SN Computer Science (2026) 7:585
https://doi.org/10.1007/s42979-026-05164-5

ORIGINAL RESEARCH

Beyond the Data: Architectural Choices for Imbalanced Learning in Vulnerability Detection

Rosmaël Zidane Lekeufack Foulefack[1] · Ulrich Duplex Djifack[1] · Alessandro Marchetto[1]

Received: 27 December 2025 / Accepted: 28 May 2026

Abstract
Static code analyzers, such as deep learning-based techniques, have proven to be essential in the security testing process. These methods rely on training an effective and reliable deep learning model to detect weaknesses and vulnerabilities. In the domain of source code analysis, the use of an adequate dataset of vulnerable and non-vulnerable code is crucial. However, it is recognized that these datasets are difficult to obtain, contain noise such as duplications, and are often strongly imbalanced. We aim to investigate the impact of these datasets on performance across various models and settings. Our methodology involves implementing two editing strategies, which include both model-level adjustments via a redefined loss function and enhancements through dataset resembling-based balancing approaches. We analyze various learning-based models, including two transformer-based models, SVDet and LineVul, two graph-based models, IVDetect and JLineVD, and two large language models, LLama and Mistral. The experimental analysis shows that resampling strategies combining over-sampling and under-sampling can improve model performance by up to 22% and yield gains of up to 1000% in terms of MCC, a suited metric for imbalanced data evaluation. Appropriate resampling methodologies, loss-function refinements, and retrieval-augmented generation techniques improve the performance of graph-based models, sequence-based models, and large language models, respectively, for vulnerability detection. However, despite these improvements, comparative analysis across models reveals that resampling and loss-function adjustments affect model stability and are not always adequate to fully address the challenges posed by severely imbalanced datasets.

Keywords Software vulnerability detection · Deep learning · Model selection · Imbalanced data

Introduction

Recent reports[1] highlight a steady rise in cyberattacks, leading to heightened security concerns worldwide. The rapid adoption of new technologies, such as the Internet of Things (IoT), 5G/6G networks, and Artificial Intelligence (AI), is fueling the continuous expansion of the software industry and its interconnectedness. However, this evolution also broadens the attack surface, thereby increasing the risks faced by software organizations [1, 2].

Detecting software vulnerabilities has become a fundamental task in security testing within the software development lifecycle. It allows for the early discovery of weaknesses in source code, many of which are documented in well-established security repositories like MITRE[2] and OWASP.[3] MITRE maintains widely used taxonomies such as the Common Weakness Enumeration (CWE) and the Common Vulnerabilities and Exposures (CVE). At the same time, OWASP identifies software security risks and provides best practices to mitigate them.

Traditional rule-driven static analysis tools often generate large numbers of false positives, which undermines their

[1] https://www.gminsights.com/industry-analysis/vulnerability-management-vm-market.

✉ Rosmaël Zidane Lekeufack Foulefack
rz.lekeufack@unitn.it

Ulrich Duplex Djifack
ulrichduplex.djifack@unitn.it

Alessandro Marchetto
alessandro.marchetto@unitn.it

[1] Department of Information Engineering and Computer Science, University of Trento, via Sommarive 9, 38123 Trento, Italy

[2] https://cwe.mitre.org/.

[3] https://owasp.org/.

Published online: 18 June 2026

reliability [3]. To overcome these limitations, the research community has increasingly turned to data-driven methods based on deep learning and large language models (LLMs) [4, 5]. Such approaches generally involve (1) building a dataset of code units (e.g., functions, code snippets, or slices) annotated as vulnerable or non-vulnerable; (2) deriving suitable representations of these code units; (3) training a machine learning model to recognize vulnerability-indicative patterns; and (4) applying the trained model to new, unseen code in order to flag potential weaknesses. Two main categories of learning-based techniques exist: sequence-oriented and graph-oriented approaches [4, 6]. The first group [7, 8] treats source code as a sequence of tokens, which are then processed by neural architectures such as LSTM networks or Transformers. The second group [9–11] represents source code through graph structures like Abstract Syntax Trees (ASTs) or Control Flow Graphs (CFGs), which are subsequently analyzed using Graph Neural Networks (GNNs). Sequence-based approaches are relatively straightforward to apply and perform well at coarse levels of granularity (e.g., function-level classification). Nevertheless, they are often less capable of pinpointing the precise source of a vulnerability compared to graph-based approaches.

In these detection paradigms, the choice of learning algorithms and the quality of training data are crucial factors. Publicly available datasets of labeled source code (e.g., [12–15]) are commonly used for evaluation, but they suffer from two critical shortcomings [16–18]: (i) the presence of noisy and redundant entries, such as incomplete or duplicated code fragments; and (ii) severe class imbalance, with non-vulnerable samples typically far outnumbering vulnerable ones. While noise in a dataset can often be reduced through a pre-processing step of data cleaning, imbalance is more challenging and it is frequently overlooked, as a limited number of tools provide details on this issue. Only a limited number of studies (e.g., [11, 19, 20]) explicitly explore countermeasures such as data resampling or specialized imbalance-aware loss functions. To date, no comprehensive examination has been carried out to assess how class imbalance affects the performance of learning-based methods for vulnerability detection. This gap in state-of-the-art leaves the model selection process lacking in terms of data efficiency compared to model parameter search.

This work aims to address this gap. Building upon a preliminary investigation [21], we analyze the impact of imbalanced training data on different types of vulnerability detection models. Initially, [21] introduced the concept of studying systematically the effect of class imbalance in software vulnerability detection and evaluated the use of resampling and loss-based strategies. However, the work focused on a single sequence-based vulnerability detection model (SVDet [22]), a limited experimental setting, and generalization. Specifically, we expand our evaluation of performance from the single sequence-based model to two sequence-based models, SVDet and Linevul [23], two graph-based methods, IVDetect [24] and JLineVD [10], and two LLM models, Llama and Mistral, through a Retrieval-Augmented Generation (RAG) approach to enhance generalization. RAG is a method of use for LLMs that integrates an external retrieval mechanism to enable the model to fetch relevant documents from a knowledge base (i.e., example samples) and utilize them to produce context-aware responses. Our experiments are conducted under multiple training settings, considering both balanced and imbalanced datasets, as well as the use of imbalance-aware loss functions.

The experimental results demonstrate that:

- The adoption of data balancing mechanisms consistently improves the performance of both sequence-based and graph-based models.
- Although refined loss functions can also enhance model performance in imbalanced data scenarios, these improvements remain insufficient, particularly for models that are highly sensitive to data imbalance, such as IVDetect.
- The integration of RAG mechanism with large LLMs improves the performance of Mistral and LLaMA for source-code analysis; however, these gains remain limited when compared to specialized sequence-based models (i.e, LineVul, SVDet) and graph-based models (i.e., JLineVD, IVDetect) trained from scratch.

The remainder of this paper is organized as follows. Section "Background" presents the background and preliminaries. Section "Methodology" describes the proposed methodology. Section "Experiment" details the experimental design. Section Experimental Results reports and analyzes the experimental results. Section "Discussion" discusses the findings and their implications. Section "Related Work" reviews related work, and Section "Conclusion" concludes the article.

Background

Software Vulnerability Detection Methods

Artificial intelligence techniques have significantly advanced static source code analysis by introducing a variety of data-driven tools. Among them, large language models (LLMs), Transformer-based architectures, and graph neural networks (GNNs) represent the most prominent approaches [25, 26]. This work focuses on turning learning methods, as recent

studies indicate that LLMs remain unsuitable for vulnerability detection due to non-determinism and hallucinations [27, 28].

Transformer architectures, originally developed for natural language processing, are widely applied to source code analysis by modeling code as token sequences. Through mechanisms such as attention masks, they capture contextual dependencies and learn patterns associated with vulnerabilities in training data. In contrast, graph-based approaches operate on graph-structured representations of code. These representations are typically generated from property graphs built by static analyzers such as Joern[4] or PHP Parser.[5] Once constructed, the graphs are used to train GNNs, which are later applied to new code fragments for vulnerability detection. Although both paradigms have shown strong results, questions remain regarding the influence of model architecture and dataset characteristics on their effectiveness.

In this study, we evaluate two sequence-based models, SVDet [22] and LineVul [23], and two graph-based models, JLineVD [10] and IVDetect [24]. Both SVDet and LineVul rely on Transformer encoders with 12 attention layers but differ in their training setups and pre-trained backbones. SVDet employs JavaBERT [29], an adaptation of BERT for Java, originally designed for function-level classification. LineVul, in contrast, uses the CodeBERT transformer encoder [30], initially trained for C/C++, LineVul detects vulnerabilities at both function and statement levels. SVDet outputs binary predictions (vulnerable or non-vulnerable) with associated probabilities. In addition, SVDet integrates SHAP, an explainability method, to highlight code tokens that contribute most to the predicted vulnerability [31].

On the graph-based side, JLineVD and IVDetect utilize different neural architectures. JLineVD is built upon LineVD [32] and enhances its subgraph construction process through the incorporation of Node2Vec-generated embeddings [33]. This approach aids in detection during source code analysis by employing Graph Attention Networks (GAT). IVDetect, instead, is based on Graph Convolutional Networks (GCN) and focuses on function-level classification.

Imbalanced Datasets

Several public datasets for vulnerability detection have been introduced in recent years [12–15, 34, 35]. These datasets typically consist of source code fragments (e.g., functions, slices, or snippets), labeled as vulnerable or non-vulnerable, and often include references to related CVEs and CWEs. While valuable, they are affected by noise (e.g., duplicates, mislabels) and, more critically, by severe class imbalance, where non-vulnerable samples largely outnumber vulnerable ones. Although data cleaning can reduce noise, imbalance remains a persistent challenge. On one hand, such distributions reflect real-world software, where vulnerabilities constitute only a small portion of the code base. On the other hand, learning algorithms generally assume a more balanced distribution and can thus become biased toward the majority class, failing to capture meaningful patterns of the minority class, the vulnerable code. As highlighted in the broader machine learning literature, ignoring imbalance may result in models that achieve high overall accuracy while underperforming on the minority class [36], which in vulnerability detection is the most critical target.

