CYBERSECURITY AND COMPUTER SCIENCE

An Explainable, Validation-Oriented LLM–SAST Hybrid Framework for Vulnerability Detection, Classification, Risk Assessment and Secure-Code Remediation: Design and Pre-Registered Evaluation Protocol

Aditya Parag Lohokare[1], Rutuja Tanawde[1], Sarjerao Ghatge[2]

[1]Department of Computer Applications and Management, Navsahyadri Group of Institutes, Savitribai Phule Pune University, Pune, Maharashtra, India
[2]Guide, Department of Computer Applications and Management, Navsahyadri Group of Institutes, Savitribai Phule Pune University, Pune, Maharashtra, India

Published November 2026

Abstract

Language models have become good at reading, explaining and rewriting code, so it is no surprise that people want to use them for security work. The trouble is that most studies tune a single stage of the job, usually detection, and report numbers from benchmarks that flatter the result. We describe a framework that treats vulnerability analysis as a chain of stages, each of which must produce evidence that the next stage can check: detection, correlation, CWE classification, risk scoring, explanation, patch generation and validation. A conventional static analyser and an LLM examine the same code side by side, and their findings are merged without losing track of who found what. Severity comes from CVSS-style metrics instead of the model's own confidence, explanations are scored against a rubric, and a patch counts only if it compiles, passes the tests, stops triggering re-analysis and actually touches the cited root cause. We also fix the evaluation in advance: PrimeVul as the benchmark, frozen prompts and baselines, six experiments, paired statistics, and a decision rule written before any test result exists. Eight research questions and four hypotheses guide the plan. This is a design and protocol paper. We report no experimental results, and every results table is an empty template.

Keywords: vulnerability detection; large language models; static analysis; explainable AI; automated vulnerability repair; CWE; patch validation; reproducibility

1 Introduction

Software today is mostly assembled: our own code, third-party libraries, cloud services and build pipelines, glued together at interfaces where a small mistake can turn into a security hole. Injection, memory corruption and broken access control keep appearing among the most exploited weakness classes [1], and a flaw caught during development is generally far cheaper to fix than one found after release. That is the basic case for analysing code for vulnerabilities early and automatically.

Static application security testing (SAST) has been the workhorse for decades. A SAST tool parses the code, rebuilds control and data flow, and matches what it finds against rules for dangerous patterns. It is deterministic, fast and easy to slot into CI/CD pipelines, which is why any new method has to be compared against it. But rules see patterns, and many vulnerabilities are about intent. Whether a length check is enough depends on how the buffer is used later, and whether a query can be injected

depends on a sanitiser somewhere else entirely. The outcome will be familiar to anyone who has used these tools: false positives that teach developers to ignore warnings, and false negatives that give a comfort nobody has earned.

Large language models (LLMs) change the picture because they read code more like a reviewer does. They can say what looks wrong, name the weakness class, guess at the consequence and sketch a fix, and the surveys of this area are multiplying quickly [2], [3]. Two things keep us from simply trusting them. First, an LLM can write a confident, well-organised explanation that is wrong, or a patch that compiles and still leaves the hole open. Second, the evaluations are often too kind. PrimeVul showed that duplicated functions, noisy labels and random splits in popular benchmarks had inflated results, and that models scored far lower once the data was deduplicated and split by time [4]. A recent systematic review also finds that, of detection, repair and explanation, explanation is the least developed [5].

As a result, most work improves one stage at a time. Detectors say a function is vulnerable without saying why. Classification is treated as a side effect of detection. Risk gets confused with how sure the model is. Patches are called fixes because they compile. What we have not found is a single methodology in which the LLM is one more source of evidence beside deterministic analysis, rather than an oracle, and in which every claim, from the first alarm to the final patch, is checked by something other than the model that produced it.

This paper makes six contributions:

- **C1 – Unified framework.** A seven-layer architecture integrating SAST and LLM analysis, CWE classification [6], CVSS-informed risk assessment [7], structured explanation, remediation generation and independent validation.
- **C2 – Frozen evaluation protocol.** A leakage-conscious design that fixes the benchmark, splits, baselines, prompts, sampling parameters and metrics before test evaluation, with six experiments (E1–E6).
- **C3 – Provenance-preserving correlation.** A conservative scheme for merging SAST and LLM findings that keeps every finding's source, enabling an explicit unique/overlap analysis of hybrid benefit.
- **C4 – Anchored explanation rubric.** A six-criterion rubric that scores explanation quality independently of detection accuracy, exposing any gap between fluency and correctness.
- **C5 – Staged patch validation.** A four-stage validation pipeline and a six-class outcome taxonomy that distinguish correct patches from functional, security, compilation, partial and invalid failures.
- **C6 – Pre-declared interpretation rule.** A decision rule, threats-to-validity analysis and set of ethical controls that bound the claims the study may make.

Scope of this paper. We present a design and an evaluation protocol, not results. Nothing has been run yet, so no performance figure appears here, and the numerical cells of every results table are marked "TBD". We chose this format on purpose: it lets us fix the analysis plan before anyone, ourselves included, has seen the test data.

1.1 Research questions and hypotheses

The study is organised around eight research questions (Table 1) and four falsifiable hypotheses (Table 2).

Table 1. Research questions.

#	Research question	Evaluation focus
RQ1	How effectively can an LLM detect vulnerabilities in source code under a controlled benchmark?	Precision, recall, F1, FPR, FNR
RQ2	How accurately can an LLM classify detected vulnerabilities against a standard weakness taxonomy?	CWE accuracy; per-class F1
RQ3	To what extent are LLM-generated explanations technically correct, localised and actionable?	Structured rubric
RQ4	How effectively can an LLM generate candidate remediation?	Patch generation and compilation rates
RQ5	What proportion of generated patches remove the weakness while preserving intended functionality?	Validated patch rate; functional preservation
RQ6	How does LLM-based detection compare with a conventional SAST baseline?	Metric differences under one protocol
RQ7	Does combining SAST findings with LLM reasoning improve coverage or precision?	Unique/overlap analysis
RQ8	What reliability, reproducibility, privacy and security limitations arise?	Threats to validity; error analysis

Table 2. Hypotheses and falsification criteria.

#	Hypothesis	Falsification criterion
H1	The hybrid SAST+LLM configuration gives complementary coverage compared with either engine alone.	Rejected if the hybrid's unique correct findings are predominantly false positives or add no coverage.
H2	LLM explanations give more contextual detail than scanner messages, but fluency does not imply technical correctness.	Confirmed if high language quality coexists with low root-cause scores on the rubric.
H3	Compiler, test and security re-analysis feedback improves the reliability of LLM remediation relative to one-shot generation.	Rejected if validated patch rates do not improve under feedback-guided regeneration.
H4	Leakage-resistant, temporally separated evaluation yields more conservative performance estimates than duplicated random splits.	Assessed by comparing protocol choices across the literature and, where feasible, across splits.

