VNU Journal of Science: Com. Science & Com. Eng., Vol. 42, No. 2 (2026) 23–41

VNU Journal of Science:
Computer Science and Communication Engineering

Journal homepage: http://www.jcsce.vnu.edu.vn/index.php/jcsce

Original Article

A Two-Phase Vulnerability Localization Framework with Local Post-Hoc Refinement

Thu-Trang Nguyen*, Phuc Bao Pham, Son Nguyen, Hieu Dinh Vo

VNU University of Engineering and Technology, 144 Xuan Thuy, Cau Giay, Hanoi, Vietnam

Received 25th August 2025
Revised 26th March 2026; Accepted 18th May 2026

Abstract: Accurately localizing vulnerable statements is critical for ensuring software security. Multiple vulnerability localization techniques have been proposed and have demonstrated promising results. However, their effectiveness is often limited by the quality issues of the training data, such as label noise and class imbalance, which are inherent in real-world datasets. To address these challenges, this paper introduces VL-Refine, a local post-hoc refinement approach designed to enhance the robustness and reliability of existing vulnerability localization techniques. VL-Refine operates on top of the initial vulnerability predictions produced by any localization model and applies a local verification mechanism to validate and refine the vulnerability assessment for each code statement. To evaluate the performance of VL-Refine, we conduct extensive experiments on both function-level and commit-level settings. Our experiment results show that VL-Refine consistently enhances the performance of multiple state-of-the-art vulnerability localization methods. Notably, VL-Refine can improve classification accuracy by up to 46% and enable developers to discover up to 43% more vulnerabilities under fixed inspection efforts.

Keywords: Vulnerability localization, software security, label noise, class imbalance.

1. Introduction

Software vulnerabilities refer to weaknesses or flaws in software programs that attackers can exploit to compromise system integrity, access confidential data, or leak sensitive information [1, 2]. A notable example is the critical SQL injection vulnerability (CVE-2023-34362) discovered in MOVEit Transfer [1], a widely used file transfer application. This vulnerability allows unauthenticated attackers to gain access to MOVEit's database, affecting both individual and organizational users across countries such as the United States, Canada, and Germany. The estimated financial impact caused by this incident reached approximately $15 billion [3]. Moreover, data from the U.S. National Vulnerability Database (NVD) has shown a rapid increase in reported

*Corresponding author.
E-mail address: trang.nguyen@vnu.edu.vn

https://doi.org/10.25073/2588-1086/vnucsce.5612

[1] https://nvd.nist.gov/vuln/detail/cve-2023-34362

vulnerabilities in recent years. In 2024 alone, 40,303 vulnerabilities were reported, which doubled the volume recorded in 2021 [4]. These figures pose the urgent need for proactive vulnerability detection and mitigation early in the software development process.

To address this problem, various solutions and tools have been introduced [1, 5–8]. Traditional static analysis tools, such as FlawFinder [9], Coverity [10], and ClangAnalyzer [11], rely on handcrafted rules or heuristics to identify potential security issues in source code. While these tools can be effective in identifying well-known vulnerability patterns, they often suffer from limited scalability and high false positive rates [12, 13]. Recently, machine learning (ML) and deep learning (DL) based methods [1, 5, 14, 15] have been proposed to automatically learn patterns associated with vulnerable code from existing datasets. By leveraging powerful learning algorithms, these approaches have shown promising results in fully automated vulnerability detection, particularly at coarse-grained levels such as files, code slices, functions, or commits.

Due to the large number of statements within a code component (e.g, a function or a commit), even when a vulnerable code snippet is successfully detected, developers still need to spend considerable effort to pinpoint the exact vulnerable statements. To further facilitate the debugging and repair process, recent research [6–8, 16] has shifted focus toward fine-grained vulnerability localization (VL), which aims to identify specific vulnerable statements. For example, LineVD [7] formulate VL as a node classification task over program dependence graphs (PGDs). It leverages graph neural networks (GNNs) and a transformer-based model to classify nodes in a PDG as vulnerable or not. COSTA [6], on the other hand, adopts a multi-view analysis approach, representing each statement by multiple contextual views, including operation, dependence, surrounding, and vulnerability type, to compute the vulnerable score. Despite their technical sophistication, these approaches still exhibit limited practical effectiveness in real-world settings [5, 6, 17].

These approaches mainly focus on *feature engineering* or *model engineering* to enhance their VL accuracy. However, as data-driven methods, their effectiveness is heavily influenced by the quality and characteristics of the underlying training data, which are often overlooked. In practice, real-world vulnerability datasets frequently suffer from common quality issues such as *label noise* and *class imbalance*, both of which can significantly degrade model performance. Despite their substantial impact, these data-centric challenges have not been thoroughly addressed in prior work, leaving a critical gap in enhancing the robustness and reliability of VL systems.

In this work, we propose VL-Refine, a novel two-phase framework for VL that focuses on addressing the data-centric challenges of this problem. VL-Refine consists of two main components: a *localization phase*, which performs initial vulnerability prediction, and a *refinement phase*, which post-hoc refines these predictions by verifying their consistency with local lexical patterns. In the *localization phase*, a DL model is trained to capture global patterns from the training data to distinguish vulnerable from non-vulnerable code statements. While such models are capable of learning high-level semantic representations, their predictions can be influenced by the presence of noisy labels and the severe class imbalance commonly found in vulnerability datasets. As a result, the initial localization results may be biased or unreliable.

To mitigate these limitations, VL-Refine introduces a *refinement phase* that performs selective post-hoc correction based on local consistency cues. This module examines each prediction in the context of lexically similar statements and identifies inconsistencies between the predicted label and the consensus of its local neighbors. Statements with strong evidence of contradiction are flagged for correction, while predic-

tions with weak or ambiguous local evidence are preserved. This selective correction mechanism helps suppress false positives and recover missed vulnerabilities without requiring model retraining. Furthermore, the refinement phase is model-agnostic, computationally efficient, and modular by design, making it easy to integrate into existing VL pipelines as a verification or enhancement layer. By decoupling prediction and correction, VL-Refine provides a robust and generalizable solution for improving VL results.

To evaluate the performance of our approach, we conduct extensive experiments on two popular vulnerability benchmarks, Big-Vul [18] and CodeJIT [19]. VL-Refine is integrated into multiple state-of-the-art VL techniques to assess its generalizability and impact. The experimental results show that VL-Refine consistently enhances the performance of existing VL techniques. Specifically, it can improve classification accuracy by up to 46% and enable developers to identify up to 43% more vulnerable statements under the same inspection effort. These improvements highlight VL-Refine's effectiveness in mitigating the negative impacts of label noise and class imbalance, thereby improving the robustness and reliability of VL in real-world scenarios.

In summary, the main contributions of this paper are as follows:

- We propose VL-Refine, a novel two-phase framework for vulnerability localization that explicitly addresses data-centric challenges of this problem.
- We design a refinement component that is modular, model-agnostic, and computationally efficient, making it easy to integrate into existing vulnerability localization pipelines as a post-hoc enhancement layer.
- We conduct comprehensive experiments on real-world vulnerability datasets to demonstrate the effectiveness, generalizability, and efficiency of VL-Refine across multiple state-of-the-art techniques.

2. Problem Formulation

2.1. Problem Formulation

Let $C = \{c_1, \ldots, c_n\}$ denote a set of code snippets, where each snippet c_i, $1 \leq i \leq n$, is a sequence of code statements $c_i = \{s_1, \ldots, s_{m_i}\}$. The task of VL is to identify which statements $s_j \in c_i$ are vulnerable. This task can be formalized as learning a labeling function f that maps each statement s_j to a binary label $y_j \in \{0, 1\}$, where $y_j = 1$ indicates s_j is vulnerable, and $y_j = 0$ otherwise. Formally,

$$f : s_j \mapsto y_j \in \{0, 1\}$$

The VL problem is typically framed as a supervised learning task, where a training dataset $\mathcal{D} = \{(c_k, \mathbf{y}_k)\}_1^M$ is provided. In this dataset, $c_k = \{s_1, \ldots, s_{m_k}\}$ is a code snippet and $\mathbf{y}_k = \{y_1, \ldots, y_{m_k}\}$ is the corresponding sequence of ground-truth vulnerability labels for each statements in c_k. The objective of the VL problem is to learn f from the training data $\mathcal{D}$ such that it can accurately predict vulnerability labels for statements in unseen code snippets in C.

