WhyFlow: Interrogative Debugger for Sensemaking Taint Analysis

Burak Yetistiren
Computer Science
UCLA
Los Angeles, CA, USA
burak@cs.ucla.edu

Hong Jin Kang
University of Sydney
Sydney, Australia
hongjin.kang@sydney.edu.au

Miryung Kim
Computer Science
UCLA
Los Angeles, CA, USA
miryung@cs.ucla.edu

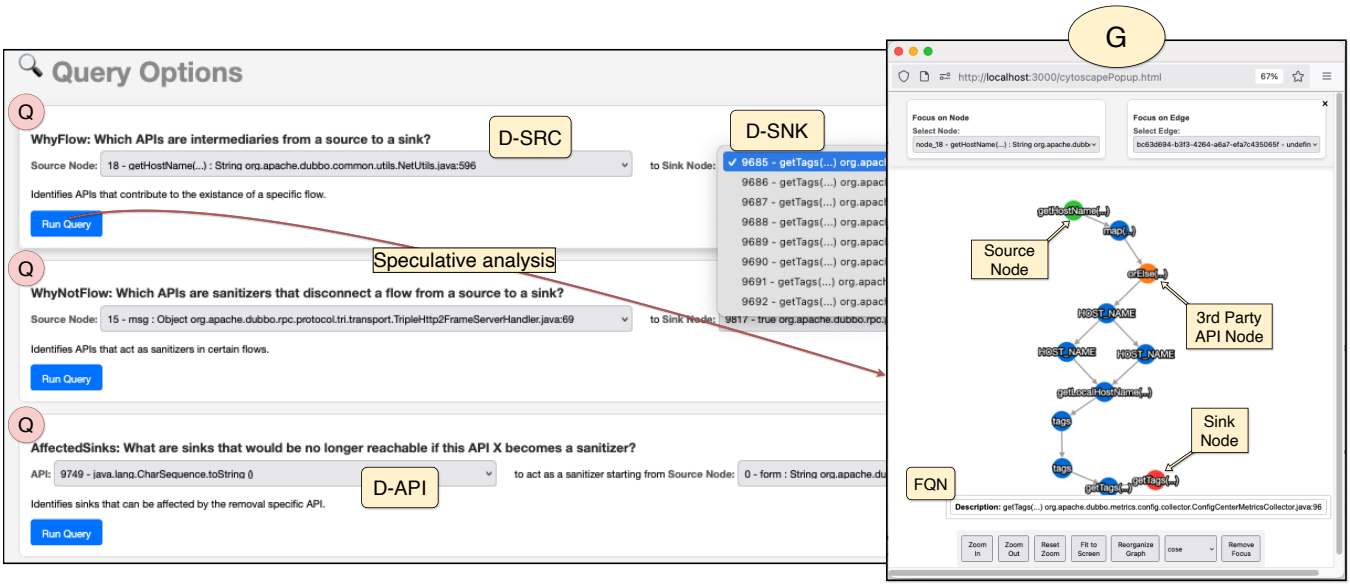

Figure 1: WhyFlow supports interrogative debugging by enabling a user to ask *why*, *why-not*, and *what-if* questions about taint analysis. A user can select from templated queries (shown in Q) and contextualize their inquiry with respect to a specific source, sink, and third-party library model using a drop-down menu, shown in (D-SRC), (D-SNK), and (D-API). Once configured, a background question-and-answer analysis is conducted to help a user with their sensemaking process of taint analysis results. To aid in their sensemaking process about global connectivity, permissible and impermissible data flows, a user can see the result in a graph view (G) with color-coded annotation.

Abstract

Taint analysis is a security analysis technique used to track the flow of potentially dangerous data through an application and its dependent libraries. Investigating why certain unexpected flows appear and why expected flows are missing is an important sensemaking process during end-user taint analysis. Existing taint analysis tools often do not provide this end-user debugging capability, where developers can ask *why*, *why-not*, and *what-if* questions about dataflows and reason about the impact of configuring sources and sinks, and models of third-party libraries that abstract permissible and impermissible data flows. Furthermore, the tree-view or list-view used in existing taint analyzer visualizations makes it difficult to reason about the global impact on connectivity between multiple sources and sinks.

Inspired by the insight that *sensemaking* tool-generated results can be significantly improved by a QA inquiry process, we propose WhyFlow, the first end-user question-answer style debugging interface for taint analysis. It enables a user to ask *why*, *why-not*, and *what-if* questions to investigate the existence of suspicious flows, the non-existence of expected flows, and the global impact of third-party library models. WhyFlow performs *speculative what-if* analysis, to help a user in debugging how different connectivity assumptions affect overall results. A user study with 12 participants shows that participants using WhyFlow achieved 21% higher accuracy on average, compared to CodeQL. They also reported a 45% reduction in mental demand (NASA-TLX) and rated higher confidence in identifying relevant flows using WhyFlow. This shows WhyFlow's potential to significantly reduce sensemaking effort.

