LOSVER: Line-Level Modifiability Signal-Guided Vulnerability Detection and Classification

Doha Nam
SPIRAL Lab, School of Computing, KAIST
Daejeon, South Korea
waroad@kaist.ac.kr

Jongmoon Baik*
SPIRAL Lab, School of Computing, KAIST
Daejeon, South Korea
jbaik@kaist.ac.kr

Abstract

The increasing prevalence of software vulnerabilities continues to pose serious threats to system security, underscoring the need for accurate and scalable techniques for vulnerability detection and classification. While Pre-trained Language Models (PLMs) have shown strong potential in vulnerability analysis, most existing methods provide no explicit guidance on which parts of the input code are more likely to be vulnerable. As a result, the model must infer token-level relevance without any indication of which parts are important, making it harder to learn the characteristics of vulnerable code during training. To address this limitation, we propose LOSVER (Line-level mOdifiability Signal-guided VulnERability analyzer), a novel two-stage framework that enhances PLM-based vulnerability analysis by incorporating line-level modifiability signals. In the first stage, LOSVER localizes modifiable lines. These are code segments likely to be changed in the future due to instability or complexity, which are often associated with vulnerabilities. In the second stage, the model assigns greater importance to the predicted modifiable lines, allowing the PLM to focus on potentially vulnerable regions during both training and inference. We evaluated LOSVER with two widely used benchmark datasets: Devign, for function-level vulnerability detection, and Big-Vul, for function-level vulnerability classification with Common Weakness Enumeration (CWE) ID labels. Experimental results show that LOSVER improves detection accuracy on Devign by approximately 4 percentage points and increases the weighted F1-score for CWE ID classification on Big-Vul by over 2 points, when applied on top of the UniXcoder baseline. We also conducted experiments on the PrimeVul dataset, which focuses on vulnerability–patch pairs, and observed meaningful improvements in pair-wise detection. These results demonstrate that integrating line-level modifiability signals significantly enhances the effectiveness of PLM-based software vulnerability analysis across both detection and classification tasks.

CCS Concepts

• **Security and privacy → Software security engineering**; • **Software and its engineering → Software defect analysis**.

* Corresponding author

ASE '25, Seoul, South Korea

ACM ISBN 978-1-4503-XXXX-X/25/11
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Doha Nam and Jongmoon Baik. 2025. LOSVER: Line-Level Modifiability Signal-Guided Vulnerability Detection and Classification. In *40th IEEE/ACM International Conference on Automated Software Engineering (ASE '25), November 16–20, 2025, Seoul, South Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Software vulnerabilities represent critical weaknesses that can be exploited to compromise the integrity, confidentiality, or availability of software systems [25]. High-profile incidents such as the Heartbleed vulnerability [1] and the Equifax data breach [47] exemplify the potentially catastrophic consequences of undetected and/or unmitigated vulnerabilities in deployed software systems. Accordingly, timely and accurate vulnerability detection is essential for mitigating security risks and maintaining secure software systems. Beyond detection, vulnerability classification, such as mapping vulnerable codes to Common Weakness Enumeration (CWE) IDs, enables security teams to assess severity more systematically and prioritize remediation efforts with greater efficiency.

Although manual code auditing remains a common practice, it has become increasingly infeasible to scale due to the growing size and complexity of modern software systems. This challenge has motivated a growing body of research focused on automated vulnerability detection and classification techniques [42, 44]. Reflecting this trend, the CodeXGLUE benchmark—curated by Microsoft to evaluate machine learning models on code understanding tasks—includes a dedicated vulnerability detection dataset [30]. This inclusion not only highlights the practical importance of automated vulnerability analysis but also provides a standardized foundation for advancing and comparing emerging techniques in software engineering research.

The widespread adoption of deep learning in computer science has led to substantial research on its application to software vulnerability analysis. A prominent strategy involves fine-tuning Pre-trained Language Models (PLMs) such as CodeBERT [12] and UniXcoder [17], which are specifically pre-trained for code-related tasks. Although Large Language Models (LLMs) like GPT have demonstrated remarkable general-purpose reasoning capabilities, PLMs often exhibit superior performance in scenarios where input lengths remain within model limits and a moderate amount of labeled data is available for fine-tuning. Their domain alignment, efficiency, and strong performance on task-specific benchmarks make them a practical choice in many software engineering tasks [12, 15].

Within this line of work, many PLM-based methods for function-level vulnerability analysis operate over entire functions, using static inputs such as token sequences, abstract syntax trees (ASTs), or historical bug data [10, 36]. While these features are informative,

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00092

```
176  176   static void emulated_push_error(EmulatedState *card, uint64_t code)
177  177   {
178       -     EmulEvent *event = (EmulEvent *)g_malloc(sizeof(EmulEvent));
     178  +     EmulEvent *event = g_new(EmulEvent, 1);
179  179
180  180       assert(event);
181  181       event->p.error.type = EMUL_ERROR;
182  182       event->p.error.code = code;
183  183       emulated_push_event(card, event);
184  184   }
```

Figure 1: Example of unsafe memory allocation caused by a single-line vulnerability.

they do not highlight which parts of the input are responsible for the vulnerability. Figure 1 illustrates a representative example in which a security vulnerability arises from a small portion of code [7]. In particular, the use of `g_malloc(sizeof(EmulEvent))` lacks safeguards against integer overflows and type mismatches. Such manual memory allocations using `sizeof(T)` are susceptible to subtle bugs. For example, if the size is miscalculated due to a type mismatch or an overflow, the program may allocate insufficient memory, potentially causing heap overflows or memory corruption. To mitigate this issue, the vulnerable line is replaced with the safer wrapper `g_new(EmulEvent, 1)`, which performs overflow checks and ensures type-safe memory allocation.

Despite the localized nature of many vulnerabilities, conventional methods provide no explicit indication of which code regions are vulnerable, so the model has to infer relevance from uniformly treated inputs, making it difficult to recognize the nuanced, context-specific characteristics of vulnerable patterns. As a result, vulnerability-related signals may be diluted or obscured, thereby reducing the model's detection effectiveness. This limitation motivates the central research question: *Can line-level guidance that highlights the most relevant code regions help PLMs learn vulnerability-specific characteristics and improve detection and classification performance?*

To explore this question, we propose LOSVER, a two-stage framework that incorporates line-level signals to focus the model's attention on code regions more likely to contain vulnerabilities. Specifically, we use modifiability—the likelihood that a line will be modified in future commits—as a proxy for vulnerability relevance, motivated by prior work showing that modified lines frequently correspond to fault-inducing or vulnerable regions [32, 54]. In the first stage, the Modifiable Line Localizer predicts which lines are likely to be modified in future revisions. In the second stage, the Weighted Vulnerability Detector/Classifier utilizes these line-level signals to guide both detection and classification tasks by assigning greater weights to the modifiable regions. This targeted approach enhances the model's ability to capture subtle and context-dependent vulnerability patterns, leading to improved performance across both binary (e.g., vulnerability detection) and multi-class (e.g., CWE ID classification) settings.

Our key contributions are as follows:

- **Line-Level Signal-Guided Framework:** A novel two-stage framework is introduced that leverages predicted modifiable lines to guide the model's attention toward vulnerability-relevant regions. These signals provide training-time supervision and serve as inference-time cues, improving the model's ability to capture vulnerability characteristics.
- **Task-Agonistic Applicability:** The proposed framework is applicable to both binary and multi-class classification tasks, demonstrating effectiveness on vulnerability detection (Devign [56]) and CWE ID classification (Big-Vul [11]).
- **Empirical Validation of Line-Level Guidance:** Quantitative results demonstrate that incorporating predicted modifiable line signals significantly improves downstream performance. Notably, detection accuracy increases by approximately 4 percentage points when using predicted modifiable lines, and by up to 9.5 points when actual modified lines are provided. This difference highlights the impact of localization quality and suggests further gains as localization techniques improves.

The code implementation is provided for reproducibility [33, 34].

