

A Review of Source Code Vulnerability Detection Methods Based on Large Language Models: Models, Techniques, and Evaluation

Letong Qin*

School of Computer Science and Technology, Beijing Jiaotong University, Beijing 100044, China

**Corresponding author: Letong Qin.*

Abstract

In source code vulnerability detection, large language models (LLMs) are no longer used only as substitutes for traditional fixed classifiers. With prompt design, parameter fine-tuning, and program-context information, they can judge whether code contains vulnerabilities and can also provide suspicious locations and explanatory evidence. This paper takes lightweight code Transformers and large-scale language models as two main lines of discussion. It reviews methods for input construction, task adaptation, the use of program-analysis information, and multi-model collaboration, and compares them in terms of datasets, evaluation metrics, and deployment conditions. Existing studies show that code pretraining, prompt engineering, supervised fine-tuning, and structural-information enhancement can improve detection or explanation results on specific benchmarks. However, such improvements are easily affected by duplicate samples, label noise, random splits, temporal leakage, and changes in model versions, which may lead to overestimated performance. Therefore, model evaluation should not rely on a single metric alone. It should also report the data source, detection granularity, temporal or project-level isolation, false-positive cost, and whether generated explanations can be verified by program evidence. On this basis, this paper further discusses an application approach centered on data governance, strict data splitting, uncertainty calibration, and SAST-LLM-human collaboration. The discussion is limited to source-code-level vulnerability detection, localization, and classification, and does not cover network intrusion detection, malware analysis, smart-contract-specific detection, vulnerability generation, or automatic repair.

Keywords

large language models, source code vulnerability detection, code pretraining models, prompt engineering, reliability evaluation

1. Introduction

1.1 Research Background and Detection Tasks

Software vulnerability detection is intended to identify exploitable implementation defects as early as possible before code is deployed in a production environment. Static application security testing (SAST) mainly relies on rule libraries and program-analysis procedures, and its results are usually traceable. In practical use, however, its coverage is constrained by the completeness of rule libraries, support from language front ends, and differences

across tool versions. By contrast, data-driven methods learn vulnerability patterns from historical code and its labels. In recent years, related research has gradually moved toward code pretraining models and large language models (LLMs). LLMs can process new vulnerability descriptions based on natural-language instructions and can further generate suspicious locations, causal explanations, and repair suggestions [1-4].

Source code vulnerability detection is not a single binary-classification task. Function-level detection answers whether a code segment contains a vulnerability. Line-level localization further points out locations that auditors should inspect first. Vulnerability type classification is used to associate results with security knowledge such as Common Weakness Enumeration (CWE), and explanation tasks require causes, triggering conditions, or possible repair directions. LineVul extends evaluation from function judgment to line-level audit cost, and later LLM studies add generative outputs such as location, cause, and suggestion. However, an increase in output dimensions also means that correctness cannot be replaced only by language fluency [5-6].

1.2 Method Evolution and Evaluation Problems

The evolution of source code vulnerability detection methods can be summarized into three stages. Graph neural network methods such as Devign learn structural semantics through code property graphs. CodeBERT and GraphCodeBERT transfer large-scale code pretraining representations to downstream tasks. General-purpose LLMs then introduce zero-shot learning, few-shot learning, supervised fine-tuning, and multi-model collaboration. These methods do not simply replace one another. Graph structures can still supplement program dependencies for Transformers or LLMs, lightweight models are suitable for stable local deployment, and general-purpose LLMs are more convenient for interaction and explanation generation [7-13].

More methods have not automatically solved the credibility problem in evaluation. After deduplication, improved labeling, and temporal splitting, the PrimeVul study reported that the same StarCoder2 model achieved an F1 of 68.26% on BigVul but only 3.09% on the PrimeVul test set. This controlled comparison shows that duplicates, label errors, or distributional similarity in older benchmarks may be used by models. It does not imply that all 7B models will show the same decline, but it is sufficient to reject the assumption that a high score under a single random split represents deployable capability [14].

1.3 Research Questions and Main Contributions

This paper is organized around three closely related questions: what capabilities existing methods actually improve, under what data conditions and evaluation protocols these improvements are obtained, and whether the available evidence is sufficient to support deployment in real-world scenarios. Cross-project studies, unified comparisons between LLMs and SAST, and analyses related to DiverseVul all indicate that whether a tested project has been seen before, differences in programming languages, vulnerability category composition, and changes in tool versions can affect evaluation results [15-17]. Therefore, this paper first introduces the necessary background, then defines the tasks and analytical framework. It then discusses lightweight Transformer-based methods and large-scale LLM-based methods separately, and finally compares datasets, evaluation metrics, performance boundaries, and practical challenges.

The main contributions of this paper are as follows:

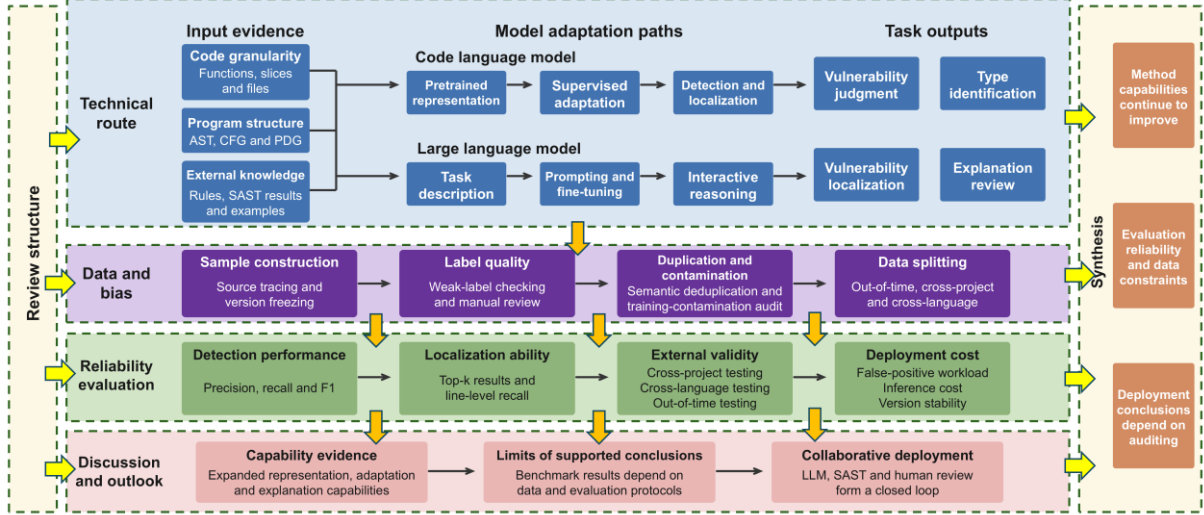
(1) Taking lightweight code Transformers and large-scale LLMs as two lines of discussion, this paper classifies detection, localization, and explanation methods according to the source of vulnerability knowledge, input organization, and type of output evidence.

