# BRIDGING IMPLICIT SEMANTIC GAPS: LLM-GUIDED GRAPH COMPLETION FOR STATIC VULNERABILITY DETECTION

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Hybrid approaches that combine large language models (LLMs) with classical program analysis have advanced static vulnerability detection, yet many real-world vulnerabilities still evade detection. A key source of these false negatives is *implicit semantic gaps*: semantically mediated relations that connect attacker-controlled inputs to security-sensitive sinks but are absent from conventional interprocedural control- and data-flow graphs, leaving the end-to-end vulnerability path disconnected. We present SEMBRIDGE, a hybrid framework that bridges these implicit gaps via LLM-guided graph completion. SEMBRIDGE first applies conventional static analysis to recover explicit interprocedural structure and identify candidate sinks. It then uses LLMs to extract *bridge facts* from heterogeneous repository artifacts and materializes them as typed edges in a *Semantic Bridge-Augmented Program Graph (SBPG)*. These edges reconnect program fragments that are semantically related but structurally disconnected, enabling end-to-end vulnerability reasoning over paths that standard analyses cannot recover. On the 213-case release of CWE-Bench-Java, our approach successfully detects 125 vulnerabilities, whereas the current state-of-the-art tool, IRIS, detects 81, yielding a 54.3% relative improvement over IRIS. More importantly, SEMBRIDGE also uncovers 27 previously unknown (0-day) vulnerabilities in 50 real-world projects.

## 1 INTRODUCTION

Recent Large Language Models (LLMs) have demonstrated strong capabilities on code intelligence and reasoning tasks (Hui et al., 2024; Huang et al., 2025), prompting growing interest in applying them to static vulnerability detection (Sheng et al., 2025). Despite encouraging progress, reliably detecting vulnerabilities in real-world software remains challenging: many vulnerable paths depend on semantically mediated control and data flows that are dispersed across the codebase and difficult for purely syntactic analyses to recover.

Existing LLM-based approaches fall into two broad categories. The first uses LLMs to augment symbolic analyzers with domain knowledge such as taint specifications (Li et al., 2025e; Wang et al., 2026; Liu et al., 2023; Ghebremichael et al., 2026; Li et al., 2025a) and contextual path filtering (Lekssays et al., 2025; Li et al., 2025b; Lekssays, 2026; Li et al., 2026; Du et al., 2025; Chen et al., 2024), as exemplified by IRIS (Li et al., 2025e;d). The second uses LLMs more directly as semantic code interpreters, extracting vulnerability-relevant properties through prompt-based reasoning. LLMDFA, LLMSAN, and RepoAudit are representative (Wang et al., 2024a;b; Guo et al., 2025). Despite their differences, both categories struggle when discovery requires end-to-end flows whose critical relations are absent from the syntax-based graph.

The existing approaches suffer from two major limitations: First, prior work reasons predominantly over explicit syntactic control- and data-flow edges. Many real vulnerabilities, however, depend on *implicit semantic gaps* that leave the vulnerability path structurally disconnected even when all relevant evidence exists in the repository. Second, the evidence needed to bridge these gaps is often sparse and distributed across heterogeneous artifacts such as source code, configuration files, templates, and documentation. For prompt-driven repository auditors, this creates a repository-scale exploration problem: the system must identify a small set of bridge-relevant facts among many



Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

syntactically related but exploit-irrelevant candidates. This makes the detection problem inherently compositional and difficult for both analyzer-centric and prompt-centric methods to solve.

In this paper, we present a hybrid framework, SEMBRIDGE, for vulnerability detection that addresses both limitations by augmenting conventional program representations with LLM-extracted semantic relations, yielding a *Semantic Bridge-Augmented Program Graph (SBPG)*. Our key insight is that many missed vulnerabilities are caused not only by unknown sources or sinks, as argued by Li et al. (2025e), but also by implicit semantic gaps between them. Accurately identifying and materializing these missing relations enables SEMBRIDGE to recover vulnerability paths that conventional graph-based analyses leave disconnected.

Concretely, SEMBRIDGE addresses these two limitations as follows: First, to reduce false negatives caused by implicit semantic gaps, SEMBRIDGE uses LLM-based analysis over repository artifacts to extract semantic information about relevant code logic. This design is based on our general observations across different projects: relations and dependencies not explicitly represented in conventional control- and data-flow graphs can often be inferred from such semantic information, allowing SEMBRIDGE to reconnect disconnected program fragments and recover vulnerability paths that would otherwise be missed. Second, to avoid unguided repository-scale exploration, SEMBRIDGE adopts a vulnerability-sink-oriented, on-demand strategy. It first identifies likely vulnerable sinks and then performs backward analysis from those program points to constrain LLM reasoning to the code and artifacts most relevant to the current exploit hypothesis. Compared with prompt-driven repository auditing systems such as RepoAudit (Guo et al., 2025), this vulnerability-centric perspective provides an analysis-guided search boundary before asking the LLM to compose scattered evidence.

We evaluate SEMBRIDGE on the 213-case release of CWE-Bench-Java provided by the state-of-the-art tool IRIS (Li et al., 2025e;d). Our approach detects 125/213 vulnerabilities, outperforming IRIS by 44 cases and RepoAudit by 73 cases. The gains are concentrated on cases whose detection depends on complex, semantically mediated flows, confirming that bridging implicit semantic gaps is both a distinct challenge and a productive target for hybrid analysis. We also evaluate SEMBRIDGE on 50 real-world applications, where it detects 27 0-day vulnerabilities, demonstrating its effectiveness in real-world settings.

**Contributions.** This paper makes the following contributions:

- **Problem Identification.** We identify implicit semantic gaps as a major source of false negatives in hybrid vulnerability detection, and characterize the problem through representative real-world cases.
- **LLM-guided Graph Completion.** We propose a hybrid framework, SEMBRIDGE, that uses LLMs to extract bridge facts from heterogeneous repository artifacts and materializes them as typed edges in a Semantic Bridge-Augmented Program Graph (SBPG), enabling end-to-end vulnerability reasoning over previously unrecoverable paths.
- **Comprehensive Evaluation.** We evaluate SEMBRIDGE on CWE-Bench-Java, which contains 213 real-world vulnerabilities, and demonstrate substantial improvements over strong baselines, particularly on vulnerabilities whose detection requires bridging implicit semantic gaps. We also evaluate SEMBRIDGE on 50 real-world applications and detect 27 0-day vulnerabilities.

## 2 MOTIVATION

### 2.1 MOTIVATING EXAMPLE

**Overview.** Figure 1 shows simplified vulnerable code snippets from CVE-2023-33246, a remote code execution (RCE) vulnerability. To exploit it (Appendix A), an attacker first modifies a configuration entry with a shell command, triggering Flow #1. The command is later retrieved from the configuration field and executed by a background task, as shown in Flow #2 from Lines 25–48.

**Implicit Gaps.** Although the exploit is conceptually straightforward, recovering the end-to-end path is difficult because the relevant control and data flows contain implicit gaps:

❶ *Data-flow gap.* The two essential flows in Figure 1 (Lines 1–24 and Lines 25–48) are *syntactically disconnected in the inter-procedural data-flow graph (IDFG)*. In particular, the exploitable parameter `cmd` (Line 46) in Flow #2 has no direct data-flow dependency on the original user input `req` (Line 3) in Flow #1. Its only relevant upstream value is `RQHome` (Line 40), read from a `BrokerConfig` instance (Lines 26, 39) stored inside the `Configuration` instance `cfg` in `BrokerController` (Lines 25–30).




Figure 1: The simplified vulnerable code of our motivating example.

However, the relationship among the `Configuration` instance, the `BrokerConfig` instance, and the two flows is not represented. This disconnect arises because `FilterServerManager` (Line 32) runs through a scheduled background task (Lines 34–37). Thus, conventional static analysis sees no explicit syntactic interaction between this component and the malicious input `req` in Flow #1, and therefore fails to establish the necessary data-flow linkage.

❷ *Control-flow gap.* Flow #1 depicts how malicious input is injected into `BrokerConfig` by invoking its setter methods (Lines 20–23). However, *a conventional inter-procedural control-flow graph (ICFG) contains no syntactic edges to these setters*. The root cause is the `update` method (Lines 7–9), which uses Java reflection (Oracle Corporation, 2024; Bracha & Ungar, 2004) to invoke setter methods (e.g., `setRQHome`) based on keys in `req.body` (Line 8), writing values into objects in `confObjs` (Line 12), including `BrokerConfig`. Because reflective calls are resolved dynamically, no explicit syntactic feature identifies the invoked setter. Java polymorphism further obscures the association between the `confObjs` list of type `Object` and the concrete `BrokerConfig` instances it may contain (Gamma et al., 1995; Liskov, 1987).

### 2.2 Challenges and Solution Overview

**Challenge I: Implicit control- and data-flow gaps disconnect vulnerability paths.** As shown in § 2.1, modern language and framework features are often not adequately captured by traditional inter-procedural data-flow graphs (IDFGs) or inter-procedural control-flow graphs (ICFGs) (Livshits & Lam, 2005; Landman et al., 2017; Smaragdakis & Balatsouras, 2015). Flow-based approaches such as IRIS (Li et al., 2025e) remain constrained by these missing graph relations, while prompt-driven repository auditors such as RepoAudit (Guo et al., 2025) lack an explicit graph representation to recover and validate the disconnected source-to-sink path in Figure 1.For example, the conventional base graph cannot represent the asynchronous background thread mechanism (Lines 34–37) or the reflective setter invocation (Lines 20–23). It may still contain local call and data-flow edges inside each fragment, but it lacks the semantic edges needed to show that a request-side configuration update can later influence the command executed by the scheduled task.

To address this challenge, SemBridge uses *LLM-guided graph completion*: it recovers semantic relations that are absent from the ICFG/IDFG and materializes them as bridge edges in an SBPG (Figure 2). The key observation is that components without explicit syntactic connections often remain semantically dependent. For instance, by analyzing the `createFilterServer` and `updateBrokerConfig` methods, SemBridge can infer their producer-consumer relationship over `BrokerConfig`, thereby restoring the missing linkage between the two flows.

