FINENIB: A QUERY SYNTHESIZER FOR STATIC ANALYSIS OF SECURITY VULNERABILITIES

Anonymous authors
Paper under double-blind review

ABSTRACT

Static analysis tools provide a powerful means to detect security vulnerabilities by specifying *queries* that encode vulnerable code patterns. However, writing such queries is challenging and requires diverse expertise in security and program analysis. To address this challenge, we present *FineNib* – an agentic framework that automatically synthesizes queries in CodeQL, a powerful static analysis engine, directly from a given CVE metadata. FineNib embeds an LLM in a synthesis loop with execution feedback, while constraining its reasoning using a custom MCP interface that allows structured interaction with a Language Server Protocol (for syntax guidance) and a RAG database (for semantic retrieval of queries and documentation). This approach allows FineNib to generate syntactically and semantically valid security queries. We evaluate FineNib on 176 existing CVEs across 111 Java projects. Building upon the Claude Code agent framework, FineNib synthesizes correct queries that detect the CVE in the vulnerable but not in the patched versions for 53.4% of CVEs. In comparison, using only Claude Code synthesizes 10% correct queries. Our generated queries achieve an F1 score of 0.7. In comparison, the general query suites in IRIS (a recent LLM-assisted static analyzer) and CodeQL only achieve F1 scores of 0.048 and 0.073, highlighting the benefit of FineNib's specialized synthesized queries.

1 INTRODUCTION

Security vulnerabilities continue to grow at an unprecedented rate, with over 40,000 Common Vulnerabilities and Exposures (CVEs) reported in 2024 alone (CVE, 2025). Static analysis, a technique to analyze programs without executing them, is a common way of detecting vulnerabilities. Static analysis tools such as CodeQL (Github, 2025), Semgrep (Semgrep, 2023), and Infer (Meta, 2025) are widely used in industry. They provide domain-specific languages that allow specifying vulnerability patterns as queries. Such queries can be executed over structured representations of code, such as abstract syntax trees, to detect potential security vulnerabilities.

Despite their widespread use, existing query suites of static analysis tools are severely limited in coverage of vulnerabilities and precision. Extending them is difficult even for experts, as it requires knowledge of unfamiliar query languages, program analysis concepts, and security expertise. Incorrect queries can produce false alarms or miss bugs, limiting the effectiveness of static analysis. Correct queries can enable reliable detection of real vulnerabilities, supporting diverse use-cases such as regression testing, variant analysis, and patch validation, among others (Figure 1).

Meanwhile, CVE databases (MITRE, 2025; NIST, 2025; GitHub, 2025) provide rich information about security vulnerabilities, including natural language descriptions of vulnerability patterns and records of buggy and patched versions of the affected software repositories. This resource remains largely untapped in the automated construction of static analysis queries. Recent advances in LLMs, particularly in code understanding and generation, open up the possibility of leveraging this information to automatically synthesize queries from CVE descriptions, thereby bridging the gap between vulnerability reports and practical detection tools.

Synthesizing such queries poses significant challenges. The syntax of static analysis query languages is low-resource, richly expressive, and evolves continually. A typical query, such as the one in Figure 2(b) specifying a global dataflow pattern leaves ample room for errors in describing predicates for sources, sinks, sanitizers, and taint propagation steps. Even if the generated syntax is correct, success is measured by whether the query can identify at least one execution path travers-

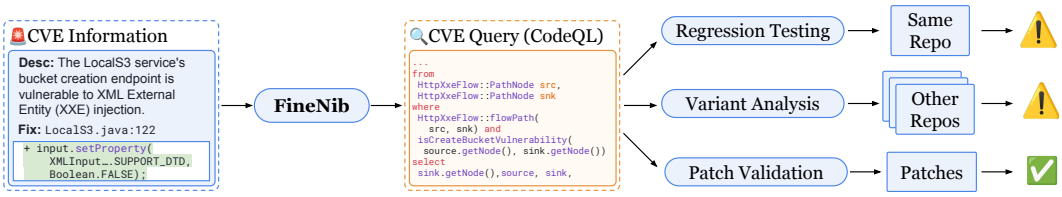


Figure 1: A CodeQL query capturing a vulnerability pattern is synthesized by FineNib from an existing CVE and subsequently reused for regression testing, variant analysis, or patch validation.

ing the bug location in the vulnerable version while producing no matches in the patched version. Achieving this requires understanding the CVE context at the level of abstract syntax trees, such as code differences that introduce a sanitizer to prevent a flow from a source to a sink. Complicating matters further, reasoning about the code changes alone is often insufficient: sources, sinks, and taint propagation steps may reside in parts of the codebase far from the modified functions or files, and the vulnerability itself may involve non-trivial dataflow chains across these components. Thus, a correct query must not only integrate information from multiple locations across the program but also capture the intricate propagation patterns to accurately characterize the vulnerability.

In this paper, we present **FineNib** – an agentic framework that synthesizes queries in CodeQL, a powerful static analysis engine, directly from a given CVE metadata. We select CodeQL because it has the richest query language, which allows capturing complex inter-procedural vulnerability patterns. FineNib addresses the above challenges by embedding an LLM in a structured synthesis loop that incorporates execution feedback to verify query correctness and allows interactive reasoning using a custom MCP (Model Context Protocol) interface. The MCP interface constrains the model's reasoning using a Language Server Protocol (for syntax guidance) and a vector database of CodeQL queries and documentation (for semantic guidance). By combining these capabilities, FineNib avoids common pitfalls of naive LLM-based approaches, such as producing ill-formed queries, hallucinating deprecated constructs, or missing subtle vulnerability patterns, and instead produces queries that are both syntactically correct and semantically precise.

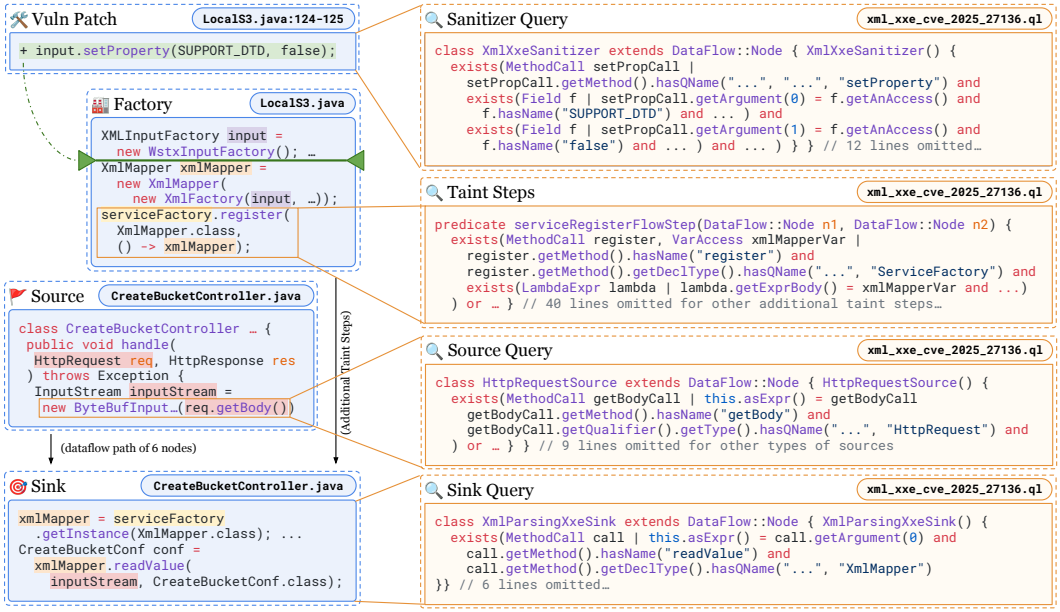
We evaluate FineNib on CWE-Bench-Java (Li et al., 2025b), which comprises 176 CVEs across 111 Java projects. These CVEs span 42 different Common Weakness Enumeration (CWE) categories and the projects range in size from 0.01 to 1.5 MLOC. To account for model training cut-offs, we include 65 CVEs reported during 2025 and target a recent CodeQL version 2.22.2 (July 2025). Using the Claude Code agent framework, FineNib achieves query compilation and success rates of 100% and 53.4%, compared to 19% and 0% for our best agentic baseline, Gemini CLI. Further, our generated queries have an F1 score of 0.7 for detecting true positive vulnerabilities, compared to 0.048 for IRIS (Li et al., 2025b), a recent LLM-assisted static analyzer, and 0.073 for CodeQL.

We summarize our main contributions:

- **Agentic Framework for CVE-to-Query Synthesis.** We present FineNib, an agentic framework that translates CVE descriptions into executable CodeQL queries, bridging the gap between vulnerability reports and static analysis. FineNib introduces a novel integration of execution-guided synthesis, semantic retrieval, and structured reasoning for vulnerability query generation.
- **Evaluation on Real-World Repositories and CVEs.** We evaluate FineNib on 176 CVEs in Java projects, covering 42 vulnerability types (CWEs) from CWE-Bench-Java. Each project involves complex inter-procedural vulnerabilities spanning multiple files. We show how FineNib can successfully identify sources, sinks, sanitizers, and taint propagation steps, and refine queries to ensure they raise alarms on vulnerable versions while remaining silent on patched versions.
- **Comparison with Baselines.** We compare FineNib against state-of-the-art agent frameworks and show that FineNib achieves substantially higher compilation, success, and F1 scores. We also compare FineNib's synthesized queries with state-of-the-art static analysis frameworks and show that our queries are more precise and have higher recall.

2 Illustrative Example

We illustrate the challenges of vulnerability query synthesis using CVE-2025-27136, an XML External Entity Injection (XXE) bug found in the repository `Robothy/local-s3`. Figure 2 depicts the vulnerability snippets, the patch, and the synthesized CodeQL query generated by FineNib.



(a) The vulnerable dataflow snippets and the patch, which adds configuration to `XMLInputFactory`.