(2) This paper incorporates data quality and evaluation reliability into method comparison, analyzes how duplicate samples, label noise, random splits, temporal leakage, and metric selection affect conclusions, and distinguishes performance gains within a benchmark from cross-project and out-of-time generalization ability.

(3) In connection with real audit workflows, this paper discusses the applicability boundaries of different methods and proposes a collaborative approach: SAST generates traceable candidates, LLMs rank and explain them, and human auditors handle high-risk judgments. It also summarizes future research directions such as data governance, uncertainty calibration, and cost evaluation.

As shown in Figure 1, this paper organizes the analysis around technical routes, data bias, reliability evaluation, and deployment boundaries.

Figure 1: Review Framework of This Paper



2. Preliminary Knowledge

2.1 Transformers and Large Language Models

Transformers use self-attention to model dependencies between tokens. A model first learns general representations from large-scale natural language or code and then adapts to specific tasks through prompts or parameter updates. In this paper, lightweight code Transformers refer to models with relatively small parameter scales that usually need classifier heads trained on vulnerability data or task-specific fine-tuning. Large-scale LLMs can receive natural-language instructions and perform zero-shot, few-shot, or instruction-tuning tasks. There is no fixed parameter boundary between the two; their differences are mainly reflected in task adaptation and deployment form.

2.2 Code Pretraining Models

Code pretraining models learn transferable representations from large-scale natural-language and programming-language corpora. CodeBERT uses a Transformer to build a shared space between text and code so that downstream tasks can be fine-tuned with less task data. GraphCodeBERT explicitly models data flow on this basis and designs structure-aware pretraining objectives [8-9]. This change shifts vulnerability detection from “training representations for a single dataset” to “first learning general code regularities and then adapting to a security task.”

The original studies of CodeBERT and GraphCodeBERT did not directly verify vulnerability detection performance. They provided transferable code representations. When they are used for security tasks, input granularity, maximum length, positive-negative sample ratio, structural information, and training objectives still need to be decided. LineVul uses code pretraining representations for function classification and line-level localization, and SIFT uses structural knowledge and multitask objectives to constrain representations [5,22]. Therefore, adopting a code pretraining model is only a methodological choice and cannot be directly regarded as performance evidence.

2.3 Traditional Detection Methods

Program-graph routes explicitly encode abstract syntax trees, control-flow graphs, program dependence graphs, or code property graphs. Devign integrates multiple kinds of program relations into a graph neural network (GNN) and establishes a representative structural-learning paradigm for function-level vulnerability identification [7]. Compared with pure sequences, graph models are more likely to aggregate nonlocal dependencies, but graph construction depends on language front ends and the accuracy of static analysis. The selection of nodes and edges also introduces new modeling assumptions, and structural information is not naturally equivalent to real vulnerability causality.

This paper discusses traditional methods only in two situations: to explain why lightweight Transformers need structural information and to use traditional methods as baselines or external tools for LLMs. SAST can provide rule hits, data flow, and call paths, and GNNs can compress program graphs. These results can be used independently for detection and can also serve as evidence in LLM prompts. To keep the topic focused, this paper does not provide a separate review of traditional detection methods.

3. Research Framework

3.1 Problem Definition and Research Questions

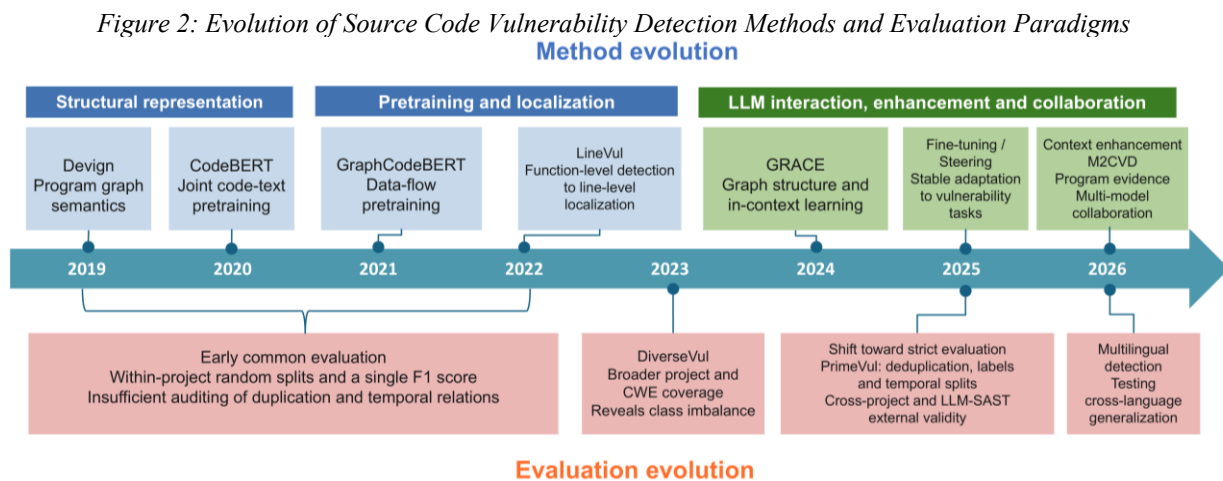
Let the source code unit be x . The detection model outputs a vulnerability label y and may optionally output a vulnerability type c , a set of locations l , and an explanation e . Function-level detection estimates $P(y|x)$, and line-level localization ranks code lines inside a function. Type classification and explanation respectively answer what class a vulnerability belongs to and why it holds. This paper includes only studies that directly analyze source code and does not mix patch generation or automatic repair into the detection task. This formal definition allows the later discussion to distinguish three types of evidence: judging the label correctly, finding the location, and giving a verifiable explanation.

This review sets four research questions. RQ1 focuses on the development of source code vulnerability detection from sequence and graph models to code pretraining models and LLM interaction. RQ2 compares the performance and capability evidence provided by direct prompting, in-context learning, fine-tuning, structural enhancement, and multi-model collaboration. RQ3 analyzes the effects of label noise, sample duplication, class imbalance, and temporal leakage on estimated results. RQ4 discusses how cross-project, out-of-time, false-positive-cost, localization, and explanation evaluations can approach actual deployment. The four questions correspond to the technical spectrum, validity evidence, sources of bias, and external validity.