ICSE '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2025-3/26/04
https://doi.org/10.1145/3744916.3773145

CCS Concepts

• **Human-centered computing → HCI design and evaluation methods**.

Keywords

taint analysis, end-user debugging, interrogative debugging, sensemaking, speculative analysis, human-centered static analysis

ACM Reference Format:

Burak Yetistiren, Hong Jin Kang, and Miryung Kim. 2026. WhyFlow: Interrogative Debugger for Sensemaking Taint Analysis. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3773145

1 Introduction

To prevent security vulnerabilities, developers use taint analysis. This technique tracks potentially dangerous data as it moves through a program. Data from untrusted sources, like user input, is marked as "tainted." The analysis then monitors how this tainted data spreads. If it reaches a critical point in the code, known as a "taint sink," a warning is generated. Tainted data can be made safe by "sanitizers," which are functions that clean or encode the data. For example, the `java.net.URLEncoder.encode(String input, String encoding)` function escapes potentially harmful characters, preventing security issues. Since analyzing the entire program is usually impossible, practical taint analysis requires configuring simplified models of external libraries [14]. These models abstract the behavior of third-party libraries, allowing an analysis to track taint flow without analyzing the libraries' internals.

Taint analysis tools often come with default configurations, including pre-built *models of external libraries*. However, these default models, which are often automatically generated, can be inaccurate. This is because they rely on assumptions, which may be incorrect, about how libraries handle data flow. Incorrect configurations can lead to significant problems. For example, if a model incorrectly allows tainted data to flow through a library function that should sanitize it, the analysis will produce unexpected taint flows (i.e., false positives) [6, 39]. This means it will report potential vulnerabilities that do not actually exist, creating extra, unexpected warnings. Conversely, if a model incorrectly blocks the flow of tainted data when it should not, the analysis will produce missing flows (i.e., false negatives). This means it will miss real vulnerabilities, failing to report flows expected by the user.

We introduce WhyFlow, a novel end-user debugger that brings interactive, question-based sensemaking to taint analysis. Drawing inspiration from interrogative debugging principles, WhyFlow enables developers to ask *why*, *why not*, and *what if* questions about their analysis configuration. This helps developers make sense of the impact of configuration and models of third-party libraries on tool-generated warnings. This sensemaking process involves tracing the reported dataflows, understanding why specific warnings are generated, and how these warnings relate to the models of third-party libraries. WhyFlow focuses on debugging an end-user configuration instead of a taint analysis implementation.

WhyFlow provides six customizable question templates, which are then concretized by a user. A user can make sense of currently detected taint flows and hypothetical flows through interactive QA. Unlike state-of-the-art tools like CodeQL that rely on list views, WhyFlow visualizes these reported and hypothetical flows graphically, offering a more intuitive understanding of configuration changes. Below, we discuss two key features: *Inquiry-based Sensemaking* and *Visualization of Global Impact* in detail.

Inquiry-based Sensemaking. To effectively debug taint analysis results, developers need to investigate why specific dataflows are reported and why others are not. Unfortunately, many tools do not allow developers to interactively ask questions about these flows. Furthermore, to efficiently narrow down their analysis, developers must understand how modifying the models of external libraries impacts the permitted or blocked dataflows.

In the *Query Options* pane of WhyFlow (shown in Figure 1), users can select from pre-defined question templates. These templates allow users to investigate taint analysis results by asking specific questions about dataflows. To make these questions concrete, users specify the configuration they are interested in, including the source of the data, the destination (sink), and the models of third-party libraries involved.

For example, a user might choose the *why-flow* template, which asks, "Why is there a taint flow from a source to a sink?" They can then concretize this question by specifying a specific source and sink. For instance, they could ask, "Which third-party library models currently allow taint flows from the source `java.net.InetAddress.getHostName(...)` to the sink `org.apache.dubbo.metrics.model.ConfigCenterMetric.getTags(. . .)`?"

Similarly, a user could select the *why-not* template, which asks, "Why is there no taint flow from a source to a sink?" They can then specify the source and sink of interest. For example, they could ask, "Which third-party library models could potentially block taint flows from the source `msg:HttpRequest` to the sink `ErrorTypeAwareLogger.warn()`?" Alternatively, a user could also select the *what-if* template, which asks a speculative analysis question: "Which sinks would no longer be reachable if third-party library models were configured as a sanitizer from source to sink?"

Visualization of Global Impact. Once a user initiates a query within WhyFlow, specifying their target source, sink, and any relevant external API calls, WhyFlow generates an interactive graph visualization, providing a holistic view of the taint flow. This graph, as seen in the G pane of Figure 1, visually maps the program's taint flows as interconnected nodes, each representing a distinct stage in the taint flow. This visualization reveals the *global impact of configuration choices* through the network of connected nodes. To draw attention to the impact of configuration choices, WhyFlow employs a color-coded scheme: source nodes are highlighted in green, sink nodes in red, and external library nodes in orange. Templated queries are designed to illuminate how third-party library models can impact multiple sinks and multiple sources, making the global impact salient through visualization.

