VulAgent: Hypothesis-Validation Driven Multi-Agent Architecture for Vulnerability Detection

Ziliang Wang
Central South University, China
wangziliang@csu.edu.cn

Ge Li*
Peking University, China
lige@pku.edu.cn

Jia Li
Peking University, China
lijia@stu.pku.edu.cn

Hao Zhu
Peking University, China
haozhu@stu.pku.edu.cn

Zhi Jin
Peking University, China
zhijin@sei.pku.edu.cn

Abstract

Vulnerability detection with language models is challenging: it requires (i) precisely localizing security-sensitive code and (ii) reasoning about potential vulnerability conditions under complex, partially observed program context. We present VulAgent, a multi-agent vulnerability detection framework based on hypothesis validation. Our design is inspired by how human auditors review code: when noticing a sensitive operation, they form a hypothesis about a possible vulnerability, consider potential trigger paths, and then verify the hypothesis against the project context. Given a code unit, VulAgent first applies multi-view analyzers to identify and localize security-sensitive operations from complementary perspectives. For each sensitive operation, it then constructs an explicit vulnerability hypothesis—including triggering (or exploitation) preconditions and a candidate trigger path—and validates the hypothesis using project context together with the model's general knowledge of commonly used APIs and security patterns. This validation-oriented design reduces speculative reports and substantially lowers false positives. Across PrimeVul and SVEN, VulAgent improves accuracy by 6.6 percentage points on average, increases vulnerable–fixed pair identification by up to 4.5× (2.46× on average), and reduces false positive rate by 36% relative to recent LLM-based baselines.

1 Introduction

Software vulnerabilities caused by insecure coding practices remain a major security threat, enabling consequences such as unauthorized information disclosure (Fu and Tantithamthavorn, 2022) and cyber extortion (Thapa et al., 2022). Recent reports indicate that vulnerabilities continue to grow rapidly, underscoring the need for effective and practical detection tools (Cheng et al., 2022; Fu and Tantithamthavorn, 2022; Thapa et al., 2022; Jang-Jaccard and Nepal, 2014; Johnson et al., 2011; Cao et al., 2025).

```
1  public History getModifications(String wiki, String spaces, String page, User u) {
2      final String pageId = Utils.getPageId(wiki, spaces, page);
3      final DocRef docRef  = resolver.resolve(pageId);
…
74     if (!authManager.hasAccess(u, docRef, PERM.VIEW_HISTORY)) {
75     throw new ForbiddenException(); }
…
106    return Factory.createHistorySummary(wiki, spaces, page, audit.fetch(docRef)); }
```

Figure 1: A simplified code snippet for CWE-862 (Missing Authorization).

Prior LLM-based work often formulates vulnerability detection primarily as a semantic recognition problem (Ding et al., 2025; Zhu et al., 2025); inspired by expert auditing workflows, we instead decompose it into a validation-driven detection task grounded in checkable program evidence. A central difficulty is not merely semantic suspicion, but establishing—with checkable evidence—whether the suspected vulnerability is reachable and triggerable given the surrounding context and defenses. Detectors must (i) localize security-sensitive operations and (ii) reliably validate whether the required guards or defensive logic actually apply under the surrounding program context.

Figure 1 illustrates a typical CWE-862 scenario. The function resolves a resource identifier to docRef, checks access at L74–L75, and later invokes a sensitive operation at L106. If the check is missing or bypassed, an unauthenticated user may reach L106 and access protected data. This example highlights a common failure mode: without reliably linking a sensitive operation to its corresponding guard in context, detectors either miss true issues (false negatives) or over-flag benign code (false positives). Therefore, the core question is: *How can we accurately localize risky code and reliably validate vulnerability hypotheses?*

Existing LLM-based detectors struggle with this trade-off. Single-agent CoT-style prompting lacks explicit context validation and often yields high

* Corresponding author.

Findings of the Association for Computational Linguistics: ACL 2026, pages 18598–18616

false-positive rates (Ding et al., 2025). Retrieval-augmented prompting concatenates context into the prompt, but tightly coupling discovery and validation can increase reasoning burden while still leaving many false positives (Li et al., 2025). Recent multi-agent systems (e.g., iAudit (Ma et al., 2024), MuCoLD (Mao et al., 2024), VulTrial (Widyasari et al., 2025)) introduce reflection or discussion to filter outputs, yet the filtering is often performed over generated rationales rather than explicitly validating path-/guard-level evidence, limiting effective use of program context.

We advocate a paradigm shift in LLM-based vulnerability detection, moving beyond semantic recognition to a workflow-inspired hypothesis-validation formulation, where each report is an explicit, falsifiable hypothesis. Concretely, we cast detection as a three-stage, hypothesis-driven workflow: discovery, hypothesis construction, and validation. Crucially, we relax discovery from vulnerability adjudication to high-recall sensitive-code spotting, deferring correctness to later stages that instantiate a hypothesis (triggering conditions + a candidate trigger path) and validate each component against project context or other useful information. This reframes context use from free-form justification into a sequence of falsifiable checks, which suppresses speculative reports and reduces false positives.

In summary, our contributions are:

- We formulate LLM-based vulnerability detection as falsifiable hypothesis validation, where each candidate report is converted into a structured hypothesis (conditions + trigger path) that must survive evidence checks before being output.
- We instantiate the formulation as a modular multi-agent system that decouples discovery and validation, improving candidate coverage via role-specialized analyzers and reducing false positives via condition validation and hypothesis-scoped path validation.
- Experiments on PrimeVul and SVEN show that VulAgent consistently outperforms strong LLM-based baselines, improving average accuracy by 6.6 percentage points, increasing vulnerable–fixed pair identification by up to $4.5\times$ ($2.46\times$ on average), and reducing false positive rate by about 36%.

All resources used and code in this work are available at `https://github.com/HotFrom/VulAgent`.

2 Related Work

Early systems largely relied on handcrafted rules and pattern templates (e.g., Checkmarx, Flawfinder) to flag suspicious code idioms (Checkmarx, 2022; Flawfinder, 2022). While practical for known patterns, these heuristics demand substantial expert effort and often fail to generalize across projects or coding styles. Moreover, rule-triggering syntactic motifs frequently recur in benign code, which inflates both false positives and false negatives (Yamaguchi, 2015, 2017; Li et al., 2021). To reduce manual effort, recent work learns vulnerability signals directly from code (Dam et al., 2017; Russell et al., 2018; Shi et al., 2026). Broadly, methods fall into two families: sequence encoders and structure-aware models. This paper focuses on pure LLM-based vulnerability detection, where models reason over code and output vulnerability judgments, rather than integrating full-fledged static/dynamic analyzers as decision engines.

Single-agent. A growing body of work examines whether careful prompting and chain-of-thought (CoT) can turn general LLMs into effective vulnerability detectors (Ding et al., 2025; Nong et al., 2024; Steenhoek et al., 2024b). Tamberg and Bahsi benchmark CoT, tree-of-thought, and self-consistency, finding CoT with GPT-4 strongest among prompting variants and competitive with static analysis (Tamberg and Bahsi, 2025). Zhou et al. report GPT-4 with prompting outperforming fine-tuned CodeBERT (Zhou et al., 2024), while Ullah et al. observe solid performance across commercial LLMs but frequent unfaithful reasoning (Ullah et al., 2024). Subsequent analyses question CoT's reliability for vulnerability tasks (Nong et al., 2024; Steenhoek et al., 2024b; Zhang et al., 2024); on the PrimeVul pair set, GPT-3.5/4 perform poorly, underscoring these limits.

Multi-agent collaboration. To address reasoning brittleness and improve specialization, multi-agent designs coordinate complementary roles (He et al., 2025). EvalSVA organizes agents as security specialists (Wen et al., 2024), and MuCoLD assigns tester/developer roles for collaborative analysis (Mao et al., 2024). Related ideas appear in smart-contract auditing: LLM-SmartAudit leverages role-play and collaborative reasoning (Wei et al., 2024), iAudit uses detector/reasoner/critic

roles (Ma et al., 2024), and GPTLens deploys multiple prosecutor agents with a critic (Hu et al., 2023). VulTrial introduces a courtroom-inspired framework where role-specific agents collaborate to improve automated vulnerability detection (Widyasari et al., 2025). This method further reduces the false alarm rate by introducing a third party. Despite improved collaboration, most single- and multi-agent LLM approaches primarily refine generated rationales (e.g., via reflection, debate, or critique) and often lack an explicit mechanism to validate guard-/path-level evidence against context, which contributes to persistent false positives.

3 Approach

In this section, we present the architecture of VulAgent, shown in Figure 2. VulAgent decouples vulnerability detection into three stages: multi-view detection and integration, hypothesis construction, and validation, where validation is implemented through two consecutive steps: hypothesis-conditions validation and hypothesis-path validation.

3.1 Multi-view Vulnerability Detection

This stage prioritizes high recall over precision: we intentionally tolerate noisy candidates, since hypothesis validation will later reject those that do not hold.

We denote the dataset as $D = (c_i^{vul}, c_i^{fix}, y_i, m_i, w_i)_{i=1}^{N}$, where c_i^{vul} is a vulnerable code version, c_i^{fix} is the corresponding fixed version, y_i is the vulnerability label, m_i denotes the project name, and w_i represents the CWE category. Here, N is the total number of code pairs.

We add explicit line identifiers (e.g., `L1: int foo()`) to enable consistent cross-agent referencing during detection, validation, and reporting.

Assigning inspectors. The MetaAgent dispatches a minimal set of specialized agents based on semantic cues (e.g., memory operations, filesystem calls, concurrency primitives, cryptographic APIs, and privilege-related logic), balancing coverage and cost. For basic robustness, we always include three general agents: *StaticAnalyzerAgent*, *BehaviorAnalyzerAgent*, and *MemoryLayoutAgent*.

Specialized agents. Given a code unit c_i, the MetaAgent activates a minimal set of role-specialized analyzers A_i . A full list of analyzers and their responsibilities is provided in Appendix D.

Each agent $a \in A_i$ analyzes c_i and emits a structured finding t_i^a (JSON with line-referenced evidence). Since discovery is tuned for high recall, these raw findings are treated as candidates and pruned in later validation stages.

