TriVul: Improving Precision of Static Vulnerability Detection through Multi-Agent Reasoning

Xinye Zhao[0000−0001−5507−4814] and Joanna C. S. Santos[0000−0001−8743−2516]⋆

Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN 46556, USA
{xzhao24,joannacss}@nd.edu

Abstract. Vulnerability detection is critical for ensuring software security, yet high false-positive (FP) rates remain a persistent challenge for practical deployment. To address this, we present TriVul, a multi-agent assisted static analysis framework for Java vulnerability detection that integrates heterogeneous static-analyzers with stage-specific LLM reasoning. TriVul employs an LLM-based filter during rule-based candidate search, followed by deterministic, family-aware validation for structural pruning, and introduces a split semantic layer in which an LLM judge validates retained candidates while a separate LLM recovery module revisits deterministically rejected ones. Evaluated on the CWE-Bench-Java dataset, the best-performing TriVul configuration achieves an F1-score of 0.374, with 33.1% precision and 51.32% recall. This strongest setting uses `GPT-4o-mini` for filtering and judging and `Mistral-7B` for recovery. Compared to the reported IRIS+GPT-4 results, TriVul improves recall (from 48.25% to 51.32%), reduces false discoveries, and nearly doubles F1-score (from 0.188 to 0.374). These findings suggest that LLMs are most effective not as standalone detectors, but as role-specialized semantic components embedded within a multi-stage static analysis approach.

Keywords: Java vulnerability detection · static analysis · large language models

1 Introduction

Static Application Security Testing (SAST) tools aim to analyze software artifacts (*e.g.*, code, binaries, bytecode, *etc.*) wihtout actually executing them to find vulnerabilities. Without the need to run the code, SAST tools can scale to large code bases, allowing them to be a useful component of modern secure development pipelines, particularly for enterprise systems written in Java, where applications frequently process untrusted inputs and interact with sensitive resources. Vulnerabilities such as path traversal, cross-site scripting, command

⋆ Corresponding author.

injection, and unsafe deserialization can therefore lead to severe consequences, including data breaches, service disruption, and remote code execution [10, 41].

Despite their widespread adoption, the practical usefulness of SAST tools is undermined by high false-positive rates. This means developers would need to manually inspect large volumes of warnings, many of which do not correspond to exploitable vulnerabilities. Prior studies have shown that this alert fatigue reduces developer trust in static analysis tools and introduces barriers to their adoption [32, 7, 69, 34, 66, 68]. However, recent work shows that false-positive-prone tools can still uncover important vulnerabilities when their outputs are carefully triaged rather than discarded entirely [3]. This suggests that the core challenge is not necessarily finding more vulnerability candidates, but rather automatically separating actionable vulnerabilities from spurious alerts before they are reported to developers [24, 11].

Existing research has largely responded to this challenge by proposing new vulnerability detection techniques tailored to specific vulnerability classes (*e.g.*, SQL injection), analysis paradigms, or benchmark datasets [43, 4, 55, 54, 53]. While these approaches improve detection performance in specific settings, they require a non-trivial amount of engineering effort and assume organizations can replace existing tools already embedded in development pipelines. In practice, many organizations already rely on established SAST tools, making full replacement costly and unrealistic. A more practical direction is to improve the precision of existing tools by automatically filtering false alarms while preserving their ability to identify true vulnerabilities [47].

Recent advances in large language models (LLMs) have created new opportunities for vulnerability analysis due to their ability to reason over source code semantics and natural language context [21, 35, 60]. Recent work has explored using LLMs for vulnerability detection, classification, and repair, as well as hybrid approaches that combine LLMs with program analysis [41, 46, 42]. However, prior studies show that while fully LLM-driven approaches work well in synthetic benchmarks they remain unreliable for realistic vulnerabilities in real-world projects, particularly when repository-level and inter-procedural reasoning are required [35, 41, 13, 76]. Moreover, existing approaches often rely on LLMs as end-to-end validators for downstream filtering, introducing non-determinism without sufficient structural safeguards for consistent false-positive reduction.

This motivates us to create a hybrid approach that combines the scalability of existing SAST tools with more precise downstream filtering. Specifically, we rely on traditional (non-LLM) deterministic analysis for broad vulnerabilities candidate discovery and structural validation, while LLMs can provide semantic reasoning in cases where traditional analyses struggle with ambiguity. Therefore, in this paper, we present **TriVul**, a multi-agent framework that improves the precision of existing static application security testing (SAST) tools by automatically filtering false positives before alerts reach developers. Each agent performs a predefined analysis task within a staged refinement process. Rather than designing a new vulnerability detector for each vulnerability family, TriVul operates on candidate findings produced by existing analysis tools and applies a

sequence of refinement stages combining deterministic validation with targeted LLM reasoning. The framework assigns role-specific agents to distinct tasks, including candidate filtering during vulnerability search, semantic judgment of structurally validated paths, and recovery of candidates rejected by deterministic validation. This design enables TriVul to preserve the scalability benefits of traditional static analysis while improving precision in semantically ambiguous cases.

We evaluate TriVul on CWE-Bench-Java [41], a benchmark of real-world vulnerable Java projects, and compare it against both traditional static-analysis baselines and recent LLM-assisted approaches. Our results show that TriVul achieves better precision-recall tradeoffs and higher F1 scores while maintaining competitive recall across multiple vulnerability categories.

The main contributions of this work[1] are as follows:

- We introduce TriVul, a staged hybrid framework that refines vulnerability candidates produced by existing Java analyzers through deterministic structural validation and role-specific LLM reasoning.
- We demonstrate the effectiveness of this refinement methodology on CWE-Bench-Java [41], achieving substantially improved precision and F1 score over traditional SAST tools and state-of-the-art LLM-assisted baselines while maintaining competitive recall.
- We analyze the robustness of TriVul across different LLM configurations, including proprietary and open-weight models, and study the impact of temperature and model choice on vulnerability refinement performance.

The remainder of this paper is organized as follows. Section 2 introduces background on vulnerability detection and LLM-assisted static analysis. Section 3 presents the design of TriVul. Section 4 describes our experimental setup and methodology. Section 5 presents the findings of empirical evaluation and ablation studies.Section 6 discusses the implications and limitations of our approach. Section 7 discusses related work, and Section 8 concludes the paper.

2 Background

2.1 Static Analysis & Taint-style Vulnerabilities

Static analyzers verifies properties by inspecting source code without executing the program [15]. Consequently, static analysis can reason about multiple possible execution paths and scale to large codebases, making it a widely adopted approach for vulnerability detection. However, reasoning about all possible program behaviors requires approximating program semantics.

To remain scalable, static analysis techniques often rely on conservative abstractions that over-approximate feasible execution paths [19]. In the context

[1] The implementation of TriVul and all data and scripts for this work are available in the replication package [79].

of security analysis, these abstractions enable broad vulnerability discovery, but frequently produce *false positives*, particularly when analyzing programs that rely on reflection, dynamic dispatch, framework-specific behaviors, or complex data transformations [37, 9, 40, 61]. As a result, developers are often required to manually inspect large volumes of alerts to determine which findings correspond to real vulnerabilities.

Static taint analysis (henceforth simply **"taint analysis"**) is a type of program analysis technique that verifies whether information flows from untrusted inputs (*sources*) to security-sensitive operations (*sinks*) [67, 73, 49, 33, 74, 6, 11, 48]. At the heart of taint analysis techniques is the *taint policy*, which specifies how taint is introduced, propagated, and removed throughout program execution. These policies are typically tailored to the underlying client analysis. A taint policy may also define *sanitizers* [63, 64], which are operations that remove taint from variables before they reach sensitive sinks. After propagating taint according to the underlying policy, a program is reported as vulnerable when tainted data reaches a sink without proper sanitization [56]. To illustrate, Fig. 1 shows a vulnerable Java program in which the input `args` is a source of tainted data. The method `rt.exec(...)` is a sink because it executes system commands. Since there is untrusted data flow into this sink (highlighted in red), a taint analyzer would report the program as vulnerable.

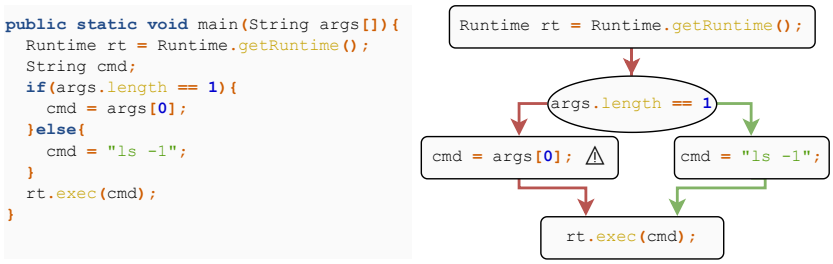


Fig. 1: An example of a vulnerable Java program

Taint analysis is widely used in modern SAST tools because it helps uncover vulnerabilities caused by untrusted flows from untrusted sources of data to security-sensitive operations in the system's source code (*i.e.*, **taint-style vulnerabilities**). However, these analyses frequently over-approximate feasible execution paths and may report infeasible source-to-sink flows that cannot occur in practice. Such imprecision is particularly common when tools lack visibility into application-specific validation logic, framework abstractions, or semantic constraints encoded in the surrounding code [54].