3.2 Research Framework and Classification Basis

Existing reviews often organize studies by model scale, core technology, or software supply chain stage [18-21]. This paper instead classifies methods by how vulnerability knowledge is acquired. Lightweight Transformers rely on code pretraining representations and supervised fine-tuning to complete relatively stable classification or localization. Large-scale LLMs can adapt to tasks through supervised fine-tuning, prompting and in-context learning, program-analysis enhancement, and multi-model collaboration. Section 2 introduces the background needed to understand the two routes. Section 3 defines the research questions and classification basis. Sections 4 and 5 discuss the two model types respectively. Section 6 analyzes datasets and evaluation metrics. Section 7 compares performance evidence, applicability conditions, and deployment cost. Section 8 summarizes common challenges and directions for reliable application.

As shown in Figure 2, method development and evaluation priorities in this field are not independent of each other but evolve together.



4. Source Code Vulnerability Detection Methods Based on Lightweight Transformers

4.1 Input Representation and Model Selection

4.1.1 Input Granularity

When lightweight Transformers are used for vulnerability detection, studies usually first convert functions, program slices, or files into token sequences. These sequences are then truncated or divided into chunks according to the maximum length supported by the model, and a pretrained encoder with a classification head is used to output vulnerability probabilities. Smaller input units make model training and vulnerability localization relatively convenient, but triggering conditions that propagate across functions may be cut off from the context. When the input scope is expanded, the model can retain more program information, but this also increases computational cost and introduces noise that is not directly related to vulnerability judgment. Therefore, input granularity is not merely a preprocessing detail; it is a key setting that directly affects detection performance.

4.1.2 Code Representation and Model Selection

Model selection mainly includes general code pretraining, structured code pretraining, and structural fusion. CodeBERT focuses on shared representations between natural language and code, GraphCodeBERT introduces data flow, and Devign-style program graphs can be used as external encoders or fused with Transformers [7-9]. The choice of representation should depend on the evidence needed to trigger vulnerabilities. Local API misuse may be identified from token context, whereas data dependencies across basic blocks depend more on structural information.

4.2 Pretraining, Fine-Tuning, and Detection Granularity

4.2.1 Pretraining and Task Adaptation

Pretraining objectives determine what information a model tends to retain, and vulnerability fine-tuning then uses general representations for security judgments. Common objectives include masked-token prediction, replaced-token detection, code-text matching, and data-flow-related tasks. Downstream tasks mostly use cross-entropy for binary function classification, and they may also predict CWE, location, or auxiliary structures. Multitask learning may be effective only when auxiliary objectives truly add constraints; task conflict and data bias still need to be judged through ablation experiments [22-23].

4.2.2 Detection Granularity and Localization Evidence

Whether sequence modeling or graph modeling is used, the choice of granularity and the ability to generalize remain unavoidable issues. LineVul applies Transformers and code pretraining representations to vulnerability detection and extends function-level classification to line-level localization. The study uses a dataset of more than 188,000 functions and, under its own test setting, reports F1 improvements of 160%-379% over baseline methods, Top-10 Accuracy improvements of 12%-25%, and Effort@20%Recall reductions of 29%-53% [5]. These results indicate that evaluation is no longer concerned only with whether the model makes a correct hit, but also with how much code an auditor needs to inspect. However, the BigVul-style data and splitting strategy used by LineVul still have limitations, so these conclusions need to be reexamined on stricter benchmarks. Results from DiverseVul also show that increasing the scale of function-level data does not necessarily lead to simultaneous improvement across all model architectures [17].

Detection granularity must also match the source of labels. Function-level labels automatically generated from repair commits do not naturally provide exact vulnerable lines, and line-level attention weights in a model cannot be directly treated as causal explanations. Therefore, if a study claims that a model has localization or explanation ability, it should separately report localization hits, audit workload, or the consistency between model outputs and program evidence. Overall, lightweight models are convenient for local deployment and control, their inference process is relatively stable, and their operating cost is lower. However, their performance is still limited by context-window length, structural encoding methods, and the quality of task data.

4.2.3 Applicability Boundaries

Lightweight Transformers are better suited to scenarios where the label system is relatively stable, data can be continuously maintained locally, and batch scanning is required. When evaluating this type of model, it is not enough to consider only the final score. The effects of code representation, the introduction of structural information, training-objective design, and the data-processing pipeline should be analyzed separately. If duplicate samples exist between the training and test sets, or if labels mainly come from repair commits and do not accurately correspond to real vulnerabilities, a high F1 score still does not show that the model has learned vulnerability semantics that can transfer across projects.

5. Source Code Vulnerability Detection Methods Based on Large-Scale Language Models

5.1 Fine-Tuning Methods

5.1.1 Supervised Fine-Tuning and Lightweight Control

Supervised fine-tuning updates model parameters with vulnerability labels, allowing a model originally designed for general generation to gradually adapt to security classification or vulnerability explanation tasks. Existing studies usually compare different model scales and training strategies, and they often observe certain task-adaptation gains on fixed public datasets [24]. Compared with direct prompting, a fine-tuned model produces a more stable output format and depends less on long prompts. However, this approach requires higher training cost and is also more likely to inherit duplicate samples, class imbalance, and label noise from the data. Therefore, a high score first indicates that the model fits the training distribution; it does not naturally prove out-of-time generalization.

In addition to full supervised fine-tuning, lightweight control methods such as steering have also been used to strengthen a model's vulnerability judgment. These methods usually influence the judgment result by adjusting the model's internal representations or output direction [25]. They reduce the number of parameters that need to be trained, but their control signals often still depend on the base model itself. Therefore, their stability should still be verified separately across different models, datasets, and random seeds.

5.1.2 Multitask Learning and Output Protocols

Multitask learning usually trains vulnerability detection together with vulnerability type identification, location prediction, explanation generation, or other auxiliary objectives within the same framework. Multitask evaluations of open-source LLMs show that a single binary classification score is difficult to use as a complete summary of a model's software security capability. Structured multitask models such as SIFT introduce auxiliary tasks to place additional constraints on shared representations [22-23]. In addition, some studies analyze how different LLM components affect performance, but identifying where the gains actually come from is still limited by baseline settings, data conditions, and training budget [26]. Therefore, multitask research needs ablation experiments or task-interference analysis, and the effectiveness of each module cannot be inferred from overall F1.