The remainder of the paper is organised as follows. The next section reviews related work and derives six research gaps. The framework architecture is then presented, followed by the evaluation protocol, the experimental design and reporting templates, and the threats to validity and ethical controls. The paper closes with a discussion of limitations and future work and the conclusion.

2 Related work

Fig. 1 traces the lineage that this section follows, from rule-based scanners to language models.

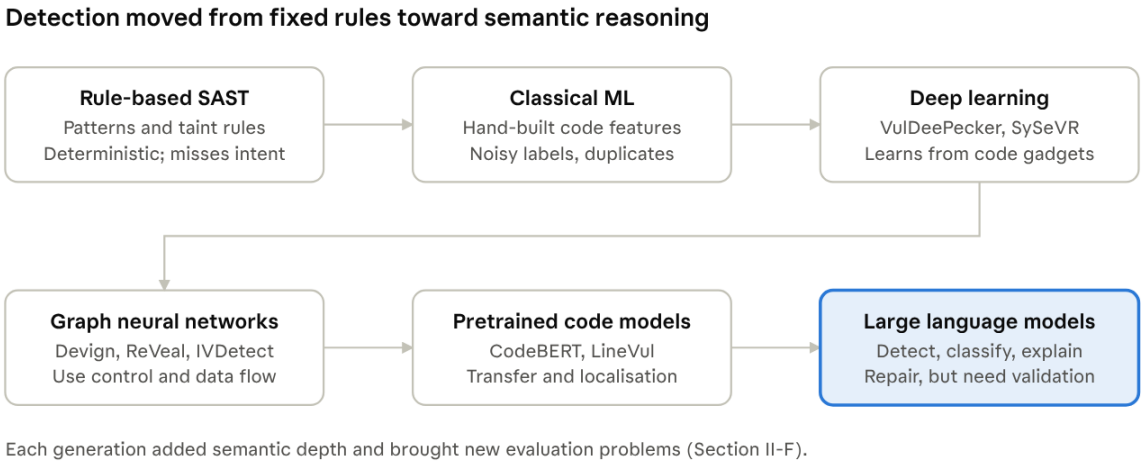

Figure 1. Evolution of automated vulnerability detection.

2.1 Rule-based, dynamic and manual analysis

SAST tools read code without running it. They build syntax trees, follow control and data flow, and track tainted values from untrusted sources to risky sinks. Their strengths are speed, determinism, coverage of paths that are hard to reach at runtime, and traceability, since every alarm points to the rule behind it. Their weaknesses come from the same place. People write the rules, so new or context-dependent flaws slip through, flow approximations raise false alarms, and the message (a rule ID, a location, a short sentence) seldom tells a developer what to do. Dynamic testing and fuzzing take the opposite route. They watch real executions, so a reproduced crash is almost never a false alarm, but they need something runnable, only cover the paths their inputs reach, and report symptoms rather than causes. Interactive testing and symbolic execution add runtime context or path reasoning, and pay for it with dependence on test coverage and path explosion. Careful manual review is still the yardstick for depth, but it does not scale. In practice teams layer these approaches, with automation for breadth and people for depth, and our hybrid design follows the same idea. Table 3 lines the families up.

Table 3. Comparison of SAST, dynamic/fuzzing, interactive and LLM-based analysis.

Criterion	SAST	DAST / fuzzing	IAST	LLM-based
Execution required	No	Yes	Yes (instrumented)	No
Code coverage	High	Path-limited	Test-limited	High (context window)
False positives	Moderate–high	Very low	Low	Variable; hallucination risk
Semantic reasoning	Rule-bounded	None	Moderate	Strong
Natural-language explanation	Weak	None	Weak	Strong
Reproducibility	Full	High	High	Sampling-dependent
Remediation suggestion	Rare / template	None	Rare	Generative (unvalidated)

No column wins everywhere. Each family fails in its own way, which is exactly why combining them can add something instead of repeating what the other already does.

2.2 Learning-based detection

Early machine-learning detectors swapped hand-written rules for classifiers over engineered features. They were limited by those features and were often evaluated on noisy, duplicated data. Deep learning removed the feature step. VulDeePecker introduced “code gadgets”, slices of statements around library calls, and fed them to bidirectional LSTMs [8]; SySeVR broadened this with syntax- and semantics-based candidates and program slicing [9]. Graph models followed from the observation that programs are graphs, not text. Devign combined the syntax tree, control-flow graph, data-flow graph and token sequence [10], ReVeal went after class imbalance and duplication in older datasets [11], IVDetect added inter-procedural context and finer-grained interpretations [12], and ReGVD argued that how the program graph is built matters a great deal [13]. We take two lessons from this line of work. Whole code units, not isolated lines, should reach the analyser. And realistic data is not optional, because convenient data gives convenient conclusions.

2.3 Pretrained code models

Pretrained transformers brought transfer learning to code. CodeBERT was pretrained on paired natural language and code [14], GraphCodeBERT added data-flow information [15], and CodeT5 combined understanding and generation in one encoder–decoder model [16]; PLBART and UniXcoder pursue similar goals in different ways [17], [18]. Two descendants are aimed squarely at vulnerabilities. LineVul fine-tunes a CodeBERT-style model for function-level detection and uses self-attention to point at suspicious lines [19], an early kind of explanation whose faithfulness is still debated. VulBERTa pretrains a BERT-style model with a tokenisation adapted to vulnerability data [20]. These models are usually fine-tuned and tested on public datasets such as BigVul [21] and CVEfixes [22], which the 'Benchmarks and evaluation methodology' subsection discusses.

2.4 Large language models for vulnerability analysis

Instruction-following LLMs change what a detector can be asked to do. Instead of returning a label, a model can be asked to find a weakness, place it in the CWE taxonomy [6], explain the root cause and impact, and propose a fix, all in one exchange. The surveys in [2] and [3] sort the literature by model, adaptation method, dataset and deployment style, and cover zero-shot and few-shot prompting, parameter-efficient fine-tuning, retrieval augmentation and agent-style workflows. Two empirical studies are close precedents for ours. Tamberg and Bahsi compared several LLMs against static-analysis tools [23], and Dozono and colleagues tested LLMs on several languages for both detection and CWE classification [24], which is part of why we measure classification separately. LLift pairs an LLM with static analysis to find bugs in the Linux kernel [25] and shows that the two can be interleaved at scale. A recurring theme across this work is variance: prompts, benchmarks and split strategies all move the numbers, so our protocol freezes all three.

2.5 Automated repair and patch validation

Automated program repair existed long before LLMs, and its stubborn problem was overfitting: a patch passes the tests but leaves the real defect in place, because a test suite is only a partial specification. LLM-generated security patches inherit this problem with one more twist, since a patch can pass every functional test while the vulnerability stays open. VRpilot is an encouraging example

of the alternative. It feeds the model output from compilers, sanitizers and tests, and in its own evaluation reports more correct patches than its baselines [26]. That is the argument for treating generation and validation as separate steps.