(b) Snippets of the synthesized vulnerability query by FineNib capturing the patterns of dataflow source, sink, taint steps, and the sanitizer indicated by the vulnerability patch.

CodeQL Query — xml_xxe_cve_2025_27136.ql

```
module HttpXxeFlow = TaintTracking::Global<HttpXxeFlowConfig>; // Config: HttpRequestSource, XmlParsingXxeSink, …
import HttpXxeFlow::PathGraph

from HttpXxeFlow::PathNode source, HttpXxeFlow::PathNode sink
where HttpXxeFlow::flowPath(source, sink)
select sink.getNode(), source, sink, "HTTP request data flows to XML parser…", source.getNode(), "HTTP request entry point"
```

(c) The synthesized CodeQL path query that ties everything together.

Figure 2: Illustration of vulnerability CVE-2025-27136 in repository `Robothy/local-s3` which exhibits an XML External Entity Injection weakness (CWE-611). When the `XmlMapper` is not configured to disable *Document Type Definition* (DTD), the function `readValue` may declare additional entities, allowing hackers to inject malicious behavior.

Vulnerability context. The vulnerability arises when the `XmlMapper` object is used to parse user-provided XML data (Figure 2a). In the vulnerable code, `XmlMapper.readValue` is called on the HTTP request body without disabling support for *Document Type Definitions* (DTDs). As a result, an attacker can inject malicious external entity declarations into the input stream, enabling server-side request forgery (SSRF) attacks, allowing for access to resources that should not be accessible from external networks, effectively leaking sensitive information. The patch mitigates the issue by configuring the underlying `XMLInputFactory` with the property `SUPPORT_DTD=false`.

Synthesizing the query. The CodeQL query that can effectively capture the vulnerability pattern needs to incorporate 1) sources such as `HttpRequest.getBody` calls where untrusted malicious information enters the program, 2) sinks such as invocations of `XmlMapper.readValue`, where the XXE vulnerability is manifested, 3) additional taint steps related to how the `XmlMapper` is constructed and configured, involving non-trivial interprocedural flows spanning multiple files, and 4) sanitizers such as calls to `setProperty(SUPPORT_DTD, false)`, so that we know that no alarm should be reported after the vulnerability has been fixed.

In general, the synthesized query must connect all these components to be able to detect the bug in the vulnerable program, while not reporting the same alarm after the vulnerability has been fixed. Figure 2b shows all the components of the CodeQL query (simplified), capturing their individual syntactic patterns. Lastly, Figure 2c connects all these components into a coherent path query by using CodeQL's `TaintTracking::Global<.>::PathGraph` and the SQL-like `from-where-select` query, which returns the exact path from source to sink.

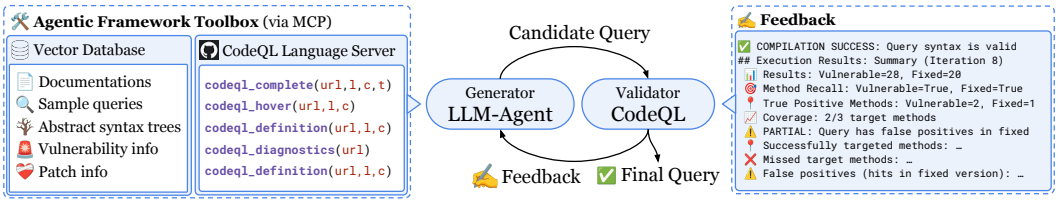


Figure 3: Overall pipeline of FineNib's iterative synthesis loop between an agentic query generator and a CodeQL-based validator. The generator uses a vector database and our CodeQL Language Server as tools while the validator produces compilation, execution, and coverage feedback.

Challenges and solutions. Vulnerability query synthesis must overcome several tightly-coupled challenges. We hereby state the challenges and explain how FineNib addresses them.

- *Rich expressiveness and fragility of syntactic patterns.* CodeQL is powerful but syntactically intricate: small mistakes in predicate names, qualifiers, or AST navigation often produce syntactically valid yet semantically useless queries. FineNib mitigates this fragility through its Language Server Protocol (LSP) interface for syntax guidance and RAG database for semantic retrieval of existing CodeQL queries and documentation. These structured interactions guide predicate selection and AST navigation during synthesis, reducing off-by-name and version-mismatch errors.
- *Inter-procedural taint propagation across a large codebase.* Sources, sinks, and sanitizers typically live in different modules or files and are connected by nontrivial inter-procedural flows (lambdas, factory patterns, etc.). While CodeQL provides robust inter-procedural analysis for many common patterns, gaps in dataflow still require bridging via additional taint propagation steps. Through its custom MCP interface, FineNib performs structured reasoning to discover candidate program points, synthesize custom taint-step predicates (e.g., service registration), and compose them into a CodeQL path query that tracks data across file and component boundaries.
- *Semantic precision: alarm on the vulnerable version, silence on the patched version.* A useful vulnerability query must not only parse correctly but also be discriminative. FineNib enforces this semantic requirement directly during synthesis. Via an iterative refinement loop, the successful criteria states that in the fixed program, there should be no alarm being raised about the vulnerability. This incentivizes the agent to synthesize sanitizer predicates (e.g., the `setProperty` call) and use them to constrain the path query so that sanitizer presence suppresses the alarm. The resulting query thus captures the exact behavior difference, producing alarms on the vulnerable snapshot and not on the patched snapshot.

Together, these capabilities let FineNib synthesize a semantically precise CodeQL query that can be reused for regression testing, variant analysis, or patch validation. We now elaborate on the detailed design and implementation of FineNib.

3 FineNib

At a high level, FineNib operates inside a repository-aware iterative refinement loop (Figure 3). In each iteration, the agent proposes a candidate CodeQL query, a CodeQL-based validator executes and scores it on both the vulnerable and patched versions of the repository, and the agent uses the validation feedback to propose targeted repairs. The loop terminates successfully when the validator accepts a query, or fails after a fixed iteration budget. In this section, we elaborate the major design components that make the loop effective.

3.1 Problem Statement

The task of vulnerability detection is generally framed as a taint analysis task, where the goal of a *query* is to find dataflow paths from a *source* (e.g., an API endpoint accepting user input) to a *sink* (e.g., a database write) that lack proper *sanitization* (e.g., filtering malicious data).

We formalize the *Vulnerability Query Synthesis* problem as follows. Assume as input a vulnerable project version P_{vuln}, its fixed version P_{fixed}, and a textual CVE description (commonly available in open vulnerability reports). Let us assume we have inter-procedural dataflow program graphs for each code version: $G_{\text{vuln}} = (V_{\text{vuln}}, E_{\text{vuln}})$ and $G_{\text{fixed}} = (V_{\text{fixed}}, E_{\text{fixed}})$. Let ΔP denote the source-level patch between P_{vuln} and P_{fixed}. We represent the patch in the dataflow-graph domain

as a patch subgraph $\Delta G = (\Delta V, \Delta E)$, where ΔV is the set of graph nodes that correspond to the modified program snippets.

A *vulnerability path query* Q evaluated on a graph G returns a set of dataflow paths, denoted as $\Pi = [\![Q]\!](G)$. We write each path $\pi \in \Pi$ as $\pi = \langle v_1, \ldots, v_k \rangle$, where each $v_i \in V$ is a node in the dataflow graph G. Consecutive nodes (v_i, v_{i+1}) should be either connected by an existing edge in E, or an *additional taint step* specified in the query Q, to compensate for missing edges via dataflow graph construction. Specifically, we call v_1 the *source* of path π and v_k the *sink* of π.

Synthesis task. We aim to synthesize a query Q from the vulnerability report satisfying the following requirements:

1. **Well-formedness.** Q is syntactically valid (based on the latest CodeQL syntax) and can be executed on the target CodeQL infrastructure (e.g., dataflow graphs) without runtime errors.
2. **Vulnerability detection.** Q generates at least one path π in the vulnerable version that traverses the patched region:
$$\exists \pi \in [\![Q]\!](G_{\text{vuln}}) \quad \text{such that} \quad \pi \cap \Delta V \neq \emptyset.$$
3. **Fix discrimination.** Q does not report the vulnerability in the fixed version. Concretely, no path reported on the fixed version should traverse the patched locations:
$$\forall \pi \in [\![Q]\!](G_{\text{fixed}}), \quad \text{we have} \quad \pi \cap \Delta V = \emptyset.$$

In other words, the synthesized query must be executable, must witness the vulnerability in the vulnerable version via a path that uses code touched by the fix, and must not attribute the same (patched) behavior in the fixed version. When only the well-formedness condition is satisfied, we say that the query Q is valid (denoted as $\texttt{valid}(Q)$); when all the conditions are satisfied, the query Q is successful (denoted as $\texttt{success}(Q; P_{\text{vuln}}, P_{\text{fixed}})$). Note that these criteria may admit potentially false positive paths in both versions. It might be possible to consider additional constraints regarding precision, but it might further complicate synthesis. In practice, we find most queries synthesized by FineNib already have high precision.

3.2 Design of FineNib

Concretely, FineNib proceeds in an iterative refinement loop indexed by $i = [0, 1, \ldots]$. Via prompting, the LLM agent-based synthesizer first proposes an initial candidate query Q_0. For each iteration i, the validator evaluates Q_i and produces a feedback report. We consider synthesis successful at iteration i iff $\texttt{success}(Q_i; P_{\text{vuln}}, P_{\text{fixed}})$ holds; in that case the loop terminates and Q_i is returned. Otherwise, the synthesizer analyzes the feedback and the previous candidate Q_i, and produces the next query candidate Q_{i+1}. The loop stops successfully when $\texttt{success}(\cdot)$ is achieved or fails once i reaches the pre-configured limit N (in our implementation $N = 10$). The remainder of the design focuses on two aspects: 1) how the agentic synthesizer performs synthesis, and 2) how the validator generates and communicates feedback. We elaborate on both below.