The remainder of this paper is organized as follows. Section II reviews background and related work on vulnerability detection, classification, and fault localization. Section III introduces the proposed methodology, including the preprocessing pipeline, the Modifiable Line Localizer, and the Weighted Vulnerability Detector. Section IV outlines the experimental setup, covering datasets, evaluation metrics, baselines, and implementation details. Section V reports and analyzes the experimental results. Section VI discusses threats to validity, and Section VII concludes the paper with suggestions for future work.

2 Background and Related Work

Accurately understanding and analyzing source code is fundamental to tasks such as vulnerability detection, classification, and fault localization. While these tasks differ in output, they share a common foundation: understanding the semantics and structure of code. Section II-A reviews the technical evolution from heuristic-based approaches to the adoption of Large Language Models (LLMs). Section II-B reviews task-specific advancements in vulnerability detection, classification, and fault localization.

2.1 Technical Foundations: From Heuristics to Larege Langage Models

Heuristic-Based Techniques. Early approaches to software analysis relied on manually defined heuristics and rule-based techniques. In vulnerability detection, static code metrics such as CK metrics [46] and Halstead complexity measures [6] were used to identify potentially vulnerable functions. In fault localization, methods such as Spectrum-Based Fault Localization (SBFL) [2] and Mutation-Based Fault Localization (MBFL) [38] prioritized faulty elements using test execution data. While heuristic-based techniques are lightweight and interpretable, these techniques required extensive manual design and failed to generalize across diverse codebases and defect types.

Machine Learning-Based Techniques. With the rise of machine learning (ML), various relative techniques were applied to vulnerability analysis and fault localization tasks [5, 19, 41], using features like control flow, complexity metrics, and execution

frequency. Though these models outperformed heuristics by learning correlations from data, their reliance on hand-crafted features limited their ability to capture deeper semantic patterns in source code [23].

Deep Learning-Based Techniques. Deep learning (DL) approaches advanced software analysis by removing the need for handcrafted features, learning patterns directly from raw or structured code representations. Token-based models (e.g., LSTMs, Transformers) process code as sequences, whereas structure-based models use graphs such as Abstract Syntax Trees (ASTs) or Control Flow Graphs (CFGs) to capture relational dependencies [4, 27, 42]. Although DL models can effectively learn meaningful code patterns, their reliance on task-specific training data limits their ability to capture broader code understanding or generalize across diverse projects.

Pre-trained Language Models-Based Techniques. Pre-trained Language Models (PLMs) such as CodeBERT [12] and UniXcoder [17] extend deep learning by first pre-training on large code corpora to capture general syntactic and semantic knowledge, and then fine-tuning on specific tasks like vulnerability detection or classification. This two-stage process enables PLMs to achieve strong performance even with limited task-specific labeled data [3, 51]. However, their input size limitations and dependence on fine-tuning constrain their applicability to larger-scale or cross-project scenarios.

Large Language Models-Based Techniques. LLMs such as GPT-4o scale pre-training to massive natural language and code corpora, enabling prompt-based adaptation without explicit fine-tuning. They generalize well across projects and languages, particularly in scenarios involving general-purpose code understanding, limited supervision, or long input contexts [24, 39, 40]. However, their effectiveness on fine-grained, project-specific tasks remains limited, and their large model size often incurs high computational costs.

Rationale for Model Selection. Given the limited size and scope of the target datasets, this study primarily adopts PLMs, which are well-suited for fine-tuning on function-level tasks with manageable context lengths. To assess generalizability, we additionally conduct zero-shot experiments using LLMs and evaluate whether line-level guidance improves their performance too.

2.2 Task-Specific Advances in Vulnerability Detection, Classification, and Localization

Vulnerability Detection. Although vulnerability detection and defect prediction address different types of software issues, they follow a similar approach—treating code snippets as inputs for binary classification based on the presence or absence of faults. As defect prediction is also an active and promising area of research, we reference relevant work from both areas to introduce recent advances and highlight cutting-edge trends that inform our approach.

Recent works in vulnerability detection and defect prediction explore diverse input representations to improve performance. For example, Šikić et al. [45] and Liu et al. [28] leverage hybrid input modalities such as ASTs, natural language, and token-level context using Graph Convolutional Neural Networks (GCNNs) or multi-channel neural architectures to improve defect prediction performance. Ni et al. [36] introduce a multi-modal model (MVulD) that integrates textual, graphical, and image-based representations for vulnerability detection. More recently, CSLS [52] integrates global, line-level, and structural semantics using custom preprocessing and a specialized transformer atop UniXcoder, yielding strong results in function-level vulnerability detection.

In parallel, strategies for adapting PLMs and LLMs have gained increasing attention. Wang et al. [48] apply prompt tuning to PLMs, enabling task-specific adaptation without full fine-tuning. Wang et al.[49] enhance PLMs by injecting lightweight adapter modules for parameter-efficient learning. Shestov et al. [43] fine-tune large-scale LLMs like WizardCoder [31], showing improved performance over PLM baselines.

These approaches enhance detection accuracy by extending input modalities or leveraging model adaptation strategies. However, they typically treat input features uniformly during training, without incorporating inductive biases that emphasize defect- or vulnerability-prone regions. In contrast, LOSVER introduces a line-level localization stage that identifies modifiable lines, which are then assigned greater weight by the subsequent detector—resulting in improved detection accuracy. This design is orthogonal to prior advances and can be combined with any existing PLMs or adaptation methods.

Vulnerability Classification. Vulnerability classification assigns vulnerable code to standardized categories such as CWE IDs. Recent work increasingly employs deep-learning models to learn patterns from code and associated metadata. Early work by Huang et al. [21] combines traditional feature extraction with deep neural networks to classify vulnerabilities based on textual descriptions. DeKeDVer [9] integrates a Recurrent Convolutional Neural Network (RCNN) and a Graph Neural Network (GNN) to capture both textual features and code structure.

Fu et al. [14] present AIBugHunter, which fine-tunes a BERT-based model using a multi-objective classification strategy with separate token representations and classification heads for CWE IDs and CWE Types. In parallel, Fu et al. [15] evaluate ChatGPT models across several vulnerability-related tasks, reporting that despite their scale, LLMs underperform compared to task-specific PLMs like CodeBERT—especially in scenarios where fine-tuning is not possible.

These studies highlight the importance of modality integration and task-specific adaptation in vulnerability classification. However, existing models largely overlook the internal structure of code, assigning equal importance to all lines when constructing representations for classification. LOSVER complements these efforts by introducing line-level guidance, providing additional context that aids in disambiguating vulnerability types, particularly for fine-grained classification tasks such as CWE ID prediction.

Fault Localization. Although the primary focus of our work is vulnerability analysis, our approach incorporates line-level signals that indicate which parts of the code are likely to be modified. This naturally aligns with the objectives of fault localization, where the aim is to identify code elements responsible for faults—particularly at line granularity. Fault localization techniques thus serve as a foundation for our first-stage model. Ji et al. [22] fine-tune PLMs by treating fault localization as a sequence generation problem, where the model takes line-numbered code as input and generates the faulty line number and corresponding code fragment. CodeAwareFL [55]

fine-tunes a graph-based PLM GraphCodeBERT [18], using code snippets and variable propagation chains to predict the suspiciousness of individual statements. Yang et al. [54] propose LLMAO, an LLM-based fault localization approach that fine-tunes only adapter layers atop LLMs.

Summary. This section provided a conceptual overview of the technical evolution from heuristics to ML-, DL-, PLM-, and LLM-based techniques (Section II-A), followed by the task-specific review of recent advances, primarily focusing on DL-, PLM-, and LLM-based methods (Section II-B).

3 Methodology

We propose a novel line-level modifiability signal-guided vulnerability detection and classification approach composed of two main stages: modifiable line localization and weighted vulnerability detection (Figure 2). Each stage is handled by a separately fine-tuned model to optimize performance.

The process begins with data preprocessing and filtering. Although not the main focus of this work, the preprocessing and filtering steps are essential for generating reliable line-level labels and ensuring consistent input formats across both stages.

