INTERNATIONAL JOURNAL OF SCIENTIFIC RESEARCH IN ENGINEERING AND MANAGEMENT (IJSREM)
VOLUME: 10 ISSUE: 11 | NOV - 2026 SJIF RATING: 8.659 ISSN: 2582-3930

LLM-Based Software Vulnerability Detection: A Verification-Grounded Hybrid Framework for Identifying and Analyzing Security Vulnerabilities in Source Code

Meet R. Dasalaniya[1] and Parth P. Pandya[2]

[1] Department of Computer Science and Engineering, M.Tech. Software Engineering, Gandhinagar University
[2] Department of Computer Science and Engineering, M.Tech. Software Engineering, Gandhinagar University
[1] 26t0102007006@gandhinagaruni.ac.in | [2] 26t0102007009@gandhinagaruni.ac.in

ABSTRACT

Source code vulnerabilities remain a leading cause of security incidents, yet automated detection is still unreliable. Static analyzers such as CodeQL and Semgrep are precise on known patterns but miss logic-dependent flaws and generate many alerts that developers must triage. Fine-tuned transformer models such as CodeBERT and GraphCodeBERT report high scores on public benchmarks, but recent work shows that duplicated samples, noisy labels and train–test leakage inflate these scores and that performance collapses on cleaner data. Prompted large language models (LLMs) can explain their reasoning, but their answers are non-deterministic, they over-report vulnerabilities, and they sometimes justify correct verdicts with incorrect reasoning. This paper proposes a hybrid, verification-grounded framework. A static-analysis stage and program slicing supply context, a fine-tuned code encoder supplies a prior, and an open-source LLM produces a structured finding (CWE type, evidence lines and rationale). A verification layer then tests the LLM's claim against the code through evidence-line grounding, static data-flow support and self-consistency across sampled runs, and a logistic fusion of these signals yields a calibrated confidence score. The contribution is the pipeline design and a leakage-aware evaluation protocol, not a new model architecture. We specify the datasets (Big-Vul, Devign, PrimeVul and Juliet), baselines, metrics, ablations and four testable hypotheses. Experimental results are not yet available and are reported as placeholders pending implementation.

Index Terms - vulnerability detection, large language models, static analysis, CodeQL, GraphCodeBERT, self-consistency, confidence calibration, software security.

I. INTRODUCTION

A. Software security and the cost of undetected vulnerabilities

A security vulnerability is a weakness in a program that an attacker can exploit to compromise confidentiality, integrity or availability. Most vulnerabilities arise from ordinary coding mistakes: unchecked buffer bounds, unsanitized input that reaches a query or command, missing authorization checks, and unsafe handling of resources. The Common Weakness Enumeration (CWE) classifies these mistakes by type [1], while the Common Vulnerabilities and Exposures (CVE) system identifies individual disclosed instances. Finding a flaw before release is generally cheaper than patching deployed software, which makes detection directly in source code a central problem in software engineering.

B. Traditional detection techniques

Three families of techniques dominate practice. Static analysis inspects code without executing it, using pattern matching, data-flow and taint analysis, and abstract interpretation [2]. It scales well and reports line-level locations, but because it approximates program behavior it produces false alarms, and developers cite this noise as a reason for not adopting such tools [3]. Dynamic analysis, including fuzzing, executes the program on generated inputs and observes failures [4]. It reports real crashes, but only on paths the inputs reach, and it needs a build and a test harness. Manual code review is accurate for the code that is reviewed but does not scale.

Data-driven detectors emerged to reduce the reliance on hand-written rules. Early deep-learning work treated code as token sequences or code gadgets and trained recurrent or convolutional networks to classify functions [5], [6]. Graph neural networks over composite code graphs followed [7], and pretrained transformer encoders such as CodeBERT [8] became the usual starting point for fine-tuning.

C. Large language models and software security

Large language models trained on code, including Codex [9] and open models such as Code Llama [10], can read a function, describe what it does and reason about how inputs flow through it. This is attractive for security because a model can in principle name the weakness, point to the responsible lines and explain the attack path, which a binary classifier cannot. The evidence so far is mixed. A large re-evaluation on a carefully constructed dataset found that code language models perform far worse than earlier benchmarks suggested [11]. A systematic evaluation found that LLMs gave non-deterministic answers, flagged already-patched code as vulnerable, and often supplied incorrect reasoning even when the verdict was right [12]. At the same time, combining an LLM with a static analyzer produced measurable gains over the analyzer alone on a Java benchmark [13]. The useful question is therefore not whether an LLM can replace existing tools, but how its output can be constrained and checked.

D. Motivation

Three observations motivate this work. First, the reported accuracy of learned detectors cannot be trusted unless the evaluation controls for duplication and leakage [11]. Second, an LLM's explanation is valuable only if it is faithful to the code, and current models do not guarantee this [12]. Third, static analysis and learned models fail in different ways, so their disagreement carries information. A framework that treats the LLM as a hypothesis generator, and uses independent evidence to accept or reject its claims, is therefore worth building and measuring.

E. Objectives and contributions

The intended contributions of this paper are:

1. A hybrid pipeline that combines static-analysis context, a fine-tuned encoder prior and a structured LLM finding.
2. A verification layer that checks the LLM's cited evidence and data-flow claim and measures answer stability across sampled runs.
3. A calibrated confidence score that fuses these signals and is evaluated by calibration error as well as by F1.
4. A leakage-aware evaluation protocol with deduplication, chronological splitting and cross-dataset testing, together with an ablation plan.

The novelty claim is deliberately limited to the combination and the evaluation design. Each individual idea (static-analysis assistance, self-consistency and calibration) already exists in the literature reviewed in Section II. This version of the paper is a design and evaluation proposal: no experiments have been run, and all result tables are templates.

F. Organization of the paper

Section II reviews related work and identifies the research gap. Section III formalizes the problem and its scope, and Section IV states the objectives, research questions and hypotheses. Section V describes the methodology, Section VI the system architecture and Section VII the implementation. Section VIII sets out the experimental design, and Section IX fixes the structure in which results will be reported. Section X discusses limitations, threats to validity and ethical considerations, and Section XI concludes with future work.

II. LITERATURE REVIEW

A. Static and dynamic analysis

Static application security testing detects insecure API use, tainted data flowing to sensitive sinks, and violations of coding rules [2]. Its weakness is precision. Because the analysis over-approximates, it flags paths that cannot be exploited, and Johnson et al. report that false positives and poor presentation of warnings discourage developers from adopting such tools [3]. Taint analysis also depends on source and sink specifications for each library, which are

laborious to maintain; this is the gap that recent LLM-assisted work targets [13]. Dynamic analysis and fuzzing avoid false alarms on the crashes they trigger, but their coverage is bounded by the inputs generated [4].

B. Machine-learning and deep-learning detectors

Russell et al. trained convolutional and recurrent models on lexed C/C++ functions [5]. VulDeePecker introduced code gadgets, which are slices of semantically related statements around library calls, and fed them to a bidirectional LSTM [6]; SySeVR extended this idea to syntax- and semantics-based vulnerability candidates [14]. Devign represented each function as a joint graph of syntax, control-flow and data-flow edges and applied a gated graph neural network [7]. Chakraborty et al. then showed that such models drop sharply on realistic data and proposed ReVeal, a graph-based pipeline with representation learning to address class imbalance [15]. The Big-Vul dataset, built from CVE-linked fixing commits in open-source C/C++ projects, provided a larger resource [16].

C. Pretrained code transformers

CodeBERT is a RoBERTa-style encoder pretrained on paired natural language and code [8]. GraphCodeBERT adds data-flow structure to pretraining so that the model attends to variable relationships rather than only to token order [17]. LineVul fine-tunes CodeBERT for function-level prediction and uses attention scores to rank lines; it reported large F1 gains over earlier baselines on a dataset of more than 188,000 C/C++ functions [18]. VulBERTa pretrains a RoBERTa model on C/C++ code with a custom tokenizer for vulnerability tasks [19]. Steenhoek et al. compared several deep-learning detectors and found that results vary widely with the training data and the random seed, and that models struggle on certain vulnerability types [20].