Fine-tuning large-scale LLMs also requires clear input and output protocols. Inputs may include functions, file fragments, vulnerability type descriptions, or program-analysis results. Outputs may be labels, structured JSON, or explanatory text. The more open the output is, the more parsing failure rate and factual verification are needed. Parameter-efficient fine-tuning reduces training cost, but it cannot remove corpus contamination and version differences in the base model. As shown in Table 1, representative methods can be compared from six aspects: code representation, learning mode, dataset, evaluation metrics, advantages, and limitations.

Table 1: Comparison of Representative Source Code Vulnerability Detection Methods

Method	Code representation	Learning mode	Main data setting	Metrics	Advantages	Limitations
Devign	Code property graph	GNN supervised learning	Four large C projects	Accuracy, F1	Explicitly aggregates program semantics	Limited by project scope and random splitting

Method	Code representation	Learning mode	Main data setting	Metrics	Advantages	Limitations
CodeBERT	NL-PL sequence	General pretraining	CodeSearchN et, etc.	Downstream task metrics	Provides transferable code representations	Original study is not vulnerability detection
GraphCodeBERT	Sequence + data flow	Structured pretraining	CodeSearchN et, etc.	Downstream task metrics	Introduces definition-use relations	Downstream security gains need separate verification
LineVul	Transformer/CodeBERT	Supervised fine-tuning + localization	188k+ C/C++ functions	F1, Top-k, Effort	Considers both function detection and line-level localization	BigVul-style data limits generalization
GRACE	AST/CFG/PDG + LLM	In-context learning + retrieval	Three public datasets	P/R/F1	Jointly enhances structure and examples	Sensitive to prompts, benchmarks, and contamination
EnStack	Multiple LLM outputs	Stacking ensemble	Public vulnerability data	Accuracy, P/R/F1	Uses model complementarity	High cost and possibly correlated errors
DLAP	Code + auxiliary model information	Two-stage prompting	Public vulnerability benchmarks	F1, etc.	Uses external models to assist LLMs	Upstream errors may propagate
Finetuning LLM	Code tokens	Supervised fine-tuning	Public vulnerability data	Classification metrics	Stable task adaptation	Inherits duplicate and label bias
LLMVulExp	LLM + key code	LoRA/instruction fine-tuning	SeVC, etc.	Classification and explanation metrics	Outputs type, location, cause, and suggestion	Explanation correctness needs independent verification
Cross-project zero-shot	Code representation	Zero-shot transfer	Cross-project data	F1, MCC, etc.	Directly tests unseen projects	Not equivalent to out-of-time deployment
Context-Enhanced	Program abstraction + LLM	Context prompting	Public vulnerability data	Classification metrics	Compares different context levels	Depends on model and analyzer versions
M2CVD	Multi-model semantic information	Generation-filtering collaboration	Public vulnerability data	F1, etc.	Mitigates insufficient single-model understanding	Cost and common contamination on risk

5.2 Prompt Engineering and In-Context Learning

5.2.1 Direct Prompting

Direct prompting organizes code and security questions into natural-language instructions and asks a general-purpose LLM to give a judgment or explanation. Early studies mainly test the feasibility of GPT-like models without task fine-tuning and also compare joint vulnerability and functionality judgments and different input modes [10,27-28]. This method adapts quickly to new output formats, but results are easily affected by prompt wording, context length, and model version, and repeated calls may produce inconsistent answers.

5.2.2 Few-Shot Learning and External Information Enhancement

Few-shot in-context learning adds labeled examples to prompts. The key is not only the number of examples but also their lexical, semantic, or program-structural relevance to the code under test. GRACE extracts information from structures such as abstract syntax trees (ASTs), control-flow graphs (CFGs), and program dependence graphs (PDGs), and combines example retrieval to enhance context. The paper reports that, in three settings, FFmpeg+Qemu, ReVeal, and Big-Vul, its F1 improvement over six baselines is at least 28.65% [11]. This result is a relative improvement rather than 28.65 percentage points, and it applies only to the tested models, baselines, and prompt configurations.

DLAP first uses a deep learning model to generate auxiliary information and then writes that information into the LLM prompt, forming a two-stage workflow of “lightweight detector-language model” [12]. This shows that an LLM does not have to analyze all code alone, and an external model can provide candidates, structures, or representations. However, errors from the previous stage also enter the later prompt. If the auxiliary model overfits duplicate samples, the LLM may only restate the original bias.

5.3 Hybrid Detection with LLMs and Program Analysis

5.3.1 Program Structure and Context

A single function or truncated token sequence often cannot present complete vulnerability triggering conditions. GRACE uses graph structures for in-context learning, and the 2026 context-enhancement study further compares program information at different abstraction levels and finds that suitable levels vary with the model [11,29]. More context is not necessarily better. Irrelevant paths occupy the context window, and static-analysis errors may also be amplified by language models.

5.3.2 Static Analysis Collaboration

A more feasible workflow is “SAST candidate generation-LLM review or explanation-human confirmation.” Program analysis provides traceable results to the LLM, and the LLM ranks alerts, gives semantic explanations, or proposes verification questions [29]. Generated text itself cannot serve as a security conclusion, and high-risk fixes still need verification through path analysis, testing, or expert review. Whether a hybrid workflow is effective should be measured by audit time, false-negative control, and explanation consistency, not only by overall F1.

The focus of hybrid methods is to preserve the source of external evidence. Program slices, control flow, data flow, and SAST alerts should all be linked to specific locations so that LLM judgments can be checked against code paths. If static-analysis outputs are only concatenated into text, tool errors may still be amplified. Only by clarifying the responsibilities of candidate generation, ranking explanation, and human confirmation can an auditable workflow be formed.

5.4 Ensemble, Multi-Model Collaboration, and Explanation

5.4.1 Model Ensemble and Collaboration

Ensemble methods usually combine predictions from multiple LLMs to reduce the effect of unstable outputs from a single model. M2CVD goes a step further by allowing different models to participate in the generation and filtering of vulnerability semantic information, forming a collaborative judgment process [13,30]. However, whether such methods can bring stable gains depends on whether the participating models truly differ in capability and error patterns. If several models use similar pretraining corpora or are fine-tuned on the same biased data, simply increasing the number of models cannot remove shared contamination. Instead, it increases calling cost and makes experimental reproduction more difficult.

5.4.2 Vulnerability Explanation and System Cost

Explanation generation moves model output beyond vulnerability labels and further includes suspicious locations, causes, and repair suggestions. In the SeVC single-type detection setting, LLMVulExp reports an average F1 of 92.4% and generates several types of explanatory information at the same time [6]. This result indicates that vulnerability detection and explanation generation can be modeled within the same framework. However, the conclusion is still limited by the experimental setting and cannot be directly generalized to

PrimeVul, cross-project testing, or real deployment scenarios. When evaluating explanation quality, at least four aspects should be examined separately: whether the language expression is complete, whether the explanation is consistent with the vulnerability location, whether the program path can be verified, and whether the repair suggestion is safe. A model's own judgment that the text is fluent is not enough to prove that the explanation itself is correct.