To address this issue, two categories of countermeasures are typically considered: *data-level* and *algorithm-level* techniques.

At the data level, resampling strategies aim to rebalance the training set. Simple under-sampling removes samples from the majority class, reducing bias but potentially discarding relevant information. Conversely, over-sampling duplicates minority class samples, which may cause overfitting. More advanced methods, such as SMOTE [37], generate synthetic minority samples, while ensemble-based approaches (e.g., Bagging, Boosting) combine resampling with multiple learners to improve robustness.

At the algorithm level, imbalance-aware strategies modify the learning process itself. A common solution is class weighting, where the loss function assigns higher penalties to misclassified minority samples [38]. This adjustment encourages the model to better capture rare but important cases. Other strategies include ensemble learners trained on multiple under-sampled subsets of the data, whose predictions are then aggregated. Although sequence and graph-based models have been compared across a wide range of datasets, they require different processing techniques, and many studies lack details about these pre-processing steps, which can significantly alter performance.

Finally, the choice of evaluation metrics plays a crucial role in imbalanced settings [39]. For instance, even if Accuracy is often used in the field, it can be misleading, since a model biased toward the majority class may still achieve high scores while failing to detect vulnerabilities. Metrics such as Precision, Recall, F1-score, and the Matthews Correlation Coefficient (MCC) provide a more reliable assessment of classifier performance in this context [31].

Methodology

The goal of this work is to investigate the impact of learning-based settings on sequential and graph-based models for source code vulnerability detection. To achieve this goal, we

[4] https://docs.joern.io/.

[5] https://github.com/nikic/PHP-Parser/tree/master/doc.

design a methodology that evaluates two sequential models and two graph-based models, namely *SVDet*, *LineVul*, *JLineVD*, and *IVDetect*, trained from scratch and eventually compare them to LLM models such as *Mistral* and *LLaMA*. These models were selected from the state of the art because they are widely adopted, frequently cited, and often used as baseline tools in vulnerability detection studies.

An additional consideration is the *detection granularity*. Most of selected models are originally designed for function-level detection, which has been shown to provide reliable results compared to line- or statement-level detection (typically assessed differently from one tool to another). Consequently, our training and evaluation setup is carried out under two major methodological considerations:

- Dataset balancing techniques,
- Loss function redefinition.

Balanced Training

In machine learning models for vulnerability detection, one critical challenge is the class imbalance problem. Empirical evidence shows that approximately 90% of publicly available vulnerability datasets are imbalanced, where non-vulnerable samples dominate [6, 40]. Unfortunately, many prior works do not disclose how this imbalance is addressed, which raises concerns about model bias and reliability (see Section "Related Work").

To address this, we implement and evaluate two balancing strategies: Random Undersampling (RUS) and Combined Undersampling and Oversampling (UnderOver)

Random Undersampling (RUS)

Random undersampling reduces the number of majority-class samples (non-vulnerable functions) by randomly discarding them until both classes have approximately equal cardinalities. Formally, given a dataset $D = \{(x_i, y_i)\}_{i=1}^{N}$, where x_i represents the code function/method and $y_i \in \{0, 1\}$ indicates the corresponding labels for non-vulnerable and vulnerable instances, we define the majority class C_{maj} and the minority class C_{min}. RUS ensures that:

$$\left|C'_{maj}\right| \approx |C_{min}|, \tag{1}$$

where $C'_{maj} \subseteq C_{maj}$ is the randomly sampled subset. While this technique reduces bias towards the majority class, it risks discarding potentially useful training data, which can lead to a loss of diversity and lower generalization ability when the imbalance ratio is high.

UnderOver Sampling

To address the limitations of pure undersampling, we investigate the adoption of a hybrid strategy, termed **UnderOver**, which involves randomly selecting and duplicating samples from the minority class. While this method is straightforward and computationally inexpensive, it introduces the risk of overfitting [41]. To mitigate such risks, more sophisticated techniques like SMOTE (Synthetic Minority Over-sampling Technique), Bagging, and Boosting have been developed. SMOTE, for instance, generates synthetic instances by interpolating between existing minority samples. In contrast, Bagging and Boosting are ensemble-based methods that enhance resampling through aggregation and weighted learning strategies.

Our approach deliberately avoids synthetic sample generation using tools such as LLMs, as this would require extensive expert validation to ensure the correctness and vulnerability of the generated code, thereby increasing the risk of compromising dataset integrity. Similarly, we refrain from incorporating vulnerable samples from external datasets to preserve intra-dataset diversity and avoid redundancy. Formally, the transformation produces balanced sets such that (2) with synthetic augmentation for both classes. To this end, we apply a controlled over-under sampling strategy to the training set. We oversample the minority class by duplicating existing samples, ensuring that no individual sample is duplicated more than once to mitigate overfitting. Then, undersample the majority class by randomly removing samples until its size matches the size of the augmented minority class, which results in a balanced training set.

$$\left|C'_{maj}\right| \approx \left|C'_{min}\right| \tag{2}$$

where $C'_{maj} \subseteq C_{maj}$ is the randomly sampled subset, and C'_{min} is the randomly over-sampled set of C_{min}.

Loss Function Redefinition

The second dimension of our methodology investigates the effect of different loss functions on model performance under imbalanced datasets. Specifically, we compare the baseline loss functions used in the selected models with a modified focal loss function.

Binary Cross-Entropy (BCE) Loss

Models such as **SVDet** and **LineVul** employ the Binary Cross-Entropy (BCE) loss, defined as:

$$\mathcal{L}_{BCE} = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \log\left(\hat{y}_i\right) + \left(1-y_i\right)\log\left(1-\hat{y}_i\right)\right], \quad (3)$$

where $y_i \in \{0,1\}$ denotes the ground truth label, and $\hat{y}_i \in [0,1]$ is the predicted probability. BCE assumes balanced class contributions, which may not hold in imbalanced datasets.

Categorical Cross-Entropy (CCE) Loss

Graph-based models such as **JLineVD** and **IVDetect** often adopt the Categorical Cross-Entropy (CCE) loss:

$$\mathcal{L}_{CCE} = -\sum_{i=1}^{N}\sum_{c=1}^{C} y_{i,c}\log\left(\hat{y}_{i,c}\right), \quad (4)$$

where C is the number of classes, $y_{i,c}$ is the binary indicator of class membership, and $\hat{y}_{i,c}$ is the predicted probability for class c. Although CCE generalizes to multi-class problems, it similarly treats all classes equally, which is problematic for highly imbalanced data.

Focal Loss

To address the imbalance problem, we integrate the **Focal Loss** [42], which modifies BCE by adding a modulating factor to focus learning on hard-to-classify examples, i.e, a function that focuses on the samples that are wrongly classified and applies a weighting factor that depends on the real number of samples of each classification class in the dataset. The focal loss is defined as:

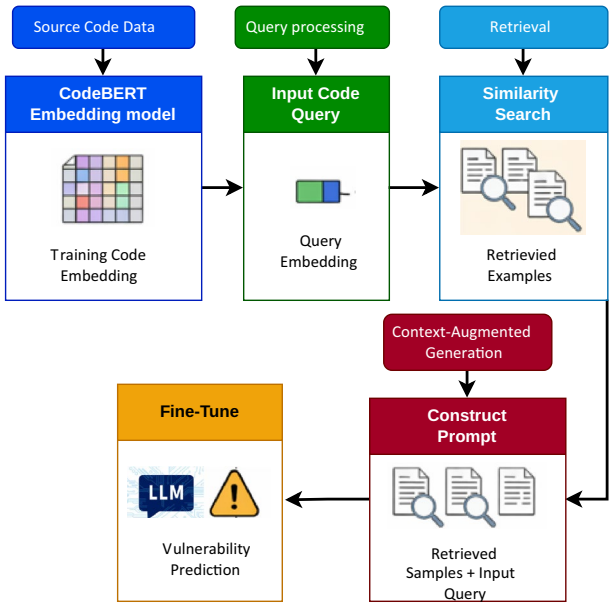

Fig. 1 RAG-based vulnerability detection pipeline

$$\begin{aligned}\mathcal{L}_{FL} = & -\frac{1}{N}\sum_{i=1}^{N}\alpha\left(1-\hat{y}_i\right)^{\beta} y_i \log\left(\hat{y}_i\right) \\ & +(1-\alpha)\,\hat{y}_i^{\beta}\left(1-y_i\right)\log\left(1-\hat{y}_i\right), \end{aligned} \quad (5)$$

where $\alpha \in [0,1]$ balances the importance of positive (vulnerable) and negative (non-vulnerable) classes. $\beta \geq 0$ is the focusing parameter that reduces the relative loss for well-classified examples, directing more focus on hard misclassified cases.

In this study, we substitute the original loss functions of each baseline model with focal loss to examine the improvement in handling the severe class imbalance present in source code vulnerability datasets.

Given that our task is formulated as a binary classification problem, Categorical Cross-Entropy (CCE) and Binary Cross-Entropy (BCE) are mathematically equivalent. Accordingly, we address loss refinement in a uniform manner for all considered models.

LLM-Based Approaches

In addition to deep learning models, we also evaluate large language models for vulnerability classification. In particular, we use *Llama-3.1-8B-Instruct*, and *Mistral-7B-Instruct-v0.3*, two widely used open-source instruction-tuned models. Llama is a family of language models developed by Meta,[6] designed to be efficient and adaptable for many code and text-related tasks. Mistral, developed by Mistral[7] AI, follows a similar objective and is known for its strong performance on reasoning and programming benchmarks despite its relatively small size.

We selected these two models because they have shown competitive results in code understanding tasks, according to recent studies and reports [43]. However, LLMs still suffer from important limitations, such as hallucination and non-deterministic behaviour, which can affect their reliability in security-related tasks [44]. To reduce these issues, we adopt a context-aware strategy based on Retrieval-Augmented Generation (RAG). This mechanism provides the model with relevant examples before performing the classification [45].