2.6 Benchmarks and evaluation methodology

BigVul collected real C/C++ vulnerabilities together with the commits that fixed them [21], and CVEfixes widened that idea to more projects and languages [22]. Devign and ReVeal each shipped a dataset alongside their models [10], [11]. PrimeVul matters most to us. Its authors found duplicated or near-duplicated functions on both sides of the train–test split, a good deal of label noise, and random splits that let models lean on project- and time-specific shortcuts, and they saw the scores of previously strong models drop sharply under deduplicated, time-ordered evaluation [4]. The practical rule for us is simple: any performance claim has to name the protocol that produced it.

2.7 Explainability in security analysis

"Explainable" means two different things in this literature. Post-hoc methods such as attention maps and feature attribution show which tokens or lines influenced a model, but how faithful they are to its actual reasoning is contested. Generative explanation asks the model to write the weakness down itself: root cause, impact, evidence and a suggested fix. That is what developers actually read, so it is what we use. The catch is that fluent text is cheap and correct text is not, which is why we score explanations against a rubric and report them separately from detection accuracy.

2.8 Research gaps

Synthesising the above yields six gaps (Table 4).

Table 4. Research gaps and how this work addresses them.

#	Gap	Evidence	Response in this work
G1	The security lifecycle is fragmented: detection, classification, explanation, remediation and validation are studied in isolation.	[2], [5]	Unified seven-layer pipeline
G2	Explanation is the least-studied stage, and fluency is routinely mistaken for correctness.	[5]	Structured rubric scored independently of detection
G3	Generated patches are rarely validated independently; compiling patches are reported as fixes.	[26]	Four-stage validation
G4	Benchmark leakage and duplication inflate reported performance.	[4]	Leakage-conscious benchmark; chronological split; frozen protocol
G5	Risk or severity is conflated with model confidence.	[3]	CVSS-informed risk layer [7] independent of confidence
G6	SAST and LLM analyses are compared competitively rather than combined complementarily.	[23], [25]	Correlation layer with explicit unique/overlap analysis

We therefore see this framework as an integration and methodology contribution, not a new detection algorithm. Because SAST-only, LLM-only and hybrid runs share one frozen protocol, any gain can be credited to the integration and not to differences in setup.

3 Proposed framework

3.1 Design principles

Three ideas shape every stage. **Prediction and validation are kept apart.** The LLM proposes a finding, a CWE label, an explanation or a patch, and something else decides whether to believe it: benchmark labels and the SAST baseline for findings, the CWE hierarchy for classifications, a rubric for explanations, and compilation, testing and re-analysis for patches. The model generates candidates and never judges its own. **The design is frozen before evaluation.** Architecture, benchmark, baselines, prompts, metrics and analysis plan are fixed before a test result is seen, which keeps us from steering the study with hindsight. **The configurations stay comparable.** SAST-only, LLM-only and hybrid runs use the same splits and the same metric definitions, and the only intended difference is where the findings come from.

3.2 Seven-layer architecture

The framework has seven logical layers (Table 5). Layers 2 and 3 run in parallel on the same preprocessed input, which makes the hybrid configuration a first-class element of the design.

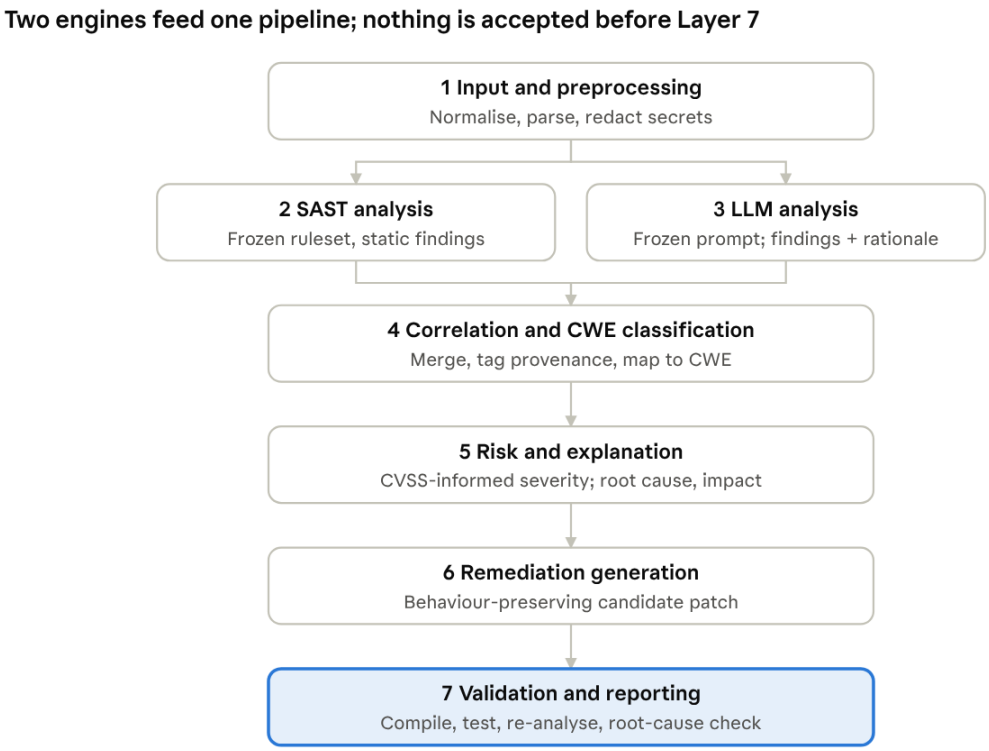

Figure 2. Seven-layer architecture.

Layers 2 and 3 run in parallel on the same input; the highlighted Layer 7 is the only gate to acceptance.

Table 5. Seven-layer architecture.

Layer	Input	Processing	Output
1. Input and preprocessing	Source code	Language detection, normalisation, safe parsing, secret redaction	Analysis-ready code
2. SAST analysis	Source / AST	Rule-based analysis with a frozen ruleset	Static findings
3. LLM analysis	Code + controlled prompt	Semantic reasoning under frozen configuration	LLM findings + explanation
4. Correlation and classification	SAST + LLM findings	Deduplication, correlation, CWE mapping	Unified classified findings
5. Risk and explanation	Unified findings	CVSS-informed severity; evidence, root cause, impact	Explainable, prioritised findings
6. Remediation	Confirmed finding	Behaviour-preserving candidate patch generation	Patch candidates
7. Validation and reporting	Patch + original tests	Compilation, functional tests, security re-analysis	Accepted/rejected patch + audit-ready report

Layer 1: input and preprocessing. This layer turns raw code into something both engines can consume without throwing away anything that might matter for security. Normalisation fixes encoding, line endings and whitespace but leaves identifiers, literals and comments alone, since the LLM draws on all three. A redaction step swaps secrets such as keys, tokens and credentials for typed placeholders before any text leaves the controlled environment. Each unit gets a stable identifier, and we keep both the raw and the normalised version so that any later claim can be traced back to the exact input.