User study: We conducted a within-subject study with a factorial crossover involving 12 participants (graduate students and professional developers) who inspected taint-analysis warnings generated by CodeQL [5, 53]. Each participant answered eight questions per task (16 total) designed to reflect a realistic sensemaking process of tool-generated warnings They had to answer questions about the impact of third-party library models on taint analysis

results by identifying pass-through APIs calls, APIs that serve as sanitizers, etc. Participants increased the average question completion rate from 71% with the baseline CodeQL visualizer to 92% with WhyFlow, especially on the kinds of questions that require global reasoning of multiple taint flows. When using WhyFlow, users were more accurate in identifying multiple sinks affected by the same third-party library model (50% vs. 17%).

We measured cognitive load via NASA-TLX questions. We found that WhyFlow led to improvement on the participants' self-reported mental demand, stress, and success rate. Considering the technology acceptance model [36], participants rated WhyFlow higher on both *Confidence* (mean 4.3 vs. 2.1) and *Ease of Use* (4.3 vs. 1.9) than the baseline CodeQL. Qualitative analysis highlighted that color-coded graphs and template queries helped alleviate *analysis paralysis* [49]. Overall, these results underscore that WhyFlow substantially enhances users' ability to make sense of complex taint paths and reduce their mental workload through QA-based debugging.

In this paper, we make the following contributions:

(1) Configuring taint analysis is tricky, often leading to unexpected results. To help users understand and fix these issues, WhyFlow is the first interactive end-user debugger that shows how different user configuration choices and modeling of third-party libraries impact the taint analysis results.
(2) WhyFlow is equipped with template queries for *'why,' 'why not,'* and *'what if'* questions, which are automatically translated into logic queries; it enables users to concretize the template queries with concrete code names embedded in tool-generated warnings. WhyFlow makes it easier to examine the global impact of models, which are difficult to reason when viewing warnings in a list view, packaged with an existing taint analyzer.
(3) We conduct a within-subject user study with a factorial crossover design (12 participants) to assess WhyFlow's effectiveness in sensemaking CodeQL-generated taint analysis warnings [5, 53]. WhyFlow raises the average question completion rate from 71% to 92% and reduces mental demand (NASA-TLX) from 5.9 to 3.3, demonstrating clear improvement in the end-user debugging of taint analysis.
(4) Improvement in accuracy, reduction in cognitive load, improvement in confidence, and ease of use compared to CodeQL's visualizer is statistically significant. The additional follow-up study with two professionals corroborates this improvement in accuracy, confidence, and ease of use.

The remainder of this paper is organized as follows. Section 2 introduces a motivating example and the design goals. Section 3 presents our tool, WhyFlow. Section 4 provides the study design. Section 5 reports the user study's results. Section 6 discusses the implications of our findings and threats to validity. Section 8 presents related work. Section 9 concludes our paper.

2 Motivation

When using modern static taint analyzers such as CodeQL [53], developers struggle to identify why unexpected flows appear or why certain expected flows are missing. These tools rely on models of external libraries, which may not always match the developer's

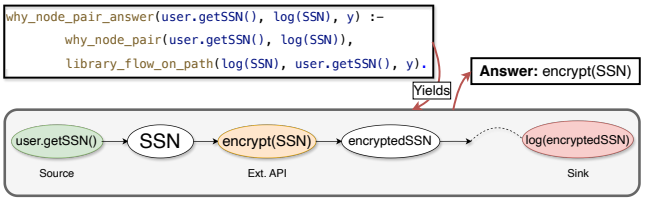

Figure 2: WhyFlow: "Why is there a taint flow from a source to a sink?"

expectations. This paper is concerned with this problem of *end-user debugging of taint analysis*—i.e., understanding the impact of user configuration choices. Two key end-user debugging challenges prevent developers from easily making sense of unexpected results.

First, traditional tools lack support for inquiry-based debugging, presenting only isolated warnings without contextual flow connections. This forces developers into manual tracing, which hinders their ability to answer questions they have during debugging.

Second, beyond isolated flow views, developers need support for reasoning about the model's impact on multiple dataflows. Current tools lack the ability to predict the ripple effects of model changes, forcing developers to manually piece together disparate flows.

Prior work introduced *WhyLine* [35] to let users ask *"why did"* or *"why didn't"* questions on a program trace, for debugging runtime errors; however, *"why and why-not"* questions remain unaddressed in the context of modern taint analysis. Developers want customizability over taint analysis [42], visual outputs of warnings [29], and the ability to inspect intermediate pass-through nodes of the analysis [35]. Yet, typical dataflow or taint analysis interfaces do not enable higher-level inquiries such as "*Why does data from source X reach sink Y?*" or "*Why is no warning raised for a known bug?*"

Table 1 shows six kinds of templated queries. In the following paragraphs, we detail the motivating scenario behind each query.