Agentic synthesizer. In each iteration i, the LLM-based agentic synthesizer runs an inner *conversation loop* of up to M turns. In each turn, the agent either performs internal reasoning or issues a tool call by emitting a JSON-formatted action. When a tool call succeeds, the tool returns a JSON-formatted response that is appended to the conversation history. Conversation histories are kept local to the current refinement iteration (i.e., not carried over between iterations) to keep context compact and relevant. In practice, we set $M = 50$, i.e., the agent may interact with tools up to 50 times before generating a candidate query for validation.

Two design choices are critical for the effectiveness of this loop: 1) the *initial prompt* that initializes and constrains the agent's behavior, and 2) the *toolbox* of callable tools, each exposed by a custom Model Context Protocol (MCP) server. We refer to the combined problem of designing these items as *Context Engineering* (discussed in Section 3.3).

CodeQL-Based Validator. The validator compiles and executes each candidate query against the vulnerable and fixed versions and returns a concise, structured feedback report that is used to drive refinement (Figure 3). The report contains: (i) CodeQL compilation results, (ii) execution counts (matches on vulnerable and fixed graphs), (iii) recall and coverage statistics, (iv) concrete counterexample traces and hit locations, and (v) a prioritized set of next-step recommendations (e.g., add qualifiers, synthesize sanitizer checks, or expand taint steps) that are programmatically generated via a template.

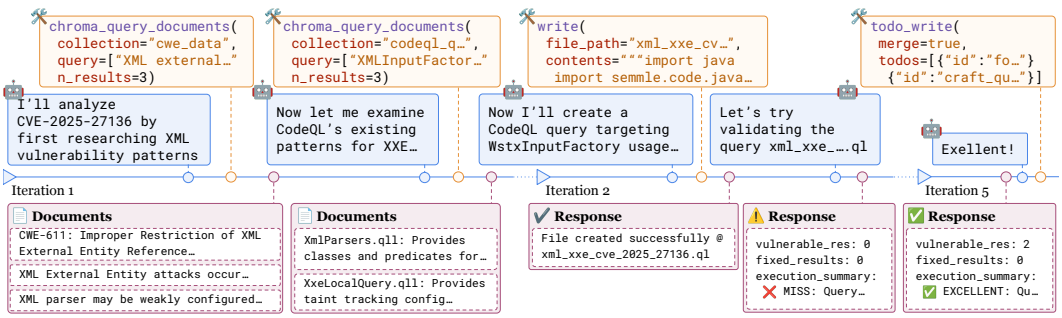


Figure 4: Illustration of example traces of conversation during the synthesis of the query in the motivating example (Figure 2). LLM-agent may think, invoke tools that are available in the toolbox, and receive responses from the MCP servers.

3.3 Context Engineering for Agentic Synthesizer

The primary goal of context engineering is to expose the LLM-based agent to the most *precise* amount of information: enough for the agent to make progress, but not so much that the LLM is confused or the cost explodes. As illustrated in Figure 3, FineNib relies on two primary MCP servers to provide demand-driven, structured information to the agent: a retrieval-augmented vector database and a CodeQL Language Server interface. We show example traces of conversation loop in Figure 4 and describe the available tools below.

Initial prompt. Each refinement iteration begins with an *initial prompt* that kicks-off the agentic conversation loop. The initial prompt in the first iteration contains a query skeleton for reference (See § A.1 for an example). In subsequent iterations, the prompt contains a summary of the synthesis goal and constraints, the previous candidate query Q_{i-1}, and the validator feedback report. Concretely, the initial prompt emphasizes: (i) the success predicate (see `Success`$(\cdot)$), (ii) concrete counterexamples from previous feedback, and (iii) an explicit list of callable tools and their purpose.

Vector database. We use a retrieval-augmented vector database (ChromaDB MCP server in our implementation) to store large reference corpora without polluting the LLM prompt. The database is pre-populated with (i) vulnerability analysis notes and diffs, (ii) Common Weaknesses Enumeration (CWE) definitions, (iii) same-version CodeQL API documentation, (iv) curated CodeQL sample queries, and (v) small abstract syntax tree (AST) snippets extracted from the target repository. During a conversation loop, the agent issues compact retrieval queries (e.g., to fetch example CodeQL queries related to the CWE) and receives ranked documents or snippets on demand.

In practice, we may populate our RAG database with tens of thousands of documents. Even with this large corpus, we observe that the LLM-agent reliably retrieves exactly the kinds of artifacts it needs: CodeQL sample queries that inspire overall query structure, small AST snippets that suggest the precise syntactic navigation, and vulnerability writeups or diff excerpts that help discriminate buggy from patched behavior. These demand-driven lookups let the agent gather high-quality information without loading the main prompt with large reference corpora.

CodeQL language server. We expose the CodeQL Language Server (Github, d) through a MCP server that the agent can call for precise syntax-aware guidance. Importantly, we developed our own CodeQL Language Server client and MCP server that ensures syntactic validity (especially for the given CodeQL version) during query generation. The LLM agent's MCP client makes the tool call which is received by the CodeQL MCP server. The MCP server forwards tool calls, such as `complete(file, loc, char)`, `diagnostics(file)`, and `definition(file, loc, char)`, to the underlying CodeQL process and returns JSON-serializable responses. Tools such as `completion` help the agent fill query templates and discover correct API or AST names, while `diagnostics` reveal compile or linter errors (e.g., unknown predicate names) that guide mutation. Appendix B shows the full specification and example request and response schemas.

3.4 Discussion: Alternative Designs

We discuss several alternative designs that we considered but found ineffective in practice. Allowing the agent unrestricted access to compile-and-run CodeQL via MCP led to severe performance degra-

Table 1: FineNib Query Success by CWE Type

CWE Type	Total CVEs	# Success	Success (%)	Avg Precision
CWE-022 (Path Traversal)	48	31	64.6	0.75
CWE-079 (Cross-Site Scripting)	36	18	50.0	0.621
CWE-094 (Code Injection)	20	12	60.0	0.606
CWE-078 (OS Command Injection)	12	7	58.3	0.628
CWE-502 (Deserialization)	6	4	66.7	0.853
CWE-611 (XXE)	5	3	60.0	0.657
Other CWEs ($\leq$4 CVEs)	49	19	38.8	0.504
Total	**176**	**94**	**53.4**	**0.631**

dation: compilation and full execution are expensive operations that the LLM soon overused, so we instead expose only lightweight diagnostics during the conversation and defer full compile-and-run to the end of each iteration. Permitting free online search for vulnerability patterns or snippets similarly proved problematic. It is both costly and easy for the agent to rely on web lookups, which quickly pollutes the working context and degrades synthesis quality. Equipping the agent with an extensive set of heterogeneous tools led to confusion and poor tool-selection behavior; in contrast, a small, well-scoped toolbox yields more reliable actions. Finally, retaining full conversation histories across refinement iterations induced context rot and ballooning prompt sizes, so we keep histories local to each iteration. Overall, our current design is a pragmatic trade-off that balances cost, responsiveness, and synthesis effectiveness.

4 Evaluation

We answer the following research questions in this work:

- **RQ 1**: For how many CVEs, can FineNib generate queries successfully?
- **RQ 2**: How useful is each component of FineNib?
- **RQ 3:** How does the choice of base agent framework affect FineNib's effectiveness?

4.1 Experimental Setup

We develop FineNib on top of the Claude Code framework (Anthropic, 2025) and use Claude Sonnet 4 for all our experiments. For agent baselines, we select Codex with GPT-5 (minimal reasoning) and Gemini CLI with Gemini 2.5 Flash. For each CVE and agent baseline, we use a maximum of 10 iterations ($N = 10$). For static analysis baselines, we select IRIS Li et al. (2025b) and CodeQL (version 2.22.2) query suites. Experiments were run on machines with the following specifications: an Intel Xeon Gold 6248 2.50GHz CPU, four GeForce RTX 2080 Ti GPUs, and 750GB RAM.

Dataset. We used CWE-Bench-Java (Li et al., 2025b) and its latest update, which added new CVEs from 2025. We were able to successfully build and use 111 (out of 120) Java CVEs evaluated in IRIS (Li et al., 2025b), and 65 (out of 91) 2025 CVEs. Each sample in CWE-Bench-Java comes with the CVE metadata and fix commit information associated with the bug.

4.2 Evaluation Metrics

Besides Valid(Q) and Success($Q; P_{\text{vuln}}, P_{\text{fixed}}$), we use the following terms and metrics when evaluating FineNib and baselines on the problem of vulnerability query synthesis:

$$\text{Rec}(Q) = \mathbb{1}[\exists \pi \in [\![Q]\!](G_{\text{vuln}}), \pi \cap \Delta V \neq \emptyset], \quad \text{Prec}(Q) = \frac{|\{\pi \in [\![Q]\!](G_{\text{vuln}}) \mid \pi \cap \Delta V \neq \emptyset\}|}{|[\![Q]\!](G_{\text{vuln}})|},$$

$$\text{F1}(Q) = 2 \cdot \frac{\text{Prec}(Q) \cdot \text{Rec}(Q)}{\text{Prec}(Q) + \text{Rec}(Q)}.$$

4.3 RQ1: FineNib Effectiveness

FineNib vs. state-of-the-art QL. Table 1 shows FineNib's overall query synthesis success rate by CWE. Table 2 shows the notable increase in precision between CodeQL, IRIS, and FineNib. We have successfully synthesized 53.4% of the CVEs. For half the queries FineNib correctly synthesized CodeQL, detect the CVE, did not find false positives when executed on the fixed version of the CVE's repository. The lack of true positive recall is why CodeQL and IRIS have significantly

Table 2: Recall Performance Comparison Across Methods (Shared CVEs: 130)