For each query, we retrieve $k = 3$ representative samples from the training data corresponding to the predicted CWE class. These retrieved examples are incorporated into the prompt to guide the model's vulnerability prediction. The retrieval and query process is summarized in Fig. 1. To enable similarity-based retrieval, we first compute vector embeddings of the training samples using the CodeBERT [30] model and store them as NumPy matrices. During inference, the query code is also embedded using the same

[6] https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct.

[7] https://huggingface.co/mistralai/Mistral-7B-Instruct-v0.3.

model, and similarity search is performed between the query embedding and the stored embeddings to retrieve the relevant samples.

CodeBERT is a pre-trained language model trained on multiple programming languages and widely used to generate contextualized vector representations of source code. These embeddings enable the retrieval of semantically similar code examples that can provide additional context to the model. To further improve determinism during generation, we fine-tune model versions following the methodology proposed in [46] and configure both models with a temperature of 0.0 and a top-p value of 0.95.

With these settings, the models are prompted to output only the final classification label without explanations. The instruction template is as follows:

```
instruct = "Return only 0 for non-vulnerable
or 1 for vulnerable.
No explanation needed."
prompt = (
"You are a security vulnerability
classification assistant.\n\n"
"Use the examples below as guidance.\n\n"
+ "\n---\n".join(blocks)
+ "\nGiven the following code, " + instruct
+ "\n\n"
+ "Query code:\n"
+ query_code
+ "\n\nAnswer:")
```

Experiment

Research Questions

With the aim of analyzing the effects of data imbalance on the performance of learning-based vulnerability detection, we designed a set of experiments involving both sequence-based and graph-based models. Specifically, we address the following research questions:

- **RQ1**: *How do different resampling strategies influence the performance of learning-based vulnerability detection models when trained on imbalanced datasets?*
- **RQ2**: *What is the impact of loss functions tailored for imbalanced data on the effectiveness of learning-based vulnerability detection models?*
- **RQ3**: *How do sequence and graph-based models compare to large language models on source code vulnerability detection task?*

To investigate **RQ1**, we trained models (e.g., SVDet, JLineVD, IVDetect, and LineVul) using both the original imbalanced training data and its balanced counterparts, generated through two approaches: undersampling (*RUS*) and a combined under-over sampling method (*UnderOver*). Public datasets were split into training, validation, and test sets using an 80:10:10 ratio. Only the training set was balanced to reduce the risk of overfitting and ensure a fair evaluation on the original test distribution.

To address **RQ2**, we assessed the effect of loss functions designed for class imbalance. We compared models trained with the standard Binary Cross-Entropy Loss (*BCELoss*) or Categorical Cross-Entropy Loss(*CCELoss*) against a version trained using the Binary Class-Balanced Focal Loss (*BCBFocalLoss*). While BCELoss treats all prediction errors equally, it may bias the model toward the majority class in imbalanced scenarios. In contrast, BCBFocalLoss incorporates both class frequency-based weighting and a focusing mechanism that emphasizes difficult-to-classify (often minority class) samples, thereby improving sensitivity to vulnerabilities.

Finally, **RQ3** evaluates the effectiveness of two LLM models compared to sequence-based and graph-based models.

Evaluation Metrics

To assess the effectiveness of the models in distinguishing between vulnerable and non-vulnerable code, we adopted commonly used classification metrics.

- **Precision:** proportion of correctly identified vulnerable samples among all predicted vulnerable samples;

$$Precision = \frac{TP}{TP+FP} \tag{6}$$

- **Recall:** proportion of actual vulnerable samples correctly detected;

$$Recall = \frac{TP}{TP+FN} \tag{7}$$

- **F1-score:** harmonic mean of Precision and Recall, balancing the two;

$$F1 = \frac{2TP}{2TP+FP+FN} \tag{8}$$

- **MCC:** a correlation coefficient considering all four outcomes of the confusion matrix;

$$MCC = \frac{(TP \times TN - FP \times FN)}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}} \tag{9}$$

Here, TP and TN denote correctly classified vulnerable and non-vulnerable fragments, respectively, whereas FP and FN denote incorrect predictions.

Datasets

Our study considers four publicly available Java datasets: CrossVul [15], CVEfixes [35], ProjectKB [13], and SARD.[8]

Table 1 summarizes their statistics, while Table 2 details the distribution of the most frequent CWE types. Table 1 reports the structural and statistical characteristics of each dataset. In particular, *Funct.* denotes the total number of Java functions/methods, *Vuln.Funct.%* represents the percentage of vulnerable ones, *CWEs* indicates the number of distinct CWE categories, *Funct. per CWE % (avg (std))* reports the average and standard deviation of the proportion of functions per CWE type, and *Textual sim.% (avg (std))* provides the average and standard deviation of pairwise Jaccard textual similarity among functions, capturing dataset homogeneity. Table 2 further reports, for each dataset, the three most frequent CWE categories (*CWE*), their proportion over the total number of functions (*Funct.%*), and the percentage of vulnerable functions within each CWE category (*Vuln.Funct.%*).

CrossVul and CVEfixes are relatively small datasets, whereas ProjectKB and SARD are considerably larger. All datasets are highly imbalanced, with vulnerable samples accounting for about 12% in CrossVul, CVEfixes, and ProjectKB, and 29.5% in SARD. Although still imbalanced, SARD provides a larger proportion of vulnerable samples, reducing skew compared to the others.

The datasets also differ in vulnerability diversity. CrossVul, CVEfixes, and ProjectKB cover between 44 and 64 unique CWE types, with relatively few samples per CWE (1.5–2.3% on average). By contrast, SARD includes only 21 CWE types but with higher representation per type (4.76% on average). Textual similarity analysis (based on Jaccard similarity) shows SARD to contain more homogeneous code, while ProjectKB is the most heterogeneous.

Table 2 further shows that, in CrossVul, CVEfixes, and ProjectKB, the three most frequent CWEs account for roughly 19–27% of all functions. In SARD, the top three CWEs represent about 45% of the dataset, with nearly 28% of these functions labeled as vulnerable. Overall, SARD emerges as the largest and least diverse dataset but provides a comparatively better balance between vulnerable and non-vulnerable samples.

Table 1 Datasets

Dataset	Funct.	Vuln.Funct.%	CWEs	Funct. per CWE %	Textual sim.%
				avg (std)	avg (std)
CrossVul	2642	12.1	44	2.27 (2.31)	3.03 (5.6)
CVEfixes	5909	12.7	58	2.26 (2.84)	2.5 (4.7
ProjectKB	20158	12.2	64	1.53 (12.34)	2.4 (9.2)
SARD	115600	29.5	21	4.76 (5.08)	6.6 (8.9)

Table 2 Datasets: most frequent CWEs

CrossVul			CVEfixes		
CWE	Funct.%	Vuln.Funct.%	CWE	Funct.%	Vuln.Funct.%
20	6.74	10.11	918	9.66	4.9
732	6.47	6.43	91	8.36	7.29
502	5.94	3.82	22	7.67	11.7
Top3	19.15	5.34	Top3	25.69	7.71
ProjecKB			SARD		
CWE	Funct.%	Vuln.Funct.%	CWE	Funct.%	Vuln.Funct.%
20	11.99	19.2	190	18.8	28.02
22	8.89	7.03	191	15.09	26.76
502	6.04	9.78	129	11.19	26.76
Top3	26.92	13.09	Top3	45.07	27.67

Process

Figure 2 presents a comprehensive overview of our methodology's workflow. Initially, we preprocess the raw datasets to derive function-level representations. Subsequently, we apply balancing techniques, such as RUS and OverUnder, to construct the balanced version of each dataset of the experiment. Models are trained with the original imbalanced and the new balanced datasets, as well as they are trained with both their standard and focal loss configurations. In the final step, we assess performance through evaluation metrics, allowing us to measure the impacts of both data balancing and the redefinition of loss functions. Our experiments is conducted on each dataset following this outlined procedure:

1. We randomly shuffled and split each dataset's Java functions into training (80%), validation (10%), and test (10%) subsets. This original version is referred to as the *Imbalanced dataset*.
2. We trained four vulnerability detection models **SVDet**, **JLineVD**, **IVDetect**, and **LineVD** on the imbalanced training set. For each model, we use parameters as presented in their corresponding paper without extra ad-hoc fine-tuning. This implies keeping the same epoch,

[8] https://samate.nist.gov/SRD/testsuite.php.

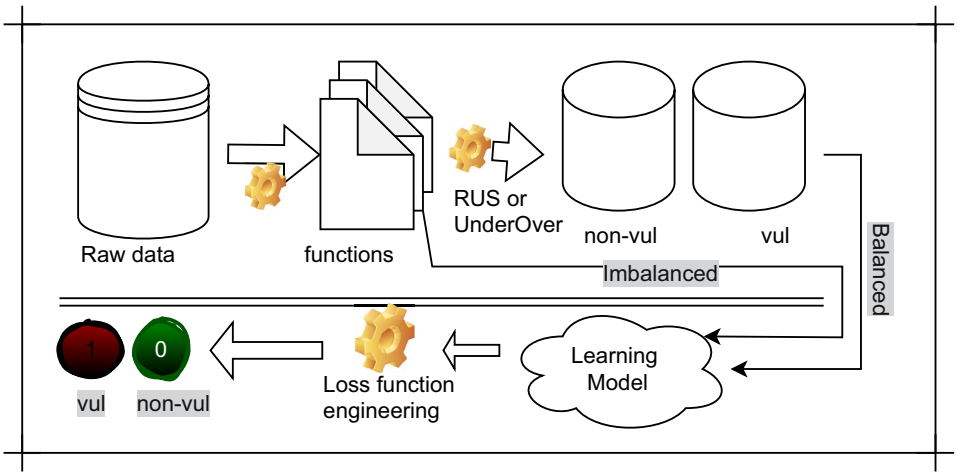

Fig. 2 Experimental overview of the methodology

Table 3 Resampling-Model training sets

Resampling	Functions in Training				Avg. CWE Imbalance (%)			
	CrossVul	CVEfixes	ProjecKB	SARD	CrossVul	CVEfixes	ProjecKB	SARD
Imbalanced	2114	3601	16126	92484	74.13	107.15	71.70	82.91
RUS	526	943	4008	54608	19.58	27.29	18.52	48.89
UnderOver	3702	6470	28244	130360	131.37	187.01	126.26	116.73

optimizer, vector length, batch size, and vocabulary size for all datasets an configuration.