1. WhyFlow: Why is there a taint flow from a source X to a sink Y? Suppose that Alice sees an unexpected taint-analysis warning in CodeQL from a source `user.getSSN()` (untrusted data) to a sink `log(SSN)` (potential vulnerability) [29, 35, 42]. She suspects the culprit is an imprecise third-party library model; however, manually tracing a long chain of calls is overwhelming. WhyFlow query highlights the taint flow path from a source in green to a sink in red, displaying pass-through third-party API calls in orange, shown in Figure 2. Alice notices an intermediate API call `encrypt(SSN)`. She then discovers that this API is a sanitizer, and thus this warning should *not* have been reported and `encrypt()`'s model should be debugged.

2. WhyNotFlow: Why is there no taint flow from a source X to a sink Y? Suppose that Alice needs to investigate a missing taint flow (i.e., a bug arises at a sink, yet CodeQL does not issue a corresponding warning). She suspects that a third-party library is mistakenly modeled as a *sanitizer*. In Figure 3, the sensitive data travels from `user.getSSN()` through an external API: `format(SSN)` to `log(SSN)`. After investigating the intermediate flow steps, Alice finds that `format(SSN)` was erroneously modeled as a sanitizer.

Table 1: Template queries for *why*, *why-not* and *what-if* questions and corresponding English interpretation and logic queries.

Query Type	Plain-English Question	Logic Query Interpretation
WhyFlow	"Why is there a taint flow from a source X to a sink Y?"	"Which third-party library models (or assumptions) currently allow data to propagate from X to Y?"
WhyNotFlow	"Why is there *no* taint flow from a source X to a sink Y?"	"Which third-party library models (or assumptions) currently terminate the flow (e.g., sanitizers), where their model change could create a flow from a source X to a sink Y?"
AffectedSinks	"If we alter a third-party library Z's model, which sinks are affected?"	"Under a new assumption of treating a third-party library Z as a sanitizer, which previously reported sinks are no longer reachable from X?"
DivergentSinks	"Which third-party library X's model could influence multiple taint flows from the same source X?"	"What are the common third-party API nodes in multiple paths originating from X that *split* into multiple different sinks?"
DivergentSources	"Which third-party library X's model could influence multiple taint flows reaching the same sink Y?"	"What are the common third-party API nodes in multiple flow paths that eventually reach the same sink Y?"
GlobalImpact	"Which third-party library Z's model could have the largest global influence on dataflows from X to Y?"	"What is the frequency of each third-party API call appearing along all paths from X to Y in terms of the overall usage counts?"

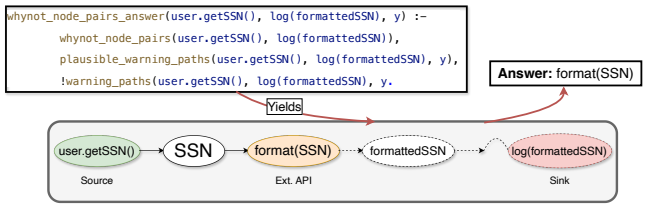

Figure 3: WhyNotFlow: "Why is there no taint flow from a source to a sink?"

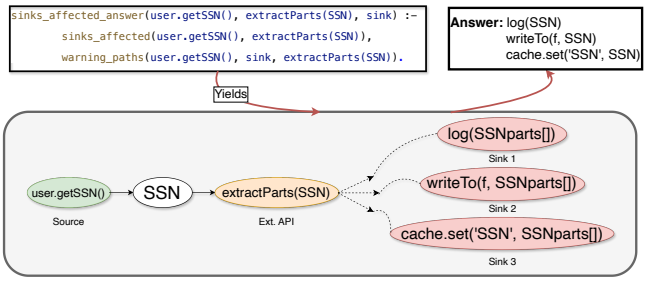

Figure 4: AffectedSinks: "If we alter a third-party library's model, which sinks are affected?

WhyNotFlow visually identifies which APIs are acting as sanitizers (dashed arrows), pinpointing which API model could be responsible for killing a flow. WhyNotFlow performs a speculative analysis by reasoning which taint flow path could have been *plausible* under a configuration where a sanitizer is instead modeled as a non-sanitizer. In Figure 3, the graphical view marks the arrow after the node for `format(SSN)` with a dashed line, indicating the flow currently does not exist, but would exist with a different model assumption.

3. AffectedSinks: If we alter a third-party library Z's model, which sinks are affected? Alice wants to reason about the global impact of updating a third-party library model [50]. Suppose that she sees an unexpected warning and considers marking a third-party library's API as a sanitizer. However, she is concerned this update might suppress other warnings. *Without* WhyFlow, she would need to sift through all reported warnings, manually trace each flow from the source, and check which sinks might become unreachable after her change—a time-consuming process. AffectedSinks automates this what-if analysis, immediately revealing all red sinks that would be "killed" (Figure 4).

