LLM-Assisted Static Analysis for Detecting Security Vulnerabilities

Ziyang Li

University of Pennsylvania liby99@cis.upenn.edu

Saikat Dutta*

Cornell University saikatd@cornell.edu

Mayur Naik

University of Pennsylvania mhnaik@cis.upenn.edu

Abstract

Software is prone to security vulnerabilities. Program analysis tools to detect them have limited effectiveness in practice. While large language models (or LLMs) have shown impressive code generation capabilities, they cannot do complex reasoning over code to detect such vulnerabilities, especially because this task requires whole-repository analysis. In this work, we propose IRIS, the first approach that systematically combines LLMs with static analysis to perform whole-repository reasoning to detect security vulnerabilities. We curate a new dataset, CWE-Bench-Java, comprising 120 manually validated security vulnerabilities in real-world Java projects. These projects are complex, with an average of 300,000 lines of code and a maximum of up to 7 million. Out of 120 vulnerabilities in CWE-Bench-Java, IRIS detects 69 using GPT-4, while the state-of-the-art static analysis tool only detects 27. Further, IRIS also significantly reduces the number of false alarms (by more than 80% in the best case).

1 Introduction

Security vulnerabilities pose a major threat to the safety of software applications and its users. In 2023 alone, more than 29,000 CVEs were reported – almost 4000 higher than in 2022 [9]. Detecting vulnerabilities is extremely challenging despite advances in techniques to uncover them. A promising such technique called static taint analysis is widely used in popular tools such as GitHub CodeQL [1], Facebook Infer [11], Checker Framework [4], and Snyk Code [38]. These tools, however, face several challenges that greatly limit their effectiveness and accessibility in practice.

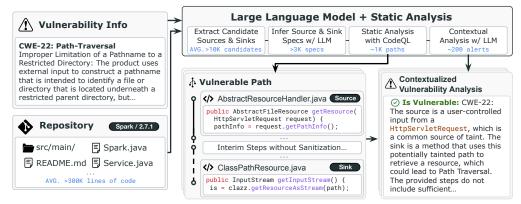


Figure 1: An overview of IRIS: our neuro-symbolic approach for vulnerability detection. Given a whole Java repository, IRIS checks whether it contains a certain type of vulnerability (CWE).

^{*}Work done while a postdoctoral scholar at the University of Pennsylvania.

False negatives due to missing taint specifications of third-party library APIs. First, static taint analysis predominantly relies on *specifications* of third-party library APIs as sources, sinks, or sanitizers. In practice, developers and analysis engineers have to manually craft such specifications based on their domain knowledge and API documentation. This is a laborious and error-prone process that often leads to missing specifications and incomplete analysis of vulnerabilities. Further, even if such specifications may exist for many libraries, they need to be periodically updated to capture changes in newer versions of such libraries and also cover new libraries that are developed.

False positives due to lack of precise context-sensitive and intuitive reasoning. Second, it is well-known that static analysis often suffers from low precision, i.e., it may often generate many false alarms [24, 21]. Such imprecision stems from multiple sources. For instance, the source or sink specifications may be spurious, or the analysis may over-approximate over branches in code or potential program inputs. Further, even if the specifications are correct, the context in which the detected source or sink is used may not lead to a vulnerability. Hence, in reality, a developer may need to triage through several (potentially false) security alerts, wasting developer time and effort.

Limitations of prior data-driven approaches to improve static taint analysis. Many techniques have been proposed to address the challenges of static taint analysis. For instance, Livshits et al. [33] proposed an automated approach, MERLIN, to mine specifications from library A more recent work, Seldon [6], improves the scalability of this approach by formulating the taint specification inference problem as a linear optimization task. However, such approaches rely on analyzing the code of third-party library to extract specifications, which is expensive and hard to scale. Researchers have also developed statistical and learning-based techniques to mitigate false positive alerts [23, 18, 35]. However, such approaches still have limited effectiveness in practice [24].

Large Language Models (or LLMs) have made impressive strides in code generation and summarization. LLMs have also been applied to code related tasks such as program repair [44], code translation [34], test generation [26], and static analysis [27]. Recent studies [41, 25] evaluated LLMs' effectiveness at detecting vulnerabilities at the method level and showed that LLMs fail to do complex reasoning with code, especially because it depends on the *context* in which the method is used in the project. On the other hand, recent benchmarks like SWE-Bench [20] show that LLMs are also poor at doing project-level reasoning. Hence, an intriguing question is whether LLMs can be combined with static analysis to improve their reasoning capabilities. In this work, we answer this question in the context of vulnerability detection and make the following contributions:

Approach. We propose **IRIS**, the *first* neuro-symbolic approach for vulnerability detection that combines the strengths of static analysis and LLMs without suffering their limitations. Given a project to analyze for a given vulnerability class (or CWE), IRIS applies LLMs for mining CWE-specific taint specifications for third-party library APIs used in the project. IRIS augments such specifications with CodeQL, a static analysis tool, and uses it to detect security vulnerabilities. Our intuition here is because LLMs have seen numerous usages of such library APIs across projects, they have an understanding of which APIs are relevant for different CWEs.

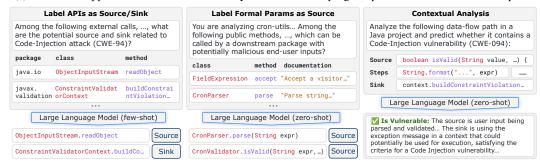
Further, to address the imprecision problem of static analysis, we develop a contextual analysis technique with LLMs that drastically reduces the false positive alarms and minimizes the triaging effort for developers. Our key insight is that encoding the code-context and path-sensitive information in the prompt elicits more reliable reasoning from LLMs. Finally, our neuro-symbolic approach allows LLMs to do more precise whole-repository reasoning and minimizes the human effort involved in using static analysis tools. Figure 1 presents the end-to-end pipeline of IRIS.

Dataset. We curate a dataset of manually vetted and compilable Java projects, **CWE-Bench-Java**, containing 120 vulnerabilities (one per project) across four common vulnerability classes. The projects in the dataset are complex, containing 300K lines of code on average, and 10 projects with more than a million lines of code, making it a challenging benchmark for vulnerability detection.

Results. We evaluate IRIS on CWE-Bench-Java using eight diverse open- and closed-source LLMs. Overall, IRIS obtains the best results with GPT-4, detecting 69 vulnerabilities, which is 42 (35%) more than CodeQL. Among open-source LLMs, DeepSeekCoder 7B, despite being much smaller than other LLMs performs the best, detecting 67 vulnerabilities, followed by Llama 3 70B with 57 vulnerabilities. Further, our context-based filtering technique reduces false positive alerts by 80%.



(a) The code snippets related to the vulnerability. We number the program points within the vulnerable path.



(b) Three LLM-based components in our pipeline that enable full automation of vulnerability detection.

Figure 2: (a) An example of Code Injection vulnerability in cron-utils (CVE-2021-41269) that CodeQL fails to detect, with relevant code snippets. (b) We demonstrate how IRIS detects it by applying LLMs to infer sources/sinks, and to reducing false alarms via contextual analysis.

2 Motivating Example

We illustrate the effectiveness of IRIS in detecting a previously known code-injection (CWE-094) vulnerability in cron-utils (ver. 9.1.5), a Java library for Cron data manipulation. Fig. 2a shows the relevant code snippets. A user-controlled string value passed into isValid function is transferred without sanitization to the parse function. If an exception is thrown, the function constructs an error message with the input. However, the error message is used to invoke method buildConstraintViolationWithTemplate of class ConstraintValidatorContext in javax.validator, which interprets the message string as a Java Expression Language (Java EL) expression. A malicious user may exploit this vulnerability by crafting a string containing a shell command such as Runtime.exec('rm -rf /') to delete critical files on the server.

Detecting this vulnerability poses several challenges. First, the cron-utils library consists of 13K SLOC (lines of code excluding blanks and comments), which needs to be analyzed to find this vulnerability. This process requires analyzing data and control flow across several internal methods and third-party APIs. Second, the analysis needs to identify relevant *sources* and *sinks*. In this case, the value parameter of the public isValid method may contain arbitrary strings when invoked, and hence may be a source of malicious data. Additionally, external APIs like buildConstraintViolationWithTemplate can execute arbitrary Java EL expressions, hence they should be treated as sinks that are vulnerable to Code Injection attacks. Finally, the analysis also requires identifying any sanitizers that block the flow of untrusted data, as well as intuitive understanding of contextual information of the source and sink to suppress false positives.

Modern static analysis tools, like CodeQL, are effective at tracing taint data flows across complex codebases. However, CodeQL fails to detect this vulnerability due to missing specifications. CodeQL includes many manually curated specifications for sources and sinks across more than 360 popular Java library modules. However, manually obtaining such specifications requires significant human effort to analyze, specify, and validate. Further, even with perfect specifications, CodeQL may often generate numerous false positives due to a lack of contextual reasoning, increasing the developer's burden of triaging the results.