3. Each model is evaluated on the test set using standard performance metrics.
4. To analyze the impact of resampling, we applied *RUS* (undersampling) and *UnderOver* (combination of under and over-sampling) strategies to the imbalanced training set, resulting in two balanced variants. These were used to retrain the models, while validation and test sets remained unchanged to prevent overfitting. Table 3 shows the number of training functions and average CWE imbalance per dataset and strategy. While balancing improved CWE distribution, especially with undersampling, some datasets still exhibited class scarcity, which could limit generalization.
5. We compared the classification performance across the three training setups: Imbalanced, RUS, and UnderOver.
6. To study the effect of imbalance-aware loss functions, we replaced the standard BCE loss or CCE loss in models with *Binary Class-Balanced Focal Loss* (*BCBFocalLoss*), denoting this variant as **Model** $_{CB}$. We used parameters $\beta = 2$ and $\alpha = 0.25$. A non-zero β emphasizes hard-to-classify samples, while $\alpha < 0.5$ prioritizes the minority class.
7. We compared the performance of each model using standard BCE or CCE loss against its BCBFocalLoss variant on the imbalanced dataset.

All experiments are repeated three times to ensure consistency. Experimentation in different settings is conducted on a local server equipped with an AMD RyzenTM 9 7950X 16 Core (CPU), 128 GB DDR5 (RAM), NVIDIA GeForce RTX 4090 24GB GDDR6X (GPU).

Experimental Results

RQ1: Balanced Data and Model Performance

To answer this research question, we analyze how the resampling methods, including Imbalanced, RUS (random undersampling), and UnderOver (combined undersampling and oversampling), affect the performance of learning-based vulnerability detection models. The results for sequence-based models (SVDet and LineVul) are reported in Table 4, while those for graph-based models (IVDetect and JLineVD) are shown in Table 5.

Sequence-based models: For both SVDet and LineVul, training on the original imbalanced datasets already yields relatively stable performance compared to graph-based approaches. Nevertheless, class imbalance still negatively affects minority-class detection. On average, SVDet trained on imbalanced data achieves a recall of 0.767 and an MCC of 0.719, while LineVul reaches a recall of 0.673 and an MCC of 0.628, indicating residual bias toward the majority class.

Applying undersampling (RUS) consistently increases recall for both models. For SVDet, recall rises from 0.790 to 0.816, corresponding to an increase of approximately 3.3%, while LineVul's recall increases from 0.673 to 0.718,

Table 4 Sequence-based models: Resampling results

Dataset	Resampling	SVDet				LineVul			
		Prec	F1	Recall	MCC	Prec	F1	Recall	MCC
CrossVul	Imbalanced	**0.803**	**0.763**	0.734	**0.767**	0.357	0.444	**0.588**	0.397
	RUS	0.596	0.563	0.748	0.654	0.258	0.333	0.470	0.271
	OverUnder	0.734	0.745	**0.757**	0.745	**0.692**	**0.6**	0.529	**0.576**
CVEfixes	Imbalanced	0.782	0.726	0.694	0.734	**0.742**	0.536	0.419	0.512
	RUS	0.638	0.646	**0.759**	0.689	0.495	0.553	0.609	0.481
	OverUnder	**0.785**	**0.742**	0.715	**0.747**	0.565	**0.595**	**0.629**	**0.531**
ProjectKB	Imbalanced	0.832	0.784	0.752	0.579	0.556	0.636	0.743	0.592
	RUS	0.644	0.663	0.773	0.396	0.455	0.581	**0.803**	0.542
	OverUnder	**0.861**	**0.839**	**0.821**	**0.681**	**0.828**	**0.744**	0.678	**0.722**
SARD	Imbalanced	**0.91**	0.899	0.889	0.899	**0.782**	**0.878**	**1**	0.832
	RUS	0.891	**0.908**	**0.941**	0.915	0.781	0.877	0.993	0.831
	OverUnder	0.891	**0.908**	**0.941**	**0.916**	0.780	0.876	0.999	**0.859**
all (avg)	Imbalanced	**0.831**	0.793	0.767	0.719	**0.693**	0.663	0.673	**0.628**
	RUS	0.692	0.695	0.805	0.663	0.497	0.586	0.718	0.531
	OverUnder	0.817	**0.808**	**0.808**	**0.772**	0.633	**0.665**	**0.724**	0.627

The bold values indicated the best metrics compared to other detection methods

Table 5 Graph-Based models: Resampling results

Dataset	Resampling	IVDetect				JLineVD			
		Prec	F1	Recall	MCC	Prec	F1	Recall	MCC
CrossVul	Imbalanced	**0.892**	0.458	0.509	0.124	0.606	0.627	0.679	0.276
	RUS	0.622	0.622	**0.669**	**0.288**	**0.911**	**0.965**	**0.994**	**0.932**
	OverUnder	0.625	**0.632**	0.647	0.272	0.884	0.930	0.992	0.870
CVEfixes	Imbalanced	0.444	0.470	0.500	0.000	0.933	0.708	0.647	0.506
	RUS	0.620	0.630	**0.747**	0.345	0.881	0.922	0.916	0.889
	OverUnder	**0.720**	**0.726**	0.733	**0.454**	**0.96**	**0.982**	**0.996**	**0.905**
ProjectKB	Imbalanced	0.438	0.466	0.500	0.000	0.519	0.521	0.528	0.047
	RUS	0.635	0.653	**0.734**	0.355	0.925	0.948	0.950	0.878
	OverUnder	**0.724**	**0.719**	0.7153	**0.439**	**0.930**	**0.953**	**0.979**	**0.909**
SARD	Imbalanced	0.352	0.413	0.500	0.000	-	-	-	-
	RUS	**0.795**	**0.806**	**0.834**	**0.627**	-	-	-	-
	OverUnder	0.771	0.775	0.821	0.590	-	-	-	-
all (avg)	Imbalanced	0.532	0.452	0.502	0.031	0.686	0.619	0.618	0.276
	RUS	0.668	0.678	**0.746**	0.404	0.906	0.945	0.953	**0.900**
	OverUnder	**0.710**	**0.713**	0.729	**0.439**	**0.925**	**0.955**	**0.989**	0.895

The bold values indicated the best metrics compared to other detection methods

representing a 6.7% improvement. However, this gain in sensitivity is accompanied by a notable reduction in precision. For LineVul, precision drops from 0.693 to 0.497, a decrease of about 28.3%, which limits the overall improvement in MCC. As a result, MCC improves only moderately or remains comparable to the imbalanced setting.

The UnderOver method provides the best overall balance between precision and recall for sequence-based models. Across all datasets, UnderOver either matches or outperforms the imbalanced setup in terms of F1-score and MCC. For SVDet, both recall and MCC increase, from 0.767 to 0.808 (+5.3%), and from 0.719 to 0.777 (+7.3%) respectively. For LineVul, UnderOver improves recall from 0.673 to 0.724, corresponding to a 7.6% increase, while maintaining an MCC comparable to the imbalanced configuration. At the dataset level, the most pronounced improvement is observed on ProjectKB, where LineVul's MCC increases from 0.592 to 0.722, yielding a relative improvement of 21.9%. These results confirm that preserving majority-class diversity while reinforcing minority-class representation leads to more robust performance.

Graph-based models: Graph-based models are substantially more affected by class imbalance. When trained on imbalanced data, IVDetect exhibits near-random behavior across most datasets, with an average recall of approximately 0.502 and an MCC of only 0.031. This corresponds to a performance level barely above random guessing and indicates a strong bias toward the majority class.

Resampling strategies improve graph-based model performance. With undersampling, IVDetect's recall increases

Table 6 Sequence-based model: Loss function results

Dataset	Resampling	SVDet				LineVul			
		Prec	F1	Recall	MCC	Prec	F1	Recall	MCC
CrossVul	BCELoss	0.803	**0.763**	**0.734**	**0.767**	0.357	0.444	**0.588**	**0.397**
	BCBFocalLoss	**0.859**	0.687	0.638	0.723	**0.361**	**0.447**	**0.588**	0.378
CVEfixes	BCELoss	0.782	**0.726**	**0.694**	**0.734**	**0.742**	0.536	0.419	0.512
	BCBFocalLoss	**0.828**	0.706	0.662	0.731	0.565	**0.595**	**0.629**	**0.531**
ProjectKB	BCELoss	0.832	0.784	0.752	0.579	**0.556**	**0.636**	0.743	0.592
	BCBFocalLoss	**0.929**	**0.8366**	**0.783**	**0.697**	**0.556**	**0.636**	**0.745**	**0.60**
SARD	BCELoss	0.91	**0.899**	**0.889**	0.889	**0.782**	**0.878**	**1**	**0.832**
	BCBFocalLoss	**0.948**	0.892	0.861	**0.902**	0.780	0.876	0.998	0.829
all (avg)	BCELoss	0.831	**0.793**	**0.767**	0.719	**0.609**	0.625	0.687	0.583
	BCBFocalLoss	**0.891**	0.78	0.736	**0.788**	0.565	**0.638**	**0.74**	**0.584**