D. Dataset quality

Croft et al. analyzed vulnerability datasets and documented label noise and duplication as recurring problems [21]. Ding et al. reported that duplication and label error undermine existing benchmarks, that up to 18.9% of test samples were leaked from the training set in some benchmarks, and that a model scoring 68.26% F1 on Big-Vul reached only 3.09% F1 on their cleaner PrimeVul dataset [11]. They also argued that accuracy is a poor metric because vulnerable code is rare, and proposed paired evaluation of a vulnerable function against its fixed version. These findings shape the evaluation design in Section VIII.

E. LLM-based detection

A recent survey organizes work on LLM-based vulnerability detection and repair and notes open issues around reliability and benchmarks [22]. Ullah et al. built SecLLMHolmes and evaluated eight LLMs on 228 code scenarios, concluding that the models gave non-deterministic outputs, had high false-positive rates, produced unfaithful reasoning, and were sensitive to simple code changes such as renaming or whitespace edits [12]. LLift uses an LLM to resolve uncertain static-analysis cases about use-before-initialization in the Linux kernel [23]. GPTScan combines GPT with static confirmation for logic vulnerabilities in smart contracts [24]. IRIS uses an LLM to infer taint specifications and filter false paths and then runs CodeQL; on CWE-Bench-Java with GPT-4 it detected 55 of 120 vulnerabilities against 27 for CodeQL, with a false discovery rate about five percentage points lower [13].

F. Comparison of existing approaches

Table I summarizes the approaches discussed above and the limitation of each that is relevant to this work.

TABLE I. COMPARISON OF EXISTING APPROACHES

Approach	Core technique	Granularity	Strength	Limitation relevant to this work
Static analysis [2], [3]	Pattern, taint and data-flow rules	Line	Precise on known patterns	False alarms; needs specifications
Fuzzing [4]	Input generation and execution	Crash site	Reports real failures	Coverage-limited; needs a harness

Approach	Core technique	Granularity	Strength	Limitation relevant to this work
VulDeePecker, SySeVR [6], [14]	Code gadgets with recurrent networks	Slice	Focuses on relevant statements	Limited semantic context
Devign, ReVeal [7], [15]	Graph neural networks	Function	Uses program structure	Large drop on realistic data [15]
CodeBERT, GraphCodeBERT, LineVul [8], [17], [18]	Fine-tuned transformer	Function, line	Strong benchmark scores	Sensitive to data leakage [11]
Prompted LLMs [12]	Zero-shot or few-shot prompting	Function	Natural-language explanations	Non-determinism; unfaithful reasoning [12]
IRIS [13]	LLM with CodeQL taint analysis	Project	Fewer false alarms than CodeQL alone	Java and CWE-specific; no learned prior or calibration
LLift, GPTScan [23], [24]	LLM with static checks	Targeted bug class	Focused verification	Narrow vulnerability scope

G. Research gap

The reviewed work leaves a specific gap. Fine-tuned detectors give scores without explanations and are evaluated on data whose quality is disputed [11]. Prompted LLMs give explanations that are not guaranteed to be faithful [12]. LLM-plus-static systems such as IRIS, LLift and GPTScan verify specific bug classes or languages [13], [23], [24]. In the sources reviewed here we did not find a general function-level framework that (i) requires the LLM to cite evidence that is mechanically checked, (ii) measures stability across sampled runs, (iii) fuses these signals with a fine-tuned prior and static support into a calibrated confidence, and (iv) is evaluated on leakage-controlled data with ablations. This is a statement about the sources reviewed, not a proof of absence, and the literature search should be repeated before submission to confirm it.

III. PROBLEM STATEMENT

A. Formal task definition

Let a code unit x be a function, optionally accompanied by its callers, callees and program slices. Let $y \in \{0, 1\}$ indicate whether x contains a vulnerability, and let c be its CWE type when $y = 1$. For each x the framework must produce a verdict $\hat{y}$, a CWE label $\hat{c}$, a set of evidence lines $E \subseteq \text{lines}(x)$, a natural-language rationale and a confidence $p \in [0, 1]$. The confidence should be calibrated, meaning that $P(y = 1 \mid p) \approx p$. Appendix B lists the notation used throughout the paper.

B. Problems in current approaches

1. **False positives and false negatives.** Static tools over-report, and learned models trade precision against recall depending on the decision threshold. Both kinds of error are costly in practice [3], [11].
2. **Contextual understanding.** Many vulnerabilities depend on code outside the function, such as a caller that validates its input. Function-only inputs lose this context.
3. **Complex vulnerability patterns.** Logic flaws and multi-step data flows are not captured by simple patterns or short token windows.

4. **Zero-day vulnerabilities.** Rule-based tools detect only what has a rule, and learned models may recognize memorized patterns rather than the underlying weakness [11].
5. **Limits of traditional static analysis.** Precision depends on specifications and approximations [3], [13].
6. **Reliability of LLM analysis.** Outputs vary between runs, may cite code that does not exist, and may justify verdicts incorrectly [12].

C. Scope and assumptions

The framework targets function-level analysis of C/C++ source code, with Python and Java as optional secondary targets. It assumes that the source code is available, that it can be parsed, and that the repository can be analyzed by Semgrep and CodeQL where a build or database can be created. Findings are reported as evidence-backed hypotheses for a human reviewer; the framework does not generate exploits and does not claim to prove the absence of vulnerabilities. The analyzed code is treated as untrusted input to the LLM, because comments or string literals may contain adversarial instructions (Section X-C).

IV. RESEARCH OBJECTIVES AND QUESTIONS

TABLE II. RESEARCH OBJECTIVES

ID	Objective (measurable)
O1	Build the hybrid pipeline for C/C++ (primary) and for Python or Java (secondary, if time permits).
O2	Reduce the false positive rate (FPR) relative to the LLM-only baseline at a stated recall, reported as ΔFPR with a confidence interval.
O3	Achieve a lower expected calibration error (ECE) for the fused score than for any single signal.
O4	Report cross-dataset performance (train on one dataset, test on another) and PrimeVul results with paired evaluation.
O5	Quantify each component's contribution through ablation.
O6	Report end-to-end latency per function and the cost of verification.

The corresponding research questions are: **RQ1**, does verification lower the false positive rate, and at what cost in recall; **RQ2**, is the fused confidence better calibrated than its components; **RQ3**, how well does the framework generalize across datasets and over time; and **RQ4**, which components matter most. Table III links each question to its objective, hypothesis and the evidence that will answer it.

TABLE III. TRACEABILITY OF RESEARCH QUESTIONS

RQ	Objective	Hypothesis	Primary evidence	Reported in
RQ1	O2	H1	FPR and recall against baseline B5; VD-S; paired metrics	Tables X, XI, XII
RQ2	O3	H2	ECE and Brier score of each signal and of the fused score; reliability diagrams	Table XIV; reliability figure
RQ3	O4	H3	In-dataset, cross-dataset and PrimeVul results against B3 and B4	Tables X, XI

RQ	Objective	Hypothesis	Primary evidence	Reported in
RQ4	O5	H4	Change in F1, FPR, ECE and latency when one component is removed	Table XIV

Objectives O1 and O6 concern the construction and cost of the system and are answered by the implementation and by the latency columns of the result tables.

V. PROPOSED METHODOLOGY

Fig. 1 gives an overview of the study. Data are prepared first, because leakage control must precede any model training. The components are then built and the verification layer and fusion are fitted on validation data only, after which the complete system and its baselines are evaluated once on the held-out test sets.

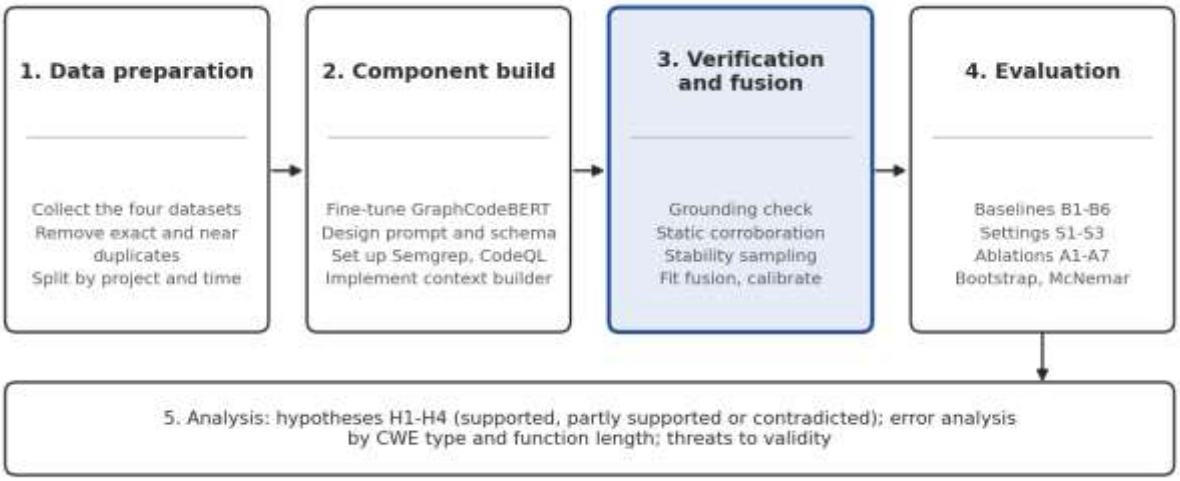


Fig. 1. Overview of the research workflow. The verification and fusion stage (highlighted) is the main design contribution.