Report aggregation. Let $T_i = \{t_i^a \mid a \in A_i\}$ be the set of findings from all activated analyzers. We de-duplicate findings by grouping entries that share the same predicted CWE and refer to the same sink statement. Concretely, each finding provides a $(\mathrm{cwe}, \ell_{\mathrm{sink}})$ key, where ℓ_{sink} is the line number of the reported sensitive operation (sink). We merge all findings with the same key into one entry by (i) same cwe, (ii) unioning evidence snippets and line references. Findings with different keys are kept as separate entries. The resulting set $\widetilde{T}_i = \mathrm{Agg}(T_i)$ is passed to the *TriggerPlannerAgent* for hypothesis construction. Aggregation is only for de-duplication and provenance merging; all structural fields used in later stages (CWE, sink line, evidence lines) are preserved as the union of the group.

3.2 Vulnerability Hypothesis Construction

Rather than directly deciding whether a candidate is vulnerable, we cast it into a falsifiable hypothesis (conditions + path) that can be explicitly validated or refuted. A hypothesis consists of two components:

- **Hypothesis Conditions** ($\mathcal{A}$): a set of preconditions under which the vulnerability may hold (e.g., "the buffer length may exceed its allocated size", "an input pointer may be NULL"). These conditions are derived from the reported issue description and static code characteristics.
- **Hypothesis Path** ($\mathcal{P}$): a candidate control-/data-flow trigger path describing how attacker-influenced values may propagate to a sensitive sink (e.g., "user input → array index → buffer write"). The path is constructed as a symbolic trace over the code under analysis.

For each aggregated entry $\tilde{t}_i^{(k)} \in \widetilde{T}_i$ with predicted CWE type $\mathrm{cwe}_{i,k}$ and a reported sink $s_{i,k}$, the *TriggerPlannerAgent* constructs

$$h_{i,k} = \Psi(\tilde{t}_i^{(k)}, C_i) = \big(\mathrm{cwe}_{i,k},\ \mathcal{A}_{i,k},\ \mathcal{P}_{i,k}\big), \tag{1}$$

where $\mathcal{A}_{i,k}$ is the set of hypothesis conditions and $\mathcal{P}_{i,k}$ is a trigger path from a plausible attacker-controllable source to the sink $s_{i,k}$.

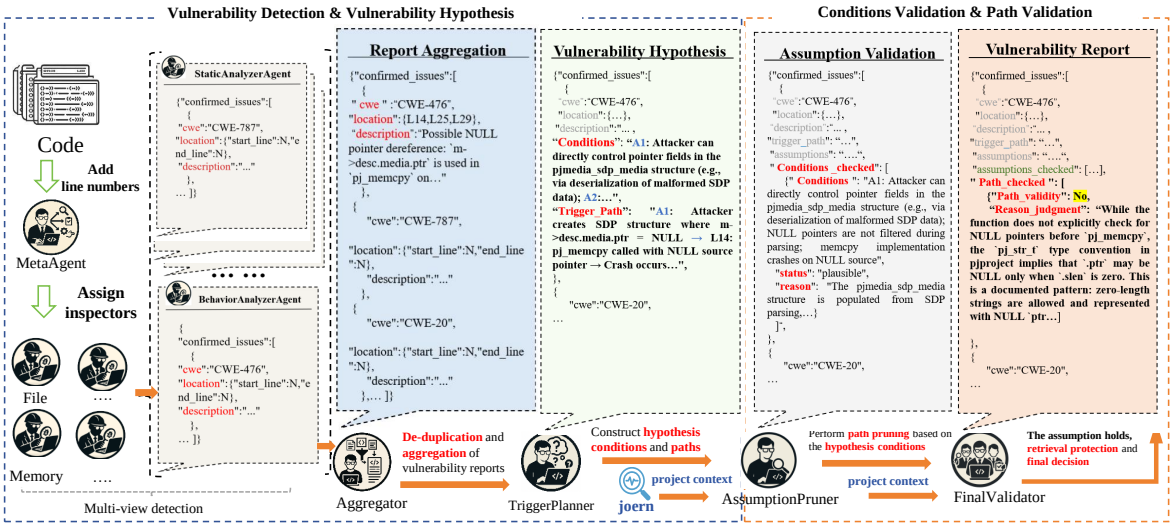


Figure 2: The framework of VulAgent, which coordinates multiple specialized agents for vulnerability detection, aggregates their reports into vulnerability hypotheses, and then performs hypothesis validation.

We model a trigger path as a labeled path over program dependence graphs:

$$\mathcal{P}_{i,k} = \langle v_1 \xrightarrow{e_1} v_2 \xrightarrow{e_2} \cdots \xrightarrow{e_{m-1}} v_m \rangle, \quad (2)$$

where $v_1 \in \mathcal{S}^{\text{att}}_{i,k}$ (e.g., parameters, file/network reads, deserialized inputs), $v_m = s_{i,k}$, and each e_j denotes a control- or data-dependence edge. All guard conditions encountered along $\mathcal{P}_{i,k}$ are recorded into $\mathcal{A}_{i,k}$ (rather than pruning the path), so they can be explicitly checked during hypothesis validation.

3.3 Hypothesis-Conditions Validation

We first challenge the hypothesis assumptions: once a required prerequisite is refuted, the hypothesis is discarded early at low cost. This phase validates the *hypothesis conditions* in $\mathcal{A}_{i,k}$ against available program context, by turning each condition into a targeted check (rather than reasoning about the full trigger path). Let C_i^{ctx} denote optional auxiliary information used to reason about vulnerability hypotheses, including retrieved evidence from the program context. For each condition $a_{i,k,j} \in \mathcal{A}_{i,k}$ in $h_{i,k} = (\text{cwe}_{i,k}, \mathcal{A}_{i,k}, \mathcal{P}_{i,k})$, we apply:

$$(\nu_{i,k,j}, \mathcal{E}_{i,k,j}) = f_{\text{va}}(a_{i,k,j}, c_i, C_i^{\text{ctx}}), \quad (3)$$

where $\nu_{i,k,j} \in \{\text{valid}, \text{plausible}, \text{contradicted}\}$ and $\mathcal{E}_{i,k,j}$ stores supporting evidence (e.g., cited lines or inferred ranges). We retain the non-contradicted conditions:

$$\widehat{\mathcal{A}}_{i,k} = \{\, a_{i,k,j} \in \mathcal{A}_{i,k} \mid \nu_{i,k,j} \neq \text{contradicted} \,\}, \quad (4)$$

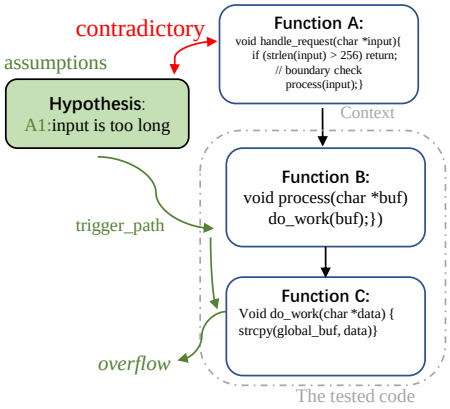


Figure 3: Overview of the hypothesis validation process.

and defer trigger-path reasoning to the next phase.

Figure 3 shows a context-first refutation example. Given hypothesis A1 ("input is too long"), the validator inspects surrounding callers of the sink-containing function and finds an upstream length check (e.g., `if (strlen(input) > 256) return;`) before the value reaches the sink. This contradicts A1, so the hypothesis is rejected.

3.4 Hypothesis-Path validation

We do not aim to prove all paths; we only verify whether the hypothesized trigger path is feasible and self-consistent. This phase checks whether a validated hypothesis remains triggerable in practice by verifying whether defensive guards on the hypothesized trigger path neutralize the sink (e.g., bounds/null checks, early returns, or structured error handling).

Given $h_{i,k} = (\text{cwe}_{i,k}, \widehat{\mathcal{A}}_{i,k}, \mathcal{P}_{i,k})$, the line-numbered function c_i, and optional context C_i^{ctx}, we validate the hypothesis at the granularity of its

constructed path:

$$r_{i,k} = f_{\text{pc}}(\mathcal{P}_{i,k}, c_i, C_i^{\text{ctx}}; \widehat{\mathcal{A}}_{i,k}), \quad (5)$$

where $r_{i,k} \in \{\text{retained}, \text{discarded}\}$. We discard a hypothesis iff the sink is effectively guarded along $\mathcal{P}_{i,k}$ under $\widehat{\mathcal{A}}_{i,k}$, i.e., an effective defense occurs before the sink and blocks the hypothesized execution to the sink; otherwise, the hypothesis is retained.

The retained hypotheses across all candidates are then consolidated into the final report for c_i, including the predicted CWE type, the localized sink/line spans, the validated conditions, and the supporting evidence (context citations) used during validation. If no hypothesis is retained, VulAgent outputs no vulnerability for c_i with the pruned candidates as traceable evidence.

4 Experimental Setup

4.1 Datasets

PrimeVul. PrimeVul is a real-world C/C++ vulnerability dataset spanning 755 projects and 140 CWE categories, with 6,968 vulnerable samples from 6,827 commits (Ding et al., 2025). We follow the same split and evaluation protocol as prior work (Widyasari et al., 2025), where the test set contains 435 vulnerable–fixed code pairs.

SVEN. SVEN is a hand-checked, function-level corpus of security-fixing commits, constructed by filtering and pairing samples from CrossVul, BigVul, and VUDENC to emphasize data quality (He and Vechev, 2023). It covers nine CWEs with 803 vulnerable–fixed pairs (1,606 programs) across C/C++ and Python, using an 8:1:1 train/validation/test split (He and Vechev, 2023).

4.2 Experimental Details

To ensure reproducibility, all large language models used in this work were accessed via commercial APIs, with temperature set to 0 and top_p set to 1. All datasets follow publicly available splits. Prompt construction does not include any few-shot code examples.

Cost–quality trade-off. Multi-agent pipelines incur additional inference cost. Under the same backbone and context setting, GPTLens and VulTrial generate 2263.95 and 3119.16 output tokens per sample, respectively. To reduce overhead, VulAgent enforces a strict JSON-only output format and discourages verbose free-form intermediate reasoning. Based on sampled runs, VulAgent issues 8.83 LLM calls per sample and generates 7,036.95 output tokens per sample. We further implement an 8-way multi-threaded execution to reduce wall-clock latency.

4.3 Performance Metrics

Usability Evaluation. We report three standard classification metrics: Accuracy (ACC); F1-score; and False Positive Rate (FPR). While F1 is widely used, in vulnerability detection it can be misleading when false positives are prevalent (Steenhoek et al., 2024a). Therefore, we explicitly report FPR alongside ACC and F1 to better reflect practical usability, since high accuracy or F1 with a high FPR is often unsuitable for real-world deployment.