In contrast, IRIS takes a different approach by inferring project- and vulnerability-specific specifications *on-the-fly* by using LLMs. As shown in Fig. 2b, with simple prompts, the LLM-based

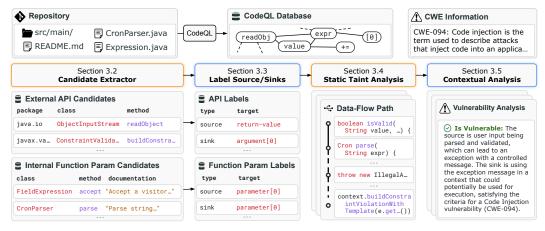


Figure 3: An illustration of IRIS's pipeline.

components in IRIS correctly identify the untrusted source and the vulnerable sink. IRIS augments CodeQL with these specifications and successfully detects the unsanitized data-flow path between the detected source and sink in the repository. However, in addition to the true vulnerable path, augmented CodeQL produces many false positives, which are hard to eliminate using concrete rules. To solve this challenge, IRIS encodes the detected code paths and the surrounding context into a simple prompt and uses an LLM to classify it as true or false positive. Specifically, out of 8 paths reported by static analysis, 5 false positives are filtered out, leaving the path in Fig. 2a as one of the final alarms. While we use GPT 4 for the above example, we show that one can easily adapt a wide range of LLMs for effective vulnerability detection by using zero- and few-shot prompting techniques. Overall, we observe that IRIS can detect many such vulnerabilities that are beyond the reach of CodeQL-like static analysis tools, while keeping false alarms to a minimum.

3 IRIS Framework

At a high level, IRIS takes a Java project P, the vulnerability class C to detect, and a large language model LLM, as inputs. IRIS statically analyzes the project P, checks for vulnerabilities specific to C, and returns a set of potential security alerts A. Each alert is accompanied by a unique code path from a taint source to a taint sink that is vulnerable to C (i.e., the path is unsanitized).

As illustrated in Fig. 3, IRIS has four main stages: First, IRIS builds the given Java project and uses static analysis to extract all candidate APIs, including invoked external APIs and internal function parameters. Second, IRIS queries an LLM to label these APIs as sources or sinks that are specific to the given vulnerability class C. Third, IRIS transforms the labeled sources and sinks into specifications that can be fed into a static analysis engine, such as CodeQL, and runs a vulnerability class-specific taint analysis query to detect vulnerabilities of that class in the project. This step generates a set of vulnerable code paths (or alerts) in the project. Finally, IRIS triages the generated alerts by automatically filtering false positives, and presents them to the developer.

3.1 Problem Statement

We formally define the static taint analysis problem for vulnerability detection. Given a project P, taint analysis extracts an inter-procedural data flow graph $\mathbb{G} = (\mathbb{V}, \mathbb{E})$, where \mathbb{V} is the set of nodes representing program expressions and statements, and $\mathbb{E} \subseteq \mathbb{V} \times \mathbb{V}$ is the set of edges representing data or control flow edges between the nodes. A vulnerability detection task comes with two sets $V^C_{source} \subseteq \mathbb{V}$, $V^C_{sink} \subseteq \mathbb{V}$ that denote source nodes where tainted data may originate and sink nodes where a security vulnerability can occur if tainted data reaches it, respectively. Naturally, different classes C of vulnerabilities (or CWEs) have different source and sink specifications. Additionally, there can be sanitizer specifications, $V^C_{sanitizer} \in \mathbb{V}$, that block the flow of tainted data (such as escaping special characters in strings).

The goal of taint analysis is to find pairs of sources and sinks, $(V_s \in \boldsymbol{V}_{source}^C, V_t \in \boldsymbol{V}_{sinks}^C)$, such that there is an unsanitized path from the source to the sink. More formally, $\textit{Unsanitized_Paths}(V_s, V_t) = \exists \textit{Path}(V_s, V_t) \text{ s.t. } \forall V_n \in \textit{Path}(V_s, V_t), V_n \notin \boldsymbol{V}_{\textit{sanitizer}}^C$. Here, $\textit{Path}(V_1, V_k)$ denotes a sequence of nodes (V_1, V_2, \dots, V_k) , such that $V_i \in \mathbb{V}$ and $\forall i \in 1$ to k-1: $(v_i, v_{i+1}) \in \mathbb{E}$.

Two key challenges in taint analysis include: 1) identifying relevant taint specifications for each class C that can be mapped to V_{source}^C , V_{sink}^C for any project P, and 2) effectively eliminating false positive paths in $Unsanitized_Paths(V_s, V_t)$ identified by taint analysis. In the following sections, we discuss how we address each challenge by leveraging LLMs.

3.2 Candidate Source/Sink Extraction

A project may use various third-party APIs whose specifications (i.e., source or sink) may be unknown – reducing the effectiveness of taint analysis. In addition, internal APIs might accept untrusted input from downstream libraries as well. Hence, our goal is to automatically infer specifications for such APIs. We define a specification S^C as a 3-tuple $\langle T, F, R \rangle$, where $T \in \{ReturnValue, Argument, Parameter, \dots\}$ is the type of node to match in \mathbb{G} , F is an N-tuple of strings describing the package, class, method name, signature, and argument/parameter position (if applicable) of an API, and $R \in \{Source, Sink, Sanitizer\}$ is the role of the API. For example, the specification $\langle Argument, (java.lang, Runtime, exec, (String[]), 0), Sink \rangle$ denotes that the first argument of exec method of Runtime class is a sink for a vulnerability class (OS command injection). A static analysis tool maps these specifications to sets of nodes V^C_{source} or V^C_{sink} in \mathbb{G} .

To identify taint specifications S_{source}^{C} and S_{sink}^{C} , we first extract S^{ext} : external (third-party) library APIs that are invoked in the given Java project and are potential candidates to be taint sources or sinks. We also extract S^{int} , internal library APIs that are public and may be invoked by a downstream library. We utilize CodeQL's query language to extract such candidates and their corresponding metadata such as method name, type signature, enclosing packages and classes, and even JavaDoc documentations, if applicable.

3.3 Inferring Taint Specifications using LLMs

We develop an automated specification inference technique: $LabelSpecs(S^\#, LLM, C, R) = S_R^C$, where $S^\# = S^{\rm ext} \cup S^{\rm int}$ are candidate specifications for sources and sinks. In this work, we do not consider sanitizer specifications, because they typically do not vary for the vulnerability classes that we consider. We use LLMs to infer taint specifications. Specifically, external APIs in $S^{\rm ext}$ can be classified as either source or sink, while internal APIs in $S^{\rm int}$ can have their formal parameters identified as sources. The left and middle panel of Fig. 2b show partial user prompts for inferring source and sink specifications from external APIs and internal function formal parameters.

Due to the sheer number of APIs to be labeled, we insert a batch of APIs in a single prompt and ask the LLM to respond with JSON formatted strings. The batch size is a tunable hyper-parameter. We adopt few-shot (usually 3-shot) prompting strategy for labeling external APIs, while zero-shot is used for labeling internal APIs. Notably for internal APIs, we also include information from repository readme and JavaDoc documentations, if applicable. In practice, we find that this extra information helps LLM understand the high-level purpose and usage of the codebase, resulting in better labeling accuracy. Due to space limitation, we leave the full prompt templates and other implementations details in the Appendix. At the end of this stage, we have successfully obtained S^{C}_{source} and S^{C}_{sink} which are going to be used by the static analysis engine in the next stage.

3.4 Vulnerability Detection

Once we obtain all the source and sink specifications from the LLM, the next step is to combine it with a static analysis engine to detect vulnerable paths, i.e., $Unsanitized_Paths(V_s, V_t)$, in a given project. In this work, we use CodeQL [14] for this step. CodeQL represents programs as data flow graphs and provides a query language, akin to Datalog [37], to analyze such graphs. Many security vulnerabilities and bugs can be modeled using *queries* written in CodeQL and can be executed against data flow graphs extracted from such programs. Given a data flow graph \mathbb{G}^P of a project P, CWE-specific source and sink specifications, and a query for a given vulnerability class C, CodeQL

returns a set of unsanitized paths in the program. Formally,

$$CodeQL(\mathbb{G}^{P}, \mathbf{S}_{source}^{C}, \mathbf{S}_{sink}^{C}, Query^{C}) = \{Path_{1}, \dots, Path_{k}\}$$

CodeQL itself contains numerous specifications of third-party APIs for each vulnerability class. However, as we show later in our evaluation, despite having such specialized queries and extensive specifications, CodeQL fails to detect a majority of vulnerabilities in real-world projects. For our analysis, we write a specialized CodeQL query for each vulnerability that uses our mined specifications instead of those provided by CodeQL. Our query for Path Traversal vulnerability (CWE 22) is shown in Listing 3 in the appendix. We develop similar queries for each CWE that we evaluate.

3.5 Triaging of Alerts via Contextual Analysis

Inferring taint specifications only solves part of the challenge. We observe that while LLMs help uncover many new API specifications, sometimes they detect specifications that are not relevant to the vulnerability class being considered, resulting in too many predicted sources or sinks and consequently many spurious alerts as a result. For context, even a few hundred taint specifications may sometimes produce thousands of $Unsanitized_Paths(V_s, V_t)$ that a developer needs to triage. To reduce the developer burden, we also develop an LLM-based filtering method, $FilterPath(Path, \mathbb{G}, LLM, C) = True|False$ that classifies a detected vulnerable path (Path) in \mathbb{G} as a true or false positive by leveraging context-based and natural language information.