The bold values indicated the best metrics compared to other detection methods

Table 7 Graph-based model: Loss function results

Dataset	Resampling	IVDetect				JLineVD			
		Prec	F1	Recall	MCC	Prec	F1	Recall	MCC
CrossVul	BCELoss	**0.892**	**0.458**	**0.509**	**0.124**	0.606	**0.627**	**0.679**	**0.276**
	BCBFocalLoss	0.390	0.438	0.500	0.000	**0.609**	0.567	0.553	0.152
CVEfixes	BCELoss	**0.444**	**0.470**	**0.500**	0.000	**0.933**	0.708	0.647	0.506
	BCBFocalLoss	**0.444**	**0.470**	**0.500**	0.000	0.917	**0.741**	**0.678**	**0.546**
ProjectKB	BCELoss	**0.438**	**0.466**	**0.500**	0.000	0.519	0.521	0.528	0.047
	BCBFocalLoss	**0.438**	**0.466**	**0.500**	0.000	**0.659**	**0.624**	**0.605**	**0.259**
SARD	BCELoss	0.352	0.413	0.500	0.000	-	-	-	-
	BCBFocalLoss	**0.863**	**0.827**	**0.806**	**0.668**	-	-	-	-
all (avg)	BCELoss	0.532	0.452	0.502	0.031	0.686	0.619	**0.618**	0.276
	BCBFocalLoss	**0.534**	**0.550**	**0.577**	**0.167**	**0.728**	**0.644**	0.612	**0.319**

The bold values indicated the best metrics compared to other detection methods

from 0.502 to 0.746, representing a 48.6% improvement, while MCC rises from 0.031 to 0.404, corresponding to an increase of more than 1200%. The UnderOver strategy further improves MCC to 0.439, yielding an overall improvement of approximately 1316% compared to the imbalanced setting. These results demonstrate that reducing majority-class dominance is essential for effective graph-based learning.

JLineVD also benefits from resampling. Across CrossVul, CVEfixes, and ProjectKB, training on imbalanced data yields an average MCC of 0.276. When undersampling is applied, MCC increases to 0.900, corresponding to a relative improvement of 226%, while UnderOver achieves a comparable MCC of 0.895 (+224%). Recall also increases markedly, from 0.618 in the imbalanced setting to 0.953 with undersampling, representing a 54.2% improvement. These results highlight the strong dependence of JLineVD on balanced training data.

JLineVD results are not reported for the SARD dataset, as it requires both vulnerable and corresponding fixed versions of the code to generate the training metadata. Since SARD is synthetic data and only provides a single code version, JLineVD could not be trained on this dataset.

RQ1: The results show that resampling methodologies are crucial for mitigating the negative effects of class imbalance. While sequence-based models exhibit moderate gains, with MCC improvements reaching up to 22% in some datasets, graph-based models achieve improvements exceeding 1000% in MCC. Among the evaluated strategies, UnderOver sampling consistently provides the best balance between recall, precision, and MCC, particularly for graph-based architectures.

RQ2: Refined Loss Function and Model Performance

This research question investigates whether refining the loss function can mitigate the effects of class imbalance in vulnerability detection models. To this end, we substitute the standard Binary Cross-Entropy loss with a Binary Class-Balanced Focal Loss (BCBFocalLoss) and analyze its impact on both sequence-based and graph-based models. The experimental results for sequence-based models are reported in Table 6, while the corresponding results for graph-based models are presented in Table 7. All experiments in this section are conducted on the original

imbalanced datasets in order to isolate the effect of loss refinement from data-level balancing strategies.

Sequence-based Models: For sequence-based models, the refined loss function primarily affects the trade-off between precision and recall. In the case of SVDet, the use of BCBFocalLoss leads to an increase in precision from 0.831 to 0.891, corresponding to a relative improvement of 7.1%. At the same time, recall decreases from 0.767 to 0.736, representing a reduction of 4%. As a result, the overall MCC increases from 0.719 to 0.788 (9.5%), indicating that the refined loss encourages more conservative predictions without improving global classification balance.

For LineVul, the effect of loss refinement is more favorable in terms of sensitivity to vulnerable code. Recall increases from 0.687 to 0.740, yielding a relative improvement of 7.7%, while precision remains nearly unchanged. This leads to a marginal improvement in MCC of approximately 0.2%. Although the gains are modest, these results suggest that focal loss can help sequence-based models better attend to hard-to-classify vulnerable samples, particularly when the baseline recall is low.

Overall, for sequence-based architectures, the refined loss function provides incremental improvements in specific metrics, mainly recall or precision depending on the model, but does not consistently outperform the standard loss across all evaluation criteria.

441

— 41 121214 21 0

Graph-based Models: For graph-based models, the impact of loss refinement is strongly dependent on the severity of class imbalance. In highly imbalanced datasets, IVDetect continues to exhibit limited performance despite the use of BCBFocalLoss. Although recall improves moderately, MCC remains low in most cases, indicating that loss reweighting alone is insufficient to overcome the dominance of the majority class in graph-based representations.

A notable exception is observed on the SARD dataset, where class imbalance is less pronounced. In this setting, IVDetect benefits substantially from the refined loss function, with MCC increasing from 0.000 to 0.668. This result demonstrates that focal loss can be effective when the imbalance ratio remains within a manageable range.

For JLineVD, the refined loss function produces small but consistent improvements across the datasets where results are available. MCC increases from 0.276 to 0.319, corresponding to a relative improvement of 15.6%. Despite these gains, the performance remains lower than that achieved through resampling strategies, highlighting the limitations of loss refinement when used in isolation.

RQ2: Refining the loss function through class-balanced focal loss leads to moderate and model-dependent improvements. While it can enhance precision or recall in certain scenarios, particularly for sequence-based models and moderately imbalanced datasets, it does not fully address the challenges posed by severe class imbalance. These findings suggest that loss refinement should be viewed as a complementary technique rather than a standalone solution, especially for graph-based vulnerability detection models.

RQ3: Performance of LLM-Based Detectors Compared to Sequence and Graph Models

Given the substantial training data and computational resources required to fine-tune LLMs for vulnerability detection, we adopt a recent methodology (described in Section "LLM-Based Approaches") to evaluate LLMs in a resource-efficient manner. Specifically, we evaluate two prompting methodologies on the ProjectKB and Crosvul dataset: (i) a simple zero-shot prompt, (ii) a Retrieval-Augmented Generation method, where relevant contextual examples are retrieved and provided to the model before inference, (iii) and fine-tuning on different balanced setting,. The results are reported in Table 8.

The results show that incorporating retrieval-based context consistently improves LLM performance across all evaluation metrics. For LLaMA, the RAG method leads to a substantial increase in MCC (from 0.203 to 0.614), indicating a better balance between precision and recall. A similar trend is observed for Mistral, where RAG improves MCC from 0.252 to 0.364. These gains highlight the importance of contextual grounding when applying general-purpose LLMs to code vulnerability detection tasks. The various tuning settings applied to the original dataset, RUS, and OverUnder did not result in significant improvements; in many cases, there was even a degradation in performance. This observation highlights the inadequacy of training samples needed to adjust the parameters of both models, which consist of over 3.4 million parameters. As a result, the model tends to lose or misinterpret its initial understanding of source code assessment.

We further compare the best-performing LLM configurations with state-of-the-art sequence- and graph-based models evaluated on the same datasets (last rows of Table 8). Despite the improvements introduced by RAG, LLM-based approaches remain inferior to specialized models. In particular, graph-based models such as JLineVD and transformer-based models such as SVDet achieve a higher MCC score. For example, JLineVD attains an MCC of 0.909, surpassing

Table 8 Comparison of the performance of LLMs on the ProjectKb and Crosvul datasets

Model	Method	ProjectKB				Crosvul			
		Prec	F1	Recall	MCC	Prec	F1	Recall	MCC
LLaMA	Simple	0.565	0.519	0.658	0.203	0.487	0.481	0.510	−0.016
	RAG	**0.697**	**0.644**	**0.664**	**0.614**	**0.587**	**0.553**	**0.571**	**0.652**
	Tune_Imbalanced	0.442	0.469	0.501	0.000	0.428	0.461	0.500	0.000
	Tune_RUS	0.238	0.323	0.500	0.000	0.241	0.325	0.500	0.000
	Tune_OverUnder	0.122	0.208	0.725	0.153	0.242	0.326	0.500	0.000
Mistral	Simple	0.534	0.384	0.365	0.252	0.526	**0.528**	**0.529**	0.057
	RAG	**0.648**	**0.622**	**0.607**	**0.364**	**0.577**	0.482	0.501	**0.549**
	Tune_Imbalanced	0.442	0.469	0.500	0.000	440.4	0.467	0.500	0.000
	Tune_RUS	0.238	0.323	0.500	0.000	0.233	0.318	0.501	0.000
	Tune_OverUnder	0.323	0.425	0.619	0.352	0.241	0.325	0.500	0.000
IVDetect	OverUnder	0.724	0.719	0.715	0.439	0.625	0.632	0.647	0.272
JLineVD	OverUnder	**0.930**	**0.953**	**0.979**	**0.909**	**0.884**	**0.930**	**0.992**	**0.870**
SVDet	OverUnder	0.861	0.839	0.821	0.681	0.734	0.745	0.757	0.745
LineVul	OverUnder	0.828	0.744	0.678	0.722	0.692	0.60	0.529	0.576

The bold values indicated the best metrics compared to other detection methods

LLaMA_{RAG} (0.614), while SVDet also exhibits a strong advantage with an MCC of 0.852.

This performance gap suggests that models explicitly designed to capture code structure are significantly more effective than LLMs for vulnerability detection tasks. While LLMs demonstrate promising adaptability through advanced prompting methodologies, they currently lack the task-specific representations learned by fine-tuned sequence and graph-based models. These findings indicate that, although LLMs constitute a complementary approach, further task-specific adaptation or fine-tuning with larger datasets is required for them to close the performance gap with dedicated vulnerability detection models.