**Challenge II: Repository-scale exploration overwhelms prompt-driven auditing.** Although semantic relationships can in principle be leveraged to recover missing data- and control-flows,

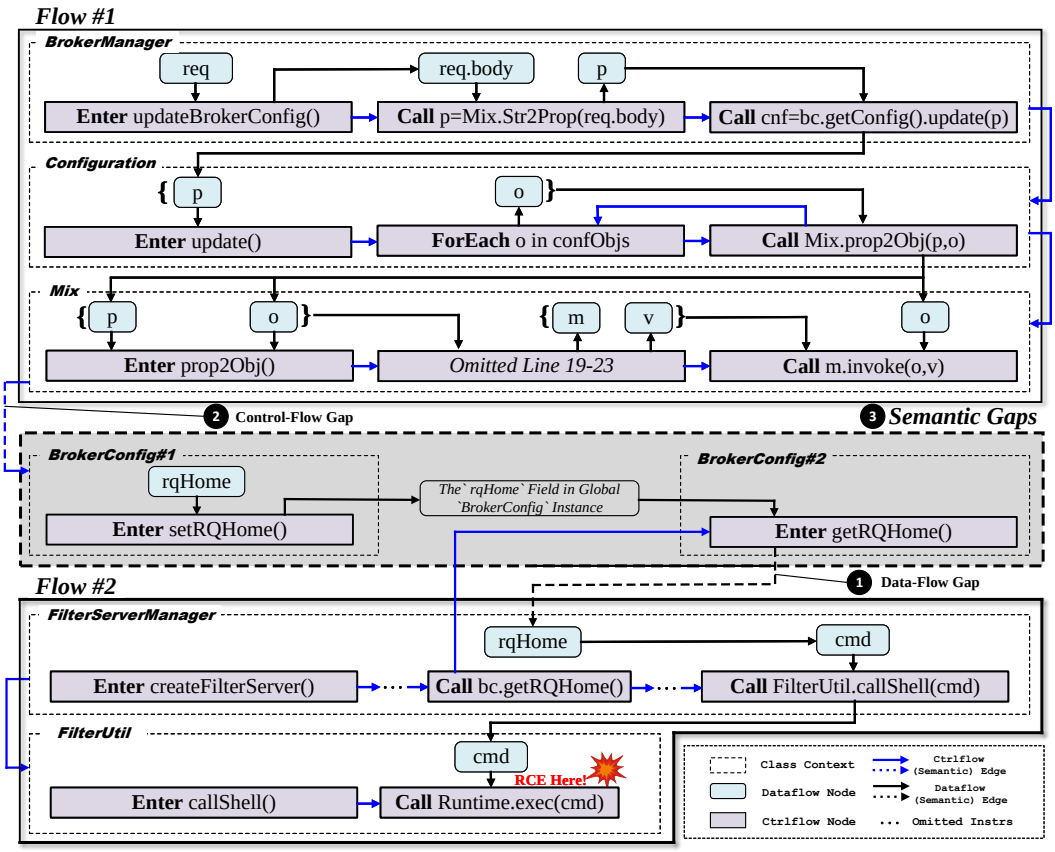


Figure 2: The simplified SBPG of the motivating example.

the practical question is *where to look*. In real repositories, bridge-relevant evidence is sparse relative to the surrounding code and is mixed with many syntactically related but exploit-irrelevant candidates. For example, even when focusing on *a single entity* such as `BrokerConfig` in Figure 1, there exist *1,944 references* spanning *203 files* across the repository. Exhaustively traversing all *1,982 classes* would result in an enormous exploration space. A purely LLM-based auditor such as `RepoAudit` (Guo et al., 2025) must either place large amounts of repository context into the prompt or rely on broad autonomous exploration. The former exceeds the useful context needed for accurate reasoning, while the latter can spend many reasoning steps on code that is semantically adjacent but unrelated to exploitability.

To address this challenge, SEMBRIDGE adopts a vulnerability-sink-oriented, on-demand analysis strategy. Instead of exploring the repository as an open-ended auditing task, SEMBRIDGE starts from a concrete security-sensitive sink and the exploit-relevant value reaching that sink, then uses the current SBPG frontier to constrain what evidence should be retrieved and composed next. This design is inspired by how human security experts conduct code audits: once a potential sink is identified, the analysis asks which upstream state, API, or artifact could control the value reaching that sink. Concretely, after identifying the sink in Figure 1 and determining that the corresponding malicious input originates from `BrokerConfig` (Lines 38–46), SEMBRIDGE prioritizes the methods that may write attacker-controlled values into this state object over the other references to `BrokerConfig`. This quickly surfaces `updateBrokerConfig` and establishes the missing linkage between the relevant code components.

# 3 DESIGN

Figure 3 shows the workflow of SEMBRIDGE, which targets implicit semantic gaps that disconnect source-to-sink paths in conventional ICFGs and IDFGs. SEMBRIDGE recovers the missing relations as bridge facts, validates them against repository evidence, and adds the validated facts as edges to a Semantic Bridge-Augmented Program Graph (SBPG).

SEMBRIDGE first builds a Semantic Database from the project's artifacts and stores program entities for later missing-edge recovery (§ 3.1). Using this database, SEMBRIDGE identifies candidate sources and sinks and constructs a Backbone SBPG over the conventional ICFG and IDFG (§ 3.2). Finally, SEMBRIDGE performs sink-guided SBPG completion and vulnerability detection (§ 3.3). Starting from each candidate sink, it tracks exploit-relevant values backward through the SBPG, validates recovered paths using forward reachability and sanitizer checks, and reports a vulnerability only after validation succeeds. When the analysis reaches an unresolved frontier, SEMBRIDGE retrieves evidence from the Semantic Database, asks the LLM analyzer to propose bridge facts, validates them against repository evidence, and materializes only validated facts as additional SBPG edges.

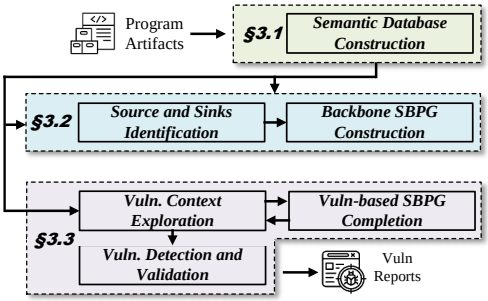


Figure 3: Workflow of SEMBRIDGE.

## 3.1 SEMANTIC DATABASE CONSTRUCTION

As § 2 illustrates, the evidence for an implicit semantic gap usually exists in the repository, but the ICFG and IDFG do not represent it as edges. SEMBRIDGE therefore builds a *Semantic Database* from the project's artifacts (Appendix B), which stores structured semantic facts and their provenance for later relation recovery (§ 3.3). The database is organized around program entities and the relations that can become graph edges. It normalizes source files, configuration files, dependency descriptors, templates, database mappings, framework conventions, and API documentation into a common schema, so that later retrieval can query evidence about a specific frontier instead of scanning the repository as plain text.

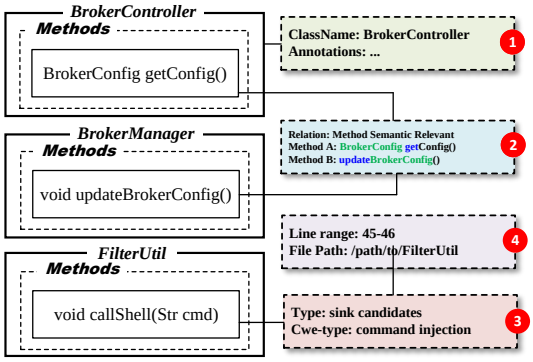


Figure 4: Simplified Semantic Database using the motivating example.

Specifically, the *Semantic Database* contains four layers (Figure 4): ❶ the Program Entity Layer defines normalized reasoning objects; ❷ the Relation Layer records typed dependencies among entities; ❸ the Security Semantic Layer identifies vulnerability-relevant roles such as potential APIs and sanitizers; and ❹ the Evidence and Provenance Layer records where each fact comes from and how it was obtained, so that facts can be validated before they become SBPG edges (Appendix C).

## 3.2 VULNERABILITY DETECTION INITIALIZATION

**Knowledge-Driven Source and Sink Identification.** Giving function slices and source/sink prompts directly to the LLM, as in prior work (Li et al., 2025d), caused many false negatives in an early version of SEMBRIDGE, because Java entry points take many forms and security-relevant sinks are often framework- or library-specific (Appendix D). SEMBRIDGE therefore uses layers ❸ and ❹ of Figure 4 to build a high-level view of the application, including its frameworks, third-party libraries, input channels, and high-risk locations, and dispatches prompts based on the template in Figure 12 to LLM sub-agents that jointly identify sources and sinks.

**Backbone SBPG Construction.** SEMBRIDGE then constructs a Backbone SBPG for the later analysis. Let $V$ contain the program nodes of the base graph and the semantic entities from repository artifacts that bridge facts refer to. We distinguish three edge sets: $E_{\text{base}}$ contains the ICFG and IDFG edges; $E_{\text{basic}}$ contains generally useful semantic edges that the base graph lacks, such as edges induced by reflective invocation or dynamic class loading; and $E_{\text{vuln}}$ contains vulnerability-specific bridge edges discovered on demand while exploring a sink. Relative to the base graph $G_0 = (V, E_{\text{base}})$, a relation $g = (u, v, \tau)$ is a *semantic gap* if repository or framework evidence establishes an execution-relevant relation of type $\tau$ between $u$ and $v$, $g \notin E_{\text{base}}$, and the absence of $g$ disconnects a candidate source-to-sink path. The type $\tau$ is data or state propagation, reflection or dynamic dispatch, deferred execution, framework or configuration binding, or cross-artifact interaction. A validated bridge fact $(u, v, \tau, \epsilon)$ adds the edge $(u, v, \tau)$ to the SBPG and keeps the evidence $\epsilon$ as provenance.

The Backbone SBPG $G_b = (V, E_{\text{base}} \cup E_{\text{basic}})$ initializes vulnerability discovery, and sink-guided completion produces the completed SBPG $G = (V, E_{\text{base}} \cup E_{\text{basic}} \cup E_{\text{vuln}})$. SEMBRIDGE builds $E_{\text{basic}}$ in advance and reuses it across candidate sinks, while $E_{\text{vuln}}$ starts empty and grows only when a sink-specific frontier needs more evidence (§3.3).