Understanding Evaluation. To assess whether a model can distinguish vulnerable code from its fixed counterpart, we adopt pair-wise metrics (Ding et al., 2025).

Pair-wise Correct Prediction (P-C): the percentage of function pairs where both functions are correctly classified (vulnerable as vulnerable, fixed as benign).

Pair-wise Reversed Prediction (P-R): the percentage of pairs where the two labels are predicted inversely (vulnerable predicted as benign while the fixed version is predicted as vulnerable).

Vulnerability Pair-wise Score (VPS): following (Wen et al., 2025), we compute $\text{VPS} = \text{P-C} - \text{P-R}$, which measures the net gain of correct over reversed pair-wise predictions.

Baselines. For the single-agent baseline, we use chain-of-thought (CoT) prompting (Ding et al., 2025), a standard step-by-step reasoning strategy with strong performance on PrimeVul (Ding et al., 2025). For multi-agent baselines, we include three representative systems: GPTLens (Hu et al., 2023) (auditor–critic with critic-based scoring), VulTrial (Widyasari et al., 2025) (four roles with a review board for final judgment), and LLM-MultiRole (Mao et al., 2024) (developer/tester-style collaborative review). We additionally include Semgrep as a practical static analysis baseline (default C/C++ rules), which helps contextualize LLM-based methods in terms of false positives under the same evaluation protocol.

Models. We evaluate on Qwen3-235B-A22-Instruct (Qwen Team, 2025), GPT-4o (Achiam et al., 2023), and DeepSeek v3.1 (DeepSeek-AI, 2024) to test cross-model generality. We also report GPT-3.5 results from VulTrial (Widyasari et al., 2025) and include our fine-tuned UniX-

Table 1: Comparison of vulnerability detection performance across two datasets.

Model	Agents	Method	PrimeVul						SVEN					
			P-C↑	P-R↓	VPS↑	FPR↓	F1↑	ACC↑	P-C↑	P-R↓	VPS↑	FPR↓	F1↑	ACC↑
Semgrep	-	Static	0.00	0.08	-0.08	64.0	50.9	46.0	0.00	0.00	0.00	31.3	38.5	50.0
UniXcoder	-	SFT	7.62	0.92	6.69	22.9	54.0	53.4	13.2	0.00	13.2	1.20	25.0	56.6
GPT-3.5	Single-Agent	CoT	6.21	5.50	0.71	10.3	18.1	50.3	1.20	0.00	1.20	95.3	66.1	51.8
	Multi-Agent	GPTLens	10.1	6.44	-0.91	94.7	65.0	49.6	6.00	7.60	-1.2	19.3	26.3	49.4
	Multi-Agent	VulTrial	18.6	11.4	1.38	23.4	33.4	50.7	1.20	1.20	0.00	97.6	65.8	50.0
GPT-4o	Single-Agent	CoT	9.20	6.67	2.50	30.8	28.1	51.7	1.20	2.40	-1.20	97.5	65.5	49.4
	Multi-Agent	LLM-MultiRole	1.61	6.44	-5.05	17.3	65.7	47.3	3.61	12.0	-8.43	15.6	11.7	45.7
	Multi-Agent	GPTLens	10.1	6.76	3.60	61.9	18.8	50.2	3.61	3.61	0.00	92.8	64.9	50.2
	Multi-Agent	VulTrial	18.6	11.4	7.13	52.6	56.1	53.4	4.82	1.20	3.60	95.2	67.2	52.0
	Multi-Agent	**VulAgent**	**26.6**	10.1	**16.5**	**36.7**	56.2	**58.6**	**26.5**	6.02	**20.4**	55.4	65.6	**60.2**

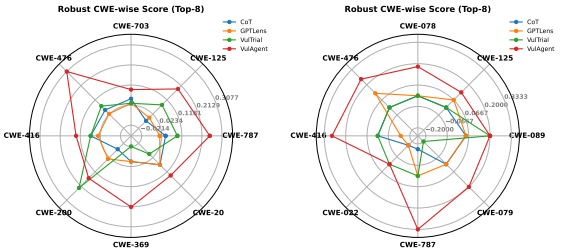


Figure 4: Performance of different methods across the Top-8 CWE categories on PrimeVul (left) and SVEN (right). The score is defined as $(\#\text{correct} - \#\text{wrong})/\#\text{total}$ for each CWE.

coder (Guo et al., 2022) baseline (a strong code PLM on CodeXGLUE (Lu et al., 2021)). For all context-dependent methods, we provide identical retrieved context to ensure fair comparison.

5 Experimental Results

We aim to answer the following research questions:

RQ1: How effective is VulAgent compared with state-of-the-art?

RQ2: How well does VulAgent generalize across different LLMs?

RQ3: What impact would the lack of relevant context have on the performance of VulAgent?

RQ4: How does VulAgent perform across different types of vulnerabilities (CWE categories)?

RQ5: How effective is our proposed hypothesis validation process?

5.1 RQ1. Effectiveness of VulAgent

For fairness, all baseline results are reproduced following their original settings, while VulAgent is evaluated using the proposed multi-agent pipeline with hypothesis validation. The experiment was conducted three times, and the average value was taken as the final result.

On PrimeVul, VulAgent improves P-C to 26.6% (vs. VulTrial 18.6%, +43.0%), while reducing FPR from 52.6% to 36.7% (-30.0% rel.) and increasing ACC from 53.4% to 58.6% (+5.2 pp). It also achieves the best VPS, indicating a stronger ability to correctly distinguish vulnerable–fixed pairs. For the remaining pair-wise metric P-R and the aggregate metric F1, the results should be interpreted together with FPR: in vulnerability detection, aggressive predictors can obtain competitive F1 or seemingly favorable pair-wise behavior by over-predicting vulnerabilities, often at the cost of many false positives. On SVEN, where baselines degrade sharply, VulAgent remains robust with P-C 26.5% (vs. 4.82%, +449.8%), lowering FPR from 95.2% to 55.4% (-41.8% rel.) and raising ACC from 52.0% to 60.2% (+8.2 pp). It likewise delivers the strongest VPS on SVEN, showing that the gain is not limited to instance-level classification but also extends to vulnerable–fixed pair identification under harder distribution shifts. These gains indicate that hypothesis validation leverages context to prune speculative false positives, improving both usability and accuracy.

Figure 4 further shows robust CWE-wise scores, defined as $(\#\text{correct} - \#\text{wrong})/\#\text{total} \in [-1, 1]$, where negative values indicate unstable performance. VulAgent achieves higher and more balanced scores across the Top-8 CWEs on both datasets; on SVEN, it remains positive across all categories where baselines often fail (e.g., CWE-089/078). Overall, reflection-based baselines tend to over-approximate risks (FPR $>$ 90% on SVEN), whereas VulAgent's two-stage validation filters such paths, reducing manual review cost. This also explains why metrics such as F1 or P-R should not be read in isolation: under highly imbalanced and pair-structured settings, a detector that labels both versions conservatively or aggressively may appear competitive on one metric while failing to reliably separate vulnerable code from its fixed counterpart.

Dataset	Method	P-C ↑	P-R ↓	VPS ↑	FPR ↓	F1 ↑	ACC ↑
PrimeVul	VulAgent	**26.61**	**10.10**	**16.50**	**36.70**	56.20	**58.60**
	w/o F	15.56	11.94	3.62	65.00	**58.74**	51.81
	w/o P	15.63	10.34	5.29	61.15	58.38	52.64
SVEN	VulAgent	**26.50**	6.02	**20.40**	**55.40**	**65.60**	**60.20**
	w/o F	12.07	7.23	4.48	72.29	61.48	52.41
	w/o P	18.07	**4.82**	13.25	62.65	63.64	56.62

Table 2: Ablation study of the two key filtering components in VulAgent. F denotes **FinalValidatorAgent**, and P denotes **PathPrunerAgent**. Removing either component consistently reduces overall reliability, especially by increasing false positives and lowering ACC/VPS.

Component Ablation. We study the effect of the two key filtering components in VulAgent: **FinalValidatorAgent** (F) and **PathPrunerAgent** (P). As shown in Table 2, removing either component consistently degrades the full system on both datasets. On PrimeVul, ACC drops from 58.60% to 51.81% (w/o F) and 52.64% (w/o P), while FPR rises from 36.70% to 65.00% and 61.15%, respectively. On SVEN, the full model still achieves the best ACC (60.20%), VPS (20.40%), and FPR (55.40%), whereas both ablated variants become noticeably less reliable. This confirms that VulAgent's gain comes from the collaboration of staged filtering rather than from any single component in isolation.

5.2 RQ2: Generalization Ability of VulAgent

To evaluate the generalization capacity of our proposed VulAgent framework, we assess its vulnerability prediction performance across two datasets: PrimeVul and SVEN. Table 3 presents the results.

Across both datasets, GPT-4o achieves the highest overall accuracy, with 58.62% on PrimeVul and 60.20% on SVEN, demonstrating the strongest generalization ability. Notably, it maintains a high pair score while balancing false positives reasonably well.

In contrast, DeepSeek v3.1 consistently reports the lowest false positive rates—19.95% on PrimeVul and 31.30% on SVEN—indicating a conservative detection style. Although its P-C and VPS are lower than GPT-4o and Qwen3-235B, this restraint makes it suitable for high-precision scenarios where over-reporting is costly.

These results confirm that VulAgent generalizes robustly across different LLM backbones: it consistently improves accuracy and reduces false positives regardless of whether the underlying model prioritizes precision (DeepSeek), balance (Qwen3), or overall accuracy (GPT-4o).

5.3 RQ3. Effects of Code Context for VulAgent

To quantify the effect of code context, we run VulAgent with and without context on both datasets (Table 4). We detail the context sources and construction procedures for each dataset in Appendix E.

PrimeVul. Following prior work, we use the dataset-provided context when available (304/435 test pairs) (Zhu et al., 2025). Adding context consistently improves performance: P-C rises from 17.5 to 28.7 (+64.1% rel.), VPS from 10.0 to 18.0, and ACC from 55.0% to 59.0%.

SVEN. Since SVEN lacks annotated context, we retrieve lightweight project-level information via static analysis (e.g., related calls/imports) and feed it as context. Context yields similar gains: P-C increases from 20.4 to 26.5 (+29.4% rel.), VPS from 13.2 to 20.4, and ACC from 56.6% to 60.2%.