A. Source code preprocessing

Files are parsed with Tree-sitter [25] to obtain function boundaries and abstract syntax trees for each supported language. Comments are retained as optional context but can be stripped in an ablation, because comments can mislead models. Exact and near-duplicate functions are removed using normalized-token hashing, and near duplicates are detected with a token-shingle Jaccard similarity above a chosen threshold. Normalization (consistent whitespace and optional identifier renaming) is applied to the encoder input only, so that the evidence line numbers produced in the LLM stage still refer to the original source. Splits are made by project and, where commit dates exist, chronologically, to limit leakage [11].

B. Tokenization and segmentation

The encoder uses the byte-pair tokenizer paired with the chosen checkpoint. Transformer encoders accept at most 512 tokens, so long functions are segmented with a sliding window of 512 tokens and a stride of 256, and the function-level score is the maximum window score. For the LLM, the whole file is not supplied. A context builder instead assembles the target function, the signatures of its callees, up to k_c callers found through the static call graph, and a backward program slice from any sink flagged by the static stage. If the assembled prompt exceeds a token budget B, slices are prioritized over unrelated surrounding code. This design responds to the difficulty that LLMs have with long inputs and with locating relevant content in the middle of a long context [26].

C. Language support

The primary target is C/C++, because Big-Vul, Devign, PrimeVul and Juliet cover it well. Python and Java are optional secondary targets, using Semgrep rules and the CodeQL packs for those languages. Because dataset support is weaker there, results for these languages are treated as exploratory.

D. Model selection and configuration

Two model roles are used. The **prior model** is GraphCodeBERT [17], with CodeBERT [8] as a baseline. It is fine-tuned for binary classification with a weighted cross-entropy loss over N training functions:

$$L = -(1/N)\,\Sigma_i\,[\,w_1\,y_i \log p_i + w_0\,(1-y_i)\log(1-p_i)\,] \qquad (1)$$

where w_1 and w_0 compensate for class imbalance. The **analysis model** is an open-source code LLM such as Code Llama [10] or StarCoder2 [27] with 7B to 15B parameters. Open weights are chosen so that the code never leaves local infrastructure and so that decoding parameters are fully controllable. Quantization (4-bit or 8-bit) is allowed to fit the available GPU memory, and its effect on accuracy is recorded.

E. Prompt engineering

The prompt has a fixed system role, the CWE catalogue entries for candidate types, the assembled context with numbered lines, and a required JSON schema (Table IV). Requiring verbatim code quotations alongside line numbers is what makes the grounding check in Section V-H mechanical. Few-shot examples include both the vulnerable and the fixed version of the same function, because paired contrast is the harder case [11], [12]. Chain-of-thought style reasoning [28] is evaluated as a variant, since its robustness is not guaranteed [12]. Decoding uses a low temperature for the primary run and a higher temperature for the k stability samples. Responses that do not parse against the schema are retried once; a second failure is recorded as an invalid output, and the invalid-output rate is reported.

TABLE IV. STRUCTURED OUTPUT REQUIRED FROM THE LLM

Field	Type	Content and use
verdict	enum	"vulnerable" or "not_vulnerable". Input to the vote share s_2 and to the status rule.
cwe	string	One identifier from the closed CWE set, or "unknown". Compared with analyzer rule mappings for s_3.
evidence_lines	list of integers	Line numbers in the supplied function. Checked for existence and overlap (v).
evidence_quotes	list of strings	Verbatim code at each cited line. Checked for an exact match after whitespace normalization.
dataflow	object	Source, transformation and sink, each with a line number. Searched for in the static data-flow results.
rationale	string	Short natural-language explanation shown to the reviewer. Not scored automatically.

F. Vulnerability classification

Classification is two-stage. Stage one is binary (vulnerable or not). Stage two assigns a CWE type from a closed set, for example the CWE Top 25 categories that occur in the datasets; constraining the label space reduces invalid outputs. An "unknown" option is allowed so that the model is not forced to guess.

G. Detection confidence scoring

Four signals are computed for each function. Let $(\hat{y}_i, \hat{c}_i)$ be the verdict and CWE label returned by run i, where run 0 is the primary run and runs 1 to k are the stability samples.

- $s_1 \in [0, 1]$: the GraphCodeBERT probability.
- $s_2 \in [0, 1]$: the vote share for the vulnerability claim, following the idea of aggregating multiple sampled reasoning paths [29], defined in Eq. (2).
- $s_3 \in \{0, 1\}$: static support. It equals 1 if CodeQL or Semgrep reports a finding with a compatible CWE inside the cited evidence region, or if the claimed source-to-sink path is confirmed by data-flow analysis. Compatibility is decided with a fixed mapping from analyzer rule identifiers to CWE identifiers that includes parent–child relations in the CWE hierarchy. When the primary verdict is "not_vulnerable", $s_3 = 1$ if the analyzers report any alert in the function, so that static tools can still raise suspicion.
- $v \in \{0, 1\}$: grounding, equal to 1 if every cited line exists in the function, the quoted code matches, and the lines overlap the flagged region. It is 0 when the primary verdict is "not_vulnerable".

$$s_2 = (1/(k+1)) \, \Sigma_{i=0}^{k} \, 1[\, \hat{y}_i = 1 \wedge \hat{c}_i = \hat{c}_{maj} \,] \quad (2)$$

where $\hat{c}_{maj}$ is the most frequent CWE label among the runs that report a vulnerability. Defining s_2 as the share of runs that support the claim, rather than the share that agree with the majority, avoids a design error in which a model that consistently answers "not vulnerable" would receive a high score. The fused confidence is

$$p = \sigma(\beta_0 + \beta_1 s_1 + \beta_2 s_2 + \beta_3 s_3 + \beta_4 v) \quad (3)$$

where σ is the logistic function. The weights β are fitted by regularized logistic regression on a first part of the validation set. A second, disjoint part is used to check calibration and, if the fused score is poorly calibrated, to fit temperature scaling or isotonic regression [30]; splitting the validation data in this way avoids fitting the weights and the calibration map on the same items. The final verdict $\hat{y}$ is 1 when $p \geq \theta$, where the threshold θ is selected on the validation set, for example to meet a target FPR. Calibration is evaluated by the expected calibration error over M equal-width confidence bins, and by the Brier score [31]:

$$\mathrm{ECE} = \Sigma_{m=1}^{M} \, (|B_m| / n) \cdot |\, \mathrm{acc}(B_m) - \mathrm{conf}(B_m) \,| \quad (4)$$

$$\mathrm{Brier} = (1/n) \, \Sigma_{i=1}^{n} \, (p_i - y_i)^2 \quad (5)$$

where, for the vulnerable class, $\mathrm{acc}(B_m)$ is the observed fraction of vulnerable functions in bin B_m and $\mathrm{conf}(B_m)$ is the mean predicted confidence in that bin. Fig. 2 shows how the signals are combined.