The Modifiable Line Localizer identifies lines within a function that are likely to be modified in future commits, regardless of whether the function is labeled as vulnerable. By learning relevance signals across both vulnerable and non-vulnerable examples, the localizer highlights code regions associated with higher risk or instability. These line-level signals are then passed to the next stage to inform detection model.

The Weighted Vulnerability Detector is responsible for determining whether the target function contains any vulnerabilities. It incorporates the modifiable line list produced by the localizer to apply greater attention weights to those lines during representation aggregation. By focusing on these likely error-prone lines, the model improves its capacity to identify vulnerable patterns within the code.

Although the explanation and structure are centered on vulnerability detection, which is a binary classification task, the framework can be readily adapted for multi-class classification by replacing the final sigmoid layer with a linear classifier. This adjustment enables the model to predict among multiple vulnerability classes, while retaining the overall structure—including the modifiable line weighting mechanism—unchanged.

3.1 Data Preprocessing and Filtering

To fine-tune the Modifiable Line Localizer, line-level labels indicating modifiability are first required. These labels were obtained by repurposing an existing dataset originally labeled at the function-level for vulnerability detection. Each function in the dataset corresponds to the pre-commit version of the code and is labeled as vulnerable if the associated commit fixes a security issue, and non-vulnerable otherwise. As the dataset includes commit hashes, we retrieved the corresponding post-commit versions and identified modifiable lines by comparing them to the pre-commit code. Lines were considered modifiable if they were either deleted or modified in the commit. Newly added lines in the post-commit version were not considered, as it is difficult to attribute responsibility to specific locations in the pre-commit code. Since modifiability labels are assigned to the original version, inserted lines have no clear correspondence, making it ambiguous to determine what motivated their addition.

In addition, we excluded cases where identifying modified lines was inherently ambiguous—such as function merges, complete deletions, or substantial changes in function semantics—as these lacked a clear or consistent mapping. To further ensure the validity of our evaluation, we also filtered out samples in which none of the modified lines appeared within the PLMs input-token limit. Since the goal of this study is to assess whether model performance improves when attention is focused on more relative code regions, including samples where all modification signals are truncated would undermine the analysis. For fairness, this filtered dataset was used consistently across all models, including PLM baselines and other state-of-the-art methods during replication.

3.2 Modifiable Line Localizer

To train the Modifiable Line Localizer, input code is tokenized using the PLM's tokenizer, and sequences exceeding the model's token limit are truncated. Since the localizer aims to estimate the probability of each line being modified in the future, newline token positions are tracked during tokenization to preserve line boundaries. The PLM's encoder then generates token-level embeddings in the original token order. We denote by C_{n,k_n} the embedding of the token at position k_n in n, where n is the line number and k_n is the token index within that line. These embeddings are subsequently grouped based on the recorded newline positions, aggregating important information at the line-level.

There are multiple ways to classify each line's modifiability with embedded tokens. Yang et al. [54] directly utilized the embeddings corresponding to newline tokens, based on the assumption that these special tokens implicitly encode information about the preceding line. In contrast, we incorporate both a line representation—similar in spirit to Yang et al.—and an additional context-aware representation to enrich each line's embedding.

(1) *Line Representation:* This representation directly utilizes the embeddings of each line after passing through the PLM's encoder. While the encoder processes the entire function to produce embeddings for each token, it does not explicitly consider line-level dependencies, which means this representation alone might lack sufficient contextual awareness.
(2) *Context-Aware Representation:* This representation captures broader contextual relationships by utilizing an attention-based adapter. Specifically, we use multi-head attention where line embeddings act as queries, and the full function is used as keys and values. This allows each line to attend to relevant context in the function, resulting in an enriched line embedding of the same dimensionality. We denote the resulting context-aware embedding by CC_{n,k_n}, where n is the line number and k_n indicates the token index within that line.

The final classifier takes the concatenation of both the line representation and the context-aware representation as input. This combined input undergoes layer normalization and GELU activation to enhance stability and introduce non-linearity. A linear layer

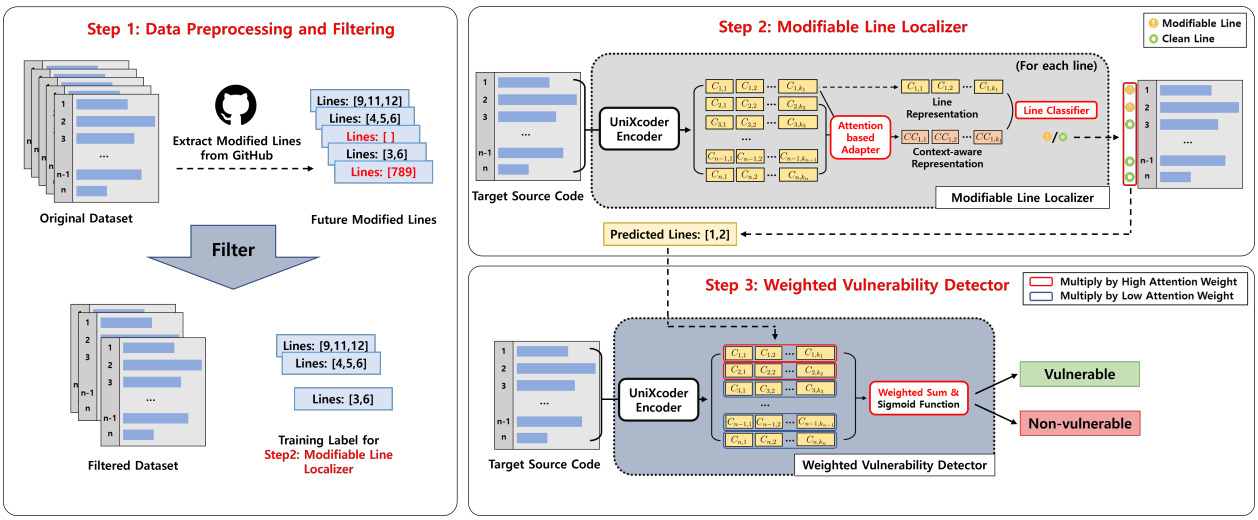

Figure 2: LOSVER overview framework for vulnerability detection.

then maps these processed features to a binary value that represents the future modifiability of each line. Since the modifiability labels are highly imbalanced—with far fewer positive instances—training is guided by a customized focal loss that assigns greater weight to positive examples.

3.3 Weighted Vulnerability Detector

Conventional PLM-based vulnerability detectors often use specialized classification heads (e.g., RobertaForSequenceClassification), which aggregate encoded representations into a single scalar and apply a sigmoid to predict vulnerability [53]. In contrast, our method adopts a weight-aware aggregation strategy guided by line-level modifiability signals.

Similar to the Modifiable Line Localizer, the Weighted Vulnerability Detector first tokenizes the input function and truncates it to fit the token limit, following the requirements of the underlying PLM. Newline token positions are recorded during tokenization to preserve line boundaries. The tokenized input is then passed through the PLM encoder to obtain token-level embeddings.

To incorporate modifiability signals, each line is assigned a scalar weight w_n depending on its predicted modifiability status:

$$w_n = \begin{cases} 5, & \text{if line } n \text{ is predicted as modifiable} \\ 1, & \text{otherwise} \end{cases} \tag{1}$$

A scaling factor of 5 was chosen, based on experiments with several values (2, 5, 10, and 20), as it consistently yielded the best performance across datasets. It was neither too low to dilute modifiability signals nor too high to suppress surrounding context. Based on this choice, line-level weights are then broadcast to the tokens in each line to construct a token-level weight vector $\{w_i\}$, where each token i inherits the weight of its corresponding line. A softmax operation is then applied across all token positions to compute normalized attention scores:

$$\alpha_i = \frac{\exp(w_i)}{\sum_j \exp(w_j)} \tag{2}$$

The resulting attention scores α_i are applied to the first dimension of each token's embedding—denoted $C_i[0]$—which serves as a scalar proxy for token-level vulnerability relevance. This dimension is explicitly trained to encode vulnerability relevance, enabling scalar-weighted aggregation with minimal additional complexity. These weighted values are then summed to obtain a single representation for the function:

$$s = \sum_i \alpha_i \cdot C_i[0] \tag{3}$$

Lastly, the weighted sum s is passed through a sigmoid activation to produce the final vulnerability probability. Training is supervised using binary cross-entropy loss, which was also employed in prior PLM-based classification tasks [12, 17, 50].