Ensemble and collaboration also increase system cost. Beyond F1, studies should report the number of model calls, latency, video memory or API cost, and whether members share pretraining corpora and error patterns. Explanatory output needs support from vulnerability location, control or data dependence, test cases, or expert judgment; language coherence cannot replace program correctness.

Large-scale LLMs can describe tasks in natural language and access examples, retrieval results, program structures, and tool outputs. The costs are prompt sensitivity, nondeterminism, computation cost, and version drift. Fine-tuning is suitable for stable tasks, prompting is convenient for quick adaptation, hybrid methods emphasize traceable evidence, and collaboration methods use model differences. The concrete choice still depends on data and deployment conditions.

6. Datasets and Evaluation Metrics

6.1 Dataset Construction and Common Benchmarks

Vulnerability data are often constructed from repair commits. Researchers locate pre-fix code according to security advisories, commit information, or patches and label the corresponding functions as vulnerable samples. This approach is convenient for expanding data scale, but it may label synchronized changes unrelated to security repair as vulnerabilities and may label related functions that were not modified in the same commit as non-vulnerable. Automatic labeling by static rules is limited by rule coverage, whereas manual verification depends on expert protocols and consistency. Recent reviews generally require model comparisons to first report label sources [2-4].

DiverseVul is constructed from 7,514 commits and contains 18,945 vulnerable functions, 330,492 non-vulnerable functions, and 150 CWE classes [17]. These numbers show wider project and vulnerability coverage and also reveal obvious class imbalance: non-vulnerable functions far outnumber vulnerable functions, and rare CWE samples are further insufficient. Increasing data volume can increase coverage, but it does not automatically repair weak labels at the commit level and cannot ensure that difficult vulnerabilities receive enough supervision.

From the perspective of construction methods, vulnerability data can be roughly divided into three types: manually designed examples, real repair commits, and data generated by automatic rules. Manually designed examples usually have clear labels, but their scale is limited, and they are still distant from real development scenarios. Data based on repair commits are closer to actual development processes, but they often contain weak labels and may also include other changes made at the same time as security fixes. Data generated by automatic rules are convenient for scale expansion, but their coverage is still limited by the rules themselves. Therefore, when using a dataset, researchers should not report only its name. They should also describe the data version, project sources, commit-time range, CWE distribution, deduplication rules, and the way the training and test sets are split.

6.2 Data Quality and Splitting

6.2.1 Duplicate Samples and Training Contamination

Data duplication does not refer only to exactly identical functions. It also includes near-duplicate samples in which only whitespace, variable names, or identifiers are changed, as well as paired code samples that are highly similar before and after the same repair commit. If such samples appear in both the training and test sets, the model may simply memorize lexical features instead of learning transferable vulnerability patterns. Random splitting within the same project has a similar problem, because shared templates, library code, and developers' coding habits may also enter the test set. The performance gap observed in PrimeVul directly reflects this risk, and the results from DiverseVul also suggest that cross-project results should be interpreted separately from random-split results [14,17].

Training contamination may also come from the pretraining corpora of base models. If public repositories, patches, or benchmark code have already been included in pretraining data, the model may not be seeing the relevant samples for the first time during testing. For closed-source models or extremely large corpora, researchers usually cannot prove whether a specific sample has appeared. Therefore, this paper does not treat contamination as a confirmed fact, but regards it as an alternative explanation that needs to be audited. High scores in comprehensive benchmarks or fine-tuning studies are more likely to rule out memorization effects only when the training data, test time, and code sources can be clearly distinguished [24,31].

6.2.2 Random Splits and Temporal Splits

Random splitting assumes that training and test samples come from the same distribution. It is suitable for estimating within-benchmark generalization, but it does not correspond to realistic scenarios of detecting future commits or unseen repositories. Coding conventions, dependent libraries, and vulnerability types in the same project may be shared between two sets, thereby reducing test difficulty. Therefore, F1 obtained from a random split should be called “within-benchmark performance” and cannot be directly described as a “real-world detection rate” [14].

Temporal splitting trains on the past and tests on the future according to commit time, and ensures that the same commit and its near duplicates belong to only one set. It is closer to use in continuous integration and can also reduce the chance that the model indirectly sees future repair information. PrimeVul observes an obvious performance drop after adopting the temporal principle, indicating that out-of-time testing should become a reliability baseline [14]. A stricter design should also combine project isolation and report time windows, data freeze dates, and model knowledge-cutoff information.

6.2.3 Data Governance and Risk Identification

Data governance needs to handle exact duplicates, near duplicates, project associations, and temporal relations at the same time. Hashing can find exactly identical code, and normalization, clone detection, or structural similarity can supplement near-duplicate identification. Project, repository, commit, and time identifiers are used to prevent associated samples from crossing sets. Pretraining contamination in closed-source base models usually cannot be fully audited, so papers should clarify unknown scopes and should not treat knowledge-cutoff dates as proof of no contamination. As shown in Table 2, dataset and evaluation risks can be checked item by item together with identification signals and mitigation measures.

Table 2: Risk Matrix for Datasets and Evaluation

Risk	Risk mechanism	Possible bias	Identification signal	Mitigation measure	Evidence and remaining limitation
Label noise	Weak labels from repair commits, rule coverage, or human disagreement	Treats non-security changes as vulnerabilities or misses related functions	Sample-review disagreement and inconsistency between patches and labels	Record label sources; use two-person review; release disputed labels	PrimeVul and DiverseVul; complete ground truth remains difficult
Exact/near duplicates	Templates, clones, and pre-/post-fix code cross sets	Counts memorization as generalization	High lexical or AST similarity; project overlap	Hashing, normalization, clone detection; commit isolation	PrimeVul; semantic thresholds still affect results
Pretraining contamination	Public repositories or patches enter base-model corpora	Test samples may have been seen by the model	Unusually familiar explanations; opaque knowledge cutoff	Report corpus information, model versions, and non-auditable parts	Most closed-source models cannot be fully ruled out
Random splitting	Same-project and same-time	Overestimates performance on unseen	Cross-project or out-of-time	Use project isolation and	PrimeVul; historically comparable results