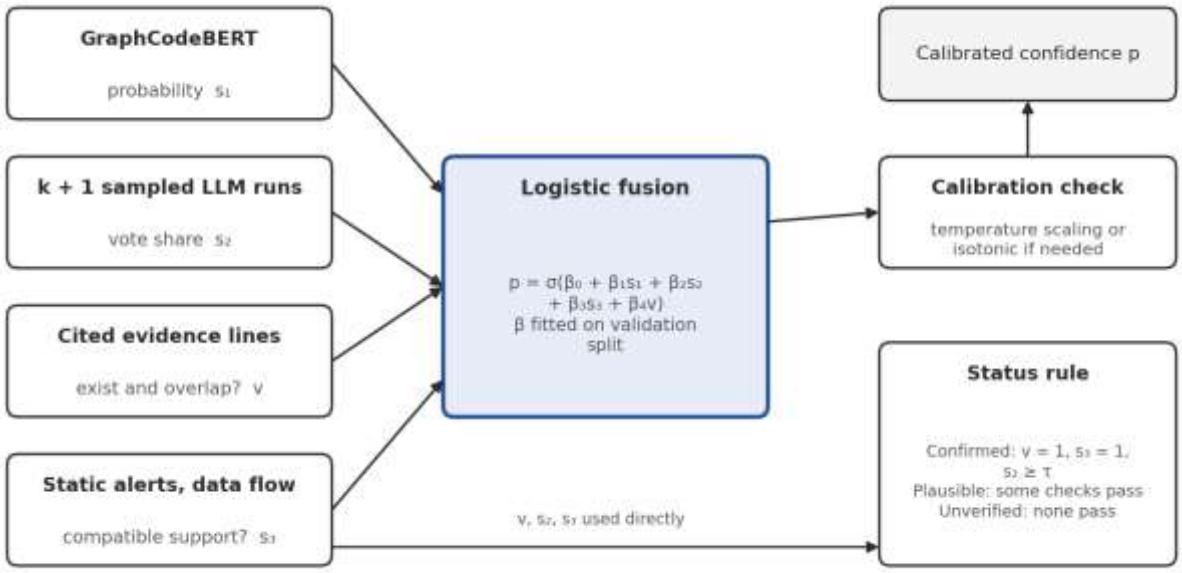


Fig. 2. Verification signals and confidence fusion. The status rule uses the grounding, stability and static-support signals directly, whereas the calibrated confidence uses all four signals and the encoder prior.

H. Validation and verification mechanisms

The verifier applies four checks. (1) **Grounding:** every cited line number must exist in the target function and the quoted code must match; failures indicate hallucinated evidence [32]. (2) **Static corroboration:** the claimed data-flow path is searched for with CodeQL taint queries or with a Semgrep rule matching the CWE. (3) **Stability:** the verdict is re-sampled k times, and low agreement lowers the confidence. (4) **Paired contrast:** where a fixed version is available (benchmark mode only), the framework must flag the vulnerable version and not the fixed one. Check 4 is an evaluation-time test and does not enter the status rule, because a fixed version is not available when scanning real code. Each finding that carries a vulnerability claim receives one of the statuses in Table V.

TABLE V. FINDING STATUS ASSIGNED BY THE VERIFIER

Status	Condition	Interpretation
Confirmed	$v = 1$, $s_3 = 1$ and $s_2 \geq \tau$	Evidence is grounded, supported by static analysis and stable across runs.
Plausible	Primary verdict is "vulnerable"; at least one, but not all, of the three conditions above hold	Partly supported. Needs reviewer attention.
Unverified	Primary verdict is "vulnerable"; none of the conditions hold	The claim is not backed by independent evidence.
No finding	Primary verdict is "not_vulnerable"	No claim to verify. The confidence p is still reported.

I. Algorithm

Algorithm 1 gives the complete procedure and Fig. 3 shows the same workflow as a flowchart.

Algorithm 1. Verification-grounded vulnerability analysis

```
Input:  function f, repository R, token budget B, samples k,
        stability threshold tau, decision threshold theta
Output: verdict, CWE, evidence E, rationale, confidence p, status

 1  (ast, callgraph) <- Parse(R)
 2  alerts <- RunStaticAnalysis(f, {Semgrep, CodeQL})
 3  ctx    <- BuildContext(f, callgraph, alerts, B)
 4  s1     <- GraphCodeBERT(Segment(f))        # max over windows
 5  out_0  <- LLM(ctx, temperature = low)      # primary finding
 6  for i = 1..k: out_i <- LLM(ctx, temperature = high)
 7  s2     <- VoteShare(out_0 .. out_k)        # Eq. (2)
 8  v      <- GroundingCheck(out_0.evidence, f)
 9  s3     <- StaticSupport(out_0.cwe, out_0.dataflow, alerts)
10  p      <- Calibrate(Sigmoid(b0 + b1*s1 + b2*s2 + b3*s3 + b4*v))
11  verdict <- 1 if p >= theta else 0
12  if out_0.verdict = not_vulnerable:   status <- NoFinding
    else if v and s3 and s2 >= tau:      status <- Confirmed
    else if v or s3 or s2 >= tau:        status <- Plausible
    else:                                status <- Unverified
13  return verdict, out_0, p, status
```

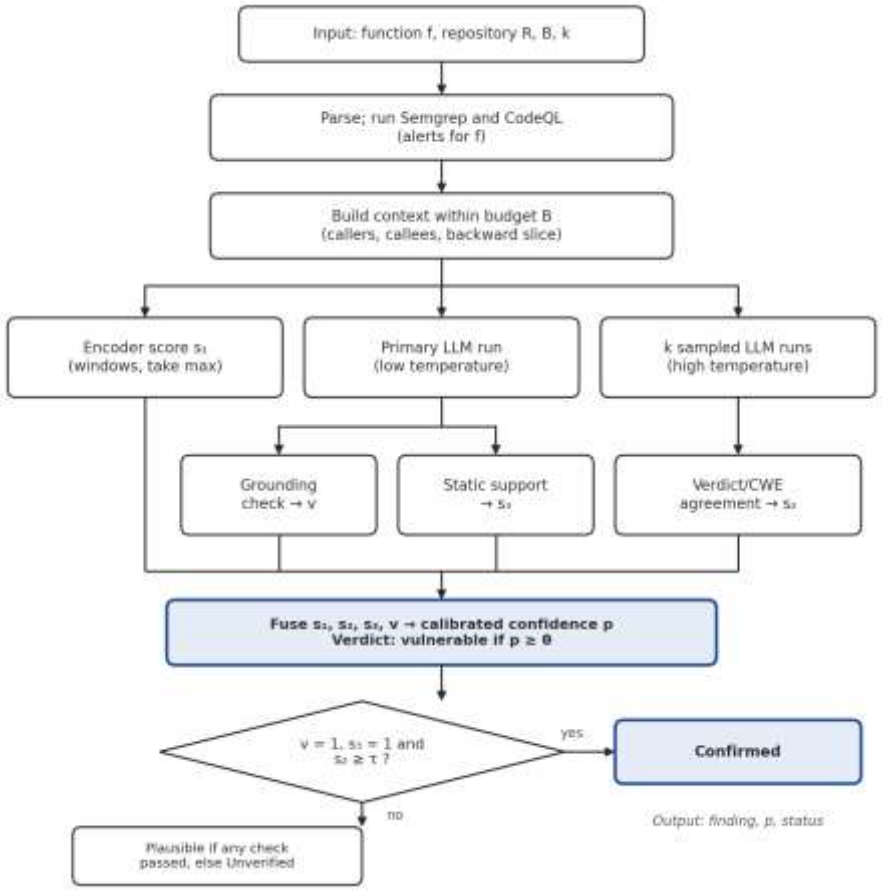


Fig. 3. Workflow of Algorithm 1. A finding is confirmed only if grounding, static support and stability all hold.

VI. PROPOSED SYSTEM ARCHITECTURE

The system is organized as six components with a one-directional data flow, so that each stage can be tested and replaced independently (Fig. 4).

1. **Ingestion and preprocessing service.** Accepts a repository, commit or file; parses it with Tree-sitter; deduplicates; and extracts functions, the call graph and dependencies.
2. **Static analysis service.** Runs Semgrep rules and CodeQL queries and stores the alerts with file, line and CWE.
3. **Context builder.** Combines the target function, callers, callees and slices within the token budget.
4. **Detection engine.** Contains the GraphCodeBERT prior and the LLM analyzer, which produces the structured finding and the k stability samples.
5. **Verification and scoring module.** Performs the checks of Section V-H and computes the calibrated confidence.
6. **Reporting layer.** Exposes results through a REST API and a simple dashboard that shows the verdict, the evidence lines highlighted in the code, the rationale, the confidence and the status.