This stage constructs only $E_{\text{basic}}$, using two groups of sub-agents: the first models dynamic mechanisms built into Java, such as reflective invocation, and the second handles the framework-specific and less standardized dynamic behaviors of third-party dependencies by querying the Evidence and Provenance Layer ❹ (Figure 4). Every candidate edge, including the vulnerability-specific bridges of §3.3, must pass four checks: (1) both endpoints exist; (2) the relation type is allowed and compatible with the endpoints; (3) all referenced evidence exists in repository artifacts or cited documentation; and (4) the code, configuration, or documented framework behavior supports the relation. Neither analysis rules nor the LLM can add an edge that fails these checks.

### 3.3 SINK-GUIDED SBPG COMPLETION AND VULNERABILITY DISCOVERY

**Vulnerability-oriented Context Exploration and SBPG Completion.** Algorithm 1 gives the sink-guided completion procedure.

**Algorithm 1:** SBPG Completion

**Input:** Sinks $S_s$, Backbone SBPG $G_b$, Sem. DB $D$, LLM $\mathcal{A}$
**Output:** Completed SBPG $G$, Candidate Paths $P$

1 $G \leftarrow G_b$, $P \leftarrow \emptyset$;
2 $Q \leftarrow$ initQueue($S_s$);
3 **while** $Q \neq \emptyset$ **do**
4  $(s, x, \pi) \leftarrow$ pop($Q$);
5  $F \leftarrow$ backTrack($G$, $s$, $x$);
6  **for** $f$ ***in*** $F$ **do**
7  **if** *isUserInput*($f$) **then**
8  $P \leftarrow P \cup$ closePath($\pi$, $f$);
9  **else**
10  $ctx \leftarrow$ collectCtx($f$);
11  $C \leftarrow$ retrieveMethods($D$, $ctx$);
12  $C^\star \leftarrow$ rankTopK($\mathcal{A}$, $C$, $ctx$, 10);
13  **for** $m$ ***in*** $C^\star$ **do**
14  $E \leftarrow$ validateBridge($G$, $m$, $f$, $ctx$);
15  **if** $E \neq \emptyset$ **then**
16  $G \leftarrow G \cup E$;
17  $Q \leftarrow Q \cup$ makeSinks($m$, $f$, $E$);
18 **return** $G$, $P$;

SEMBRIDGE initializes the working SBPG and a queue of candidate sinks (Lines 1–2), then repeatedly pops an exploit-relevant sink parameter and backward-tracks it over the current graph (Lines 3–5). If a frontier already corresponds to user input, SEMBRIDGE closes a candidate vulnerability path (Lines 6–8). Otherwise, it compresses the frontier into a context containing the variable, type, enclosing class, and accessed fields; retrieves related methods from the Semantic Database; asks the LLM analyzer to rank the top candidates; validates vulnerability-specific bridge facts; materializes valid edges into $E_{\text{vuln}}$; and enqueues the selected methods as new sink frontiers (Lines 9–17). The loop returns the completed SBPG and candidate paths after all unresolved frontiers are processed (Line 18). Compressing the frontier into this context keeps bridge recovery bounded. The LLM only decides which methods or artifacts can explain one unresolved frontier of the current exploit hypothesis, so repository exploration stays tied to the SBPG state and does not turn into broad prompt-driven auditing.

**LLM Inputs and Output Constraints.** For each unresolved frontier, the LLM receives the frontier variable with its type, field, and enclosing class, the sink API and vulnerability type, local graph facts, candidate methods, code excerpts, and retrieved evidence, and returns structured JSON with candidate entity identifiers, the proposed relation, and its evidence (Appendix E gives the output formats). The LLM cannot add edges or report vulnerabilities: outputs that reference nonexistent entities or evidence, or that fail the checks in §3.2, are discarded.

**Semantics-Guided Vulnerability Detection and Validation.** Backward completion shows that a candidate path is connected, but not that the upstream method can drive the downstream frontier at run time. SEMBRIDGE therefore validates each candidate with a forward reachability analysis over the completed SBPG from the recovered source to the sink, which tracks the exploit-relevant parameters and asks the LLM analyzer whether any method applied to them is a sanitizer. A candidate is reported only if the sink is forward-reachable and no effective sanitizer neutralizes the exploit-relevant data.

# 4 EVALUATION

We evaluate SEMBRIDGE by answering four research questions:

- **RQ1:** How effective is SEMBRIDGE at detecting known vulnerabilities compared with state-of-the-art tools?
- **RQ2:** How much does each component of SEMBRIDGE contribute to its effectiveness?
- **RQ3:** How does the choice of LLM affect SEMBRIDGE?
- **RQ4:** How effective is SEMBRIDGE at discovering 0-day vulnerabilities?

**Implementation.** We implemented SEMBRIDGE in about 10K lines of Python on top of CodeQL (GitHub, 2026), which extracts the initial ICFG/IDFG, candidate sinks, call relations, method signatures, field accesses, and local data-flow summaries. The Semantic Database combines structured and vector storage: program entities, semantic relations, security roles, and provenance are stored in SQLite tables (SQLite Development Team, 2026), and artifact evidence is split into 512-token chunks with a 128-token overlap and indexed by FAISS (Douze et al., 2024) as 1,024-dimensional vectors linked to file paths, line ranges, artifact types, and entity IDs. We use Claude Sonnet 4.6 (Anthropic, 2026) as the LLM backbone with temperature 0, a 60K-token input budget, and a 4K-token output limit per query. All experiments run on a 64-vCPU AWS EC2 instance with 1 TB of RAM (Appendix F).

**Benchmark and Baselines.** We use CWE-Bench-Java (Li et al., 2025c), which contains 213 confirmed CVEs from real-world Java projects; we manually identified and verified the missing vulnerable commits of nine CVEs, so all cases have complete pre- and post-patch versions. Because the benchmark does not label every possible tool report, RQ1 measures recall over the labeled CVEs and does not estimate precision; we checked every detection counted for SEMBRIDGE against the vulnerable code and its patch.

We compare SEMBRIDGE with two baselines: i) *IRIS* (Li et al., 2025e), the state-of-the-art hybrid static detector that uses LLMs to infer taint specifications (latest open-source v2 branch), and ii) *RepoAudit* (Guo et al., 2025), a repository-level LLM auditor. RepoAudit originally targets only memory leaks, null pointer dereferences, and use-after-free bugs, so we extend its vulnerability profiles for this benchmark and run it with the same LLM backbone and decoding configuration as SEMBRIDGE (Appendix G).

Table 1: Detection results on CWE-Bench-Java; CWE mapping is in Appendix L. Shading shows deltas against the stronger baseline.

| | | Effectiveness Comparison (RQ1) | | | Ablation Study (RQ2) | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Vuln. Type** | **Vuln. #** | **RepoAudit** | **IRIS** | **SEMBRIDGE** | **SEMBRIDGE-NSS** | **SEMBRIDGE-NSE** | **w/o Structured Retrieval** | **Base Sinks Only** | **w/o Evidence Validation** | **Direct Context** |
| Path Traversal/File Access | 63 | 22 | 40 | 60 (+20) | 52 (+12) | 43 (+3) | 50 (+10) | 54 (+14) | 51 (+11) | 45 (+5) |
| Cross-site Scripting | 39 | 8 | 17 | 28 (+11) | 20 (+3) | 19 (+2) | 21 (+4) | 23 (+6) | 23 (+6) | 19 (+2) |
| Code Injection | 24 | 7 | 15 | 20 (+5) | 17 (+2) | 15 (0) | 16 (+1) | 17 (+2) | 16 (+1) | 15 (0) |
| Broken Access Control | 28 | 1 | 0 | 0 (-1) | 0 (-1) | 0 (-1) | 0 (-1) | 0 (-1) | 0 (-1) | 0 (-1) |
| Denial of Service | 15 | 1 | 0 | 0 (-1) | 0 (-1) | 0 (-1) | 0 (-1) | 0 (-1) | 0 (-1) | 0 (-1) |
| OS Command Injection | 13 | 4 | 6 | 7 (+1) | 6 (0) | 5 (-1) | 5 (-1) | 5 (-1) | 5 (-1) | 5 (-1) |
| Information Disclosure | 9 | 1 | 0 | 0 (-1) | 0 (-1) | 0 (-1) | 0 (-1) | 0 (-1) | 0 (-1) | 0 (-1) |
| Deserialization | 7 | 1 | 0 | 1 (0) | 1 (0) | 1 (0) | 1 (0) | 1 (0) | 1 (0) | 1 (0) |
| XXE Injection | 6 | 4 | 0 | 6 (+2) | 5 (+1) | 4 (0) | 5 (+1) | 4 (0) | 4 (0) | 4 (0) |
| SQL Injection | 3 | 1 | 1 | 1 (0) | 1 (0) | 1 (0) | 1 (0) | 1 (0) | 1 (0) | 1 (0) |
| SSRF | 2 | 1 | 2 | 2 (0) | 2 (0) | 2 (0) | 2 (0) | 2 (0) | 2 (0) | 1 (-1) |
| Other | 4 | 1 | 0 | 0 (-1) | 0 (-1) | 0 (-1) | 0 (-1) | 0 (-1) | 0 (-1) | 0 (-1) |
| ***Overall*** | 213 | 52 | 81 | 125 (+44) | 104 (+23) | 90 (+9) | 101 (+20) | 107 (+26) | 103 (+22) | 91 (+10) |

## 4.1 RQ1: EFFECTIVENESS

**Overall Results.** As shown in Table 1, SEMBRIDGE detects 125 of the 213 vulnerabilities (58.7%), 44 more than IRIS (81/213, 38.0%) and 73 more than RepoAudit (52/213, 24.4%). Missing source/sink

specifications explain only part of the baselines' misses; many others are caused by semantic gaps that IRIS cannot bridge within the ICFG and IDFG. RepoAudit does not reconstruct complete paths, but its DoS and access-control detections show that pure LLM-based code understanding can find some property violations beyond taint flows.