Method	Recall Rate (%)	Avg Precision	Avg F1 Score
CodeQL	20.0	0.055	0.073
IRIS	35.4	0.031	0.048
FineNib	80.0	0.672	0.700

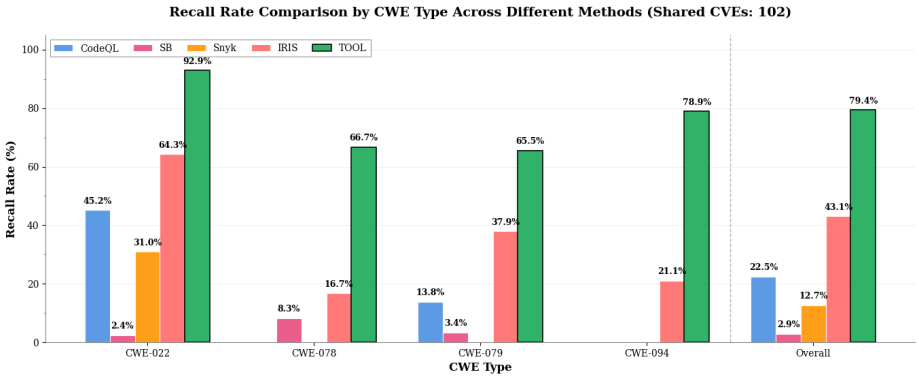


Figure 5: Recall Rate Comparison by CWE Type Across Different Methods (102 CVEs).

lower precision. CodeQL's queries are broad, categorized by CWE queries. IRIS generates all of the predicates for potential sources and sinks wit CodeQL, and does not generate sanitizer or taint step predicates.

Impact of training cut-off. We also want to take note that Claude Sonnet 4's training cut-off is March 2025. Table 3 shows that FineNib performs consistently regardless of CVEs before or after the cut-off period. The CodeQL version, 2.22.2, was released in July 2025. New versions of CodeQL often include analysis improvements and new QL packs (Github, a).

Table 3: Tool Performance Before vs After Training Cutoff

CVE Period	Total CVEs	# Recall	Success (%)	Avg Precision	Avg F1 Score
Pre-2025 (2011-2024)	111	64	57.7	0.676	0.702
2025+ (Post-cutoff)	65	30	46.2	0.555	0.583
Overall	**176**	**94**	**53.4**	**0.631**	**0.658**

4.4 RQ2: Ablation Studies

For ablations, we chose 20 CVEs and ran FineNib with one of the FineNib components removed (Table 4). The ablation with no tools refers to only running Claude Code with the iterative feedback system. The high recall rate when removing access to the AST cache while lowered recall rates without the LSP server or documentation access show that the LSP and documentation lookup impact the synthesis performance more. We also include FineNib's performance on the same set of CVEs, and point out its significantly higher query success rate and precision score. Claude Code without tools scored a high recall rate, yet failed to synthesize queries without false positives when executed on the fixed version.

4.5 RQ3: State of the Art Agent Comparison

FineNib can be transferred to other coding agents by MCP configuration changes, since its tools are MCP servers. Changing agents involves using a different CLI command to start the coding agent, which was a minor adjustment for using FineNib. We used Gemini CLI with Gemini 2.5 Flash, and Codex with GPT-5 minimal. We evaluated their performance on 20 CVEs, and although there were no successful queries, we achieved an increase in compilation success for both agents compared to using the agents without FineNib in Table 5.

Table 4: Ablation Study (out of 20 CVEs)

Variant	% Successful	Recall Rate	Avg Precision	Avg F1 Score
FineNib	**55%**	**80%**	**0.67**	**0.69**
w/o LSP	25% (−30%)	55% (−25%)	0.32	0.36
w/o Doc/Ref	20% (−35%)	55% (−25%)	0.32	0.36
w/o AST	25% (−30%)	80% (±0%)	0.41	0.47
w/o Tools	10% (−45%)	55% (−25%)	0.33	0.36

Table 5: LLM-agent baselines' compilation rate on 20 CVEs.

Agent Baselines	Configuration	Compilation Rate (%)
Gemini CLI	Gemini 2.5 Flash (with tools)	24
Gemini CLI	Gemini 2.5 Flash (without tools)	19
Codex	GPT 5 (with tools)	24
Codex	GPT 5 (without tools)	0

5 Related Work

LLMs and vulnerability detection. LLMs have been used extensively for vulnerability detection and repair using techniques such as fine-tuning and prompt engineering (Zhou et al., 2024). LLMs have also been combined with existing program analysis tools for vulnerability detection. The combination of LLMs can be used from vulnerability analysis like IRIS's (Li et al., 2025b) source and sink identification, however IRIS depends on a limited set of CWE templates derived from CodeQL's CWE queries. IRIS also only the LLM for identifying sources and sinks. KNighter synthesizes CSA checkers given a fix commit of a C repository(Yang et al., 2025), however the checkers are written in C which has more available training data. MocQ's uses an LLM to derive a subset DSL of CodeQL and Joern, and then provides a feedback loop to the LLM though prompting via API calls is used rather than an agent with tools and MocQ uses significantly higher iterations, with a max threshold of 1,000 iterations per vulnerability experiment. (Li et al., 2025a).

LLM agents and tool usage. SWE-agent pioneered the idea of autonomous LLM agents using tools for software engineering tasks Yang et al. (2024). LSPAI Go et al. (2025), an IDE plugin, uses LSP servers to guide LLM-generated unit tests. Hazel, a live program sketching environment, uses a language server (Blinn et al., 2024) to assist code completions synthesized by LLMs. The Hazel Language Server provides the typing context of a program hole to be filled.

Low resource LLM code generation. SPEAC uses ASTs combined with constraint solving to repair LLM-generated code for low resource programming languages (Mora et al., 2024). SPEAC converts a buggy program into an AST and uses a solver to find the minimum set of AST nodes to replace, to satisfy language constraints. MultiPL-T generates datasets for low resource languages by translating high resource language code to the target language and validates translations with LLM generated unit tests (Cassano et al., 2024).

6 Conclusion and Limitations

We present FineNib, an agentic framework for synthesizing syntactically correct and precise CodeQL queries given known vulnerability patterns. We will also open source our CodeQL LSP MCP server and FineNib. In future work, we plan to explore efficient ways to synthesize, and to combine our synthesized queries with dynamic analysis tools.

Limitations. We omit CVEs where the vulnerability involves non-Java code such as configuration files or other languages. FineNib can be used with exploit generation to find vulnerabilities that are realized during dynamic execution. For supporting other languages that can be queried by CodeQL, the vector database can be filled with references, documentation, and example queries in other CodeQL supported languages. We also want to note that Claude Sonnet 4's official training cut-off is March 2025, however the 2025 CVEs evaluated were reported between January to August 2025.

REFERENCES

Anthropic. Claude code: Agentic coding assistant. https://github.com/anthropics/claude-code, 2025. Version 1.0. Released February 2025. Accessed September 2025.

Andrew Blinn, Xiang Li, June Hyung Kim, and Cyrus Omar. Statically Contextualizing Large Language Models with Typed Holes. *Artifact for Statically Contextualizing Large Language Models with Typed Holes*, 8(OOPSLA2):288:468–288:498, October 2024. doi: 10.1145/3689728. URL https://dl.acm.org/doi/10.1145/3689728.

Federico Cassano, John Gouwar, Francesca Lucchetti, Claire Schlesinger, Anders Freeman, Carolyn Jane Anderson, Molly Q Feldman, Michael Greenberg, Abhinav Jangda, and Arjun Guha. Knowledge transfer from high-resource to low-resource programming languages for code llms, 2024. URL https://arxiv.org/abs/2308.09895.

CVE. Common vulnerabilities and exposures (cve). https://www.cve.org/about/Metrics, 2025.

Github. CodeQL 2.23.0 (2025-09-04) — CodeQL, a. URL https://codeql.github.com/docs/codeql-overview/codeql-changelog/codeql-cli-2.23.0/.

Github. codeql/java/ql/lib/semmle/code/java/security at main · github/codeql, b. URL https://github.com/github/codeql/tree/main/java/ql/lib/semmle/code/java/security.

Github. codeql/java/ql/src/Security/CWE at main · github/codeql, c.

Github. execute language-server, d. URL https://docs-internal.github.com/en/code-security/codeql-cli/codeql-cli-manual/execute-language-server.

Github. CodeQL. https://codeql.github.com/, 2025.

GitHub. Github security advisory database. https://github.com/advisories, 2025.

Gwihwan Go, Chijin Zhou, Quan Zhang, Yu Jiang, and Zhao Wei. LSPAI: An IDE Plugin for LLM-Powered Multi-Language Unit Test Generation with Language Server Protocol. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, pp. 144–149, Clarion Hotel Trondheim Trondheim Norway, June 2025. ACM. ISBN 979-8-4007-1276-0. doi: 10.1145/3696630.3728540. URL https://dl.acm.org/doi/10.1145/3696630.3728540.

Penghui Li, Songchen Yao, Josef Sarfati Korich, Changhua Luo, Jianjia Yu, Yinzhi Cao, and Junfeng Yang. Automated static vulnerability detection via a holistic neuro-symbolic approach, 2025a. URL https://arxiv.org/abs/2504.16057.

Z. Li, S. Dutta, and M. Naik. IRIS: LLM-Assisted Static Analysis for Detecting Security Vulnerabilities, Apr. 2025b. arXiv:2405.17238.

Meta. Infer. https://fbinfer.com/, 2025.

MITRE. Common vulnerabilities and exposures (cve). https://www.cve.org/, 2025.

Federico Mora, Justin Wong, Haley Lepe, Sahil Bhatia, Karim Elmaaroufi, George Varghese, Joseph E Gonzalez, Elizabeth Polgreen, and Sanjit A Seshia. Synthetic programming elicitation for text-to-code in very low-resource programming and formal languages. *Advances in Neural Information Processing Systems*, 37:105151–105170, 2024.