4. DivergentSinks & 5. DivergentSources: Which third-party library model could influence multiple taint flows reaching the same sink (or originating from the same source)?

Suppose that Alice would like to know whether a known vulnerability reaching multiple *sinks* could be fixed at once [45, 50]. In a typical taint analyzer, it can be cumbersome to trace how a single piece of sensitive data, such as an `SSN`, propagates through multiple taint flows. Consequently, she may prefer to identify a common interception point instead of fixing one sink at a time. With DivergentSinks query, she can quickly locate a common point from the source that "splits" into multiple sinks. (see Figure 5).

To illustrate another scenario, imagine a single sensitive destination (sink) that receives data from multiple untrusted sources. Using the DivergentSources query, Alice can readily identify if these diverse sources converge towards the same vulnerable point. This allows her to pinpoint the specific model responsible for the convergence, i.e., the 'culprit model.' Unlike traditional list-based warning views, WhyFlow visually highlights where taint paths intersect, making it much easier to pinpoint the culprit model.

6. GlobalImpact: Which third-party library's model could have the largest global influence on multiple flows from a source to a sink? When multiple models could explain a false or

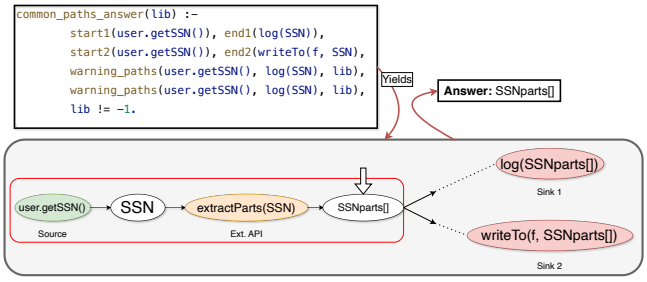

Figure 5: DivergentSinks: "Which third-party library model could influence multiple taint flows from the same source?"

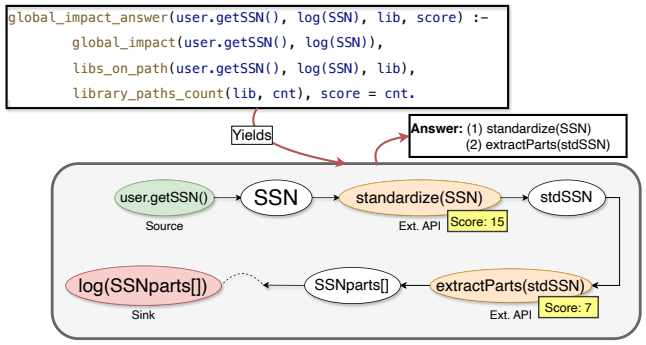

Figure 6: GlobalImpact: "Which third-party library model could have the largest global influence on dataflows from a source to a sink?"

missing warning, developers like Alice often prefer a conservative fix that impacts the fewest flows [41]. *Without* WhyFlow, identifying the frequency of each API in *all* taint paths would require Alice to painstakingly review every single warning and count occurrences manually. GlobalImpact automatically computes how often each API appears across multiple paths, ranking them by frequency. In Figure 6, because `extractParts(standardizedSSN)` appears in more paths, it has a higher score, signaling a larger global impact, which provides Alice with a better way to view the global impact of the APIs existing in the flows she is examining.

3 Speculative Analysis for Inquiry-based Debugging

WhyFlow enables inquiry-based debugging of spurious flows or missing flows. Figure 7 shows the precomputation of the taint facts by executing both the original CodeQL taint query and a less restricted version of this query (Section 3.1). After this, we convert the taint analysis results into *Soufflé* facts and store them in a "Results Database" (Section 3.2). Subsequently, WhyFlow queries this database to provide answers to the template questions (Section 3.3). We rely on several assumptions.

Assumption 1. We presume that a user of WhyFlow is familiar with the program they are inspecting, meaning that they can recognize that some flows are unexpected flows and some expected flows are missing, serving as a starting point for end-user debugging.

Assumption 2. We leverage CodeQL to obtain taint analysis results as is, and then we utilize Soufflé to run the interrogative debugging queries on top of CodeQL results.

Assumption 3. We assume that CodeQL results are created with the information flow models of third-party libraries, which is typical for modern taint analysis. We assume that inaccuracies in these plug-in models should be debugged, when a user suspects a missing flow or a spurious flow.

The goal of WhyFlow is not to disambiguate false positives from false negatives. Rather, WhyFlow helps with end-user debugging, explaining how configuration choices, such as source/sink definitions and third-party models, may lead to spurious or missing paths.

3.1 CodeQL

CodeQL is an open-source static analysis framework. Users can use QL, which is a declarative, object-oriented Domain-Specific Language (DSL) specifically designed for writing CodeQL queries to detect potential security vulnerabilities or other issues [16].