**Complementarity.** SEMBRIDGE detects 75 of the 81 CVEs found by IRIS and 44 of the 52 found by RepoAudit, plus 34 CVEs that both miss. Of the 10 CVEs it misses but a baseline detects, the four found only by RepoAudit are property violations that the taint-style formulation does not model (Appendix H).

**False Negatives.** Of the 88 benchmark vulnerabilities that SEMBRIDGE misses, our manual inspection attributes 56 (63.6%) to unsupported non-taint vulnerability models, 14 (15.9%) to library-native runtime semantics, 11 (12.5%) to incomplete multi-step bridge inference, and 7 (8.0%) to conservative bridge validation, so only 32 are directly related to bridge recovery (Appendix I).

### 4.2 RQ2: ABLATION STUDY

To evaluate the contribution of semantic reasoning, we compare SEMBRIDGE with the six ablated variants in Table 1: i) SEMBRIDGE-NSS removes semantic-guided source and sink identification and uses the base static analyzer with predefined specifications; ii) SEMBRIDGE-NSE keeps semantic-guided source and sink identification but adds no LLM-extracted bridge edges to the SBPG; iii) *w/o Structured Retrieval* replaces structure-aware retrieval with vector-based text retrieval; iv) *Base Sinks Only* replaces knowledge-guided sink identification with the sinks from the base analyzer and predefined specifications; v) *w/o Evidence Validation* keeps only the schema and endpoint checks needed to build graph edges; and vi) *Direct Context* gives the LLM the retrieved evidence and validated bridge facts as text without adding them to the SBPG. Variants iii)–vi) each change a single component and keep all other settings, including the retrieval and analysis budgets, unchanged.

The ablation results show that each component of SEMBRIDGE, together with its corresponding design, contributes to the overall effectiveness, since removing or replacing any of them reduces the number of detected vulnerabilities. Removing semantic-guided source and sink identification (SEMBRIDGE-NSS) loses 21 of the 125 detections, mainly because predefined specifications miss framework-specific entry points and library-specific security APIs. Removing semantic edge augmentation (SEMBRIDGE-NSE) loses 35, so many vulnerabilities remain unreachable even when the relevant sources and sinks are known. Of these 35 bridge-dependent cases, 18 need only data-flow bridges, 7 only control-flow bridges, and 10 both (Appendix H); data-flow gaps thus occur in 28 cases (80.0%) and control-flow gaps in 17 (48.6%). The two components are complementary, and semantic edge augmentation contributes the larger gain. Text retrieval loses 24 cases (*w/o Structured Retrieval*), so retrieving program entities and their relations finds more of the methods that explain an unresolved frontier. Using only the base analyzer's sinks loses 18 cases (*Base Sinks Only*), because predefined sink specifications miss many framework- and library-specific security APIs. Removing the evidence and relation-consistency checks loses 22 cases (*w/o Evidence Validation*), so validation contributes to detection in addition to keeping unsupported edges out of the SBPG. Giving the validated bridge facts to the LLM as text loses 34 cases (*Direct Context*), only one fewer than SEMBRIDGE-NSE: without materialization, the facts recover almost none of the gain from graph completion, because only graph edges persist as typed state that guides later retrieval and can be composed and checked for connectivity and sanitization.

### 4.3 RQ3: IMPACT OF MODEL CHOICE

To limit cost, we study model sensitivity on a 25-vulnerability injection-style subset of CWE-Bench-Java that covers common injection families and semantic-gap-heavy cases (Appendix K). Besides the default Claude Sonnet 4.6, we evaluate the Bedrock-hosted Claude Haiku 4.5, Qwen3 Coder Next, and DeepSeek V3.2 (Amazon Web Services, 2026b;c;a).

As shown in Figure 5, Claude Sonnet 4.6 detects 23/25 cases, followed by Qwen3 Coder Next (21/25), DeepSeek V3.2 (20/25), and Claude Haiku 4.5 (17/25). Stronger code-reasoning models extract bridges more accurately, but all backbones keep most of the effectiveness, so the design does not depend on a single model.

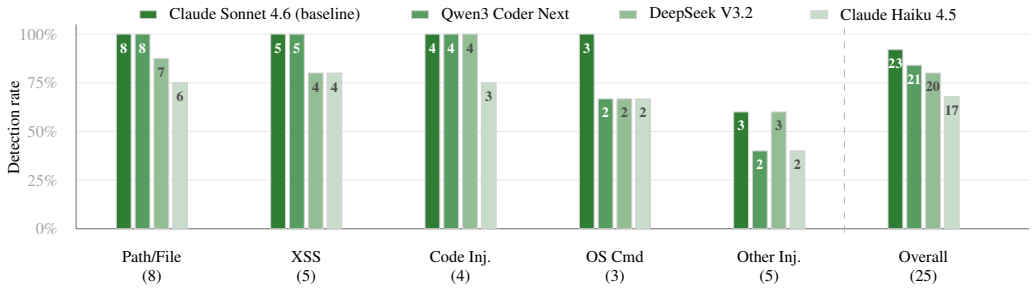


Figure 5: Model sensitivity on the injection-style subset.

### 4.4 RQ4: Zero-Day Discovery

On 50 popular open-source Java projects, SemBridge discovers 27 previously unknown vulnerabilities in 12 projects, 16 of which require relations recovered by SemBridge. We count a finding as a zero-day only if, when found, it was present in the analyzed version, had no matching CVE, GHSA, advisory, issue, or patch, had a manually verified source-to-sink path and security impact, and did not share its root cause with another finding. All findings were reported to and acknowledged by the affected vendors or developers, and we are working with the maintainers on fixes (case study in Appendix J).

## 5 Limitations and Future Work

SemBridge relies on the LLM to infer correct bridge facts; when the repository evidence is weak, implicit, or scattered across many files, the model may miss key relations. Future work should improve retrieval, validate uncertain bridges at run time, and explore specialized models for cheaper and more reliable bridge extraction (see also Appendix M).

## 6 Related Work

**LLM-Augmented Static Vulnerability Analysis**. Traditional static vulnerability detection relies on explicit program representations such as interprocedural control-flow graphs, data-flow graphs, taint analysis, and code property graphs (Livshits & Lam, 2005; Avgustinov et al., 2016; Yamaguchi et al., 2014; Smaragdakis & Balatsouras, 2015). Recent LLM-assisted systems infer security specifications or synthesize analysis logic such as taint specifications, checkers, and vulnerability queries (Li et al., 2025e; Wang et al., 2026; Liu et al., 2023; Ghebremichael et al., 2026; Li et al., 2025a; Yang et al., 2025), or use LLMs for path reasoning, false-positive reduction, and context-aware classification (Lekssays et al., 2025; Du et al., 2025; Chen et al., 2024; Li et al., 2025b; Lekssays, 2026; Li et al., 2026). These approaches complement SemBridge but assume that the graph already contains the vulnerability-relevant relations; when a semantic gap disconnects the path, better specifications or downstream reasoning cannot restore it, so SemBridge adds validated bridge edges before taint analysis.

**LLM-Based Repository-Level Vulnerability Auditing**. Another line of work uses LLMs directly as semantic analyzers or repository-level auditors (Wang et al., 2024a;b; Guo et al., 2025); the closest system, RepoAudit (Guo et al., 2025), audits repositories autonomously with LLM agents. These systems can use semantic clues that fixed rules cannot encode, but their prompt-driven analysis has no program representation that a static analyzer can query, validate, and reuse. SemBridge combines sink-guided exploration with graph completion, so the LLM answers targeted questions and the taint analysis reuses its validated results.

## 7 Conclusion

We presented SemBridge, a hybrid framework that bridges implicit semantic gaps, the relations between sources and sinks that conventional program representations miss, by extracting bridge facts from repository artifacts, validating them, and adding them as SBPG edges. On CWE-Bench-Java, SemBridge detects 44 more vulnerabilities than the strongest baseline, and it discovers 27 previously unknown vulnerabilities in real-world projects.

### AI USE STATEMENT

LLMs are a core component of the proposed method: SEMBRIDGE uses them for semantic bridge extraction, candidate ranking, and sanitizer validation. Sections 3 and 4 describe their role and configuration, Section 4.3 studies the impact of the backbone model, and Appendix E provides the prompt templates. Beyond the method itself, we used generative AI tools to help implement parts of the system and the experiment scripts, to draft and translate parts of the paper, and to edit the text for readability. We did not use generative AI tools to propose hypotheses, design the methodology or experiments, interpret results, or generate or clean datasets. The authors reviewed and tested all AI-assisted code and reviewed and revised all AI-assisted text, and all references were checked against bibliographic databases. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

### ETHICS STATEMENT

This work studies automated vulnerability detection, which is dual-use: capabilities that help developers find and fix vulnerabilities could also help attackers identify exploitable code paths. We reported all real-world 0-day findings to the affected vendors or developers (Section 4.4), anonymize product-specific identifiers in the zero-day case study (Appendix J), and present exploits only in minimized or harmless proof form. Appendix N further discusses these security impacts. The work does not involve human subjects or crowdsourcing, and it does not release a pretrained model or a scraped high-risk dataset.

### REPRODUCIBILITY STATEMENT

Section 4 describes the implementation, benchmark, baselines, and LLM backbone and decoding configuration, and Appendix F gives the hardware and software environment. Appendix E provides the key prompt templates, Appendix G describes how we extend the RepoAudit baseline, Appendix H reports the benchmark annotation audit, the CVE-level overlap, and the gap-type results, Appendix K details the model-choice experiment, and Appendix L gives the CWE-to-vulnerability-family mapping for CWE-Bench-Java. The benchmark is the public CWE-Bench-Java release. Some of the zero-day vulnerabilities reported in Section 4.4 are still being fixed together with the affected maintainers; to avoid exposing unpatched vulnerabilities, we do not include the implementation in this submission. Upon acceptance, we will open-source SEMBRIDGE, including its prompts, JSON schemas, evaluation scripts, and per-CVE results.

## REFERENCES

Amazon Web Services. DeepSeek V3.2 - Amazon Bedrock. Amazon Bedrock documentation, 2026a. URL https://docs.aws.amazon.com/bedrock/latest/userguide/model-cards-deepseek.html. Accessed 2026-05-07.