2.2. Data-Centric Challenges in Vulnerability Localization Problem

Since the VL model f is learned from $\mathcal{D}$, its effectiveness is highly dependent on the quality of that data. However, in practice, vulnerability datasets are often imperfect and commonly suffer from data quality issues, most notably, ***class imbalance*** [20, 21] and ***label noise*** [22–24]. These challenges can significantly hinder the performance and reliability of VL models.

Class imbalance: In real-world software projects, vulnerable functions/statements are far fewer than non-vulnerable ones, leading to an extremely imbalanced class distribution. For example, the ratios of vulnerable and non-vulnerable functions in Reveal [5] and Big-Vul [18] datasets are approximately 1:10 and 1:16, respectively. This imbalance becomes even more severe at the statement level. In Big-Vul, only 0.88% of all

statements are labeled as vulnerable [8, 18]. Such a severe imbalance could bias ML/DL models toward the majority class (i.e., non-vulnerable), resulting in high overall accuracy but low recall for the minority class (i.e., vulnerable). This is particularly problematic in security-critical applications, where failing to detect vulnerabilities can lead to potential exploitation risks.

Label noise: Vulnerability labels are often derived using automated techniques, such as mining version control diffs, analyzing commit messages, or leveraging static analysis tools [5, 18, 25]. While these processes are scalable and efficient, they are error-prone. They can produce incorrect labels due to incomplete/ambiguous commit messages, unrelated code changes [5], or false positives/negatives from static analyzers [12, 13]. Such noisy labels can mislead the learning process, causing models to learn spurious patterns and make wrong predictions.

Therefore, addressing these data-centric challenges is crucial for building more effective and trustworthy VL systems.

3. VL-Refine: Vulnerability Localization and Local Post-hoc Refinement

3.1. Approach Overview

The overall workflow of VL-Refine is illustrated in Figure 1. VL-Refine operates in two main phases: *localization* and *refinement*. Given a code snippet c as input, the localization phase predicts an initial vulnerable label for each statement in c. The refinement phase then evaluates these predictions by verifying their consistency with local lexical patterns and selectively correcting them when necessary. The final output of VL-Refine is label vector $\mathbf{y}$, where each element corresponds to the predicted vulnerability label of a statement in c.

3.2. Localization Phase

The localization phase is responsible for performing the initial vulnerability prediction at the statement level. This process typically involves training an ML/DL model on labeled data to learn global patterns that distinguish vulnerable statements from non-vulnerable ones. A variety of VL approaches have been proposed in prior work [6–8, 16], including GNNs, multi-view representation learning, or transformer-based models. While these methods differ in architectures and representation techniques, they all aim to capture both syntactic and semantic features of code to support accurate predictions.

To ensure broad applicability, VL-Refine is designed to be model-agnostic, allowing any off-the-shelf VL method to be seamlessly incorporated as its localization module. Specifically, any approach that takes a code snippet $c = \{s_1, \ldots, s_m\}$ as input and produces vulnerability label predictions, $\hat{\mathbf{y}} = \{\hat{y}_1, \ldots, \hat{y}_m\}$, for each statement in c, can be integrated into VL-Refine. This flexibility ensures that VL-Refine can be easily integrated with a wide range of state-of-the-art approaches, enabling users to benefit from the robustness of the refinement phase without altering their existing models or training pipelines.

3.3. Refinement Phase

Given a code snippet c and its initial vulnerability predictions $\hat{\mathbf{y}}$, the refinement phase aims to selectively correct unreliable predictions in $\hat{\mathbf{y}}$. Such unreliability could arise from data quality issues, such as class imbalance and label noise, that can bias the learning process of the VL model. To address this, we introduce a post-hoc refinement phase that verifies and adjusts initial predictions by assessing local consistency among code statements. Unlike the localization phase, which captures *global semantic* patterns using DL/ML algorithms, this phase focuses on *local lexical* patterns, enabling it to identify more explicit and reliable signals for correction.

Our key intuition is that *highly vulnerable* (or *highly safe*) statements often share similar lexical structures. For example, vulnerable statements often involve the use of dangerous or deprecated

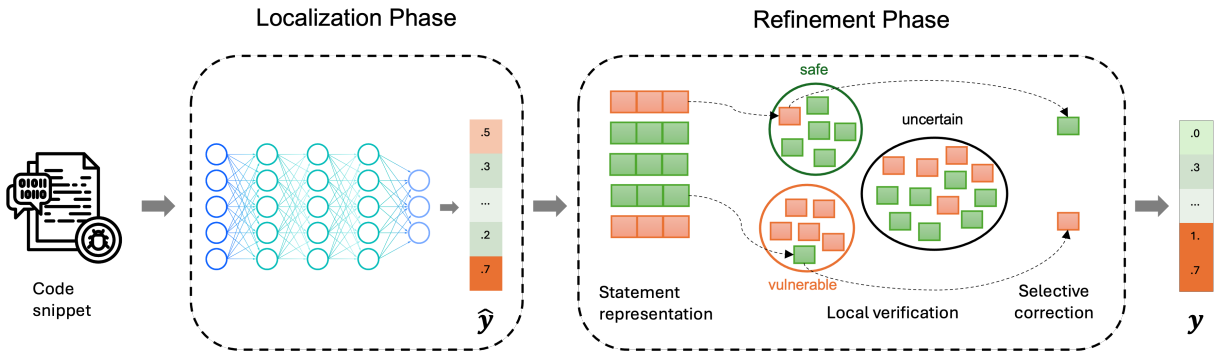

Figure 1. Approach overview.

API calls [26], while safe statements commonly include defensive programming constructs (e.g., input validation, bounds checking). By exploiting such lexical cues, the refinement phase can act as a lightweight validator that either confirms or revises the initial predictions made by the earlier stage. The refinement phase in VL-Refine contains three main steps: statement representation, local verification, and selective correction.

Statement representation. Each statement s_i is first transformed into a vector representation $\mathbf{v}_i \in \mathbb{R}^d$ using Term Frequency–Inverse Document Frequency (TF-IDF). Unlike semantic embeddings derived from pre-trained models (e.g., CodeBERT [27]), TF-IDF emphasizes surface-level lexical features, such as identifiers, operators, and control flow tokens. These features are particularly effective for identifying explicit patterns associated with vulnerability (i.e., the use of insecure APIs) or safety (e.g., input validation), making TF-IDF a suitable choice for this phase. To prevent data leakage, the TF-IDF vectorizer is fitted exclusively on the training data. The vocabulary and IDF statistics are computed using only the training set and then applied to transform both training and testing instances.

Local verification. To assess the reliability of each prediction, we compute a *refined vulnerability score* that quantifies the degree of local agreement between s_i and its similar labeled statements in the training set. This score reflects how closely s_i aligns with known vulnerable or safe patterns. A high score suggests that s_i resembles vulnerable statements, while a low score indicates stronger similarity to safe ones.

To compute the refined vulnerability score for each statement s_i, we retrieve its k nearest neighbors using the K-NN algorithm in the TF-IDF feature space. Let $\mathbf{N} = \{(s'_1, y'_1), \ldots, (s'_k, y'_k)\}$ denote the top-k neighbors of s_i, where $y'_j \in \{0, 1\}$ is the ground-truth label of the neighbor statement s'_j. Note that, $\mathbf{N}$ contains *only* statements in the training set where the ground-truth labels are available. The refined vulnerability score for s_i is computed as the similarity-weighted average of its neighbors' labels [28, 29]:

$$score_i = \frac{\sum_{j=1}^{k} sim(s_i, s'_j) \cdot y'_j}{\sum_{j=1}^{k} sim(s_i, s'_j) + \epsilon} \quad (1)$$

where $sim(s_i, s'_j)$ measures the similarity of s_i and s'_j and ϵ is a smoothing value to prevent division by zero, $\epsilon = 1e{-}12$. In this work, we employ the popular cosine similarity [30] to measure the similarity of s_i and s'_j.

$$sim(s_i, s'_j) = \frac{v_i \cdot v'_j}{||v_i|| \cdot ||v'_j||} \quad (2)$$

The resulting score $score_i \in [0, 1]$ serves as a local confidence that s_i is vulnerable or safe. This is then used to determine whether to retain

or revise the original prediction $\hat{y}_i$ of s_i, which was produced by the localization phase.