2.2 LLM-Assisted Vulnerability Reasoning

Recent works have explored the use of LLMs for vulnerability detection [81, 71]. Unlike deterministic static analyses, LLMs can reason about semantic ambiguity

in cases where the exploitability of a reported path depends on contextual behaviors such as custom sanitization logic, framework-specific validation routines, or implicit assumptions embedded in application logic. Despite these advantages, relying exclusively on LMMs for vulnerability detection presents practical limitations [59]. LLM-based approaches are computationally expensive, may produce inconsistent outputs, and often lack the scalability guarantees provided by traditional static analysis.

These tradeoffs motivate hybrid approaches that combine the scalability of traditional static analysis for broad candidate discovery with selective LLM reasoning for resolving semantically ambiguous findings [39]. Our work builds on this observation by using deterministic analysis for candidate discovery and structural validation, while selectively invoking LLM agents only when deeper semantic reasoning is required.

3 TriVul Framework

In this section, we first define our problem (§ 3.1) and then describe our framework's architecture (§ 3.2).

3.1 Problem Statement

Let p be a Java project and let v_t denote a target vulnerability type. We define the task of vulnerability detection as whether there exists a source-to-sink path π in p that introduces an exploitable vulnerability related to v_t. Let $\mathcal{C}(p, v)$ be the set of candidate findings produced by multiple static analyzers, where each candidate $c_i \in C$ provides partial evidence about a possible vulnerability instance. Given a project p and a candidate vulnerability path $c \in C(p, v)$, our goal is to determine whether c should be retained as a plausible instance of vulnerability family v through staged structural and semantic validation. Equivalently, rather than treating vulnerability detection as raw alert emission, we treat it as the problem of selecting a subset $\mathcal{C}^*(p, v) \subseteq \mathcal{C}(p, v)$ that maximizes precision while preserving coverage over true vulnerability instances.

This formulation reflects a key practical limitation of existing static analyzers: while they are effective at surfacing broad candidate sets, they often produce many false positives [80, 7], making precise downstream validation the key challenge. Accordingly, TriVul is designed to solve this problem by combining broad candidate generation with staged semantic and deterministic reasoning to validate whether a candidate corresponds to a real vulnerability path.

3.2 Architecture of TriVul

As shown in Fig. 2, TriVul is a three-stage multi-agent vulnerability detection approach that combines multiple static analyzers with targeted semantic refinement. Given a target Java project p, TriVul first searches possible vulnerable paths (candidates) using multiple worker agents. These agents include existing

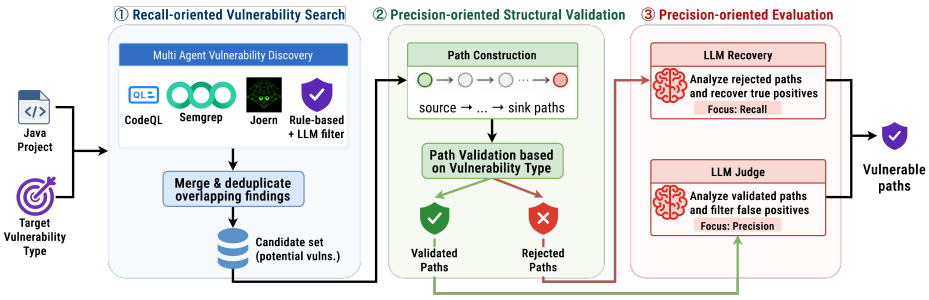


Fig. 2: Overview of the TriVul framework.

static analyzers and a rule-based suspicious function/API detector. The outputs of these agents are then normalized and fused into a shared candidate representation. In the second stage, TriVul applies deterministic structural validation to construct *source-to-sink* paths and filter out candidates that do not match the expected shape of the target vulnerability family. In the third stage, we apply a split LLM-based semantic layer to evaluate the outcomes of the prior stage: an LLM judge evaluates the candidates that survive deterministic validation, while a separate LLM recovery agent revisits candidates rejected by deterministic rules. The final output is the union of semantically accepted deterministic survivors and semantically recovered deterministic rejects.

This design assigns different roles to deterministic analysis and LLM reasoning. Static and rule-based components provide scalable candidate generation and structural constraints, while the LLM is reserved for late-stage semantic decisions on a much smaller search space. Importantly, the LLM is not used as a standalone detector. Instead, it operates after deterministic reduction, either to reject semantically implausible survivors or to recover borderline true positives that deterministic validation may have pruned too aggressively. Because the LLM serves different semantic roles at different points in the approach, Fig. 3 summarizes the compact prompt templates used for Stage I filtering, Stage III path judging, and Stage III rejected-path recovery.

Stage I: Candidate Search The first stage is recall-oriented candidate discovery. Starting from the target project, TriVul loads the relevant Java source files, and launches multiple detection branches in parallel. Our implementation uses four complementary static analyzers in our worker agents: CodeQL [18], Semgrep [58], Joern [72], and a *rule-based vulnerability search*. The rule-based vulnerability search performs lightweight lexical source–sink matching over Java files, seeding taint from source-like APIs and public inputs, propagating it through simple assignments, and checking reachability to CWE-specific sinks. It applies minimal family-aware suppression to filter clearly sanitized cases, and forwards the resulting coarse findings to the Stage I LLM filter. Unlike a purely rule-based detector, however, this analysis branch also contains an LLM-based filtering step

Stage I Filter

System prompt:

```
Judge whether an existing Java static analysis
       finding is semantically plausible. Do not
       search for new bugs.
Return JSON: KEEP / REJECT / UNSURE with
       confidence and reason.
```

User prompt:

```
Target family: {{family}}
Target CWE: {{cwe}}
Finding:
- source statement
- sink statement
- source context
- sink context

Output `KEEP` if plausible, or `REJECT` only if
       clearly wrong. Otherwise, output `UNSURE`.
```

Stage III Judge

System prompt:

```
Judge whether a candidate path is a
       plausible unsanitized
       vulnerable path for the target
       CWE.
Return JSON: ACCEPT / REJECT / UNKNOWN
```

User prompt:

```
Target CWE: {{cwe}}
Vulnerability Family: {{family}}
Sink hints: {{sink_hints}}
Candidate path: {{path_text}}


Rules:
Output `ACCEPT` if plausible, `REJECT`
       only if clearly invalid.
       Otherwise, output `UNKNOWN`.
```

Stage III Recovery

System prompt:

```
System:
Review a path rejected by
the deterministic validator.
Decide whether it should
be rescued.
Return JSON:
RESCUE / KEEP_REJECTED / UNSURE
```

User prompt:

```
User:
Target CWE: {target_cwe}
Family: {family}
Rejected reason
Validator confidence
Candidate path
Source context
Sink context
```

Fig. 3: Prompt templates used in each LLM-based agents in TriVul.

that semantically judges rule-based findings before they enter the merge step. This LLM component does not perform broad search by itself; rather, it acts as an early-stage candidate filter that rejects only findings judged to be clearly spurious while conservatively preserving plausible or ambiguous ones. At this step, we give to the model a prompt as shown in the left panel of Fig. 3. As a result, Stage I already reflects an interaction between rule-based candidate generation and lightweight LLM-based candidate control. For readability, Fig. 3 shows simplified prompt templates.

To better illustrate this first stage, the left panel of Fig. 4 shows a simplified example of this stage, where different worker branches surface overlapping coarse candidates for the same target vulnerability family. In this stage, each worker branch searches for suspicious source-to-sink paths using its own analysis strategy and produces a set of candidate vulnerable paths together with vulnerability type labels and local path metadata (*e.g.*, source lines).

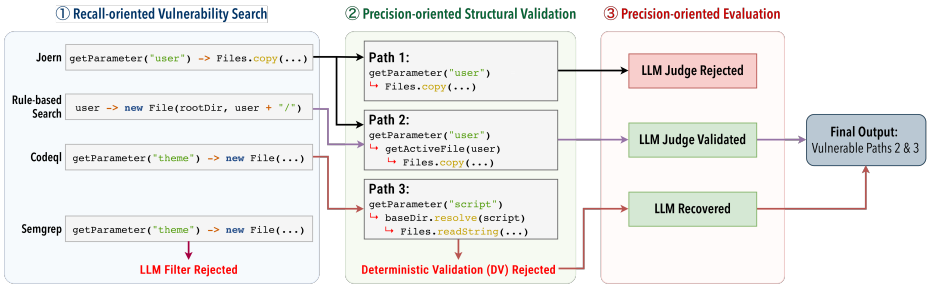


Fig. 4: TriVul walkthrough example.

The purpose of this stage is not to make final vulnerability decisions, but to maximize coverage across heterogeneous vulnerability patterns. Since different worker agents often produce overlapping reports with different path granular-

ities, the end of Stage I performs candidate fusion. The fusion module groups findings by a unified internal vulnerability family and sink location, merges supporting evidence from different tools, and retains the most informative path representation. This yields a consolidated candidate set that becomes the input to deterministic validation. Before downstream validation, analyzer outputs are normalized into a shared internal candidate representation with a common path-based structure and a normalized vulnerability-family label, so that findings from different branches can be fused and compared consistently.