Amazon Web Services. Claude Haiku 4.5 - Amazon Bedrock. Amazon Bedrock documentation, 2026b. URL https://docs.aws.amazon.com/bedrock/latest/userguide/model-card-anthropic-claude-haiku-4-5.html. Accessed 2026-05-07.

Amazon Web Services. Qwen3 Coder Next - Amazon Bedrock. Amazon Bedrock documentation, 2026c. URL https://docs.aws.amazon.com/bedrock/latest/userguide/model-card-qwen-qwen3-coder-next.html. Accessed 2026-05-07.

Anthropic. Introducing Sonnet 4.6. Anthropic product announcement, 2026. URL https://www.anthropic.com/news/claude-sonnet-4-6. Accessed 2026-05-06.

Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. QL: Object-oriented queries on relational data. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, volume 56 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 2:1–2:25. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi: 10.4230/LIPIcs.ECOOP.2016.2. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2016.2.

Gilad Bracha and David Ungar. Mirrors: Design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2004.

Jinbao Chen, Hongjing Xiang, Zuohong Zhao, Luhao Li, Yu Zhang, Boyao Ding, Qingwei Li, and Songyuan Xiong. Utilizing precise and complete code context to guide LLM in automatic false positive mitigation, 2024. URL https://arxiv.org/abs/2411.03079.

Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. The Faiss library. *arXiv preprint arXiv:2401.08281*, 2024. doi: 10.48550/arXiv.2401.08281. URL https://arxiv.org/abs/2401.08281.

Xueying Du, Kai Yu, Chong Wang, Yi Zou, Wentai Deng, Zuoyu Ou, Xin Peng, Lingming Zhang, and Yiling Lou. Minimizing false positives in static bug detection via LLM-enhanced path feasibility analysis, 2025. URL https://arxiv.org/abs/2506.10322.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

Jonah Ghebremichael, Saastha Vasan, Saad Ullah, Greg Tystahl, David Adei, Christopher Kruegel, Giovanni Vigna, William Enck, and Alexandros Kapravelos. Multi-agent taint specification extraction for vulnerability detection, 2026. URL https://arxiv.org/abs/2601.10865.

GitHub. CodeQL. Software and documentation, 2026. URL https://codeql.github.com/docs/codeql-overview/about-codeql/. Accessed 2026-05-06.

Jinyao Guo, Chengpeng Wang, Xiangzhe Xu, Zian Su, and Xiangyu Zhang. RepoAudit: An autonomous LLM-agent for repository-level code auditing. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 21083–21100. PMLR, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/guo25n.html.

Siming Huang, Tianhao Cheng, Jason Klein Liu, Weidi Xu, Jiaran Hao, Liuyihan Song, Yang Xu, Jian Yang, Jiaheng Liu, Chenchen Zhang, Linzheng Chai, Ruifeng Yuan, Xianzhen Luo, Qiufeng Wang, YuanTao Fan, Qingfu Zhu, Zhaoxiang Zhang, Yang Gao, Jie Fu, Qian Liu, Houyi Li, Ge Zhang, Yuan Qi, Yinghui Xu, Wei Chu, and Zili Wang. OpenCoder: The open cookbook for top-tier code large language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 33167–33193, Vienna, Austria, July 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.acl-long.1591. URL https://aclanthology.org/2025.acl-long.1591/.

Binyuan Hui, Jian Yang, Zeyu Cui, et al. Qwen2.5-Coder Technical Report. *arXiv preprint arXiv:2409.12186*, 2024. doi: 10.48550/arXiv.2409.12186. URL https://arxiv.org/abs/2409.12186.

Davy Landman, Alexander Serebrenik, and Jurgen J. Vinju. Challenges for static analysis of Java reflection – literature review and empirical study. In *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering (ICSE)*, pp. 507–518, 2017. doi: 10.1109/ICSE.2017.53.

Ahmed Lekssays. Bridging code property graphs and language models for program analysis. In *4th International Workshop on Software Vulnerability Management (SVM '26)*, 2026. doi: 10.1145/3786165.3788441. URL https://arxiv.org/abs/2603.24837.

Ahmed Lekssays, Hamza Mouhcine, Khang Tran, Ting Yu, and Issa Khalil. LLMxCPG: Context-aware vulnerability detection through code property graph-guided large language models. In *34th USENIX Security Symposium (USENIX Security 25)*, pp. 489–507, 2025. URL https://www.usenix.org/conference/usenixsecurity25/presentation/lekssays.

Penghui Li, Songchen Yao, Josef Sarfati Korich, Changhua Luo, Jianjia Yu, Yinzhi Cao, and Junfeng Yang. Neuro-symbolic static analysis with LLM-generated vulnerability patterns, 2025a. URL https://arxiv.org/abs/2504.16057.

Xiang Li, Yueci Su, Jiahao Liu, Zhiwei Lin, Yuebing Hou, Peiming Gao, and Yuanchao Zhang. VULSOLVER: Vulnerability detection via LLM-driven constraint solving, 2025b. URL https://arxiv.org/abs/2509.00882.

Yikun Li, Ting Zhang, Jieke Shi, Chengran Yang, Junda He, Xin Zhou, Jinfeng Jiang, Huihui Huang, Wen Bin Leow, Yide Yin, Eng Lieh Ouh, Lwin Khin Shar, and David Lo. Beyond function-level analysis: Context-aware reasoning for inter-procedural vulnerability detection, 2026. URL https://arxiv.org/abs/2602.06751.

Ziyang Li, Saikat Dutta, and Mayur Naik. CWE-Bench-Java. Dataset in the IRIS GitHub repository, 2025c. URL https://github.com/iris-sast/iris/tree/master/data. Accessed 2026-05-06.

Ziyang Li, Saikat Dutta, and Mayur Naik. iris-sast/iris. GitHub repository, 2025d. URL https://github.com/iris-sast/iris. Includes the expanded CWE-Bench-Java release; accessed 2026-04-08.

Ziyang Li, Saikat Dutta, and Mayur Naik. IRIS: LLM-assisted static analysis for detecting security vulnerabilities. In *International Conference on Learning Representations*, 2025e. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/582d4e27fa24168f3af1f4582655034b-Abstract-Conference.html.

Barbara Liskov. Keynote address – data abstraction and hierarchy. In *Addendum to the Proceedings on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '87)*, pp. 17–34, 1987. doi: 10.1145/62138.62141.

Puzhuo Liu, Chengnian Sun, Yaowen Zheng, Xuan Feng, Chuan Qin, Yuncheng Wang, Zhenyang Xu, Zhi Li, Peng Di, Yu Jiang, and Limin Sun. Harnessing the power of LLM to support binary taint analysis, 2023. URL https://arxiv.org/abs/2310.08275.

Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In *USENIX Security Symposium*, 2005.

National Vulnerability Database. CVE-2023-33246 Detail. https://nvd.nist.gov/vuln/detail/CVE-2023-33246, 2025. Accessed 2026-04-08.

Oracle Corporation. The java reflection api. https://docs.oracle.com/javase/tutorial/reflect/, 2024. Accessed: 2026-04-16.

Ze Sheng, Zhicheng Chen, Shuning Gu, Heqing Huang, Guofei Gu, and Jeff Huang. Llms in software security: A survey of vulnerability detection techniques and insights, 2025. URL https://arxiv.org/abs/2502.07049.

Yannis Smaragdakis and George Balatsouras. Pointer analysis. *Foundations and Trends in Programming Languages*, 2015.

SQLite Development Team. SQLite. Software and documentation, 2026. URL https://www.sqlite.org/about.html. Accessed 2026-05-06.

Chengpeng Wang, Wuqi Zhang, Zian Su, Xiangzhe Xu, Xiaoheng Xie, and Xiangyu Zhang. LLMDFA: Analyzing dataflow in code with large language models. In *Advances in Neural Information Processing Systems*, volume 37, 2024a. doi: 10.52202/079017-4181. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/ed9dcde1eb9c597f68c1d375bbecf3fc-Abstract-Conference.html.

Chengpeng Wang, Wuqi Zhang, Zian Su, Xiangzhe Xu, and Xiangyu Zhang. Sanitizing large language models in bug detection with data-flow. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pp. 3790–3805, Miami, Florida, USA, November 2024b. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-emnlp.217. URL https://aclanthology.org/2024.findings-emnlp.217/.

Claire Wang, Ziyang Li, Saikat Dutta, and Mayur Naik. QLCoder: A query synthesizer for static analysis of security vulnerabilities. In *International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=J91IKwJrqv.

Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pp. 590–604, 2014.

Chenyuan Yang, Zijie Zhao, Zichen Xie, Haoyu Li, and Lingming Zhang. Knighter: Transforming static analysis with LLM-synthesized checkers, 2025. URL https://arxiv.org/abs/2503.09002.

## A THE EXPLOIT FOR CVE-2023-33246

```java
Properties properties = new Properties();
properties.setProperty("filterServerNums", "1"); // trigger Flow #2
properties.setProperty(
    "rocketmqHome",                                  // RQHome in Figure 1
    "-c bash${IFS}-c${IFS}\"touch${IFS}/tmp/sembridge_poc\";");

DefaultMQAdminExt admin = new DefaultMQAdminExt();
admin.setNamesrvAddr("localhost:9876");
admin.start();
admin.updateBrokerConfig("192.168.87.128:10911", properties); // Flow #1
admin.shutdown();
```

Figure 6: Simplified exploit code for CVE-2023-33246. The code uses RocketMQ's administrative client to send a broker-configuration update containing a harmless proof command.

Figure 6 shows the exploit for CVE-2023-33246 (National Vulnerability Database, 2025), which follows the two stages of the motivating example. In Flow #1, the exploit creates two configuration entries and sends them through `DefaultMQAdminExt.updateBrokerConfig`. This client API wraps the properties into a configuration-update request to the broker, whose request handler then calls `updateBrokerConfig(req)` in Figure 1 (Lines 3–9). The attacker-controlled `rocketmqHome` entry corresponds to the `RQHome` field in the simplified example. In `Configuration.update`, `Mix.prop2Obj` (Lines 16–23) translates the property key into a setter name and calls the setter through reflection, which writes the attacker-controlled value into the `BrokerConfig` object in `confObjs`. This step relies on the reflective control-flow gap, because the ICFG has no call edge to the setter.