Selective correction. VL-Refine selectively refines the label for s_i only if there is strong local evidence suggesting a contradiction with the initial prediction $\hat{y}_i$. Specifically, we define two confidence thresholds: θ_v for confirming that a statement is vulnerable, and θ_s for confirming that it is safe. Based on the refined vulnerability score $score_i$, the final label y_i is determined as:

$$y_i = \begin{cases} 1 & \text{if } score_i > \theta_v, \\ 0 & \text{if } score_i > \theta_s, \\ \hat{y}_i & otherwise \end{cases} \quad (3)$$

This selective correction mechanism ensures that labels are modified only when there is clear and consistent local support. For ambiguous or borderline cases (i.e., $\theta_s \leq score_i \leq \theta_v$), the initial prediction is preserved. The thresholds θ_v and θ_s can be adjusted to balance precision and recall under different different application requirements. The impacts of these thresholds are empirically evaluated in Sec. 5.2.2 and Sec. 5.2.3.

3.4. Rationale for Using TF-IDF and K-NN in the Refinement Phase

The refinement phase in VL-Refine leverages K-NN over TF-IDF representation to validate the reliability of initial vulnerability predictions by examining local consistency of each statement. While K-NN may not be as powerful as sophisticated DL models used in the localization phase, it offers distinct advantages of robustness to both class imbalance and label noise that make it particularly suitable for the refinement phase.

First, *TF-IDF provides a lightweight yet effective representation for capturing recurring lexical patterns associated with vulnerability and safety*. Unlike deep semantic embeddings, TF-IDF emphasizes surface-level features such as identifiers, operators, and control-flow tokens, which often encode explicit signals of vulnerable behavior (e.g., usage of unsafe APIs or missing validation checks). While TF-IDF alone may not capture subtle logic-based vulnerabilities, it is well-suited for identifying *locally consistent patterns* among similar statements. This property is particularly valuable in the refinement phase, where the objective is not to relearn semantic representations, but to validate the predictions produced by the localization phase through a complementary lexical perspective.

Second, *K-NN is often more robust to class imbalance*. DL models typically optimize a loss function over the entire training dataset, which can lead to a bias toward the majority class. In contrast, K-NN is a non-parametric method that does not require global parameter learning. It makes decisions based solely on local distributions. This locality allows K-NN to be less affected by imbalanced class distributions, enabling it to better recognize rare vulnerable patterns, especially when they form tight clusters.

Third, *K-NN is inherently more resilient to label noise*. While DL models can propagate the influence of noisy labels throughout the network during training, K-NN performs instance-based inference using only a small subset of nearby samples. As a result, noisy labels affect predictions only when they are present among the top neighbors of a given instance, limiting the potential for error propagation. This makes K-NN a suitable choice in noisy labeling environments.

Furthermore, VL-Refine is designed as a two-phase framework consisting of localization and refinement. The localization phase captures global semantic patterns using powerful DL models, while the refinement phase focuses on local lexical patterns through TF-IDF and K-NN. Together, they complement each other and enhance the robustness and reliability of VL even in the presence of class imbalance and label noise.

4. Experimental Methodology

For evaluation, we seek to answer the following research questions:

- **RQ1. Performance Analysis:** How effective VL-REFINE in improving the performance of existing VL approaches?
- **RQ2: Parameter Analysis:** How do different parameters of VL-REFINE contribute to its overall results?
- **RQ3. Training Size Analysis:** How do different training data sizes impact VL-REFINE's performance?

4.1. Dataset

We evaluate our approach on two popular benchmarks BIG-VUL [18] and CODEJIT [19], both of which are widely used in vulnerability detection and localization research [6–8, 16–19, 31]. Specifically, BIG-VUL contains labeled vulnerable and non-vulnerable functions, while CODEJIT consists of vulnerable and non-vulnerable code commits. In both datasets, vulnerability labels are also available at the statement level.

To prepare the data for VL, we apply a standardized preprocessing and cleaning pipeline to each vulnerable function or commit, following the procedures established in prior work [6, 7]. First, we normalize the statement granularity to ensure alignment with the statement-level VL task. Since a single code statement may span multiple lines or a single line may contain multiple statements, we normalize the format by representing each code statement in a single line. This normalization ensures a uniform analysis unit across all datasets and VL techniques. Second, we remove trivial statements that do not carry meaningful semantics or vulnerability-related information, such as lines containing only structural tokens, e.g., "{" or "}".

Finally, to ensure consistency and fair comparison with prior work, we adopt the same training and testing splits as in LINEVUL [8] for BIG-VUL and follow the original cross-project splits for CODEJIT [19]. Specifically, for BIG-VUL, methods are randomly partitioned into training and testing sets with an approximate ratio of 8:1. For CODEJIT, we apply a project-level split, where the whole set of projects is divided into training and testing subsets using the same 8:1 ratio; all commits belonging to the selected projects are assigned to the corresponding splits.

Table 1 shows the overview of the numbers of vulnerable (#Vul. stmts) and non-vulnerable statements (#Non-vul. stmts) after preprocessing and cleaning in each set.

Table 1. Dataset overview

		#Vul. stmts	#Non-vul. stmts	Total
BIG-VUL	Training	12,383	67,619	80,002
	Testing	1,414	8,215	9,629
CODEJIT	Training	34,645	864,075	898,720
	Testing	5,276	211,755	217,031

4.2. Experimental Procedure

RQ1. Performance Analysis: To evaluate the effectiveness and generalizability of VL-REFINE, we conduct a series of controlled experiments by integrating the refinement phase into a variety of state-of-the-art VL methods. These methods cover both VL in function-level and commit-level, enabling a comprehensive evaluation of VL-REFINE across different granularities.

For localizing vulnerable statements within individual functions, we incorporate VL-REFINE into the following VL baselines:

- LINEVD [7]: This approach formulates statement-level VL as a node classification task on PDGs. It extracts features from both the graph structure and source code, which are then encoded using a GNN combined with a transformer model to estimate the vulnerability likelihood of each statement.
- VELVET [16]: VELVET combines graph-based and sequence-based neural networks to capture rich contextual dependencies, and uses them to predict vulnerable lines.

- COSTA [6]: This method represents each statement using multiple contextual views, such as operation, surrounding context, and control/data dependencies. These views are encoded using Word2Vec and processed by BiLSTM networks to learn semantic representations for vulnerability prediction.

- LLM-based VL [32]: We also include a Large Language Model (LLM)-based baseline that performs zero-shot vulnerability prediction. In this setting, we query the model (i.e., `GPT-4.1-nano` via the OpenAI API) to determine whether a code statement is vulnerable. Each statement is evaluated independently using a standardized prompt of the form: *"Given the following code statement, determine whether it is vulnerable. Answer with 'Yes' or 'No' and briefly explain."* followed by the input statement. To ensure deterministic and consistent behavior, we use a decoding configuration with temperature set to 0 and top-p set to 1.0, and limit the maximum output length to 256 tokens. The model outputs are then post-processed by mapping "Yes" to the vulnerable label and "No" to the non-vulnerable label; in ambiguous cases, we default to non-vulnerable. All queries are executed with a single run per instance without additional context or few-shot examples to ensure a fair zero-shot comparison.

For commit-level VL, where the goal is to identify vulnerable statements introduced in a code change, we integrate VL-Refine with the following techniques:

- JITFine [33]: JITFine employs CodeBERT [27] to embed both changed code and commit message. These embeddings are combined with expert features to detect buggy commits. Statement-level localization is achieved by interpreting the attention weights over tokens.

- JITLine [34]: This tool also combines changed code and expert features for commit-level classification. It then employs LIME [35], a local explainability method, to attribute prediction outcomes to individual statements for fine-grained localization.