Stage II: Deterministic Structural Validation The second stage applies deterministic validation to the fused candidate set. Its purpose is to enforce family-specific structural constraints before invoking expensive semantic reasoning. First, the framework builds explicit source-to-sink paths from the fused findings. Then, it applies a vulnerability-family-aware validator that checks whether the constructed path is consistent with the expected structural shape of the predicted vulnerability family. Typical checks include whether the final node behaves like a valid family-specific sink, whether the path contains plausible source evidence, and whether there are deterministic signs that the candidate is structurally implausible. The middle panel of Fig. 4 illustrates this stage, where overlapping coarse findings are fused into explicit source-to-sink paths and then partitioned into structurally validated and structurally rejected candidates.

This stage serves as the main precision-oriented deterministic filter. It is deliberately stricter than Stage I, because its role is to reduce obvious noise before the semantic layer. At the same time, we retain the rejected candidates instead of discarding them permanently. This is important because some true positive paths may contain indirect flows, helper methods, or project-specific sink patterns that may not be captured perfectly by deterministic family rules alone. Accordingly, Stage II produces two explicit outputs: a set of structurally validated candidates and a set of structurally rejected candidates.

Stage III: Precision-oriented Path Evaluation The third stage introduces a split LLM-based semantic layer that treats deterministic possibly vulnerable path survivors and deterministic path rejects differently. For candidates that pass deterministic validation, the framework first applies de-duplication and ranking, and then sends the ranked validated paths to an LLM judge. The judge receives the vulnerability family, target CWE context, local source and sink context, and the candidate path itself, and decides whether the path remains semantically plausible as an unsanitized vulnerability. This branch is precision-oriented: it filters structurally valid survivors whose local source-to-sink evidence remains too incomplete, ambiguous, or weakly supported to sustain a confident late-stage semantic judgment. In Fig. 4, this corresponds to the judge branch in the right panel, where one structurally valid but semantically weak path is rejected before final output.

In parallel, the framework applies an LLM recovery agent to candidates rejected by deterministic validation. This recovery branch is recall-oriented and is

designed to compensate for cases in which deterministic structural rules over-prune valid but indirect or weakly expressed vulnerability paths. To control inference cost, the recovery agent does not reconsider all rejected candidates. Instead, it assigns each deterministic reject a lightweight priority score and applies LLM rescue only to the top-K candidates. Rather than re-running broad search, this branch focuses only on a small subset of structurally rejected candidates and asks whether any should be reinstated despite incomplete, indirect, or benchmark-specific structural evidence. Recovered candidates bypass the judge branch and are added directly to the final result set. The right panel of Fig. 4 illustrates this recovery lane, where a deterministically rejected path is reinstated after LLM-based semantic review. The corresponding prompt template is shown in the right panel of Fig. 3. In our implementation, we also include a fallback recovery step for the special case where the judge rejects all deterministically validated candidates; in that case, a small subset of judge-rejected paths is reconsidered using a stricter recovery configuration.

The final output of the approach is therefore constructed from two semantic lanes: (1) vulnerable path candidates accepted by the LLM judge from the deterministic-survivor branch, and (2) vulnerable path candidates recovered by the recovery agent from the deterministic-reject branch. This split design allows the LLM to play two distinct roles at the end of the approach: conservative semantic filtering on high-confidence candidates and targeted recall recovery on borderline rejects. Thus, the framework avoids using a single monolithic LLM filter and instead uses LLM reasoning in a role-aware manner that better matches the error modes of the deterministic stages.

4 Evaluation

We conduct extensive experiments to evaluate the effectiveness of TriVul and to understand the contribution of its stage-specific LLM components.

4.1 Research Questions

Our evaluation aim to answer the following research questions (RQs):

RQ1 How effective is TriVul at detecting real-world Java vulnerabilities on datasets with ground-truth CWE annotations?

RQ2 To what extent does the LLM-based candidate filtering stage improve the quality of vulnerability candidates in TriVul?

RQ3 What are the respective effects of deterministic validation and LLM-based semantic validation in the stage-wise refinement process of TriVul?

RQ4 To what extent do the LLM-based semantic validation and recovery stages reduce false positives in TriVul?

4.2 LLM Model Selection

To answer our RQs, we first selected three representative LLMs for our study: `GPT-4o-mini` [50], `Mistral 7B` [31], and `Code Llama2` [52]. These models cover

both proprietary and open-weight families, as well as different tradeoffs between capability, efficiency, and deployment cost [75, 16]. `GPT-4o-mini` represents a compact proprietary model suitable for repeated inference in multi-stage approachs [50]. Among open-weight models, we include `Mistral 7B` [31] and `Code Llama2` [52], both of which have shown strong performance on reasoning and code-related tasks [77, 23]. This model set allows us to evaluate whether TriVul remains effective across LLM families with different architectures, scales, and cost profiles.

4.3 Datasets

To evaluate TriVul's effectiveness, we use `CWE-Bench-Java` as the primary dataset [41]. This dataset contains 120 manually curated vulnerability instances in Java projects spanning four vulnerability classes (CWEs): path traversal (CWE-22), OS command injection (CWE-78), cross-site scripting (CWE-79), and code injection (CWE-94) [41, 30]. This benchmark is suitable for our study because it was explicitly designed for repository-scale Java vulnerability detection and was used in a state-of-the-art LLM-based vulnerability detection works [41]. Therefore, it provides a natural basis for comparing our approach against both hybrid and static-analysis baselines.

It is important to highlight that seven vulnerability instances could not be used because the corresponding GitHub repositories could not be retrieved successfully (they were no longer available at the time we have conducted our evaluation). Thus, all reported CWE-Bench-Java results are computed on the remaining **113** samples. Table 1 shows number of samples we used in experiments per vulnerability type (CWE).

Table 1: Benchmark composition and usable samples in our experimental setup. Seven instances were excluded due to repository retrieval failures.

CWE	**Vulnerability Class**	**#Samples**	**#Evaluated**
CWE-22	Path Traversal	55	53
CWE-78	OS Command Injection	13	12
CWE-79	Cross-Site Scripting	31	28
CWE-94	Code Injection	21	20
Total		**120**	**113**

4.4 Evaluation Metrics

We evaluate performance using sample-level *recall*, *precision*, and *F1-score*. Since each sample in CWE-Bench Java contains a single known vulnerability, we adopt a *sample-level recall* definition: a sample is considered correctly detected if at

least one reported path overlaps with the ground-truth vulnerable method; otherwise, it is considered missed. Moreover, *precision* is computed as the proportion of reported paths that correspond to true vulnerabilities, *i.e.*, $\text{Prec}(P) = \frac{\#\text{true positive paths}}{\#\text{reported paths}}$. The *F1-score* is computed as the harmonic mean of precision and recall (*i.e.*, $F_1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$), providing a balanced measure of detection capability and path accuracy.

4.5 Baselines

We compare TriVul against four baselines spanning both LLM-based and traditional (non-LLM) vulnerability detection:

- **CodeQL [17].** An industrial static-analysis framework based on declarative queries. We use it as a precision-oriented taint-analysis baseline with strong rules but limited semantic flexibility.
- **Semgrep [8].** A lightweight rule-based analyzer that matches syntactic and semantic code patterns, representing practical pattern-based detection used in real-world pipelines.
- **IRIS+GPT-4 [41].** A state-of-the-art LLM-assisted static-analysis framework for repository-level Java vulnerability detection. We report published results as a hybrid baseline combining static structure with LLM reasoning.
- **IRIS+GPT-4o-mini.** A reproduced IRIS variant using `GPT-4o-mini`, enabling us to isolate the effect of framework design from raw model strength.

5 Results

5.1 Overall Performance (RQ1)

We compare TriVul against standard static-analysis baselines and the reported IRIS results on the matched CWE-Bench-Java samples. For TriVul, we report both the best single-model configurations and the strongest mixed-LLM configuration. The mixed setting uses different LLMs to different roles in each stage: in our best case, `GPT-4o-mini` is used for the LLM filter and LLM judge with *temperature* = 0.2, while `Mistral-7B` is used for LLM recovery with *temperature* = 0.6.

Table 2 summarizes the overall benchmark-level results. Compared with CodeQL and Semgrep, all TriVul variants improve both detection rate and F1-score, showing the value of combining multi-source candidate search with structural and semantic refinement. The strongest overall result is achieved by the mixed-LLM configuration, which detects 58 of 113 matched samples and reaches a detection rate of 51.32%, an average FDR of 66.94%, and an average F1-score of 0.374. This not only exceeds both static-analysis baselines by a large margin, but also slightly surpasses the reported IRIS+GPT-4 result in detection rate (51.32% vs. 48.25%) while delivering a much lower false-discovery rate and nearly doubling F1-score (0.374 vs. 0.188). Importantly, this improvement is not

obtained by using a uniformly stronger model, but by assigning different models to different semantic roles. In particular, the mixed configuration outperforms every single-model TriVul variant, including `GPT-4o-mini`, `Mistral-7B`, and `CodeLlama2-7B` when each is used alone across all LLM stages. This suggests that the filter, judge, and recovery roles benefit from different model characteristics, and that role specialization is more effective than forcing a single LLM to handle all semantic decisions equally well.

Table 2: Overall performance on the 113 matched CWE-Bench-Java samples. Det.=detected vulnerabilities, DR=sample-level detection rate, FDR=false detection rate (lower FDR is better). Subscripts indicate temperature (t).