A key factor in CodeQL's comprehensiveness lies in its *third-party library modeling*. Each library API (including third-party dependencies) can be modeled to reflect how data flows through its methods. As manual creation of such models is time-consuming and error-prone, the CodeQL team has explored machine-learning-based techniques to infer library behaviors [42]. Nonetheless, incomplete or incorrect modeling remains a practical challenge.

Rather than relying on specific third-party library models and source and sink definitions, a general query may relax these configuration assumptions, revealing the "maximal" set of reachable flows from all possible sinks to all possible sources, ignoring third-party library models. Such an approach can help users uncover paths that would otherwise be missed due to incomplete or inaccurate library modeling. Moreover, comparing results from both model-aware and "maximal" queries can expose discrepancies, hints that the current model assumptions may need refinement. Our work builds on these insights by introducing an interactive, interrogative debugger, where users can easily toggle assumptions about third-party methods with pre-defined template queries.

3.2 Soufflé and Logic Query Implementation

WhyFlow converts the CodeQL taint-analysis results into a set of *facts* that can be analyzed by Soufflé's Datalog engine. In this factbase, each node, edge, and taint-related annotation (e.g., source, sink, and library API) is expressed as a logical predicate. Once the output of a single CodeQL query is transformed into these facts, WhyFlow can quickly re-evaluate multiple "secondary" queries, such as those in Table 1, without needing to re-run CodeQL's underlying taint analysis itself.

This arrangement is particularly useful for tasks like missing-flow analysis (WhyNotFlow) or identifying global impact. Instead of incurring repeated analysis times of 10 to 14 seconds (excluding the compile time of the query) in CodeQL for each reconfiguration of the library model, we rely on Soufflé's Datalog engine [30] to query the precomputed facts in the order of 5 seconds—a speed-up that supports interactive debugging.

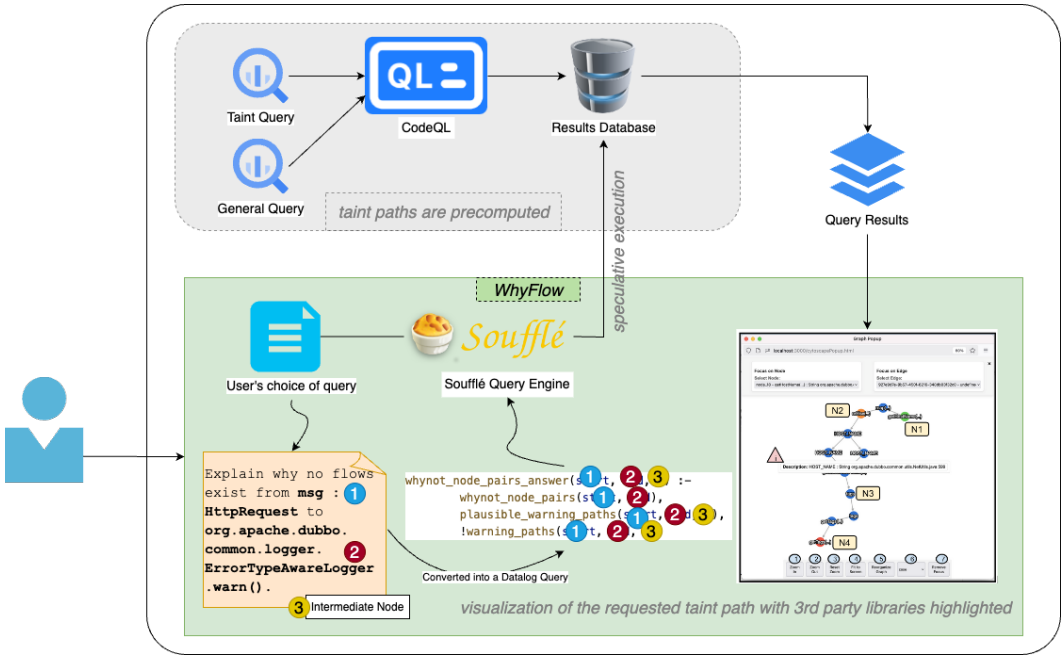

Figure 7: A user selects a *WhyNotFlow* template query to investigate a suspected missing flow `msg:HttpRequest` to `ErrorLogger.warn()`. WhyFlow concretizes a corresponding logic query. For instance, to explain a potential missing flow from `msg:HttpRequest` (node 1) to `ErrorLogger.warn()` (node 2), a plausible path via an intermediate node (node 3) is first identified by the rule `whynot_node_pairs_answer(1,2,3)`. The query result is shown in the *Graph View*.