Data flow. Source code enters component 1 and leaves as function units and a call graph. Component 2 produces alerts and component 3 produces the context. Component 4 turns the context into a prior score and LLM findings, component 5 turns these into a verified and scored finding, and component 6 presents the report. Only the verifier and the reporting layer consume all upstream outputs, which keeps the earlier stages loosely coupled.

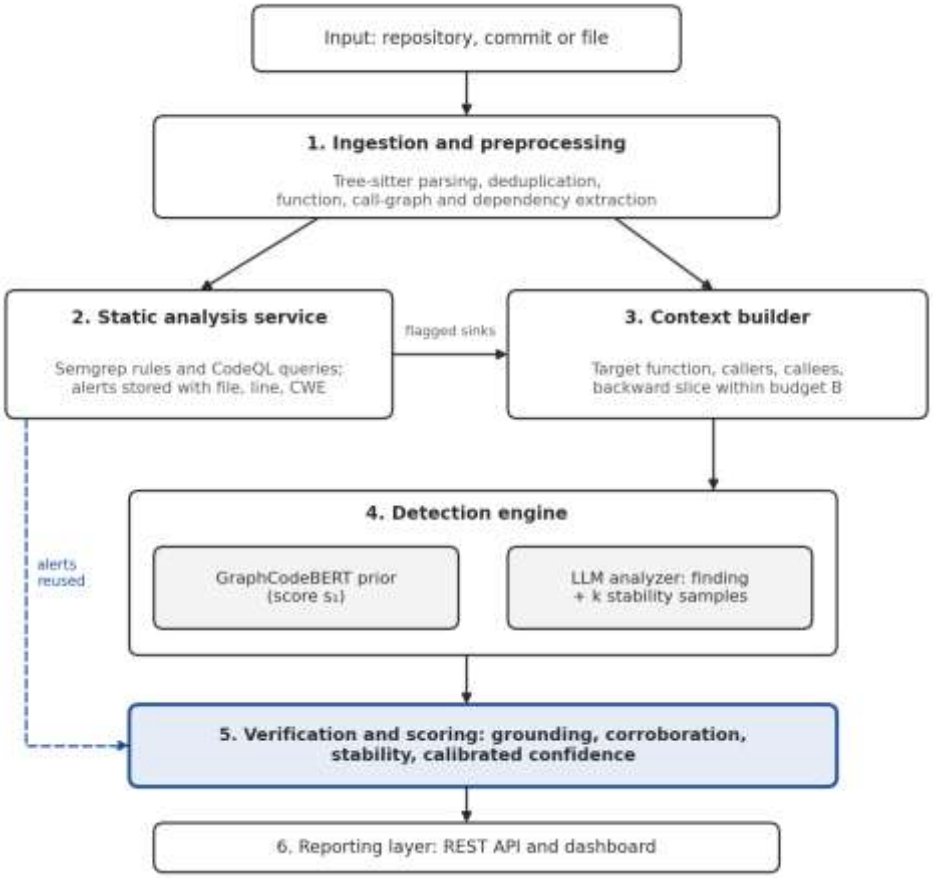


Fig. 4. System architecture. Data flows from top to bottom; the static-analysis alerts are also reused by the verifier (dashed arrow).

VII. IMPLEMENTATION DETAILS

A. Technology stack and rationale

TABLE VI. TECHNOLOGY STACK

Component	Choice	Why it is appropriate
Language	Python 3.11	Native ecosystem for PyTorch [33], Hugging Face Transformers [34] and static-analysis tooling; fast prototyping.
Service layer	FastAPI [35]	Typed request and response models, automatic OpenAPI documentation, and asynchronous endpoints suited to long-running analysis jobs.
Parsing	Tree-sitter [25]	Error-tolerant incremental parsers for many languages; gives function boundaries and ASTs without a full build.
Prior model	GraphCodeBERT [17]; CodeBERT [8] as baseline	Data-flow-aware pretraining suits reasoning about variable dependencies, and a CodeBERT-based detector is an established fine-tuning baseline [18].
Analysis LLM	Code Llama [10] or StarCoder2 [27] (7B–15B)	Open weights allow local deployment, so source code stays on-premises; decoding settings are controllable and experiments are reproducible.

Component	Choice	Why it is appropriate
Static analysis	Semgrep [36], CodeQL [37]	Semgrep gives fast pattern rules; CodeQL gives inter-procedural taint queries. Their differing precision and recall profiles make them complementary. CodeQL licence terms must be checked for the intended use.
Model serving	vLLM [38] or llama.cpp	Efficient batched inference; quantized runs on limited GPU memory.
Storage	SQLite or PostgreSQL	Stores findings, signals and run metadata for reproducibility.
Packaging	Docker	Pins the versions of Semgrep, CodeQL and the model checkpoints for repeatable runs.

B. Datasets

Table VII summarizes the four datasets and their roles. Each dataset is deduplicated before splitting, and the number of removed samples is reported.

TABLE VII. DATASETS AND THEIR ROLES

Dataset	Language	Labels and structure	Role in this study	Known concerns
Big-Vul [16]	C/C++	Functions linked to CVE fixing commits; large and widely used	Training; in-dataset test (S1); comparison with prior work	Label noise and duplication [11], [21]
Devign [7]	C	Function-level labels; the commonly used public release covers FFmpeg and QEMU [39]	Cross-dataset test (S2)	Few projects, so project-level splits are limited; label noise [21]
PrimeVul [11]	C/C++	Deduplicated; chronological split; paired vulnerable and fixed functions	Primary evaluation set (S2, S3, ablations)	Built to reduce the optimism of older benchmarks, but still limited to C/C++
Juliet [40]	C/C++, Java	Synthetic cases with known CWE labels and good/bad variants	Sanity check and CWE classification ground truth	Artificial and regular; not evidence of real-world performance

C. Configuration to be recorded

Every run logs the model checkpoint and revision, the quantization level, the prompt template version, the decoding parameters (temperature, top-p and maximum new tokens), the number of stability samples k, the token budget B, the Semgrep rule-set version, the CodeQL version and query packs, the random seeds and the hardware. These values will be reported in the final paper so that the results can be reproduced.

VIII. EXPERIMENTAL DESIGN AND EVALUATION

A. Data splits

For Big-Vul, a project-level split is used so that no project appears in more than one partition (for example 80% training, 10% validation and 10% test, with the proportions fixed after deduplication). For PrimeVul, the chronological

split provided by its authors is used [11]. The validation set is used for early stopping, for selecting the decision threshold θ, for fitting the fusion weights β and for fitting any calibration map, using the two disjoint parts described in Section V-G. The test set is used once, for final reporting. Three evaluation settings are run (Fig. 5): **S1**, an in-dataset test; **S2**, a cross-dataset test, such as training on Big-Vul and testing on Devign and PrimeVul; and **S3**, a paired evaluation on PrimeVul in which each vulnerable function is paired with its fixed version.

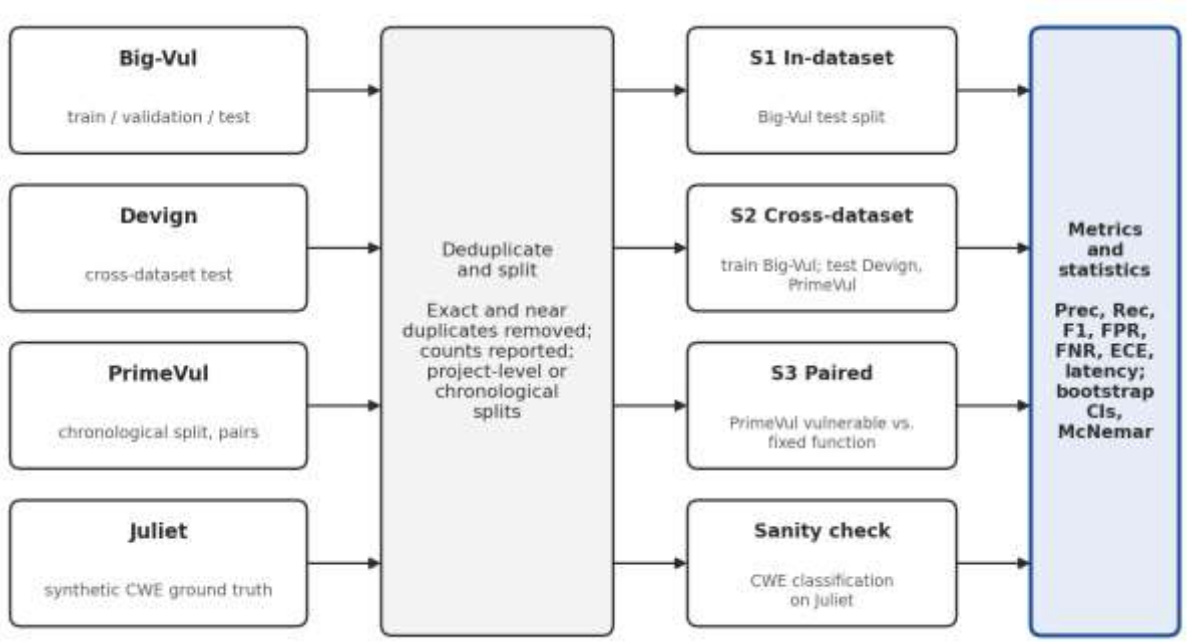


Fig. 5. Evaluation protocol. All datasets are deduplicated before splitting, and every setting reports the same metrics.