RQ3: Retrieval-Augmented Generation significantly enhances the effectiveness of zero-shot LLM-based vulnerability detection; however, LLMs such as LLaMA and Mistral still do not outperform state-of-the-art graph-based (e.g., JLineVD) or transformer-based models (e.g., SVDet, LineVul) on the ProjectKB dataset. The superior MCC scores achieved by these specialized models indicate that task-specific architectures and fine-tuning remain crucial for accurate vulnerability detection.

Discussion

The experimental results show that data balancing methodologies significantly improve the performance of AI-based vulnerability detection models. While the RUS (undersampling) enhances some models, it remains limited because it substantially reduces the number of training samples in highly imbalanced datasets and may remove important code context. Moreover, RUS restricts model training across different dataset structures compared to the combined OverUnder methodology, which better preserves data diversity while avoiding excessive redundancy.

The OverUnder method achieves the most consistent improvements across datasets. It increases training diversity and evaluation robustness but introduces (a limited amount of) duplicated samples through oversampling, which may lead to overfitting and reduced generalization to unseen data. On highly imbalanced real-world datasets, such as Project KB and CVEFixes, OverUnder consistently outperforms pure RUS and the original imbalanced setting. In contrast, on synthetic datasets such as SARD, where sufficient training samples remain after undersampling, the performance gap is reduced. These results highlight that sample diversity is more critical than training set size, and that unique samples are preferable to larger datasets with a large number of duplicated instances. Hence, in this respect, real-world datasets seem to be more adequate for training purpose.

Similar trends are observed for loss function design. Focal loss improves performance in several settings for both sequence-based and graph-based models. For example, it increases MCC on SARD for SVDet and recall on CVEFixes for LineVul. However, performance degradation is observed in some cases, such as on CrossVul with JLineVD, and models highly sensitive to imbalance, such as IVDetect, show limited benefit. This indicates that focal loss effectiveness depends on both model architecture and the degree of dataset imbalance, with more consistent gains on synthetic datasets.

To better evaluate different models' performance, we analyze the F1-score of each model across different

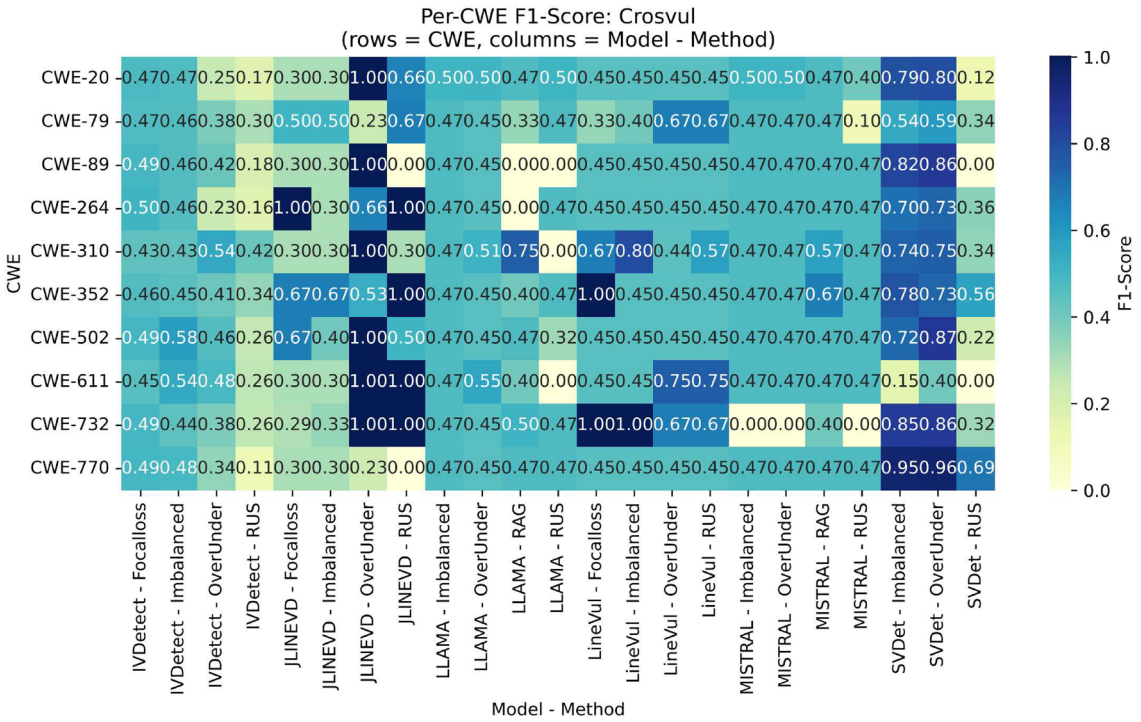

Fig. 3 F1-score comparison across the top 10 CWE categories on the CrossVul dataset under different training settings and model architectures

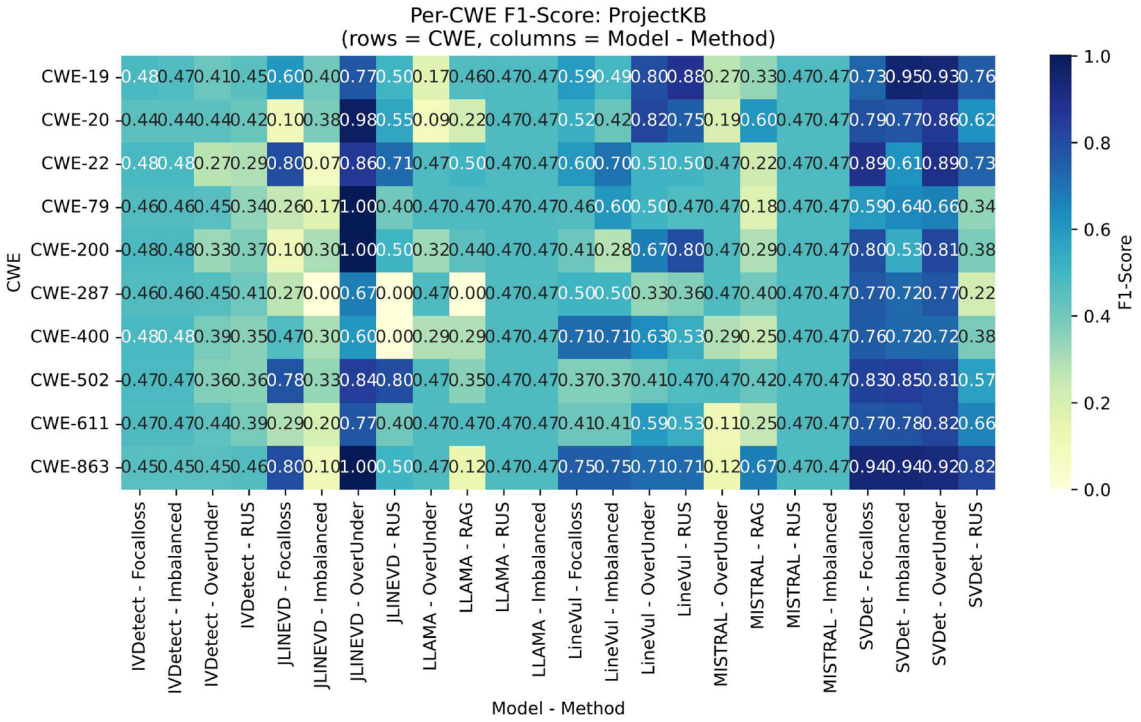

Fig. 4 F1-score comparison across the top 10 CWE categories on the ProjectKB dataset under different training settings and model architectures

settings for the top 10 representative CWEs in two datasets, CrossVul and ProjectKB. We focus on CrossVul and ProjectKB as they exhibit distinct characteristics (see 1) and were applied to all the evaluated models. For instance, the former is a small dataset composed of diverse code, while the latter is larger and strongly imbalanced, and it also contains a larger diversity of code. Figure 3 summarizes the results for CrossVul, while Fig. 4 presents the results for

ProjectKB. We observe that sequence-based models generally achieve more stable performance across these top 10 CWEs. For instance, we observe an increase of the performance of JLineVD in most of the cases (82.5% of the classified top 10 CWEs) in which we balance the datasets (by using RUS or OverUnder), while SVDet increases only in a limited number of cases (20% of CWEs). However, there are cases where sequence-based models underperform compared to graph-based models, and vice versa. For instance, this is observed for CWE-611 and CWE-770 in the CrossVul dataset when comparing SVDet and JLineVD. By analyzing samples of vulnerable code, we can see that this is mainly due to the type of CWE, in fact, CWE-611 is related to a wrong configuration in the code, while CWE-770 is due to a wrong implementation of the logical flow control. However, we can notice that concerning this trend is not confirmed in ProjectKB, where sequence-based models tend to outperform graph-based models for CWE-611. Furthermore, the stability of some models, such as LLM-based approaches and IVDetect, remains a concern in this analysis. Since we use macro-average, F1-scores close to or below 50% often indicate low recall (i.e., recall $\simeq 50\%$), reflecting the model's inability to correctly identify positive (vulnerable) samples. Additionally, although sequence-based and graph-based models exhibit complementary behavior, the OverUnder results suggest that graph-based models tend to outperform sequence-based models when the training data is sufficiently representative and balanced across classes. We then attribute this difference to the model architectures: sequence-based models rely on token-level patterns, whereas graph-based models depend on structural relationships that require sufficient and balanced samples to be effectively learned.

In comparison with LLM-based approaches, our results indicate that while LLMs can achieve competitive performance in some cases, they are not yet consistently comparable to task-specific models. However, more robust tuning and training strategies may improve their effectiveness for source code analysis tasks. Moreover, our study suggests that, from an architectural perspective, sequence-based models are more suitable when training data is highly imbalanced, whereas graph-based models are more effective under balanced conditions. Although the analyses presented in this paper highlight the complementary strengths of different models across CWEs, LLM-based approaches might be more suitable when large-scale training data is available, given their significantly higher number of parameters compared to specialized deep learning models.

Threats to Validity

This study is subject to some validity threats. The analysis is limited to six learning-based models: two transformer-based, two graph-based approaches, and two LLMs. While these are widely adopted and representative of current research trends, other architectures might yield different results.