Layer 2: SAST baseline. A deterministic scanner runs under a frozen configuration, and its output is parsed into one common finding schema (file, function, line range, source, vulnerability, CWE, severity, confidence, explanation, patch and validation status). Rule identifier, location, severity and message are stored verbatim, so correlation never depends on our reinterpretation of what the tool meant.

Layer 3: LLM semantic analysis. The code unit goes to the model under a frozen prompt and sampling setup (see 'LLM configuration and prompt control'). The reply is parsed into the same schema, together with the written explanation and a candidate CWE. If the model answers in free text, a schema-constrained prompt with one re-prompt on parse failure keeps parsing deterministic without narrowing what the model may say. Raw replies are stored untouched.

Layer 4: correlation and CWE classification. Findings from both engines are normalised, exact duplicates are dropped, and overlapping findings are correlated (see 'Detection procedure and finding correlation'). Matches become dual-evidence records; findings without a partner stay as single-source records and are never discarded. Each record keeps its provenance (SAST-only, LLM-only or both), so that each engine's unique contribution can be counted later.

Layer 5: risk assessment and explanation. Each finding receives two separate things. Risk is computed from the CVSS v3.1 base-metric groups [7], exploitability context and evidence quality, and it does not use the LLM's own confidence. The explanation is the LLM's structured account of root cause, impact, code evidence and the direction a fix should take. Keeping the two under different

authorities is deliberate: we want the model's ability to explain without handing it the decision on how serious a finding is.

Layer 6: remediation generation. For each confirmed vulnerability the LLM gets the finding, its CWE label, the explanation and the vulnerable code. It must return a patch that can be applied directly (a unified diff or a replacement function) plus a rationale, and the patch must keep intended behaviour. The patch and rationale are stored apart from the validation verdict, so the model's own argument cannot sway the people checking it.

Layer 7: validation and reporting. Every candidate patch goes through the staged pipeline of the 'Remediation generation and validation' subsection and receives an outcome from Table 9, and only patches that pass every stage are accepted. The layer then writes an audit-ready report in which each field, from the first alarm to the final verdict, links back to stored evidence. If one layer defines the framework, it is this one, because it is what turns plausible model output into something we can rely on.

4 Evaluation protocol

4.1 Benchmark and data preparation

The primary detection experiment uses PrimeVul [4], because it addresses the problems described in the 'Benchmarks and evaluation methodology' subsection. Functions are deduplicated so near-identical code cannot sit on both sides of the split, labels follow stricter criteria than earlier datasets, and the split is chronological: models are tested on vulnerabilities disclosed after the training cut-off. That mirrors real deployment, where weaknesses are new, and it discourages memorising project- or era-specific patterns.

Data preparation has five steps. (1) The benchmark release is pinned to an exact version and checksum. (2) Each sample is reduced to the function-level experimental unit with its ground-truth label and, where available, its CWE annotation. (3) The Layer 1 preprocessing is applied identically to vulnerable and benign samples, so that preprocessing cannot leak label information. (4) The frozen split is materialised and stored, and no sample may move between splits afterwards. (5) A small development slice, disjoint from the test split, is reserved for engineering debugging and is the only data on which prompts or configurations may be tuned. The test split is touched exactly once, at final evaluation.

4.2 SAST baseline configuration

The baseline has to be strong enough to be a fair opponent and frozen enough to be reproducible. We will pick a widely used, actively maintained scanner with documented C/C++ coverage and machine-readable output, and record its name, exact version, full ruleset (including disabled rules), options and environment. It is configured and frozen before the LLM experiment begins, and its findings on the test split are computed once and stored. Two choices are spelled out because they affect fairness. The ruleset stays at the tool's recommended default for the language, since tuning the scanner to the benchmark would make the baseline look better or worse depending on our own choices rather than the tool's merit. And the scanner's severity and confidence fields are parsed but never remapped, so the framework's risk assessment (see 'CWE classification, risk assessment and explanation evaluation') is the only severity authority downstream, applied equally to both engines.

4.3 LLM configuration and prompt control

Before any test data is observed, the study freezes the exact model identifier and version, the system instruction, the task prompt template, the format in which code is presented, temperature and all other sampling controls, the maximum output length, and the retrieval configuration (disabled in the primary evaluation). Prompt changes after seeing test results are not permitted in the primary evaluation, because prompt engineering against test feedback is a form of training on the test set. Iteration occurs only on the development slice, and every prompt version is logged.

The detection prompt has a fixed five-part structure: role and task definition; output schema; the security criteria to consider; the code unit; and an explicit instruction to abstain when evidence is insufficient (Table 6). Abstention is a designed behaviour, not a failure. A model forced to judge every input manufactures findings, whereas a model permitted to answer "uncertain" yields findings whose precision can be meaningfully measured, and the abstention rate is itself reported.

Table 6. Frozen detection prompt template (abridged).

Section	Template content
Role and task	"You are a security analyst. Analyse the following C/C++ function and determine whether it contains a security vulnerability. Do not assume code outside the shown unit."
Output schema	JSON object: vulnerable (yes/no/uncertain); cwe (identifier or null); location (lines); root_cause; impact; evidence; confidence_note; remediation_direction
Security criteria	Consider memory safety, injection, integer errors, unchecked input, improper access control and other CWE-listed weaknesses. Cite the exact code behaviour supporting any positive finding.
Code unit	Function source, normalised, with line numbers
Abstention rule	"If the evidence is insufficient to decide, answer 'uncertain' and state what additional context would be required. Do not guess."

4.4 Detection procedure and finding correlation

A single scripted harness executes the same sequence for every sample in the frozen test split, so that no manual decision intervenes between input and stored results: load the sample with its label and CWE annotation; normalise the code; run the frozen SAST baseline and parse its output; run the LLM under the frozen configuration and store the raw response; parse the response into the common schema (re-prompting once on parse failure and recording the retry); correlate findings across engines; map findings to CWE identifiers; and store raw and normalised outputs from every stage. Any run whose reproducibility stamp (see 'Reproducibility record') is incomplete is excluded from reporting.

Findings from the two engines are correlated on five signals: file identity, function identity, line-range overlap, vulnerability-category agreement (after mapping toward CWE), and semantic similarity of the described weakness. A weighted rule yields one of three outcomes: **matched** (merged into a dual-evidence record), **unmatched** (kept as a single-source record) or **uncertain** (kept separately and flagged for manual review). Correlation never silently declares two findings equivalent, so the error analysis can measure correlation quality itself rather than hiding it inside merged records. The resulting unique-finding sets are the direct evidence for H1: if SAST-unique and LLM-unique findings are both largely correct, the engines are complementary; if one engine's unique findings are largely false alarms, the hybrid's value lies in corroboration rather than coverage.