B. Baselines

TABLE VIII. BASELINE SYSTEMS

ID	Baseline	Purpose
B1	Semgrep (standard security rule sets) [36]	Pattern-based static baseline
B2	CodeQL (security queries) [37]	Taint-based static baseline
B3	Fine-tuned CodeBERT (LineVul-style) [8], [18]	Established transformer baseline
B4	Fine-tuned GraphCodeBERT [17]	The prior model on its own
B5	Prompt-only LLM (same model as the framework, no verification)	Isolates the effect of verification
B6	Prompt-only LLM with self-consistency voting	Tests whether stability alone suffices

C. Metrics

Let TP, FP, TN and FN denote the confusion-matrix counts with *vulnerable* as the positive class. Table IX lists the metrics. Because vulnerable code is rare in practice, accuracy is reported but not used as a headline metric [11]. Where

the class balance of a benchmark differs from the real world, results are also reported on a reweighted test set with a realistic vulnerable-to-benign ratio, and that ratio is stated.

TABLE IX. EVALUATION METRICS

Metric	Definition	Purpose
Precision	TP / (TP + FP)	Share of alerts that are real
Recall (TPR)	TP / (TP + FN)	Share of vulnerabilities found
F1	2 · Precision · Recall / (Precision + Recall)	Balance of the two
Accuracy	(TP + TN) / (TP + TN + FP + FN)	Reported only; misleading under imbalance
FPR	FP / (FP + TN)	Triage burden on benign code
FNR	FN / (FN + TP)	Missed vulnerabilities
VD-S [11]	FNR at a fixed low FPR	Detection quality at a tolerable alert rate
Paired metrics [11]	P-C: both versions classified correctly; P-V: both labeled vulnerable; P-B: both labeled benign; P-R: labels reversed	Whether the model separates a flaw from its fix
ECE, Brier score	Eqs. (4) and (5)	Calibration of confidence
CWE accuracy, macro-F1	Agreement of ĉ with the labeled CWE (Juliet; datasets with CWE labels)	Quality of type assignment
Evidence localization	Share of vulnerable functions with at least one cited line on a line changed by the fixing commit; precision of cited lines	Quality of evidence lines. Fix-changed lines are only a proxy, because fixing commits can also modify unrelated lines [11]
Grounding failure rate	Share of findings with v = 0 among claims of vulnerability	Frequency of hallucinated evidence
Invalid-output rate	Share of runs that fail schema parsing after one retry	Reliability of structured output
Latency	Wall-clock time per function, median and 95th percentile	Practical cost

D. Statistical analysis

Each learned component is trained with at least five random seeds, and the mean and standard deviation are reported. Confidence intervals for F1 and FPR are computed by bootstrap resampling of test items [41]. Differences between systems evaluated on the same test items are tested with McNemar's test [42], and p-values are adjusted for multiple comparisons with the Holm procedure [43]. LLM runs use fixed decoding settings and k samples, so that run-to-run variability is part of the reported result instead of being hidden.

E. Ablation plan

Starting from the full framework, one component is removed at a time: (A1) no static corroboration, (A2) no grounding check, (A3) no self-consistency, (A4) no encoder prior, (A5) no context builder (function only), (A6) no paired few-

shot examples, and (A7) fusion replaced by a simple average. Each row of the ablation table reports the change in F1, FPR, ECE and latency.

F. Hypotheses

- **H1.** Verification lowers the FPR relative to B5, at a reduction in recall that is small compared with the FPR gain.
- **H2.** The fused confidence has a lower ECE than any single signal.
- **H3.** Performance drops from the in-dataset setting to the cross-dataset setting and to PrimeVul, but the framework drops less than B3 and B4.
- **H4.** Static corroboration and grounding contribute the largest share of the FPR reduction.

These are hypotheses, not findings. If the data do not support them, the paper will report that outcome.

IX. RESULTS AND DISCUSSION (STRUCTURE AND PLACEHOLDERS)

No experiments have been run. The tables below fix the reporting structure, and "[TBD]" marks values that will be filled with measured results.

TABLE X. IN-DATASET PERFORMANCE (S1), BIG-VUL TEST SPLIT

System	Prec.	Recall	F1	Acc.	FPR	FNR	Latency (s)
B1 Semgrep	[TBD]	[TBD]	[TBD]	[TBD]	[TBD]	[TBD]	[TBD]
B2 CodeQL	[TBD]	[TBD]	[TBD]	[TBD]	[TBD]	[TBD]	[TBD]
B3 CodeBERT	[TBD]	[TBD]	[TBD]	[TBD]	[TBD]	[TBD]	[TBD]
B4 GraphCodeBERT	[TBD]	[TBD]	[TBD]	[TBD]	[TBD]	[TBD]	[TBD]
B5 LLM only	[TBD]	[TBD]	[TBD]	[TBD]	[TBD]	[TBD]	[TBD]
B6 LLM + voting	[TBD]	[TBD]	[TBD]	[TBD]	[TBD]	[TBD]	[TBD]
Proposed framework	[TBD]	[TBD]	[TBD]	[TBD]	[TBD]	[TBD]	[TBD]

TABLE XI. CROSS-DATASET AND PAIRED PERFORMANCE (S2, S3)

Train → test	System	F1	FPR	FNR	Paired P-C	Paired P-R
Big-Vul → Devign	B4	[TBD]	[TBD]	[TBD]	n/a	n/a
Big-Vul → Devign	Proposed	[TBD]	[TBD]	[TBD]	n/a	n/a
Big-Vul → PrimeVul	B4	[TBD]	[TBD]	[TBD]	[TBD]	[TBD]
Big-Vul → PrimeVul	Proposed	[TBD]	[TBD]	[TBD]	[TBD]	[TBD]

TABLE XII. VERIFICATION STATUS OF FLAGGED FUNCTIONS (PRIMEVUL TEST)

Status	Share of flagged functions	Precision in group	Median confidence p
Confirmed	[TBD]	[TBD]	[TBD]
Plausible	[TBD]	[TBD]	[TBD]
Unverified	[TBD]	[TBD]	[TBD]

TABLE XIII. CWE CLASSIFICATION, EVIDENCE QUALITY AND OUTPUT RELIABILITY

Evaluation set	Metric	B5 (LLM only)	Proposed
Juliet	CWE accuracy; macro-F1	[TBD]	[TBD]
Big-Vul test	Evidence localization (hit rate; line precision)	[TBD]	[TBD]
PrimeVul test	Grounding failure rate	[TBD]	[TBD]
All sets	Invalid-output rate	[TBD]	[TBD]

TABLE XIV. ABLATION (PRIMEVUL TEST)

Variant	ΔF1	ΔFPR	ECE	Latency (s)
Full framework	—	—	[TBD]	[TBD]
A1 no static corroboration	[TBD]	[TBD]	[TBD]	[TBD]
A2 no grounding check	[TBD]	[TBD]	[TBD]	[TBD]
A3 no self-consistency	[TBD]	[TBD]	[TBD]	[TBD]
A4 no encoder prior	[TBD]	[TBD]	[TBD]	[TBD]
A5 no context builder	[TBD]	[TBD]	[TBD]	[TBD]
A6 no paired few-shot examples	[TBD]	[TBD]	[TBD]	[TBD]
A7 simple-average fusion	[TBD]	[TBD]	[TBD]	[TBD]

Planned figures. (i) Precision–recall curves for all systems on PrimeVul. (ii) A reliability diagram of confidence against observed accuracy for each signal and for the fused score, read as in Fig. 6. (iii) A bar chart of FPR and FNR by CWE category. (iv) The latency distribution (median and 95th percentile) by pipeline stage. (v) A coverage–precision curve showing precision as low-confidence findings are withheld.