Converting the CodeQL Output to Facts. Each CodeQL dataflow node is mapped to a `node(id)` predicate, capturing its unique identifier and associated metadata (e.g., filename, line/column, symbol name). Likewise, each dataflow edge is encoded as `edge(edgeid, sourceid, targetid)`, indicating potential taint propagation. We also store "plausible edges"—those that might be inactive under certain sanitizers or library assumptions—using a similar `plausible_edge` predicate. Sources and sinks become `source(nodeid)` and `sink(nodeid)`, while known library-flow relations (e.g., which arguments flow into return values) are recorded with `library_flow(edgeid, fact_id)`.

Answering "WhyFlow" and "WhyNotFlow". Once loaded into Soufflé, the questions in Table 1 are encoded as logic queries shown in Figures 2 and 3. For example, WhyFlow query uses transitive closure over edge to find whether data can reach a sink from a source, intersecting with `library_flow` to highlight which third-party library's APIs (orange nodes) appear along the path. WhyNotFlow similarly checks for "broken" transitive paths and identifies `library_flow` edges that act as sanitizers, which terminate flows.

Supporting "DivergentSources/Sinks" and "GlobalImpact". DivergentSinks and DivergentSources queries use logic rules to detect the last common node or first common node in two paths. They compute the intersection of reachable sets for each source/sink pair. Meanwhile, GlobalImpact queries count how frequently an API appears in distinct `reachable` flows; each API node is assigned a score via an aggregation rule. WhyFlow sizes the node (in the visualization) based on its score, visualizing its "global impact."

3.3 WhyFlow's User Interface

To support sensemaking, key design elements in WhyFlow include:

Color-Coding and Visual Clarity. Nodes in the flow graph are color-coded: **green** for sources, **red** for sinks, **orange** for external APIs or libraries, and **blue** for other intermediate nodes.

Graphical Flows and Expandable Paths. WhyFlow overlays the taint flows onto a single graph. Solid edges represent dataflow steps. Dashed edges indicate "plausible flows" currently unreported by the analyzer, but can be reported under a different configuration. Each flow can be expanded to narrow down onto suspicious segments.

Clickable Nodes and Code Navigation. Clicking on a node opens the corresponding code snippet in the user's IDE for easy reference of the source code. WhyFlow includes hover popups showing details such as fully-qualified names of identifiers referenced by the intermediate steps along the flow.

Customizable Layouts and Node Sizing. WhyFlow supports multiple layout algorithms (e.g., breadth-first, concentric). A *GlobalImpact* query resizes API nodes by their number of occurrences on different flows.

4 User Study

To evaluate WhyFlow's usefulness in sensemaking taint flows, we designed a within-subject study. We assess how users can reason about the impact of user configurations on multiple taint flows,

Figure 8: CodeQL's tree-based view shows each warning one by one and does not support inquiry-based sensemaking of multiple taint flows and the impact of user choices and third-party library models.

Metric	Value
# of edges	6,901
# of nodes	8,101
# of sources	26
# of sinks	265
# of third-party API functions	85

Table 2: Statistics of Taint Analysis Facts for `Apache Dubbo [32]`.

```
import java
import semmle.code.java.dataflow.FlowSources
import semmle.code.java.dataflow.TaintTracking
import semmle.code.java.security.ExternalAPIs
import UntrustedDataToExternalApiFlow::PathGraph

from UntrustedDataToExternalApiFlow::PathNode source,
     UntrustedDataToExternalApiFlow::PathNode sink
where UntrustedDataToExternalApiFlow::flowPath(source, sink)
select sink, source, sink,
  "Call to " + sink.getNode().(ExternalApiDataNode).getMethodDescription() +
  " with untrusted data from $@.", source, source.toString()
```

Listing 1: CodeQL Taint Analysis Query [17].

including third-party taint analysis models. We use CodeQL's Visual Studio Code plugin (CodeQL Visualizer in short), as the baseline.

Study Design. Participants were asked to answer eight sensemaking questions about taint analysis results and how the configuration of third-party libraries impacts taint flows. Each user study task consists of eight questions shown in Table 3. The first six questions are centered on why, why-not, and what-if questions about taint flows and the remaining two are focused on quantifying taint flows.

Research Questions.

(1) How much does WhyFlow improve the participants' ability to answer questions about the configuration's impact on taint flows?
(2) How does WhyFlow influence cognitive load and user confidence in sensemaking taint flows?
(3) What are the participants' perceptions of WhyFlow's usability and functionality in enhancing their workflow?

4.1 Study Protocol

The study task is based on 383 taint warnings generated on Apache Dubbo [32] with the CodeQL query shown in Listing 1 [17]. The resulting facts from this query are presented in Table 2.

4.1.1 Baseline. We selected the CodeQL plugin in VSCode (CodeQL visualizer in short), shown in Figure 8 as our baseline tool. It provides an interface where warnings are listed one-by-one, grouped under each analysis kind. Users can click and expand each warning to reveal the flows that contributed to the warning. Each flow can be expanded to show the steps in the flow. A user inspects warnings one-by-one.