Figure 4 presents an example prompt for contextual analysis. The prompt includes CWE information and code snippets for nodes along the path, with an emphasis on the source and sink. Specifically, we include ± 5 lines surrounding the exact source and sink location, as well as the enclosing function and class. The exact line of source and sink is marked with a comment. For the intermediate steps, we include the file names and the line of code. When the path is too long, we keep only a subset of nodes to limit the size of the prompt. As such, we provide the full context for the potential vulnerability to be thoroughly analyzed.

We expect the LLM to respond in JSON format with the final verdict as well as an explanation to the verdict. The JSON format prompts the LLM to generate the explanation before delivering the final verdict, as presenting the judgment after the reasoning process is known to yield better results. In addition, if the verdict is false, we ask the LLM to indicate whether the source or sink is a false positive, which helps to prune other paths and thereby save on the number of calls to the LLM.

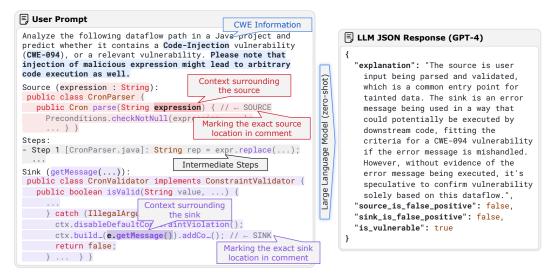


Figure 4: LLM user prompt and response for contextual analysis of data-flow paths. We modify the snippets and left out the system prompt for clearer presentation.

3.6 Evaluation Metrics

We evaluate the performance of IRIS using one key metrics, VulDetected. For evaluation, we assume that the label for a project P is provided as a set of crucial program points $\mathbf{V}_{\mathrm{fix}}^P = \{V_1, \dots, V_n\}$

where the vulnerable paths should pass through. In practice, these are typically the patched methods that can be collected from each vulnerability report. If at least one detected vulnerable path passes through a fixed location for the given vulnerability, then we consider the vulnerability detected. Suppose $Paths_P$ is the set of detected and/or filtered paths from prior stages, the metrics, including #Vuls the total number of detected vulnerabilities across a set of projects, is formally defined as:

$$VulDetected(Paths_P, \mathbf{V}_{\text{fix}}^P) = \mathbb{1}_{|\{Path \in Paths_P \mid Path \cap \mathbf{V}_{\text{fix}} \neq \emptyset\}| > 0}$$

$$#Vuls(P_1, \dots, P_n) = \sum_i VulDetected(Paths_{P_i}, \mathbf{V}_{\text{fix}}^{P_i})$$

4 CWE-Bench-Java: A Dataset of Security Vulnerabilities in Java

To evaluate our approach, we require a dataset of vulnerable versions of Java projects with several important characteristics: 1) Each benchmark should have relevant **vulnerability metadata**, such as the CWE ID, CVE ID, fix commit, and vulnerable project version, 2) each project in the dataset must be **compilable**, which is a key requirement for static analysis and data flow graph extraction, 3) the projects must be **real-world**, which are typically more complex and hence challenging to analyze compared to synthetic benchmarks, and 4) finally, each vulnerability and its location (e.g., method) in the project must be **validated** so that this information can be used for robust evaluation of vulnerability detection tools. Unfortunately, no existing dataset satisfies all these requirements. Table 5 presents a comparison of our dataset, which we discuss next, with prior vulnerability datasets.

To address these requirements, we curate our own dataset of vulnerabilities. For this paper, we focus only on vulnerabilities in Java libraries that are available via the widely used Maven package manager. We choose Java because it is commonly used to develop server-side, Android, and web applications, which are prone to security risks. Further, due to Java's long history, there are many existing CVEs in numerous Java projects that are available for analysis. We initially use the GitHub Advisory database [15, 16] to obtain such vulnerabilities, and further filter it with cross-validated information from multiple sources, including manual verification. Fig. 5 illustrates the complete set of steps for curating CWE-Bench-Java.

As shown in the statistics (Fig. 5), the sheer size of these projects make them challenging to analyze for any static analysis tool or ML-based tool. Each project in CWE-Bench-Java comes with GitHub information, vulnerable and fix version, CVE metadata, a script that automatically fetches and builds, and the set of program locations that involve the vulnerability. Our tools are easily extensible and can be directly used to extract vulnerabilities with other CWEs.

5 Evaluation

We perform extensive experimental evaluations of IRIS and demonstrate its practical effectiveness in detecting vulnerabilities in real-world Java repositories in CWE-Bench-Java. Due to space limits, we include additional results and analyses in the appendix. We answer the following research questions:

- **RO 1:** How many previously known vulnerabilities can IRIS detect?
- **RO 2:** How effective is the contextual filtering approach in reducing false positive alerts?
- RQ 3: How good are the inferred source/sink specifications by IRIS?



Figure 5: Steps for curating CWE-Bench-Java, and dataset statistics.

Table 1: Performance comparison of CodeQL (QL) vs IRIS on vulnerability detection. We present results when using different LLMs in IRIS including Gemma (G) 7B, Llama 3 (L3) 8B and 70B, Mistral (Mi) 7B, DeepSeekCoder (DS) 7B and 33B, GPT 3.5 and GPT 4. The values in parentheses compare the number of vulnerabilities detected by IRIS with a given LLM against CodeQL.

CWE #Vuls #Vuls Detected										
		QL	IRIS with							
			G 7B	L3 7B	Mi 7B	DS 7B	DS 33B	L3 70B	GPT 3.5	GPT 4
22	55	22	3 (\ 19)	19 (\ 3)	28 († 6)	28 († 6)	10 (\ 12)	29 († 7)	27 († 5)	34 († 12)
78	13	1	0 (\ 1)	3 († 2)	4 († 3)	6 († 5)	8 († 7)	2 († 1)	1 (= 0)	7 († 6)
79	31	4	11 († 7)	13 († 9)	$11 (\uparrow 7)$	19 († 15)	20 († 16)	16 († 12)	15 († 11)	17 († 13)
94	21	0	8 († 8)	12 († 12)	6 († 6)	14 († 14)	7 († 7)	10 († 10)	10 († 10)	11 († 11)
Total	120	27	22 (\ 5)	47 († 20)	49 († 22)	67 († 40)	45 († 18)	57 († 30)	53 († 26)	69 († 42)

5.1 Experimental Setup

LLM selection and CodeQL baseline. We select two closed-source LLMs from OpenAI: GPT 4 (gpt-4-0125-preview) and GPT 3.5 (gpt-3.5-turbo-0125) for our evaluation. We also select instruction-tuned versions of six state-of-the-art open-source LLMs via huggingface API: Llama 3 8B and 70B, DeepSeekCoder 7B and 33B, Mistral 7B, and Gemma 7B. For baseline, we use CodeQL 2.15.3 and its built-in Security queries specifically designed for each CWE.

5.2 RQ1: Vulnerability Detection Effectiveness of IRIS

The results, as summarized in Table 1, demonstrate the superior performance of IRIS across all CWEs with most LLMs, compared to CodeQL. Specifically, IRIS successfully identifies 69 vulnerabilities with GPT-4, 42 more than CodeQL. While GPT-4 excels the most, smaller but more specialized LLM like DeepSeekCoder 8B can also detect 67. To further contextualize the significance of results, we note that the dataset contains many exceptionally large and complex codebases such as Apache Camel, which contains over 1M SLOC. Notably with GPT-4, IRIS identifies the vulnerability within Camel, further underscoring its scalability. The primary reason for such gain in performance is due to the relevant taint specifications that are inferred by LLMs. Except for Gemma 7B, we observe that IRIS obtains significant better results than CodeQL with all other LLMs.

Effectiveness across LLMs. Out of 120 vulnerabilities, 34 are not detected by IRIS with any LLM, 78 are detected by at least two LLMs, and 10 are detected by all. In fact, all but one vulnerability detected by CodeQL cannot be detected by IRIS with GPT-4. The undetected case is due to a missing source specification related to Java annotations, which we currently do not support. Moreover, 8 vulnerabilities are detected by only one LLM: GPT-4 (1), Mistral 7B (1), DeepSeekCoder 7B (4), and DeepSeekCoder 33B (2). Observing only small discrepancies, we conclude that LLMs share a lot of common knowledge and are generally applicable to specification inference.

Effectiveness across CWEs. We observe that CWE-78 (OS Command Injection) is particularly challenging to detect for most LLMs. Our manual investigation revealed that the vulnerability patterns in CWE-78 are particularly complex, involving injection of OS commands through gadget-chains [2] or external side-effects like file writes, which cannot be tracked using static analysis. This underscores the fundamental limitations of static analysis (vs dynamic), which is beyond the scope of this work. Notwithstanding, we find IRIS to improve recall across all CWEs.