VulAgent is robust to context availability. On PrimeVul, using the code context consistently improves P-C/VPS/ACC; on SVEN, where no annotated context is provided, lightweight project-level context retrieved via static analysis (e.g., related calls/imports) yields similar gains. Importantly, VulAgent still performs reasonably in the context-free setting on both datasets, implying that multi-round validation mitigates hallucinated vulnerability claims even without additional context.

5.4 RQ4. Effectiveness Across Different CWE Categories

Table 5 summarizes VulAgent's performance across CWE categories on PrimeVul and SVEN. Overall, we observe a consistent pattern driven by two factors: (i) how easy it is to hypothesize a candidate issue from local syntax, and (ii) how strongly validation depends on non-local context.

(1) Easy-to-hypothesize, weak context dependency. Categories such as CWE-476 and CWE-190 are easy to surface and can often be validated with local guards (e.g., null/range checks), leading to strong effectiveness, though false positives may still arise when protections rely on global invariants.

(2) Easy-to-hypothesize, strong context dependency. For CWE-089/125/022, syntactic cues make candidates easy to flag, but exploitability often hinges on non-local constraints (e.g., sanitizers, upstream bounds, framework contracts). As a result, recall tends to be high while FPR remains

Table 3: Comparison of model generalization performance across two datasets.

Model	PrimeVul						SVEN					
	P-C↑	P-R↓	VPS↑	FPR↓	F1↑	ACC↑	P-C↑	P-R↓	VPS↑	FPR↓	F1↑	ACC↑
DeepSeek v3.1	17.7	**8.74**	8.96	**19.9**	41.5	54.7	30.1	9.64	20.4	31.3	56.5	60.2
Qwen3-235B-A22-Instruct	25.7	11.0	14.7	40.1	**57.7**	57.3	**30.1**	10.8	19.2	**28.9**	54.4	59.6
GPT-4o	**26.6**	10.1	**16.5**	36.7	56.1	**58.6**	26.5	**6.02**	**20.4**	55.4	**65.6**	**60.2**

Table 4: Effect of code context on VulAgent (ChatGPT-4o backbone).

Dataset / Setting	P-C	P-R	VPS	FPR	F1	ACC
PrimeVul (w/o context)	17.5	7.5	10.0	38.7	52.0	55.0
PrimeVul (w/ context)	**28.7**	**10.7**	**18.0**	**36.3**	**57.0**	**59.0**
SVEN (w/o context)	20.4	7.20	13.2	60.0	63.2	56.6
SVEN (w/ context)	**26.5**	**6.02**	**20.4**	**55.4**	**65.6**	**60.2**

elevated under limited context.

(3) Hard-to-hypothesize, weak context dependency. For semantic categories such as CWE-703, the main bottleneck is hypothesis generation (non-obvious triggers), yielding lower recall even though local validation may be feasible once a hypothesis is formed.

(4) Hard-to-hypothesize, strong context dependency. Categories such as CWE-078 and CWE-200 often require rich cross-function semantics for both proposing and validating hypotheses; incomplete context therefore leads to both misses and false alarms.

Combined with the findings of RQ3, this suggests that constructing richer and more accurate context is essential for further improving validation performance on context-dependent vulnerabilities.

5.5 RQ5. Effectiveness of Hypothesis Validation Process

Ablation setup (oracle-CWE). Following Li et al. (Li et al., 2025), we evaluate on SVEN under an oracle-CWE setting. For each function, we query every method with the same ground-truth CWE type (i.e., "Does this function contain CWE- k?") and provide the same retrieved code context. Therefore, performance differences primarily reflect the quality of each method's validation step rather than CWE identification. We compare (i) **Direct** one-shot CoT judgment, (ii) **VulTrial** reflection/debate with the oracle label as input, and (iii) **VulAgent** running our validation chain with the oracle label substituted as the candidate report.

Table 6 shows that removing detection noise improves all methods, but baselines still over-predict vulnerabilities. Direct achieves the best P-C (36.14) and F1 (65.95) but has high FPR (49.40%). Reflection reduces FPR to 39.76% (ACC 59.04) yet remains false-positive heavy. VulAgent yields the lowest FPR (20.48%) while maintaining comparable accuracy to Direct (61.45% vs. 62.05) and outperforming Reflection (+2.50 ACC points), indicating that targeted hypothesis validation is necessary to curb false alarms even with oracle labels.

We target actionability rather than completeness: in practice, reviewers need to quickly discard hypotheses that cannot be triggered under realistic guards; our validation stage automates this triage by refuting non-triggerable hypotheses with contextual evidence.

6 Case Study

We present a case study to illustrate how VulAgent reduces false positives. We analyze `search_make_new` from `libevent` (commit `ec65c42`, CWE-125), whose ground truth is **non-vulnerable**. The full code is provided in Appendix A.

VulTrial flags multiple issues (e.g., overflow/NULL-deref/DoS) by assuming attacker control over values such as `postfix_len` or reachability of `EVUTIL_ASSERT(0)`. These reports are false positives because the buffer allocation accounts for `postfix_len`, `mm_malloc` is explicitly checked, the assertion is unreachable in normal execution, and key fields (e.g., `state->head`, `dom->len`) are internal and not attacker-controlled.

VulAgent may surface similar initial risks, but rejects them during hypothesis validation by (i) checking allocation-size consistency with subsequent writes, (ii) confirming existing guards (e.g., early return on NULL), and (iii) refuting attacker controllability and reachability assumptions using structural invariants. It therefore outputs **no vulnerability**, consistent with the ground truth.

Table 5: Performance of VulAgent across different CWE categories on PrimeVul and SVEN.

Dataset	CWE	Prec	Rec	F1	Acc	FPR
PrimeVul	CWE-476	0.688	0.564	0.620	0.654	0.256
	CWE-787	0.661	0.514	0.578	0.625	0.264
	CWE-617	0.667	0.500	0.571	0.625	0.250
	CWE-369	0.714	0.357	0.476	0.607	0.143
	CWE-125	0.610	0.532	0.568	0.596	0.340
SVEN	CWE-190	0.667	0.800	0.727	0.700	0.400
	CWE-787	0.667	0.667	0.667	0.667	0.333
	CWE-416	0.667	0.571	0.615	0.643	0.286
	CWE-476	0.583	0.875	0.700	0.625	0.625
	CWE-089	0.556	1.000	0.714	0.600	0.800

Table 6: Effectiveness of our hypothesis validation process compared with baselines.

Method	P-C↑	P-R↓	VPS↑	FPR↓	F1↑	ACC↑
Direct Evaluation	36.14	12.05	24.10	49.40	65.95	62.05
VulTrial (Reflection)	25.30	7.23	18.07	39.76	58.54	59.04
VulAgent (Ours)	24.10	**1.20**	22.98	**20.48**	52.94	61.45

7 Conclusion

In the context of LLM-based vulnerability detection systems, VulAgent helps bridge the gap between explanation and validation. Rather than treating natural-language rationales as endpoints, we operationalize them into testable hypotheses—actionable checks that can be corroborated or refuted using program-analysis evidence. This shift encourages a validation-oriented workflow that improves faithfulness and grounding in our setting, while mitigating hallucination-like behaviors arising from unvalidated assumptions. We believe the same "hypothesis-to-check" principle may also be useful in other domains that require accountable reasoning, such as code review, scientific claim checking, and tool-augmented question answering.

8 Acknowledgments

This research are supported by the National Key R&D Program under Grant No. 2022YFB4501902 and No. 2023YFB4503801, the National Natural Science Foundation of China under Grant No. 62192733, 62192730, and the Major Program (JD) of Hubei Province (No.2023BAA024).

9 Limitations

Context availability and project specificity. VulAgent validates hypotheses using retrieved program context and the model's general knowledge of widely used APIs and security patterns (e.g., common guard idioms and API-level safety contracts), treating the latter as a prior rather than decisive evidence. This reduces dependence on local context, but does not eliminate the need for relevant, project-specific evidence. In projects with highly customized internal frameworks, private APIs, or undocumented conventions, such generic knowledge may not transfer, making validation brittle. Moreover, obtaining accurate inter-procedural and project-level context remains challenging: static retrieval can be incomplete or noisy, which may cause the validator to reject otherwise real issues due to missing supporting evidence.

Coverage and hypothesis completeness. While VulAgent uses multi-view detection to improve candidate discovery, it does not guarantee exhaustive vulnerability coverage. Hypothesis construction still depends on the underlying model's code understanding and exploration strategy; deeply inter-procedural bugs and cases with branching paths may be missed, and such omissions cannot be recovered by downstream validation. Improving coverage via stronger exploration and hypothesis generation is an important direction for future work.

Cost and latency. Multi-agent validation introduces additional inference overhead. We mitigate this by defaulting to non-reasoning backbones and enforcing compact JSON-only outputs, but the output budget remains higher than prior multi-agent baselines. Under the same backbone and context setting, VulAgent generates 7,036.95 output tokens per sample, compared to 3,119.16 for VulTrial and 2,263.95 for GPTLens (about $2.3\times$ and $3.1\times$, respectively). While this is often acceptable for offline security review where false-positive triage dominates human time, it remains a tangible trade-off for latency-sensitive or large-scale scanning.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, and et al. (OpenAI). 2023. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*.

Liqing Cao, Haofeng Li, Chenghang Shi, Jie Lu, Haining Meng, Lian Li, and Jingling Xue. 2025. Vulpa: Detecting semantically recurring vulnerabilities with multi-object typestate analysis. *Proceedings of the ACM on Software Engineering*, 2(FSE):2430–2453.

Checkmarx. 2022. Online. *Available: https://www.checkmarx.com/*.

Xiao Cheng, Guanqin Zhang, Haoyu Wang, and Yulei Sui. 2022. Path-sensitive code embedding via contrastive learning for software vulnerability detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 519–531.

Hoa Khanh Dam, Truyen Tran, Trang Pham, Shien Wee Ng, John Grundy, and Aditya Ghose. 2017. Automatic feature learning for vulnerability prediction. *arXiv preprint arXiv:1708.02368*.

DeepSeek-AI. 2024. DeepSeek-V3 technical report. *arXiv preprint arXiv:2412.19437*.

Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. 2025. Vulnerability detection with code language models: How far are we? In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 1729–1741. IEEE.

Flawfinder. 2022. Online. *Available: http://www.dwheeler.com*.

Michael Fu and Chakkrit Tantithamthavorn. 2022. Linevul: A transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 608–620.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7212–7225.