4 Experimental Setup

4.1 Research Questions

We design the following research questions (RQs) to evaluate the effectiveness and generality of LOSVER.

- **RQ1:** Does LOSVER improve vulnerability detection performance over base PLMs and prior state-of-the-art methods? *This question evaluates whether the proposed line-level guidance improves detection accuracy beyond existing models.*
- **RQ2:** How much does LOSVER's detection performance improve with increasing accuracy of line-level localization? *This question examines whether improved localization enhances detection, suggesting room for further gains.*
- **RQ3:** Does LOSVER also enhance performance on vulnerability classification, particularly in multi-class settings such as CWE ID prediction?

This explores the generality of the benefits of LOSVER in fine-grained classification tasks beyond binary detection.

4.2 Datasets

We evaluate LOSVER on three widely used datasets: Devign [56] and PrimeVul [8] for vulnerability detection, and Big-Vul [11] for vulnerability classification. Devign and Big-Vul were preprocessed and filtered to align with our methodology, whereas PrimeVul was preprocessed but not further filtered to avoid excessive data reduction. Table 1 summarizes the dataset sizes before and after applying the filtering procedures.

Devign (Vulnerability Detection). Devign is one of the most widely used benchmark dataset for vulnerability detection, composed of function-level C/C++ code snippets annotated with binary vulnerability labels (0 for non-vulnerable, 1 for vulnerable). Although Devign originally includes four projects, only two (FFmpeg and QEMU) are publicly available and used in our experiments. Notably, Devign is also part of Microsoft's official CodeXGLUE [30] benchmark. Following the preprocessing and filtering procedure described earlier, the dataset size was reduced from 27,318 to 19,771 instances when using the UniXcoder tokenizer with 512-token input limit. We followed the original 8:1:1 train/validation/test split provided by CodeXGLUE and applied our filtering procedure to each subset individually.

Big-Vul (Vulnerability Classification). Big-Vul is a large-scale dataset constructed from 348 GitHub repositories, originally designed for general vulnerability research. Each instance includes pre-commit and post-commit versions of function-level source code, along with various vulnerability-related annotations, including the Common Weakness Enumeration (CWE) ID. In this work, we use Big-Vul for multi-class vulnerability classification based on the CWE ID.

To prepare Big-Vul for our classification experiments, we applied the following additional filtering steps:

- Only vulnerable functions were retained, as the objective is to classify the type of vulnerability.
- Rare CWE IDs (those with fewer than 20 examples) were excluded to ensure a minimal amount of training data per class. Samples with blank CWE IDs were grouped into an additional category 'others' and retained.
- Instances showing no code differences between the pre- and post-commit versions were removed.

Following these procedures, the dataset was reduced from 10,900 to 6,924 instances (based on the UniXcoder tokenizer with 512-token input limit), covering 36 CWE ID categories. Figure 3 presents the distribution of these CWE ID labels. Note that, although classes with fewer than 20 instances were initially excluded, the subsequent removal of samples with no modified lines within the token limit led to a few classes falling below this threshold. This filtering order was chosen to maintain a consistent set of CWE ID labels across all PLMs and token limit configurations.

PrimeVul (Vulnerability Detection). PrimeVul is a recently introduced dataset that provides function-level vulnerable–patch pairs. It includes two variants: a full version, where clean functions greatly outnumber vulnerable ones, and a paired version, where each vulnerable function is explicitly paired with its corresponding fixed version in a one-to-one manner. In this work, we use only the paired dataset to directly evaluate detection performance on matched pairs. Unlike Devign and Big-Vul, no additional filtering was applied, as further reduction would make the dataset too small for meaningful training. The key motivation of PrimeVul is that high accuracy or F1-score does not necessarily indicate that a model truly understands vulnerabilities; rather, correctly predicting both the vulnerable and the patched functions provides stronger evidence of effective detection [8]. Following the original split provided in the PrimeVul paper, our experiments use 4,354 training pairs, 562 validation pairs, and 564 test pairs.

Table 1: Dataset Sizes Before and After Filtering

Dataset	Original	After Filtering
Devign	27,318	19,771
Big-Vul (Vulnerable functions only)	10,900	6,924
PrimeVul (paired version)	5,480 pairs	5,480 pairs

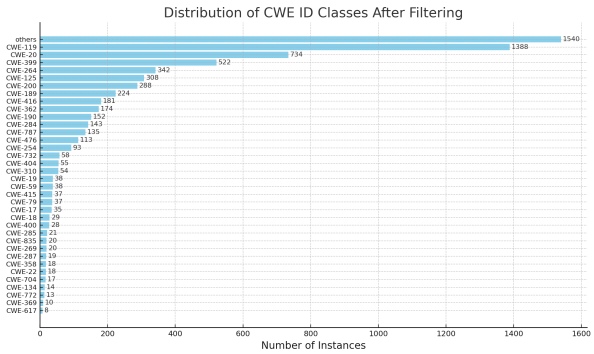

Figure 3: Distribution of CWE ID classes after all filtering steps, based on the UniXcoder tokenizer with a 512-token input limit.

4.3 Models and Baselines

To evaluate the effectiveness and generalizability of the proposed approach, we applied it across multiple Pre-trained Language Models (PLMs). Each experiment used the same PLM for both the Modifiable Line Localizer and the Weighted Vulnerability Detector to ensure consistency. We additionally compared LOSVER against two prior methods—CSLS [52] and AIBugHunter [14]—both re-evaluated using our filtered datasets.

Pre-trained Language Models.

- *UniXcoder:* We selected UniXcoder [17] as our primary PLM due to its strong performance and frequent usage in prior studies [8, 52]. We used the base-nine version (2023) with both 512- and 1,024-token input limits for compatibility and full-capacity evaluation.
- *CodeT5+:* We used the 220M encoder-only variant [50], as classification and detection tasks do not require generation capabilities. While not used as the primary PLM, CodeT5+

has also achieved state-of-the-art results across a wide range of software engineering tasks [16, 37].
- *CodeBERT:* We included CodeBERT [12], a 125M RoBERTa-based encoder, as a foundational baseline. This enables fair comparison with prior studies that used earlier-generation PLMs.

Large Language Models.
- *GPT-4o:* We evaluated GPT-4o in a zero-shot setting to investigate whether it also benefits from explicit modifiable line annotations without task-specific fine-tuning. This experiment assesses whether line-level guidance can enhance vulnerability detection performance even without parameter updates.

4.4 Evaluation Metrics

Binary Classification. For vulnerability detection, we follow standard practice in prior work [48, 50, 52] and report **accuracy** as the primary evaluation metric, given the approximately balanced class distribution (roughly 45:55). Additional metrics such as **precision**, **recall**, and **F1-score** are also included to provide a more complete picture of performance. These metrics are defined as follows:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{4}$$

$$\text{Precision} = \frac{TP}{TP + FP}, \quad \text{Recall} = \frac{TP}{TP + FN} \tag{5}$$

$$\text{F1-score} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{6}$$

In addition, for experiments on the PrimeVul dataset, we adopt the pair-wise evaluation protocol proposed in its original study [8]. This protocol treats each vulnerable–patch pair as a single unit, emphasizing whether the model can simultaneously identify the presence and absence of vulnerabilities in a textually similar context. Four outcomes are defined:

- **Pair-wise Correct Prediction (P-C):** Both elements of the pair are predicted correctly.
- **Pair-wise Vulnerable Prediction (P-V):** Both elements are incorrectly predicted as vulnerable.
- **Pair-wise Benign Prediction (P-B):** Both elements are incorrectly predicted as benign(clean).
- **Pair-wise Reversed Prediction (P-R):** The vulnerable and patched elements are incorrectly predicted with opposite labels.