Method	**Det.**	**DR (%)**	**Avg FDR (%)**	**Avg F1**
CodeQL	12	10.62	89.66	0.105
Semgrep	11	9.82	94.81	0.062
IRIS+GPT-4	55	48.25	83.96	0.188
IRIS+GPT-4o-mini	27	23.89	89.27	0.040
TriVul + Mixed-LLM	**58**	**51.32**	**66.94** ↓ 20.3%	**0.374** ↑ 99.0%
TriVul + GPT-4o-mini$_{t=0.2}$	54	47.79	67.60 ↓ 19.5%	0.361 ↑ 92.0%
TriVul + Mistral-7B$_{t=0.6}$	54	47.79	68.23 ↓ 18.7%	0.356 ↑ 89.4%
TriVul + CodeLlama2-7B$_{t=0.2}$	53	46.90	68.72 ↓ 18.1%	0.350 ↑ 86.2%

The single-model results show the same overall trend. Among them, `GPT-4o-mini` is the strongest, detecting 54 samples with a 47.79% detection rate, a 67.60% average FDR, and an F1-score of 0.361. `Mistral-7B` is very close, while `CodeLlama2-7B` remains competitive but slightly weaker. In contrast, the reproduced IRIS+GPT-4o-mini result performs much worse, detecting only 27 samples with an F1-score of 0.040, which further highlights that simply replacing GPT-4 with a smaller model in IRIS does not preserve effectiveness. Overall, these results show that the main advantage of TriVul is not only stronger raw detection, but a better precision–coverage balance. The gains are especially meaningful because they are achieved while keeping false discoveries much lower than IRIS, indicating a substantially cleaner alert set for downstream use.

Table 3 further breaks down the results by CWE category using sample-level recall and alert-level precision. The strongest gains appear in `CWE-22`, where TriVul achieves the best recall and F1-score among the compared methods. TriVul also performs strongly on `CWE-79`, where it exceeds both CodeQL and IRIS+GPT-4 in balanced effectiveness. For `CWE-94`, TriVul again improves substantially over Semgrep and the reported IRIS result in both precision and F1-score, although absolute recall remains moderate. The most challenging category is `CWE-78`, where all methods perform relatively poorly, suggesting that command-injection-style cases remain difficult for both deterministic and LLM-assisted approaches. Taken together, the per-CWE results are consistent with the benchmark-level findings: the strength of TriVul comes from achieving

Table 3: Per-CWE comparison using alert-level precision and sample-level recall on the matched CWE-Bench-Java samples. "Detected" denotes the number of benchmark vulnerabilities successfully detected for that CWE.

CWE	Method	Detected	Recall	Precision	F1
CWE-22 (53 samples)	CodeQL	9	0.1698	**0.8611**	0.2835
	Semgrep	4	0.0769	0.4211	0.1301
	IRIS+GPT-4	31	0.5849	0.0600	0.1088
	TriVul+GPT-4o-mini	**34**	**0.6415**	0.4427	**0.5239**
CWE-78 (12 samples)	CodeQL	1	0.0833	**0.5000**	0.1429
	Semgrep	**3**	0.2500	0.3077	**0.2759**
	IRIS+GPT-4	**3**	**0.2727**	0.0275	0.0499
	TriVul+GPT-4o-mini	1	0.0833	0.1429	0.1053
CWE-79 (28 samples)	CodeQL	2	0.0714	**1.0000**	0.1333
	Semgrep	2	0.0714	0.3793	0.1202
	IRIS+GPT-4	**13**	**0.4643**	0.0257	0.0487
	TriVul+GPT-4o-mini	**13**	**0.4643**	0.6104	**0.5274**
CWE-94 (20 samples)	CodeQL	0	0.0000	0.0000	0.0000
	Semgrep	2	0.1000	0.0423	0.0594
	IRIS+GPT-4	**8**	**0.3810**	0.0198	0.0376
	TriVul+GPT-4o-mini	6	0.3000	**0.1286**	**0.1800**

better semantic discrimination on the dominant vulnerability categories while preserving competitive overall recall.

5.2 Effectiveness of LLM Filtering in Stage I (RQ2)

To evaluate the contribution of the Stage I LLM filter independently from the later semantic stages, we isolate the approach up to deterministic validation and measure performance immediately after that point (Post-DV). We evaluate Stage I at this point rather than on the raw rule-based search output, because the rule-based search branch participates in multi-tool fusion before deterministic validation, making the quality of the fused candidate pool more meaningful than the standalone size of the upstream candidate set. This design therefore allows us to assess whether the Stage I LLM filter improves the quality of the candidate pool that survives deterministic structural pruning. We compare a rule-only baseline against the best-performing LLM-filtered configuration for each model.

Table 4 shows that the impact of the Stage I LLM filter is strongly model-dependent. Compared with the rule-only baseline, the CodeLlama2-based filter improves all three evaluation metrics after deterministic validation, increasing recall from 0.3805 to 0.3982, precision from 0.3727 to 0.3873, and F1 from 0.3766 to 0.3927. This gain is also reflected in the alert counts: relative to the baseline, the CodeLlama2-based filter yields 8 fewer false-positive alerts and 4 more true-positive alerts. These results suggest that an appropriately chosen LLM filter

Table 4: Effect of the Stage I LLM filter compared with the rule-only baseline. Results are reported after deterministic validation (Post-DV). The baseline uses only rule-based, LLM-free filter, while the LLM-filtered settings use the best-performing temperature for each model.

Configuration	Recall	Precision	F1	FP Alerts	TP Alerts
Rule-based only (Baseline)	0.3805	0.3727	0.3766	239	142
+ GPT-4o-mini filter	0.3805	0.3776	0.3791	239 ±0	145 ↑ 3
+ CodeLlama2 filter	**0.3982**	**0.3873**	**0.3927**	231 ↓ 8	146 ↑ 4
+ Mistral filter	0.3805	0.3691	0.3747	241 ↑ 2	142 ±0

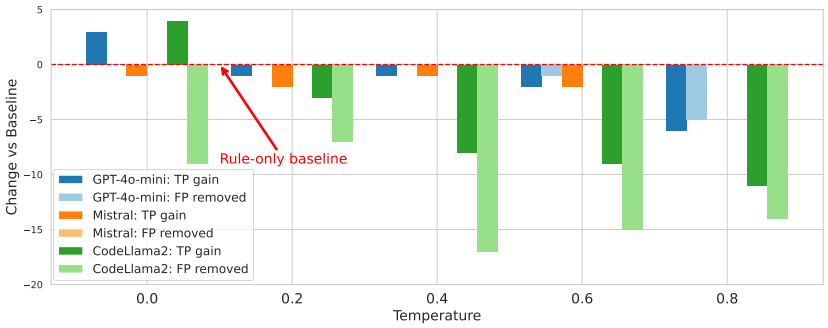


Fig. 5: Sensitivity of the Stage I LLM filter to temperature, measured relative to the rule-only baseline.

can improve the quality of the upstream candidate pool before later semantic refinement.

At the same time, the table also shows that not all LLMs provide the same benefit. The best Mistral-based filter achieves nearly identical recall to the baseline, but slightly lower precision and F1, while also producing two additional false-positive alerts. The best GPT-4o-mini-based filter remains close to the baseline as well: it retains the same recall, increases the true-positive alert count slightly, but does not reduce false-positive alerts. Overall, these results show that Stage I LLM filtering can be beneficial, but its contribution is not universal; rather, it depends strongly on the model used in the filtering role.

Temperature Sensitivity of Candidate Filtering To study temperature sensitivity in the Stage I filtering role, we use the same evaluation setting as in RQ2 and measure performance immediately after deterministic validation, before any downstream semantic judgment or recovery is applied. For each model, we vary only the decoding temperature of the Stage I LLM filter while keeping the rest of the deterministic approach unchanged. Fig. 5 reports the resulting changes relative to the rule-only baseline, allowing us to examine how temperature affects the filter's ability to preserve useful candidates while suppressing

noisy ones. A clear pattern is that the effect of temperature is model-dependent. For the CodeLlama-based filter, lower temperatures generally produce the most favorable behavior, with positive gains over the rule-only baseline in both retained true-positive alerts and removed false-positive alerts. As temperature increases, these gains become smaller and less stable, suggesting that the filter becomes less selective in preserving useful candidates.

The GPT-4o-mini-based filter exhibits a similar but milder trend. At low temperatures, it remains slightly above the baseline in retained true-positive alerts, but the gains are modest and decline as temperature increases. In contrast, the Mistral-based filter stays close to the baseline across all tested temperatures, with little variation and no clear region of strong improvement. This indicates that Mistral is relatively insensitive to temperature in this role, but also less beneficial as a Stage I filter.

Overall, the figure suggests that the Stage I LLM filter is sensitive to temperature, but the degree of sensitivity depends strongly on the underlying model. In our setting, lower temperatures are more favorable for Stage I filtering, especially for CodeLlama, whereas higher temperatures tend to weaken the filter's ability to preserve useful candidates while suppressing noisy ones.

5.3 Stage-wise Comparison of Structural and Semantic Validation (RQ3)

To evaluate the role of deterministic validation (DV), we use a shared stage-wise ablation that measures performance at three checkpoints of the approach: before deterministic validation (*Without DV*), immediately after deterministic validation (*Only-DV*), and after the downstream semantic stage (*DV + Judge + Recovery*). In this subsection, we focus on the first two settings to isolate the effect of Stage II structural pruning. Table 5 shows that the pre-validation stage achieves the highest recall, because the candidate set is still recall-oriented and retains many plausible but weakly supported findings. However, this comes with limited precision, indicating that a substantial amount of structural noise is still present before deterministic validation.