The request handler does not execute the command. Flow #2 runs later, when the scheduled filter-server manager in Figure 1 (Lines 32–37) executes. The `filterServerNums=1` entry makes `createFilterServer` try to start a filter server. This background task reads the same `BrokerConfig` instance, takes the attacker-controlled `RQHome`/`rocketmqHome` value, concatenates it into a shell command (Lines 38–41), and passes the resulting `cmd` to `FilterUtil.callShell` and then `Runtime.exec` (Lines 44–46). The injected value uses shell argument expansion through `${IFS}` to turn the configured home directory into an executable shell fragment; the payload shown here only creates `/tmp/sembridge_poc` as a harmless proof marker. The exploit therefore also relies on the data-flow gap described in the main text, because the admin request and the `Runtime.exec` sink have no syntactic connection and interact only through the shared configuration state.

## B THE ARTIFACTS FOR SEMANTIC DATABASE CONSTRUCTION

SEMBRIDGE builds the Semantic Database from the following types of artifacts.

*i) Source Code.* SEMBRIDGE uses AST-based analysis to extract program entities, such as class names, method names, and enum constants, and builds a *Metadata Dependency Graph* from them. For example, in Figure 1, the analysis of `BrokerController` shows that it depends on `BrokerConfig` and `Configuration`, so later lookups can prioritize these related classes.

*ii) Configuration Files.* Java applications often load components dynamically, which is a major source of semantic gaps. SEMBRIDGE parses configuration files to recover information that the source code alone does not contain. In Figure 7(a), the configuration routes all incoming HTML requests through `SanitizeTransformer`, which the Apache Cocoon framework loads dynamically. A statically constructed ICFG misses the corresponding call edges unless the configuration is analyzed.

*iii) Database Artifacts.* Database schemas, including tables, fields, and foreign keys, describe how the application stores and relates its data. In Figure 7(b), a MyBatis mapping file relates `selectItem` and `updateItem` to their data object `com.dao.ItemDao`. Such information helps recover data dependencies across application layers.

*iv) Dependency Management Files.* Static analyzers usually exclude third-party libraries to reduce analysis cost. SEMBRIDGE does the same but adds a lightweight dependency analysis. It queries external services such as Maven repositories to find libraries with known vulnera-

```xml
<map:match pattern="user/input">
  <!-- other tags … -->
  <map:transform type="safehtml">
</map:match>
<map:components>
  <map:transformer name="safehtml"
     src="com.example.HTMLSantizer"/>
</map:components>
```

(a) An example of `sitemap.xmap`

bilities and potential sink APIs, and it identifies the frameworks that the application uses, which helps interpret dynamic behaviors. In Figure 7(c), the FastJson dependency declares a vulnerable version (1.2.47), and the Spring Boot plugin shows that the application is built on the Spring framework. The later analysis then focuses on the mechanisms and APIs of these frameworks.

*v) API Documentation.* SEMBRIDGE parses official documentation and Javadoc to extract information that is hard to recover from source code. Many behaviors that matter for semantic gaps, such as dynamic execution, deserialization, and callbacks, are documented but do not appear in static call graphs. With this information, SEMBRIDGE can identify dangerous APIs and relate program fragments that look unrelated in the code, which helps reconstruct complete vulnerability paths.

# C DETAILED DESIGN FOR THE FOUR TYPES OF SEMANTIC KNOWLEDGE

This appendix formalizes the four categories of knowledge in the *Semantic Database*. The formalization defines the logical interface of the facts and leaves the physical storage schema open. Each record is normalized into a fact that the source/sink identification agents and the SBPG completion procedure can retrieve and audit.

The four categories correspond to the layers in Figure 4. The Program Entity Layer defines the objects that the analysis reasons about, the Relation Layer records typed dependencies among them, the Security Semantic Layer stores vulnerability-relevant roles and behavior summaries, and the Evidence and Provenance Layer records where each fact comes from and how it was produced, so that facts can be validated before they become SBPG edges.

**BNF representation.** We represent the Semantic Database as a set of typed facts grouped into four layers, and Figures 8–11 give their logical interface. We omit low-level terminals such as concrete identifiers, timestamps, line-range syntax, and JSON metadata, which are implementation details produced by artifact parsers, CodeQL queries, documentation analyzers, or LLM agents.

```
<program-entity-layer> ::= <entity-fact>
                         | <entity-fact> <program-entity-layer>

<entity-fact> ::= "Entity" "(" <entity-id> "," <entity-kind> ","
                  <name> "," <where> "," <attributes> ")"

<entity-kind> ::= "code" | "artifact" | "configuration"
                | "dependency" | "document"

<attributes> ::= <signature> | <version> | <parser-metadata>
               | <domain-note>
```

Figure 8: BNF for the Program Entity Layer, which places code entities, repository artifacts, configuration entries, dependencies, and documentation in one namespace.

```
<relation-layer> ::= <relation-fact>
                   | <relation-fact> <relation-layer>

<relation-fact> ::= "Relation" "(" <relation-id> ","
                    <source-entity-id> "," <target-entity-id> ","
                    <relation-kind> "," <intent> "," <support> ")"

<relation-kind> ::= "program" | "binding" | "dependency"
                  | "dynamic-dispatch" | "cross-artifact"
                  | "semantic-bridge"

<intent> ::= "analysis" | "vulnerability" | "evidence"
```

Figure 9: BNF for the Relation Layer, which records typed dependencies among entities, such as framework bindings, dynamic dispatch, cross-artifact links, and candidate semantic bridges.

```
<security-semantic-layer> ::= <security-fact>
                            | <security-fact> <security-semantic-layer>

<security-fact> ::= <role-fact>
                  | <behavior-summary>
                  | <project-summary>

<role-fact> ::= "SecurityRole" "(" <fact-id> ","
                <entity-id> "," <security-role> ","
                <vulnerability-type> "," <support> ")"

<behavior-summary> ::= "Behavior" "(" <summary-id> ","
                       <entity-id> "," <effect> ","
                       <condition> "," <support> ")"

<project-summary> ::= "ProjectSummary" "(" <project-id> ","
                      <frameworks> "," <entrypoints> ","
                      <security-assumptions> "," <support> ")"

<security-role> ::= "source" | "sink" | "sanitizer"
                  | "guard" | "high-risk"
```

Figure 10: BNF for the Security Semantic Layer, which stores the vulnerability-relevant roles and summaries used to build source/sink prompts.

```
<evidence-provenance-layer> ::= <evidence-fact>
                              | <evidence-fact> <evidence-provenance-layer>

<evidence-fact> ::= <artifact-fact>
                  | <evidence-span>
                  | <provenance-record>

<artifact-fact> ::= "Artifact" "(" <artifact-id> ","
                    <artifact-kind> "," <where> "," <digest> ")"

<evidence-span> ::= "Evidence" "(" <evidence-id> ","
                    <artifact-id> "," <where> ","
                    <excerpt> ")"

<provenance-record> ::= "Provenance" "(" <fact-id> ","
                        <evidence-id-list> "," <extractor> ","
                        <confidence> ")"

<artifact-kind> ::= "source" | "config" | "template"
                  | "build" | "dependency" | "documentation"
```

Figure 11: BNF for the Evidence and Provenance Layer, which links each entity, relation, and security fact to the artifacts or external documentation it comes from.

**Interpretation.** Because the Program Entity Layer puts all artifacts in one namespace, code entities, configuration keys, templates, dependencies, and documentation chunks can be linked in the same way. The Relation Layer holds two kinds of edges: general edges needed by the analysis, such as configuration bindings and framework callbacks, and candidate edges for a specific vulnerability, such as semantic bridges that CodeQL-guided exploration checks later. The Security Semantic Layer provides the information for the source and sink prompts, including framework entry points, library-specific sink APIs, sanitizers, and security guards. The Evidence and Provenance Layer ties each fact to concrete artifacts or external documentation, so an LLM inference without supporting evidence cannot enter the SBPG.

## D Challenges for Source and Sink Identification

**Source Identification.** In principle, every variable that may receive user input should be marked as a taint source. Java applications, however, receive user input in many different ways, because the

ecosystem contains many frameworks and third-party libraries. Web development alone has several frameworks with different input-handling mechanisms, such as Spring[1], Struts2[2], JAX-RS[3], and Servlet[4], and message queues and RPC frameworks add further variants. Without knowledge of these frameworks and libraries, an LLM cannot reliably decide whether a method or its parameters are sources.

**Sink Identification.** Sinks raise a similar problem. A method name alone does not show whether a call into a third-party library is security-sensitive. For example, a deserialization call such as `JSON.parse` from `FastJson` is not necessarily vulnerable, because newer versions of the library may have fixed the issue.

## E Partial Key Prompt Templates

This appendix lists shortened versions of the main prompt templates in the current SemBridge implementation. We omit routine JSON handling and long evidence payloads. At run time, the placeholders are filled from the Semantic Database, CodeQL facts, artifact chunks, documentation evidence, and the target-profile configuration.

**Knowledge-driven source/sink identification.** Figure 12 shows the prompt template of the multi-agent source/sink subsystem in Section 3.2. All seven specialized agents use this template and differ only in their focus.

**Template E.1: Multi-agent Source/Sink/Sanitizer Identification**

```
SYSTEM:
You are the {agent_name} agent for a Java vulnerability scanner.
Goal: identify grounded {source|sink|sanitizer} facts for {role_goal}.

AGENT FOCUS:
- framework_source / config_route_source: framework-managed ingress,
  configured routes, servlet/RPC/message handlers, template callbacks.
- third_party_api_sink: direct sinks or wrappers around high-risk platform
  and library APIs (command execution, SQL, deserialization, SSRF, JNDI, etc.).
- consumer_usage / semantic_summary / codeql_fact / sanitizer: infer roles
  from usage sequences, persisted summaries, CodeQL facts, and sanitization logic.

KEY INPUTS:
target profile; project summary; selected method inventory; semantic summaries;
retrieved documentation/artifact evidence; consumer usage; entity dependencies;
existing security facts; curated high-risk API catalog.

OUTPUT JSON ONLY:
facts[{entity_id, role, vulnerability_type, parameter, confidence, evidence}]

RULES:
Only cite input entity_id values. Ground every fact in supplied evidence.
Prefer reachable wrapper methods over hidden callee APIs. Omit uncertain facts.
```

Figure 12: Prompt template for knowledge-driven source, sink, and sanitizer identification. SemBridge fills it once per enabled sub-agent, resolves the returned facts by `entity_id`, and writes the merged roles into the Semantic Database and the SBPG.