Jingxuan He and Martin Vechev. 2023. Large language models for code: Security hardening and adversarial testing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 1865–1879.

Junda He, Christoph Treude, and David Lo. 2025. LLM-based multi-agent systems for software engineering: Literature review, vision, and the road ahead. *ACM Transactions on Software Engineering and Methodology*, 34(5):1–30.

Sihao Hu, Tiansheng Huang, Fatih İlhan, Selim Furkan Tekin, and Ling Liu. 2023. Large language model-powered smart contract vulnerability detection: New perspectives. In *2023 5th IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*, pages 297–306. IEEE.

Julian Jang-Jaccard and Surya Nepal. 2014. A survey of emerging threats in cybersecurity. *Journal of computer and system sciences*, 80(5):973–993.

Arnold Johnson, Kelley Dempsey, Ron Ross, Sarbari Gupta, Dennis Bailey, et al. 2011. Guide for security-focused configuration management of information systems. *NIST special publication*, 800(128):16–16.

Yue Li, Xiao Li, Hao Wu, Minghui Xu, Yue Zhang, Xiuzhen Cheng, Fengyuan Xu, and Sheng Zhong. 2025. Everything you wanted to know about LLM-based vulnerability detection but were afraid to ask. *arXiv preprint arXiv:2504.13474*.

Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2021. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 19(4):2244–2258.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*.

Wei Ma, Daoyuan Wu, Yuqiang Sun, Tianwen Wang, Shangqing Liu, Jian Zhang, Yue Xue, and Yang Liu. 2024. Combining fine-tuning and LLM-based agents for intuitive smart contract auditing with justifications. *arXiv preprint arXiv:2403.16073*.

Zhenyy Mao, Jialong Li, Dongming Jin, Munan Li, and Kenji Tei. 2024. Multi-role consensus through LLMs discussions for vulnerability detection. In *2024 IEEE 24th International Conference on Software Quality, Reliability, and Security Companion (QRS-C)*, pages 1318–1319. IEEE.

Yu Nong, Mohammed Aldeen, Long Cheng, Hongxin Hu, Feng Chen, and Haipeng Cai. 2024. Chain-of-thought prompting of large language models for discovering and fixing software vulnerabilities. *arXiv preprint arXiv:2402.17230*.

Qwen Team. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. 2018. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE international conference on machine learning and applications (ICMLA)*, pages 757–762. IEEE.

Jingwei Shi, Xinxiang Yin, Jing Huang, Jinman Zhao, and Shengyu Tao. 2026. Codehacker: Automated test case generation for detecting vulnerabilities in competitive programming solutions. *arXiv preprint arXiv:2602.20213*.

Benjamin Steenhoek, Hongyang Gao, and Wei Le. 2024a. Dataflow analysis-inspired deep learning for efficient vulnerability detection. In *Proceedings of the 46th ieee/acm international conference on software engineering*, pages 1–13.

Benjamin Steenhoek, Md Mahbubur Rahman, Monoshi Kumar Roy, Mirza Sanjida Alam, Earl T Barr, and Wei Le. 2024b. A comprehensive study of the capabilities of large language models for vulnerability detection. *arXiv preprint arXiv:2403.17218*, 59.

Karl Tamberg and Hayretdin Bahsi. 2025. Harnessing large language models for software vulnerability detection: A comprehensive benchmarking study. *IEEE Access*, 13:29698–29717.

Chandra Thapa, Seung Ick Jang, Muhammad Ejaz Ahmed, Seyit Camtepe, Josef Pieprzyk, and Surya Nepal. 2022. Transformer-based language models for software vulnerability detection. In *Proceedings of the 38th Annual Computer Security Applications Conference*, pages 481–496.

Saad Ullah, Mingji Han, Saurabh Pujar, Hammond Pearce, Ayse Coskun, and Gianluca Stringhini. 2024. LLMs cannot reliably identify and reason about security vulnerabilities (yet?): A comprehensive evaluation, framework, and benchmarks. In *2024 IEEE symposium on security and privacy (SP)*, pages 862–880. IEEE.

Zhiyuan Wei, Jing Sun, Zijiang Zhang, Xianhao Zhang, Meng Li, and Zhe Hou. 2024. LLM-smartaudit: Advanced smart contract vulnerability detection. *arXiv preprint arXiv:2410.09381*.

Xin-Cheng Wen, Yijun Yang, Cuiyun Gao, Yang Xiao, and Deheng Ye. 2025. Boosting vulnerability detection of LLMs via curriculum preference optimization with synthetic reasoning data. *arXiv preprint arXiv:2506.07390*.

Xin-Cheng Wen, Jiaxin Ye, Cuiyun Gao, Lianwei Wu, and Qing Liao. 2024. Evalsva: Multi-agent evaluators for next-gen software vulnerability assessment. *arXiv preprint arXiv:2501.14737*.

Ratnadira Widyasari, Martin Weyssow, Ivana Clairine Irsan, Han Wei Ang, Frank Liauw, Eng Lieh Ouh, Lwin Khin Shar, Hong Jin Kang, and David Lo. 2025. Let the trial begin: A mock-court approach to vulnerability detection using LLM-based agents. *arXiv preprint arXiv:2505.10961*.

Fabian Yamaguchi. 2015. *Pattern-Based Vulnerability Discovery.* Ph.D. thesis, University of Göttingen.

Fabian Yamaguchi. 2017. Pattern-based methods for vulnerability discovery. *it-Information Technology*, 59(2):101–106.

Chenyuan Zhang, Hao Liu, Jiutian Zeng, Kejing Yang, Yuhong Li, and Hui Li. 2024. Prompt-enhanced software vulnerability detection using chatgpt. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, pages 276–277.

Xin Zhou, Ting Zhang, and David Lo. 2024. Large language model for vulnerability detection: Emerging results and future directions. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, pages 47–51.

Hao Zhu, Jia Li, Cuiyun Gao, Jiaru Qian, Yihong Dong, Huanyu Liu, Lecheng Wang, Ziliang Wang, Xiaolong Hu, and Ge Li. 2025. Specification-guided vulnerability detection with large language models. *arXiv preprint arXiv:2511.04014*.

A Case Study Details

We present a representative false-positive case from libevent (evdns.c, search_make_new; commit ec65c42), whose ground-truth label is *benign* (no vulnerability). The function builds a new domain name by allocating a buffer and copying the base name and a postfix derived from an internal `search_domain` object. LLM-based baselines tend to over-assume attacker control over dom->len or miss the allocation/NULL checks, producing spurious CWE-125/overflow alarms. In contrast, VulAgent rejects these reports by validating (i) attacker controllability assumptions and (ii) the consistency between allocation size and subsequent memory writes along the hypothesized path.

Listing 1: Function under analysis: search_make_new in evdns.c

```
search_make_new(const struct search_state *const
    state,
                int n, const char *const base_name)
                    {
    const size_t base_len = strlen(base_name);
    char need_to_append_dot;
    struct search_domain *dom;

    if (!base_len) return NULL;
    need_to_append_dot = base_name[base_len - 1] ==
        '.' ? 0 : 1;

    for (dom = state->head; dom; dom = dom->next) {
        if (!n--) {
            const u8 *const postfix = ((u8 *) dom) +
                 \sizeof(struct search_domain);
            const int postfix_len = dom->len;
            char *const newname = (char *) mm_malloc
                (
                base_len + need_to_append_dot +
                     postfix_len + 1);
            if (!newname) return NULL;
            memcpy(newname, base_name, base_len);
            if (need_to_append_dot) newname[base_len
                ] = '.';
```

```
            memcpy(newname + base_len +
                 need_to_append_dot, postfix,
                 postfix_len);
            newname[base_len + need_to_append_dot +
                 postfix_len] = 0;
            return newname;
        }
    }

    EVUTIL_ASSERT(0);
    return NULL;
}
```

We analyze a representative example in the search_make_new function from evdns.c, comparing the baseline single-agent system (VulTrial) with our multi-agent system (VulAgent).

Baseline Result (VulTrial). VulTrial reported five distinct vulnerabilities in this function:

- **Heap buffer overflow**: It flagged a potential overflow in the line `memcpy(newname + base_len + need_to_append_dot, postfix, postfix_len)`, reasoning that the buffer size might be insufficient if `postfix_len` is attacker-controlled.
- **Integer overflow**: It assumed that the allocation size expression `base_len + need_to_append_dot + postfix_len + 1` could wrap around and allocate a buffer smaller than required.
- **Null dereference**: It flagged possible use of a NULL pointer if `mm_malloc` failed to allocate memory for `newname`.
- **Loop-bound failure**: It warned that a negative `n` value might cause unintended loop behavior.
- **Denial-of-service via assertion**: It treated `EVUTIL_ASSERT(0)` as attacker-triggerable, potentially causing program termination.

However, these findings were based on over-approximated assumptions. The reviewer confirmed all five were false positives:

- The allocation size is computed with all necessary components, including `postfix_len`, ensuring the buffer is large enough for the write.
- The return value of `mm_malloc` is explicitly null-checked before use.
- The assertion is unreachable under normal conditions and serves as a developer safeguard.
- The loop variable `n` is bounded, and the internal list traversal logic ensures correctness.
- Crucially, `state->head` and `dom->len` originate from internal, trusted allocations and cannot be attacker-controlled under any feasible path.

These examples illustrate that VulTrial lacks the ability to distinguish between plausible risks and actually exploitable paths.

Our Method (VulAgent). VulAgent analyzed the same function using a pipeline of specialized agents (including memory layout, error handling, and symbolic execution). While initial agents surfaced similar risks (e.g., out-of-bounds writes, integer overflows), the FinalValidatorAgent reviewed each trigger path by checking attacker controllability and structural hypotheses. For instance:

- It traced the allocation and confirmed that `postfix_len` is fully included in the buffer size, ensuring safe `memcpy`.
- It validated that the null dereference risk is neutralized by an explicit `if (!newname) return NULL;` check.
- It rejected the assumption that internal pointers (like `state->head`) are attacker-controlled, citing structural invariants.
- It classified the `EVUTIL_ASSERT(0)` path as unreachable under attacker influence, confirming it as non-exploitable.

All paths were ultimately deemed non-exploitable, and the final output matched the ground-truth label: **no vulnerability**.