5.3 RQ2: Effectiveness of Contextual Analysis Approach in IRIS

We next study the effectiveness of contextual analysis in reducing false alarms. While filtering paths with contextual analysis can reduce false alarms, it may also potentially reduce the number of detected vulnerabilities. Table 2 presents the average number of reported paths per project and the number of detected vulnerabilities that remain after filtering. Due to the large number of paths per project, we only perform this experiment on a randomly selected set of 40 projects (10 per CWE)

Table 2: Effectiveness of Contextual Analysis. We present the average number of reported vulnerable paths (#Paths) and the number of detected vulnerabilities (#Vuls) after contextual analysis using LLMs across a subset of 40 projects. The numbers in parentheses indicate the reduction in #Paths (percentage) or detected vulnerabilities compared to before the contextual analysis.

CWE	CodeQL DeepSeekCoder 7B		der 7B	Llama 3 7	70B	GPT-4		
	#Paths	#Vuls	#Paths (%Red.)	#Vuls	#Paths (%Red.)	#Vuls	#Paths (%Red.)	#Vuls
22	62	6	282 (\ 59%)	7 (= 0)	152 (\ 46%)	7 (= 0)	93 (↓ 70%)	10 (= 0)
78	4	1	347 (\psi 55%)	4 (\ 1)	24 (\ 42%)	2 (= 0)	22 (\ 95%)	3 (4)
79	32	2	1002 (\ 44%)	10 (= 0)	406 (\ 34%)	10 (= 0)	350 (\psi 79%)	10 (= 0)
94	0	0	1037 (\psi 50%)	7 (= 0)	795 (\ 34%)	7 (= 0)	270 (\ 82%)	6 (\ 1)
All	26	9	648 (\ 50%)	28 (1)	328 (\ 36%)	26 (= 0)	179 (↓ 81%)	29 (\$\sqrt{5})

across the three best LLMs from Section 5.2. In all cases, we see a minor drop in #Vuls but a much more significant drop in average #Paths reported, up to a reduction rate of 81% with GPT-4.

Better reasoning capability in larger LLMs helps contextual analysis. Our manual investigation reveals that GPT-4 is very good at reasoning about the predicted paths, which aligns with its highest reduction rate in Table 2. GPT-4 not only reasons about the trustworthiness of taint sources but also understands the developer's intent using natural language information in code and comments, which allows it to filter out many false positives. We also observe that when true positives are marked false by GPT-4, it is typically because the given context is insufficient for analysis. However, based on the path information alone, GPT-4 can already perform a very precise and reasonable analysis. It remains to be seen how further context improves precision, but we leave it for future work.

5.4 RQ3: Quality of Inferred Source and Sink Specifications

Next, we analyze whether LLMs are capable of inferring high-quality source and sink specifications. Table 3 presents the statistics on LLM-inferred specifications, showing that about 4% of candidates are labeled on average. We perform manual analysis on ten random samples each of source and sink specifications, per combination of CWE and LLM, and estimate the overall precision of specifications (Fig. 6). Clearly, GPT-4 obtains significantly higher precision (over 70%) than other LLMs.

Over-approximate specifications are beneficial. While the precision for other LLMs is lower, we note that the over-approximation is, in fact, a partial solution to overcome a fundamental limitation of CodeQL, viz., limited taint specifications. LLMs thus have the potential to expand the coverage of taint analysis. However, we also note that the effect of this imprecision can be mitigated by a lot via contextual analysis, which we presented in Sec. 5.3.

Continuous taint specification inference is necessary. Our results show that there is a high number of both unique and recurring sources and sinks: about 900 per CWE unique and 700 recurring (tables in Appendix). This indicates that even if previously inferred specifications are useful, a significant number of new relevant APIs still remain and need to be labeled for effective vulnerability detection.

Table 3: Ratio of candidates labeled as source (S) or sink (N) by GPT-4 and DeepSeekCoder (DS) 7B.

CWE #Cand.		GP	T-4	DS 7B		
		%S	%N	%S	%N	
22	130,974	2.03%	1.90%	4.27%	4.01%	
78	25,605	4.73%	3.37%	3.67%	3.33%	
79	37,138	5.69%	4.69%	4.28%	4.56%	
94	36,325	5.12%	7.83%	6.11%	6.21%	
Total	230,042	3.41%	3.45%	4.50%	4.37%	

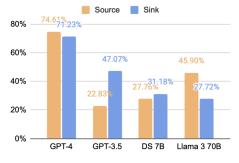


Figure 6: Estimated precision of inferred source and sink specs by selected LLMs.

This observation strongly motivates the design of IRIS that infers these specifications *on-the-fly* for each project via LLMs, instead of relying on a fixed corpus of specifications like CodeQL.

6 Related Work

ML-based approaches for vulnerability detection. Numerous prior techniques incorporate deep learning for detecting vulnerabilities. This includes techniques that use Graph Neural Network (GNN)-based representations of code such as Devign [48], Reveal [3], LineVD [19], and IVDetect [29]; LSTM-based models for representing program slices and data dependencies such as VulDeePecker [31] and SySeVR [30]; and fine-tuning of Transformer-based models such as Line-Vul [13], DeepDFA [40], and ContraFlow [5]. These approaches focus on method-level detection of vulnerabilities and provide only a binary label classifying a method as vulnerable or not. In contrast, IRIS performs whole-project analysis and provides a distinct code path from a source to a sink and can be tailored for detecting different CWEs. More recently, multiple studies demonstrated that LLMs are not effective at detecting vulnerabilities in real-world code [41, 10, 25]. While these studies only focused on method-level vulnerability detection, it reinforces our motivation that detecting vulnerabilities requires whole-project reasoning, which LLMs currently cannot do alone.

Static analysis tools. Apart from CodeQL [1], other static analysis tools like CppCheck [8], Semgrep [36], FlawFinder [12], Infer [11], and CodeChecker [7] also include analyses for vulnerability detection. But, these tools are not as feature-rich and effective as CodeQL [28, 32]. Recently, proprietary tools such as Snyk [38] and SonarQube [39] are also gaining in popularity. However, like CodeQL, these tools share the same fundamental limitations of missing specifications and false positives, which IRIS improves upon. Potentially, our techniques stand to benefit all such tools.

LLMs for software engineering. Researchers are increasingly combining LLMs with software engineering tools for challenging tasks such as fuzzing [26, 45], program repair [44, 22, 43], and fault localization [46]. While we follow a similar direction, to the best of our knowledge, our work is the first to combine LLMs with static analysis to detect security vulnerabilities via whole-project analysis. Recently, LLM-based agents such as AutoCodeRover [47] and SWE-Agent [42] are also pushing the boundaries on whole-project repair. Hence, in the future, we plan to explore a richer combination of tools in IRIS to further improve the performance of vulnerability detection.

7 Conclusion and Limitations

We presented IRIS, a novel neuro-symbolic approach that combines LLMs with static analysis for vulnerability detection. We also curated a dataset, CWE-Bench-Java, containing 120 security vulnerabilities across four classes in real-world projects. Our experimental results show that IRIS can significantly surpass state-of-the-art static analysis tools, like CodeQL, in detecting critical vulnerabilities, even when using smaller LLMs. Further, IRIS's context filtering techniques also drastically reduce the number of false positives. Overall, our results show that effectively combining LLMs with static analysis leads to more effective analysis and reduces the developer burden. However, there are still many vulnerabilities that can yet be detected by this approach. Hence, future approaches may explore a tighter integration of these two tools to further improve performance.

Limitations. Determining whether a vulnerability is detected is a difficult task. Hence, we rely on manual validation to find vulnerable method locations in each case, limiting the size of CWE-Bench-Java. However, because each project is still quite large, we believe CWE-Bench-Java will remain a challenging benchmark for future works. IRIS makes numerous calls to LLMs for specification inference and filtering false positives, increasing the potential cost of analysis. However, as our results show, even smaller models, like Llama 3 8B and DeepSeekCoder 7B, can perform well on these tasks and can be potentially deployed locally for a project. For any given project, the cost of labeling will consolidate quickly across versions, making IRIS a viable choice for potential integration into CI/CD pipelines. While our results on Java benchmarks are promising, it is unknown if IRIS will perform well on other languages. We plan to explore this further in future work.