Risk	Risk mechanism	Possible bias	Identification signal	Mitigation measure	Evidence and remaining limitation
	distributions are shared	repositories and future commits	results drop clearly	temporal splitting	still need to be retained
Cross-project/language failure	Differences in coding conventions, ecosystems, and syntax	Overall scores hide domain imbalance	Grouped differences by project, language, and CWE	Leave-project-out testing; per-language reports; domain adaptation	Cross-project and multilingual studies; sample composition is mixed
Class imbalance	Non-vulnerable functions far outnumber vulnerable functions	Inflated Accuracy and missed rare CWEs	High majority-class baseline; low grouped Recall	Report P/R/F1, PR curves, and grouped results	DiverseVul; resampling may change the distribution
False-positive/false-negative cost	Classification metrics are not mapped to audit workload	High F1 may still produce unusable alerts	Alert volume under fixed budget and cost per true positive	Report FPR/FNR, Top-k, and Effort@Recall	SAST and LineVul; organizational cost varies by scenario
Explanation/version validity	Language coherence is not program correctness; APIs update continuously	Misleading explanations or irreproducible experiments	Path inconsistency, repeated-call fluctuation, version drift	Verify program evidence; lock versions, prompts, and parameters	LLMVulExp and context enhancement; human protocols still need unification

6.3 Evaluation Metrics

Different classification metrics reflect different questions. Precision focuses on how many alerts produced by the model are truly credible, Recall focuses on how many real vulnerabilities are found, and F1 takes the harmonic mean of the two. When vulnerability data are highly imbalanced, reporting Accuracy alone can easily hide errors on the minority class. For this reason, PrimeVul introduces paired samples, VD-Score, and other checks that are closer to the vulnerability identification process. Studies on SAST also point out that both rule coverage and the number of false positives affect practical usability [1,14].

Evaluation for deployment should also convert model alerts into audit workload. LineVul uses Top-k Accuracy and Effort@20%Recall to measure the priority of localization results. A unified LLM-SAST benchmark compares both vulnerability detection and false positives, showing that the two kinds of tools may complement each other on different vulnerability types [5,15]. For generative explanations, it is not enough to check whether the text is complete. It is also necessary to examine whether the vulnerability location, causal explanation, and repair suggestion are consistent with the code. LLMVulExp reflects this extended evaluation direction, but automatic metrics, human annotation protocols, and consistency standards for explanation evaluation still need further unification [6].

Therefore, an evaluation report should at least include Precision, Recall, F1, and a confusion matrix, and should provide grouped results by project, time, programming language, or CWE type. For line-level localization methods, Top-k results and the amount of code that needs to be audited under a fixed recall should also be reported. For generative explanations, location, type, cause, and suggestion should be checked separately against program evidence. Accuracy can only be used as a supplementary metric. If an experiment calls an LLM multiple times, it should also report the mean results, fluctuation range, and parsing failure rate.

7. Comparison of Method Performance and Applicability

7.1 Comparison with Deep Learning and SAST

Studies on prompting and fine-tuning do provide evidence of performance improvement on public datasets, but these gains are not fixed. They vary with model type, dataset, parameter configuration, and training budget

[24,31]. Therefore, this paper gives only medium confidence to C2: prompting, fine-tuning, and structural enhancement can improve performance under the tested settings, but existing results are still insufficient to show that any single route can remain comprehensively superior across all programming languages, project environments, and future vulnerabilities.

Low scores under strict evaluation do not mean that LLMs have no value in vulnerability detection. A more important lesson from PrimeVul is that the evaluation protocol itself can change experimental conclusions; the single result of 3.09% cannot be directly generalized to all models. Conversely, high scores on SeVC or BigVul cannot simply be treated as detection capability in production environments. In other words, current evidence shows that model capability has indeed improved, while also exposing insufficient external validity. Both points need to be retained.

When comparing lightweight Transformers with general-purpose LLMs, model scale is not the only criterion. Lightweight Transformers are more suitable for scenarios with fixed label systems, local batch processing, and lockable versions. General-purpose LLMs are more convenient for few-shot adaptation, explanation generation, and complex context processing. SAST is usually more stable within the scope covered by existing rules, whereas LLMs are better at semantic summarization and alert explanation. Existing evidence from unified benchmarks supports complementarity rather than complete replacement by one tool: the two types of tools may find different vulnerabilities and may also produce different kinds of false positives [1,15].

7.2 Comparison of Different LLM Methods

Prompting, fine-tuning, structural enhancement, and multi-model collaboration solve different problems. Prompting lowers the adaptation threshold, fine-tuning improves output stability, structural enhancement supplements program dependencies, and collaboration uses model differences. GRACE, DLAP, context enhancement, and M2CVD all report gains in their own settings, but datasets, model versions, and evaluation protocols differ, so they cannot be directly ranked on this basis [11-13,29]. Comprehensive benchmark studies also show that preprocessing and prompt changes alter model rankings [26,31].

Table V in PrimeVul compares five code language models. When trained and tested on BigVul, F1 is 62.88%-68.26%. When BigVul-trained models are transferred to PRIMEVUL testing, F1 drops to 1.91%-5.82%. When PRIMEVUL is used for both training and testing, F1 rises to 18.05%-21.43% [14]. Under the strict setting, VD-S is 88.78%-91.51%, both paired functions are correct in only 1.06%-3.01% of cases, and both are judged safe in 84.22%-88.30% of cases. The above decline is not an isolated case of StarCoder2, but the conclusion is limited to the five models and three settings in Table V.

7.3 Cost and Deployment Choices

Deployment plans depend on whether code can be sent outside, whether the task is stable, whether context depends on cross-function or tool evidence, and the false-positive and computation costs that an organization can bear. Sensitive code and local large-scale scanning are usually more suitable for lightweight models or SAST. Rapidly changing vulnerability descriptions, interactive auditing, and explanation tasks are more suitable for LLMs. High-risk scenarios should use hybrid workflows and retain human confirmation. API calls, video memory, latency, model updates, and prompt maintenance should all be counted as costs; a one-time offline F1 improvement cannot directly represent engineering gain.

8. Practical Challenges and Future Prospects

8.1 Main Bottlenecks of the Two Types of Models

8.1.1 Constraints of Lightweight Models

Lightweight Transformers are mainly constrained by context length, interprocedural structure, and task data. Expanding inputs increases computation and noise, and simply concatenating callers, callees, or program graphs may also dilute key evidence. Future research should select context around vulnerability triggering paths and compare function, slice, file, and cross-function representations under a fixed budget, rather than assuming that longer input is always better.