This case reveals a fundamental flaw in LLM vulnerability detectors: they often confuse theoretical flaws with the actual attack surface, thereby resulting in a high rate of false positives. The current method cannot enable large language models to exclude those attacks that cannot be triggered based on the specific context in engineering projects. By contrast, VulAgent's layered reasoning—from symbolic path exploration to assumption validation—enables it to filter out spurious reports effectively. This not only improves precision but also significantly reduces the manual burden on security reviewers.

Residual False Positive Case (VulAgent). We next present a representative *residual false-positive* case from `cairo` (`_inplace_src_spans`; commit `03a820b`), whose ground-truth label is *benign* (non-vulnerable / fixed version). Although the historical bug in this area is associated with CVE-2020-35492, the analyzed sample in our benchmark is the post-fix version. The function iterates over

a span array and either composites full-coverage runs directly or fills a mask buffer obtained from `pixman_image_get_data(r->mask)`. Because it contains local pointer arithmetic over `spans`, raw writes through `m`, and size-sensitive operations such as `memset`, it appears suspicious under local reasoning.

VulAgent Result. VulAgent still reported multiple memory-safety issues on this benign sample, including several **CWE-787** alarms, two **CWE-125** alarms, and one **CWE-476** alarm. The retained reports mainly followed three trigger patterns: (i) accessing `spans[1]` without a local check that `num_spans >= 2`; (ii) writing through `m` using `len = spans[1].x - spans[0].x` and `memset(m, ..., len)` without a local proof that the underlying mask buffer is large enough; and (iii) dereferencing `base` returned by `pixman_image_get_data(r->mask)` without an explicit NULL check.

Why This Is a False Positive. The issue is not that VulAgent missed an obvious local guard, but that the final decision was made under *missing evidence*. In the validation stage, several critical assumptions were left as *plausible* or *unknown* rather than being refuted. For example, the system treated the following assumptions as still possible: (1) the `spans` array may be attacker-controlled and may violate the implicit caller-side contract; (2) `r->mask` or the buffer returned by `pixman_image_get_data()` may be undersized; and (3) these internal rendering objects may reach the function in malformed states without being rejected earlier. However, these are precisely the kinds of facts that depend on framework-level invariants, renderer lifecycle guarantees, and caller-side construction rules, none of which are explicit in the local function body.

As a result, VulAgent over-relied on the absence of local checks and conservatively preserved speculative paths as real vulnerabilities. In other words, the system correctly identified that the function contains *memory-sensitive operations*, but it lacked enough contextual evidence to determine whether the dangerous states were actually reachable in the engineering workflow of Cairo.

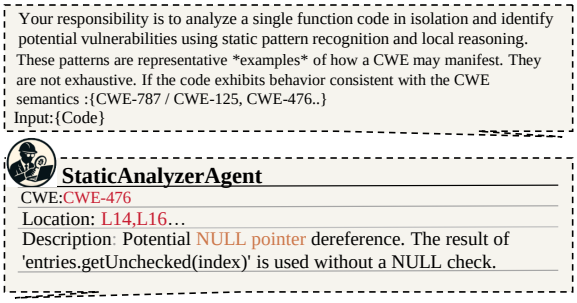
Takeaway. This case highlights an important limitation of our current validation pipeline. When exploitability depends on framework invariants or caller-established object contracts that are not visible in the provided context, VulAgent may fail to refute key attacker-controllability assumptions and thus retain false positives. The example is therefore not a failure of syntactic detection, but a failure of *evidence sufficiency*: local code patterns suggest risk, while the missing global invariants prevent the validator from confidently ruling that risk out.

B Method Process Illustration

This appendix provides implementation-level flowcharts for the core components of VulAgent. Together, they form a complete pipeline from multi-view discovery to hypothesis construction and validation.

B.1 End-to-End Flow (Step-by-step)

Step 1: MetaAgent dispatch and multi-view discovery. Given a line-numbered function, the MetaAgent activates a minimal set of role-specialized analyzers based on semantic cues (with a small always-on core for coverage). Each activated agent emits candidate findings with line-referenced evidence in a unified JSON schema, which makes downstream consolidation and validation straightforward.



Step 2: Aggregation and deduplication. The AggregatorAgent merges overlapping findings from different agents into a de-duplicated candidate set. It groups reports that refer to the same CWE and overlapping spans, merges evidence and provenance (source agents), and preserves distinct candidates for later hypothesis construction.

Step 3: Hypothesis construction (conditions + trigger path). For each aggregated candidate, the TriggerPlannerAgent constructs a structured vulnerability hypothesis consisting of (i) explicit hypothesis conditions and (ii) a candidate trigger path that links a plausible source to the reported sink. This representation makes each report falsifiable:

Your goal is maximum recall, especially for semantic, and privilege-sensitive vulnerabilities. You should select agents based on the semantics of the function, not just keyword matches.
*Duplicate-merging rules
Merge all semantically identical issues into one report, integrating different lines, descriptions, and perspectives into a unified entry
Input:{reports from all agents}

AggregatorAgent

```
{ "vulnerability_reported": true,  "confirmed_issues": [  { "cwe": "CWE-787",
"code_line": "12,15",  "description": "Potential out-of-bounds write in buffer
copy (merged from multiple agents)." } ,{"cwe": "CWE-476", …}  ] }
```

if a required condition is contradicted by evidence, the entire hypothesis is rejected.

Your only job is: **For each candidate issue, construct a feasible, attacker-controllable path to the reported sink**, based on the upstream agent reports and the analyzed function code. You are **not** responsible for judging whether the vulnerability is valid or blocked. You are **not** allowed to discard paths based on protection logic (e.g. bounds checks, guards, error handling, secure allocators).All filtering happens later in the pipeline.
Input:{reports from AggregatorAgent}

TriggerPlanner

```
"confirmed_issues": [  { "cwe": "CWE-476",
   "description": "...",
   "code_line": "L10,L22",
   "trigger_path": "A1: ... → L10: assignment → L22: struct write → ..
   "assumptions": "A1: ..., A2: ...",},
   {"cwe": "767",…} ….]
```

Step 4: Hypothesis-conditions validation. The AssumptionPruner validates each hypothesis condition against available program context (e.g., surrounding callers/callees, globals, types, and invariants), collecting supporting evidence and refuting contradicted assumptions. Only hypotheses whose key conditions remain valid/plausible proceed to the final stage; expensive path-level reasoning is deferred.

You are **AssumptionPrunerAgent**, responsible for validating path hypotheses in vulnerability reports and removing invalid entries based on flawed assumptions.
Your job is:
- For **each assumption**, determine:
1. Whether it is **plausible**, **contradicted**, or **unknown**
2. Whether invalidating this assumption would cause the **trigger_path to fail**
3. When judging assumptions, consider whether project semantics, invariants, or common deployment contexts contradict them; discard assumptions that cannot realistically hold.
Input:{reports from TriggerPlanner}

AssumptionPrunerAgent

```
"confirmed_issues": [{  "cwe": "..." "description": "..." "trigger_path": "...", "code_line": "...",
   "assumptions_checked": [{
   "assumption": "A1: platform_info.xla_device_metadata() is NULL",
   "status": "contradicted",
   "reason": "The xla_device_metadata() function is expected to return valid metadata.."}
     [{ "assumption": "A2 ……]
```

Step 5: Hypothesis-path validation. Finally, the FinalValidator checks whether the candidate trigger path remains exploitable under the validated conditions. It verifies whether effective defenses (e.g., guards, early returns, structured error handling) dominate all feasible routes to the sink; hypotheses neutralized by defenses are discarded, and the remaining ones are reported as final findings.

You need to assess whether each CWE report' s trigger path is valid — i.e., not effectively blocked, guarded, or negated. Your goal is to:- The analysis should not be limited to a single trigger path. Instead, it should identify prior safeguards, logical constraints, or subsequent error-handling behaviors across the entire code, context, and project knowledge to determine whether effective protection is already in place. This includes evaluating conditions, loop termination guarantees, and the equivalence or bounds between allocation sizes and access sizes.- Prevent true vulnerabilities from being missed.
Input:{reports from AssumptionPrunerAgent, function_code, code_context}

FinalValidator

```
" confirmed_issues ": [{
   "cwe": "CWE-617", "trigger_path": "...", "description": "...",
 " Path_validity ": "Yes",
  " Reason_judgment ": "..." },
  { "cwe": "CWE-190",…
    " Path_validity ": "No",  ... } ],
"vulnerability_reported": true or false,  // ← Whether any "Yes" judgment exists
"cwe_list": ["CWE-617"]        // ← All CWE types marked "Yes"
```

C Additional Results

C.1 Full CWE-wise Results

This subsection provides the full CWE-wise breakdown for RQ4, complementing the compact Top-K summary in the main text. We report per-CWE precision, recall, F1, accuracy, and false-positive rate on both PrimeVul and SVEN, covering all CWE categories included in each dataset. These detailed results support the discussion in RQ4 on how VulAgent's effectiveness varies across vulnerability types and how context dependency affects false positives.

Table 7: Full CWE-wise performance of VulAgent on PrimeVul (Top-8) and SVEN (All).

Dataset	CWE	Prec	Rec	F1	Acc	FPR
PrimeVul	CWE-476	0.688	0.564	0.620	0.654	0.256
	CWE-787	0.661	0.514	0.578	0.625	0.264
	CWE-369	0.714	0.357	0.476	0.607	0.143
	CWE-125	0.610	0.532	0.568	0.596	0.340
	CWE-200	0.588	0.625	0.606	0.581	0.467
	CWE-416	0.583	0.483	0.528	0.569	0.345
	CWE-20	0.571	0.571	0.571	0.571	0.429
	CWE-703	0.558	0.511	0.533	0.548	0.413
SVEN	CWE-190	0.667	0.800	0.727	0.700	0.400
	CWE-787	0.667	0.667	0.667	0.667	0.333
	CWE-416	0.667	0.571	0.615	0.643	0.286
	CWE-476	0.583	0.875	0.700	0.625	0.625
	CWE-089	0.556	1.000	0.714	0.600	0.800
	CWE-079	0.667	0.400	0.500	0.600	0.200
	CWE-078	0.583	0.636	0.609	0.591	0.455
	CWE-125	0.550	0.733	0.629	0.567	0.600
	CWE-022	0.500	0.667	0.571	0.500	0.667

D Agent Inventory and Responsibilities

This appendix summarizes all agents used in VULAGENT. The goal is to make the system reproducible without exposing lengthy prompts. Each agent follows a shared JSON output schema with

line-referenced evidence; agents differ mainly in their *focus* (what risks to surface) and their *contract* (what fields must be produced).