Applying DV substantially reduces recall in all three configurations, confirming that it functions as a deliberately conservative structural pruning stage. For `GPT-4o-mini`, recall drops from 0.5310 to 0.3805, while precision improves from 0.3565 to 0.3776. A similar precision gain is observed for `CodeLlama2`, where precision increases from 0.3070 to 0.3231 after validation. These results indicate that deterministic validation is effective at removing structurally implausible candidates and tightening the candidate set before later semantic analysis. At the same time, the recall reduction shows that this stage is intentionally biased toward precision-oriented pruning rather than standalone end-to-end optimality. For `Mistral`, this tradeoff is even sharper, as both recall and precision decrease slightly after DV, suggesting that fixed structural rules do not always align equally well with all upstream candidate distributions.

Overall, these results suggest that deterministic validation should not be viewed as a final decision stage, but as an intermediate structural bottleneck

within the broader approach. Its purpose is to suppress obvious structural noise and produce a more constrained candidate set for downstream semantic judgment and recovery. Under this interpretation, the intermediate recall loss is not evidence of a flawed design, but evidence that Stage II and Stage III serve complementary roles: deterministic validation narrows the search space, and the later semantic stage repairs overly conservative pruning when necessary.

5.4 Effectiveness of LLM Judge and LLM Recovery in Stage III (RQ4)

We use the same stage-wise ablation in Table 5 to evaluate the contribution of the downstream semantic layer. In this subsection, we focus on the comparison between the *Only-DV (Deterministic Validation)* and *DV + Judge + Recovery* settings in order to isolate the effect of Stage III after deterministic structural pruning. Across all three representative configurations, enabling the Stage III LLM judge and recovery modules consistently improves performance over the DV-only setting. For `GPT-4o-mini`, precision rises from 0.3776 to 0.4047, recall increases from 0.3805 to 0.4602, and F1 improves from 0.3791 to 0.4306, yielding the best overall balance among the three stages. `Mistral` and `CodeLlama2` show the same qualitative trend: although DV alone is intentionally conservative, the downstream semantic stage recovers part of the lost recall while also improving precision relative to the post-validation stage.

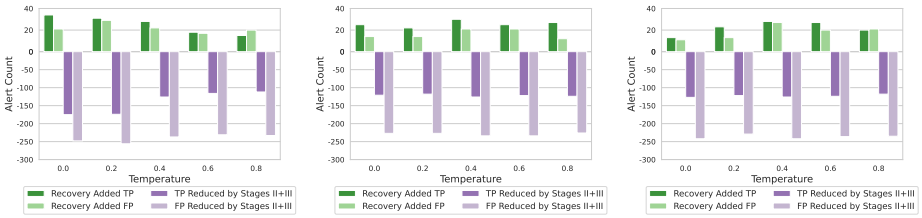
This pattern suggests that the LLM-based downstream stage complements deterministic validation rather than replacing it. While deterministic validation removes many structurally implausible candidates, it also discards borderline true positives. The LLM judge and recovery modules help correct this over-pruning behavior by adding semantic reasoning after structural filtering. As a result, the final system achieves a stronger precision–recall balance than either the recall-oriented pre-validation stage or the deterministic validator alone. In this sense, the main contribution of the downstream LLM stage is not simply additional filtering, but semantic repair and refinement of the structurally constrained candidate set.

Temperature Sensitivity of LLM Judge/Recovery To isolate the effect of late-stage semantic modules in RQ3, we fix the upstream Stage I LLM filter to `GPT-4o-mini` at temperature 0.2 and vary only the model and temperature used in the downstream LLM judge and recovery. Fig. 6 shows that Stage III behavior is also temperature-sensitive, although the trend remains model-dependent. The figure reports the TP/FP reductions produced by Stage II and Stage III together; since Stage II is fixed and deterministic within each setting, the relative differences across temperatures are driven by the Stage III LLM judge. Overall, lower temperatures produce more stable and stronger end-to-end performance, with the best mean F1 observed at T=0.2 and the second-best at T=0.0, while performance degrades progressively at higher temperatures.

At the model level, `GPT-4o-mini` achieves the strongest overall result, with its best configuration at T=0.0 (recall 0.4513, precision 0.4137, F1 0.4317). `Mistral`

Table 5: Stage-wise effect of deterministic validation (DV) and downstream semantic refinement.

Model	Setting	Recall	Precision	F1
GPT-4o-mini	Without DV	0.5310	0.3565	0.4266
	Only-DV	0.3805	0.3776	0.3791
	DV + Judge + Recovery	0.4602	0.4047	**0.4306**
Mistral	Without DV	0.5221	0.3106	0.3899
	Only-DV	0.3363	0.3067	0.3208
	DV + Judge + Recovery	0.4425	0.3543	**0.3935**
CodeLlama2	Without DV	0.5221	0.3070	**0.3869**
	Only-DV	0.3540	0.3231	0.3378
	DV + Judge + Recovery	0.3982	0.3442	0.3692



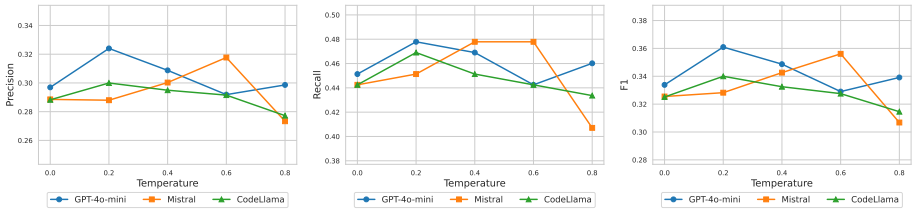
(a) GPT-4o-mini (b) Mistral (c) CodeLlama

Fig. 6: TP and FP changes for three LLM backends in Stage III. Across all models, the LLM recovery stage restores part of lost true positives, while the LLM judge provides an additional semantic filtering layer over remaining candidates.

is more stable across the sweep and performs best at T=0.8, while `CodeLlama` reaches its best result at T=0.6. Taken together, these results suggest that temperature has a measurable effect even when LLMs are used only in the late semantic stages, and that low-temperature decoding remains the most reliable default for Stage III semantic validation.

5.5 End-to-End Sensitivity to LLM Temperature

Fig. 7 shows how end-to-end precision, recall, and F1-score change when varying the temperature of the LLM-based stages across the three model backends. Overall, the framework remains reasonably stable across the tested range, but the temperature effect is still visible and clearly model-dependent. Across all settings, recall varies within a relatively narrow band for most models, while precision and F1 are more sensitive to temperature changes. This suggests that temperature primarily affects the quality of semantic filtering and recovery rather than the system's ability to surface vulnerability candidates at all.



(a) Precision (b) Recall (c) F1-score

Fig. 7: Comparison of precision, recall, and F1-score.

For `GPT-4o-mini`, the best overall setting is at low temperature. At T=0.2, it achieves the highest recall (0.4779), the highest precision (0.3240), and the highest F1-score (0.3609) among all tested configurations. At T=0.0, performance is slightly lower but still strong, with recall 0.4513, precision 0.2969, and F1 0.3338. As temperature increases beyond 0.2, all three metrics decline gradually: F1 drops to 0.3487 at T=0.4, 0.3290 at T=0.6, and 0.3391 at T=0.8. This pattern indicates that `GPT-4o-mini` performs best under relatively conservative decoding, where semantic judgments remain more stable and less noisy.

`Mistral` exhibits a different trend. Rather than peaking at the lowest temperature, it improves steadily from T=0.0 to T=0.6, where it reaches its best overall configuration with recall 0.4779, precision 0.3177, and F1 0.3561. Compared with its low-temperature setting, this represents a clear gain in both precision and F1. However, the improvement is not monotonic across the full range: at T=0.8, performance drops noticeably, with recall falling to 0.4071 and F1 to 0.3068. This suggests that `Mistral` benefits from a moderate amount of stochasticity, but degrades once the temperature becomes too high.

`CodeLlama` shows the mildest overall variation, but also the weakest performance among the three models. Its best configuration appears at T=0.2, where it attains recall 0.4690, precision 0.3000, and F1 0.3400. At T=0.0, 0.4, and 0.6, the changes are relatively small, with F1 remaining in the range of roughly 0.325–0.333, while at T=0.8 the performance drops more clearly to an F1-score of 0.3146. This indicates that `CodeLlama` is somewhat less sensitive than `Mistral`, but also less capable of benefiting from temperature tuning.

Taken together, the results suggest that TriVul is moderately robust to temperature variation, but the optimal operating point differs across models. Among all tested settings, the strongest overall configuration is `GPT-4o-mini` at T=0.2, which achieves the best combined precision, recall, and F1-score. `Mistral` at T=0.6 is competitive and narrows the gap substantially, while `CodeLlama` remains consistently weaker. Based on this sweep, we adopt `GPT-4o-mini` with T=0.2 as the default setting in the remainder of the paper, since it provides the best overall balance between vulnerability coverage and false-positive control.

6 Discussion and Limitations

Our results show that the main strength of TriVul lies in improving the precision–coverage balance through staged validation rather than simply maximizing raw detection counts. Across the benchmark, TriVul achieves a better F1-score and lower false-discovery rate than the reported IRIS configuration while maintaining competitive recall. These gains are most visible for `CWE-22` and `CWE-79`, whereas `CWE-78` remains challenging, suggesting that some vulnerability classes require richer semantic modeling.