8.1.2 Constraints of Large Language Models

Large-scale LLMs face problems of prompt robustness, hallucination, nondeterminism, version drift, and deployment cost. A model may generate explanations that cannot be supported by code paths, and may change its judgment when prompts change slightly. Experiments need to preserve system prompts, model identifiers, call dates, temperatures, and repetition counts, and should set mechanisms for refusal and insufficient evidence. For closed-source APIs, the inability to fully lock versions is itself a research limitation.

8.2 Common Challenges in Data and Evaluation

Evaluation should combine out-of-time, cross-project, and cross-language designs. Cross-project testing alone may still share era and library ecosystems, and temporal splitting alone may preserve patterns from the same project. Multilingual studies also need to report the data volume and CWE composition for each language [16,32]. A feasible minimum plan is to retain random splits for historical comparison while treating out-of-time or project-isolated results as the main reliability evidence.

Uncertainty and usage cost should also be included in evaluation. A model should not only provide a judgment result, but also offer confidence that can be calibrated. When the evidence is insufficient, the model should be allowed to refuse to answer. Under a fixed false-positive budget, the corresponding recall should also be reported. For explanation systems, it is also necessary to distinguish between the model's own expression of confidence and whether the program evidence is sufficient. Human-machine collaboration experiments can further record the audit time required to find one real vulnerability, the number of false positives that need to be handled, and whether defects still appear after model suggestions are adopted, rather than only measuring similarity between explanation texts.

At present, there is still no unified evaluation protocol for explanation correctness. A more feasible approach is to decompose an explanation into vulnerability location, triggering condition, data path or control path, CWE type, and repair suggestion, and then verify each part separately. At the same time, cross-project, cross-language, and out-of-time tests should be used together to reduce the effect of hidden distribution sharing on conclusions. Future research also needs to set reviewable acceptance evidence for context selection, data governance, evaluation design, and deployment responsibility.

8.3 Research Directions for Reliable Application

Reliable application first requires a traceable data pipeline. The data records should include public commits, time information, label sources, and statistics on exact and near-duplicate samples, together with updateable data cards. In evaluation, out-of-time testing and project-isolated testing should serve as the main evidence, while random-split results are more suitable for comparison with previous studies. Uncertainty calibration and risk-coverage curves can help models issue selective alerts under a fixed false-positive budget. Human-machine collaboration studies should record the audit time required to find real vulnerabilities, the amount of false-positive handling, and the results of explanation verification.

The combination of LLMs and SAST should not simply concatenate their outputs. A more reasonable approach is that static analyzers provide replayable program paths and candidate alerts, LLMs perform ranking, semantic summarization, and interactive explanation, and human auditors handle high-risk judgments. When evaluating such a system, candidate recall, the filtering gain brought by the LLM, explanation correctness, and total audit cost can be measured separately. Only when the responsibility and possible failure modes of each stage are clearly defined can system improvements be accurately located and reproduced in later research or deployment.

9. Conclusion

This paper reviews research on source code vulnerability detection based on large language models from the perspectives of preliminary knowledge, research framework, lightweight Transformers, large-scale LLMs, data and metrics, performance comparison, and practical challenges. Lightweight models usually rely on task-specific data for fine-tuning and are more suitable for scenarios with stable labels, local deployment, and batch processing. Large-scale LLMs extend task adaptation and explanation generation through prompting, fine-tuning, program-analysis context, and multi-model collaboration. Both types of methods may achieve

performance improvements on specific benchmarks, but the related conclusions are affected by label quality, duplicate samples, data splitting, evaluation metrics, and model versions.

Therefore, evaluation cannot remain at a single F1 score. It is also necessary to examine whether the data are independent, whether task granularity is consistent with the label source, whether the results remain stable under cross-project and out-of-time settings, and whether the costs caused by false positives, localization, and explanation are acceptable. Existing evidence better supports a complementary relationship among LLMs, SAST, and human auditing, rather than replacing the complete security process with a single model. Only when method design advances together with data governance and strict evaluation can advantages on benchmarks be transformed into auditable and reproducible engineering capability.

References

- [1] Li, Z., Liu, Z., Wong, W. K., Ma, P., & Wang, S. (2024). Evaluating C/C++ vulnerability detectability of query-based static application security testing tools. *IEEE Transactions on Dependable and Secure Computing*, 21(5), 4600-4618. <https://doi.org/10.1109/TDSC.2024.3354789>
- [2] Liang, C., Wei, Q., Du, J., Wang, Y., & Jiang, Z. (2025). Survey of source code vulnerability analysis based on deep learning. *Computers & Security*, 148, 104098. <https://doi.org/10.1016/j.cose.2024.104098>
- [3] Shiri Harzevili, N., Boaye Belle, A., Wang, J., Wang, S., Jiang, Z. M., & Nagappan, N. (2024). A systematic literature review on automated software vulnerability detection using machine learning. *ACM Computing Surveys*, 57(3), 1-36.
- [4] Su, H., Xu, Z., Zhang, Y., & Tan, Q. (2026). Source code vulnerability detection based on deep learning: a review. *Cybersecurity*, 9(1), 2. <https://doi.org/10.1186/s42400-025-00518-7>
- [5] Fu, M., & Tantithamthavorn, C. (2022, May). Linevul: A transformer-based line-level vulnerability prediction. In *Proceedings of the 19th international conference on mining software repositories* (pp. 608-620). <https://doi.org/10.1145/3524842.3528452>
- [6] Mao, Q., Li, Z., Hu, X., Liu, K., Xia, X., & Sun, J. (2025). Towards explainable vulnerability detection with large language models. *IEEE Transactions on Software Engineering*, 51(10). <https://doi.org/10.1109/TSE.2025.3605442>
- [7] Zhou, Y., Liu, S., Siow, J., Du, X., & Liu, Y. (2019). Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32. arXiv:1909.03496
- [8] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., ... & Zhou, M. (2020, November). Codebert: A pre-trained model for programming and natural languages. In *Findings of the association for computational linguistics: EMNLP 2020* (pp. 1536-1547). <https://doi.org/10.18653/v1/2020.findings-emnlp.139>
- [9] Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., ... & Zhou, M. (2020). Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*.
- [10] Zhou, X., Zhang, T., & Lo, D. (2024, April). 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* (pp. 47-51). <https://doi.org/10.1145/3639476.3639762>
- [11] Lu, G., Ju, X., Chen, X., Pei, W., & Cai, Z. (2024). GRACE: Empowering LLM-based software vulnerability detection with graph structure and in-context learning. *Journal of Systems and Software*, 212, 112031. <https://doi.org/10.1016/j.jss.2024.112031>
- [12] Yang, Y., Zhou, X., Mao, R., Xu, J., Yang, L., Zhang, Y., ... & Zhang, H. (2025). Dlap: A deep learning augmented large language model prompting framework for software vulnerability detection. *Journal of Systems and Software*, 219, 112234. <https://doi.org/10.1016/j.jss.2024.112234>