- JIT-DIL [36]: This two-phase framework uses expert-engineered features to detect vulnerable commits and then applies a statistical language model (n-gram) to rank statements by their likelihood to be vulnerable.

- July [17]: July models code changes using Code Transformation Graphs (CTGs) and applies GNNs to classify whether the commits are vulnerable. For statement-level localization, it encodes each statement using a combination of operation, context, and topic features to measure vulnerability score.

- LLM-based VL [32]: Similar to the function-level setting, we evaluate an LLM-based baseline (GPT-4.1-nano) on commit-level localization by prompting it to assess whether each changed statement introduced by a commit is potentially vulnerable.

RQ2. Parameter Analysis: We study how VL-Refine's parameters contribute to its overall performance. In this experiment, we create different variants of VL-Refine by systematically varying key parameters in the refinement phase, including the number of nearest neighbors k, the vulnerability confirmation threshold θ_v, and the safety confirmation threshold θ_s. This analysis provides insight into the influence of each parameter and helps identify optimal configurations.

RQ3. Training Size Analysis: We examine how training data size influences the performance of VL-Refine. This analysis aims to assess the robustness of VL-Refine under varying training data conditions and identify scenarios where its performance may degrade.

4.3. Evaluation Metrics

The VL task can be formulated as a binary classification problem. Given a predicted vulnerability score for a statement, if the score is greater than a threshold θ (i.e., with $\theta = 0.5$ by default), the statement is classified as vulnerable (label 1); otherwise, it is non-vulnerable (label 0). To evaluate classification performance, we adopt standard metrics [37], ***Accuracy***, ***F1-score***. In this formulation, each statement is treated as an independent classification instance. We then compute metrics by comparing predicted and ground-truth labels across all statements in the testing set.

In addition, VL inherently involves ranking statements within a vulnerable code snippet based on their predicted scores. Ideally, truly vulnerable statements should receive higher scores and appear earlier in the ranking. Therefore, we also evaluate the performance of the VL approaches using ranking-based metrics:***Top-k Accuracy*** (Top-k Acc), ***Mean Reciprocal Rank*** (MRR), and ***Recall@X%Effort*** (R@XEff).

Top-k Acc measures whether vulnerable statements in a given code snippet appear within the top-k ranked statements. For a code snippet c_i, $topk(c_i) = 1$ if at least one vulnerable statement of c_i is ranked in top-k positions, otherwise $topk(c_i) = 0$. For a dataset of n code snippets, Top-k Acc is measured as:

$$TopK\_Acc = \frac{1}{n}\sum_{i=1}^{n} topk(c_i).$$

Following prior work [33, 34, 36, 38], we report results for $k = [3, 5]$.

MRR measures how deep one must search to find the first correctly localized vulnerable statement. For a code snippet c_i, let $rank_{c_i}$ be the rank of its first correctly localized vulnerable statement, then MRR is measured as:

$$MRR = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{rank_{c_i}}.$$

R@XEff measures the proportion of vulnerable statements that can be correctly identified within the allowed inspection effort, i.e., the top $X\%$ of examined statements in a code snippet. In this work, we applied the same procedure in existing work [17, 33], considering $X = 20\%$ statements of a given code snippet, i.e., R@20Eff.

5. Experimental Results

5.1. Performance Analysis

Table 2 shows the performance of VL-Refine when integrated with different off-the-shelf VL approaches across two settings: commit-level and function-level code snippets. Overall, VL-Refine *consistently improves the performance of all studied VL techniques in both evaluation settings*.

In the commit-level scenario, VL-Refine achieves an average improvement of 46% in classification accuracy. Especially, when integrated to JIT-DIL, VL-Refine significantly increases its accuracy from 0.31 to 0.96, and boosts the F1-Score by 275%. In the function-level setting, VL-Refine achieves an average classification accuracy improvement of 41%. The most substantial relative gain is observed with LineVD, where accuracy increases 147%, from 0.32 to 0.79.

Beyond improving overall classification performance, VL-Refine *also enhances the prioritization of vulnerable statements,* enabling developers to locate vulnerabilities earlier in the ranked list. VL-Refine improves Top-3 Acc by an average of 16% across both settings. Moreover, when reviewing the same fixed percentage of statements, VL-Refine significantly increases the number of discovered vulnerabilities. For example, inspecting 20% of the statements in a commit, JIT-DIL alone identifies 21% of the true vulnerabilities. With VL-Refine 's refinement, this number nearly doubles to 40%. On average, VL-Refine enables developers to uncover 43% more vulnerabilities in commits and 33% more in functions under the same investigation effort.

The improvement margin varies across approaches. For example, VL-Refine improves the F1-score of COSTA by only 2%, while it yields a

Table 2. Vulnerability localization performance

Setting	VL technique	Accuracy	F1-Score	Top-3 Acc	Top-5 Acc	MRR	R@20Eff
Commit-level	JULY	0.72	0.06	0.37	0.45	0.29	0.38
	JULY + VL-REFINE	0.81	0.07	0.35	0.44	0.29	0.38
	JITLINE	0.96	0.09	0.28	0.38	0.25	0.29
	JITLINE + VL-REFINE	0.98	0.08	0.32	0.42	0.29	0.36
	JITFINE	0.98	0.00	0.30	0.37	0.27	0.28
	JITFINE + VL-REFINE	0.98	0.04	0.35	0.41	0.30	0.38
	JIT-DIL	0.31	0.04	0.31	0.39	0.27	0.21
	JIT-DIL + VL-REFINE	0.96	0.15	0.49	0.54	0.42	0.40
	LLM	0.89	0.05	0.37	0.45	0.33	0.32
	LLM + VL-REFINE	0.93	0.05	0.36	0.43	0.32	0.36
Function-level	VELVET	0.87	0.31	0.72	0.83	0.63	0.41
	VELVET + VL-REFINE	0.87	0.34	0.75	0.82	0.66	0.45
	COSTA	0.80	0.51	0.81	0.88	0.73	0.53
	COSTA + VL-REFINE	0.82	0.52	0.80	0.87	0.73	0.53
	LINEVD	0.32	0.25	0.53	0.67	0.47	0.22
	LINEVD + VL-REFINE	0.79	0.37	0.71	0.83	0.63	0.43
	LLM	0.77	0.16	0.60	0.75	0.50	0.25
	LLM + VL-REFINE	0.87	0.26	0.73	0.82	0.64	0.42

substantial 113% improvement for the LLM. Importantly, across all VL approaches and evaluation settings, VL-REFINE does not degrade the performance of any baseline approaches. This highlights that VL-REFINE is a safe refinement module that can be applied to any existing VL system to improve its accuracy and ranking effectiveness.

In practice, the distribution of vulnerable and non-vulnerable statements is highly imbalanced. Particularly, the ratios of vulnerable to non-vulnerable statements in BIG-VUL and CODEJIT are approximately 1:6 and 1:40, respectively. Under such extreme class imbalance, VL approaches are prone to bias toward the majority class. For example, due to the overwhelming dominance of non-vulnerable statements in CODEJIT, JITFINE predicts almost all statements as non-vulnerable, resulting in a very high accuracy (0.98). However, this behavior almost entirely suppresses minority class predictions, leading to an F1-score of 0, as virtually no vulnerable statements are correctly identified.

By contrast, approaches such as LINEVD and JIT-DIL exhibit the opposite bias. These approaches represent the statements within the same function using highly similar contextual features, which allows vulnerability-related signals to propagate across statements. As a result, they tend to classify a large proportion of statements as vulnerable. Although this behavior yields high recall (0.77 for LINEVD and 0.72 for JIT-DIL), it also results in substantially lower precision. These results indicate a substantial number of false positives, which can significantly increase manual inspection effort and reduce the practical usefulness of the localization results.