A key takeaway is that deterministic validation in Stage II is intentionally designed as a conservative structural bottleneck rather than a standalone final decision stage. Its role is to remove obvious structural noise and narrow the candidate space before semantic refinement, even at the cost of pruning some true positives. At the same time, the current Stage II implementation is still relatively simple, relying on family-aware structural rules and lightweight local search. This simplicity limits its ability to capture interprocedural or highly indirect flows, but does not limit the broader three-stage design principle, which can naturally accommodate stronger deterministic reasoning in future extensions. More broadly, our results suggest that the interaction between Stage I candidate filtering and Stage III semantic judgment and recovery is itself an important design question. In particular, our benchmarking indicates that a mixed-model configuration, in which different LLMs are assigned to different stages based on ablation results, can outperform using a single LLM across all three semantic components. This observation further supports the view that stage-specific model compatibility may matter more than raw model strength alone. However, the mechanism underlying these cross-stage interactions remains unclear, and a more systematic study of joint model assignment, calibration, and end-to-end optimization is an important direction for future work.

Beyond the empirical gains, our results suggest three considerations when integrating LLM reasoning with static analysis. First, LLM-based recovery in vulnerability detection should remain selective rather than revisiting all rejected candidates, since broader recovery can make the approach overly permissive and degrade end-to-end performance. Second, LLM reasoning is most effective after deterministic structural narrowing, where the search space has already been filtered to structurally plausible candidates instead of raw tool outputs. Third, end-to-end effectiveness depends on cross-stage interactions, so the best model for an individual stage is not necessarily the best model for the overall approach. In summary, our findings support a hybrid design in which deterministic analysis provides scalable candidate discovery and pruning, while LLMs are assigned narrow, role-specific semantic tasks.

The comparison between Mistral and CodeLlama2 further suggests that a code-focused model does not automatically achieve better performance than a general-purpose model in vulnerability-analysis approaches. Although CodeLlama2 is more explicitly specialized for code, its advantage is not consistent across filtering, validation, and recovery. This indicates that the critical capa-

bility in our setting is not simply code modeling, but reliable semantic decision making in the presence of noisy candidates and conservative structural pruning.

A further threat to validity is potential benchmark contamination in the evaluated LLMs. However, CWE-Bench-Java was released after Code Llama, Mistral-7B, and the reported knowledge cutoff of GPT-4o-mini, making direct memorization of the benchmark less likely. Nevertheless, benchmark contamination remains a general concern for LLM-based evaluations and should be considered when interpreting the results

7 Related Work

7.1 Traditional (non-LLM) Static Vulnerability Detection for Java

Traditional vulnerability detection for Java has largely been driven by static application security testing (SAST) tools based on pattern matching, taint tracking, and program analysis. Semgrep is representative of lightweight, rule-based SAST, providing semantic pattern matching and practical integration into local and CI-based workflows for Java security scanning [57, 58]. SpotBugs represents a classic bytecode-level bug-pattern analyzer for Java; although it remains useful in practice, its documented security checks are intentionally conservative and often focus on obvious vulnerability instances [62]. CodeQL offers a stronger semantic-analysis baseline by modeling code as a queryable database and supporting dataflow and taint-style reasoning over Java/Kotlin programs [17, 18].

Recent empirical studies highlight both the value and the limitations of these traditional analyzers. Afrose et al. evaluated Java cryptographic misuse detectors and showed substantial variation across tools, especially on interprocedural and path-sensitive cases [1, 25]. Alqaradaghi and Kozsik [2] conducted a broader empirical evaluation of Java static analyzers on Juliet and found that performance varies sharply across CWEs, with no single tool performing uniformly well. These findings suggest that conventional SAST tools provide an important foundation, but often struggle to simultaneously achieve broad coverage and high precision.

7.2 Specialized Static Analysis for Java Vulnerabilities

A related line of work improves precision by designing Java-specific analyses for narrower vulnerability classes or framework settings. Early work by Livshits and Lam showed that static analysis can effectively detect Java web vulnerabilities such as SQL injection and cross-site scripting when supported by precise points-to reasoning and vulnerability-specific specifications [43, 5]. Later work emphasized that precision in Java also depends on modeling the surrounding framework ecosystem: Antoniadis et al. [4] showed that enterprise Java analysis becomes incomplete and imprecise without explicit support for framework abstractions and cache-heavy program structure, while SemTaint improved taint tracking for modern Java web frameworks through rule-based framework modeling and more precise composite-container analysis [28]. Another major direction focuses on vulnerability-specific misuse detection. For cryptographic APIs,

CrySL [36] introduced a declarative specification language for secure API usage, CryptoGuard demonstrated high-precision detection of cryptographic vulnerabilities in large Java projects [51], and Seader combined example-based pattern inference with interprocedural analysis to detect and repair Java cryptographic API misuses [78]. For Java deserialization, GCMiner and Tabby showed that modeling overriding behavior, runtime polymorphism, and gadget-chain structure can substantially improve deserialization vulnerability discovery [12, 14]. These studies show that improving precision in static analysis via Java-specific semantics, framework awareness, or vulnerability-focused reasoning would typically reduces cross-vulnerability generality [45].

7.3 LLM-Based and Hybrid Vulnerability Detection

More recent work explores large language models (LLMs) for vulnerability detection, motivated by their stronger semantic understanding of source code and surrounding context. GRACE augments LLM-based vulnerability detection with graph structure and in-context learning, showing that structural code information can improve detection performance over earlier deep-learning baselines [44]. In the Java setting, Hussain et al. proposed a hybrid deep-learning architecture that combines graph and sequence representations of code, illustrating the growing interest in moving beyond purely rule-based detection toward learned vulnerability reasoning [29, 26]. However, many such approaches are evaluated primarily as classification systems rather than as repository-level detection pipelines.

Other works explored combining LLM reasoning with static analysis rather than replacing static analysis outright. IRIS [41] is a representative neuro-symbolic system that uses LLMs to infer taint specifications and contextual information while retaining a static-analysis backbone for repository-scale vulnerability detection. IRIS also introduces CWE-Bench-Java, a manually validated benchmark of real-world Java vulnerabilities, and demonstrates improved detection over CodeQL on that benchmark [41]. More recent work has continued this hybrid direction. QLPro [27] integrates LLM reasoning with static analysis to automate project-specific vulnerability discovery , while Vul-RAG [20] shows that external knowledge and structured guidance can substantially improve raw LLM judgments. At the same time, recent empirical studies at project scale report that standalone or weakly constrained LLM detectors still suffer from shallow interprocedural reasoning and high false-discovery rates [38, 22]. Taken together, these results suggest that LLMs are most effective when used to refine, contextualize, or selectively recover candidates produced by traditional program analysis, rather than to replace static analysis entirely.

7.4 Agent-Based Vulnerability Detection

A further step is to organize LLM reasoning as an explicit agent workflow rather than a single-pass prediction. In this setting, agents can iteratively gather repository context, inspect interprocedural flows, formulate vulnerability hypotheses, and validate those hypotheses against surrounding code. Yildiz et al. [76] provide

one of the clearest evaluations of this design space by benchmarking standard LLMs against ReAct-style agents for repository-level vulnerability detection, showing that agents perform better when relevant context must be retrieved and analyzed incrementally across multiple functions . However, they also show that both standalone LLMs and agents remain inconsistent, with common failures including missed vulnerabilities, over-analysis of defensive code, and unstable reasoning [76]. Related work in adjacent security settings points in a similar direction. Toprani and Madisetti [65] propose an agentic workflow for vulnerability detection and remediation in infrastructure-as-code, combining LLMs with retrieval and a knowledge base to improve contextual reasoning over security misconfigurations. More recent agentic systems also explore stronger decomposition strategies, such as hypothesis-validation approaches and multi-agent specialization for different security perspectives, further suggesting that agent workflows can improve vulnerability localization and reasoning when repository context is large and heterogeneous [70]. Taken together, these studies suggest that agent-based systems are promising for repository-scale vulnerability analysis, but that they still require grounding, structured retrieval, and explicit validation mechanisms to control false positives and stabilize decision quality.

8 Conclusion

We presented TriVul, a three-stage multi-agent assisted static analysis framework for Java vulnerability detection. TriVul combines recall-oriented candidate search, deterministic structural validation, and LLM-based semantic judgment and recovery to improve the precision–recall balance of vulnerability detection. Evaluated on matched CWE-Bench-Java samples, TriVul achieves substantially higher F1-score and lower false-discovery rate than standard static-analysis baselines and the reported IRIS configuration, while maintaining competitive recall.

Our results show that the effectiveness of LLMs in static analysis is highly role-dependent. In TriVul, LLMs are most useful not as standalone end-to-end detectors, but as targeted semantic components for candidate filtering, semantic validation, and reject recovery. We further show that deterministic validation, while simple and conservative, becomes effective when embedded in a broader staged approach that allows downstream semantic recovery to repair over-pruned true positives. Overall, this work suggests that staged validation is a practical design principle for integrating LLMs with static analysis, and provides a foundation for future extensions with stronger deterministic analysis and richer semantic reasoning.