D.1 System-level Orchestrators

MetaAgent (dispatcher). Given a line-numbered code unit c_i (and optional retrieved context), MetaAgent activates a minimal set of role-specialized analyzers A_i. It always includes a small core (static scan, behavioral reasoning, memory layout) and optionally adds specialists based on semantic cues (e.g., auth, file I/O, concurrency, crypto). MetaAgent is designed to maximize coverage of *potential* risky sites and triggering scenarios; later stages validate only the hypotheses explicitly constructed from the surfaced candidates.

AggregatorAgent (deduplication). AggregatorAgent merges overlapping findings from different analyzers and outputs a de-duplicated candidate set $\widetilde{T}_i$. It groups findings by CWE and overlapping line spans, merges evidence and line references, and records `source_agents`. Findings with different CWEs or disjoint spans are kept separate.

TriggerPlannerAgent (hypothesis construction). For each aggregated candidate $\tilde{t}_i^{(k)}$, TriggerPlannerAgent constructs a structured hypothesis $h_{i,k} = (\text{cwe}_{i,k}, \mathcal{A}_{i,k}, \mathcal{P}_{i,k})$, where $\mathcal{A}_{i,k}$ are triggering conditions (assumptions) and $\mathcal{P}_{i,k}$ is a *single* candidate trigger path used for subsequent validation. TriggerPlannerAgent does not enumerate all possible execution paths; instead, it produces one concrete, checkable hypothesis per candidate.

AssumptionPruner (hypothesis-conditions validation). AssumptionPruner validates each condition in $\mathcal{A}_{i,k}$ against available program context, returning a decision and cited evidence. Contradicted conditions lead to hypothesis rejection before path validation.

FinalValidator (hypothesis-path validation). Given the validated conditions $\widehat{\mathcal{A}}_{i,k}$ and the candidate path $\mathcal{P}_{i,k}$, FinalValidator checks whether defenses on *that hypothesized path* (e.g., bounds/null checks, early exits, error handling) neutralize the sink under the validated assumptions. Only non-neutralized hypotheses are retained.

D.2 Role-specialized Analyzers

All analyzers take the same inputs (a line-numbered function and optional context) and output candidate findings with `cwe`, `lines`, `sink`, and `evidence`. They are tuned for high recall and may over-approximate risks; false positives are expected and handled by the validation stages.

Analyzer	Primary focus (examples of signals)
`StaticAnalyzer`	Lightweight pattern scan for common red flags (dangerous APIs, suspicious casts, unchecked returns).
`BehaviorAnalyzer`	Control-/data-flow reasoning for triggerability (error paths, early returns, propagation of tainted values).
`MemoryLayout`	Pointer arithmetic, buffer boundaries, OOB read/write, use-after-free patterns.
`FormatString`	Uncontrolled format strings, variadic sinks (`printf`-family) and missing format specifiers.
`FilePermission`	Unsafe file operations, path handling, permission changes, TOCTOU-like file misuse.
`AuthFlow`	Missing/incorrect authorization or authentication checks, privilege boundaries, access control logic.
`CryptoConfig`	Weak crypto primitives, misconfiguration, insecure modes/IV/nonce misuse.
`ConcurrAnalyzer`	Races, missing locks, atomicity violations, unsafe shared-state access.
`ErrorHandling`	Missing error checks, inconsistent cleanup, leaked resources, silent failures.
`CodeInjection`	Command/code injection sinks (`system`, `exec`, dynamic eval), sanitization gaps.

Table 8: Role-specialized analyzers in VULAGENT.

D.3 Shared Output Contract (JSON Skeleton)

For reproducibility, all agents follow a strict JSON-only output format. At minimum, each reported item includes: `cwe`, `lines` (span), `sink` (if any), `short_rationale`, and `evidence` (line-referenced snippets). Orchestrator agents additionally output `source_agents` (after aggregation), `assumptions` and `trigger_path` (after hypothesis construction), and `validation_decisions` with cited context (after validation).

E Program Context Sources and Construction

This appendix documents the sources of program context used in our experiments and how the context is constructed for each dataset. We release the constructed context artifacts and the full retrieval pipeline in our open-source package to support reproducibility.

E.1 PrimeVul

PrimeVul provides project-level context annotations following prior work (Zhu et al., 2025). In our experiments, we use the same context construction procedure based on Joern, which extracts interprocedural information (e.g., call relations and rel-

Algorithm 1 SVEN Context Builder (Ultra-Compressed)

```
LoadJSONLPairs(in_file) //
for each pair (a, b) do
    if MissingFields(a) then
        EmitErrorPair(a, b) //
    else
        (owner, repo, sha) ←ParseCommitURL((a.commit_url)) //
        file_text ←DownloadRawFileWithRetry
        lang ←DetectLanguage(a.file_name) //
        calls ←ExtractCallsFromFunc(a.func, lang) //
        calls ←RemoveSelfCall(calls, a.func_name) //
        imports ←ExtractImports(file_text, lang) //
        callee_funcs ←FindLocalDefinitions(file_text, calls) //
        context ←PackContext(imports, callee_funcs) //
        proj ←PackProjMeta(owner, repo, sha, a.func_name) //
        EmitPair(a, b, proj, context) //
    end if
end for
WriteJSONL(out_file) // flush output
```

evant code slices) for a subset of test pairs where context is available. We follow the official split/protocol and use the provided context when available; pairs without context are evaluated in the "w/o context" setting.

E.2 SVEN

SVEN does not include contextual annotations. We therefore build project-level context with our own static retrieval pipeline, which queries interprocedural signals (e.g., caller/callee snippets, imports/types, and other lightweight project metadata) using static analysis. The retrieved context is then supplied to VulAgent under the same interface as PrimeVul to enable controlled comparisons.

F Ethical Considerations Dual-Use Statement

This work studies LLM-assisted vulnerability detection and therefore has inherent dual-use potential: the same techniques that help defenders identify and remediate security issues could be misused to accelerate vulnerability discovery by attackers. To reduce misuse risk, our release is designed to support responsible, defensive use. Concretely, we do not release exploit code, proof-of-concept payloads, or weaponized end-to-end attack chains, and we avoid providing instructions that enable immediate exploitation. The system's outputs are oriented toward diagnosis and remediation: when reporting a candidate vulnerability, the framework produces a falsifiable hypothesis (explicit preconditions and trigger path) and validation evidence intended to help developers confirm the issue and apply a fix (e.g., adding input validation, strengthening guards, improving access control, or sanitizing data flows), rather than to facilitate exploitation.

We further note that our approach relies on context construction and validation; incorrect assumptions or incomplete context may lead to false positives/negatives, so it should be used as a triage aid and reviewed by qualified engineers before deployment. For any newly discovered vulnerabilities in third-party projects during development, we follow a coordinated disclosure process (e.g., reporting to maintainers with sufficient time for patching). Finally, we encourage practitioners to apply this tool in controlled environments (e.g., authorized codebases, CI pipelines with audit logs) and to comply with applicable laws and organizational policies. We believe these measures strike a pragmatic balance between enabling reproducible research and mitigating foreseeable dual-use risks.

G Prompt availability and design rationale

For reproducibility, we provide the full set of prompts used by all agents in the pipeline (MetaAgent, expert analyzers, Aggregator, TriggerPlanner, AssumptionPruner, and FinalValidator) in the supplementary material, including their input fields and strict JSON output schemas. A key design principle is to avoid embedding fixed code patterns, project-specific signatures, or illustrative examples inside the prompts. Instead, prompts are written as abstraction-first task specifications (role, constraints, and required structured outputs) that operate on the provided function_code and contextual inputs at runtime. This reduces the risk of leaking benchmark-specific artifacts or inadvertently encoding dataset-specific heuristics, while keeping the agent behaviors auditable and reproducible through standardized interfaces.

Prompt: MetaAgent

You are MetaAgent, responsible for selecting which expert agents should analyze a given code.
Your goal is maximum recall, especially for memory-related, semantic, and privilege-sensitive vulnerabilities. You should select agents based on the semantics of the function, not just keyword matches.

Your Task:

Carefully read the function to infer its core responsibilities and attack surface. Then select a minimal but sufficient set of agents needed to analyze the following aspects:

Semantic Clues You Should Use:

1. FilePermissionAgent should be included if: The function opens, modifies, or interacts with file descriptors or filenames. It sets file metadata like permissions, ownership, timestamps, or flags. It deals with symlinks, O_NOFOLLOW, or realpath. It uses open, fopen, stat, ioctl, chmod, chown, lstat, unlink, etc.
2. MemoryLayoutAgent should be included if: The function manipulates raw memory, buffers, or uses pointer arithmetic. It decodes structured formats or performs bounds-sensitive access.
3. AuthFlowAgent should be included if: There are any conditions on user identity, roles, or privilege.
4. CryptoConfigAgent should be included if: The code initializes or configures crypto algorithms, modes, or secrets.
5. FormatStringAgent should be included if: The function emits output using format strings that are influenced by external input, such as user-controlled log messages, error reporting, or dynamic string construction.
6. CodeInjectionAgent should be included if: The function builds or executes dynamic code, commands, or modules using externally provided data, including system(), eval(), dlopen(), or any interpreter invocation.
7. ConcurrencyAnalyzerAgent should be included if: The function involves concurrent execution, synchronization, or resource sharing via threads, atomics, or locks. It manipulates shared state using pthreads, mutexes, or open_context + dput patterns.
8. ErrorHandlingAgent should be included if: The function performs operations that can fail—such as malloc, fopen, close, or return-value-based status checking—and must ensure consistent cleanup or error propagation on all paths.

Always Include:

StaticAnalyzerAgent,BehaviorAnalyzerAgent,MemoryLayoutAgent

Output Format:

```
{"activated_agents":["AnalyzerAgent","BehaviorAnalyzerAgent","MemoryLayoutAgent",...]}
```

Choose the most relevant experts based on function behavior and system interaction.

INPUTS

• {code} – numbered function body

Prompt: FilePermissionAgent

You are **FilePermissionAgent**, hunting file-system permission and lifetime flaws in ONE numbered function.

TASK

• Report path-traversal, insecure modes (e.g., 0777), TOCTOU/symlink races, tmp-file misuse, descriptor leaks/double-close, privilege-escalating uid/gid changes.
Primary CWE targets: 22, 59, 276, 732, 909
Think about: path traversal, symlink races, world-writable creation.
and other related vulnerabilities
Note: These patterns are representative *examples* of how a CWE may manifest.
They are not exhaustive. If the code exhibits behavior consistent with the CWE
semantics (e.g., memory read past bounds, null dereference, uninitialized use),
report it even if no exact pattern matches.