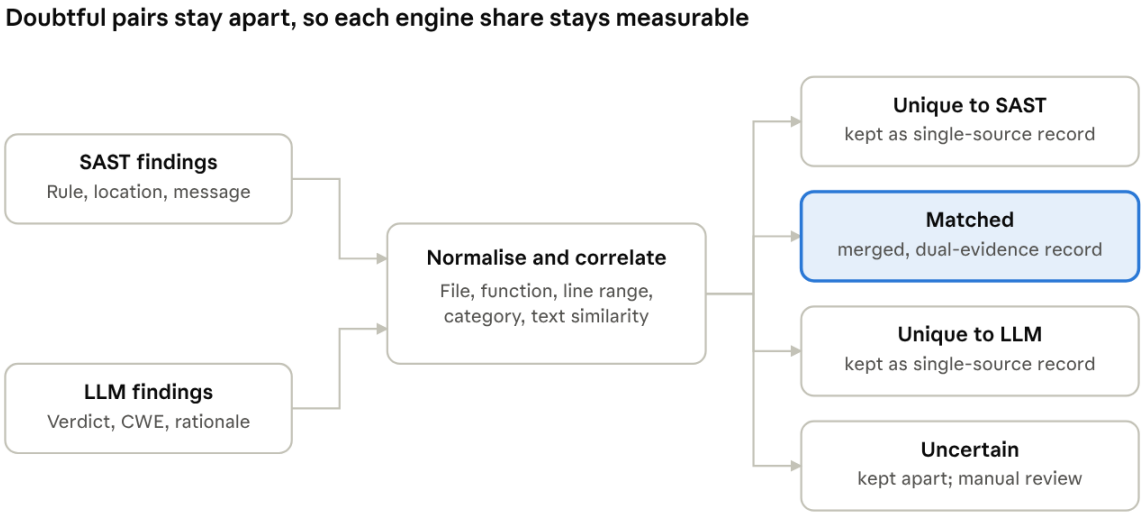

Figure 3. Hybrid finding correlation.

Every output keeps its provenance, which is what the unique/overlap analysis needs.

4.5 CWE classification, risk assessment and explanation evaluation

Classification. Classification maps each unified finding to the CWE taxonomy [6] and is treated as a distinct task, because a detector can be right that code is vulnerable while being wrong about why, and the "why" drives prioritisation and remediation. The LLM proposes a candidate CWE, which is cross-checked against the CWE hierarchy (for example, CWE-787, out-of-bounds write, is consistent with its ancestor memory-buffer category, and both are recorded). Classification is evaluated at three granularities: exact CWE match, parent-class match, and top-k accuracy when ranked candidates are returned.

Risk. Risk answers a different question from detection: not "is this a weakness?" but "how much does it matter?". It combines three inputs: severity information derived from the CVSS v3.1 base-metric groups [7] (Table 7); exploitability context such as attack vector, privileges required and exposure; and evidence quality (dual-source, single-source or uncertain; reproduces under re-analysis or not). The LLM may supply the narrative of attack scenario and affected assets, but it is never the sole authority for severity, so a fluent but miscalibrated model cannot inflate or deflate a finding's priority.

Table 7. CVSS v3.1 base-metric groups used for risk assessment.

Metric group	Metrics	Role in the framework
Exploitability	Attack vector; attack complexity; privileges required; user interaction	How easily the weakness can be triggered in the analysed context
Scope	Scope (changed / unchanged)	Whether exploitation affects components beyond the vulnerable one
Impact	Confidentiality; integrity; availability	Worst credible consequence if exploited

Explanation. Explanations are scored against a six-criterion rubric (Table 8), each criterion on a 0–2 scale, by assessors blinded to whether the underlying detection was correct. The rubric operationalises H2: a fluent explanation that misidentifies the root cause must score low on root cause however

good its prose. Two independent assessors score a double-scored sample, disagreements are resolved by discussion after independent scoring, and inter-assessor agreement (Cohen's kappa) is reported. The key output is the cross-tabulation of explanation correctness against detection correctness. An explanation that is confident but attached to a false positive is not merely inaccurate; it is actively misleading.

Table 8. Explanation quality rubric with scoring anchors.

Criterion	0 (absent / wrong)	1 (partial)	2 (correct and complete)
Localisation	No region identified, or wrong region	Correct function, imprecise lines	Exact vulnerable lines or statements
Root cause	Misidentified or generic	Correct weakness family, wrong mechanism	Mechanism by which the weakness arises stated correctly
Impact	Wrong or missing	Plausible but ungrounded	Accurate, evidence-linked consequence
Evidence	No code behaviour cited	Vague reference to code	Specific code behaviour tied to the claim
Remediation relevance	Absent or unrelated	Generic advice ("validate input")	Advice addresses the stated root cause
Unsupported claims	Material invented facts	Minor unsupported details	No claim beyond the shown evidence

4.6 Remediation generation and validation

For each confirmed vulnerability the LLM produces a candidate patch under three controls. The prompt carries the full finding context so the patch targets the identified mechanism rather than the symptom; the output is constrained to a directly applicable form so that application is mechanical; and the patch and rationale are stored separately from the validation verdict. Validation is the framework's trust anchor. Every candidate passes four stages in order and is rejected at the first stage it fails:

- **Syntax and compilation.** The patched unit must build cleanly under the project's toolchain; failure is a generation failure.
- **Functional tests.** The patched code must pass the existing tests plus any behaviour-preservation checks defined for the unit; failure means the patch changed intended behaviour.
- **Security re-analysis.** Both engines re-analyse the patched code under the same frozen configuration; if the original weakness is still reported, the patch is a security failure.
- **Comparison with the original vulnerability.** A structured diff-based check confirms that the change addresses the identified root cause (for example, a bounds check on the exact operation the finding cited), guarding against patches that merely silence the alarm.

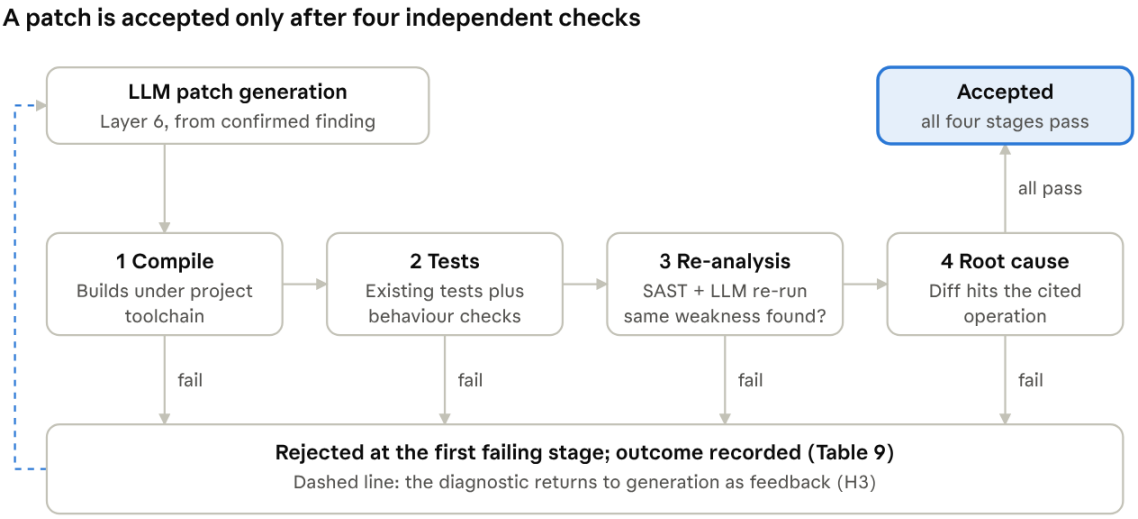

Figure 4. Remediation and validation pipeline.