References

- [1] P. Avgustinov, O. de Moor, M. P. Jones, and M. Schäfer. Ql: Object-oriented queries on relational data. In *European Conference on Object-Oriented Programming*, 2016. URL https://api.semanticscholar.org/CorpusID:13385963.
- [2] S. Cao, X. Sun, X. Wu, L. Bo, B. Li, R. Wu, W. Liu, B. He, Y. Ouyang, and J. Li. Improving java deserialization gadget chain mining via overriding-guided object generation. In *Proceedings of the 45th International Conference on Software Engineering (ICSE)*, 2023. doi: 10.1109/ICSE48619.2023.00044.
- [3] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray. Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering*, 48:3280–3296, 2020. URL https://api.semanticscholar.org/CorpusID:221703797.
- [4] Checker Framework, 2024. https://checkerframework.org/.
- [5] X. Cheng, G. Zhang, H. Wang, and Y. Sui. Path-sensitive code embedding via contrastive learning for software vulnerability detection. *Proceedings of the 31st ACM SIG-SOFT International Symposium on Software Testing and Analysis*, 2022. URL https://api.semanticscholar.org/CorpusID:250562410.
- [6] V. Chibotaru, B. Bichsel, V. Raychev, and M. Vechev. Scalable taint specification inference with big code. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 760–774, 2019.
- [7] Code Checker, 2023. https://github.com/Ericsson/codechecker.
- [8] CPPCheck, 2023. https://cppcheck.sourceforge.io/.
- [9] CVE Trends, 2024. https://www.cvedetails.com.
- [10] Y. Ding, Y. Fu, O. Ibrahim, C. Sitawarin, X. Chen, B. Alomair, D. Wagner, B. Ray, and Y. Chen. Vulnerability detection with code language models: How far are we? *arXiv* preprint *arXiv*:2403.18624, 2024.
- [11] Fb Infer, 2023. https://fbinfer.com/.
- [12] FlawFinder, 2023. URL https://dwheeler.com/flawfinder.
- [13] M. Fu and C. Tantithamthavorn. Linevul: A transformer-based line-level vulnerability prediction. In 2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR). IEEE, 2022.
- [14] GitHub. Codeql, 2024. https://codeql.github.com.
- [15] GitHub. Github advisory database, 2024. https://github.com/advisories.
- [16] GitHub. Github security advisories, 2024. https://github.com/github/advisory-database.
- [17] J. He and M. Vechev. Large language models for code: Security hardening and adversarial testing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 1865–1879, 2023.
- [18] S. Heckman and L. Williams. A model building process for identifying actionable static analysis alerts. In 2009 International conference on software testing verification and validation, pages 161–170. IEEE, 2009.
- [19] D. Hin, A. Kan, H. Chen, and M. A. Babar. Linevd: Statement-level vulnerability detection using graph neural networks. 2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR), pages 596-607, 2022. URL https://api.semanticscholar.org/CorpusID:247362653.
- [20] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan. Swe-bench: Can language models resolve real-world github issues? arXiv preprint arXiv:2310.06770, 2023.
- [21] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In 2013 35th International Conference on Software Engineering (ICSE), pages 672–681. IEEE, 2013.

- [22] H. Joshi, J. C. Sanchez, S. Gulwani, V. Le, G. Verbruggen, and I. Radiček. Repair is nearly generation: Multilingual program repair with llms. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 5131–5140, 2023.
- [23] Y. Jung, J. Kim, J. Shin, and K. Yi. Taming false alarms from a domain-unaware c analyzer by a bayesian statistical post analysis. In *International Static Analysis Symposium*, pages 203–217. Springer, 2005.
- [24] H. J. Kang, K. L. Aw, and D. Lo. Detecting false alarms from automatic static analysis tools: How far are we? In *Proceedings of the 44th International Conference on Software Engineering*, pages 698–709, 2022.
- [25] A. Khare, S. Dutta, Z. Li, A. Solko-Breslin, R. Alur, and M. Naik. Understanding the effectiveness of large language models in detecting security vulnerabilities. *arXiv* preprint *arXiv*:2311.16169, 2023.
- [26] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *International conference on software engineering (ICSE)*, 2023.
- [27] H. Li, Y. Hao, Y. Zhai, and Z. Qian. Enhancing static analysis for practical bug detection: An llm-integrated approach. *Proceedings of the ACM on Programming Languages (PACMPL), Issue OOPSLA*, 2024.
- [28] K. Li, S. Chen, L. Fan, R. Feng, H. Liu, C. Liu, Y. Liu, and Y. Chen. Comparison and evaluation on static application security testing (sast) tools for java. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 921–933, 2023.
- [29] Y. Li, S. Wang, and T. N. Nguyen. Vulnerability detection with fine-grained interpretations. Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2021. URL https://api.semanticscholar.org/CorpusID:235490574.
- [30] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, Z. Chen, S. Wang, and J. Wang. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable* and Secure Computing, 19:2244–2258, 2018. URL https://api.semanticscholar.org/ CorpusID:49869471.
- [31] Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu, and H. Jin. Vuldeelocator: A deep learning-based fine-grained vulnerability detector. *IEEE Transactions on Dependable and Secure Computing*, 19:2821–2837, 2020. URL https://api.semanticscholar.org/CorpusID:210064554.
- [32] S. Lipp, S. Banescu, and A. Pretschner. An empirical study on the effectiveness of static c code analyzers for vulnerability detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 544–555, 2022.
- [33] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee. Merlin: Specification inference for explicit information flow problems. *ACM Sigplan Notices*, 44(6):75–86, 2009.
- [34] R. Pan, A. R. Ibrahimzada, R. Krishna, D. Sankar, L. P. Wassi, M. Merler, B. Sobolev, R. Pavuluri, S. Sinha, and R. Jabbarvand. Lost in translation: A study of bugs introduced by large language models while translating code. In 2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE), pages 866–866. IEEE Computer Society, 2024.
- [35] I. A. A. Ranking. Finding patterns in static analysis alerts. In *Proceedings of the 11th working conference on mining software repositories*. Citeseer, 2014.
- [36] Semgrep. The semgrep platform. https://semgrep.dev/, 2023.
- [37] Y. Smaragdakis and M. Bravenboer. Using datalog for fast and easy program analysis. In *International Datalog 2.0 Workshop*, pages 245–251. Springer, 2010.
- [38] Snyk.io, 2024. https://snyk.io.
- [39] SonarQube, 2024. https://www.sonarsource.com/products/sonarqube.
- [40] B. Steenhoek, H. Gao, and W. Le. Dataflow analysis-inspired deep learning for efficient vulnerability detection, 2023.

- [41] B. Steenhoek, M. M. Rahman, M. K. Roy, M. S. Alam, E. T. Barr, and W. Le. A comprehensive study of the capabilities of large language models for vulnerability detection. *arXiv* preprint *arXiv*:2403.17218, 2024.
- [42] SWE Agent, 2024. https://swe-agent.com.
- [43] C. S. Xia and L. Zhang. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 959–971, 2022.
- [44] C. S. Xia, Y. Wei, and L. Zhang. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023)*. Association for Computing Machinery, 2023.
- [45] C. S. Xia, M. Paltenghi, J. Le Tian, M. Pradel, and L. Zhang. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.
- [46] A. Z. Yang, R. Martins, C. L. Goues, and V. J. Hellendoorn. Large language models for test-free fault localization. *arXiv preprint arXiv:2310.01726*, 2023.
- [47] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury. Autocoderover: Autonomous program improvement. *arXiv preprint arXiv:2404.05427*, 2024.
- [48] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Neural Information Processing Systems*, 2019. URL https://api.semanticscholar.org/CorpusID: 202539112.

A Implementation Details of IRIS

A.1 Selecting Candidate Specifications

While extracting external APIs, we filter out commonly-used Java libraries that are unlikely to contain any potential sources or sinks. Such libraries include testing libraries like JUnit and Hamcrest or mocking libraries like Mockito. While we filter out methods that are defined in the project, we specifically allow methods that are inherited from an external class or interface. An example is the getResource method of the generic class Class in java.lang package, which takes a path as a string and accesses a file in the module. Many projects commonly inherit this class and use this method. If the input path is unchecked, it may lead to a Path-Traversal vulnerability if the path accesses resources outside the given module. Hence, detecting such API usages is crucial.

Taint sources are typically values returned by methods that obtain inputs from external sources, such as response of an HTTP request or a command line argument. Hence, we select external APIs that have a "non-void" return type as candidate sources. Another type of taint sources are commonly seen in Java libraries. When used by downstream libraries, tainted information maybe passed into the library through function calls. Therefore, we also collect the formal parameters for public internal function as source candidates. Due to the excessive amount of such candidates, we pose a further constraint that the public internal function must be directly invoked by a unit test case within the same repository. Here, the test cases are identified by checking whether the residing file path has src/test within it.

On the other hand, taint sinks are typically arguments to an external API. This involves *explicit* arguments, such as the command argument passed to Runtime.exec(String command) method, and *implicit* this argument to non-static functions, such as the file variable in the function call file.delete(). This is the only type of sink that we consider within IRIS.

We note that this is not the entire story as there might be other kinds of sources and sinks. Other types of source candidates include the formal parameter of protected but overridden internal functions (the req parameter in protected HTTPServeletResponse doGet(HTTPServeletRequest req)), arguments to an impure external function (the buffer argument to void read(byte[] buffer, int size)), etc. Sink candidates include the return value of public facing functions, thrown exceptions, and even static methods without any parameter (System.exit()). Due to the complexity, we do not tackle such kind of sources of sinks in this work. However, we plan to explore further in future works.

A.2 LLM Prompts for Specification Inference

There are two prompts that we use to query LLM for specification inference. The first one is used to label external APIs as either sources or sinks, illustrated in Listing 1. At a high level, this is a classification task that classifies each API into one of {Source, Sink, Taint-Propagator, None}. As shown in the listing, the system prompt involves general instruction about the task and the expected output format, which is JSON. In the user prompt, we give the description of CWE, since the source and sink specifications of external APIs are dependent on the CWE. We additionally give few-shot examples that cover both sources and sinks for the given CWE. At the end, we list out a batch of methods akin to the format of CSV. Notably for sink specifications, we expect the LLM to give extra information about which exact argument to be considered as the sink. This include explicit arguments as well as the implicit this argument. We also note that while taint-propagators are included in the prompt, we do not actually use it in the subsequent stages of IRIS. Primarily, the notion of taint-propagator is to help LLMs differentiate between sinks and summary models, which are sometimes mistakened as sinks. In general, we find the prompt to serve the purpose well.