VL-REFINE mitigates these extremes by refining the predicted labels of both vulnerable

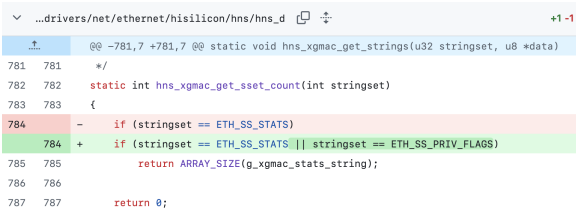

Figure 2. Vulnerability CVE-2017-18222 in Linux Kernel.

and non-vulnerable statements with high confidence, rather than favoring a single class. Specifically, VL-Refine retains the high accuracy of models like JITFine by preserving correct majority-class predictions, while selectively correcting confidently misclassified vulnerable statements, thereby improving minority-class detection. Simultaneously, when applied to models that over-predict vulnerabilities, VL-Refine reduces false positives by reclassifying confidently non-vulnerable statements. This balanced refinement enables VL-Refine to achieve a more favorable trade-off between precision and recall, leading to substantial improvements in both F1-score and overall accuracy.

These improvements stem from VL-Refine's design, which explicitly exploits local relationships among statements to guide refinement, making it more robust to severe class imbalance. Moreover, VL-Refine performs selective correction by updating labels only when the refinement confidence is sufficiently high. This strategy allows VL-Refine to retain the strengths of DL models employed in the localization phase while effectively enhancing prediction quality during the refinement phase.

Figure 2 illustrates the vulnerability CVE-2017-18222 in the Linux Kernel. In kernel version prior to 4.12, the Hisilicon Network Subsystem fails to properly handle the case of `ETH_SS_PRIV_FLAGS` when retrieving data in function `hns_xgmac_get_sset_count`. This missing check results in an incompatibility between `hns_xgmac_get_sset_count` and `ethtool_get_strings`, which can be exploited to trigger a DoS attack via buffer overflow and memory corruption, or potentially lead to other undefined behaviors. The issue is resolved in the patched version by explicitly checking `ETH_SS_PRIV_FLAGS` at line 784, as shown in Figure 2. Accordingly, in the pre-patch version, the vulnerable statement in `hns_xgmac_get_sset_count` corresponds to the conditional statement at line 784, `if (stringset == ETH_SS_STATS)`.

For this vulnerability, LineVD initially classifies all four statements in this function as vulnerable, i.e., the predicted vulnerability probability of each statement is all greater than 0.5. When the statements are ranked by these probabilities, the two non-vulnerable statements at lines 782 and 787, which have the highest predicted scores, are ranked first and second, while the truly vulnerable statement at line 784 is only ranked third. After applying VL-Refine to refine the predictions, VL-Refine correctly filters out two non-vulnerable statements, substantially improving localization accuracy. Notably, VL-Refine successfully suppresses a non-vulnerable statement that originally had a higher predicted probability than the true vulnerable statement. As a result, the actual vulnerable statement is promoted to the second position in the ranking, and the prediction accuracy for this function increases from 0.25 to 0.75.

Furthermore, to assess the robustness of the observed improvements, we perform statistical significance analysis across multiple experimental runs. Specifically, each experiment is repeated with different random seeds and the average performance is reported. We then apply the paired Wilcoxon signed-rank test to compare the performance of VL approaches with and without integrating VL-Refine. We compute 95% bootstrap confidence intervals for key metrics, including Accuracy and F1-score. The results indicate that integrating VL-Refine yields statistically significant improvements (p-value < 0.05) across all evaluated approaches, demonstrating that the ob-

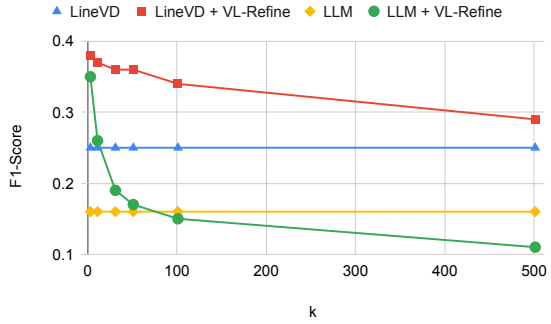

Figure 3. Impact of the number of nearest neighbors.

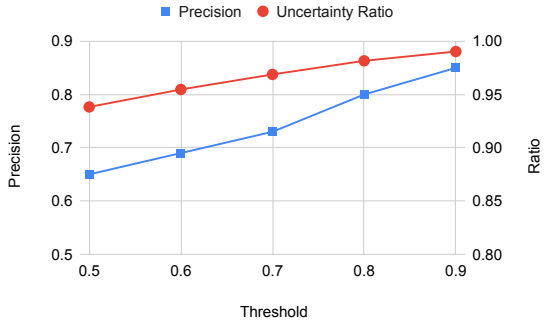

Figure 4. Impact of the vulnerability confirmation threshold θ_v.

served gains are consistent and unlikely to be attributed to random variation.

> **Answer to RQ1**: VL-Refine consistently improves the performance of existing VL techniques. It can increase vulnerable statement classification accuracy by up to 46%. Moreover, VL-Refine significantly improves the prioritization of vulnerable statements. Under the same investigation effort, it enables developers to discover up to 43% more vulnerabilities.

5.2. Parameter Analysis

5.2.1. Impact of the Number of Nearest Neighbors

Figure 3 illustrates how the number of nearest neighbors (k in the K-NN algorithm) used in the local verification step affects VL-Refine 's performance. In general, *increasing k tends to degrade the* VL-Refine*'s effectiveness*. For instance, when integrated with LLM, VL-Refine obtains a F1-Score of 0.35 at $k = 3$, which drops significantly to 0.19 at $k = 31$. A similar trend is observed when using VL-Refine with LineVD, where the F1-Score decreases from 0.38 to 0.36 as k increases from 3 to 31.

This performance degradation can be attributed to the inclusion of more distant and potentially dissimilar neighbors when k becomes larger. These less relevant neighbors introduce noise into the verification process, making it more difficult to reliably determine whether a given statement s is vulnerable or not. As a result, the accuracy of the local verification step is compromised, negatively affecting the overall performance of VL-Refine.

Furthermore, when k is set to an extremely large value, the verification step may rely on an overly broad and uninformative set of neighbors. This not only undermines the refinement process but can also deteriorate the original predictions from the localization phase. For example, LLM alone yields an F1-Score of 0.16, but when combined with VL-Refine using $k = 501$, the score drops to 0.11. These findings suggest that k should be kept small to ensure that local verification remains focused and effective.

5.2.2. Impact of the Vulnerability Confirmation Threshold

Figure 4 shows the impact of the vulnerability confirmation threshold θ_v on VL-Refine's performance. This threshold, used in Eq. 3, determines whether a code statement is confidently classified as vulnerable based on its refined vulnerability score. In this experiment, we evaluate the *precision* of the vulnerable set, i.e., the subset of statements whose refined scores exceed θ_v and are thus confidently labeled as vulnerable. All remaining statements are considered *uncertain*, as they lack sufficient local evidence for a reliable prediction.

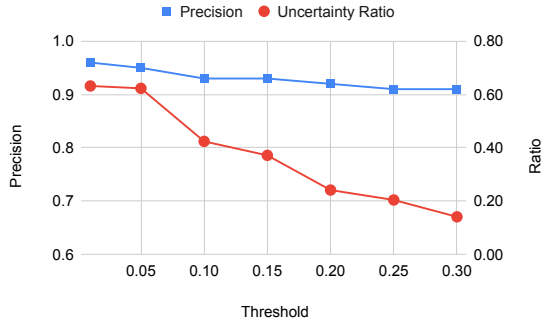

Figure 5. Impact of the safety confirmation threshold θ_s.

As seen, *increasing θ_v improves the precision of the vulnerable set, but also leads to a larger proportion of uncertain statements.* A higher threshold enforces a stricter criterion in the local verification step, allowing only statements with very strong supporting evidence to be classified as vulnerable. This leads to a more reliable identified vulnerable set, i.e, higher precision. However, the stricter condition also reduces the number of statements that meet the requirement, resulting in more statements being left unresolved, i.e., a higher uncertainty ratio.