References

1. Afrose, S., Xiao, Y., Rahaman, S., Miller, B.P., Yao, D.: Evaluation of static vulnerability detection tools with java cryptographic api benchmarks. IEEE Transactions on Software Engineering **49**(2), 485–497 (2022)

2. Alqaradaghi, M., Kozsik, T.: Comprehensive evaluation of static analysis tools for their performance in finding vulnerabilities in java code. IEEE Access **12**, 55824–55842 (2024)
3. Ami, A.S., Moran, K., Poshyvanyk, D., Nadkarni, A.: " false negative-that one is going to kill you": Understanding industry perspectives of static analysis based security testing. In: 2024 IEEE Symposium on Security and Privacy (SP). pp. 3979–3997. IEEE (2024)
4. Antoniadis, A., Filippakis, N., Krishnan, P., Ramesh, R., Allen, N., Smaragdakis, Y.: Static analysis of java enterprise applications: frameworks and caches, the elephants in the room. In: Proceedings of the 41st ACM SIGPLAN conference on programming language design and implementation. pp. 794–807 (2020)
5. Arzt, S., Miltenberger, M., Näumann, J.: Dynamically generating callback summaries for enhancing static analysis (artifact). In: Dagstuhl Artifacts Ser. (2024), https://api.semanticscholar.org/CorpusID:272596962
6. Avancini, A., Ceccato, M.: Towards security testing with taint analysis and genetic algorithms. In: Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems. pp. 65–71 (2010)
7. Baca, D., Petersen, K., Carlsson, B., Lundberg, L.: Static code analysis to detect software security vulnerabilities-does experience matter? In: 2009 International Conference on Availability, Reliability and Security. pp. 804–810. IEEE (2009)
8. Bennett, G., Hall, T., Winter, E., Counsell, S.: Semgrep*: Improving the limited performance of static application security testing (sast) tools. Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering (2024), https://api.semanticscholar.org/CorpusID:270490727
9. Bodden, E., Sewe, A., Sinschek, J., Oueslati, H., Mezini, M.: Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In: Proceedings of the 33rd International Conference on Software Engineering. pp. 241–250. ICSE'11, ACM, New York, NY, USA (2011). https://doi.org/10.1145/1985793.1985827
10. Bui, Q.C., Scandariato, R., Ferreyra, N.E.D.: Vul4j: A dataset of reproducible java vulnerabilities geared towards the study of program repair techniques. 2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR) pp. 464–468 (2022), https://api.semanticscholar.org/CorpusID:248660221
11. Cai, L., Song, F., Chen, T.: Towards efficient verification of constant-time cryptographic implementations. Proceedings of the ACM on Software Engineering **1**, 1019 – 1042 (2024), https://api.semanticscholar.org/CorpusID:267770495
12. Cao, S., Sun, X., Wu, X., Bo, L., Li, B., Wu, R., Liu, W., He, B., Ouyang, Y., Li, J.: Improving java deserialization gadget chain mining via overriding-guided object generation. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). pp. 397–409. IEEE (2023)
13. Ceka, I., Qiao, F., Dey, A., Valechia, A., Kaiser, G.E., Ray, B.: Can llm prompting serve as a proxy for static analysis in vulnerability detection. ArXiv **abs/2412.12039** (2024), https://api.semanticscholar.org/CorpusID:274788879
14. Chen, X., Wang, B., Jin, Z., Feng, Y., Li, X., Feng, X., Liu, Q.: Tabby: Automated gadget chain detection for java deserialization vulnerabilities. In: 2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). pp. 179–192. IEEE (2023)
15. Chess, B., McGraw, G.: Static analysis for security. IEEE Security Privacy **2**(6), 76–79 (2004)

16. Chiang, W.L., Zheng, L., Sheng, Y., Angelopoulos, A.N., Li, T., Li, D., Zhang, H., Zhu, B., Jordan, M.I., Gonzalez, J.E., Stoica, I.: Chatbot arena: An open platform for evaluating llms by human preference. In: International Conference on Machine Learning (2024), https://api.semanticscholar.org/CorpusID:268264163
17. CodeQL: About codeql (2026), https://codeql.github.com/docs/codeql-overview/about-codeql/, accessed: 2026-04-21
18. CodeQL: Codeql query help for java and kotlin (2026), https://codeql.github.com/codeql-query-help/java/, accessed: 2026-04-21
19. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of programming languages. pp. 269–282 (1979)
20. Du, X., Zheng, G., Wang, K., Zou, Y., Wang, Y., Deng, W., Feng, J., Liu, M., Chen, B., Peng, X., et al.: Vul-rag: Enhancing llm-based vulnerability detection via knowledge-level rag. ACM Transactions on Software Engineering and Methodology (2024)
21. Fu, M., Tantithamthavorn, C.K., Nguyen, V., Le, T.: Chatgpt for vulnerability detection, classification, and repair: How far are we? In: 2023 30th Asia-Pacific Software Engineering Conference (APSEC). pp. 632–636. IEEE (2023)
22. Gnieciak, D., Szandala, T.: Large language models versus static code analysis tools: A systematic benchmark for vulnerability detection. IEEE Access **13**, 198410–198422 (2025)
23. Guo, J., Li, Z., Liu, X., Ma, K., Zheng, T., Yu, Z., Pan, D., Li, Y., Liu, R., Wang, Y., Guo, S., Qu, X., Yue, X., Zhang, G., Chen, W., Fu, J.: Codeeditorbench: Evaluating code editing capability of large language models. ArXiv **abs/2404.03543** (2024), https://api.semanticscholar.org/CorpusID:268889606
24. Guo, Z., Tan, T., Liu, S., Liu, X., Lai, W., Yang, Y., Li, Y., Chen, L., Dong, W., Zhou, Y.: Mitigating false positive static analysis warnings: Progress, challenges, and opportunities. IEEE Transactions on Software Engineering **49**(12), 5154–5188 (2023)
25. He, W., Di, P., Ming, M., Zhang, C., Su, T., Li, S., Sui, Y.: Finding and understanding defects in static analyzers by constructing automated oracles. Proceedings of the ACM on Software Engineering **1**, 1656 – 1678 (2024), https://api.semanticscholar.org/CorpusID:271169421
26. Heo, K., Oh, H., Yi, K.: Machine-learning-guided selectively unsound static analysis. 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE) pp. 519–529 (2017), https://api.semanticscholar.org/CorpusID:2804863
27. Hu, J., Jin, X., Zeng, Y., Liu, Y., Li, Y., Du, D., Xie, K., Zhu, H.: Qlpro: Automated code vulnerability discovery via llm and static code analysis integration. arXiv preprint arXiv:2506.23644 (2025)
28. Huang, H., Yan, R., Gao, J.: Semtaint: A scalable taint analysis approach for javaweb frameworks and composite containers. Computers & Security p. 104821 (2026)
29. Hussain, S., Nadeem, M., Baber, J., Hamdi, M., Rajab, A., Al Reshan, M.S., Shaikh, A.: Vulnerability detection in java source code using a quantum convolutional neural network with self-attentive pooling, deep sequence, and graph-based hybrid feature extraction. Scientific Reports **14**(1), 7406 (2024)
30. IRIS-SAST: Cwe-bench-java (2026), https://github.com/iris-sast/cwe-bench-java, accessed: 2026-04-21
31. Jiang, A.Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D.S., de las Casas, D., Bressand, F., Lengyel, G., Lample, G., Saulnier, L., Renard Lavaud, L., Lachaux, M.A., Stock, P., Le Scao, T., Lavril, T., Wang, T.,

Lacroix, T., El Sayed, W.: Mistral 7b. arXiv preprint arXiv:2310.06825 (2023), https://arxiv.org/abs/2310.06825
32. Johnson, B., Song, Y., Murphy-Hill, E., Bowdidge, R.: Why don't software developers use static analysis tools to find bugs? In: 2013 35th International Conference on Software Engineering (ICSE). pp. 672–681 (2013). https://doi.org/10.1109/ICSE.2013.6606613
33. Jovanovic, N., Kruegel, C., Kirda, E.: Static analysis for detecting taint-style vulnerabilities in web applications. Journal of Computer Security **18**(5), 861–907 (2010)
34. Kang, H.J., Aw, K.L., Lo, D.: Detecting false alarms from automatic static analysis tools: How far are we? In: Proceedings of the 44th International Conference on Software Engineering. pp. 698–709 (2022)
35. Khare, A., Dutta, S., Li, Z., Solko-Breslin, A., Alur, R., Naik, M.: Understanding the effectiveness of large language models in detecting security vulnerabilities. arxiv. arXiv preprint arXiv:2311.16169 (2023)
36. Krüger, S., Späth, J., Ali, K., Bodden, E., Mezini, M.: Crysl: An extensible approach to validating the correct usage of cryptographic apis. IEEE Transactions on Software Engineering **47**(11), 2382–2400 (2019)
37. Landman, D., Serebrenik, A., Vinju, J.J.: Challenges for static analysis of java reflection: Literature review and empirical study. In: Proceedings of the 39th International Conference on Software Engineering. pp. 507–518. ICSE'17, IEEE Press (2017). https://doi.org/10.1109/ICSE.2017.53
38. Li, F., Jiang, J., Chen, D., Xiong, Y.: Llm-based vulnerability detection at project scale: An empirical study. arXiv preprint arXiv:2601.19239 (2026)
39. Li, H., Zhang, H., Pei, K., Qian, Z.: Towards more accurate static analysis for taint-style bug detection in linux kernel. In: 2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 380–392. IEEE (2025)
40. Li, Y., Tan, T., Xue, J.: Understanding and analyzing java reflection. ACM Transactions on Software Engineering and Methodology (TOSEM) **28**(2), 1–50 (2019)
41. Li, Z., Dutta, S., Naik, M.: Iris: Llm-assisted static analysis for detecting security vulnerabilities (2025)
42. Liu, P., Sun, C., Zheng, Y., Feng, X., Qin, C., Wang, Y., Li, Z., Sun, L.: Harnessing the power of llm to support binary taint analysis. ArXiv **abs/2310.08275** (2023), https://api.semanticscholar.org/CorpusID:263909501
43. Livshits, V.B., Lam, M.S.: Finding security vulnerabilities in java applications with static analysis. In: USENIX security symposium. vol. 14, pp. 18–18 (2005)
44. Lu, G., Ju, X., Chen, X., Pei, W., Cai, Z.: Grace: Empowering llm-based software vulnerability detection with graph structure and in-context learning. Journal of Systems and Software **212**, 112031 (2024)
45. Miltenberger, M., Arzt, S.: Vusc: An extensible research platform for java-based static analysis. 2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE) pp. 3977–3980 (2025), https://api.semanticscholar.org/CorpusID:285106360
46. Mohajer, M.M., Aleithan, R., Harzevili, N.S., Wei, M., Belle, A.B., Pham, H.V., Wang, S.: Skipanalyzer: A tool for static code analysis with large language models (2023), https://api.semanticscholar.org/CorpusID:264590515
47. Muske, T., Serebrenik, A.: Survey of approaches for postprocessing of static analysis alarms. ACM Comput. Surv. **55**(3) (Feb 2022). https://doi.org/10.1145/3494521, https://doi.org/10.1145/3494521