The second prompt, depicted in Listing 2, is used to label the formal parameters of internal APIs as sources. Since we are analyzing internal API, the information such as project README and function documentations are commonly available. The goal is to find whether this internal API might be invoked by a downstream library with a malicious input passed to this formal parameter. This information is not CWE specific, hence no CWE information is included in this prompt.

We hypothesize that since LLMs are pre-trained on internet-scale data, they have knowledge about the behavior of widely used libraries and their APIs. Hence, it is natural to ask whether LLMs can be

```
potential taint sources, sinks, or APIs that propagate taints. Taint sources
      are values that an attacker can use for unauthorized and malicious operations
      when interacting with the system. Taint source APIs usually return strings or
      custom object types. Setter methods are typically NOT taint sources. Taint
      sinks are program points that can use tainted data in an unsafe way, which
      directly exposes vulnerability under attack. Taint propagators carry tainted
      information from input to the output without sanitization, and typically have
      non-primitive input and outputs. Return the result as a json list with each
      object in the format:
  { "package": <package name>,
    "class": <class name>,
    "method": <method name>,
    "signature": <signature of the method>,
    "sink_args": <list of arguments or 'this'; empty if the API is not sink>,
    "type": <"source", "sink", or "taint-propagator"> }
  DO NOT OUTPUT ANYTHING OTHER THAN JSON.
  User: [CWE_LONG_DESCRIPTION]
  Some example source/sink/taint-propagator methods are:
15
  [CWE_SOURCE_SINK_EXAMPLES]
18 Among the following methods, \
  assuming that the arguments passed to the given function is malicious, \
19
  what are the functions that are potential source, sink, or taint-propagators to [
      CWE_TITLE] attack (CWE-[CWE_ID])?
 Package, Class, Method, Signature
23 [Package1], [Class1], [Method1], [Signature1]
[Package2], [Class2], [Method2], [Signature2]
  [...]
```

System: You are a security expert. You are given a list of APIs to be labeled as

Listing 1: LLM prompt for labelling external APIs as sources or sinks

used to identify APIs that are relevant as sources or sinks for any vulnerability class. If successful, LLMs can alleviate manual effort, and drastically improve the effectiveness of static analysis tools.

A.3 CodeQL Queries for Static Analysis

Listing 3 presents our CodeQL query for Path Traversal vulnerability (CWE 22). In Listing 3, Lines 10-29 describe a taint analysis configuration that describes which nodes in the data flow graph should be considered as sources or sinks. Here, Line 12 specifies our custom predicate isLLMDetectedSource that checks whether the method called is taint source based on our specifications. Similarly, our predicates isLLMDetectedSink checks whether the node is a taint sink based on our specifications. Line 16 checks if a method call or method argument node is a taint sink based on our specifications. We generate the source and sink specifications as predicates in QL file as shown in Listings 4 and 5 respectively. Given a taint configuration and the source and sink specifications, CodeQL can automatically perform taint analysis on a given project.

We use template to convert LLM inferred specifications into CodeQL queries. There are three kinds of queries:

- 1. a formal parameter of an internal function as a source;
- 2. the return value of an external function as a source;
- 3. an argument to an external function as a sink.

```
System: You are a security expert. You are given a list of APIs implemented in
      established Java libraries, and you need to identify whether some of these APIs
       could be potentially invoked by downstream libraries with malicious end-user (
      not programmer) inputs. For instance, functions that deserialize or parse
      inputs might be used by downstream libraries and would need to add sanitization
       for malicious user inputs. On the other hand, functions like HTTP request
      handlers are typically final and won't be called by a downstream package.
      Utility functions that are not related to the primary purpose of the package
      should also be ignored. Return the result as a json list with each object in
      the format:
  { "package": <package name>,
    "class": <class name>,
    "method": <method name>,
    "signature": <signature>,
    "tainted_input": <a list of argument names that are potentially tainted> }
  In the result list, only keep the functions that might be used by downstream
      libraries and is potentially invoked with malicious end-user inputs. Do not
      output anything other than JSON.
  User: You are analyzing the Java package [PROJECT_AUTHOR]/[PROJECT_NAME]. Here is
      the package summary:
  [PROJECT_README_SUMMARY]
15
  Please look at the following public methods in the library and their documentations
      (if present). What are the most important functions that look like can be
      invoked by a downstream Java package that is dependent on [PROJECT_NAME], and
      that the function can be called with potentially malicious end-user inputs? If
      the package does not seem to be a library, just return empty list as the result
      . Utility functions that are not related to the primary purpose of the package
      should also be ignored.
Package, Class, Method, Doc
  [Package1], [Class1], [Method1], [Documentation1]
19
  [Package2], [Class2], [Method2], [Documentation2]
```

Listing 2: LLM prompt for labeling formal parameters of internal APIs as sources.

Example queries for the two kinds of sources are specified in Listing 4, while the example query for the sink is illustrated in Listing 5. As shown in the listings, we not only match on function package, class, and name, but also match on individual arguments or parameters. Moreover, our query handles generic functions or function in generic classes through the getSourceDeclaration() predicate provided by CodeQL. Notably, when the number of inferred specifications is too large, we will split the single predicate into multiple hierarchical ones, improving the CodeQL performance.

A.4 Visualization of Metrics

We provide a visualization of our *VulDetected* metric in Fig. 7. For evaluation, we assume that the label for a project P is provided as a set of crucial program points $\mathbf{V}_{\text{fix}}^P = \{V_1, \dots, V_n\}$ where the vulnerable paths should pass through. In practice, these are typically the patched methods that can be collected from each vulnerability report. As illustrated in Fig. 7, if at least one detected vulnerable path passes through a fixed location for the given vulnerability, then we consider the vulnerability detected. Suppose $Paths_P$ is the set of detected and/or filtered paths from prior stages, the metrics, including #Vuls the total number of detected vulnerabilities across a set of projects, is

```
import java
  // other imports ...
  import MySources
  import MySinks
   * A taint-tracking configuration for tracking flow from remote sources to the
   * creation of a path.
  module MyTaintedPathConfig implements DataFlow::ConfigSig {
    predicate isSource(DataFlow::Node source) {
      isLLMDetectedSource(source)
13
14
    predicate isSink(DataFlow::Node sink) {
15
      isLLMDetectedSink(sink)
16
17
18
    predicate isBarrier(DataFlow::Node sanitizer) {
19
20
      sanitizer.getType() instanceof BoxedType or
21
      sanitizer.getType() instanceof PrimitiveType or
      sanitizer.getType() instanceof NumberType or
23
      sanitizer instanceof PathInjectionSanitizer
24
26
    predicate isAdditionalFlowStep(DataFlow::Node n1, DataFlow::Node n2) {
27
      isLLMDetectedStep(n1, n2)
    }
28
29
  }
30
  /** Tracks flow from remote sources to the creation of a path. */
32 module MyTaintedPathFlow = TaintTracking::Global<MyTaintedPathConfig>;
34 from MyTaintedPathFlow::PathNode source, MyTaintedPathFlow::PathNode sink
  where MyTaintedPathFlow::flowPath(source, sink)
    getReportingNode(sink.getNode()),
37
38
    source,
    sink,
    "This path depends on a $0.",
    source.getNode(),
    sourceType(source.getNode())
```

Listing 3: QL Script for Detecting Vulnerabilities for Path Traversal (CWE 22)

formally defined as:

```
VulDetected(Paths_P, \mathbf{V}_{\mathrm{fix}}^P) = \mathbb{1}_{|\{Path \in Paths_P \mid Path \cap \mathbf{V}_{\mathrm{fix}} \neq \emptyset\}| > 0}
\#Vuls(P_1, \dots, P_n) = \sum_i VulDetected(Paths_{P_i}, \mathbf{V}_{\mathrm{fix}}^{P_i})
```

A.5 Discussion of Precision Metrics

In contrast to *VulDetected*, computing the precision of the results is more challenging. For instance, even if a detected code path does not intersect with a fixed file or method, it may actually point to a true vulnerability (e.g., a different CVE in the same version) in the project. Moreover, even if there is a Hence, manual analysis is required to compute precision.

```
predicate isLLMDetectedSource(DataFlow::Node src) {
      // Sources: Return value from external APIx
          src.asExpr().(Call).getCallee().getName() = "getName" and
          src.asExpr().(Call).getCallee().getDeclaringType().getSourceDeclaration().
      hasQualifiedName("java.util.zip", "ZipEntry")
      or
      // Sources: Function formal parameters of internal API
      exists(Parameter p |
          src.asParameter() = p and
          p.getCallable().getName() = "setUserName" and
          p.getCallable().getDeclaringType().getSourceDeclaration().hasQualifiedName("
      org.apache.dolphinscheduler.dao.entity", "DqRule") and
          ( p.getName() = "userName" )
15
16
17
  }
```

Listing 4: QL Predicates for Source Specifications

```
predicate isLLMDetectedSink(DataFlow::Node snk) {
    exists(Call c |
        c.getCallee().getName() = "createTempFile" and
        c.getCallee().getDeclaringType().getSourceDeclaration().hasQualifiedName("
    java.io", "File") and
        ( c.getArgument(0) = snk.asExpr().(Argument) )
    )
    or
    ...
}
```

Listing 5: QL Predicates for Sink Specification

Step	CWE-22	CWE-78	CWE-79	CWE-94	Total
Initial CVEs	236	39	681	109	1065
W/ Github URL and Version	119	37	219	55	430
W/ Fix Commit	89	27	99	50	265
Compilable	56	17	50	26	149
Fixes in Java Code	56	16	25	47	144
Manual Validation	55	13	31	21	120

Table 4: Vulnerability Dataset Collection Statistics

B Additional Details of CWE-Bench-Java

B.1 Details of Dataset Extraction Process

Because we use CodeQL for static analysis, we further need to build each project for CodeQL to extract data flow graphs from the projects. To build each project, we need to determine the correct Java and Maven compiler versions. We developed a semi-automated script that tries to build each project with different combinations of Java and Maven versions. The fourth row in Table 4 presents the number of projects we were able to build successfully. Overall, this results in (*) 149 projects.