A rejected patch may be regenerated with the failing stage's diagnostic; every round is logged.

A patch that compiles but leaves the weakness in place is classified as unsuccessful, which is the failure mode that single-stage evaluation cannot detect. Rejected patches may be returned to Layer 6 with the failing stage's diagnostic as feedback (testing H3); each regeneration round is logged, and both first-try and feedback-assisted validated patch rates are reported. Table 9 defines the outcome taxonomy.

Table 9. Remediation outcome taxonomy.

Outcome	Definition
Correct patch	Compiles, passes functional tests and removes the vulnerability under security re-analysis
Functional failure	Changes or breaks intended behaviour (fails Stage 2)
Security failure	Passes Stages 1–2 but does not remove the original vulnerability (fails Stage 3 or 4)
Compilation failure	Cannot be built or parsed (fails Stage 1)
Partial repair	Addresses some security conditions but the weakness remains detectable
Invalid generation	Output cannot be interpreted as a usable patch

4.7 Metrics and statistical analysis

Primary detection metrics are precision, recall, F1-score, false-positive rate (FPR) and false-negative rate (FNR), computed from the confusion matrix (Table 12). Precision measures the trustworthiness of positive findings, recall measures coverage, FPR measures the benign-code alarm rate that drives warning fatigue, and FNR measures the missed-vulnerability rate that drives false confidence. Because no single number captures security utility, all five are always reported together. Stage-specific metrics are defined in Table 10, and runtime and token or compute cost per configuration are reported so that the operational price of any gain is visible.

Table 10. Evaluation metrics.

Metric	Formula	Purpose
Precision	TP / (TP + FP)	Correctness of positive findings
Recall	TP / (TP + FN)	Vulnerability coverage
F1-score	2·P·R / (P + R)	Balance of precision and recall
False-positive rate	FP / (FP + TN)	Benign-code alarm rate
False-negative rate	FN / (FN + TP)	Missed-vulnerability rate
CWE accuracy	Correct CWE / classified findings	Taxonomy-mapping quality
Patch success rate	Validated patches / attempted patches	Remediation effectiveness
Functional preservation	Behaviour-preserving patches / validated patches	Non-regression of fixes

The unit of analysis is the code unit (function), not the token or line, because labels are assigned per function and treating finer elements as independent would manufacture spurious sample size. Bootstrap confidence intervals over the experimental unit are the default for all primary metrics. Because all configurations see the same samples, comparisons use paired tests, and effect sizes accompany p-values, since at benchmark scale trivial differences can be statistically significant yet operationally meaningless. Multiple-comparison correction is applied when families of related tests, such as per-CWE comparisons, are reported together.

4.8 Reproducibility record

Every reported run is accompanied by a record containing the benchmark name, version, checksum and split; the preprocessing script version; the exact model identifier and provider snapshot; the prompt template version; all sampling parameters and random seeds; the SAST tool name, version, ruleset and invocation; the hardware and software environment; the evaluation script versions; and pointers to stored raw outputs. Any number that cannot be traced to a complete record is excluded from the reported results.

5 Experimental design and reporting templates

5.1 Experiments

Six experiments operationalise the research questions (Table 11). E1–E3 form the comparative core, E4–E5 evaluate remediation end to end, and E6 probes generalisation where feasible.

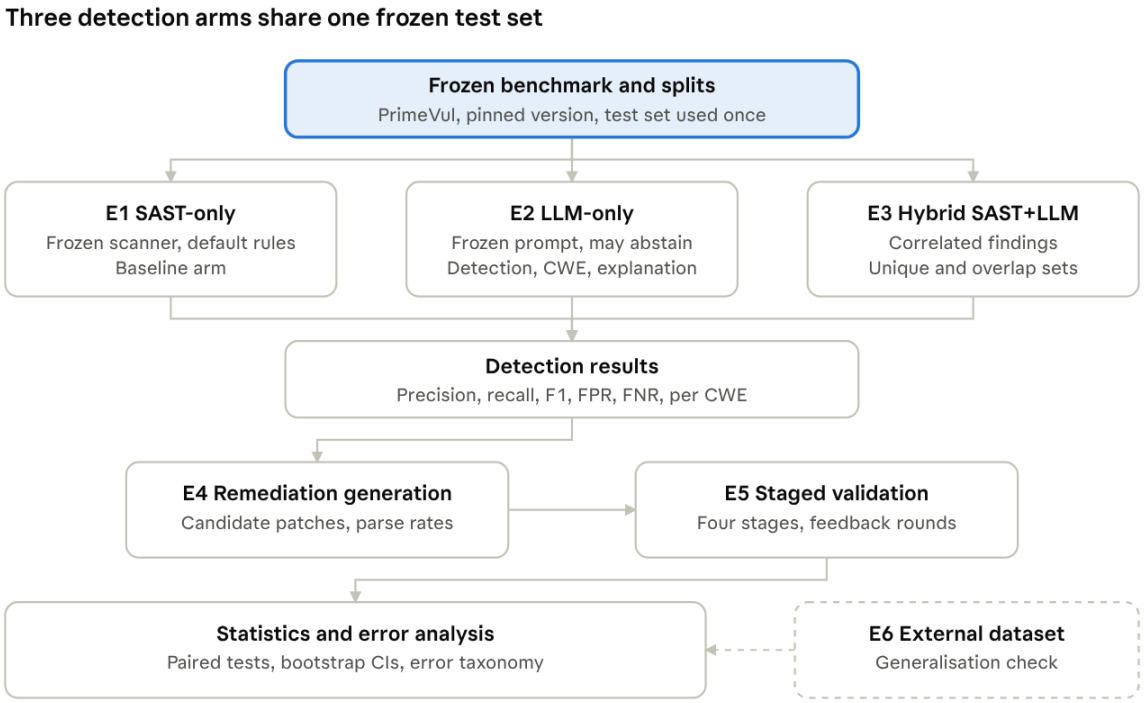

Figure 5. Experimental design.

E1–E3 share one frozen test split; E6 is an optional external check.

Table 11. Experimental configurations.