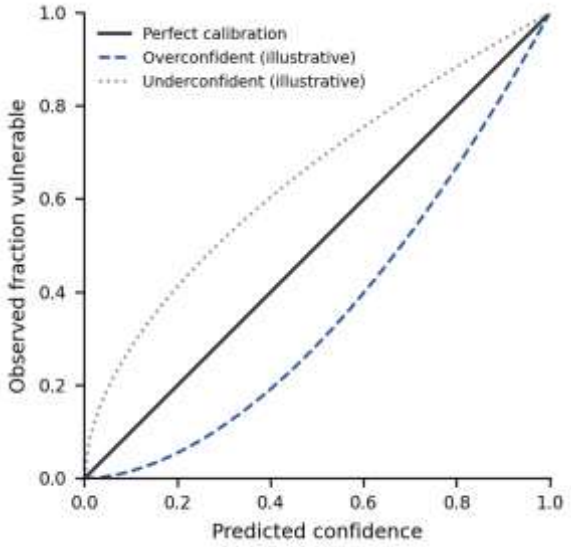


Fig. 6. How the planned reliability diagram will be read. The curves are schematic and contain no measured data. Points below the diagonal indicate overconfidence, and points above it indicate underconfidence.

Discussion structure. The discussion will (1) compare each system's F1 with its FPR and FNR rather than F1 alone; (2) report, for H1 to H4, whether the data support, partly support or contradict the hypothesis; (3) analyze errors by

CWE type and function length, with a manual review of a random sample of false positives and false negatives and a count of how many grounding failures were hallucinated evidence; (4) examine whether verification changes only the confidence of a finding or also its correctness; and (5) relate the numbers to the dataset-quality concerns in [11], including how much of the benchmark gap is explained by leakage.

X. LIMITATIONS, THREATS TO VALIDITY AND ETHICAL CONSIDERATIONS

A. Limitations and planned mitigations

TABLE XV. LIMITATIONS AND PLANNED MITIGATIONS

Limitation	Effect	Planned mitigation
LLM hallucination	Models can cite non-existent lines or invent data flows [12], [32]. Grounding catches fabricated line references but not plausible reasoning about real lines.	Static corroboration and stability checks; manual review of sampled errors. Benefit is limited to patterns the analyzers can express.
Model size	Models of 7B–15B parameters may reason less well than larger proprietary models, and results depend on the checkpoint and prompt.	Report the checkpoint and prompt version; test more than one model where resources allow. Findings may not transfer to other models.
Computational cost	Running k + 1 LLM samples per function plus static analysis increases latency and memory use.	Report the cost–accuracy trade-off; consider a cheaper first-stage filter for large-scale scanning.
Dataset bias	Big-Vul and Devign contain label noise and duplicates [11], [21]; Juliet is synthetic; public data mostly cover C/C++ and few CWE types.	Deduplication, PrimeVul as primary set, Juliet only as a sanity check; treat other languages as exploratory.
Language specificity	Memory-safety weaknesses dominate the C/C++ datasets, whereas injection and access-control flaws dominate web languages.	Per-language rules, slicing and prompts.
Context window limits	Cross-file and project-level reasoning can exceed the token budget, and long inputs can degrade model attention [26].	Slice-based context; vulnerabilities spanning many files may still be missed.

B. Threats to validity

- **Internal validity.** Hyperparameters, prompts and thresholds are tuned on validation data. The test sets are used once, and every tuned choice is logged.
- **External validity.** The study covers a limited set of languages and datasets, so the results may not generalize to other languages or to logic flaws.
- **Construct validity.** A function label may be wrong, or a flagged function may be vulnerable in a way that the label does not record. Fix-changed lines are an imperfect proxy for the vulnerable lines.
- **Training-data contamination.** Big-Vul, Devign and the open-source projects behind them are public and may appear in the pretraining data of the analysis LLM. Memorization could therefore inflate LLM results. The chronological PrimeVul split limits this risk but cannot remove it, so model release dates are compared with dataset cut-off dates and the issue is discussed alongside the results.

C. Security, privacy and ethical considerations

Source code can be confidential. Local open models avoid sending code to external services, but logs and stored findings still contain code and must be access-controlled. The LLM prompt can also be attacked through malicious comments or strings in the analyzed code, a form of indirect prompt injection [44]. The system therefore treats the analyzed code as untrusted input: the model has no tool access beyond the analysis pipeline, outputs are validated against the schema, and verification does not rely on the model's own statements. Vulnerability detection is also dual-use. The framework produces evidence for a reviewer and does not generate exploits, and any previously unknown vulnerability found in open-source code during the study will be reported privately to the maintainers before it is described publicly.

XI. CONCLUSION AND FUTURE SCOPE

This paper proposed a verification-grounded hybrid framework for source-code vulnerability detection. It combines static-analysis context, a fine-tuned GraphCodeBERT prior and a structured LLM finding, and it checks the LLM's claims through evidence grounding, static corroboration and sampled-run stability before producing a calibrated confidence. The paper also specified a leakage-aware evaluation protocol covering Big-Vul, Devign, PrimeVul and Juliet, a set of baselines and ablations, and four hypotheses. The contribution is the design and the evaluation plan, and no performance claims are made until the experiments are completed.

Future work includes extending the context builder to repository-level analysis, learning the fusion function from richer features, testing larger and more recent open models, adding automated patch suggestions that are validated by re-running the detectors, and evaluating robustness to semantics-preserving code changes of the kind that disrupted LLMs in earlier work [12]. A user study on whether confidence scores and evidence presentation reduce developer triage time would also test the practical value of the approach.

DATA AND CODE AVAILABILITY

The implementation, prompt templates, configuration files and evaluation scripts will be released at [repository URL to be added] once the experiments are complete. The datasets used are public and are cited in Section VII-B; the exact splits and the list of removed duplicates will be released with the code.

REFERENCES

[1] MITRE, "Common Weakness Enumeration (CWE)." [Online]. Available: https://cwe.mitre.org/ (accessed: [date to be added]).

[2] B. Chess and G. McGraw, "Static analysis for security," *IEEE Security & Privacy*, vol. 2, no. 6, pp. 76–79, 2004, doi: 10.1109/MSP.2004.111.

[3] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?," in *Proc. 35th Int. Conf. Software Engineering (ICSE)*, 2013, pp. 672–681, doi: 10.1109/ICSE.2013.6606613.

[4] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, 1990, doi: 10.1145/96267.96279.

[5] R. Russell *et al.*, "Automated vulnerability detection in source code using deep representation learning," in *Proc. 17th IEEE Int. Conf. Machine Learning and Applications (ICMLA)*, 2018, arXiv:1807.04320.

[6] Z. Li *et al.*, "VulDeePecker: A deep learning-based system for vulnerability detection," in *Proc. Network and Distributed System Security Symp. (NDSS)*, 2018, doi: 10.14722/ndss.2018.23158.

[7] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 32, 2019, arXiv:1909.03496.

[8] Z. Feng *et al.*, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, 2020, doi: 10.18653/v1/2020.findings-emnlp.139.

[9] M. Chen *et al.*, "Evaluating large language models trained on code," arXiv:2107.03374, 2021.

[10] B. Rozière *et al.*, "Code Llama: Open foundation models for code," arXiv:2308.12950, 2023.

[11] Y. Ding *et al.*, "Vulnerability detection with code language models: How far are we?," in *Proc. IEEE/ACM 47th Int. Conf. Software Engineering (ICSE)*, 2025, pp. 1729–1741, doi: 10.1109/ICSE55347.2025.00038.

[12] S. Ullah, M. Han, S. Pujar, H. Pearce, A. Coskun, and G. Stringhini, "LLMs cannot reliably identify and reason about security vulnerabilities (yet?): A comprehensive evaluation, framework, and benchmarks," in *Proc. IEEE Symp. Security and Privacy (SP)*, 2024, pp. 862–880, doi: 10.1109/SP54263.2024.00210.