Finally, we manually check each fix commit and validate whether the commit actually contains a fix to the given CVE in a Java file. For instance, we found that in some cases the fix is in files written in other languages (such as Scala or JSP). While code written in other languages may flow to the Java components in the project during runtime or via compilation, it is not possible to correctly

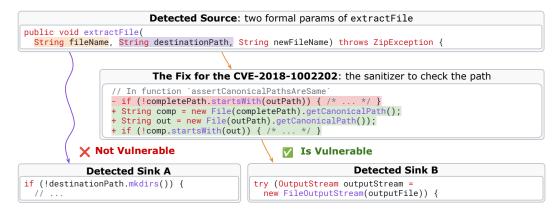


Figure 7: A visualization of our metrics used for vulnerability detection, where the snippets are adapted from Zip4j v2.11.5-2, with slight changes for clearer presentation. While both sinks are potential causes of path-traversal (CWE-22), only the dataflow path on the right passes through the fixed sanitizer function. Therefore, we consider only the path on the right as a vulnerability.

determine if static analysis can correctly detect such a vulnerability. Hence, we exclude such CVEs. Further, we exclude cases where the vulnerability was in a dependency and the fix was just a version upgrade or if the vulnerability was mis-classified. Finally, we end up with (*) 120 projects that we evaluate with IRIS. For this task, we divide the CVEs among two co-authors of the project, who independently validate each case. The co-authors cross-check each other's results and discuss together to come up with the final list of projects.

The closest dataset to ours, in terms of features, is the Java dataset curated by Li et al. [28], containing 165 CVEs. While we initially considered using their dataset for our work, we found several issues. First, their dataset does not come with build scripts, which makes it hard to automatically run each project with CodeQL. Second, their dataset only has few CVEs for all but one CWE, which makes it difficult to thoroughly analyze a tool for different vulnerability classes. Finally, they do not provide any automated scripts to curate more CVEs. Hence, we curated our own dataset and our framework also allows to easily extend to more vulnerability classes.

B.2 Comparison of our CWE-Bench-Java with Existing Vulnerability Datasets

We compare CWE-Bench-Java with existing datasets for vulnerability detection in Java, C, and C++ codebases, on the following criteria:

- 1. **CVE Metadata**: whether CVE Metadata is contained in the dataset;
- 2. **Real-World**: whether the dataset contains real-world projects;
- 3. **Fix Locations**: whether the dataset contains fix information about the vulnerabilities;
- 4. **Compilable**: whether the dataset ensures that the projects are end-to-end and automatically compilable;
- 5. **Vetted**: whether the vulnerability in the dataset is manually verified and confirmed.

As shown in Table 5, compared to existing datsets, CWE-Bench-Java, is the only one that checks every criterion. This underscores the significance of our new dataset.

C Evaluation Details

C.1 Experimental Settings

We select two closed-source LLMs from OpenAI: GPT 4 (gpt-4-0125-preview) and GPT 3.5 (gpt-3.5-turbo-0125) for our evaluation. GPT 4 and GPT 3.5 queries used in the paper are performed through OpenAI API during April and May of 2024.

Table 5: Comparison of CWE-Bench-Java with existing vulnerability datasets

					-	
Dataset	Languages	CVE Metadata	Real-World	Fix Locations	Compilable	Vetted
BigVul	C/C++	✓	/	×	X	X
Reveal	C/C++	×	1	×	X	X
CVEFixes	C/C++, Java,	✓	1	✓	✓	X
DiverseVul	C/C++	×	1	✓	X	X
DeepVD	C/C++	X	1	×	X	X
Juliet	C++, Java	X	X	✓	1	1
Li et al. [28]	Java	✓	1	✓	X	/
SVEN [17]	C++	×	✓	✓	×	1
Our Dataset	Java	✓	/	/	✓	1

We also select instruction-tuned versions of six state-of-the-art open-source LLMs via huggingface API: Llama 3 8B and 70B, DeepSeekCoder 7B and 33B, Mistral 7B, and Gemma 7B. To run the open-source LLMs we use two groups of machines: a 2.50GHz Intel Xeon machine, with 40 CPUs, four GeForce RTX 2080 Ti GPUs, and 750GB RAM, and another 3.00GHz Intel Xeon machine with 48 CPUs, 8 A100s, and 1.5T RAM.

We use CodeQL version 2.15.3 as the backbone of our static analysis.

C.2 CodeQL Baseline

For baseline comparison with CodeQL, we use the built-in Security queries specifically designed for each CWE that comes with CodeQL 2.15.3. Note that there are multiple security queries for each CWE, and each produce alarms of different levels (error, warning, and recommendation). For each CWE, we take the union of alerts generated by all queries and do not differentiate between alarms of different levels. For instance, there are 3 queries from CodeQL for detecting CWE-22 vulnerabilities, namely TaintedPath, TaintedPathLocal, and ZipSlip. While TaintedPath and ZipSlip produce error level alarms, TaintedPathLocal produces only alarm recommendations. To CodeQL's advantage, all alarms are treated equally in our comparisons.

C.3 Hyper-Parameters and Few-Shot Examples

During IRIS, we have 2 prompts that are used to label external and internal APIs. Recall that the prompts contain batched APIs. We use batch size of 20 and 30 for internal and external, respectively. In terms of few-shot examples passed to labeling external APIs, we use 4 examples for CWE-22, 3 examples for CWE-78, 3 examples for CWE-79, and 3 examples for CWE-94. We use a temperature of 0, maximum tokens to 2048, and top-p of 1 for inference with all the LLMs. For GPT 3.5 and GPT 4, we also fix a seed to mitigate randomness as much as possible.

C.4 Details of Selected LLMs

We include the versions of selected LLMs in Table 6.

Table 6: Selected LLM Versions

LLM Version and Size	Model ID
GPT 4	gpt-4-0125-preview
GPT 3.5	gpt-3.5-turbo-0125
Llama 3 8B	meta-llama/Meta-Llama-3-8B-Instruct
Llama 3 70B	meta-llama/Meta-Llama-3-70B-Instruct
DeepSeekCoder 33B	deepseek-ai/deepseek-coder-33b-instruct
DeepSeekCoder 7B	deepseek-ai/deepseek-coder-7b-instruct
Gemma 7B	google/gemma-1.1-7b-it
Mistral 7B	mistralai/Mistral-7B-Instruct-v0.2

C.5 Ablation Studies

Table 7 presents additional results when using either only the source or sink specification from an LLM in IRIS. For this experiment, we only use the results with GPT 4 for comparison. Each row present the average number of paths and number of detected vulnerabilities per CWE, for each combination. We observe that omitting either source or sink specifications inferred by GPT 4 causes a drastic reduction in overall recall. For CWEs 22, and 78, inferring additional source specifications seems to be more beneficial for improving recall (row 2). This is because for CWE 22 (path traversal), the sinks are typically File related methods (e.g., FileOutputStream) and for CWE 78, sinks are usually system calls (e.g., Runtime.exec) – many of which are available in CodeQL's specifications. Howver, the potential sources for these CWEs can be more diverse and hence missing in CodeQL. On the other hand, for CWE 79 (Cross-Site Scripting) and CWE 94 (Code Injection), the sinks can vary widely including XML/HTML manipulation APIs or deserialization-related APIs, many of which are missing in existing specifications. Hence, sink specifications are more useful for these two CWEs.

Table 7: Ablation on LLM inferred source and sink specifications (CodeQL vs GPT 4)

Metrics		#Detected Vuls							
CWE	22	2	78	3	79	9	94	1	
Combination	#Paths	#Vuls	#Paths	#Vuls	#Paths	#Vuls	#Paths	#Vuls	Total
$Src_{CodeQL} + Snk_{CodeQL}$	172	22	3	1	31	4	1	0	27 (\ 42)
$Src_{GPT4} + Snk_{CodeQL}$	202	28	7	3	43	5	0	0	36 (\ 33)
$Src_{CodeQL} + Snk_{GPT4}$	82	10	57	1	335	9	369	4	24 (\ 45)
$Src_{GPT4} + Snk_{GPT4}$	760	34	529	7	2139	17	1590	11	69

C.6 Statistics of Unique and Recurring Specifications

Table 8: Unique Source and Sink Specifications Across All Projects in CWE-Bench-Java.