External Validity. The evaluation relied on four Java datasets, including real-world and synthetic code. Although these data sets are commonly used in prior studies, the findings may not be generalized to other programming languages. Future work will extend the evaluation to a broader range of datasets and languages. In addition, LLMs may have been exposed to parts of publicly available datasets during their pre-training process. This potential data contamination could influence the reported performance if the models have previously seen similar or identical samples.

Internal Validity. The experimental setup, including partitioning the data set and hyperparameter configurations, may have also influenced the results. Alternative setups could lead to different conclusions, highlighting the need for more extensive experiments. To mitigate this, we followed common practices used in prior studies and applied consistent configurations across models whenever possible. However, parameter tuning or different data splits could still affect the obtained results.

Construct Validity. The imbalance handling strategies used represent another limitation. We relied on widely used resampling techniques, but more advanced methods, such as adaptive resampling or sophisticated loss functions, may further improve performance. Investigating these approaches is left as future work.

Related Work

Several works investigate approaches and techniques for detecting and localizing software bugs [52]. Some of such works adopt methods based on machine learning, deep-learning, and LLMs. For instance, [53] presents a tool that applies statistical causal inference and random forests to localize bugs in software. Line-DP [54] combines logistic regression and a model-agnostic explainability method to identify the defective statements, i.e., statements that contain tokens leading to bugs. [55] empirically demonstrated that LLMs can support developer in localizing software bugs.

In this work, we focus on specific bug types that lead to security issues: security vulnerabilities. Existing approaches use machine and deep-learning methods, as well as LLMs, to identify vulnerable code [56, 57]. Deep learning and

Table 9 Overview of related work (n/a: not applicable)

References	Dataset	Lang.	Sampling	Loss	Model	Gran.	Metrics
[47]	SARD, NVD	C/C++	No	n/a	Sequence	Function	n/a
[7]	SARD, FFmpeg, LibPNG	C	No	CE	Sequence	Function	Top-k
[23]	Big-Vul	C/C++	No	CE	Transformer	Statement	n/a
[32]	Big-Vul	C/C++	Under	CE	Graph	Statement	n/a
[8]	SARD	Java	No	CE	Transformer	Function	n/a
[9]	SARD, NVD	C/C++	Under	n/a	Graph	Statement	n/a
[48]	ReVEAL, FFMpeg, D2A, MVD	C/C++	No	FL+CBFL	Seq.+Graph	Function	n/a
[49]	CodeXGLUE	C	No/Wtd	CE/WCE	Seq.+Graph	Function	n/a
[19]	NVD, FFmpeg, LibPNG, LibTIFF	C/C++	Under+Over	n/a	Graph	Function	n/a
[20]	SARD, FUNDED, Vul4j, CVEfixes, CXG	Java	Balance	CE	Seq.+Graph	Function	n/a
[22]	ProjectKB	Java	No	CE	Transformer	Statement	Macro
[10]	ProjectKB	Java	No	CE	Graph	Statement	n/a
[11]	CVEfixes, Vudenc	Python	No	WCE	Graph	Statement	Macro
[50]	FFmpeg,libtiff, libpng	C/C++	Over	CE	Sequence +ML	Function	n/a
[51]	ProjectKB, MegaVul, Big-Vul, CVEFixes	Java, C/C++, Python	Under	CE	Graph+ Transformer	Statement	n/a

LLMs have been increasingly explored for vulnerability detection in source code, for instance, [25–28].

Table 9 summarizes a representative set of studies by focusing on datasets, programming languages, modeling approaches, and evaluation practices they adopt. This overview is not intended to be exhaustive; it contains representative works. Other existing surveys on vulnerability detection methods and tools can complement the table by providing further information in the field [4, 6, 25].

Overall, from Table 9, it is evident that most contributions focus on C/C++ datasets, while Java and Python are less frequently studied. A further observation is that class imbalance has often been neglected. The majority of works adopt standard cross-entropy loss without explicitly addressing imbalance. For instance, [10, 11, 22, 40] proposed novel tools and an evaluation process, but did not explicitly address the effects of limited samples in the vulnerable class within the datasets used for model training. Fu et al. [23] presented LineVul, a transformer-based approach to detect vulnerabilities in C/C++ source code. In particular, LineVul uses CodeBERT to identify the lines of code that lead to a vulnerability, thus reducing the manual effort required by developers to locate and fix bugs. In both approach and evaluation, they do not explicitly consider the impact of the imbalanced datasets.

Only a few attempts introduce countermeasures, such as resampling or imbalance-aware loss functions (e.g., Weighted Cross-Entropy, Focal Loss, and Class-Balanced Focal Loss). For example, Although Kaya et al. [58] evaluated rebalancing techniques involving seven traditional machine learning models, such as Random Forest and Support Vector Machine. However, they fail to incorporate newer advanced models like transformers (e.g., used by approaches such as SVdet and LineVul) and LLMs. Liu et al. [50] presented DeepBalance that utilizes a bidirectional long short-term memory network to learn invariants and discriminative code representations. They applied a fuzzy oversampling method to rebalance the training data by generating synthetic samples for the vulnerable code class. However, they rely on unvalidated synthetic code, which may compromise data integrity, and do not systematically evaluate different model architectures. Foulefack et al. [51] conducted an in-depth study on employing domain knowledge to enhance the graph-based model for vulnerability detection, their exploration of model parameter searches highlights the impact of imbalanced datasets on model performance, even if they randomly down-sampled the non-vulnerable training code. Tang et al. [49] combined resampling with weighted cross entropy loss function (WCE), reporting improvements in recall at the cost of lower precision. Guo et al. [59] empirically identified the problems of imbalanced datasets in software vulnerability detection. However, they considers two state-of-the-art models not specialized for vulnerability detection (CodeBERT [30] and CodeGraphBERT [60] and a set of relatively old C/C++ projects as dataset for training and testing the models (e.g., FFmpeg, QEMU), which limits the generalizability to the broader range of models recently developed for static source code vulnerability detection.

In terms of evaluation, most studies report metrics such as Precision, Recall, F1, or AUROC. However, the explicit use of macro or weighted variants of these metrics is rarely discussed, despite their relevance in imbalanced settings.

To the best of our knowledge, no prior work has conducted a systematic investigation of both resampling and imbalance-aware loss functions in the context of vulnerability

detection. Our work contributes to this gap by providing a structured evaluation of these techniques and analyzing their impact on detection performance, thus extending our prior analysis [21].

Conclusion

This paper presents a systematic analysis of data-driven methods for source-code vulnerability detection under different experimental settings, with a particular focus on the imbalanced nature of existing datasets. The adoption of various resampling methodologies demonstrates performance gains across state-of-the-art models. However, the results also indicate that, while resampling techniques and loss-function refinements can improve model performance, their effectiveness remains limited under certain conditions, especially for approaches that are highly sensitive to data imbalance, such as IVDetect. Furthermore, the comparison with LLMs, including LLaMA and Mistral, shows that although these models are designed to address a wide range of software-related tasks, such as code generation and understanding, they are not yet competitive with specialized sequence-based and graph-based models for vulnerability detection.

Future work will explore alternative methodologies to better address dataset imbalance, including the integration of additional handcrafted domain knowledge and enriched structural information to further improve detection robustness and generalization.

Acknowledgements The work was partially supported by the European Union Horizon Europe Research and Innovation Programme under Grant 101120393 (Sec4Ai4Sec) and partially by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU.

Author Contributions All authors contributed equally to this work.

Funding Open access funding provided by Università degli Studi di Trento within the CRUI-CARE Agreement.

Data Availability The datasets used in this study are publicly available.

Declarations

Conflict of interest The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Ethical Approval and Consent to Participate Not applicable.

Consent for Publication Not applicable.

References

1. Abdel Hakeem SA, Hussein HH, Kim H. Security requirements and challenges of 6G technologies and applications. Sensors. 2022;22(5):1969.
2. Ha T, Dang TK, Le H, Truong TA. Security and privacy issues in deep learning: a brief review. SN Comput Sci. 2020;1(5):253.
3. Chukwuani EN, Odunsi OR, Ikemefuna CD. Machine learning techniques for real-time malware classification and threat detection in distributed systems. World J Adv Res Rev. 2025;26(3):2378–98.
4. Wu B, Zou F. Code vulnerability detection based on deep sequence and graph models: a survey. Secur Commun Netw. 2022;2022(1):1176898.
5. Zhou X, Zhang T, Lo D. Large language model for vulnerability detection: Emerging results and future directions. In: Proceedings of the 2024 ACM/IEEE 44th International Conference on software engineering: new ideas and emerging results; 2024; p. 47–51.
6. Lekeufack Foulefack RZ, Marchetto A. A rapid review on graph-based learning vulnerability detection. In: International Conference on the quality of information and communications technology. Springer; 2024. p. 355–372.
7. Lin G, Zhang J, Luo W, Pan L, De Vel O, Montague P, et al. Software vulnerability discovery via learning multi-domain knowledge bases. IEEE Trans Depend Secure Comput. 2019;18(5):2469–85.
8. Mamede C, Pinconschi E, Abreu R. A transformer-based ide plugin for vulnerability detection. In: Proceedings of the 37th IEEE/ACM International Conference on automated software engineering; 2022; p. 1–4.
9. Dong Y, Tang Y, Cheng X, Yang Y, Wang S. SedSVD: statement-level software vulnerability detection based on Relational Graph Convolutional Network with subgraph embedding. Inf Softw Technol. 2023;158:107168.
10. Lekeufack Foulefack RZ, Marchetto A. Enhanced graph neural networks for vulnerability detection in java via advanced subgraph construction. In: IFIP International Conference on testing software and systems. Springer. 2024; p. 131–148.
11. Tran HC, Tran AD, Le KH. DetectVul: a statement-level code vulnerability detection for Python. Futur Gener Comput Syst. 2025;163:107504.
12. Lin G, Zhang J, Luo W, Pan L, Xiang Y, De Vel O, et al. Cross-project transfer representation learning for vulnerable function discovery. IEEE Trans Ind Inf. 2018;14(7):3289–97.
13. Ponta SE, Plate H, Sabetta A, Bezzi M, Dangremont C. A manually-curated dataset of fixes to vulnerabilities of open-source software. In: 2019 IEEE/ACM 16th International Conference on mining software repositories (MSR). IEEE. 2019; p. 383–387.
14. Fan J, Li Y, Wang S, Nguyen TN. AC/C++ code vulnerability dataset with code changes and CVE summaries. In: Proceedings

