulProbe is designed as a post-hoc interpretation method that translates the internal states of a vulnerability detection model into human-interpretable code locations. Rather than claiming to discover causal relationships between code patterns and vulnerabilities, VulProbe focuses on recovering syntactic evidence encoded in the model’s hidden representations and mapping this evidence back to source code lines. The output of VulProbe is a ranked list of lines indicating where the model “looks” when making a vulnerability prediction. This serves as a practical signal for developers to prioritize code review, even though it reflects correlation with, rather than causation of, the model’s decision process.

In summary, this paper makes the following contributions:

- **A Novel Approach for Vulnerability Localization.** We propose VulProbe, a post-hoc explanation framework that probes hidden representations of vulnerability detection models to identify syntactic structures that are highly correlated with the model’s vulnerability predictions. By analyzing these structures, VulProbe enables precise vulnerability localization. To the best of our knowledge, this is the first application of the probing techniques in the context of vulnerability localization.
- **Architecture-Agnostic Vulnerability Localization.** VulProbe is architecture-agnostic and can be integrated with any vulnerability detection model that provides access to its hidden representations. By extracting syntactic information from these representations, VulProbe enables precise vulnerability localization. Its flexibility allows it to be applied across various architectures, including transformer-based models (e.g., CodeBERT, UniXcoder) and traditional neural networks (e.g., BiLSTM, MLP). We note that this assumption holds for open-source and white-box models, which are the primary focus of explainability research in software engineering.
- **Comprehensive Performance Evaluation.** We evaluate VulProbe on the widely used BigVul dataset [Fan et al. \(2020\)](#) using six evaluation metrics. Experimental results show that VulProbe consistently outperforms all eight baseline methods, achieving statistically significant improvements across all metrics.

The rest of our paper is organized as follows. Section 2 introduces the details of our approach. Section 3 describes the experiment setup. Section 4 reports the results of the experiments. Section 5 presents the threats to the validity of our study. Section 6 introduces the related work. We draw conclusions in Section 7.

2 Approach

In this section, we introduce VulProbe. Figure 1 shows the overall framework of our approach. VulProbe consists of three main phases:

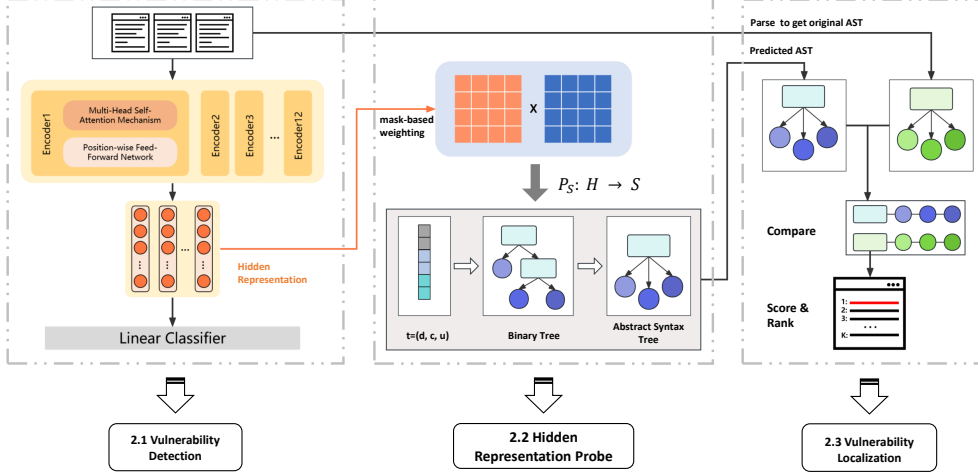


Fig. 1 The overall framework of VulProbe.

1. **Vulnerability Detection.** Firstly, we employ pre-trained model for vulnerability detection and fine-tune the model with an appended linear classifier to generate the final detection results. During this process, the hidden representations, which capture the features of the input code, are directly utilized by the linear classifier for prediction.
2. **Hidden Representation Probe.** Subsequently, we learn an orthogonal projection to map the hidden representation of the input code into a syntactic subspace. After the projection, the hidden representation is transformed into a vector tuple t . To enhance the probe’s ability to capture vulnerability-related syntactic structures, we introduce a mask-based weighting mechanism during probe training. Due to the established correspondence between syntactic subspace vector tuples and ASTs, we can convert the projected tuples into ASTs. The resulting tree structure offers an intuitive representation of the syntactic information encoded in the hidden representations.
3. **Vulnerability Localization.** Finally, after obtaining the predicted AST (P-AST) from the hidden representations, we parse the input code to generate its original AST (O-AST). Compared to the O-AST, the P-AST retains only a subset of the full syntactic structure, indicating that the hidden representations encode only limited syntactic information relevant for vulnerability detection. We consider these preserved syntactic elements critical for vulnerability detection. Therefore these

retained syntactic structures can serve as the key evidence for vulnerability localization. By comparing the P-AST with the O-AST, we can identify the key syntactic components of the input code and, consequently, localize the vulnerability.

2.1 Vulnerability Detection

VulProbe employs a Transformer-based pre-trained model for vulnerability detection. In our work, we perform vulnerability detection by attaching a linear classifier to a pre-trained CodeBERT [Feng et al. \(2020\)](#) model and fine-tuning the entire model on our dataset. CodeBERT, built upon the BERT architecture [Devlin \(2018\)](#), consists of 12 stacked encoder layers and is pre-trained on large-scale code datasets [Husain et al. \(2019\)](#). This pre-training enables it to learn general programming language representations that serve as effective features for downstream tasks. Fine-tuning on the vulnerability detection task further adapts these hidden representations to capture task-specific features, thereby enhancing their effectiveness for vulnerability detection.

The 12 stacked encoder layers play a crucial role in progressively integrating and refining features extracted from the input. Within each encoder layer, the multi-head self-attention and position-wise feed-forward network work together to capture and encode rich structural and semantic information.

The self-attention mechanism [Vaswani et al. \(2017\)](#) calculates the relevance of each token to every other token using three components: query (Q), key (K), and value (V). Here, Q represents the current token, K encompasses all tokens in the sequence, and V contains the token information to be aggregated. The attention operation is defined as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V, \quad (1)$$

where d_k is a scaling factor that stabilizes gradients by normalizing the dot product scores. This mechanism enables the model to capture long-range dependencies and contextual relationships, thereby enabling the hidden representations to capture and encode rich structural and semantic information from the source code.

Additionally, the encoder layer employs multiple independent attention heads in parallel. This multi-head self-attention mechanism allows the model to learn distinct subspace representations and capture various aspects of the input. Each attention head is computed independently as:

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V), \quad (2)$$

and the outputs are concatenated and projected back to the desired dimensionality with W^O :

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O. \quad (3)$$

This process effectively integrates features from different subspaces into the hidden representations, enhancing the model’s ability to capture complex relationships in the code.

Following self-attention, a position-wise feed-forward network (FFN) applies a nonlinear transformation to further refine the contextualized features:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2. \quad (4)$$

This further processing elevates the hidden representations into higher-level abstract features, thereby enhancing the model’s ability to distinguish the vulnerability.