Multi-Class Classification. For vulnerability classification, the dataset is substantially imbalanced across 36 CWE ID labels. We report the **weighted F1-score** as the main metric, along with precision, recall, and accuracy to better reflect model performance across all classes. The weighted F1-score is defined as:

$$\text{Weighted F1-score} = \sum_{i=1}^{K} \frac{N_i}{N} \cdot \text{F1}_i \tag{7}$$

where K is the number of classes, N_i is the number of samples in class i, N is the total number of samples, and F1_i is the F1-score for class i.

Line-Level Evaluation Metrics. To evaluate the performance of the Modifiable Line Localizer, we employ **top-N accuracy** (top-1, top-3, and top-5) and F1-score. Top-N accuracy measures whether the ground-truth modified lines are included among the top-N lines ranked by predicted modifiability scores, providing a practical indication of the model's ability to prioritize semantically critical lines that are likely to change. In contrast, the F1-score captures overall classification performance at the line level by balancing precision and recall. Together, these metrics provide a comprehensive assessment of the predicted line-level signals, which subsequently guide downstream detection and classification tasks.

4.5 Implementation Details

To ensure experimental consistency and reproducibility, we fixed the random seed to 123456 across all runs, following best practices in prior work [35, 52]. All models—including the Modifiable Line Localizer and the Weighted Vulnerability Detector/Classifier—were trained using the same hyperparameters: a learning rate of 2e-5, a batch size of 8, and the AdamW optimizer with decoupled weight decay [29]. Gradient accumulation was employed to stabilize optimization, and a linear learning rate scheduler with a 10% warm-up ratio was used to prevent abrupt updates during early training stages.

The Modifiable Line Localizer was trained for 40 epochs for both detection and classification tasks. The Weighted Vulnerability Detector was trained for 8 epochs for detection and 16 epochs for classification, as the latter required longer convergence due to its multi-class nature. Best-performing checkpoints were selected based on primary validation metrics: F1-score for localization, accuracy for detection, and weighted F1-score for classification.

For the vulnerability classification task, since the Big-Vul dataset does not provide a predefined train-test split, we employed 5-fold cross-validation to ensure robustness. Each fold maintained the original class distribution across 36 CWE ID categories. In each iteration, 4 folds were used for training, and the remaining fold was evenly divided into validation (½ fold) and test (½ fold) sets. Final results were reported as the average performance across all 5 folds.

For experiments involving GPT-4o, we used UniXcoder to first predict modifiable lines, delegating only the final vulnerability detection phase to the LLM. This hybrid setup leverages the strength of supervised PLMs in line-level localization [20], while compensating for the known limitations of general-purpose LLMs like GPT-4o, which struggle with fine-grained code-level tasks without task-specific supervision. To guide the LLM, we injected a special token <!> before each predicted modifiable line. Identical prompts were used for both the baseline and the modifiability-informed variants to ensure fair comparison. Each experiment was repeated three times, and results are reported as average performance to mitigate stochastic variance. The full prompt template is available in our replication package.

Experiments were conducted on Tesla V100-PCIE-32GB GPUs within a shared academic cluster. Due to CPU bottlenecks and scheduling constraints, full GPU capacity was not consistently utilized. For reference, end-to-end training with UniXcoder (512-token limit) for the vulnerability detection task took 12 hours on a single GPU, including both localization and detection stages.

5 EXPERIMENTAL RESULTS

RQ1: Does LOSVER improve vulnerability detection performance over base PLMs and prior state-of-the-art method?

To assess LOSVER's effectiveness, we applied it to three state-of-the-art PLMs—CodeBERT [12], CodeT5+ [50], and UniXcoder [17]—and evaluated their performance on the vulnerability detection task (Devign [56]). Table 2 summarizes the results across all models, including baseline PLMs, LOSVER-enhanced variants, the state-of-the-art CSLS method [52], and two versions of GPT-4o: with and without modifiability guidance. Across all PLMs, LOSVER provided consistent performance gains, achieving absolute accuracy improvements of 2–4 percentage points over the base models. For instance, with a 512-token input limit, integrating LOSVER with UniXcoder increased the accuracy from 65.94% to 69.94%, and the F1-score from 0.5234 to 0.6337. These results highlight LOSVER's model-agnostic design: it generalizes across both encoder-only and encoder-decoder PLMs without requiring architectural changes or task-specific adjustments.

Notably, the 512-token variant of UniXcoder outperformed its 1,024-token variant. This phenomenon likely stems from dataset filtering procedure: functions whose modifiable lines appeared between tokens 513 and 1,024 were retained for the 1,024-token setting but excluded under the 512-token setting, resulting in a harder evaluation set for the former. To verify this, UniXcoder was additionally trained and evaluated with a 1,024-token limit on the 512-token filtered dataset . It achieved the best performance among all settings, with an accuracy of 70.56%, precision of 0.7087, recall of 0.5569, and an F1-score of 0.6237. A detailed discussion of this filtering-induced data bias is provided in Section VI.

We also replicated CSLS using its publicly available implementation and our filtered version of the Devign dataset. The reproduced accuracy was 67.06%, lower than the originally reported 70.57%. This discrepancy likely stems from dataset size reduction during filtering, which may have limited generalization performance. Replication on the original Devign dataset likewise failed to reproduce the reported score, possibly due to subtle differences in experimental setup or tuning details that were not fully specified.

To examine LOSVER's applicability beyond fine-tuned PLMs, we conducted zero-shot experiments with GPT-4o, using line-level signals produced by UniXcoder to guide LLM-based vulnerability detection. Adding special tokens <!> before predicted modifiable lines increased the F1-score from 0.3007 to 0.3711, while accuracy remained around 54%. This suggests that LOSVER's line-level signals can provide useful guidance even to large-scale LLMs without task-specific tuning. Nonetheless, the performance gap between GPT-4o and fine-tuned PLMs highlights the limitations of LLMs in project-specific tasks such as vulnerability detection when used without supervision.

We further evaluated LOSVER on the PrimeVul dataset [8], which consists of vulnerability–patch pairs. For this experiment, UniXcoder with a 512-token limit was used as the base model. As shown in Table 3, LOSVER improved the pair-wise correct prediction (P-C) from 0.1011 to 0.1401, an absolute gain of about 4 percentage points. Other metrics such as F1-score and accuracy also exhibited modest improvements. Despite these gains, the P-C remains in the low teens, indicating that while LOSVER provides benefits, the model still struggles to capture the joint characteristics of vulnerable and clean code segments in this paired setting.

> **RQ1 Summary.** LOSVER consistently improves detection performance across all evaluated PLM baselines and outperforms the state-of-the-art method, CSLS. Although the improvements on GPT-4o are modest, LOSVER still provides meaningful guidance for LLMs. Similarly, on the PrimeVul dataset, LOSVER shows limited but positive gains, suggesting potential utility even on small paired datasets.

RQ2: How much does LOSVER's detection performance improve with increasing accuracy of line-level localization?

To assess the sensitivity of LOSVER to modifiability signal quality, we first evaluated the standalone performance of the Modifiable Line Localizer using the UniXcoder model with a 512-token input limit. On the filtered Devign dataset, the localizer achieved top-1, top-3, and top-5 accuracies of 55.1%, 70.5%, and 78.4%, respectively, along with an F1-score of 86.1%. These results substantially outperform LLMAO [54], a recent LLM-based fault localization approach (Table 4). Although replication was attempted on our filtered data, the publicly released implementation of LLMAO failed to converge and produced invalid outputs; thus, results are reported as published. These findings validate the effectiveness of our localization stage. Nevertheless, the predictions remain imperfect, motivating an investigation into whether further improvements in line-level accuracy could enhance downstream detection outcomes.