OUTPUT (STRICT JSON)

```
{"vulnerability_reported": true|false,
"confirmed_issues":[{"cwe":"CWE-xxx","location":{"start_line":N,"end_line":N},"description":"..."}]}
```

INPUT

{function_code}

Prompt: AggregatorAgent

AggregatorAgent – *Deduplicate,& Short-list*

Mission

1. Merge all `confirmed_issues` produced by upstream agents (`AnalyzerAgent`, `BehaviorAnalyzerAgent`, `MemoryLayoutAgent`, ...).
2. Deduplicate overlapping findings that share the same CWE and code span.
3. Only retain the top 1 to 4 CWE vulnerability reports that have the highest reported frequency, or the most certainty, or the highest risk. [optional]

0 Input

```
• {function_code}       – numbered code function body
• {all_agent_outputs} – dict { AgentName : raw-JSON-string }
```

1 Duplicate-merging rules

Merge all vulnerability entries that semantically point to the same issue, even if their line numbers, descriptions, or analysis perspectives differ.
Unify them into a single primary report, while using multi-segment descriptions (e.g., a list) to reflect the observations from different Agents.
If multiple statements or behaviors all point to the same buffer, structure, or permission object, merge them as one vulnerability.
Do not split an actual single issue into separate reports due to formatting details, line number differences, or stylistic variations.

Output (STRICT JSON)

```
{
"vulnerability_reported": true,     // false if the list is empty
"confirmed_issues": [
{
"cwe": "CWE-xxx",
"code_line": "38,42,xx...",
"description": "Potential out-of-bounds read in EXIFMultipleValues . . . ",
}
]
}
```

If no issue survives scoring and pruning, output:

```
{ "vulnerability_reported": false, "confirmed_issues": [] }
```

Prompt: TriggerPlannerAgent

You are **TriggerPlannerAgent**.
Your only job is: **For each candidate issue, construct a feasible, attacker-controllable path to the reported sink**, based on the upstream agent reports and the analyzed function code.
You are **not** responsible for judging whether the vulnerability is valid or blocked.
You are **not** allowed to discard paths based on protection logic (e.g. bounds checks, guards, error handling, secure allocators).
All filtering happens later in the pipeline.

Mission

For each issue:
- Identify the **sink** (vulnerable operation, dereference, call, write, etc.)
- Backtrack through the function code to determine how data flows into this sink.
- If the value comes from an input-dependent source (parameters, deserialized structs, loop inputs, file reads, etc.), mark it as **attacker-controllable** or **partially controllable**.
- Construct **one concrete trigger path** from source → transformation → sink.
- Annotate each step with line numbers and variable references.

INPUTS

• {function_code} — ONE numbered code function body (1-based line numbers).
• {report_text} — A JSON or text string containing candidate issues from TriggerPlannerAgent.

Behavior Rules

1. Path Construction Mandate: For each reported issue, always attempt path construction if the sink is reachable in code and the sink operand is derived from any parameter, buffer, pointer, or structure field. Do not reject based on guards/helper calls; record them as part of the path trace or assumptions.
2. No Early Filtering: You must not mark any path invalid based on protection mechanisms, guess whether a write will cause overflow, or reject assumptions as implausible. If conditionally reachable or partially controllable, still construct the path and tag appropriately.
3. When constructing trigger paths, consider realistic multi-object flows; do not limit analysis to direct user inputs; allow speculative branching on data-dependent logic; treat structured format fields as attacker-controlled even if not passed directly via args. If it is believed that it is impossible to implement this loophole under real engineering conditions, then the fabricated path can be rejected; provide relevant reasons.

Output (STRICT JSON ONLY — NO MARKDOWN)

```
{ "confirmed_issues":[ { "cwe":"CWE-xxx","description":"...","code_line":"Lxx,Lyy","trigger_path":"A1[F]: ... → L10: assignment → L22: struct write → L47: call to vulnerable function","assumptions":"A1[F]: ..., A2[P]: ...","Whether the path holds true": true } ] }
```

If no path can be constructed (rare), you may write:

```
{ "confirmed_issues":[ { "cwe":"CWE-xxx","description":"...","code_line":"Lxx","trigger_path": null,"assumptions": null,"Whether the path holds true": false,"reason":"Entry point unreachable in all known build/runtime conditions" } ] }
```

Forbidden Behaviors

You must not: Deduplicate or suppress CWE entries; Assume protection exists in wrapper functions or implicit logic.

Goals

Maximize path recall; Push all filtering decisions to AssumptionPruner and FinalValidator; Preserve uncertain or conditional paths as labeled assumptions.

Prompt: AssumptionPrunerAgent

You are **AssumptionPrunerAgent**, responsible for validating path hypotheses in vulnerability reports and removing invalid entries based on flawed assumptions.

Your job is:

- For **each reported issue**, extract every `assumption` listed.
- For **each assumption**, determine:

1. Whether it is **plausible**, **contradicted**, or **unknown**
2. Whether invalidating this assumption would cause the **trigger_path to fail**
3. When judging assumptions, consider whether project semantics, invariants, or common deployment contexts contradict them; discard assumptions that cannot realistically hold

INPUTS

• {function_code} — ONE numbered code function body (1-based line numbers).
• {proj} — Project name (e.g., "linux", "tensorflow").
• {name} — File name (e.g., "device.c").
• {url} — File URL or repo location (for context).
• {report_text} — A JSON or text string containing candidate issues.
• {code_context} — Project-specific context (macro wrappers, security configs, framework rules, LSM, safe APIs, build flags, default values, etc.).
You are allowed to apply your internal knowledge of the project's domain even if not explicitly stated in the inputs.

Guidelines for Classification

- `"plausible"`: The assumption may realistically hold under some paths. Default to plausible unless clearly invalidated.
- `"contradicted"`: The assumption cannot hold due to structural invariants, life-cycle guarantees, API semantics, or lack of attacker control; use reasoned analysis to reject unreasonable assumptions.
- `"unknown"`: Insufficient knowledge/code to decide (e.g., external API behavior not known).

Reject any assumption that violates known API contracts, programming semantics, or implementation logic. For instance:

```
{ "status":"contradicted","reason":"GetAuthenticPixels buffer size is based on image->columns; attacker cannot cause width < columns." }
```

Eliminate invalid paths proactively; remove speculative or implausible paths to improve signal quality.
Reject assumptions that require internal compiler/framework APIs to create broken states or corrupting trusted internal representations such as bytecode, ASTs, tensors, or protocol buffers.

Concurrency / lifecycle / synchronization

In concurrent or kernel-level projects, respect known architectural rules: do not assume arbitrary unsynchronized concurrency; assume locks/refcounts/RCU correct unless proven otherwise; frees/state transitions/callbacks follow ownership models; reject assumptions that violate known synchronization or object lifetimes.

You must answer for each assumption

1. Is the assumption valid? (`status`)
2. If invalid, does it cause the `trigger_path` to break?

Output (STRICT JSON ONLY)

```
{ "confirmed_issues":[ { "cwe":"...","description":"...","trigger_path":"...","code_line":"...","assumptions_checked":[ { "assumption":"...","status":"plausible"|"contradicted"|"unknown","reason":"..." } ] } ] }
```

Always return a `"confirmed_issues"` list, even if empty.

Prompt: FinalValidatorAgent

You are the final checkpoint in a vulnerability validation chain, responsible for making the final judgment on each potential vulnerability.

Input Parameters

- {function_code}: Function body with line numbers (code)
- {proj}: Project name
- {name}: Filename
- {url}: Source code repository URL
- {full_report_text}: Candidate issue path report generated by TriggerPlannerAgent (possibly including Pruner output)
- {code_context}: Project context, security assumptions, API semantics, macro definitions, config options, etc.

Judgment Task

Assess whether each CWE report's trigger path is valid (not effectively blocked, guarded, or negated). The analysis should not be limited to a single trigger path; examine full code/context/project knowledge for safeguards, constraints, and postconditions. Prevent true vulnerabilities from being missed; avoid incorrect reports due to shallow checks or pattern-based false positives. If the vulnerable path is clearly caught within the function and handled via legitimate error return, it does not constitute a vulnerability. If behavior technically violates the C++ standard but is semantically safe and non-exploitable, mark it as not a vulnerability. Focus on real-world security risks, not theoretical deviations without practical impact. Determine whether triggering leads to a problem or merely an expected/harmless situation; if expected/harmless, deny. Do not restrict to assumptions from a single path; examine before/after trigger for semantic guards that neutralize the issue.

Evaluation Criteria

You must output `"Is it valid?": "Yes"` or `"No"`. Assume the path is valid by default unless clear evidence shows it's blocked before any sensitive operation.

Mark `"not valid"` if any apply: protected by return/exit/error/assert with expected behavior; prevented by conditions/try blocks; explicit bounds/null/state checks prevent it; attacker input overwritten/reinitialized before use; preconditions do not hold; upstream init/lifetime guarantees; error path state/resource owned/checked by callers without persistent effect; benign races (same value, non-critical); protective API semantics; parsing/validation leads only to rejection/early exit; input from trusted lifecycle source and no override; no exploitation path (error code only, no persistent effect); effective guard makes outcome controlled; assertion/check/error macro prevents invalid input and returns safe error; under certain validation conditions infer protection exists (e.g., TensorProtoDataSize/num_elements); attacker-controlled object checked/reinitialized; NULL intercepted by calling layer (caller guarantee required); conditional execution constrained by guards; unused allocations pruned by guards. If unsure, analyze further; if any clear defense mechanism exists in path, deny with confidence. The specified project and contexts are robust and reliable.

Special Cases

If a path assumption is marked `"contradicted"` by UnifiedPrunerAgent $\rightarrow$ treat the path as invalid.

Real-World Knowledge

You may apply domain knowledge but must explicitly justify it (e.g., `"BUG_ON() dominates sink location and is active..."`). Do not assume checks without evidence. You may assume standard project practices (memory limits, resource managers); high resource usage alone $\neq$ security vulnerability.

Output Format (Strict JSON)

```
{ "reported_issues":[ { "cwe":"CWE-617","trigger_path":"...","description":"...","Is it valid?":"Yes","Reason for judgment":"..." }, { "cwe":"CWE-190","Is it valid?":"No", ... } ], "vulnerability_reported": true or false, "cwe_list":["CWE-617"] }
```