While the stacked encoder layers extract hierarchical representations from the source code, these features alone may not be optimal for vulnerability detection. Fine-tuning further refines these representations by reinforcing vulnerability-specific patterns and filtering out irrelevant information, ultimately enhancing their discriminative power.

Finally, these refined hidden representations not only encode critical features for vulnerability detection but also serve as direct inputs to the classifier, forming the basis for the model’s predictions. Inspired by this, we aim to analyze and interpret the encoded information in hidden representations to recover explanatory evidence associated with vulnerability detection. This requires access to the model’s intermediate layer outputs, which is a standard assumption for white-box interpretation methods in the software engineering XAI literature. Ultimately, this deeper understanding can enable the vulnerability localization.

2.2 Hidden Representation Probe

The final hidden representation $\mathbf{h}$ in the vulnerability detection model encodes essential information for vulnerability discrimination. Analyzing $\mathbf{h}$ can reveal what information the model encodes and how that information influences detection results. To faithfully interpret what is encoded in $\mathbf{h}$, we introduce the probing method [Belinkov \(2022\)](#); [Belinkov et al. \(2017\)](#); [López et al. \(2022\)](#); [Hewitt and Manning \(2019\)](#); [Karmakar and Robbes \(2021\)](#); [Troshin and Chirkova \(2022\)](#); [Wan et al. \(2022\)](#). Unlike prior work that attempts to establish causality, our goal is to recover the syntactic structures that are most strongly associated with the model’s predictions, providing a faithful explanation of the model’s reasoning. This approach helps identify the features captured by the model, offering insights into the reasoning behind its predictions.

Let H denote the set of all hidden representations $\mathbf{h}$, where each $\mathbf{h} \in H$ is a high-dimensional encoding of the source code. Although H is rich in information, its content is mixed. To avoid interference from other aspects of information, in this paper we focus exclusively on probing the syntactic information within the hidden representations. We hypothesize that there exists a syntactic subspace S that encodes AST information. The objective of our probe is to learn a projection P_S that projects the hidden representation space H to the syntactic subspace S :

$$P_S : H \rightarrow S. \quad (5)$$

2.2.1 Transformation between AST and Vector Tuple

However, mapping hidden representations directly to ASTs is challenging because ASTs are hierarchical structures rather than simple vectors. To address this, we first

transform ASTs into a structured vector representation, thereby establishing a direct learning objective for P_S . L’opez et al. López et al. (2022) have verified the existence of the syntactic subspace and established a bijection between an AST and a vector tuple representation $\mathbf{t} = (\mathbf{d}, \mathbf{c}, \mathbf{u})$, where $\mathbf{d}$, $\mathbf{c}$ and $\mathbf{u}$ collectively represent the AST structure.

Following the rules outlined in López et al. (2022); Shen et al. (2018), there is a bidirectional transformation between an AST and its vector tuple representation $\mathbf{t} = (\mathbf{d}, \mathbf{c}, \mathbf{u})$. As shown in Figure 2, this transformation proceeds in two steps:

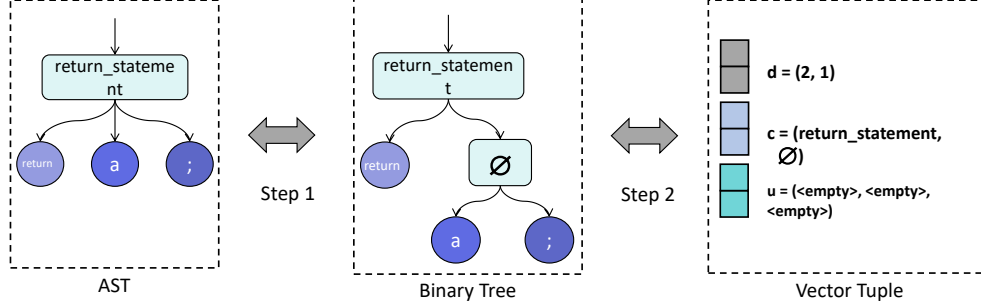


Fig. 2 An example of the 2-step transformation between AST and vector tuple. The rounded rectangles are nonterminal nodes and the circles are terminal nodes.

Step 1: AST $\rightarrow$ Binary Tree Transformation. To convert an AST into a binary tree, we need to binarize both unary and n -ary nodes.

- If a node is n -ary, a special node $\emptyset$ is inserted to binarize it. For instance, the 3-ary non-terminal node return statement is transformed into binary structure by adding $\emptyset$.
- As for the unary node, it should be merged with its child to form a new node. This rule applies to both terminal and non-terminal nodes.
- To revert the binary tree back to the AST, remove the $\emptyset$ node and reconnect the tree. Additionally, expand any unary nodes that were compressed during binarization.

Step 2: Binary Tree $\rightarrow$ Vector Tuple Transformation. Next, we convert the binary tree into a tuple $\mathbf{t} = (\mathbf{d}, \mathbf{c}, \mathbf{u})$ to finally complete the transformation from AST to vector tuple representation.

- $\mathbf{d}$ encodes the structure information. Given the terminal words $\mathbf{w} = (w_0, w_1, \dots, w_n)$ in the binary tree, $\mathbf{d}$ is defined as $d_0, d_1, \dots, d_{n-1}$, where each d_i denotes the height of the lowest common ancestor for two terminal nodes (w_i, w_{i+1}) .
- $\mathbf{c}$ encodes the label information. $\mathbf{c}$ stores the label of the lowest common ancestor for every two consecutive nodes. Hence, $\mathbf{c}$ has the same dimension as $\mathbf{d}$.
- $\mathbf{u}$ also encodes the label information, having a dimension equal to n . For every terminal node, $\mathbf{u}$ records the label of any merged non-terminal nodes, or $\langle \text{empty} \rangle$ if terminal node is not merged.

- For example, in Figure 2, $\mathbf{w} = (\text{return}, a, \cdot)$, then $\mathbf{d} = (2, 1)$, $\mathbf{c} = (\text{return_statement}, \emptyset)$, and $\mathbf{u} = (\langle \text{empty} \rangle, \langle \text{empty} \rangle, \langle \text{empty} \rangle)$.
- Similarly, leveraging the $\mathbf{t} = (\mathbf{d}, \mathbf{c}, \mathbf{u})$, the binary tree can be converted from the vector tuple.

2.2.2 Learn the Projection P_S

The two-step transformation described above enables a bidirectional transformation between ASTs and vector tuples, allowing us to both learn the projection P_S and apply the learned P_S to extract the AST from the hidden representation $\mathbf{h}$. In the learning process, we first parse the input code corresponding to $\mathbf{h}$ to obtain the O-AST, then apply the two-step transformation to generate the vector tuple $\mathbf{t}_0 = (\mathbf{d}_0, \mathbf{c}_0, \mathbf{u}_0)$. Next, treating $\mathbf{h}$ as the input and $\mathbf{t}_0$ as the target, we learn the orthogonal projection P_S . In the extraction process, we feed $\mathbf{h}$ into the learned projection P_S to obtain a predicted vector tuple $\mathbf{t}_p = (\mathbf{d}_p, \mathbf{c}_p, \mathbf{u}_p)$. Applying the two-step transformation to $\mathbf{t}_p$ yields the P-AST, thereby completing the extraction of syntactic information from the hidden representation.