ID	Configuration	Answers	Primary output
E1	SAST-only detection on the frozen test split	RQ6 (baseline arm)	Baseline detection metrics
E2	LLM-only detection under the frozen prompt and configuration	RQ1, RQ2, RQ3, RQ6	LLM detection metrics; abstention rate
E3	Hybrid SAST+LLM with finding correlation	RQ7, H1	Unique/overlap analysis; combined metrics
E4	Remediation generation for confirmed findings	RQ4	Patch candidates; parse and compile rates
E5	Staged remediation validation, with feedback rounds	RQ5, H3	Validated patch rate; outcome counts
E6	External evaluation on a disjoint dataset	H4; external validity	Cross-dataset metric deltas

5.2 Detection reporting templates

Counts of true and false positives and negatives are aggregated per configuration (SAST-only, LLM-only, hybrid) and per CWE category (Table 12). Because the same test split is used throughout, the differences between configurations, not their absolute levels, carry the study's principal claims. Table 13 is the primary output of the study; its cells will be populated after E1–E3 are executed and will carry bootstrap 95% confidence intervals.

Table 12. Confusion matrix.

	Actual vulnerable	Actual benign
Predicted vulnerable	TP	FP
Predicted benign	FN	TN

Table 13. Configuration comparison template.

Configuration	Precision	Recall	F1	FPR	FNR	Runtime / cost
E1 – SAST-only	TBD	TBD	TBD	TBD	TBD	TBD
E2 – LLM-only	TBD	TBD	TBD	TBD	TBD	TBD
E3 – Hybrid SAST+LLM	TBD	TBD	TBD	TBD	TBD	TBD

To be populated; no results reported.

Three analyses derive from this table: whether LLM-only detection matches, exceeds or falls short of a well-configured scanner and along which axes (RQ6); whether the hybrid's precision–recall trade-off dominates either single engine or merely trades one axis for another (RQ7, H1); and the compute and latency price of any accuracy gain.

5.3 Cwe-level, explanation and remediation reporting

Per-CWE performance is reported alongside aggregates, because aggregates conceal a common failure mode: strong results on frequent, syntactically distinctive categories such as obvious buffer overflows, and collapse on rarer or semantically subtle ones such as time-of-check-to-time-of-use races or authorisation flaws. Categories with too few samples are pooled into a residual bucket that is reported rather than dropped. Explanation results comprise the mean and distribution of scores per criterion, inter-assessor agreement, and the explanation-versus-detection cross-tabulation that tests H2. Remediation results comprise the outcome distribution of Table 9, validated patch rates for one-shot and feedback-guided generation (H3), and a per-CWE patch-success view, since the ability to fix memory-safety weaknesses is not evidence of an ability to fix authorisation weaknesses.

5.4 Hybrid analysis

The hybrid analysis examines three sets: findings unique to SAST, findings unique to the LLM, and overlapping findings. Four qualitatively different outcomes are possible, and the conclusion each supports is stated in advance:

- **Both unique sets are largely correct:** the engines are genuinely complementary (H1 supported by coverage).
- **Only one unique set is largely correct:** the other engine contributes mainly corroboration, and the recall improvement is asymmetric.
- **Overlap dominates and unique sets are small:** the engines cover the same ground, and the hybrid's value lies in raising precision through agreement.
- **Both unique sets are largely wrong:** the engines add noise to each other, and hybrid detection reduces to re-weighting rather than genuine coverage gain.

Aggregate F1 improvements can arise in any of the four regimes, but only the first supports the complementarity claim, which is why the analysis is structured this way.

5.5 Error taxonomy

Errors are grouped into eight mechanism-defined categories: **semantic misunderstanding** (the model misreads what the code means); **missing context** (the finding depends on a caller, header or build flag outside the unit, and the model invents or ignores it); **incorrect taxonomy mapping** (right weakness, wrong CWE); **hallucinated vulnerability** (a finding with no support in the code); **benchmark-label disagreement** (a defensible finding that conflicts with the label, reported separately as label noise); **duplicate finding** (the same weakness reported repeatedly within or across engines); **incomplete remediation** (one instance fixed, related instances left); and **regression-inducing remediation** (the weakness is removed but behaviour changes, caught at Stage 2).

5.6 Interpretation rule

Declared before any test-set result is observed. No configuration will be declared superior solely because it achieves the highest F1-score. Interpretation will weigh, jointly and with equal weight, detection quality, false alarms, per-CWE coverage, explanation correctness, remediation reliability, validation success and operational cost. A configuration wins an axis only when its advantage exceeds a paired statistical test and a pre-declared effect-size threshold, and wins the study only when it wins a majority of axes without losing any by a substantial margin. This rule is binding on the final analysis.

6 Threats to validity and ethical controls

Table 14 lists the principal threats and the mitigation each receives in the design. They are reported rather than concealed because several of them, including prompt sensitivity, model drift and benchmark bias, are themselves findings of the literature on which this work builds.

Table 14. Threats to validity and mitigations.

Threat	Description	Mitigation
Dataset bias and label noise	Benchmark labels may misclassify borderline code	Strict-label benchmark; manual audit of a sample
Data leakage	Near-duplicates across splits inflate scores	Deduplicated, chronologically split benchmark; frozen splits
Model variation and drift	Provider updates change behaviour over time	Exact version recorded; raw outputs stored; spot-check reruns
Prompt sensitivity	Paraphrases change accuracy	Prompts frozen before testing; iteration confined to development slice; all versions logged
Researcher degrees of freedom	Post-hoc design changes bias results	Design frozen before test evaluation; single scripted harness
SAST configuration differences	Baseline strength depends on setup	Default recommended ruleset; full configuration recorded
Hallucination	Fluent but false findings or explanations	Structured rubric with unsupported-claims criterion; independent validation
Patch overfitting	Patches pass tests without fixing the weakness	Four-stage validation including security re-analysis and root-cause check
Rubric construct validity	Explanation scoring may be subjective	Anchored rubric; blinded assessors; agreement reported

Threat	Description	Mitigation
External validity	C/C++ function-level results may not generalise	Scope stated explicitly; E6; multilingual and repository-level work deferred
Hardware and cost limits	Compute budget limits scale	Cost recorded as a metric; sample sizes planned in advance

Ethical and security controls. Source code submitted for analysis is treated as potentially confidential and is transmitted only to the configured model endpoint under the applicable data-processing terms; secrets are redacted before any content leaves the controlled environment. Untrusted code is never executed on the main development host: compilation and tests run in isolated, disposable environments with restricted privileges and no outbound network access. The research analyses and remediates vulnerabilities but does not develop, optimise or demonstrate exploitation techniques. Benchmark data is used under its licences, and all datasets, tools and models are acknowledged.

7 Discussion, limitations and future work

Positioning. We do not assume that LLMs will replace traditional security tools. The question is narrower: can LLM reasoning add something to deterministic analysis, can generated fixes be validated reliably, and can an integrated pipeline be evaluated with the care that security work demands? If the experiments support our hypotheses, the result is a template for using LLMs in security without giving up auditability. If they do not, reporting under a protocol fixed in advance will at least show clearly where and why things fail.