Six experimental conditions were created by gradually replacing incorrect predictions with ground-truth modifiable lines:

- Baseline (0% corrected): Using localizer's original predicted modifiable lines
- 20% to 80% corrected: Gradually replacing 20%, 40%, 60%, and 80% of the localizer's incorrect predictions (both false positives and false negatives) with ground-truth modified lines
- Oracle (100% corrected): Replacing all predicted modifiable lines with the ground-truth modified lines

In all settings, only the modifiable line inputs to the Weighted Vulnerability Detector were altered. All other configurations were held constant. As shown in Table 5, detection accuracy improved steadily with localization quality: gains of 1.4%, 2.4%, 3.8%, 5.7%, and 7.8% were observed under the 20% to 100% correction settings, relative to the baseline.

We further examined this phenomenon on the PrimeVul dataset. Using UniXcoder with a 512-token limit, LOSVER achieved only 0.1401 pair-wise correct prediction when relying on the localizer's predicted lines. However, replacing them with the ground-truth modified lines increased the score dramatically to 0.3316 (Table 3). This gap can be attributed to the small size and paired nature of PrimeVul, where no filtering was applied and many samples contained modified lines beyond the 512-token window. Consequently,

Table 2: Vulnerability Detection Results From Devign Dataset. PLM-based results are from a single run with fixed seed; GPT-4o results are averaged over three runs (± std)

Methods	Token Limit	Accuracy	Precision	Recall	F1-score
CodeBERT	512	64.51	0.6166	0.4423	0.5151
LOSVER (CodeBERT)	512	66.67	0.6750	0.4202	0.5180
CodeT5+	512	65.46	0.6471	0.4524	0.5325
LOSVER (CodeT5+)	512	69.29	0.6781	0.5593	0.6130
UniXcoder	512	65.94	0.6685	0.4301	0.5234
LOSVER (UniXcoder)	512	**69.94**	**0.6741**	0.5979	**0.6337**
UniXcoder	1024	65.60	0.6267	0.5307	0.5747
LOSVER (UniXcoder)	1024	68.61	0.6516	**0.6089**	0.6295
CSLS (UniXcoder) [52]	512	67.06	0.6330	0.5769	0.6037
CSLS (UniXcoder)	1024	66.02	0.6421	0.5063	0.5662
GPT-4o	-	53.47 ± 0.25	0.5203 ± 0.0069	0.2114 ± 0.0008	0.3007 ± 0.0016
GPT-4o (w/ special tokens)	-	**54.36 ± 0.48**	**0.5325 ± 0.0086**	**0.2848 ± 0.0128**	**0.3711 ± 0.0130**

Table 3: Vulnerability Detection Results on the PrimeVul dataset (paired); 512-token limit used

Methods	P-C	P-V	P-B	P-R	Accuracy	Precision	Recall	F1-score
UniXcoder	0.1011	0.2695	0.6152	0.0142	0.5434	**0.5664**	0.3706	0.4480
LOSVER (predicted lines)	**0.1401**	0.4539	0.3741	0.0319	**0.5541**	0.5501	**0.5940**	**0.5712**
LOSVER (ground-truth lines)	**0.3316**	0.4415	0.2021	0.0248	**0.6534**	**0.6237**	**0.7730**	**0.6904**

Table 4: Line-Level Localization Performance on the Filtered Devign Dataset

Method	Top-1	Top-3	Top-5
Modifiable Line Localizer (UniXcoder, ours)	**55.1%**	**70.5%**	**78.4%**
LLMAO [54]	28.1%	39.0%	60.3%

Table 5: Ablation Study: Effect of Modifiable Line Localization Accuracy

Methods	Accuracy	Precision	Recall	F1-score
LOSVER baseline (UniXcoder)	69.94	0.6741	0.5979	0.6337
LOSVER w/ 20% corrected	70.91	0.6807	0.6235	0.6509
LOSVER w/ 40% corrected	71.62	0.6737	0.6737	0.6737
LOSVER w/ 60% corrected	72.63	0.6920	0.6678	0.6797
LOSVER w/ 80% corrected	73.95	0.7145	0.6678	0.6904
LOSVER w/ 100% corrected	**75.37**	**0.7366**	**0.6748**	**0.7044**

the localizer achieved only limited performance on PrimeVul (Top-1/3/5 accuracies of 17.5%, 28.9%, and 35.1%). This weakness is largely due to the characteristics of the dataset: although the training split contains 8,708 functions, each instance is paired with its patched counterpart, effectively reducing the diversity to 4,354 unique cases. Moreover, truncation at 512 tokens often removed the actual modified lines, leaving the localizer with little or no signal to learn from in several instances. These factors collectively limited its ability to capture line-level patterns, and supplying the ground-truth lines therefore yielded substantial improvements.

These results demonstrate that the effectiveness of LOSVER is closely tied to the accuracy of line-level localization. On Devign, progressively correcting predicted lines steadily improved detection, and the 100% correction (oracle) condition reached 75.37% accuracy—an absolute improvement of 5.43 percentage points over the 69.94% LOSVER baseline—establishing a practical upper bound on the framework's potential. A similar pattern was observed on PrimeVul, where replacing predicted lines with ground-truth modified lines increased pair-wise correct prediction from 0.1401 to 0.3316 and substantially boosted overall detection performance. These findings validates the weighted aggregation design and highlights that further improvements in localization techniques can lead to substantial downstream gains.

RQ2 Summary. Detection accuracy increases monotonically as localization accuracy improves. On Devign, replacing incorrect modifiability predictions with ground-truth lines yielded up to a 7.8% relative improvement (5.43 percentage points) in accuracy. On PrimeVul, providing ground-truth modified lines raised pair-wise correct prediction from 0.1401 to 0.3316, demonstrating that accurate line-level guidance is critical for effective vulnerability detection.

RQ3: Does LOSVER also enhance performance on vulnerability classification, particularly in multi-class settings such as CWE ID prediction?

To evaluate LOSVER's effectiveness beyond binary detection, we applied it to a multi-class vulnerability classification task using CWE ID labels. While AIBugHunter [14] also evaluated on Big-Vul, their experiment used a single arbitrary data split. In contrast, we employed a more rigorous 5-fold cross-validation setup on our filtered version of Big-Vul to ensure robustness and fair comparison.

Three configurations were evaluated: (1) AIBugHunter [14], (2) CodeBERT [12], CodeT5+ [50], and UniXcoder [17] as base PLMs, and (3) each PLM combined with LOSVER. As shown in Table 6, LOSVER consistently improved classification performance across all models. The best result was achieved by LOSVER (UniXcoder, 512-token), which achieved a weighted F1-score of 73.17%, outperforming the base UniXcoder (71.10%) and AIBugHunter (69.16%).

Additionally, CodeT5+ showed relatively lower performance in the classification task compared to detection. One possible reason is the smaller dataset size used for classification, which included approximately 7,000 samples, in contrast to around 20,000 samples used for detection. Given CodeT5+'s larger parameter count (220M), the model may require more data to fully utilize its generalization capacity. In comparison, UniXcoder and CodeBERT have approximately 125M parameters each, making them less sensitive to data scarcity in this context.

RQ3 Summary. LOSVER consistently improves weighted F1-score on CWE ID classification across every PLM backbone evaluated and outperforms the recent state-of-the-art method, AIBugHunter. These findings demonstrate that LOSVER provides a robust and generalizable enhancement for multi-class code classification tasks as well, including CWE ID classification.

6 THREATS TO VALIDITY

6.1 Internal Validity

Modifiable Line Definition. In this study, modifiable lines were identified using a straightforward heuristic: lines that were deleted or edited in the post-commit version were treated as modifiable. While this approach effectively captured critical indicators for training, it did not encompass all meaningful types of modifications. Notably, our approach did not treat newly added lines as modifiable, since they lack clear counterparts in the pre-commit code and thus cannot be reliably mapped across the dataset. In our preliminary experiments, a heuristic that marked lines preceding insertions as modifiable increased the number of labels by 18.9% but degraded detection accuracy by roughly two percentage points, suggesting that the added supervision was noisy. Consistent with prior works [13, 54], we therefore rely on deleted and edited lines as more stable indicators. Nevertheless, exploring more effective ways to define modifiable lines remains an important direction, and future work could include CWE-type analyses to assess the type-specific impacts of different strategies for identifying modification signals.