**Semantic Database comprehension prompts.** Figure 13 shows the prompts that build project-level and method-level semantic facts before source/sink identification. Their outputs fill the Security Semantic Layer and the Relation Layer, which the template in Figure 12 later uses.

[1] `https://docs.spring.io/spring-framework/reference/`
[2] `https://struts.apache.org/getting-started/`
[3] `https://jakarta.ee/specifications/restful-ws/`
[4] `https://jakarta.ee/specifications/servlet/`

**Template E.2: Semantic Knowledge Extraction**

```
PROJECT SUMMARY:
Infer purpose, frameworks, entrypoints, security assumptions, and confidence
from target profile, method samples, and documentation evidence.

EVIDENCE SELECTION:
Select evidence chunks that describe a target method's purpose, usage, security
behavior, side effects, preconditions, sources, or sinks.
Output: valuable_chunk_ids.

METHOD SECURITY SUMMARY:
For each method, summarize grounded security behavior from signature, source
excerpt, selected evidence, and consumer usage.
Output: {summary, roles, preconditions, side_effects, confidence}.
Allowed roles: ingress, sink_wrapper, state_writer, state_reader, dispatcher,
deferred_trigger, reflective_dispatch, sanitizer, config_binding,
cross_artifact_link.

SEMANTIC RELEVANCE:
Given a target method and candidates, return relevant_ids when methods are
designed to be used together, affect the same state, bridge control/data flow,
or one changes how the other behaves.
```

Figure 13: Prompts for constructing the Semantic Database. They summarize the project's purpose, select useful evidence chunks, summarize the security behavior of each method, and select related method pairs for later bridge recovery.

**Sink-guided completion and validation prompts.** Figure 14 shows the two prompts used during SBPG completion and vulnerability validation. The ranking prompt orders the candidate methods by how likely they are to resolve the current frontier, and the sanitizer prompt keeps SEMBRIDGE from reporting paths that effective validation or escaping neutralizes.

**Template E.3: Sink-guided Ranking and Validation**

```
CANDIDATE RANKING:
Rank candidate methods or artifacts for semantic bridge validation.

Key context:
frontier variable/type/field/enclosing class; sink API; vulnerability type;
incoming/outgoing edge kinds.

Key candidate evidence:
method identity and signature; calls; read/write fields; source excerpt;
semantic summaries; retrieved evidence.

Return: ranked_ids only.
Prefer candidates that read/write the same state, use reflection, provide
deferred execution, define framework routing, or expose non-canonical ingress.

SANITIZER JUDGMENT:
Given vulnerability_type and methods on a recovered taint path, return
sanitizer_ids only for effective validation, authorization, escaping, encoding,
canonicalization, or allow-list checks. Be conservative.
```

Figure 14: Prompts for sink-guided SBPG completion and path validation. The ranker only orders candidate bridge endpoints, and the sanitizer judge only identifies effective mitigation methods on a path that has already been recovered.

## F EXPERIMENTAL ENVIRONMENT

All experiments were conducted on an AWS EC2 Linux instance with a 64-vCPU Intel(R) Xeon(R) 6975P-C processor and 1 TB of RAM, running Ubuntu 24.04.3 LTS and OpenJDK 21.0.10. To keep the execution environment consistent across benchmark subjects, we set up and run the applications under test with Docker 29.3.0 and Docker Compose 5.1.0.

## G REPOAUDIT ENHANCEMENT

Upstream RepoAudit exposes only a small set of data-flow bug detectors, which is insufficient for evaluating it on CWE-Bench-Java. We therefore extended RepoAudit only through its intended customization interface. Specifically, we added Java vulnerability profiles for the vulnerability categories used in our evaluation, including source/sink specifications and category-specific validation prompts. These profiles allow RepoAudit to run on the same benchmark categories as the other tools, including path traversal/file access, cross-site scripting, SQL injection, command/code injection, deserialization, XXE, SSRF, information disclosure, denial of service, and access-control bugs.

To keep the comparison fair, the enhanced RepoAudit preserves the original repository-level auditing and DFBScan workflow. It does not use SEMBRIDGE's Semantic Database, SBPG construction, semantic bridge extraction, or CodeQL-guided graph completion.

## H ADDITIONAL EVALUATION RESULTS

This appendix provides additional details for the benchmark annotations, the CVE-level overlap among the evaluated tools, and the gap-type analysis in Section 4.

**Benchmark annotations.** Of the 213 CVEs in CWE-Bench-Java, 204 (95.8%) came with complete pre-patch and post-patch commit annotations, while nine lacked the vulnerable-commit annotation. We manually identified and verified the missing commits, so all 213 evaluated CVEs have complete version-level annotations. We also manually checked every SEMBRIDGE report counted as a detection against the corresponding vulnerable code and patch.

**CVE-level overlap.** Table 2 gives the CVE-level overlap among the three tools: SEMBRIDGE uniquely detects 34 CVEs, while 10 CVEs are detected by at least one baseline but not by SEMBRIDGE.

| Detection region | CVEs |
|---|---|
| SEMBRIDGE only | 34 |
| IRIS only | 2 |
| RepoAudit only | 4 |
| SEMBRIDGE and IRIS only | 47 |
| SEMBRIDGE and RepoAudit only | 16 |
| IRIS and RepoAudit only | 4 |
| All three tools | 28 |
| None of the three tools | 78 |
| **Total** | **213** |

Table 2: CVE-level overlap among SEMBRIDGE, IRIS, and RepoAudit.

**Data-flow and control-flow gaps.** To isolate the effect of semantic graph completion, we examine the 35 CVEs detected by the full SEMBRIDGE but missed by SEMBRIDGE-NSE; they account for the entire difference between 125 and 90 detections in Table 1. We classify a case as a *data-flow gap* when the missing relation transfers an attacker-controlled value across state, configuration, serialization, or another implicit data channel, and as a *control-flow gap* when the missing relation represents an invocation or execution dependency, such as reflection, callbacks, scheduled tasks, or framework dispatch. A case counts as *both* when recovering the complete vulnerability path requires both kinds of relations. Table 3 reports the results by vulnerability type.

## I FALSE NEGATIVE ANALYSIS

We manually inspect the 88 benchmark cases missed by SEMBRIDGE and assign each to one of four causes (Table 4). The last three causes, which cover 32 cases, are directly related to bridge recovery.

**Unsupported vulnerability models (56 cases).** SEMBRIDGE currently formulates detection mainly as taint-style reachability from attacker-controlled inputs to security-sensitive sinks. This formulation

| Vulnerability type | Data-flow only | Control-flow only | Both | Bridge-dependent total |
|---|---|---|---|---|
| Path Traversal/File Access | 9 | 3 | 5 | 17 |
| Cross-site Scripting | 5 | 2 | 2 | 9 |
| Code Injection | 2 | 1 | 2 | 5 |
| OS Command Injection | 1 | 1 | 0 | 2 |
| XXE Injection | 1 | 0 | 1 | 2 |
| **Overall** | **18** | **7** | **10** | **35** |

Table 3: Gap types among the 35 CVEs that SEMBRIDGE detects only with semantic edge augmentation.

| Cause | Cases | Share | Explanation |
|---|---|---|---|
| Unsupported vulnerability models | 56 | 63.6% | The vulnerability is not naturally expressed as taint-style source-to-sink reachability. |
| Library-native runtime semantics | 14 | 15.9% | The required relation depends on behavior inside a third-party library or runtime mechanism for which the repository provides insufficient evidence. |
| Incomplete multi-step bridge inference | 11 | 12.5% | The LLM recovers some relevant relations but fails to connect all stages of a long source-to-sink path. |
| Conservative bridge validation | 7 | 8.0% | A possible bridge is identified, but its supporting evidence is insufficient for safe graph materialization. |
| **Total** | **88** | **100%** | |

Table 4: Causes of the 88 benchmark vulnerabilities missed by SEMBRIDGE.

does not directly capture access-control bugs, resource-exhaustion DoS, information disclosure, and other property violations whose exploitability depends on missing authorization checks, unsafe default configurations, protocol-level state, logging behavior, or resource bounds, and which involve no tainted value reaching a conventional sink. This limitation is reflected in Table 1, where SEMBRIDGE reports no cases in these families.

**Library-native deserialization and runtime semantics (14 cases).** SEMBRIDGE can detect simple deserialization cases where attacker-controlled data explicitly flows into sinks such as `readObject` or Fastjson parsing APIs. However, it does not model third-party native deserialization mechanisms, gadget-chain behavior, or library-internal dispatch such as XStream. More broadly, highly customized class loading, dependency injection, event dispatch, native calls, or concurrency patterns may still leave control- or data-flow gaps in the SBPG.

**Incomplete multi-step bridge inference (11 cases).** In these cases, the relevant evidence is distributed across several state updates, configuration artifacts, callbacks, or background tasks. The LLM recovers part of the required relation chain but does not connect all stages of the long source-to-sink path, so the path remains disconnected in the SBPG.

**Conservative bridge validation (7 cases).** To avoid false positives, SEMBRIDGE only materializes bridge facts that can be grounded in repository evidence and pass deterministic consistency checks. This conservative strategy may discard plausible but weakly evidenced relations, especially when the relevant evidence is split across multiple files or only implied by framework conventions. In these 7 cases, SEMBRIDGE identifies a possible bridge, but the relation does not pass the evidence and consistency checks.