NIST. National vulnerability database (nvd). https://nvd.nist.gov/, 2025.

Semgrep. Semgrep. the semgrep platform. https://semgrep.dev/, 2023.

C. Yang, Z. Zhao, Z. Xie, H. Li, and L. Zhang. KNighter: Transforming Static Analysis with LLM-Synthesized Checkers, Apr. 2025. arXiv:2503.09002.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering, 2024. URL https://arxiv.org/abs/2405.15793.

Xin Zhou, Sicong Cao, Xiaobing Sun, and David Lo. Large language model for vulnerability detection and repair: Literature review and the road ahead, 2024. URL https://arxiv.org/abs/2404.02525.

A CodeQL Queries

A.1 CodeQL Query Structure Template

The template below is given to the LLM agent at the start of the iterative query synthesis task. The prompt instructs the LLM to use the AST nodes, along with the CodeQL LSP and CodeQL references in the vector database, to fill in this template. The prompt also takes note to find similar queries related to the given CVE's vulnerability.

```
/**
 * @name [Vulnerability Name based on analysis]
 * @description [Description derived from the vulnerability pattern]
 * @problem.severity error
 * @security-severity [score based on severity]
 * @precision high
 * @tags security
 * @kind path-problem
 * @id [unique-id]
 */
import java
import semmle.code.java.frameworks.Networking
import semmle.code.java.dataflow.DataFlow
import semmle.code.java.dataflow.FlowSources
import semmle.code.java.dataflow.TaintTracking
private import semmle.code.java.dataflow.ExternalFlow

class Source extends DataFlow::Node {
  Source() {
    exists([AST node type from analysis] |
      /* Fill based on AST patterns for sources identified in Phase 1 & 2 */
      and this.asExpr() = [appropriate mapping]
    )
  }
}

class Sink extends DataFlow::Node {
  Sink() {
    exists([AST node type] |
      /* Fill based on AST patterns for sinks */
      and this.asExpr() = [appropriate mapping]
    ) or
    exists([Alternative AST pattern] |
      /* Additional sink patterns from analysis */
      and [appropriate condition]
    )
  }
}

class Sanitizer extends DataFlow::Node {
  Sanitizer() {
    exists([AST node type for sanitizers] |
```

```
        /* Fill based on sanitizer patterns from Phase 1 & 2 */
    )
  }
}

module MyPathConfig implements DataFlow::ConfigSig {
  predicate isSource(DataFlow::Node source) {
    source instanceof Source
  }

  predicate isSink(DataFlow::Node sink) {
    sink instanceof Sink
  }

  predicate isBarrier(DataFlow::Node sanitizer) {
    sanitizer instanceof Sanitizer
  }

  predicate isAdditionalFlowStep(DataFlow::Node n1, DataFlow::Node n2) {{
    /* Fill based on additional taint steps from analysis */
  }}
}}

module MyPathFlow = TaintTracking::Global<MyPathConfig>;
import MyPathFlow::PathGraph

from
  MyPathFlow::PathNode source,
  MyPathFlow::PathNode sink
where
  MyPathFlow::flowPath(source, sink)
select
  sink.getNode(),
  source,
  sink,
  "[Alert message based on vulnerability]",
  source.getNode(),
  "[source description]"
```

A.2 Iterative Feedback Example

The following is an excerpt from the prompt used in a new context window that instructs the agent to improve the last iteration's query. The feedback is derived from automatically running synthesized queries against the fixed and vulnerable versions of the CVE's repository, and comparing the query's results against the changed methods in the CVE fix commit diff. The feedback also includes any successful method hits, false positive method hits, and which fixed methods have not been detected by the query.

```
## Objective
Refine the CodeQL query based on previous iteration feedback to
 ↪  improve vulnerability detection.

## Previous Iteration Feedback
## Iteration 2 Results

## Previous Query (Iteration 2)
```ql
/**
 * @name AntiSamy XSS Bypass in Style Tags
```
```

```
 * @description Detects potential XSS vulnerabilities where AntiSamy HTML sanitization fails to properly filter malicious style tags with event handlers
 * @problem.severity error
 * @security-severity 6.1
 * @precision high
 * @tags security
 * @kind path-problem
 * @id java/antisamy-style-xss-bypass
 */
import java
import semmle.code.java.dataflow.DataFlow
import semmle.code.java.dataflow.TaintTracking

class MaliciousHtmlInput extends DataFlow::Node {
  MaliciousHtmlInput() {
    exists(StringLiteral sl |
      this.asExpr() = sl and
      (
        sl.getValue().matches("%<style%onload%>%") or
        sl.getValue().matches("%<style%onclick%>%") or
        sl.getValue().matches("%<style%onerror%>%") or
        sl.getValue().matches("%<style%onmouseover%>%") or
        sl.getValue().matches("%style%alert%") or
        sl.getValue().matches("%<style%javascript%") or
        sl.getValue().matches("%<style%expression%")
      )
    ) or
    exists(Parameter p |
      this.asParameter() = p and
      (
        p.getName().toLowerCase().matches("%html%") or
        p.getName().toLowerCase().matches("%input%") or
        p.getName().toLowerCase().matches("%content%")
      )
    )
  }
}

class AntiSamyProcessing extends DataFlow::Node {
  AntiSamyProcessing() {
    exists(MethodCall mc |
      this.asExpr() = mc and
      mc.getMethod().hasName("scan") and
      mc.getMethod().getDeclaringType()(RefType)
      .hasQualifiedName("org.owasp.validator.html", "AntiSamy")
    ) or
    exists(MethodCall mc |
      this.asExpr() = mc and
      mc.getMethod().hasName("getCleanHTML") and
      mc.getQualifier().getType().(RefType)
      .hasQualifiedName("org.owasp.validator.html",
       "CleanResults")
    )
  }
}

class PotentialXssOutput extends DataFlow::Node {
  PotentialXssOutput() {
```

```
    exists(MethodCall mc |
      this.asExpr() = mc and
      (
        mc.getMethod().hasName("contains") and
        exists(StringLiteral sl |
          mc.getAnArgument() = sl and
          (
            sl.getValue().matches("%alert%") or
            sl.getValue().matches("%script%") or
            sl.getValue().matches("%javascript%") or
            sl.getValue().matches("%onload%") or
            sl.getValue().matches("%onclick%")
          )
        )
      )
    ) or
    exists(MethodCall mc |
      this.asExpr() = mc and
      mc.getMethod().hasName("getCleanHTML") and
      mc.getQualifier().getType().(RefType)
      .hasQualifiedName("org.owasp.validator.html",
       ↪  "CleanResults")
    )
  }
}

class ImproperSanitizationBarrier extends DataFlow::Node {
  ImproperSanitizationBarrier() {
    exists(MethodCall mc |
      this.asExpr() = mc and
      mc.getMethod().hasName("assertTrue") and
      exists(LogNotExpr lne |
        mc.getAnArgument() = lne and
        lne.getExpr().(MethodCall).getMethod().hasName("contains")
      )
    )
  }
}

module AntiSamyXssConfig implements DataFlow::ConfigSig {
  predicate isSource(DataFlow::Node source) {
    source instanceof MaliciousHtmlInput
  }

  predicate isSink(DataFlow::Node sink) {
    sink instanceof PotentialXssOutput
  }

  predicate isBarrier(DataFlow::Node sanitizer) {
    sanitizer instanceof ImproperSanitizationBarrier
  }

  predicate isAdditionalFlowStep(DataFlow::Node n1, DataFlow::Node
   ↪  n2) {
    exists(MethodCall mc |
      n1.asExpr() = mc.getQualifier() and
      n2.asExpr() = mc and
      mc.getMethod().hasName("scan") and
      mc.getMethod().getDeclaringType().(RefType)
```

```
      .hasQualifiedName("org.owasp.validator.html", "AntiSamy")
  ) or
  exists(MethodCall mc |
    n1.asExpr() = mc.getQualifier() and
    n2.asExpr() = mc and
    mc.getMethod().hasName("getCleanHTML")
  )
 }
}

module AntiSamyXssFlow = TaintTracking::Global<AntiSamyXssConfig>;
import AntiSamyXssFlow::PathGraph

from
  AntiSamyXssFlow::PathNode source,
  AntiSamyXssFlow::PathNode sink
where
  AntiSamyXssFlow::flowPath(source, sink)
select
  sink.getNode(),
  source,
  sink,
  "Potential XSS vulnerability: HTML input with malicious style tags may bypass AntiSamy sanitization",
  source.getNode(),
  "malicious HTML input"
```

Compilation Results
COMPILATION SUCCESS: Query syntax is valid

Execution Results
Query Evaluation Summary (Iteration 2)
Results: Vulnerable=8, Fixed=8
Method Recall: Vulnerable=True, Fixed=True
True Positive Methods: Vulnerable=2, Fixed=2
Coverage: 1/1 target methods
PARTIAL: Query hits targets but has **false** positives in fixed version
Method location format is path/to/hit/file.java:[Class of hit method]:[Hit method]
Successfully targeted methods:

-src/main/java/org/owasp/validator/html/scan/MagicSAXFilter.java:MagicSAXFilte
False positives (hits in fixed version):
-
src/main/java/org/owasp/validator/html/scan/MagicSAXFilter.java:MagicSAXFilter

Detailed Evaluation Analysis

Method Coverage: 1/1 target methods
File Coverage: 1/1 target files
Successfully targeted files:
  - MagicSAXFilter.java

Successfully targeted methods:
  - startElement

False positives (hit in fixed version):

```
    - startElement

**Evaluation Summary**:
- Vulnerable DB File Recall: True
- Fixed DB File Recall: True
- Vulnerable DB Method Recall: True
- Fixed DB Method Recall: True
- Total Query Results: Vulnerable=8, Fixed=8
- Code Flow Paths: Vulnerable=8, Fixed=8

## Next Steps
We want vulnerable DB method recall and we don't want fixed DB method recall!
**Priority**: Reduce false positives by adding more specific conditions to avoid hitting the methods listed above.
```