Filtering-Induced Dataset Bias. To isolate the impact of modifiable lines, we filtered out samples in which no modifiable lines fell within the model's input-token limit. This procedure enabled a clearer analysis of how modifiability signals impacted detection performance but introduced a potential data inclusion bias. Specifically, longer functions—whose modified lines fell beyond shorter token limits (e.g., 512)—were excluded more frequently, but retained under longer limits (e.g., 1,024). These longer functions generally exhibited greater semantic complexity, thereby increasing classification difficulty. As a result, UniXcoder with a 512-token limit achieved better performance than its 1,024-token variant on both classification and detection tasks. CSLS also showed stronger detection performance under the 512-token setting. Despite this tradeoff, the filtering procedure ensured fair comparison within each token limit.

6.2 External Validity

Dataset Distribution Bias. The primary dataset used in this study, Devign [56], is included in the CodeXGLUE benchmark [30] and has become a widely adopted standard for evaluating vulnerability detection models. However, it contains vulnerabilities at an unusually high frequency, and non-vulnerable samples often include general defects. This differs from real-world code, where vulnerabilities are rarer. Nevertheless, the primary objective of this work is to advance model design and evaluate methodological effectiveness under controlled experimental settings, rather than to replicate the exact distribution of vulnerabilities in production software.

Function-Level Granularity. We acknowledge the inherent limitations of function-level vulnerability detection. Many vulnerabilities depend on inter-procedural or system-wide context, making some cases undecidable when restricted to single functions. Despite this limitation, function-level analysis remains the dominant paradigm due to data availability and continued research interest [8, 26]. We view our work as advancing this paradigm, while noting that the proposed line-level guidance mechanism could be extended to broader contexts (e.g., file- or module-level analysis) by applying similar attention principles at higher granularities.

7 Conclusion

This paper introduced LOSVER, a novel two-stage framework that enhances software vulnerability detection and classification by leveraging line-level modifiability signals. The framework integrates two PLM-based components: a Modifiable Line Localizer, which combines line-specific representations with context-aware attention over the PLM encoder, and a Weighted Vulnerability Detector/Classifier, which emphasizes high-risk code lines during representation aggregation. This explicit line-level guidance addresses limitations in prior PLM-based approaches that process all code uniformly at the input level, enabling more focused and interpretable vulnerability analysis.

Empirical evaluations on benchmark datasets demonstrate that LOSVER consistently improves performance across multiple PLMs. For instance, applying LOSVER to UniXcoder increased vulnerability detection accuracy from 65.94% to 69.94% on Devign, and classification F1-score on the Big-Vul dataset from 71.10% to 73.17. LOSVER also outperforms prior state-of-the-art methods such as

Table 6: Comparison Results of Different Models for Vulnerability Classification. Results are reported as mean ± standard deviation, based on 5-fold cross-validation

Methods	Token Limit	Accuracy	Precision	Recall	Weighted F1-score
AIBugHunter [14]	512	67.90 ± 3.02	0.6790 ± 0.0302	0.6747 ± 0.0285	0.6916 ± 0.0259
CodeBERT	512	68.37 ± 1.59	0.6878 ± 0.0137	0.6837 ± 0.0159	0.6798 ± 0.0145
LOSVER (CodeBERT)	512	70.15 ± 1.54	0.7055 ± 0.0142	0.7015 ± 0.0154	0.6969 ± 0.0158
CodeT5+	512	67.29 ± 1.29	0.6782 ± 0.0135	0.6729 ± 0.0129	0.6690 ± 0.0136
LOSVER (CodeT5+)	512	70.52 ± 1.33	0.7108 ± 0.0080	0.7052 ± 0.0133	0.7029 ± 0.0119
UniXcoder	512	71.55 ± 1.22	0.7220 ± 0.0142	0.7155 ± 0.0122	0.7110 ± 0.0120
LOSVER (UniXcoder)	512	**73.77 ± 1.59**	**0.7376 ± 0.0190**	**0.7377 ± 0.0159**	**0.7317 ± 0.0172**
UniXcoder	1024	71.03 ± 1.84	0.7173 ± 0.0195	0.7103 ± 0.0184	0.7065 ± 0.0185
LOSVER (UniXcoder)	1024	73.43 ± 2.41	0.7364 ± 0.0252	0.7343 ± 0.0241	0.7293 ± 0.0248

CSLS [52] and AIBugHunter [14] on both tasks. Ablation studies further reveal that enhanced line-level localization quality yields additional gains, with detection accuracy reaching 75.37% when ground-truth modified lines are provided. On the PrimeVul dataset, providing ground-truth lines increased pair-wise correct prediction from 0.1011 to 0.3316, further confirming that highlighting modifiable lines and assigning greater weight to them is a valid and effective approach. These results validate the core design of LOSVER and underscore the benefits of leveraging localized guidance in vulnerability analysis. By directing model attention to semantically critical code regions, LOSVER enhances the effectiveness and interpretability of PLMs, while offering a flexible and extensible foundation for line-aware code modeling.

While LOSVER demonstrates strong performance under controlled settings, it does not come without limitations. First, this study did not include statistical significance testing. Incorporating such analysis in future work would support more rigorous empirical comparisons. Its generalizability to other programming languages and real-world software systems also remains to be validated. Additionally, the dataset filtering procedure—used to align modifiability signals with model input limits—complicating cross-token-limit comparisons and highlighting the need for more controlled evaluation in future studies.

There is a multitude of potential research areas building on this work. These include extending the framework beyond vulnerability analysis to other software engineering tasks such as defect prediction, automated code review, and test case prioritization, where localized guidance could be beneficial. Another promising direction is adapting the method to LLM-based few-shot or in-context learning scenarios by incorporating line-level cues through prompt engineering.

Acknowledgments

This work was supported by the IITP(Institute of Information & Communications Technology Planning & Evaluation)-ITRC(Information Technology Research Center) grant funded by the Korea government (Ministry of Science and ICT) (IITP-2025-RS-2020-II201795).

References

[1] [n. d.]. The Heartbleed Bug. https://www.heartbleed.com/. Accessed: 2025-04-29.
[2] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2007. On the accuracy of spectrum-based fault localization. In *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 89–98.
[3] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333* (2021).
[4] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2017. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740* (2017).
[5] Luciano C Ascari, Lucilia Y Araki, Aurora RT Pozo, and Silvia R Vergilio. 2009. Exploring machine learning techniques for fault localization. In *2009 10th Latin American Test Workshop*. IEEE, 1–6.
[6] Bill Curtis, Sylvia B. Sheppard, Phil Milliman, MA Borst, and Tom Love. 1979. Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics. *IEEE Transactions on software engineering* 2 (1979), 96–104.
[7] Decorum-OS Project. 2024. QEMU Third Party Repository Commit 98f3433. https://github.com/Decorum-OS/third_party-qemu/commit/98f343395e937fa1db3a28dfb4f303f97cfddd6c. Commit 98f3433, Accessed: 2025-04-29.
[8] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. 2024. Vulnerability detection with code language models: How far are we? *arXiv preprint arXiv:2403.18624* (2024).
[9] Yukun Dong, Yeer Tang, Xiaotong Cheng, and Yufei Yang. 2023. DeKeDVer: A deep learning-based multi-type software vulnerability classification framework using vulnerability description and source code. *Information and Software Technology* 163 (2023), 107290.
[10] Xiaoting Du, Chenglong Li, Xiangyue Ma, and Zheng Zheng. 2024. How Does Pre-trained Language Model Perform on Deep Learning Framework Bug Prediction?. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*. 346–347.
[11] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. AC/C++ code vulnerability dataset with code changes and CVE summaries. In *Proceedings of the 17th international conference on mining software repositories*. 508–512.
[12] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *EMNLP*.
[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*. 608–620.
[14] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Yuki Kume, Van Nguyen, Dinh Phung, and John Grundy. 2024. AIBugHunter: A Practical tool for predicting, classifying and repairing software vulnerabilities. *Empirical Software Engineering* 29, 1 (2024), 4.
[15] Michael Fu, Chakkrit Kla Tantithamthavorn, Van Nguyen, and Trung Le. 2023. Chatgpt for vulnerability detection, classification, and repair: How far are we?. In *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 632–636.
[16] Shuzheng Gao, Wenxin Mao, Cuiyun Gao, Li Li, Xing Hu, Xin Xia, and Michael R Lyu. 2024. Learning in the wild: Towards leveraging unlabeled data for effectively tuning pre-trained code models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[17] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850* (2022).