For example, the precision of the vulnerable set is relatively increased 31%, from 0.65 to 0.85, when increasing θ_v from 0.5 to 0.9. At the same time, the proportion of statements classified as uncertain also increases by 6%. Therefore, θ_v should be carefully chosen to strike a balance, maintaining high precision while ensuring sufficient coverage of vulnerable statements. This trade-off allows VL-Refine to effectively correct unreliable predictions without leaving too many statements unresolved.

5.2.3. Impact of the Safety Confirmation Threshold

Figure 5 shows the impact of the safety confirmation threshold θ_s on VL-Refine performance. This threshold, used in Eq. 3, determines whether a code statement can be confidently classified as safe (i.e., non-vulnerable) based on its refined score. Similar to the previous experiment, we evaluate the *precision* of the safety set, i.e., the subset of statements with refined scores below θ_s that are confidently labeled as safe. Statements that do not meet this criterion are marked as *uncertain*, reflecting insufficient local evidence to support a reliable classification.

Overall, *increasing θ_s slightly reduces the precision of the identified safe statements but significantly decreases the proportion of uncertain statements*. For example, raising θ_s from 0.05 to 0.3 results in only a 5% drop in precision. Meanwhile, the proportion of unresolved statements sharply decreases by 4.5 times, from 62% to 14%.

A relatively small uncertain set suggests that a large number of statements are classified as safe based on the refined scores, leaving only a few statements to be annotated by the localization phase. However, this can increase the risk of misclassifying truly vulnerable statements as safe. For instance, at $\theta_s = 0.3$, the uncertain ratio is only 14%, which is even lower than the actual vulnerable ratio of the dataset (11%). This implies that some truly vulnerable statements may have been prematurely marked as safe due to the overly permissive threshold. Therefore, θ_s should be carefully calibrated to strike a balance between confident filtering and preserving true positives.

Answer to RQ2: The findings reveal that each parameter plays a critical role in balancing refinement accuracy, precision, and coverage. A larger number of neighbors (k) tends to introduce dissimilar and noisy neighbors, which can reduce the reliability of local verification. In addition, both θ_v and θ_s should be carefully tuned to strike a balance between precision and coverage in identifying confidently vulnerable and safe statements.

5.3. Training Size Analysis

Figure 6 shows that VL-Refine*'s performance consistently improves with increasing*

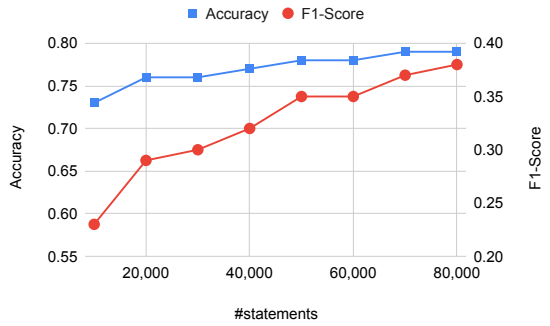

Figure 6. Impact of training data size.

training data size. For instance, with 10K training statements, VL-Refine achieves an F1-Score of 0.23, which rises by 65% to 0.38 when trained with 80K statements. This trend is expected, as a larger training set provides more diverse and representative examples for local verification, enabling more accurate vulnerability assessments. These results highlight that VL-Refine benefits substantially from larger training sets. Increasing data size enhances both prediction quality and refinement robustness, underscoring the importance of high-coverage training data in VL.

Answer to RQ3: The performance of VL-Refine is influenced by the size of the training data. A larger training set leads to more accurate and robust VL performance.

5.4. Threats to Validity

One threat to validity lies in the choice of evaluation metrics. To mitigate this threat, we follow prior work [6, 7, 16, 17] by formulating the VL problem as a binary classification task and adopting standard classification metrics such as Accuracy and F1-Score. In addition, we incorporate ranking-based metrics to assess how effectively vulnerable statements are prioritized, providing a more comprehensive evaluation.

One potential threat concerns the reliance of the refinement phase on local evidence. VL-Refine assumes that similar statements exhibit consistent labels; however, when evidence is weak, sparse, or ambiguous, the local neighborhood may not provide sufficient consensus for reliable correction. In such scenarios, VL-Refine adopts a conservative strategy and preserves the original prediction, preventing erroneous corrections but limiting its ability to fix systematic errors. To mitigate this threat, we position VL-Refine as a complementary module that applies refinement only when strong local evidence is available. As future work, incorporating richer structural and semantic representations (e.g., data-flow, control-flow, or learned embeddings) may improve its effectiveness in these challenging cases.

Another potential threat is related to evaluation benchmarks, which are primarily limited to C/C++ code. While VL-Refine is designed to be independent of programming language-specific properties, we acknowledge that empirical validation on different programming languages and environments is necessary to fully confirm its generalization. To mitigate this threat, we employ the widely-used datasets from existing studies [6–8, 16, 17, 19, 39], ensuring consistency and comparability with prior work. Furthermore, we evaluate our approach across both function-level and commit-level settings to enhance the robustness and generalizability of our findings. As future work, we plan to extend our evaluation to other programming languages and different environments to further validate the broader applicability of our method.

6. Related Work

Vulnerability Detection. Various DL-based methods have been proposed for software vulnerability detection at different levels of granularity, including components, files, and functions [1, 7, 8, 14, 16, 26, 40–42]. These approaches can be broadly categorized into two groups based on how source code is represented: token-based and graph-based models.

Token-based models. Inspired by natural language processing, token-based approaches treat source code as sequences of lexical tokens [1, 14, 26]. For instance, Russell *et al.* [14] proposed a function-level approach that encodes entire functions using Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs). Then, they used a Random Forest classifier to classify the functions as vulnerable or not. However, not every line in the code is equally important for vulnerability detection. To mitigate the influence of irrelevant statements, VulDeePecker [26] and SySeVR [1] extract code slices and encode them using Bidirectional Long Short-Term Memory Networks (BiLSTM). At the commit-level, JITFine [33] employs CodeBERT [43] to encode semantic embeddings of code changes for vulnerability prediction.

Graph-based models. Graph-based techniques model programs as structural representations [16, 39, 41, 42] such as Abstract Syntax Trees (AST), Control Flow Graphs (CFG), and Code Property Graphs (CPG). For example, Devign [15] parses functions into CPGs and applies convolutional and dense neural layers to learn graph-level features for classification. Similarly, Chakraborty *et al.* [5] leverage CPGs for function-level vulnerability detection by embedding these graphs into latent representations. At finer granularity, IVDetect [39] detects vulnerabilities at the subgraph level. For commit-level, CodeJIT [19] introduces CTGs to represent code changes, and applies GNNs to capture vulnerability-relevant patterns.

Despite their promising results, most of these approaches produce coarse-grained detection. Even when they correctly identify a vulnerable function, developers must still spend significant effort manually inspecting the code to figure out the exact vulnerable statements. To complement these approaches, VL-Refine focuses on the fine-grained VL, aiming to identify the specific vulnerable statements within a code snippet.

Vulnerability Localization. Multiple studies have tackled the problem of identifying software vulnerabilities at the statement level [7, 8, 16, 33, 34, 36, 38, 44]. For example, LineVD [7], VELVET [8], COSTA [6] are state-of-the-art release-time approaches that perform VL within a function-level code snippet. Meanwhile, JIT-Line [34], JIT-DIL [36], July [17] are just-in-time (JIT) methods that focus on localizing vulnerable statements within code changes. i.e., at the commit-level.

VL is commonly formulated as a *binary classification* task, where each statement is treated as an independent instance to be classified as either vulnerable or not. For example, LineVD [7], COSTA [6], and July [17] represent each statement along with its context, and apply neural models for classification. Specifically, LineVD [7] leverages CodeBERT [27] to encode statements along with their control/data dependent code statements. To enrich statement representations, COSTA [6] and July [17] encode each statements by multiple contextual views such as operation, dependence, and type. After feature extraction, a neural model is trained to predict the vulnerability label for each statement.

Moreover, *interpreting the detection results* to identify important features can also aid in localizing vulnerable statements. For example, JITLine [34] and IVDetect [39] first apply a vulnerability detection model to detect vulnerable functions. Then, JITLine [34] employs Lime [35], while IVDetect [39] leverages GNNExplainer [45], to explain the models' predictions to pinpoint the most likely vulnerable statements within the detected vulnerable functions.