A.3 FINENIB SYNTHESIZED QUERY EXAMPLES

Below are examples of successful synthesized queries. The queries successfully find > 0 true positive method hits on the vulnerable CodeQL database of the CVE's source code, and no false positive method hits on the fixed version's CodeQL database. For reference queries to compare with, CWE queries can be found on the official CodeQL repository (Github, c;b)

CVE-2025-27136, CWE-611 - Improper Restriction of XML External Entity Reference

```
/**
 * @name XML External Entity vulnerability in WstxInputFactory without secure configuration
 * @description WstxInputFactory used in XmlFactory without disabling DTD support and external entities allows XXE attacks
 * @problem.severity error
 * @security-severity 9.1
 * @precision high
 * @tags security
 * @kind path-problem
 * @id java/wstxinputfactory-xxe
 */

import java
import semmle.code.java.dataflow.DataFlow
import semmle.code.java.dataflow.FlowSources
import semmle.code.java.dataflow.TaintTracking
private import semmle.code.java.dataflow.ExternalFlow

class WstxInputFactoryCreation extends DataFlow::Node {
  WstxInputFactoryCreation() {
    exists(MethodCall mc |
      mc.getMethod().hasQualifiedName("com.ctc.wstx.stax", "WstxInputFactory", "newInstance") or
      mc.getMethod().hasQualifiedName("com.ctc.wstx.stax", "WstxInputFactory", "newFactory")
    |
      this.asExpr() = mc
    ) or
    exists(ClassInstanceExpr cie |

      cie.getConstructedType().hasQualifiedName("com.ctc.wstx.stax", "WstxInputFactory") and
```

```
      this.asExpr() = cie
    ) or
    // Include variable access to WstxInputFactory instances (like "input" parameter)
    exists(Variable v, VarAccess va |
      v.getType().(RefType).hasQualifiedName("com.ctc.wstx.stax", "WstxInputFactory") and
      va.getVariable() = v and
      this.asExpr() = va
    )
  }
}

class UnsafeXmlFactoryUsage extends DataFlow::Node {
  UnsafeXmlFactoryUsage() {
    exists(ClassInstanceExpr xmlFactoryCall |
      // XmlFactory constructor with WstxInputFactory parameter
      xmlFactoryCall.getConstructedType()
      .hasQualifiedName("com.fasterxml.jackson.dataformat.xml", "XmlFactory") and
      xmlFactoryCall.getArgument(0) = this.asExpr()
    ) or
    exists(ClassInstanceExpr xmlMapperCall, ClassInstanceExpr xmlFactoryCall |
      // XmlMapper constructor using XmlFactory with WstxInputFactory
      xmlMapperCall.getConstructedType()
      .hasQualifiedName("com.fasterxml.jackson.dataformat.xml", "XmlMapper") and
      xmlFactoryCall.getConstructedType()
      .hasQualifiedName("com.fasterxml.jackson.dataformat.xml", "XmlFactory") and
      xmlMapperCall.getArgument(0) = xmlFactoryCall and
      xmlFactoryCall.getArgument(0) = this.asExpr()
    )
  }
}

class WstxInputFactorySanitizer extends DataFlow::Node {
  WstxInputFactorySanitizer() {
    exists(MethodCall setPropertyCall, VarAccess factoryVar |

      setPropertyCall.getMethod().hasQualifiedName("javax.xml.stream", "XMLInputFactory", "setProperty") and
      setPropertyCall.getQualifier() = factoryVar and
      (
        // DTD support disabled
        (exists(Field f |
          setPropertyCall.getArgument(0) = f.getAnAccess() and
          f.hasName("SUPPORT_DTD") and

          f.getDeclaringType().hasQualifiedName("javax.xml.stream", "XMLInputFactory")
        ) and
        exists(Field f |
          setPropertyCall.getArgument(1) = f.getAnAccess() and
          f.hasName("FALSE") and
          f.getDeclaringType().hasQualifiedName("java.lang", "Boolean")
```

```
        )) or
        // External entities disabled
        (exists(Field f |
          setPropertyCall.getArgument(0) = f.getAnAccess() and
          f.hasName("IS_SUPPORTING_EXTERNAL_ENTITIES") and
          f.getDeclaringType().hasQualifiedName("javax.xml.stream",
          "XMLInputFactory")
        ) and
        exists(Field f |
          setPropertyCall.getArgument(1) = f.getAnAccess() and
          f.hasName("FALSE") and
          f.getDeclaringType().hasQualifiedName("java.lang",
          "Boolean")
        ))
      ) and
      this.asExpr() = factoryVar
    )
  }
}

module WstxInputFactoryFlowConfig implements DataFlow::ConfigSig {
  predicate isSource(DataFlow::Node source) {
    source instanceof WstxInputFactoryCreation
  }

  predicate isSink(DataFlow::Node sink) {
    sink instanceof UnsafeXmlFactoryUsage
  }

  predicate isBarrier(DataFlow::Node sanitizer) {
    sanitizer instanceof WstxInputFactorySanitizer
  }

  predicate isAdditionalFlowStep(DataFlow::Node n1, DataFlow::Node
  n2) {
    // WstxInputFactory passed to XmlFactory constructor
    exists(ClassInstanceExpr xmlFactoryCall |
      xmlFactoryCall.getConstructedType()
      .hasQualifiedName("com.fasterxml.jackson.dataformat.xml",
      "XmlFactory") and
      xmlFactoryCall.getArgument(0) = n1.asExpr() and
      n2.asExpr() = xmlFactoryCall
    ) or
    // XmlFactory passed to XmlMapper constructor
    exists(ClassInstanceExpr xmlMapperCall |
      xmlMapperCall.getConstructedType()
      .hasQualifiedName("com.fasterxml.jackson.dataformat.xml",
      "XmlMapper") and
      xmlMapperCall.getArgument(0) = n1.asExpr() and
      n2.asExpr() = xmlMapperCall
    )
  }
}

module WstxInputFactoryFlow =
  TaintTracking::Global<WstxInputFactoryFlowConfig>;
import WstxInputFactoryFlow::PathGraph
```

```
from
  WstxInputFactoryFlow::PathNode source,
  WstxInputFactoryFlow::PathNode sink
where
  WstxInputFactoryFlow::flowPath(source, sink)
select
  sink.getNode(),
  source,
  sink,
  "WstxInputFactory used without secure configuration flows to XML
   ↪  parser, allowing XXE attacks",
  source.getNode(),
  "WstxInputFactory usage"
```