[18] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366* (2020).

[19] Jacob A Harer, Louis Y Kim, Rebecca L Russell, Onur Ozdemir, Leonard R Kosta, Akshay Rangamani, Lei H Hamilton, Gabriel I Centeno, Jonathan R Key, Paul M Ellingwood, et al. 2018. Automated software vulnerability detection with machine learning. *arXiv preprint arXiv:1803.04497* (2018).

[20] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology* 33, 8 (2024), 1–79.

[21] Guoyan Huang, Yazhou Li, Qian Wang, Jiadong Ren, Yongqiang Cheng, and Xiaolin Zhao. 2019. Automatic classification method for software vulnerability based on deep neural network. *IEEE Access* 7 (2019), 28291–28298.

[22] Suhwan Ji, Sanghwa Lee, Changsup Lee, Hyeonseung Im, and Yo-Sub Han. 2024. Impact of Large Language Models of Code on Fault Localization. *arXiv preprint arXiv:2408.09657* (2024).

[23] In-Ho Kang and Hae-Yeoun Lee. 2022. A Systematic Literature Review of Software Defect Prediction Studies Using Source Code Semantic Information. *IEEE Access* 10 (2022), 36403–36425. doi:10.1109/ACCESS.2022.3164331

[24] Sungmin Kang, Gabin An, and Shin Yoo. 2024. A quantitative and qualitative evaluation of LLM-based explainable fault localization. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1424–1446.

[25] Ivan Victor Krsul. 1998. *Software vulnerability analysis*. Purdue University.

[26] Zhen Li, Ning Wang, Deqing Zou, Yating Li, Ruqian Zhang, Shouhuai Xu, Chao Zhang, and Hai Jin. 2024. On the effectiveness of function-level vulnerability detectors for inter-procedural vulnerabilities. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12.

[27] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681* (2018).

[28] Jingyu Liu, Jun Ai, Minyan Lu, Jie Wang, and Haoxiang Shi. 2023. Semantic feature learning for software defect prediction from source code and external knowledge. *Journal of Systems and Software* 204 (2023), 111753.

[29] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. arXiv:1711.05101 [cs.LG] https://arxiv.org/abs/1711.05101

[30] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664* (2021).

[31] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568* (2023).

[32] Xiangxin Meng, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2022. Improving fault localization and program repair with deep semantic features and transferred knowledge. In *Proceedings of the 44th International Conference on Software Engineering*. 1169–1180.

[33] Doha Nam. 2025. Backup code and checkpoints for Localizer and Detector from paper "LOSVER: Line-Level Modifiability Signal-Guided Vulnerability Detection and Classification". https://doi.org/10.6084/m9.figshare.29192708. Conference contribution.

[34] Doha Nam. 2025. LOSVER: Implementation of the proposed framework. https://github.com/waroad/losver. Accessed: 2025-10-02.

[35] Van-Anh Nguyen, Dai Quoc Nguyen, Van Nguyen, Trung Le, Quan Hung Tran, and Dinh Phung. 2022. Regvd: Revisiting graph neural networks for vulnerability detection. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 178–182.

[36] Chao Ni, Xinrong Guo, Yan Zhu, Xiaodan Xu, and Xiaohu Yang. 2023. Function-level vulnerability detection through fusing multi-modal knowledge. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1911–1918.

[37] Changan Niu, Chuanyi Li, Vincent Ng, Dongxiao Chen, Jidong Ge, and Bin Luo. 2023. An empirical comparison of pre-trained models of source code. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2136–2148.

[38] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: mutation-based fault localization. *Software Testing, Verification and Reliability* 25, 5-7 (2015), 605–628.

[39] Yihao Qin, Shangwen Wang, Yiling Lou, Jinhao Dong, Kaixin Wang, Xiaoling Li, and Xiaoguang Mao. 2024. Agentfl: Scaling llm-based fault localization to project-level context. *arXiv preprint arXiv:2403.16362* (2024).

[40] Md Nakhla Rafi, Dong Jae Kim, Tse-Hsun Chen, and Shaowei Wang. 2024. Enhancing Fault Localization Through Ordered Code Analysis with LLM Agents and Self-Reflection. *arXiv preprint arXiv:2409.13642* (2024).

[41] Sergio Ruggieri, Angelo Cardellicchio, Valeria Leggieri, and Giuseppina Uva. 2021. Machine-learning based vulnerability analysis of existing buildings. *Automation in Construction* 132 (2021), 103936.

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

[43] Aleksei Shestov, Rodion Levichev, Ravil Mussabayev, Evgeny Maslov, Pavel Zadorozhny, Anton Cheshkov, Rustam Mussabayev, Alymzhan Toleu, Gulmira Tolegen, and Alexander Krassovitskiy. 2025. Finetuning large language models for vulnerability detection. *IEEE Access* (2025).

[44] Nima Shiri Harzevili, Alvine Boaye Belle, Junjie Wang, Song Wang, Zhen Ming Jiang, and Nachiappan Nagappan. 2024. A systematic literature review on automated software vulnerability detection using machine learning. *Comput. Surveys* 57, 3 (2024), 1–36.

[45] Lucija Šikić, Adrian Satja Kurdija, Klemo Vladimir, and Marin Šilić. 2022. Graph neural network for source code defect prediction. *IEEE access* 10 (2022), 10402–10415.

[46] Ramanath Subramanyam and Mayuram S. Krishnan. 2003. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE Transactions on software engineering* 29, 4 (2003), 297–310.

[47] U.S. House of Representatives Committee on Oversight and Government Reform. 2018. The Equifax Data Breach. https://oversight.house.gov/wp-content/uploads/2018/12/Equifax-Report.pdf. Accessed: 2025-04-29.

[48] Chaozheng Wang, Yuanhang Yang, Cuiyun Gao, Yun Peng, Hongyu Zhang, and Michael R Lyu. 2022. No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence. In *Proceedings of the 30th ACM joint European software engineering conference and symposium on the foundations of software engineering*. 382–394.

[49] Xuanye Wang, Lu Lu, Zhanyu Yang, Qingyan Tian, and Haisha Lin. 2024. Parameter-efficient multi-classification software defect detection method based on pre-trained LLMs. *International Journal of Computational Intelligence Systems* 17, 1 (2024), 152.

[50] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922* (2023).

[51] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).

[52] Ziliang Wang, Ge Li, Jia Li, Yihong Dong, Yingfei Xiong, and Zhi Jin. 2024. Line-level Semantic Structure Learning for Code Vulnerability Detection. *arXiv preprint arXiv:2407.18877* (2024).

[53] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*. 38–45.

[54] Aidan ZH Yang, Claire Le Goues, Ruben Martins, and Vincent Hellendoorn. 2024. Large language models for test-free fault localization. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.

[55] Zhuo Zhang, Ya Li, Sha Yang, Zhanjun Zhang, and Yan Lei. 2024. Code-aware fault localization with pre-training and interpretable machine learning. *Expert Systems with Applications* 238 (2024), 121689.

[56] 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. *Advances in neural information processing systems* 32 (2019).