CWE	22	78	79	94
#Unique Sources	1348	899	598	810
#Unique Sinks	1069	575	514	1281

Table 9: Recurring Source and Sink Specifications in CWE-Bench-Java.

CWE	22	78	79	94
#Recurring Sources	908	232	1118	626
#Recurring Sinks	919	201	911	961

Continuous taint specification inference is necessary. Our results show that there is a high number of both unique and recurring sources and sinks. Table 8 presents the number of inferred source and sink specifications that occur only in a single project in CWE-Bench-Java, whereas Table 9 presents the specifications that occur in at least two projects. This indicates that even if previously inferred specifications are useful, a significant number of new relevant APIs still remain and need to be labeled for effective vulnerability detection. This observation strongly motivates the design of IRIS that infers these specifications *on-the-fly* for each project via LLMs, instead of relying on a fixed corpus of specifications like CodeQL.

C.7 Analysis Runtime

We include the full table containing statistics to provide more details about projects and our analysis (Table 10). For each project, we present its corresponding CWE ID, the lines-of-code (SLOC), the time it takes to run the full analysis, the number candidate APIs and the number of labeled source and sinks by Llama 3 8B. We also color code cells of interest: For SLOC, we mark a cell as red if >1M; yellow if >100k. For Time, we mark a cell as red if \geq 1h; yellow if \geq 5m. For the number of candidates, we mark a cell as red if >10k. Lastly for sources and sinks, we mark a cell as red if the number is larger than 200.

Table 10: Details of analysis runtime, candidates, and inferred sources and sinks for all projects (Llama $3\,8B$)

CWE-ID	Project	SLOC	Time	#Candidates	#Sources	#Sinks
22	DSpace	218.2K	15s	3.61K	162	217
22	spark	10.7K	1m	679	35	27
22	spark	9.77K	57s	598	33	22
22	wildfly	496.28K	4m	14.13K	457	425
22	vertx-web	51.01K	1m	2.06K	80	77
22	camel	1.16M	8m	293	22	9
22	hutool	135.34K	4m	6.17K	115	211
22	tika	106.3K	2m	3.84K	277	177
22	retrofit	19.28K	1m	880	28	13
22	jspwiki	149.45K	1m	1.83K	62	80
22	camel	1.21M	11m	4.43K	53	80
22	tapestry-5	160.06K	1m	3.04K	91	66
22	spring-cloud-co	18.56K	1m	1.16K	40	64
22	spring-cloud-co	18.44K	59s	1.16K	40	64
22	rocketmq	94.64K	1m	2.78K	28	54
22	mpxj	181.55K	1m	1.6K	37	43
22	flink	1.14M	2h	5.16K	39	61
22	java	1M	2m	8.04K	96	41
22	commons-io	29.24K	58s	1.07K	12	47
22	karaf	135.22K	1m	5.43K	150	210
22	james-project	434.32K	4m	14.58K	209	226
22	vertx-web	49.28K	1m	2.36K	83	96
22	esapi-java-lega	35.26K	59s	1.48K	43	67
22	xwiki-commons	103.05K	1m	3.76K	104	137
22	zip4j	16.78K	58s	532	6	34
22		5.19K	51s	327	11	20
22	one-java-agent myfaces	161.02K	318 1m	2.4K	68	44
22	•				66	93
22	undertow	86.03K	1m 1m	2.58K	47	93 66
22	DependencyCheck	28.57K	51s	1.23K		
22	plexus-archiver	13.04K		573	34	47 47
	plexus-archiver	13.04K	51s	573	34	47
22	zt-zip	6.64K	52s	337	14	31
22	curekit	511	43s	73	2	4
22	aws-sdk-java	7.72M	38m	12K	62	65
22	venice	115.44K	1m	2.27K	36	79
22	DSpace	237.33K	1m	3.67K	179	233
22	Payara	1.12M	7m	16.05K	379	427
22	DSpace	237.33K	1m	3.67K	179	233
22	goomph	12.68K	59s	1.12K	35	111
22	dolphinschedule	90.69K	1m	3.36K	65	92
22	dolphinschedule	91.94K	1m	3.4K	65	92
22	testng	95.53K	1m	2.08K	33	73
22	uima-uimaj	226.81K	2m	5.66K	103	176
22	keycloak	614.82K	12m	13.34K	325	252
22	glassfish	1.19M	5m	12.19K	293	346
22	graylog2-server	382K	4m	13.3K	227	171
22	mina-sshd	130.14K	1m	3.64K	52	120
22	shiro	38.68K	1m	1.5K	41	42
22	plexus-archiver	15.51K	57s	666	37	56
22	plexus-utils	23.3K	58s	754	16	36
22	yames	693.6K	2m	11K	98	113
	•	693.6K	2m	11K	98	113
22	vames	095.01				
22 22	yamcs shiro					
22 22 22	yamcs shiro sling-org-apach	38.94K 8.34K	1m 54s	1.53K 695	41 28	43 25

78	xstream	59.79K	1m	1.64K	107	42
78	xstream	52.25K	1m	1.64K	107	43
78	docker-commons-	2.79K	54s	362	25	20
78	workflow-cps-pl	17.02K	1m	1.38K	72	61
78	workflow-cps-gl	4.31K	55s	523	40	38
78	workflow-multib	3.45K	53s	500	30	30
78	activemq	442.42K	4m	6.34K	234	192
78	plexus-utils	22.76K	1m	714	34	17
78 78	git-client-plug	16.41K	1m	1.06K	83	50
78 70	perfecto-plugin	667	54s	107	5	10
78 70	nifi	915.95K	11m	22.44K	894	614
78 70	script-security	8.17K	1m	678	40	46
79 79	antisamy	6.38K 6.38K	57s 56s	381 381	42 42	33 33
79 79	antisamy įspwiki	149.33K	30s 1m	1.84K	156	33 110
79 79	jspwiki	149.33K 149.33K	1m	1.84K	156	110
79	jspwiki	149.33K	1m	1.84K	156	110
79	jspwiki	157.09K	1m	1.85K	157	110
79	hibernate-valid	93.6K	1m	2.06K	79	57
79	cxf	798.53K	1h	16.54K	821	756
79	xxl-job	9.32K	60s	540	42	41
79	json-sanitizer	1.47K	52s	67	4	5
79	hawkbit	112.09K	1m	4.07K	144	151
79	nacos	203.78K	2m	4.08K	201	139
79	antisamy	4.93K	1m	362	43	34
79	esapi-java-lega	35.26K	1m	1.48K	107	85
79	antisamy	5.14K	1m	377	44	36
79	jolokia	29.97K	1m	1.66K	117	97
79 70	keycloak	60.6K	1m	2.1K	170	136
79 79	cxf	722.83K 1.37K	15m 55s	15.09K	766 4	710
79 79	sling-org-apach DSpace	237.33K	2m	136 3.67K	347	320
79 79	keycloak	615.6K	3h	13.37K	606	461
79	keycloak	615.6K	3h	13.37K	606	461
79	xwiki-commons	105.92K	1m	3.94K	244	151
79	xwiki-commons	105.94K	1m	3.94K	244	151
79	xwiki-rendering	97.01K	1m	1.22K	73	92
79	xwiki-commons	106.87K	1m	3.99K	254	161
79	jspwiki	158.7K	1m	2.22K	176	126
79	keycloak	617.15K	4h	14.04K	643	479
79	xwiki-commons	107.09K	1m	3.03K	209	143
79	jstachio	53.02K	54s	792	40	46
79	xwiki-rendering	97.63K	1m	1.24K	74	92
94	spring-security	43.9K	1m	1.83K	120	176
94	xstream	52.25K	1m	1.64K	111	145
94	cron-utils	13.08K	1m	476	13	26
94	struts	160.51K	12m	4.39K	301	357
94	activemq	547.68K	1h	7.55K	370	607
94 94	spring-framewor	666.11K	45m	17.71K 2.01K	688	846
94 94	spring-cloud-ga dubbo	25.56K 175.63K	1m 2m	6.73K	130 342	153 383
94	incubator-dubbo	96.35K	2m	3.68K	194	255
94	spring-security	57.34K	1m	2.43K	192	234
94	kubernetes-clie	806.35K	3m	2.33K	93	130
94	commons-text	24.87K	1m	962	40	47
94	ff4j	46.21K	1m	2.39K	133	274
94	spring-boot-adm	18.29K	1m	1.83K	92	157
94	sqlite-jdbc	17.71K	59s	732	50	74
94	nifi	993.76K	25m	57	2	11

94	rocketmq	108.39K	2m	3.4K	117	164
94	nifi	1.01M	27m	261	27	24
94	rocketmq	197.78K	2m	6.28K	205	252
94	dolphinschedule	154.95K	4m	5.78K	229	353
94	dolphinschedule	154.95K	4m	5.78K	229	353