CVE-2025-0851, CWE-22 - Path Traversal

```
/**
 * @name Archive path traversal vulnerability (ZipSlip) -
  ↪  CVE-2025-0851
 * @description Archive entries with path traversal sequences can
  ↪  write files outside the intended extraction directory
 * @problem.severity error
 * @security-severity 9.8
 * @precision high
 * @tags security
 * @kind path-problem
 * @id java/archive-path-traversal-cve-2025-0851
 */

import java
import semmle.code.java.dataflow.DataFlow
import semmle.code.java.dataflow.TaintTracking

/**
 * Sources: Archive entry names from ZipEntry.getName() and
  ↪  TarArchiveEntry.getName()
 */
class ArchiveEntryNameSource extends DataFlow::Node {
  ArchiveEntryNameSource() {
    exists(MethodCall mc |
      mc.getMethod().getName() = "getName" and
       ↪  (mc.getMethod().getDeclaringType().hasQualifiedName("java.util.zip",
       ↪  "ZipEntry") or mc.getMethod()
            .getDeclaringType()
             ↪  .hasQualifiedName("org.apache.commons.compress.archivers.tar",
             ↪  "TarArchiveEntry")
      ) and
      this.asExpr() = mc
    )
  }
}

/**
 * Sinks: Path resolution operations that lead to file creation
 */
class PathCreationSink extends DataFlow::Node {
  PathCreationSink() {
    // Arguments to Path.resolve() calls
    exists(MethodCall resolveCall |
```

```
      resolveCall.getMethod().getName() = "resolve" and
      resolveCall.getMethod().getDeclaringType()
      .hasQualifiedName("java.nio.file", "Path") and
      this.asExpr() = resolveCall.getAnArgument()
    )
    or
    // Arguments to file creation operations
    exists(MethodCall fileOp |
      (
        fileOp.getMethod().getName() = "createDirectories" or
        fileOp.getMethod().getName() = "newOutputStream" or
        fileOp.getMethod().getName() = "write" or
        fileOp.getMethod().getName() = "copy"
      ) and
      fileOp.getMethod().getDeclaringType().hasQualifiedName("java.nio.file",
      "Files") and
      this.asExpr() = fileOp.getAnArgument()
    )
  }
}

/**
 * Sanitizers: Proper validation that prevents path traversal
 */
class PathTraversalSanitizer extends DataFlow::Node {
  PathTraversalSanitizer() {
    // The validateArchiveEntry method call that properly
    validates paths
    // This blocks flow after the validation call is made
    exists(MethodCall validateCall |
      validateCall.getMethod().getName() = "validateArchiveEntry"
      and
      (
        // Any variable assigned from validateArchiveEntry call
        result
        exists(Variable v |
          this.asExpr() = v.getAnAccess() and
          exists(AssignExpr assign |
            assign.getDest() = v.getAnAccess() and
            assign.getRhs() = validateCall
          )
        )
        or
        // Variables passed through validateArchiveEntry calls
        this.asExpr() = validateCall.getAnArgument() and
        exists(ExprStmt stmt | stmt.getExpr() = validateCall)
      )
    )
    or
    // Proper ".." validation with exception throwing (complete
    pattern)
    exists(MethodCall containsCall, IfStmt ifStmt, ThrowStmt
    throwStmt |
      containsCall.getMethod().getName() = "contains" and
      containsCall.getAnArgument().(StringLiteral).getValue() =
      ".." and
      ifStmt.getCondition().getAChildExpr*() = containsCall and
      ifStmt.getThen().getAChild*() = throwStmt and
```

```
      this.asExpr() = containsCall.getQualifier()
    )
    or
    // Path normalization combined with startsWith validation
    exists(MethodCall normalizeCall, MethodCall startsWithCall |
      normalizeCall.getMethod().getName() = "normalize" and
      normalizeCall.getMethod().getDeclaringType()
      .hasQualifiedName("java.nio.file", "Path") and
      startsWithCall.getMethod().getName() = "startsWith" and
      startsWithCall.getMethod().getDeclaringType().
      hasQualifiedName("java.nio.file", "Path") and
      DataFlow::localFlow(DataFlow::exprNode(normalizeCall),
        DataFlow::exprNode(startsWithCall.getQualifier())) and
      this.asExpr() = normalizeCall.getQualifier()
    )
  }
}

/**
 * Additional predicate to detect validation barriers at method
 ↪  level
 */
predicate hasValidationCall(Callable method) {
  exists(MethodCall validateCall |
    validateCall.getEnclosingCallable() = method and
    validateCall.getMethod().getName() = "validateArchiveEntry"
  )
}

module PathTraversalConfig implements DataFlow::ConfigSig {
  predicate isSource(DataFlow::Node source) {
    source instanceof ArchiveEntryNameSource
  }

  predicate isSink(DataFlow::Node sink) {
    sink instanceof PathCreationSink
  }

  predicate isBarrier(DataFlow::Node sanitizer) {
    sanitizer instanceof PathTraversalSanitizer
  }

  predicate isBarrierIn(DataFlow::Node node) {
    // Barrier at method entry if method contains
     ↪  validateArchiveEntry call
    node instanceof DataFlow::ParameterNode and
    hasValidationCall(node.getEnclosingCallable())
  }

  predicate isAdditionalFlowStep(DataFlow::Node n1, DataFlow::Node
   ↪  n2) {
    // Flow through variable assignments and declarations
    exists(LocalVariableDeclExpr decl |
      decl.getInit() = n1.asExpr() and
      n2.asExpr() = decl.getVariable().getAnAccess()
    )
    or
    exists(AssignExpr assign |
      assign.getRhs() = n1.asExpr() and
```

```
      n2.asExpr() = assign.getDest()
    )
    or
    // Flow through string manipulation methods that preserve taint
    exists(MethodCall mc |
      mc.getAnArgument() = n1.asExpr() and
      n2.asExpr() = mc and
      (
        mc.getMethod().getName() = "removeLeadingFileSeparator" or
        mc.getMethod().getName() = "trim" or
        mc.getMethod().getName() = "toString" or
        mc.getMethod().getName() = "substring"
      )
    )
    or
    // Flow through Path operations
    exists(MethodCall pathOp |
      pathOp.getAnArgument() = n1.asExpr() and
      n2.asExpr() = pathOp and
      pathOp.getMethod().getName() = "resolve" and
      pathOp.getMethod().getDeclaringType()
      .hasQualifiedName("java.nio.file", "Path")
    )
  }
}

module PathTraversalFlow = TaintTracking::Global<PathTraversalConfig>;

import PathTraversalFlow::PathGraph

from PathTraversalFlow::PathNode source, PathTraversalFlow::PathNode sink
where
  PathTraversalFlow::flowPath(source, sink) and
  // Focus on the specific vulnerable files and methods
  (
    source.getNode().getEnclosingCallable().getDeclaringType()
    .hasName("TarUtils") or
    source.getNode().getEnclosingCallable().getDeclaringType()
    .hasName("ZipUtils")
  ) and
  (
    source.getNode().getEnclosingCallable().getName() = "untar" or
    source.getNode().getEnclosingCallable().getName() = "unzip"
  ) and
  // Only report flows where validation is NOT properly done
  not hasValidationCall(source.getNode().getEnclosingCallable())
select sink.getNode(), source, sink,
  "Archive entry name from $@ flows to file system operation without proper path traversal validation, allowing ZipSlip attack.",
  source.getNode(), "archive entry name"
```

CCVE-2025-27528, CWE-502 - Deserialization of Untrusted Data

```
/**
 * @name MySQL JDBC URL parameter injection vulnerability
```

```
 * @description Detects MySQL JDBC URLs with dangerous bracket parameters that bypass inadequate filtering in vulnerable code
 * @problem.severity error
 * @security-severity 8.8
 * @precision high
 * @tags security
 * @kind path-problem
 * @id java/mysql-jdbc-url-injection
 */

import java
import semmle.code.java.dataflow.DataFlow
import semmle.code.java.dataflow.TaintTracking

class MySQLDangerousBracketUrlSource extends DataFlow::Node {
  MySQLDangerousBracketUrlSource() {
    // String literals with dangerous MySQL parameters in bracket notation
    exists(StringLiteral lit |
      lit.getValue().matches("*mysql*") and
      lit.getValue().matches("*[*]*") and
      (
        lit.getValue().matches("*allowLoadLocalInfile*") or
        lit.getValue().matches("*allowUrlInLocalInfile*") or
        lit.getValue().matches("*autoDeserialize*") or
        lit.getValue().matches("*allowPublicKeyRetrieval*") or
        lit.getValue().matches("*serverTimezone*") or
        lit.getValue().matches("*user*") or
        lit.getValue().matches("*password*")
      ) and
      this.asExpr() = lit
    )
    or
    // Parameters to filterSensitive method that may contain dangerous bracket content
    exists(Method m, Parameter p |
      m.hasName("filterSensitive") and
      m.getDeclaringType().getName() = "MySQLSensitiveUrlUtils" and
      p = m.getAParameter() and
      this.asParameter() = p
    )
  }
}

class VulnerableCodePatternSink extends DataFlow::Node {
  VulnerableCodePatternSink() {
    // The vulnerability: calls to filterSensitive in vulnerable code patterns
    exists(Method m, MethodCall filterCall |
      m.hasName("filterSensitive") and
      m.getDeclaringType().getName() = "MySQLSensitiveUrlUtils" and
      filterCall.getMethod() = m and
      this.asExpr() = filterCall and
      // Key vulnerability condition: this code exists where filterSensitiveKeyByBracket method is NOT available
```

```
      // In the vulnerable version, filterSensitiveKeyByBracket doesn't exist
      not exists(Method bracketMethod |
        bracketMethod.hasName("filterSensitiveKeyByBracket") and
        bracketMethod.getDeclaringType().getName() = "MySQLSensitiveUrlUtils" and
        bracketMethod.getDeclaringType() = m.getDeclaringType()
      )
    )
    or
    // Additional sink: method calls that use the result of inadequate filtering
    exists(MethodCall mc, MethodCall filterCall |
      filterCall.getMethod().hasName("filterSensitive") and
      filterCall.getMethod().getDeclaringType().getName() = "MySQLSensitiveUrlUtils" and
      DataFlow::localFlow(DataFlow::exprNode(filterCall), DataFlow::exprNode(mc.getArgument(_))) and
      this.asExpr() = mc and
      // Only vulnerable if no proper bracket filtering exists in the same class
      not exists(Method bracketMethod |
        bracketMethod.hasName("filterSensitiveKeyByBracket") and
        bracketMethod.getDeclaringType().getName() = "MySQLSensitiveUrlUtils" and
        bracketMethod.getDeclaringType() = filterCall.getMethod().getDeclaringType()
      )
    )
  }
}

class ProperBracketFilteringSanitizer extends DataFlow::Node {
  ProperBracketFilteringSanitizer() {
    // The proper bracket-based sanitization method (present only in fixed version)
    exists(MethodCall mc |
      mc.getMethod().hasName("filterSensitiveKeyByBracket") and
      mc.getMethod().getDeclaringType().getName() = "MySQLSensitiveUrlUtils" and
      this.asExpr() = mc
    )
  }
}

module MySQLJDBCUrlInjectionConfig implements DataFlow::ConfigSig {
  predicate isSource(DataFlow::Node source) {
    source instanceof MySQLDangerousBracketUrlSource
  }

  predicate isSink(DataFlow::Node sink) {
    sink instanceof VulnerableCodePatternSink
  }

  predicate isBarrier(DataFlow::Node sanitizer) {
    sanitizer instanceof ProperBracketFilteringSanitizer
  }
```

```
  predicate isAdditionalFlowStep(DataFlow::Node n1, DataFlow::Node
  ↪ n2) {
    // Flow through string concatenation operations
    exists(AddExpr addExpr |
      n1.asExpr() = addExpr.getLeftOperand() and
      n2.asExpr() = addExpr
    )
    or
    exists(AddExpr addExpr |
      n1.asExpr() = addExpr.getRightOperand() and
      n2.asExpr() = addExpr
    )
    or
    // Flow through variable assignments
    exists(Assignment assign |
      n1.asExpr() = assign.getSource() and
      n2.asExpr() = assign.getDest()
    )
    or
    // Flow through return statements
    exists(ReturnStmt ret |
      n1.asExpr() = ret.getResult() and
      n2.asParameter() =
      ↪ ret.getEnclosingCallable().getAParameter()
    )
  }
}

module MySQLJDBCUrlInjectionFlow =
↪ TaintTracking::Global<MySQLJDBCUrlInjectionConfig>;

import MySQLJDBCUrlInjectionFlow::PathGraph

from MySQLJDBCUrlInjectionFlow::PathNode source,
↪ MySQLJDBCUrlInjectionFlow::PathNode sink
where MySQLJDBCUrlInjectionFlow::flowPath(source, sink)
select sink.getNode(), source, sink,
  "MySQL JDBC URL with dangerous bracket parameters flows to
  ↪ vulnerable filtering logic at $@ that lacks proper
  ↪ bracket-based sanitization",
  source.getNode(), "dangerous URL source"
```