Building on the transformation process described above, we adopt the settings outlined in López et al. (2022) to train P_S . The loss function is defined as follows:

$$\mathcal{L} = \mathcal{L}_d + \mathcal{L}_c + \mathcal{L}_u + \lambda \cdot \text{OR}(B), \quad (6)$$

$$\mathcal{L}_d = \sum_{i,j>i} \text{ReLU} \left(1 - \text{sign}(d_i - d_j)(\hat{d}_i - \hat{d}_j) \right), \quad (7)$$

$$\mathcal{L}_c = - \sum_i \log \frac{\exp(\langle P_S(h_i) - P_S(h_{i+1}), v_{c_i} \rangle)}{\sum_{i \in C} \exp(\langle P_S(h_i) - P_S(h_{i+1}), v \rangle)}, \quad (8)$$

$$\mathcal{L}_u = - \sum_i \log \frac{\exp(\langle P_S(h_i), v_{u_i} \rangle)}{\sum_{i \in U} \exp(\langle P_S(h_i), v \rangle)}. \quad (9)$$

Here, h_i denotes a hidden representation, and Bh_i corresponds to the coordinates of $P_S(h_i)$ in the syntactic subspace S . $\hat{d}_i$ is defined as

$$\hat{d}_i = \|Bh_i - Bh_{i+1}\|_2^2. \quad (10)$$

Moreover, $v_{c_i} \in C$ and $v_{u_i} \in U$ represent the corresponding label vectors for $\mathbf{c}$ and $\mathbf{u}$, respectively. The regularization term is defined as Limisiewicz and Mareček (2020):

$$\text{OR}(B) = \|BB^\top - I\|_F^2, \quad (11)$$

where $\|\cdot\|_F$ denotes the Frobenius norm and I is the identity matrix. Further details regarding the training settings can be found in López et al. (2022).

2.2.3 Mask-based weighting for vulnerability-aware probe training.

To encourage the probe to better preserve syntactic relations associated with vulnerable code regions, we introduce a mask-based weighting mechanism during probe

training. For each training sample, we extract the vulnerable lines and map the line-level labels to the token level. Specifically, a token is assigned a label of 1 if it resides in a vulnerable line, and 0 otherwise. Therefore, we obtain a binary token mask $\mathbf{m} \in \{0, 1\}^n$, where $m_i = 1$ indicates that token i is mapped to a vulnerable line (padding tokens are ignored). Notably, this mask is used only to reweight the distance ranking objective during probe training; at inference time, the probe extracts the P-AST from $\mathbf{h}$ without requiring any vulnerability annotations.

Following the distance ranking formulation in López et al. (2022), i and j index terminal-token positions used by the probe objective. We define a pair-wise indicator M_{ij} and assign weights w_{ij} as:

$$M_{ij} = \begin{cases} 1, & \text{if } m_i = 1 \text{ or } m_j = 1, \\ 0, & \text{otherwise,} \end{cases} \quad (12)$$

$$w_{ij} = 1 \cdot M_{ij} + (1 - \alpha)(1 - M_{ij}), \quad (13)$$

where $\alpha \in [0, 1]$ controls the suppression of non-vulnerable pairs; in our implementation the weight of vulnerable pairs is fixed to 1.0, while the weight of non-vulnerable pairs is $1 - \alpha$ (e.g., $\alpha = 0.99$). This asymmetric design ensures that all vulnerability-related pairs contribute fully (weight = 1.0) to the loss, while non-vulnerable pairs are down-weighted to $1 - \alpha$, focusing the probe’s capacity on preserving syntactic structures potentially relevant to vulnerabilities.

Accordingly, we use the following weighted distance ranking loss:

$$\mathcal{L}_{\mathbf{d}}^{\text{weighted}} = \frac{\sum_{i,j>i} w_{ij} \cdot \text{ReLU}\left(1 - \text{sign}(d_i - d_j)(\hat{d}_i - \hat{d}_j)\right)}{\sum_{i,j>i} w_{ij}}. \quad (14)$$

We keep $\mathcal{L}_{\mathbf{c}}$, $\mathcal{L}_{\mathbf{u}}$, and $\text{OR}(B)$ unchanged, and replace $\mathcal{L}_{\mathbf{d}}$ with $\mathcal{L}_{\mathbf{d}}^{\text{weighted}}$ when training P_S .

To this end, as illustrated in Figure 1, we probe the syntactic information in hidden representation by recovering the corresponding AST. Specifically, we feed the hidden representation $\mathbf{h}$ from the vulnerability detection model into the learned projection P_S . This projection will produce a vector tuple $\mathbf{t}$. Then, by applying the two-step transformation, we recover the AST embedded in $\mathbf{h}$. In the next subsection, we will investigate the extracted ASTs to identify the key syntactic structures that are associated with the prediction process of the vulnerability detection model. With these key syntactic structures, we can ultimately achieve vulnerability localization.

2.3 Vulnerability Localization

The P-AST is extracted from the hidden representation, and the syntactic information encoded within it is directly associated with the prediction process of the vulnerability detection model. During fine-tuning, vulnerability-related features are enhanced while irrelevant features are suppressed. Based on this understanding, we hypothesize that variations in the syntactic features embedded in the hidden representation

will be reflected in corresponding changes in the extracted P-AST, and these changes are indicative of the vulnerability model’s decision process. We treat these retained structures as explanatory evidence rather than causal factors.

In this section, we explore how to achieve vulnerability localization by analyzing the differences between P-AST and O-AST. We experiment with two AST representation methods for comparison and propose the scoring rule to quantify the model’s focus on different code structures. Finally, we select the most effective method and present pseudocode that outlines the process of AST comparison and scoring.

2.3.1 Compare

To analyze the syntactic information retained in hidden representations, we extract the P-AST from the model’s hidden states and compare it to the O-AST, parsed directly from the source code. This comparison reveals key syntactic structures that the model considers important for vulnerability detection.

To compare P-AST and O-AST, we first need to establish a suitable representation for AST substructures. L’opez et al. [López et al. \(2022\)](#) employ a tree constituent representation [Black et al. \(1991\)](#) to evaluate the degree of overlap between ASTs. The tree constituent representation is defined as

$$\langle \text{subtree root node} \rangle \langle i \rangle - \langle i + 1 \rangle - \dots - \langle j \rangle, \quad (15)$$

where $i, i+1, \dots, j$ denote the token IDs of the input code. This representation captures the root node and all leaf nodes of a subtree, but it ignores the internal structure. To obtain a more complete representation of each subtree, we perform a preorder traversal and obtain the representation:

$$\langle \text{subtree root node} \rangle \langle \text{preorder traversal} \rangle \dots \quad (16)$$

Unlike the tree constituent representation, which only records the root and leaves, preorder traversal representation preserves the hierarchical structure of the subtree more comprehensively. This structured representation better captures key syntactic patterns for comparisons.

2.3.2 Score

After comparing the ASTs, we identify syntactic structures retained in the hidden representation. We consider the lines of code where these structures are located to be highly correlated with vulnerabilities. To facilitate vulnerability localization, we score the tokens corresponding to matched subtrees to reflect their importance in the model’s prediction process for vulnerability detection. After completing the scoring process, we sum the token scores per line to compute the line score. The lines of code are sorted by their scores, and those with higher scores are considered more relevant to the vulnerability, thus completing the vulnerability localization.