of the 17th International Conference on mining software repositories. 2020, p. 508–512.
15. Nikitopoulos G, Dritsa K, Louridas P, Mitropoulos D. CrossVul: a cross-language vulnerability dataset with commit data. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2021; p. 1565–1569.
16. Chakraborty S, Krishna R, Ding Y, Ray B. Deep learning based vulnerability detection: are we there yet? IEEE Trans Softw Eng. 2021;48(9):3280–96.
17. Croft R, Babar MA, Kholoosi MM. Data quality for software vulnerability datasets. In: 2023 IEEE/ACM 45th International Conference on software engineering (ICSE). IEEE, 2023; p. 121–133.
18. Guo Y, Bettaieb S, Casino F. A comprehensive analysis on software vulnerability detection datasets: trends, challenges, and road ahead. Int J Inf Secur. 2024;23(5):3311–27.
19. Wang S, Huang C, Yu D, Chen X. VulGraB: graph-embedding-based code vulnerability detection with bi-directional gated graph neural network. Softw Pract Exp. 2023;53(8):1631–58.
20. Hussain S, Nadeem M, Baber J, Hamdi M, Rajab A, Al Reshan MS, et al. Vulnerability detection in Java source code using a quantum convolutional neural network with self-attentive pooling, deep sequence, and graph-based hybrid feature extraction. Sci Rep. 2024;14(1):7406.
21. Lekeufack Foulefack RZ, Marchetto A. On the use of imbalanced datasets for learning-based vulnerability detection. In: IFIP International Conference on testing software and systems. Springer. 2025; p. 307–324.
22. Marchetto A. Can explainability and deep-learning be used for localizing vulnerabilities in source code? In: Proceedings of the 5th ACM/IEEE International Conference on automation of software test (AST 2024). 2024; p. 110–119.
23. Fu M, Tantithamthavorn C. Linevul. A transformer-based line-level vulnerability prediction. In: Proceedings of the 19th International Conference on mining software repositories, 2022; p. 608–620.
24. Li Y, Wang S, Nguyen TN. 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; p. 292–303.
25. Yang Y, Li G. On the code vulnerability detection based on deep learning: a comparative study. IEEE Access. 2024;12:152377–152391.
26. Kaniewski S, Schmidt F, Enzweiler M, Menth M, Heer T. A systematic literature review on detecting software vulnerabilities with large language models. 2025. arXiv preprint arXiv:2507.22659.
27. Ullah S, Han M, Pujar S, Pearce H, Coskun A, Stringhini G, et al. In: IEEE Symposium on security and privacy (SP). IEEE. 2024;2024:862–80.
28. Li J, Cui L, Zhang J, Li L, Xi R, Zhu H. PVDetector: pretrained vulnerability detection on vulnerability-enriched code semantic graph. ACM Trans Softw Eng Methodol. 2025;35.
29. De Sousa NT, Hasselbring W. Javabert: Training a transformer-based model for the java programming language. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW). IEEE, 2021; p. 90–95.
30. Feng Z, Guo D, Tang D, Duan N, Feng X, Gong M, et al. Codebert: a pre-trained model for programming and natural languages. 2020. arXiv preprint arXiv:2002.08155.
31. Burkart N, Huber MF. A survey on the explainability of supervised machine learning. J Artif Intell Res. 2021;70:245–317.
32. Hin D, Kan A, Chen H, Babar MA. Linevd: statement-level vulnerability detection using graph neural networks. In: Proceedings of the 19th International Conference on mining software repositories, 2022; p. 596–607.
33. Grover A, Leskovec J. node2vec: scalable feature learning for networks. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge discovery and data mining, 2016; p. 855–864.
34. Ni C, Shen L, Yang X, Zhu Y, Wang S. MegaVul: AC/C++ vulnerability dataset with comprehensive code representations. In: Proceedings of the 21st International Conference on mining software repositories. 2024; p. 738–742.
35. Bhandari G, Naseer A, Moonen L. CVEfixes: automated collection of vulnerabilities and their fixes from open-source software. In: Proceedings of the 17th International Conference on predictive models and data analytics in software engineering. 2021; p. 30–39.
36. Leevy JL, Khoshgoftaar TM, Bauder RA, Seliya N. A survey on addressing high-class imbalance in big data. J Big Data. 2018;5(1):1–30.
37. Turan DS, Ordin B. The incremental SMOTE: a new approach based on the incremental k-means algorithm for solving imbalanced data set problem. Inf Sci. 2025;711:122103.
38. Mahmoodi N, Shirazi H, Fakhredanesh M, DadashtabarAhmadi K. Automatically weighted focal loss for imbalance learning. Neural Comput Appl. 2025;37(5):4035–52.
39. Chawla NV. Data mining for imbalanced datasets: an overview. In: Data mining and knowledge discovery handbook. 2010;875–886.
40. Lin Y, Li Y, Gu M, Sun H, Yue Q, Hu J, et al. Vulnerability dataset construction methods applied to vulnerability detection: A survey. In: 2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W). IEEE, 2022; p. 141–146.
41. Galdran A, Carneiro G, González Ballester MA. Balanced-mixup for highly imbalanced medical image classification. In: International Conference on medical image computing and computer-assisted intervention. Springer 2021; p. 323–333.
42. Ross TY, Dollár G, et al. Focal loss for dense object detection. In: proceedings of the IEEE Conference on computer vision and pattern recognition. IEEE, 2017; p. 2980–2988.
43. Ouchebara DS, Dupont S. Llama-based source code vulnerability detection: prompt engineering vs fine tuning. In: European Symposium on Research in Computer Security. Springer, 2025; p. 289–308.
44. Huang L, Yu W, Ma W, Zhong W, Feng Z, Wang H, et al. A survey on hallucination in large language models: principles, taxonomy, challenges, and open questions. ACM Trans Inform Syst. 2025;43(2):1–55.
45. Sheng Z, Wu F, Zuo X, Li C, Qiao Y, Lei H. Research on the LLM-driven vulnerability detection system using LProtector. In: 2024 IEEE 4th International Conference on Data Science and Computer Application (ICDSCA). IEEE, 2024; p. 192–196.
46. Yin X, Ni C, Wang S. Multitask-based evaluation of open-source llm on software vulnerability. IEEE Trans Softw Eng. 2024;50(11):3071–87.
47. Li Z, Zou D, Xu S, Ou X, Jin H, Wang S, et al. Vuldeepecker: a deep learning-based system for vulnerability detection. 2018. arXiv preprint arXiv:1801.01681.
48. Islam NT, Parra GDLT, Manuel D, Bou-Harb E, Najafirad P. An unbiased transformer source code learning with semantic vulnerability graph. In: IEEE 8th European Symposium on Security and Privacy (EuroS&P). IEEE. 2023;2023:144–59.
49. Tang W, Tang M, Ban M, Zhao Z, Feng M. CSGVD: a deep learning approach combining sequence and graph embedding for source code vulnerability detection. J Syst Softw. 2023;199:111623.
50. Liu S, Lin G, Han QL, Wen S, Zhang J, Xiang Y. DeepBalance: deep-learning and fuzzy oversampling for vulnerability detection. IEEE Trans Fuzzy Syst. 2019;28(7):1329–43.

51. Foulefack RZL, Marchetto A. Domain-aware graph neural networks for source code vulnerability detection. Inform Softw Technol. 2026;p. 108104.
52. Zou D, Liang J, Xiong Y, Ernst MD, Zhang L. An empirical study of fault localization families and their combinations. IEEE Trans Softw Eng. 2021;47(2):332–47. https://doi.org/10.1109/TSE.2019.2892102.
53. Podgurski A, Küçük Y. CounterFault: value-based fault localization by modeling and predicting counterfactual outcomes. In: 2020 IEEE International Conference on software maintenance and evolution (ICSME). 2020; p. 382–393.
54. Wattanakriengkrai S, Thongtanunam P, Tantithamthavorn C, Hata H, Matsumoto K. Predicting defective lines using a model-agnostic technique. IEEE Trans Softw Eng. 2022;48(5):1480–96.
55. Xu H, Liu H, Wu Y, Kang X, Chen X, Liu Y. Exploring the potential and limitations of large language models for novice program fault localization. J Syst Softw. 2026;234:112731.
56. Chakraborty S, Krishna R, Ding Y, Ray B. Deep learning based vulnerability detection: are we there yet? IEEE Trans Softw Eng. 2022;48(09):3280–96.
57. Shiri Harzevili N, Boaye Belle A, Wang J, Wang S, Jiang ZMJ, Nagappan N. A systematic literature review on automated software vulnerability detection using machine learning. ACM Comput Surv. 2025. https://doi.org/10.1145/3699711.
58. Kaya A, Keceli AS, Catal C, Tekinerdogan B. The impact of feature types, classifiers, and data balancing techniques on software vulnerability prediction models. J Softw Evol Process. 2019;31(9):e2164.
59. Guo Y, Hu Q, Tang Q, Traon YL. An empirical study of the imbalance issue in software vulnerability detection. In: European Symposium on Research in Computer Security. Springer 2023; p. 371–390.
60. Du Y, Yu Z. Pre-training Code Representation with Semantic Flow Graph for Effective Bug Localization. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. New York, NY, USA: Association for Computing Machinery 2023; p. 579–591. Available from: https://doi.org/10.1145/3611643.3616338.