## J ZERO-DAY CASE STUDY

This appendix describes one previously unknown vulnerability that SEMBRIDGE found in a real-world Java workflow application. The issue was not yet public at the time of writing, so we anonymize product-specific identifiers and reduce the code to the relevant data flow. In the application, users can create a task whose execution content is stored as a JSON string. When the task is started, the server copies this JSON into a task instance and sends only the instance identifier to a worker process. The worker reloads the instance, dispatches it to a handler selected by a type field, deserializes the JSON, and evaluates one nested map value as a Spring Expression Language (SpEL) expression.

```
class CreateEndpoint {
  Result create(CreateRequest r) {
    StoredTask t = new StoredTask();
    t.kind = r.kind;
    t.configJson = r.configJson;        // attacker-controlled JSON
    taskStore.save(t);
    return ok(t.id);
  }
}

class StartEndpoint {
  Result start(long taskId, Vars runtimeVars) {
    StoredTask t = taskStore.find(taskId);
    TaskInstance i = new TaskInstance();
    i.kind = t.kind;
    i.configJson = t.configJson;        // persistent state copy
    i.vars = merge(t.vars, runtimeVars);
    instanceStore.save(i);
    workerClient.postStart(i.id);       // only the id crosses the process boundary
    return ok(i.id);
  }
}

class WorkerEndpoint {
  Result start(String instanceId) {
    TaskInstance i = instanceStore.find(instanceId);
    Handler h = handlerFactory.create(i.kind);
    h.setInstance(i);
    executor.submit(() -> h.execute()); // deferred execution
    return ok();
  }
}

class SelectedHandler extends Handler {
  void execute() {
    StepSpec spec = Json.parse(instance.configJson, StepSpec.class);
    String response = runConfiguredAction(spec);
    patchVars(spec.varMap, response);
  }

  void patchVars(Map<String,String> varMap, String response) {
    Context ctx = new Context(Json.parse(response));
    for (String key : varMap.keySet()) {
      String expr = varMap.get(key);
      String value = spel.parseExpression(expr).getValue(ctx, String.class);
      instance.outVars.put(key, value);
    }
  }
}
```

Figure 15: Anonymized and simplified code for the zero-day case study. Red highlights mark taint-carrying fields, purple highlights mark semantic-bridge objects and APIs, and green highlights mark key propagation, dispatch, or sink calls.

**Vulnerability path.** Figure 15 shows the minimized path. The source is the `configJson` field of the external create request, and the payload sits in the nested map that later becomes `StepSpec.varMap`. The sink is `spel.parseExpression(expr).getValue(...)` in `patchVars`. The complete flow is:

$$\begin{aligned} \texttt{CreateRequest.configJson} &\rightarrow \texttt{StoredTask.configJson} \\ &\rightarrow \texttt{TaskInstance.configJson} \\ &\rightarrow \texttt{StepSpec.varMap[*]} \rightarrow \texttt{parseExpression}. \end{aligned}$$

No sanitizer, allow-list, or expression-language restriction is applied before the sink, so a user who can create and start such a task can make the application evaluate an attacker-controlled string as a SpEL expression.

**Why conventional analysis misses it.** The vulnerable path crosses four semantic gaps that a conventional ICFG/IDFG represents poorly or not at all. First, the create endpoint stores the value in a persistent task record, and a later start action reads it. Second, the master component sends the worker only an instance identifier, and the worker reloads the data from a shared store. Third, a factory map selects the concrete handler by the `kind` discriminator, so the call from the worker entry point to `SelectedHandler.execute` cannot be resolved statically. Fourth, the sink argument is

nested in a JSON string and appears only after the framework deserializes it into `StepSpec.varMap`. Each fragment is simple on its own; the exploit path appears only when the analysis recognizes that all fragments operate on the same task instance.

**How SEMBRIDGE connects the flow.** SEMBRIDGE first uses the Security Semantic Layer and its library/API knowledge to identify `parseExpression(...).getValue(...)` as a code-execution sink. From this sink, sink-guided completion tracks the sink argument backward to the map value `varMap[*]`, where the static graph reaches an unresolved frontier: `varMap` comes from the JSON deserialization of `instance.configJson`, and no Java statement assigns it. SEMBRIDGE retrieves evidence about the handler, the JSON target class, and the task-instance fields, and adds a bridge edge from `TaskInstance.configJson` to `StepSpec.varMap`.

The next frontier is `TaskInstance.configJson`. Conventional analysis sees that the worker reads this field, but cannot tell where the instance comes from, because the master-worker request carries only an identifier. From semantic summaries and relation facts, SEMBRIDGE infers that `instanceStore.save(i)` in the start action and `instanceStore.find(instanceId)` in the worker entry point form a producer-consumer pair over the same task instance, and adds a persistent-state bridge edge across the process boundary. It also recognizes `handlerFactory.create(i.kind)` as a dispatch on a type field and links the type value to the handler that parses `configJson`. Finally, it connects the creation stage to the start stage, because `StoredTask.configJson` is copied into `TaskInstance.configJson` when a stored task is launched.

Once these bridge facts are validated and added to the SBPG, the fragments form one path from the external input to the SpEL evaluation. Forward validation confirms that the path is reachable through the create-and-start workflow and that no effective sanitizer lies between the JSON map value and the expression sink. In this case, the LLM recovers what persistence, triggering by identifier, handler dispatch, and JSON field projection mean in this application, and SEMBRIDGE turns this knowledge into bridge edges that static taint analysis can follow.

## K MODEL CHOICE EXPERIMENT DETAILS

For RQ3, we select 25 injection-style vulnerabilities from CWE-Bench-Java. The subset contains 8 path traversal/file-access cases, 5 XSS cases, 4 code-injection cases, 3 OS-command-injection cases, and 5 other injection-style cases, including XXE, deserialization, SQL injection, and SSRF. We prioritize cases whose detection requires semantic gap recovery and avoid over-sampling repeated cases from the same project.

We run the same SEMBRIDGE pipeline and decoding configuration for all model backbones. Claude Sonnet 4.6 is the default backbone used in the main evaluation. The three additional models are selected from models available through Amazon Bedrock: Qwen3 Coder Next as a code-specialized model, DeepSeek V3.2 as a recent general coding/reasoning model, and Claude Haiku 4.5 as the lightweight Claude-family model. Before we completed the experiment, Claude Haiku 4.6 was not available in the Bedrock model catalog.

| Vulnerability subset | Claude Sonnet 4.6 | Qwen3 Coder Next | DeepSeek V3.2 | Claude Haiku 4.5 |
|---|---|---|---|---|
| Path traversal/file access (8) | 8 | 8 | 7 | 6 |
| XSS (5) | 5 | 5 | 4 | 4 |
| Code injection (4) | 4 | 4 | 4 | 3 |
| OS command injection (3) | 3 | 2 | 2 | 2 |
| Other injection-style (5) | 3 | 2 | 3 | 2 |
| **Overall (25)** | **23** | **21** | **20** | **17** |

Table 5: Detailed RQ3 model-sensitivity results on the injection-style subset.

## L MAPPING BETWEEN CWE AND VULNERABILITY TYPES IN CWE-BENCH-JAVA

Table 6 reports the vulnerability-family mapping used for CWE-Bench-Java. The benchmark contains 213 CVEs across 45 distinct CWEs. We normalize CWE identifiers before aggregation because the source metadata uses inconsistent zero padding, e.g., `CWE-022` and `CWE-22` are treated as the same CWE.

| # | Vulnerability family | Count | Pct. | Underlying CWEs |
|---|---|---|---|---|
| 1 | Path Traversal / File Access | 63 | 29.6% | CWE-22 (60), CWE-44 (1), CWE-23 (1), CWE-178 (1) |
| 2 | Cross-Site Scripting (XSS) | 39 | 18.3% | CWE-79 (38), CWE-116 (1) |
| 3 | Authentication / Authorization / Access Control | 28 | 13.1% | CWE-287 (5), CWE-862 (3), CWE-284 (3), CWE-288 (2), CWE-863 (2), CWE-264 (1), CWE-269 (1), CWE-346 (1), CWE-285 (1), CWE-668 (1), CWE-297 (1), CWE-352 (1), CWE-290 (1), CWE-522 (1), CWE-327 (1), CWE-267 (1), CWE-276 (1), CWE-426 (1) |
| 4 | Code Injection / Dynamic Eval | 24 | 11.3% | CWE-94 (23), CWE-913 (1) |
| 5 | Denial of Service / Resource | 15 | 7.0% | CWE-400 (5), CWE-770 (3), CWE-1333 (2), CWE-835 (2), CWE-190 (1), CWE-407 (1), CWE-1325 (1) |
| 6 | OS Command Injection | 13 | 6.1% | CWE-78 (13) |
| 7 | Information Disclosure | 9 | 4.2% | CWE-200 (5), CWE-532 (3), CWE-312 (1) |
| 8 | Deserialization | 7 | 3.3% | CWE-502 (7) |
| 9 | XML External Entity (XXE) | 6 | 2.8% | CWE-611 (6) |
| 10 | SSRF | 2 | 0.9% | CWE-918 (2) |
| 11 | SQL / Query Injection | 3 | 1.4% | CWE-89 (2), CWE-83 (1) |
| 12 | Other | 4 | 1.9% | CWE-20 (2), CWE-113 (1), CWE-164 (1) |
| | **Total** | **213** | **100%** | **45 distinct CWEs** |

Table 6: Mapping between vulnerability families and normalized CWE IDs in CWE-Bench-Java. Counts in parentheses denote the number of CVEs assigned to each CWE within the family.

## M FUTURE WORK

**Automatic Exploit Generation.** Validated SBPG paths provide a structured starting point for proof-of-concept generation. Future work can combine these paths with sandbox feedback to synthesize exploits that remain explainable and are easier to validate.

**Beyond Taint.** Semantic bridge completion can also support vulnerabilities that are not explicit source-to-sink flows, such as missing authorization and resource-exhaustion DoS. Future work can encode property-specific invariants as graph facts and check them with lightweight constraint or state reasoning.

## N SECURITY IMPACTS

SEMBRIDGE helps developers and security analysts find vulnerabilities that conventional static analysis misses, which can improve the security of widely used software. Automated vulnerability detection is also dual-use, because the same capabilities can help attackers find exploitable code paths. We mitigate this risk by reporting real-world findings to maintainers, anonymizing undisclosed case-study details, and presenting exploits only in minimized or harmless proof form.