The varying relevance of syntactic structures to vulnerabilities is considered in the scoring process. To reflect their significance, we assign higher scores to key syntactic structures. In particular, we focus on control relationships, which govern execution

flow, branching, and loops. Misconfigurations in these structures can lead to logic flaws, privilege escalation, or infinite loops, increasing security risks. Based on these considerations, we select a set of control-related AST node types. Subtrees rooted at these nodes are identified as control-related and assigned higher scores during the scoring process.

2.3.3 Pseudocode

Finally, we summarize the comparison and scoring process using Algorithm 1.

Algorithm 1 Score the Tokens.

Require: P-AST, O-AST

Ensure: TOKEN_SCORE

```

1: subTreesp ← getSubTree(P-AST)
2: subTreeso ← getSubTree(O-AST)
3: TOKEN_SCORE ← {0, ..., 0}
4: for each subtree st in subTreesp do
5:   if st ∈ subTreeso then
6:     if RootNode(st) is control-related then
7:       for each token i in st do
8:         TOKEN_SCORE[i] ← TOKEN_SCORE[i] + controlScore
9:       end for
10:    else
11:      for each token i in st do
12:        TOKEN_SCORE[i] ← TOKEN_SCORE[i] + uniformScore
13:      end for
14:    end if
15:  end if
16: end for
17: return TOKEN_SCORE

```

First, we extract subtrees from the input AST (Lines 1-2). Using a preorder traversal, we generate the representation for each subtree. Next, we compare the subtrees extracted from the P-AST and the O-AST one by one (Lines 4-5). When a matching subtree is found, we assign scores to all tokens within that subtree. If the subtree's root node is control-related, its tokens receive a *controlScore*; otherwise, they receive a *uniformScore* (Lines 6-15). As a result, we obtain TOKEN_SCORE, which represents the importance of each token in the vulnerability classification process.

Next, we aggregate the token-level scores to compute a score for each line of code. Specifically, for each line, we sum the scores of all tokens in that line to compute the line scores. Finally, we rank the code lines in descending order based on their scores to localize the vulnerability.

3 Experiment Setup

In this section, we introduce the setup of our experiments. We first introduce the studied baselines. Then we describe the evaluation metrics we adopt. Next, we introduce the dataset used in our experiments. Finally, we introduce the effort we do to reduce the impact of random factors during evaluation.

All experiments are performed on a server with an Intel Xeon Gold 6148 CPU @ 2.40GHz, 128GB physical memory, and two NVIDIA RTX 4090 GPUs with 24GB memory.

3.1 Baselines

LineVD [Hin et al. \(2022\)](#). LineVD formulates vulnerability localization as a binary node classification problem, and integrates both function-level and line-level information to enhance detection performance. In our experiments, we use the publicly available implementation provided by the authors of LineVD.

LineVul [Fu and Tantithamthavorn \(2022\)](#). LineVul employs the CodeBERT pre-trained model for vulnerability detection and leverages the attention scores produced by CodeBERT for vulnerability localization. We also use the source code released by the authors of LineVul for our experiments.

Saliency [Simonyan et al. \(2013\)](#). Saliency is a gradient-based feature attribution method that utilizes a first-order Taylor expansion of the network’s prediction function at the input. To implement this baseline, we adopt the vulnerability detection model of our VulProbe and then use the implementation provided in [Fu and Tantithamthavorn \(2022\)](#) to generate saliency scores for vulnerability localization.

DeepLIFT [Shrikumar et al. \(2017\)](#). DeepLIFT is a feature attribution method that computes contribution scores by comparing each neuron’s activation to a reference activation. We implement DeepLIFT in the same manner as Saliency.

Gradient [Lundberg and Lee \(2017\)](#). The Gradient baseline approximates feature importance scores by computing the expected gradients obtained through random sampling from a distribution of baseline inputs. We implement Gradient in the same manner as Saliency.

FlawFinder [Ferschke et al. \(2012\)](#). FlawFinder is an open-source static code scanning tool for C/C++ that detects potential security vulnerabilities by matching code patterns against an internal vulnerability signature database.

RATS [Secure Software, Inc. \(2026\)](#). RATS (Rough Auditing Tool for Security) employs a set of heuristic rules and regular expressions to scan source code, identifying potential issues such as buffer overflows, format string vulnerabilities, and SQL injection flaws.

GPT-4o [Hurst et al. \(2024\)](#). Considering the growing impact of large language models on many tasks, we also employ GPT-4o as a baseline for vulnerability localization.

3.2 Evaluation Metrics

Top-k Accuracy (Top-1/3/5 ACC). This metric is computed only over functions that are predicted to be vulnerable. It measures the proportion of actual vulnerable

lines that are ranked within the top- k of the sorted list. Higher values indicate that the model is able to rank true vulnerable lines near the top, thereby enabling developers to review code more efficiently.

Initial False Alarm (IFA). For each vulnerable function, the lines are first sorted in descending order according to the predicted line scores. IFA is defined as the number of non-vulnerable lines that are reviewed before encountering the first actual vulnerable line. A lower IFA indicates that the model is more effective in prioritizing vulnerable lines over clean ones.

Recall@1%Effort (R@1%E). For each vulnerable function, after sorting all lines in descending order by their predicted scores, this metric calculates the fraction of the total actual vulnerable lines that are identified within the top 1% of the code lines. Higher recall at the same effort implies that the model can efficiently capture a substantial portion of vulnerabilities with fewer code inspection.

Effort@20%Recall (E@20%R). This metric quantifies the proportion of the total lines of code that need to be reviewed in order to identify 20% of the actual vulnerable lines. Lower values indicate that fewer lines must be inspected to capture a significant portion of vulnerabilities, reflecting greater efficiency in vulnerability localization.

3.3 Dataset

In this work, we uniformly employ the BigVul dataset [Fan et al. \(2020\)](#) for our experiments. The dataset is partitioned into training, validation, and test sets in an 8:1:1 ratio. Following the experimental setup in [López et al. \(2022\)](#), we filter out entries with a tokenized length greater than 512. As a result, the final dataset comprises 37,736 training samples, 4,718 validation samples, and 4,718 test samples.

3.4 Statistical Analysis

To mitigate the influence of random variations on our evaluation, we conduct five independent runs for each baseline and perform statistical analyses on the aggregated results. This allows us to more accurately assess both the performance and the stability of each model. Specifically, we analyze the following two aspects [Chen et al. \(2024\)](#); [Wattanakriengkrai et al. \(2020\)](#).

Performance Gain. To determine whether our VulProbe outperforms the baselines, we compute the performance gain as the percentage difference between the performance of VulProbe and that of the baselines. This is calculated as follows:

$$\%Diff = \frac{\sum(\text{Perf}_{VulProbe} - \text{Perf}_{baseline})}{\sum \text{Perf}_{baseline}}. \quad (17)$$

A higher %Diff indicates a greater performance improvement of VulProbe relative to the baselines.