A.4 AST Extraction Query

Given a fix diff, FineNib automatically parses the changed methods and files, and inserts them into an AST pretty printing query template. Below is an example of the AST extraction query used for CVE-2014-7816.

```
/**
 * @name Expressions and statements for CVE-2014-7816 changed code areas
 * @description Extract expressions and statements from vulnerability fix areas
 * @id java/expr-stmt-diff-CVE_2014_7816
 * @kind problem
 * @problem.severity recommendation
 */

import java

from Element e, Location l
```

```
where
  l = e.getLocation() and ((l.getFile().getBaseName() = "PathSeparatorHandler.java"
  and l.getStartLine() >= 1 and l.getEndLine() <= 100) or

  (l.getFile().getBaseName() = "URLDecodingHandler.java"
  and l.getStartLine() >= 17 and l.getEndLine() <= 128)

  or (l.getFile().getBaseName() = "ResourceHandler.java"
  and l.getStartLine() >= 158 and l.getEndLine() <= 172)

  or (l.getFile().getBaseName() = "io.undertow.server.handlers.builder.HandlerBuilde
  and l.getEndLine() <= 29)

  or (l.getFile().getBaseName() = "DefaultServlet.java"
  and l.getStartLine() >= 39
  and l.getEndLine() <= 150)

  or (l.getFile().getBaseName() = "ServletPathMatches.java"
  and l.getStartLine() >= 32
  and l.getEndLine() <= 140))
select e,
  e.toString() as element,
  e.getAPrimaryQlClass() as elementType,
  l.getFile().getBaseName() as file,
  l.getStartLine() as startLine,
  l.getEndLine() as endLine,
  l.getStartColumn() as startColumn,
  l.getEndColumn() as endColumn
```

B CodeQL Language Server via MCP

The following are the MCP tool specifications and example usage for our custom CodeQL LSP client, wrapped as an MCP server.

B.1 Tool Specifications

`codeql_complete` Provides code completions at a specific position in a CodeQL file. Supports pagination for large completion lists and trigger character-based completions.

Inputs:

- `file_uri` (string): The URI of the CodeQL file
- `line` (number): Line number (0-based)
- `character` (number): Character position in the line (0-based)
- `trigger_character` (string, optional): Optional trigger character (e.g., ".", "::")
- `limit` (number, optional): Maximum number of completion items to return (default: 50)
- `offset` (number, optional): Starting position for pagination (default: 0)

Returns: `CompletionList` with pagination metadata containing completion items, each with label, kind, documentation, and text edit information.

Example usage:

```
{
  "tool": "codeql_complete",
  "arguments": {
    "file_uri": "file:///workspace/security-query.ql",
    "line": 5,
```

```
    "character": 12,
    "trigger_character": ".",
    "limit": 25
  }
}
```

codeql_hover Retrieves hover information (documentation, type information) at a specific position. Provides rich markdown documentation for CodeQL predicates, classes, and modules.

Inputs:

- `file_uri` (string): The URI of the CodeQL file
- `line` (number): Line number (0-based)
- `character` (number): Character position in the line (0-based)

Returns: `Hover | null` containing documentation content in markdown or plain text format, with optional range highlighting.

Example usage:

```
{
  "tool": "codeql_hover",
  "arguments": {
    "file_uri": "file:///workspace/security-query.ql",
    "line": 8,
    "character": 15
  }
}
```

codeql_definition Navigates to the definition location for a symbol at a specific position. Supports both single definitions and multiple definition locations.

Inputs:

- `file_uri` (string): The URI of the CodeQL file
- `line` (number): Line number (0-based)
- `character` (number): Character position in the line (0-based)

Returns: `Location | Location[] | null` containing URI and range information for definition locations.

Example usage:

```
{
  "tool": "codeql_definition",
  "arguments": {
    "file_uri": "file:///workspace/security-query.ql",
    "line": 12,
    "character": 8
  }
}
```

codeql_references Finds all references to a symbol at a specific position across the workspace. Includes both usage references and declaration references.

Inputs:

- `file_uri` (string): The URI of the CodeQL file
- `line` (number): Line number (0-based)
- `character` (number): Character position in the line (0-based)

Returns: `Location[] | null` containing an array of all reference locations with URI and range information.

Example usage:

```
{
  "tool": "codeql_references",
  "arguments": {
    "file_uri": "file:///workspace/security-query.ql",
    "line": 6,
    "character": 20
  }
}
```

`codeql_diagnostics` Retrieves diagnostics (errors, warnings, information messages) for a CodeQL file. Provides real-time syntax and semantic analysis results.

Inputs:

- `file_uri` (string): The URI of the CodeQL file

Returns: `Diagnostic[]` containing an array of diagnostic objects with severity, message, range, and optional related information.

Example usage:

```
{
  "tool": "codeql_diagnostics",
  "arguments": {
    "file_uri": "file:///workspace/security-query.ql"
  }
}
```

`codeql_format` Formats a CodeQL file or a specific selection within the file according to CodeQL style guidelines.

Inputs:

- `file_uri` (string): The URI of the CodeQL file
- `range` (Range, optional): Optional range to format with start and end positions

Returns: `TextEdit[]` containing an array of text edits that describe the formatting changes to be applied.

Example usage:

```
{
  "tool": "codeql_format",
  "arguments": {
    "file_uri": "file:///workspace/security-query.ql",
    "range": {
      "start": { "line": 10, "character": 0 },
      "end": { "line": 25, "character": 0 }
    }
  }
}
```

`codeql_update_file` Updates the content of an open CodeQL file in the language server. This allows for dynamic content modification and analysis of unsaved changes.

Inputs:

- `file_uri` (string): The URI of the CodeQL file

- `content` (string): The new complete content of the file

Returns: `string` containing a success confirmation message.

Example usage:

```
{
  "tool": "codeql_update_file",
  "arguments": {
    "file_uri": "file:///workspace/security-query.ql",
    "content": "import cpp\n\nfrom Function f\nwhere f.hasName(\"strcpy\")\nselect f,
  }
}
```

C Evaluation Details

Table 6 is a more detailed breakdown of the successful query synthesis rate by CWE.

C.1 Evaluation Limitations

Codex CLI. Claude Code and Gemini CLI allow users to configure how many max turns an agent can take in a context window. As of 9/23/2025, Codex CLI does not offer this configuration. Thus we were not able to force Codex to always take up to 50 max turns each context window.

IRIS. The original IRIS evaluation consists of 120 Java projects from CWE-Bench-Java. Many of these projects are old with deprecated dependencies, thus we were only able to build and use 112 of the projects with CodeQL 2.22.2. As of 9/23/2025, IRIS supports 11 CWEs and out of the 65 2025 CVEs, we were able to use 24 of them with IRIS. When running some of the IRIS queries, the amount of sources and sink predicates in the query led to out of memory errors. This impacted 9 out of the 24 IRIS 2025 CVE queries, thus we treat those as queries with 0 results and false recall.

Table 6: FineNib Query Success by CWE Type

CWE Type	Total CVEs	# Success	Success (%)	Avg Precision
CWE-022 (Path Traversal)	48	31	64.6%	0.750
CWE-079 (Cross-Site Scripting)	36	18	50.0%	0.621
CWE-094 (Code Injection)	20	12	60.0%	0.606
CWE-078 (OS Command Injection)	12	7	58.3%	0.628
CWE-502 (Deserialization)	6	4	66.7%	0.853
CWE-611 (XXE)	5	3	60.0%	0.657
CWE-287 (Authentication)	4	1	25.0%	0.875
CWE-200 (Information Exposure)	3	0	0.0%	0.667
CWE-400 (Resource Consumption)	3	1	33.3%	0.556
CWE-532 (Information Exposure)	3	3	100.0%	0.686
CWE-770 (Resource Exhaustion)	3	1	33.3%	0.444
CWE-020 (Improper Input Validation)	2	2	100.0%	0.650
CWE-089 (SQL Injection)	2	2	100.0%	1.000
CWE-1333 (ReDoS)	2	0	0.0%	0.000
CWE-284 (Access Control)	2	0	0.0%	0.500
CWE-862 (Authorization)	2	0	0.0%	0.000
CWE-918 (SSRF)	2	1	50.0%	0.500
CWE-023 (Relative Path Traversal)	1	1	100.0%	1.000
CWE-044 (Path Equivalence)	1	1	100.0%	0.667
CWE-083 (Improper Neutralization)	1	1	100.0%	0.052
CWE-1325 (Improperly Controlled Memory)	1	0	0.0%	0.000
CWE-164 (Foreign Code)	1	0	0.0%	0.000
CWE-178 (Case Sensitivity)	1	0	0.0%	1.000
CWE-190 (Integer Overflow)	1	0	0.0%	0.000
CWE-264 (Permissions)	1	0	0.0%	0.000
CWE-267 (Privilege Defined)	1	0	0.0%	0.000
CWE-276 (Incorrect Permissions)	1	0	0.0%	1.000
CWE-285 (Improper Authorization)	1	1	100.0%	1.000
CWE-288 (Authentication Bypass)	1	0	0.0%	0.000
CWE-290 (Authentication Bypass)	1	1	100.0%	1.000
CWE-297 (Improper Certificate)	1	1	100.0%	1.000
CWE-312 (Cleartext Storage)	1	0	0.0%	0.000
CWE-327 (Cryptographic Issues)	1	0	0.0%	0.000
CWE-346 (Origin Validation)	1	0	0.0%	0.200
CWE-352 (CSRF)	1	1	100.0%	0.941
CWE-426 (Untrusted Search Path)	1	0	0.0%	0.000
CWE-835 (Infinite Loop)	1	0	0.0%	0.000
CWE-863 (Authorization)	1	1	100.0%	1.000
Total	**176**	**94**	**53.4%**	**0.631**