Rather than proposing a new VL model, VL-Refine aims to enhance existing VL techniques through a local post-hoc refinement mechanism. This design specifically targets two key data-centric challenges in the VL problem: label noise and class imbalance. By leveraging local neighborhood consistency, VL-Refine refines the initial predictions of existing tools to improve precision and robustness. Importantly, VL-Refine is

model-agnostic and can be seamlessly integrated as a plug-in component with various VL methods. As demonstrated in our experiments, incorporating VL-Refine leads to significant improvements in localization performance across both function-level and commit-level settings.

LLMs for Vulnerability Detection and Localization. Recently, LLMs have gained widespread attention and shown strong potential across a variety of Software Engineering tasks [46]. Multiple techniques [32, 47–52] have been proposed to leverage LLMs for detecting and localizing software vulnerabilities. For example, Shu *et al.* [32] show that LLMs are promising for this task, with GPT-4o achieving superior performance for both function and statement level vulnerability detection.

However, Steenhoek *et al.* [50] and Li *et al.* [53] reveal that LLMs still struggle with subtle semantic reasoning, especially in scenarios involving small but critical code changes. Moreover, Sovrano *et al.* [54] indicate that LLMs' performance is sensitive to the localization of the vulnerabilities. Specifically, the models tend to underperform when vulnerabilities appear near the end of large files. These weaknesses limit their ability in real-world projects.

7. Conclusion

This paper presents VL-Refine, a lightweight and effective post-hoc refinement approach that enhances existing VL techniques. By incorporating a local verification step based on k nearest neighbors, VL-Refine refines misclassified statements through neighborhood consistency. Extensive evaluation across both function-level and commit-level benchmarks demonstrates that VL-Refine consistently improves not only classification accuracy but also the prioritization of vulnerable statements. These results enable developers to identify more vulnerabilities with less inspection effort. Moreover, VL-Refine is model-agnostic and can be seamlessly integrated into existing VL systems without requiring retraining. Its ability to mitigate the impact of label noise and class imbalance makes it a valuable addition to the VL pipeline.

Acknowledgement

This research is supported by Vietnam National Foundation for Science and Technology Development (NAFOSTED) under grant number 102.03-2023.14.

References

[1] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, Z. Chen, Sysevr: A Framework for Using Deep Learning to Detect Software Vulnerabilities, IEEE Transactions on Dependable and Secure Computing, Vol. 19, No. 4, 2021, pp. 2244–2258. doi:10.1109/TDSC.2021.3051525.

[2] S. Elder, M. R. Rahman, G. Fringer, K. Kapoor, L. Williams, A Survey on Software Vulnerability Exploitability Assessment, ACM Computing Surveys, Vol. 56, No. 8, 2024, pp. 1–41. doi:10.1145/3648610.

[3] Unpacking the MOVEit Breach: Statistics and Analysis (2023). URL https://www.emsisoft.com/en/blog/44123/unpacking-the-moveit-breach-statistics-and-analysis/ (accessed on August 6, 2025).

[4] CVE Details (2025). URL https://www.cvedetails.com (accessed on August 14, 2025).

[5] S. Chakraborty, R. Krishna, Y. Ding, B. Ray, Deep Learning Based Vulnerability Detection: Are We There Yet?, IEEE Transactions on Software Engineering, Vol. 48, No. 9, 2021, pp. 3280–3296. doi:10.1109/TSE.2021.3087402.

[6] T.-T. Nguyen, H. D. Vo, Context-Based Statement-Level Vulnerability Localization, Information and Software Technology, Vol. 169, 2024, pp. 107406. doi:10.1016/j.infsof.2024.107406.

[7] D. Hin, A. Kan, H. Chen, M. A. Babar, LineVD: Statement-Level Vulnerability Detection Using Graph Neural Networks, in: IEEE/ACM 19th International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022, IEEE, 2022, pp. 596–607. doi:10.1145/3524842.3527949.

[8] M. Fu, C. Tantithamthavorn, LineVul: A Transformer-Based Line-Level Vulnerability Prediction, in: Proceedings of the 19th International Conference on Mining Software Repositories, 2022, pp. 608–620. doi:10.1145/3524842.3528452.

[9] Flawfinder (2025). URL https://dwheeler.com/flawfinder/ (accessed on August 3, 2025).

[10] Coverity (2025). URL http://scan.coverity.com/ (accessed on August 17, 2025).

[11] ClangAnalyzer (2025). URL https://clang-analyzer.llvm.org/ (accessed on August 11, 2025).

[12] T. T. Nguyen, P. Maleehuan, T. Aoki, T. Tomita, I. Yamada, Reducing False Positives of Static Analysis for SEI CERT C Coding Standard, in: 2019 IEEE/ACM Joint 7th International Workshop on Conducting Empirical Studies in Industry (CESI) and 6th International Workshop on Software Engineering Research and Industrial Practice (SER&IP), IEEE, 2019, pp. 41–48. doi:10.1109/CESSER-IP.2019.00015.

[13] K.-T. Ngo, D.-T. Do, T.-T. Nguyen, H. D. Vo, Ranking Warnings of Static Analysis Tools Using Representation Learning, in: 2021 28th Asia-Pacific Software Engineering Conference (APSEC), IEEE, 2021, pp. 327–337. doi:10.1109/APSEC53868.2021.00040.

[14] R. L. Russell, L. Y. Kim, L. H. Hamilton, T. Lazovich, J. A. Harer, O. Ozdemir, P. M. Ellingwood, M. W. McConley, Automated Vulnerability Detection in Source Code Using Deep Representation Learning, 2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA) 2018, pp. 757–762doi:10.1109/ICMLA.2018.00120.

[15] Y. Zhou, S. Liu, J. Siow, X. Du, Y. Liu, Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks, Advances in neural information processing systems, Vol. 32, (2019).

[16] Y. Ding, S. Suneja, Y. Zheng, J. Laredo, A. Morari, G. Kaiser, B. Ray, VELVET: A Novel Ensemble Learning Approach to Automatically Locate Vulnerable Statements, in: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2022, pp. 959–970. doi:10.1109/SANER53432.2022.00114.

[17] H. D. Vo, Just-in-Time Vulnerability Detection and Localization, Journal of Computer Science and Cybernetics, Vol. 40, No. 1, 2024, pp. 79–101. doi:10.15625/1813-9663/19102.

[18] J. Fan, Y. Li, S. Wang, T. N. Nguyen, A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries, in: Proceedings of the 17th International Conference on Mining Software Repositories, 2020, pp. 508–512. doi:10.1145/3379597.3387501.

[19] S. Nguyen, T.-T. Nguyen, T. T. Vu, T.-D. Do, K.-T. Ngo, H. D. Vo, Code-Centric Learning-Based Just-in-Time Vulnerability Detection, Journal of Systems and Software, Vol. 214, 2024, pp. 112014. doi:10.1016/j.jss.2024.112014.

[20] X. Yang, S. Wang, Y. Li, S. Wang, Does Data Sampling Improve Deep Learning-Based Vulnerability Detection? Yeas! and Nays!, in: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), IEEE, 2023, pp. 2287–2298. doi:10.1109/ICSE48619.2023.00192.

[21] T. H. M. Le, M. Ali Babar, Mitigating Data Imbalance for Software Vulnerability Assessment: Does Data Augmentation Help?, in: Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, 2024, pp. 119–130. doi:10.1145/3674805.3686674.

[22] X. Nie, N. Li, K. Wang, S. Wang, X. Luo, H. Wang, Understanding and Tackling Label Errors in Deep Learning-Based Vulnerability Detection (Experience Paper), in: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, 2023, pp. 52–63. doi:10.1145/3597926.3598037.

[23] X.-C. Wen, X. Wang, C. Gao, S. Wang, Y. Liu, Z. Gu, When Less Is Enough: Positive and Unlabeled Learning Model for Vulnerability Detection, in: 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2023, pp. 345–357. doi:10.1109/ASE56229.2023.00144.

[24] W. Wang, Y. Li, A. Li, J. Zhang, W. Ma, Y. Liu, An Empirical Study on Noisy Label Learning for Program Understanding, in: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, 2024, pp. 1–12. doi:10.1145/3597503.3639217.