Statistical Test. We employ a one-sided Wilcoxon signed-rank test to assess the statistical significance of the performance differences between VulProbe and the baseline methods. The Wilcoxon signed-rank test is a non-parametric method well-suited for pairwise comparisons of related samples, which makes it appropriate for

our experimental setting. In addition, we compute the effect size r to quantify the magnitude of the observed differences:

$$r = \frac{Z}{\sqrt{n}}, \quad (18)$$

where Z is the Z-score obtained from the Wilcoxon signed-rank test and n is the total number of samples [Tomczak and Tomczak \(2014\)](#). An effect size of $r > 0.5$ is considered large, $0.3 < r \leq 0.5$ medium, and $0.1 < r \leq 0.3$ small; values below 0.1 are regarded as negligible [Field \(2024\)](#).

4 Experiment Result

In this section, we present the motivation, design and result for the following three research questions:

- RQ1: How effective is VulProbe in vulnerability localization?
- RQ2: How do different vulnerability detection models affect the performance of VulProbe?
- RQ3: How does the mask-based weighting mechanism affect the performance of VulProbe?

4.1 RQ1: How effective is VulProbe in vulnerability localization?

4.1.1 Motivation

Most existing vulnerability detection methods operate at a coarse-grained level, even though vulnerable lines constitute only a small fraction of the total code. The fine-grained vulnerability detection can significantly help developers quickly pinpoint the exact locations of vulnerabilities with minimal effort. In this work, we propose a novel vulnerability localization method, VulProbe. We are the first to incorporate probing techniques into the vulnerability detection task, leveraging the syntactic information encoded in hidden representations to identify vulnerable code lines. In this RQ we aim to evaluate the effectiveness of our proposed VulProbe by comparing it against other state-of-the-art vulnerability localization methods.

4.1.2 Design

To address this RQ, we select eight baselines: LineVD, LineVul, Saliency, DeepLIFT, Gradient, FlawFinder, RATS and GPT-4o. All methods are evaluated on the same dataset and compared using six evaluation metrics.

For all six metrics, larger values of Top-1, Top-3, Top-5 Accuracy and Recall@1%Effort indicate better performance, while smaller values of IFA and Effort@20%Recall signify improved efficiency.

4.1.3 Results

Figure 3 illustrates that our VulProbe outperforms all baselines across all evaluation metrics. The red line in the figure represents the median. At the median value, VulProbe achieves Top-1 Accuracy of 0.2586 (108.21% – 660.03% higher than the baselines), Top-3 Accuracy of 0.4224 (52.47% – 888.07% higher), and Top-5 Accuracy of 0.4914 (25.76% – 723.47% higher). Table 1 further demonstrates that VulProbe surpasses all baselines in Top-1, Top-3, and Top-5 Accuracy. These high Top- k Accuracy scores indicate that VulProbe effectively ranks and prioritizes vulnerable lines, thereby improving the efficiency of vulnerability detection for developers.

Furthermore, at the median, VulProbe achieves IFA of 6.03 (a reduction of 17.15% – 57.59% compared to the baselines), Recall@1%Effort of 0.422414 (62.58% – 9950.26% higher), and Effort@20%Recall of 0.003951 (a reduction of 51.28% – 96.78%). These results demonstrate that, given the same level of effort, VulProbe can detect vulnerable lines more efficiently, significantly improving the effectiveness of vulnerability localization.

Additionally, statistical analysis results in Table 1 demonstrate the robustness of our findings. The one-sided Wilcoxon signed-rank tests indicate statistical significance ($p < 0.05$) across all evaluation metrics, with a large effect size.

Among the selected baselines, FlawFinder and RATS are static analysis tools and perform the worst, particularly in Recall@1%Effort and Effort@20%Recall. As shown in Table 1, RATS fails to recall any vulnerable lines (its Recall@1%Effort is denoted as “-”), and neither FlawFinder nor RATS identifies at least 20% of the total vulnerable lines (their Effort@20%Recall is marked as “-”). Moreover, in Figure 3, the performance of FlawFinder, RATS, and GPT-4o is significantly worse than that of the other methods. This discrepancy severely affects the overall visual presentation of the figure. To ensure clarity and readability, we have excluded these methods from the visualization.

4.2 RQ2: How do different vulnerability detection models affect the performance of VulProbe?

4.2.1 Motivation

In our VulProbe method, we adopt a BERT-based architecture for vulnerability detection and then probe the output hidden representations to derive the AST for vulnerability localization. Our probing method is model-agnostic, meaning it can be applied to any vulnerability detection model that produces hidden representations, thereby broadening the applicability of our approach.

In this RQ, we investigate how different vulnerability detection models influence VulProbe’s performance in vulnerability localization by replacing the vulnerability detection model. Subsequently, we analyze the factors that affect the quality of vulnerability localization.

Table 1 A comparative summary of the performance between VulProbe and the baselines (updated with new VulProbe results).

Baseline	Top-1 ACC↑		Top-3 ACC↑		Top-5 ACC↑		IFA↓		R@1%E↑		E@20%R↓	
	Diff%	E(p)	Diff%	E(p)	Diff%	E(p)	Diff%	E(p)	Diff%	E(p)	Diff%	E(p)
LineVD	120.35	L(0.03)	65.78	L(0.03)	35.82	L(0.03)	-17.15	L(0.03)	94.06	L(0.03)	-57.05	L(0.03)
LineVul	186.91	L(0.03)	84.31	L(0.03)	32.10	L(0.03)	-26.19	L(0.03)	62.58	L(0.03)	-51.28	L(0.03)
Saliency	171.53	L(0.03)	55.34	L(0.03)	30.92	L(0.03)	-24.99	L(0.03)	90.31	L(0.03)	-52.28	L(0.03)
DeepLIFT	108.21	L(0.03)	52.47	L(0.03)	25.76	L(0.03)	-23.69	L(0.03)	98.58	L(0.03)	-55.04	L(0.03)
Gradient	145.21	L(0.03)	58.34	L(0.03)	35.22	L(0.03)	-25.29	L(0.03)	135.80	L(0.03)	-63.07	L(0.03)
FlawFinder	508.20	L(0.03)	606.24	L(0.03)	475.97	L(0.03)	-56.42	L(0.03)	9952.60	L(0.03)	—	—
RATS	660.03	L(0.03)	888.07	L(0.03)	723.47	L(0.03)	-57.59	L(0.03)	—	—	—	—
GPT-4o	157.83	L(0.03)	105.83	L(0.03)	74.58	L(0.03)	-34.95	L(0.03)	2856.00	L(0.03)	-96.78	L(0.03)

Note: ‘Diff%’ indicates the relative improvement of VulProbe over each baseline. ‘E’ represents the effect size, and ‘p’ represents the p-value from the Wilcoxon signed-rank test. Effect size: Large(L) $r > 0.5$, Medium(M) $0.3 < r \leq 0.5$, Small(S) $0.1 < r \leq 0.3$, Negligible(N) $r < 0.1$.