- [13] Wang, Z., Li, G., Li[♂], J., Li, J., Yan, M., Xiong, Y., & Jin, Z. (2026). M2CVD: Enhancing vulnerability understanding through multi-model collaboration for code vulnerability detection. *ACM Transactions on Software Engineering and Methodology*, 35(7), 1-25. <https://doi.org/10.1145/3771923>
- [14] Ding, Y., Fu, Y., Ibrahim, O., Sitawarin, C., Chen, X., Alomair, B., ... & Chen, Y. (2024). Vulnerability detection with code language models: How far are we?. *arXiv preprint arXiv:2403.18624*.
- [15] Gnieciak, D., & Szandala, T. (2025). Large language models versus static code analysis tools: A systematic benchmark for vulnerability detection. *IEEE Access*, 13, 198410-198422. <https://doi.org/10.1109/ACCESS.2025.3635168>
- [16] Haque, R., Ali, A., McClean, S., & Khan, N. (2026). A zero-shot framework for cross-project vulnerability detection in source code. *Empirical Software Engineering*, 31(1), 3. <https://doi.org/10.1007/s10664-025-10749-4>
- [17] Chen, Y., Ding, Z., Alowain, L., Chen, X., & Wagner, D. (2023, October). Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection. In *Proceedings of the 26th international symposium on research in attacks, intrusions and defenses* (pp. 654-668). <https://doi.org/10.1145/3607199.3607242>
- [18] Jiang, Y., Yang, Z.-H., Su, X.-H., Huang, S., & Wang, T.-T. (2026). A survey of source code vulnerability detection methods based on large language models. *Chinese Journal of Computers*, 49(6), 1385–1423. <https://doi.org/10.11897/SP.J.1016.2026.01385>.
- [19] Liu, J., Tian, X., Shu, Y., Zhu, X., Liu, Y., & Liu, Q. (2024). A review of security research in the development stage of software supply chain enhanced by large models. *Journal of Cyber Security*, 9(5), 87–109. <https://doi.org/10.19363/J.cnki.cn10-1380/tn.2024.09.11>.
- [20] Taghavi Far, S. M., & Feyzi, F. (2025). Large language models for software vulnerability detection: a guide for researchers on models, methods, techniques, datasets, and metrics. *International Journal of Information Security*, 24(2), 78. <https://doi.org/10.1007/s10207-025-00992-7>
- [21] Sheng, Z., Chen, Z., Gu, S., Huang, H., Gu, G., & Huang, J. (2025). Llms in software security: A survey of vulnerability detection techniques and insights. *ACM Computing Surveys*, 58(5), 1-35. <https://doi.org/10.1145/3769082>
- [22] Wang, L., Lu, G., Chen, X., Dai, X., & Qiu, J. (2025). SIFT: enhance the performance of vulnerability detection by incorporating structural knowledge and multi-task learning. *Automated Software Engineering*, 32(2), 38. <https://doi.org/10.1007/s10515-025-00507-7>
- [23] Yin, X., Ni, C., & Wang, S. (2024). Multitask-based evaluation of open-source llm on software vulnerability. *IEEE Transactions on Software Engineering*, 50(11), 3071-3087. <https://doi.org/10.1109/TSE.2024.3470333>
- [24] Shestov, A., Levichev, R., Mussabayev, R., Maslov, E., Zadorozhny, P., Cheshkov, A., ... & Krassovitskiy, A. (2025). Finetuning large language models for vulnerability detection. *IEEE Access*, 13, 38889-38900. <https://doi.org/10.1109/ACCESS.2025.3546700>
- [25] Li, J., Cui, L., Zhang, J., Fei, H., Chen, Y., & Zhu, H. (2025, April). Steering large language models for vulnerability detection. In *ICASSP 2025-2025 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (pp. 1-5). IEEE. <https://doi.org/10.1109/ICASSP49660.2025.10887736>
- [26] Xuan, C. D., Quang, D. B., & Quang, V. D. (2025). Large language models based vulnerability detection: How does it enhance performance? *International Journal of Information Security*, 24(1), 69. <https://doi.org/10.1007/s10207-025-00983-8>
- [27] Ingemann Tuffveson Jensen, R., Tawosi, V., & Alamir, S. (2024, April). Software Vulnerability and Functionality Assessment using Large Language Models. In *Proceedings of the Third ACM/IEEE International Workshop on NL-based Software Engineering* (pp. 25-28). <https://doi.org/10.1145/3643787.3648036>

- [28] Mahyari, A. A. (2024, October). Harnessing the power of llms in source code vulnerability detection. In MILCOM 2024-2024 IEEE Military Communications Conference (MILCOM) (pp. 251-256). IEEE. <https://doi.org/10.1109/MILCOM61039.2024.10774025>
- [29] Yang, Y., Xu, B., Gao, X., & Sun, H. (2026). Context-enhanced vulnerability detection based on large language models. *ACM Transactions on Software Engineering and Methodology*, 35(8), 1-47. <https://doi.org/10.1145/3779222>
- [30] Ridoy, S. Z., Shaon, M. S. H., Cuzzocrea, A., & Akter, M. S. (2024, December). Enstack: An ensemble stacking framework of large language models for enhanced vulnerability detection in source code. In 2024 IEEE International Conference on Big Data (BigData) (pp. 6356-6364). IEEE. <https://doi.org/10.1109/BIGDATA62323.2024.10825609>
- [31] Tamberg, K., & Bahsi, H. (2025). Harnessing large language models for software vulnerability detection: A comprehensive benchmarking study. *IEEE Access*, 13, 29698-29717. <https://doi.org/10.1109/ACCESS.2025.3541146>
- [32] Yu, J., Shu, H., Fu, M., Wang, D., Tantithamthavorn, C., Kamei, Y., & Chen, J. (2025, June). A preliminary study of large language models for multilingual vulnerability detection. In *Proceedings of the 34th ACM SIGSOFT International Symposium on Software Testing and Analysis* (pp. 161-168). <https://doi.org/10.1145/3713081.3731746>

Funding

This research received no external funding.

Conflicts of Interest

The authors declare no conflict of interest.

Acknowledgment

This paper is an output of the science project.

Copyrights

Copyright for this article is retained by the author (s), with first publication rights granted to the journal. This is an open-access article distributed under the terms and conditions of the Creative Commons Attribution license (<http://creativecommons.org/licenses/by/4.0/>).