[25] Y. Zheng, S. Pujar, B. Lewis, L. Buratti, E. Epstein, B. Yang, J. Laredo, A. Morari, Z. Su, D2A: A Dataset Built for AI-Based Vulnerability Detection Methods Using Differential Analysis, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), IEEE, 2021, pp. 111–120. doi:10.1109/ICSE-SEIP52600.2021.00020.

[26] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, Y. Zhong, VulDeePecker: A Deep Learning-Based System for Vulnerability Detection, arXiv preprint arXiv:1801.01681 (2018). doi:10.14722/ndss.2018.23158.

[27] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, M. Zhou, CodeBERT: A Pre-Trained Model for Programming and Natural Languages, in: Findings of the Association for Computational Linguistics: EMNLP 2020, Association for Computational Linguistics, Online, 2020, pp. 1536–1547. doi:10.18653/v1/2020.findings-emnlp.139.

[28] S. A. Dudani, The Distance-Weighted K-Nearest-Neighbor Rule, IEEE Transactions on Systems, Man, and CyberneticsNo. 4, 1976, pp. 325–327. doi:10.1109/TSMC.1976.5408784.

[29] T. Hastie, R. Tibshirani, J. Friedman, et al., The Elements of Statistical Learning (2009). doi:10.1007/978-0-387-84858-7.

[30] H. Schütze, C. D. Manning, P. Raghavan, Introduction to Information Retrieval, Vol. 39, Cambridge University Press Cambridge, 2008. doi:10.1017/CBO9780511809071.

[31] H. D. Vo, T. T. Vu, S. Nguyen, Silent Vulnerability-Fixing Commit Identification Based on Graph Neural Networks, VNU Journal of Science: Computer Science and Communication Engineering, Vol. 40, No. 1, (2024). doi:10.25073/2588-1086/vnucsce.1432.

[32] H. Shu, M. Fu, J. Yu, D. Wang, C. Tantithamthavorn, J. Chen, Y. Kamei, Large Language Models for Multilingual Vulnerability Detection: How Far Are We?, arXiv preprint arXiv:2506.07503 (2025). doi:10.48550/arXiv.2506.07503.

[33] C. Ni, W. Wang, K. Yang, X. Xia, K. Liu, D. Lo, The Best of Both Worlds: Integrating Semantic Features with Expert Features for Defect Prediction and Localization, in: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2022, pp. 672–683. doi:10.1145/3540250.3549165.

[34] C. Pornprasit, C. K. Tantithamthavorn, JITLine: A Simpler, Better, Faster, Finer-Grained Just-in-Time Defect Prediction, in: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), IEEE, 2021, pp. 369–379. doi:10.1109/MSR52588.2021.00049.

[35] M. T. Ribeiro, S. Singh, C. Guestrin, 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, 2016, pp. 1135–1144. doi:10.1145/2939672.2939778.

[36] M. Yan, X. Xia, Y. Fan, A. E. Hassan, D. Lo, S. Li, Just-in-Time Defect Identification and Localization: A Two-Phase Framework, IEEE Transactions on Software Engineering, Vol. 48, No. 1, 2020, pp. 82–101. doi:10.1109/TSE.2020.2978819.

[37] G. James, D. Witten, T. Hastie, R. Tibshirani, An Introduction to Statistical Learning, Vol. 112, Springer, 2013. doi:10.1007/978-1-4614-7138-7.

[38] F. Qiu, Z. Gao, X. Xia, D. Lo, J. Grundy, X. Wang, Deep Just-in-Time Defect Localization, IEEE Transactions on Software Engineering, Vol. 48, No. 12, 2021, pp. 5068–5086. doi:10.1109/TSE.2021.3135875.

[39] Y. Li, S. Wang, T. N. Nguyen, 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, 2021, pp. 292–303. doi:10.1145/3468264.3468597.

[40] H. Hanif, M. H. N. M. Nasir, M. F. Ab Razak, A. Firdaus, N. B. Anuar, The Rise of Software Vulnerability: Taxonomy of Software Vulnerabilities Detection and Machine Learning Approaches, Journal of Network and Computer Applications, Vol. 179, 2021, pp. 103009. doi:10.1016/j.jnca.2021.103009.

[41] S. Cao, X. Sun, L. Bo, Y. Wei, B. Li, BGNN4VD: Constructing Bidirectional Graph Neural-Network for Vulnerability Detection, Information and Software Technology, Vol. 136, 2021, pp. 106576. doi:10.1016/j.infsof.2021.106576.

[42] S. Chakraborty, R. Krishna, Y. Ding, B. Ray, Deep Learning Based Vulnerability Detection: Are We There Yet?, IEEE Transactions on Software Engineering (2021). doi:10.1109/TSE.2021.3087402.

[43] U. Alon, M. Zilberstein, O. Levy, E. Yahav, Code2vec: Learning Distributed Representations of Code, Proceedings of the ACM on Programming Languages, Vol. 3, No. POPL, 2019, pp. 1–29. doi:10.1145/3290353.

[44] Y. Mirsky, G. Macon, M. Brown, C. Yagemann, M. Pruett, E. Downing, S. Mertoguno, W. Lee, {VulChecker}: Graph-Based Vulnerability Localization in Source Code, in: 32nd USENIX Security Symposium (USENIX Security 23), 2023, pp. 6557–6574.

[45] Z. Ying, D. Bourgeois, J. You, M. Zitnik, J. Leskovec, GNNExplainer: Generating Explanations for Graph Neural Networks, Advances in neural information processing systems, Vol. 32, (2019).

[46] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, H. Wang, Large Language Models for Software Engineering: A Systematic Literature Review, ACM Transactions on Software Engineering and Methodology, Vol. 33, No. 8, 2024, pp. 1–79. doi:10.1145/3695988.

[47] X. Yin, C. Ni, S. Wang, Multitask-Based Evaluation of Open-Source LLM on Software Vulnerability, IEEE Transactions on Software Engineering (2024). doi:10.1109/TSE.2024.3470333.

[48] M. D. Purba, A. Ghosh, B. J. Radford, B. Chu, Software Vulnerability Detection Using Large Language Models, in: 2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW), IEEE, 2023, pp. 112–119. doi:10.1109/ISSREW60843.2023.00058.

[49] A. Zibaeirad, M. Vieira, Reasoning with LLMs for Zero-Shot Vulnerability Detection, arXiv preprint arXiv:2503.17885 (2025). doi:10.48550/arXiv.2503.17885.

[50] B. Steenhoek, M. M. Rahman, M. K. Roy, M. S. Alam, H. Tong, S. Das, E. T. Barr, W. Le, To Err Is Machine: Vulnerability Detection Challenges LLM Reasoning, arXiv preprint arXiv:2403.17218 (2024). doi:10.48550/arXiv.2403.17218.

[51] J. Zhang, C. Wang, A. Li, W. Sun, C. Zhang, W. Ma, Y. Liu, An Empirical Study of Automated Vulnerability Localization with Large Language Models, arXiv preprint arXiv:2404.00287 (2024). doi:10.48550/arXiv.2404.00287.

[52] Y. Li, X. Li, H. Wu, Y. Zhang, X. Cheng, Y. Liu, F. Xu, S. Zhong, If LLMs Would Just Look: Simple Line-by-Line Checking Improves Vulnerability Localization, arXiv preprint arXiv:2410.15288 (2024). doi:10.48550/arXiv.2410.15288.

[53] Y. Li, X. Li, H. Wu, M. Xu, Y. Zhang, X. Cheng, F. Xu, S. Zhong, Everything You Wanted to Know about LLM-Based Vulnerability Detection but Were Afraid to Ask, arXiv preprint arXiv:2504.13474 (2025). doi:10.48550/arXiv.2504.13474.

[54] F. Sovrano, A. Bauer, A. Bacchelli, Large Language Models for In-File Vulnerability Localization Can Be "Lost in the End", Proceedings of the ACM on Software Engineering, Vol. 2, No. FSE, 2025, pp. 891–913. doi:10.1145/3715758.