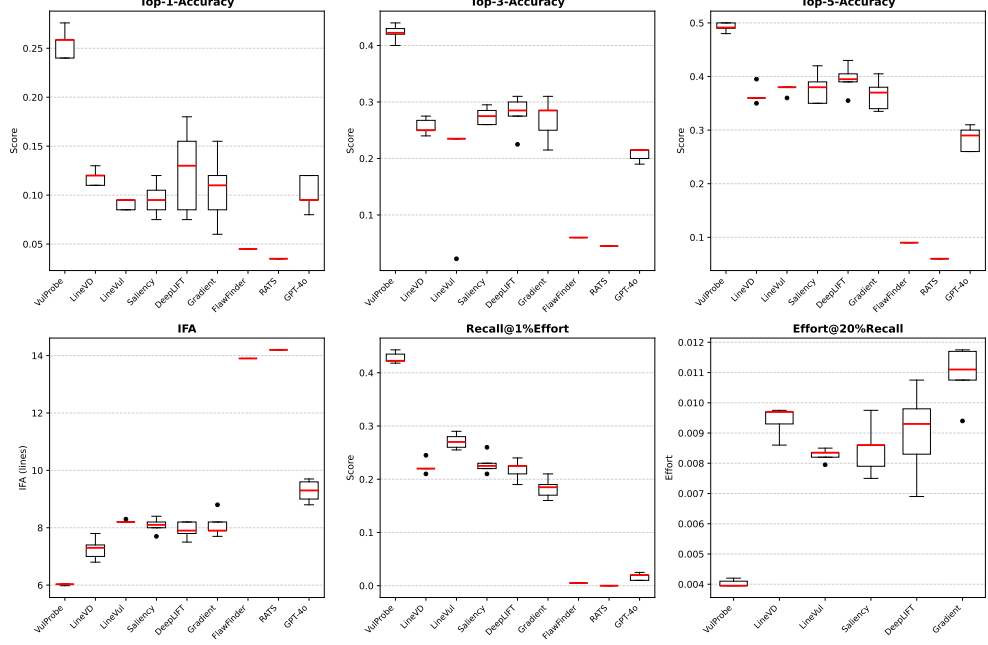


Fig. 3 The vulnerability localization performance of VulProbe and baselines.

4.2.2 Design

In this RQ, we replace CodeBERT with alternative models, including UniXcoder [Guo et al. \(2022\)](#), BiLSTM [Zhang et al. \(2015\)](#), and MLP [Kruse et al. \(2022\)](#). Then, on the same dataset, we replace different models for vulnerability detection and accordingly complete vulnerability localization by extracting hidden representations. From there, we evaluate the influence of the vulnerability detection model’s capability on VulProbe in the vulnerability localization task.

4.2.3 Results

Table 2 illustrates the performance of VulProbe using CodeBERT, UniXcoder, BiLSTM, and MLP as the vulnerability detection backbones. These results demonstrate that Transformer-based models (CodeBERT and UniXcoder) significantly outperform traditional architectures (BiLSTM and MLP) in vulnerability localization. During training, CodeBERT and UniXcoder are more effective at capturing vulnerability-related features. Consequently, VulProbe leverages these features to reconstruct critical syntactic structures in the P-AST, thereby enhancing vulnerability localization performance.

In summary, the choice of vulnerability detection backbone significantly influences VulProbe’s localization performance. As shown in Table 2, Transformer-based models (CodeBERT and UniXcoder) substantially outperform traditional architectures (BiLSTM and MLP) across all six metrics. Notably, although CodeBERT and UniXcoder

Table 2 The vulnerability localization performance of VulProbe with different vulnerability detection models.

Vulnerability Detection Model	Top-1 ACC $\uparrow$	Top-3 ACC $\uparrow$	Top-5 ACC $\uparrow$	IFA $\downarrow$	R@1%E $\uparrow$	E@20%R $\downarrow$
CodeBERT	0.2586	0.4224	0.4914	6.03	0.422414	0.003951
UniXcoder	0.2500	0.4224	0.4828	6.07	0.400862	0.005345
BiLSTM	0.1293	0.3190	0.3879	6.90	0.362069	0.004638
MLP	0.1379	0.2759	0.3534	7.28	0.340517	0.004471

Note: The reported values are the average results over five independent runs of each model.

achieve similar Top-3 Accuracy, CodeBERT yields better results in Top-1 Accuracy, Top-5 Accuracy, Recall@1%Effort, IFA, and Effort@20%Recall. This indicates that CodeBERT captures more discriminative features for vulnerability localization under the same probing framework.

4.3 RQ3: How does the mask-based weighting mechanism affect the performance of VulProbe?

4.3.1 Motivation

The mask-based weighting mechanism is designed to encourage the probe to focus more on token pairs associated with vulnerable code lines during training, thereby improving the recovery of vulnerability-related syntactic structures and ultimately enhancing localization accuracy. However, it remains unclear whether this mechanism indeed brings performance gains and how different mask weights (i.e., α) influence the trade-off between focusing on vulnerable regions and maintaining generalization. In this RQ, we aim to answer two questions: (1) Does the mask-based weighting mechanism improve vulnerability localization compared to the vanilla probe? (2) What is the optimal mask weight α that balances the emphasis on vulnerable token pairs and overall robustness?

4.3.2 Design

We compare the vanilla probe (trained with the standard distance ranking loss $\mathcal{L}_d$) against mask-weighted probes trained with different α values. Based on preliminary experiments, we select three representative mask weights: $\alpha = 0.4$ (low emphasis), $\alpha = 0.7$ (medium emphasis), and $\alpha = 0.99$ (high emphasis). For each configuration, we evaluate the vulnerability localization performance using six metrics: Top-1/3/5 Accuracy (higher is better), IFA (lower is better), Recall@1%Effort (higher is better), and Effort@20%Recall (lower is better). All experiments are conducted on the same dataset (BigVul) and follow the same evaluation protocol as RQ1 and RQ2.

4.3.3 Results

Table 3 summarizes the localization performance of the vanilla probe alongside mask-weighted variants configured with different weighting factors α . The results

Table 3 The vulnerability localization performance of VulProbe with different mask weights α in the mask-based weighting mechanism.

Configuration	Top-1 ACC $\uparrow$	Top-3 ACC $\uparrow$	Top-5 ACC $\uparrow$	IFA $\downarrow$	R@1%E $\uparrow$	E@20%R $\downarrow$
Vanilla (without mask)	0.2586	0.4052	0.4828	6.10	0.409483	0.004201
Mask ($\alpha = 0.99$)	0.2586	0.4224	0.4914	6.03	0.422414	0.003951
Mask ($\alpha = 0.7$)	0.2500	0.4052	0.4914	6.10	0.400862	0.004097
Mask ($\alpha = 0.4$)	0.2586	0.4052	0.4828	6.10	0.405172	0.004222

Note: The reported values are the average results over five independent runs of each configuration.

demonstrate that the mask-based weighting mechanism yields consistent improvements when α is set to a sufficiently high value. Specifically, with ($\alpha = 0.99$), the probe achieves a Top-3 accuracy of 0.4224 and a Recall@1%Effort of 0.4224, surpassing the vanilla baseline (0.4052 and 0.4095, respectively). Concurrently, it reduces both IFA (6.03 vs. 6.10) and Effort@20%Recall (0.003951 vs. 0.004201). These gains indicate that imposing a strong emphasis on vulnerable token pairs enables the probe to capture more discriminative syntactic structures, which directly translates to higher precision in line-level vulnerability localization. Additionally, the choice of α proves to be a critical factor influencing model performance. When α is set to relatively low (0.4) or moderate (0.7) values, the mask-weighted probe does not consistently outperform the vanilla baseline; in certain cases, performance even exhibits slight degradation, as evidenced by the Top-1 Accuracy dropping to 0.2500 for ($\alpha = 0.7$).