[13] Z. Li, S. Dutta, and M. Naik, "IRIS: LLM-assisted static analysis for detecting security vulnerabilities," in *Proc. Int. Conf. Learning Representations (ICLR)*, 2025, arXiv:2405.17238.

[14] Z. Li *et al.*, "SySeVR: A framework for using deep learning to detect software vulnerabilities," *IEEE Trans. Dependable and Secure Computing*, 2021, arXiv:1807.06756.

[15] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet?," *IEEE Trans. Software Engineering*, vol. 48, no. 9, 2022, doi: 10.1109/TSE.2021.3087402.

[16] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "A C/C++ code vulnerability dataset with code changes and CVE summaries," in *Proc. 17th Int. Conf. Mining Software Repositories (MSR)*, 2020, pp. 508–512, doi: 10.1145/3379597.3387501.

[17] D. Guo *et al.*, "GraphCodeBERT: Pre-training code representations with data flow," in *Proc. Int. Conf. Learning Representations (ICLR)*, 2021, arXiv:2009.08366.

[18] M. Fu and C. Tantithamthavorn, "LineVul: A transformer-based line-level vulnerability prediction," in *Proc. 19th Int. Conf. Mining Software Repositories (MSR)*, 2022, pp. 608–620, doi: 10.1145/3524842.3528452.

[19] H. Hanif and S. Maffeis, "VulBERTa: Simplified source code pre-training for vulnerability detection," in *Proc. Int. Joint Conf. Neural Networks (IJCNN)*, 2022, arXiv:2205.12424.

[20] B. Steenhoek, M. M. Rahman, R. Jiles, and W. Le, "An empirical study of deep learning models for vulnerability detection," in *Proc. 45th Int. Conf. Software Engineering (ICSE)*, 2023, pp. 2237–2248, arXiv:2212.08109.

[21] R. Croft, Y. Xie, and M. A. Babar, "Data quality for software vulnerability datasets," in *Proc. 45th Int. Conf. Software Engineering (ICSE)*, 2023, arXiv:2301.05456.

[22] X. Zhou, S. Cao, X. Sun, and D. Lo, "Large language model for vulnerability detection and repair: Literature review and the road ahead," arXiv:2404.02525, 2024.

[23] H. Li, Y. Hao, Y. Zhai, and Z. Qian, "The Hitchhiker's guide to program analysis: A journey with large language models," arXiv:2308.00245, 2023.

[24] Y. Sun *et al.*, "GPTScan: Detecting logic vulnerabilities in smart contracts by combining GPT with program analysis," in *Proc. 46th Int. Conf. Software Engineering (ICSE)*, 2024, arXiv:2308.03314.

[25] "Tree-sitter documentation." [Online]. Available: https://tree-sitter.github.io/tree-sitter/ (accessed: [date to be added]).

[26] N. F. Liu *et al.*, "Lost in the middle: How language models use long contexts," *Trans. Association for Computational Linguistics*, vol. 12, 2024, arXiv:2307.03172.

[27] A. Lozhkov *et al.*, "StarCoder 2 and The Stack v2: The next generation," arXiv:2402.19173, 2024.

[28] J. Wei *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 35, 2022, arXiv:2201.11903.

[29] X. Wang *et al.*, "Self-consistency improves chain of thought reasoning in language models," in *Proc. Int. Conf. Learning Representations (ICLR)*, 2023, arXiv:2203.11171.

[30] C. Guo, G. Pleiss, Y. Sun, and K. Q. Weinberger, "On calibration of modern neural networks," in *Proc. 34th Int. Conf. Machine Learning (ICML)*, 2017, arXiv:1706.04599.

[31] G. W. Brier, "Verification of forecasts expressed in terms of probability," *Monthly Weather Review*, vol. 78, no. 1, pp. 1–3, 1950.

[32] Z. Ji *et al.*, "Survey of hallucination in natural language generation," *ACM Computing Surveys*, vol. 55, no. 12, 2023, doi: 10.1145/3571730.

[33] A. Paszke *et al.*, "PyTorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 32, 2019.

[34] T. Wolf *et al.*, "Transformers: State-of-the-art natural language processing," in *Proc. Conf. Empirical Methods in Natural Language Processing: System Demonstrations (EMNLP)*, 2020, pp. 38–45.

[35] S. Ramírez, "FastAPI documentation." [Online]. Available: https://fastapi.tiangolo.com/ (accessed: [date to be added]).

[36] "Semgrep documentation." [Online]. Available: https://semgrep.dev/docs/ (accessed: [date to be added]).

[37] GitHub, "CodeQL documentation." [Online]. Available: https://codeql.github.com/docs/ (accessed: [date to be added]).

[38] W. Kwon *et al.*, "Efficient memory management for large language model serving with PagedAttention," in *Proc. 29th ACM Symp. Operating Systems Principles (SOSP)*, 2023, arXiv:2309.06180.

[39] S. Lu *et al.*, "CodeXGLUE: A machine learning benchmark dataset for code understanding and generation," in *Proc. NeurIPS Datasets and Benchmarks Track*, 2021, arXiv:2102.04664.

[40] T. Boland and P. E. Black, "Juliet 1.1 C/C++ and Java test suite," *Computer*, vol. 45, no. 10, pp. 88–90, 2012, doi: 10.1109/MC.2012.345.

[41] B. Efron, "Bootstrap methods: Another look at the jackknife," *The Annals of Statistics*, vol. 7, no. 1, pp. 1–26, 1979.

[42] Q. McNemar, "Note on the sampling error of the difference between correlated proportions or percentages," *Psychometrika*, vol. 12, no. 2, pp. 153–157, 1947.

[43] S. Holm, "A simple sequentially rejective multiple test procedure," *Scandinavian Journal of Statistics*, vol. 6, no. 2, pp. 65–70, 1979.

[44] K. Greshake, S. Abdelnabi, S. Mishra, C. Endres, T. Holz, and M. Fritz, "Not what you've signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection," in *Proc. 16th ACM Workshop on Artificial Intelligence and Security (AISec)*, 2023, arXiv:2302.12173.

APPENDIX A. ABBREVIATIONS

TABLE XVI. ABBREVIATIONS

Term	Meaning	Term	Meaning
AST	Abstract syntax tree	FNR	False negative rate
CVE	Common Vulnerabilities and Exposures	FPR	False positive rate
CWE	Common Weakness Enumeration	LLM	Large language model
ECE	Expected calibration error	TPR	True positive rate (recall)
VD-S	Vulnerability detection score [11]	REST	Representational state transfer

APPENDIX B. NOTATION

TABLE XVII. NOTATION

Symbol	Meaning
x, y, c	Code unit (function with optional context); vulnerability label in {0, 1}; CWE type of a vulnerable unit
$\hat{y}$, $\hat{c}$, E	Predicted verdict, predicted CWE label, set of cited evidence lines
k, B	Number of stability samples; token budget for the LLM context
s_1	Encoder probability from GraphCodeBERT
s_2	Vote share for the vulnerability claim across $k + 1$ runs, Eq. (2)
s_3	Static support, 1 if analyzers corroborate the claim
v	Grounding indicator, 1 if cited evidence exists, matches and overlaps the flagged region
$\beta_0 \ldots \beta_4$	Fusion weights fitted on the validation set, Eq. (3)
p	Calibrated confidence that the function is vulnerable
τ, θ	Stability threshold for the "Confirmed" status; decision threshold on p for the final verdict
M, B_m	Number of calibration bins; the set of items in bin m

APPENDIX C. ILLUSTRATIVE STRUCTURED OUTPUT

The following example shows the form of the JSON object required from the LLM (Table IV). The values are invented for illustration only and do not come from any experiment.

Listing 1. Example of the required output schema (illustrative values)

```
{
 "verdict": "vulnerable",
 "cwe": "CWE-787",
 "evidence_lines": [12, 14],
 "evidence_quotes": ["char buf[16];", "strcpy(buf, input);"],
 "dataflow": {
  "source": {"line": 10, "text": "input (function parameter)"},
  "transformation": "none",
  "sink": {"line": 14, "text": "strcpy(buf, input)"}
 },
 "rationale": "Unbounded copy of caller-controlled input into a
          16-byte stack buffer."
}
```