Limitations. Our primary scope is function-level C/C++ code, and results will not carry over automatically to whole repositories, other languages, or weaknesses that need system-wide reasoning. PrimeVul was chosen for its rigour, so generalisation to other datasets has to be checked (E6) and not assumed. A hosted model can drift, and version pinning reduces that risk without removing it. The validation stack is much stronger than accepting a patch at first sight, but it is still only as good as the functional tests available for each unit, a weakness we inherit from program repair. Finally, this paper reports a design and a protocol, so every claim of benefit remains a hypothesis.

Future work. Six extensions each attach to a specific layer. Extending to Python, Java and JavaScript would show whether the gains transfer to other weakness landscapes (Layers 1–3). Repository-level analysis would bring in call graphs, build configuration and related tests (Layers 1, 4 and 7). Retrieval over CWE descriptions and known vulnerabilities could ground classification and cut down on hallucination (Layer 3). Agent-style analysis, where the model runs compilers, scanners and test harnesses, would turn Layer 3 into a bounded workflow; it promises the most capability and carries the most dual-use risk, so it would need explicit safety limits. Running the framework inside CI/CD would test whether it holds up under real latency, cost and warning-fatigue constraints. Last, adversarial evaluation on obfuscated or deliberately misleading code would show where each layer breaks.

8 Conclusion

We proposed a framework in which detection, correlation, classification, risk assessment, explanation, remediation and validation form one evidence-producing pipeline, with SAST and LLM analysis acting as complementary sources of evidence. Its distinctive commitments are about method. The

design and protocol are fixed before test evaluation, the benchmark is chosen to avoid leakage, risk is kept separate from model confidence, explanations are scored apart from detection, and no generated patch is accepted until it survives four independent checks. The paper sets out six experiments, eight research questions, four falsifiable hypotheses and an interpretation rule written in advance. Numbers will follow once the framework is implemented, and they will be reported under that rule.

Declarations

Data and code availability: No datasets were generated in this study. The evaluation harness, prompt-control records and reproducibility records will be released with the empirical results.

Funding: This research received no external funding.

Competing interests: The authors declare no competing interests.

Acknowledgements: None.

References

[1] OWASP Foundation, "OWASP Top 10 – 2021: The Ten Most Critical Web Application Security Risks," *2021*. [Online]. Available: https://owasp.org/Top10/

[2] X. Zhou, S. Cao, X. Sun, and D. Lo, "Large Language Model for Vulnerability Detection and Repair: Literature Review and the Road Ahead," *ACM Trans*. Softw. Eng. Methodol., vol. 34, no. 5, Art. 145, 2025, doi: 10.1145/3708522.

[3] Z. Sheng, Z. Chen, S. Gu, H. Huang, G. Gu, and J. Huang, "LLMs in Software Security: A Survey of Vulnerability Detection Techniques and Insights," *ACM Comput*. Surv., vol. 58, no. 5, Art. 134, 2025, doi: 10.1145/3769082.

[4] Y. Ding, Y. Fu, O. Ibrahim, C. Sitawarin, X. Chen, B. Alomair, D. Wagner, B. Ray, and Y. Chen, "Vulnerability Detection with Code Language Models: How Far Are We?" in *Proc*. IEEE/ACM Int. Conf. Software Engineering (ICSE), 2025.

[5] L. B. Germano, R. R. Goldschmidt, R. Choren, and J. C. Duarte, "A Systematic Review on Detection, Repair, and Explanation of Vulnerabilities in Source Code Using Large Language Models," *IEEE Access*, vol. 13, pp. 192263–192293, 2025, doi: 10.1109/ACCESS.2025.3631363.

[6] MITRE Corporation, "Common Weakness Enumeration (CWE)." *[Online]*. Available: https://cwe.mitre.org/

[7] FIRST.org, "Common Vulnerability Scoring System v3.1: Specification Document," 2019. [Online]. Available: https://www.first.org/cvss/v3.1/specification-document

[8] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "VulDeePecker: A Deep Learning-Based System for Vulnerability Detection," in *Proc*. Network and Distributed System Security Symp. (NDSS), 2018.

[9] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities," *IEEE Trans*. Dependable Secure Comput., 2021, doi: 10.1109/TDSC.2021.3051525.

[10] 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)*, 2019.

[11] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep Learning Based Vulnerability Detection: Are We There Yet?" *IEEE Trans*. Softw. Eng., vol. 48, no. 9, pp. 3280–3296, 2022.

[12] Y. Li, S. Wang, and T. N. Nguyen, "Vulnerability Detection with Fine-Grained Interpretations," in *Proc*. ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering (ESEC/FSE), 2021.

[13] V.-A. Nguyen, D. Q. Nguyen, V. Nguyen, T. Le, Q. H. Tran, and D. Phung, "ReGVD: Revisiting Graph Neural Networks for Vulnerability Detection," in *Proc*. ICSE Companion, 2022.

[14] Z. Feng et al., "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," in *Findings of EMNLP*, 2020.

[15] D. Guo et al., "GraphCodeBERT: Pre-training Code Representations with Data Flow," in *Int*. Conf. Learning Representations (ICLR), 2021.

[16] Y. Wang, W. Wang, S. Joty, and S. C. H. Hoi, "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation," in *Proc*. EMNLP, 2021.

[17] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified Pre-training for Program Understanding and Generation," in *Proc*. NAACL, 2021.

[18] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "UniXcoder: Unified Cross-Modal Pre-training for Code Representation," in *Proc*. ACL, 2022.

[19] M. Fu and C. Tantithamthavorn, "LineVul: A Transformer-based Line-Level Vulnerability Prediction," in *Proc*. IEEE/ACM Int. Conf. Mining Software Repositories (MSR), 2022.

[20] H. Hanif and S. Maffeis, "VulBERTa: Simplified Source Code Pre-Training for Vulnerability Detection," in *Proc*. Int. Joint Conf. Neural Networks (IJCNN), 2022.

[21] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries," in *Proc*. Int. Conf. Mining Software Repositories (MSR), 2020.

[22] G. Bhandari, A. Naseer, and L. Moonen, "CVEfixes: Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software," in *Proc*. Int. Conf. Predictive Models and Data Analytics in Software Engineering (PROMISE), 2021.

[23] K. Tamberg and H. Bahsi, "Harnessing Large Language Models for Software Vulnerability Detection: A Comprehensive Benchmarking Study," *arXiv:2405.15614*, 2024.

[24] M. Dozono, T. Gasiba, and A. Stocco, "Empirical Multi-Language Evaluation of Large Language Models for Software Vulnerability Detection and CWE Classification," *arXiv:2408.06428*, 2024.

[25] H. Li, Y. Hao, Y. Zhai, and Z. Qian, "Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach," *Proc*. ACM Program. Lang., vol. 8, no. OOPSLA1, 2024.

[26] U. Kulsum, H. Zhu, B. Xu, and M. d'Amorim, "A Case Study of LLM for Automated Vulnerability Repair: Assessing Impact of Reasoning and Patch Validation Feedback," in *Proc*. ACM Int. Conf. AI-Powered Software Engineering (AIware), 2024.