In summary, the mask-based weighting mechanism with an appropriately high weight ($\alpha = 0.99$) effectively improves vulnerability localization by guiding the probe to prioritize syntactic relationships in vulnerable code regions. Lower mask weights provide marginal or no benefit, highlighting the importance of a strong emphasis on vulnerability-related token pairs during probe training.

5 Threats To Validity

Construct Validity. To ensure a fair comparison with baselines, we evaluate the performance of our method using six different metrics. We quantify the effort required for code review by the number of lines of code (LOC). However, code complexity may also influence the effort needed. Future work could incorporate additional factors, such as code complexity, to improve the accuracy of effort estimation and mitigate this threat.

Internal Validity. Errors in the implementation of baselines may introduce unintended biases in our experiments. To minimize this threat, we adopt open-source implementations provided by the original studies whenever available. Additionally, we use the same dataset splits and evaluation protocols across all methods to ensure consistency. The randomness in model training and data partitioning may also affect the results. To mitigate this, we conduct five independent runs for each baseline and report median values, as well as perform statistical significance tests to confirm the robustness of our findings.

External Validity. Our experiments are conducted on Big-Vul [Fan et al. \(2020\)](#), a C/C++ vulnerability dataset. While our method demonstrates strong performance on this dataset, its generalizability to real-world software systems and other programming languages remains uncertain. To further validate its effectiveness, future work will explore extending the dataset to include vulnerabilities from a broader range of software projects and programming languages.

Generalizability to black-box models. VulProbe requires access to hidden representations, which may not be available for proprietary or API-based models (e.g., GPT-4o). This is a common limitation of white-box interpretation methods in the explainable AI literature. In our evaluation, we focus on open-source models (CodeBERT, UniXcoder) and traditional neural networks (BiLSTM, MLP) where such access is feasible—a setting representative of many practical software engineering scenarios where models are trained in-house or publicly released. We explicitly acknowledge this as a scope limitation rather than a claim of universal model-agnosticism.

6 Related Work

In this section, we review the related work about vulnerability localization, and model interpretation.

6.1 Vulnerability Localization

Vulnerability localization aims to efficiently pinpoint vulnerable lines in source code, thereby reducing the manual effort required for code review. Li et al. [Li et al. \(2021\)](#) proposed the Feature-Attention Graph Convolution Network (FA-GCN) for vulnerability detection and utilized GNNExplainer [Ying et al. \(2019\)](#) to identify the minimal key subgraphs, localizing vulnerabilities in the process. Hin et al. [Hin et al. \(2022\)](#) formalized vulnerability localization as a binary node classification problem, predicting whether each line in a function is vulnerable based on contextual information. Attention-based approaches, such as [Fu and Tantithamthavorn \(2022\)](#) and [Pornprasit and Tantithamthavorn \(2022\)](#), leverage attention scores to assess token importance for vulnerability classification. These methods aggregate token-level scores to compute line-level risk estimates, highlighting potentially vulnerable lines. Wattanakriengkrai et al. [Wattanakriengkrai et al. \(2020\)](#) adopted a feature attribution technique to calculate line scores, enabling vulnerability localization. Unlike prior works that rely on attention scores or feature attribution, we analyze the hidden representations, which inherently encode the model’s learned vulnerability patterns. By probing the syntactic structures embedded in these representations, we identify key features relevant to vulnerabilities, thereby enabling more precise localization. We emphasize that this approach is architecture-agnostic but assumes white-box access to the detection model, consistent with the standard setting of structural probing in the literature.

6.2 Model Interpretation

As ML-based approaches achieve remarkable success in software engineering tasks such as code completion, comment generation, and vulnerability detection, researchers have increasingly sought to interpret these models to enhance their credibility and usability. LIME [Ribeiro et al. \(2016\)](#) interprets individual model predictions by locally approximating the model’s behavior around a given prediction. DeepLIFT [Shrikumar et al. \(2017\)](#) compares each neuron’s activation to a reference activation and assigns contribution scores based on their differences. SHAP [Lundberg and Lee \(2017\)](#) identifies commonalities among feature attribution methods and unifies them into a single framework. The strong performance of Transformer-based models has further drawn attention to their interpretability. Works such as [Bai et al. \(2021\)](#); [Jain and Wallace \(2019\)](#); [Wiegrefe and Pinter \(2019\)](#) analyze the interpretability of attention mechanisms from multiple perspectives. [Fu et al. \(2022\)](#) leverage the attention mechanism within the BERT architecture to explain model predictions. In this paper, we adopt the probing technique, an approach for assessing whether the hidden representations of neural networks encode specific properties [Belinkov \(2022\)](#); [Hewitt and Manning \(2019\)](#). [Karmakar et al. \(2021\)](#) investigated code pre-trained language models and found that their hidden representations capture various properties of code, such as cyclomatic complexity and input code length. Similarly, [L’opez et al. \(2022\)](#) introduced the AST-Probe, which recovers Abstract Syntax Trees from the hidden representations of code. By leveraging the probe technique, we extract encoded syntactic structures as faithful explanations of the model’s predictions, providing deeper insights into which code structures the model attends to and enhancing our ability to localize vulnerabilities effectively. Unlike attention-based explanations that may be unfaithful to model decisions, our probing-based approach recovers structures that are directly associated with the model’s hidden representations.

7 Conclusion And Future Work

Vulnerability localization is a critical yet challenging task in software security. Existing approaches often rely on attention mechanisms or feature attribution scores to estimate token importance, which may not faithfully reflect the factors driving model decisions. We propose **VulProbe**, which enhances vulnerability localization by probing hidden representations to extract key syntactic structures. Specifically, VulProbe first fine-tunes CodeBERT for vulnerability detection, then employs a probe with a **mask-based weighting mechanism** that emphasizes token pairs associated with vulnerable lines during training. VulProbe reconstructs ASTs from these representations and compares them with original ASTs derived from source code. By analyzing their differences, it identifies important syntactic structures and improves localization accuracy. We evaluate VulProbe on the Big-Vul dataset, comparing its performance against eight baselines using six evaluation metrics. To ensure reliability, we perform five independent runs per model and aggregate the results. The experimental findings show that VulProbe consistently outperforms all baselines with statistical significance.

Furthermore, we conduct additional experiments to analyze factors influencing VulProbe’s performance. Our results reveal that the mask-based weighting mechanism significantly improves vulnerability localization: with a sufficiently high mask weight (e.g., $\alpha = 0.99$), VulProbe achieves better Top-3 Accuracy, Recall@1%Effort, IFA, and Effort@20%Recall compared to the vanilla probe without masking.

In future work, we aim to probe additional types of information from hidden representations. While this study focuses on extracting ASTs, future research could explore semantic information to understand which semantic features the model attends to during decision-making. By integrating multiple aspects of extracted information, we can gain a more comprehensive understanding of the evidence the model relies on, leading to further improvements in vulnerability localization and faithful model interpretability.

Declarations

Funding. Not applicable.

Conflict of interest. The authors declare that they have no conflict of interest.

Ethics approval and consent to participate. Not applicable.

Consent for publication. Not applicable.

Data availability. The BigVul dataset [Fan et al. \(2020\)](#) used in this study is publicly available.