48. Navas, J.A., Gehani, A.: Occam-v2: Combining static and dynamic analysis for effective and efficient whole-program specialization. Communications of the ACM **66**, 40 – 47 (2023), https://api.semanticscholar.org/CorpusID:254361535
49. Newsome, J., Song, D.X., et al.: Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In: NDSS. vol. 5, pp. 3–4 (2005)
50. OpenAI: Gpt-4o mini: advancing cost-efficient intelligence. https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/ (2024), openAI release post, accessed 2026-04-28
51. Rahaman, S., Xiao, Y., Afrose, S., Shaon, F., Tian, K., Frantz, M., Kantarcioglu, M., Yao, D.: Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. pp. 2455–2472 (2019)
52. Roziere, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X.E., Adi, Y., Liu, J., Sauvestre, R., Remez, T., et al.: Code llama: Open foundation models for code. arXiv preprint arXiv:2308.12950 (2023)
53. Santos, J.C., Jones, R.A., Mirakhorli, M.: Salsa: static analysis of serialization features. In: Proceedings of the 22nd ACM SIGPLAN International Workshop on Formal Techniques for Java-Like Programs. pp. 18–25 (2020)
54. Santos, J.C., Mirakhorli, M., Shokri, A.: Seneca: Taint-based call graph construction for java object deserialization. Proceedings of the ACM on Programming Languages **8**(OOPSLA1), 1125–1153 (2024)
55. Sayar, I., Bartel, A., Bodden, E., Traon, Y.L.: An in-depth study of java deserialization remote-code execution exploits and vulnerabilities. ACM Transactions on Software Engineering and Methodology **32**, 1 – 45 (2022), https://api.semanticscholar.org/CorpusID:251322251
56. Schwartz, E.J., Avgerinos, T., Brumley, D.: All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: Proceedings of the 2010 IEEE Symposium on Security and Privacy. pp. 317–331. SP '10, IEEE Computer Society, Washington, DC, USA (2010). https://doi.org/10.1109/SP.2010.26, http://dx.doi.org.ezproxy.rit.edu/10.1109/SP.2010.26
57. Semgrep: Introduction to semgrep (2026), https://semgrep.dev/docs/introduction, accessed: 2026-04-21
58. Semgrep: Java support (2026), https://semgrep.dev/docs/languages/java, accessed: 2026-04-21
59. Sheng, Z., Chen, Z., Gu, S., Huang, H., Gu, G., Huang, J.: Llms in software security: A survey of vulnerability detection techniques and insights. ACM Computing Surveys **58**(5), 1–35 (2025)
60. Singh, G.: Building trust and safety in artificial intelligence with abstract interpretation. In: Sensors Applications Symposium (2023), https://api.semanticscholar.org/CorpusID:264382524
61. Smaragdakis, Y., Balatsouras, G., Kastrinis, G., Bravenboer, M.: More sound static handling of java reflection. In: Feng, X., Park, S. (eds.) Programming Languages and Systems. pp. 485–503. Springer International Publishing, Cham (2015)
62. SpotBugs Project: Bug descriptions — spotbugs 4.9.8 documentation (2026), https://spotbugs.readthedocs.io/en/latest/bugDescriptions.html, accessed: 2026-04-21
63. Thomé, J., Shar, L.K., Bianculli, D., Briand, L.C.: Joanaudit: A tool for auditing common injection vulnerabilities. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. pp. 1004–1008 (2017)

64. Thomé, J., Shar, L.K., Bianculli, D., Briand, L.: An integrated approach for effective injection vulnerability analysis of web applications through security slicing and hybrid constraint solving. IEEE Transactions on Software Engineering **46**(2), 163–195 (2020)
65. Toprani, D., Madisetti, V.K.: Llm agentic workflow for automated vulnerability detection and remediation in infrastructure-as-code. IEEE Access (2025)
66. Tripp, O., Guarnieri, S., Pistoia, M., Aravkin, A.: Aletheia: Improving the usability of static security analysis. In: Proceedings of the 2014 ACM SIGSAC conference on computer and communications security. pp. 762–774 (2014)
67. Tripp, O., Pistoia, M., Fink, S.J., Sridharan, M., Weisman, O.: TAJ: effective taint analysis of web applications. ACM Sigplan Notices **44**(6), 87–97 (2009)
68. Urban, C., Subotić, P., Drobnjaković, F.: Static analysis by abstract interpretation against data leakage in machine learning. Sci. Comput. Program. **246**, 103338 (2025), https://api.semanticscholar.org/CorpusID:278952209
69. Wadhams, Z.D., Izurieta, C., Reinhold, A.M.: Barriers to using static application security testing (sast) tools: A literature review. In: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops. pp. 161–166 (2024)
70. Wang, Z., Li, G., Li, J., Zhu, H., Jin, Z.: Vulagent: Hypothesis-validation based multi-agent vulnerability detection. arXiv preprint arXiv:2509.11523 (2025)
71. Xu, H., Wang, S., Li, N., Wang, K., Zhao, Y., Chen, K., Yu, T., Liu, Y., Wang, H.: Large language models for cyber security: A systematic literature review. ACM Transactions on Software Engineering and Methodology (2024)
72. Yamaguchi, F., Golde, N., Arp, D., Rieck, K.: Modeling and discovering vulnerabilities with code property graphs. 2014 IEEE Symposium on Security and Privacy pp. 590–604 (2014), https://api.semanticscholar.org/CorpusID:2231082
73. Yamaguchi, F., Maier, A., Gascon, H., Rieck, K.: Automatic inference of search patterns for taint-style vulnerabilities. In: 2015 IEEE symposium on security and privacy. pp. 797–812. IEEE (2015)
74. Yamaguchi, F., Wressnegger, C., Gascon, H., Rieck, K.: Chucky: Exposing missing checks in source code for vulnerability discovery. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. pp. 499–510 (2013)
75. Yan, W., Liu, H., Wang, Y., Li, Y., Chen, Q., Wang, W., Lin, T., Zhao, W., Zhu, L., Deng, S., Sundaram, H.: Codescope: An execution-based multilingual multitask multidimensional benchmark for evaluating llms on code understanding and generation. ArXiv **abs/2311.08588** (2023), https://api.semanticscholar.org/CorpusID:265212753
76. Yildiz, A., Teo, S.G., Lou, Y., Feng, Y., Wang, C., Divakaran, D.M.: Benchmarking llms and llm-based agents in practical vulnerability detection for code repositories. In: Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). pp. 30848–30865 (2025)
77. Yin, X., Ni, C., Wang, S.: Multitask-based evaluation of open-source llm on software vulnerability. IEEE Transactions on Software Engineering **50**, 3071–3087 (2024), https://api.semanticscholar.org/CorpusID:268857084
78. Zhang, Y., Xiao, Y., Kabir, M.M.A., Yao, D., Meng, N.: Example-based vulnerability detection and repair in java code. In: Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension. pp. 190–201 (2022)
79. Zhao, X., Santos, J.C.S.: TRIVUL: Replication package. https://github.com/s2e-lab/TRIVUL (2026), code, data, and scripts. Accessed: 2026-07-20

80. Zheng, J., Williams, L., Nagappan, N., Snipes, W., Hudepohl, J.P., Vouk, M.A.: On the value of static analysis for fault detection in software. IEEE transactions on software engineering **32**(4), 240–253 (2006)
81. Zhou, X., Cao, S., Sun, X., Lo, D.: Large language model for vulnerability detection and repair: Literature review and the road ahead. ACM Transactions on Software Engineering and Methodology **34**(5), 1–31 (2025)