Code availability. The replication package of VulProbe is available at <https://github.com/alfulort/VulProbe>.

References

- Bai B, Liang J, Zhang G, et al (2021) Why attentions may not be interpretable? In: Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining. ACM, New York, NY, USA, pp 25–34
- Belinkov Y (2022) Probing classifiers: Promises, shortcomings, and advances. Computational Linguistics 48(1):207–219
- Belinkov Y, Durrani N, Dalvi F, et al (2017) What do neural machine translation models learn about morphology? arXiv preprint arXiv:170403471
- Black E, Abney S, Flickinger D, et al (1991) A procedure for quantitatively comparing the syntactic coverage of english grammars. In: Speech and Natural Language: Proceedings of a Workshop Held at Pacific Grove, California, February 19-22, 1991
- Chen X, Xu F, Huang Y, et al (2024) Jit-smart: A multi-task learning framework for just-in-time defect prediction and localization. Proceedings of the ACM on Software Engineering 1(FSE):1–23
- Cheng B, Zhao S, Wang K, et al (2024) Beyond fidelity: Explaining vulnerability localization of learning-based detectors. ACM Transactions on Software Engineering

and Methodology 33(5):1–33

Cppcheck Team (2026) Cppcheck - A Tool for Static C/C++ Code Analysis. URL <https://cppcheck.sourceforge.io/>, accessed: 2026-04-04

Devlin J (2018) Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805

Fan J, Li Y, Wang S, et al (2020) A c/c++ code vulnerability dataset with code changes and cve summaries. In: Proceedings of the 17th International Conference on Mining Software Repositories. ACM, pp 508–512

Feng Z, Guo D, Tang D, et al (2020) Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155

Ferschke O, Gurevych I, Rittberger M (2012) Flawfinder: A modular system for predicting quality flaws in wikipedia. In: CLEF (Online Working Notes/Labs/Workshop), pp 1–10

Field A (2024) Discovering Statistics Using IBM SPSS Statistics. Sage Publications Limited

Fu M, Tantithamthavorn C (2022) Linevul: A transformer based line-level vulnerability prediction. In: Proceedings of the 19th International Conference on Mining Software Repositories. ACM, pp 608–620

Guo D, Lu S, Duan N, et al (2022) Unixcoder: Unified cross-modal pre-training for code representation. arXiv preprint arXiv:2203.03850

Hewitt J, Manning CD (2019) A structural probe for finding syntax in word representations. In: Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), pp 4129–4138

Hin D, Kan A, Chen H, et al (2022) LineVD: Statement-level vulnerability detection using graph neural networks. In: Proceedings of the 19th International Conference on Mining Software Repositories. ACM, pp 596–607

Hurst A, Lerer A, Goucher AP, et al (2024) Gpt-4o system card. arXiv preprint arXiv:2410.21276

Husain H, Wu HH, Gazit T, et al (2019) Codesearchnet challenge: Evaluating the state of semantic code search. arXiv preprint arXiv:1909.09436

Jain S, Wallace BC (2019) Attention is not explanation. In: Proceedings of NAACL-HLT, pp 3543–3556

- Jang-Jaccard J, Nepal S (2014) A survey of emerging threats in cybersecurity. *Journal of Computer and System Sciences* 80(5):973–993
- Karmakar A, Robbes R (2021) What do pre-trained code models know about code? In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp 1332–1336
- Kruse R, Mostaghim S, Borgelt C, et al (2022) Multi-layer perceptrons. In: *Computational Intelligence: A Methodological Introduction*. Springer, p 53–124
- Li Y, Wang S, Nguyen TN (2021) Vulnerability detection with fine-grained interpretations. In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, pp 292–303
- Li Z, Zou D, Xu S, et al (2018) Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:180101681*
- Limisiewicz T, Mareček D (2020) Introducing orthogonal constraint in structural probes. *arXiv preprint arXiv:201215228*
- Lundberg SM, Lee SI (2017) A unified approach to interpreting model predictions. *Advances in Neural Information Processing Systems* 30:4765–4774
- López JAH, Weyssow M, Cuadrado JS, et al (2022) AST-Probe: Recovering abstract syntax trees from hidden representations of pre-trained language models. In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. ACM, pp 1–11
- National Vulnerability Database (2026) Common Vulnerabilities and Exposures (CVE). URL <https://www.cve.org/>, accessed: 2026-04-04
- Pornprasit C, Tantithamthavorn C (2022) Deeplinedp: Towards a deep learning approach for line-level defect prediction. *IEEE Transactions on Software Engineering* 49(1):84–98
- Ribeiro MT, Singh S, Guestrin C (2016) Why should i trust you? explaining the predictions of any classifier. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, pp 1135–1144
- Secure Software, Inc. (2026) RATS - Rough Auditing Tool for Security. URL <https://code.google.com/p/rough-auditing-tool-for-security/>, accessed: 2026-04-04
- Shen Y, Lin Z, Jacob AP, et al (2018) Straight to the tree: Constituency parsing with neural syntactic distance. *arXiv preprint arXiv:180604168*

- Shrikumar A, Greenside P, Kundaje A (2017) Learning important features through propagating activation differences. In: International Conference on Machine Learning. PMLR, pp 3145–3153
- Simonyan K, Vedaldi A, Zisserman A (2013) Deep inside convolutional networks: Visualising image classification models and saliency maps. arXiv preprint arXiv:1312.6034
- Tomczak M, Tomczak E (2014) The need to report effect size estimates revisited. an overview of some recommended measures of effect size. *Trends in Sport Sciences* 21(1):19–25
- Troshin S, Chirkova N (2022) Probing pretrained models of source code. arXiv preprint arXiv:2202.08975
- Vaswani A, Shazeer N, Parmar N, et al (2017) Attention is all you need. *Advances in Neural Information Processing Systems* 30
- Wan Y, Zhao W, Zhang H, et al (2022) What do they capture? a structural analysis of pre-trained language models for source code. In: Proceedings of the 44th International Conference on Software Engineering. ACM, pp 2377–2388
- Wattanakriengkrai S, Thongtanunam P, Tantithamthavorn C, et al (2020) Predicting defective lines using a model-agnostic technique. *IEEE Transactions on Software Engineering* 48(5):1480–1496
- Wiegrefe S, Pinter Y (2019) Attention is not not explanation. In: 2019 Conference on Empirical Methods in Natural Language Processing and 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP 2019). Association for Computational Linguistics, pp 11–20
- Wu Y, Zou D, Dou S, et al (2022) Vulcnn: An image-inspired scalable vulnerability detection system. In: Proceedings of the 44th International Conference on Software Engineering. ACM, pp 2365–2376
- Ying Z, Bourgeois D, You J, et al (2019) Gnnexplainer: Generating explanations for graph neural networks. In: *Advances in Neural Information Processing Systems*
- Zhang J, Liu Z, Hu X, et al (2023) Vulnerability detection by learning from syntax-based execution paths of code. *IEEE Transactions on Software Engineering* 49(8):4196–4212
- Zhang S, Zheng D, Hu X, et al (2015) Bidirectional long short-term memory networks for relation classification. In: Proceedings of the 29th Pacific Asia Conference on Language, Information and Computation, pp 73–78
- Zhou Y, Liu S, Siow J, et al (2019) Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In: *Advances*

in Neural Information Processing Systems