4.1.2 Participants. We conducted a within-subject user study with a crossover design. We recruited 12 participants, including graduate students and professional developers from the industry. Their programming experience ranged from 1–3 years (4 participants), 4–6 years (3 participants), 7–10 years (3 participants), to over 10 years (2 participants). The average self-reported familiarity with taint analysis was 2.1 out of 5, suggesting that most participants had only moderate experience with this type of dataflow analysis.

7 participants are PhD students, 3 are MS students, 1 is an undergraduate and 1 is a professional developer from industry. We dropped one participant from the study, since the participant failed to complete half of both tasks.

Number of participants. While between-subject user studies require a large number of participants to account for variations among individual participants, within-subject user studies minimize variability, as each participant uses both tools following a different order. The order of tool usage and task assignment is randomized and counterbalanced [12, 55]. Within-subject user studies with 8

to 16 participants are standard practice in both software engineering [3, 21, 31] and HCI research [27, 28, 52].

4.1.3 Protocol. Each participant took part in a 1.5-hour session. The study involved using both WhyFlow and the baseline CodeQL visualizer for the two tasks. The order of assigned tool (WhyFlow first vs. CodeQL first) and the assigned task (Problem Set #1 and Problem Set #2) was counterbalanced across participants through random assignment. We gave a 3-minute pre-study survey to collect background information, followed by a tutorial.

Next, each participant proceeded with the two tasks, each lasting 20 minutes. Each task had 8 questions that participants were tasked to answer. They had the option of skipping over questions and ending the task early. After each task, participants filled out a 5-minute post-task survey, which included the NASA-TLX questions [25] (e.g., mental demand, time pressure, perceived success, effort, and frustration). In the final 12-minute post-study survey, we asked participants to compare WhyFlow and CodeQL, and rate WhyFlow's features.

4.1.4 Tutorial. We conducted a tutorial session to introduce participants to the functionality of both WhyFlow and the CodeQL VSCode plugin. For WhyFlow, we guided users through all six query types. This involved selecting relevant sources, sinks, or API calls from dropdown menus and interpreting the graphical output rendered by WhyFlow. Participants were encouraged to interact with WhyFlow's features to familiarize themselves with them.

We also conducted a tutorial session on using CodeQL's visualizer. We walked the participants through the process of viewing query results in a tabular list, and expanding or collapsing the flows to inspect the intermediate steps between source and sink.

Finally, we explained the role of CodeQL "models" of external libraries, highlighting their role in the analysis. We reinforced core concepts—sources, sinks, and sanitizers—and provided examples.

4.1.5 Tasks. Each task had a different set of questions (Problem Set #1 and Problem Set #2). The questions and their answers are shown in Table 3. Each task contained eight questions about 'why', 'why-not', 'what-if,' as shown in Table 1, and two additional questions about the quantitative aspects of taint flows (e.g., how many taint paths exist or how many third-party APIs appear in the taint paths). The questions were multiple-choice questions, some having multiple answers. We granted partial credit if participants selected some correct answers, but missed or added incorrect choices.

4.1.6 Criteria for Study Tasks. We designed questions based on realistic scenarios involving the analysis of taint flows. These questions correspond to "why", "why-not", and "what-if" questions that developers may ask about a taint analysis or assess potential changes to models of the external libraries.

Post-Study Questionnaire. After completing each task, participants completed a brief post-task questionnaire, which included the NASA-TLX questions. They also described their strategies to find answers to the questions in this questionnaire. At the end of the session, participants rated user-interface features, confidence, and ease of use.

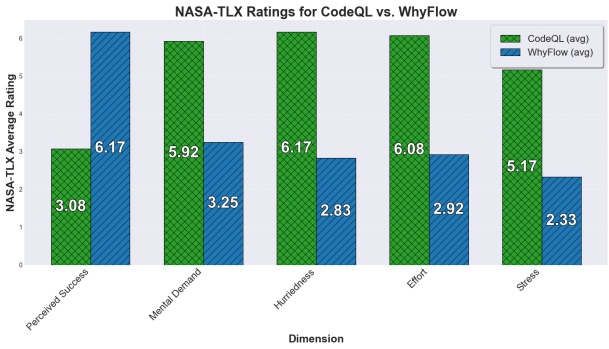

Figure 9: Average NASA-TLX Ratings for WhyFlow vs. CodeQL visualizer.

5 Results

In this section, we analyze the results of our user study. In our evaluation, use non-parametric statistical tests because our data is inherently ordinal, including Likert-scale responses (NASA-TLX ratings from 1–7, confidence ratings from 1–5, ease-of-use ratings from 1–5) and categorical accuracy assessments (correct, partial, incorrect, empty). Specifically, we apply the two-sided *Mann-Whitney U test* [4, 51] to compare distributions between WhyFlow and CodeQL, as is standard practice for ordinal data in user studies [7, 20, 38, 56, 58]. We report statistical significance at the $\alpha = 0.05$ level. We also compute Cohen's d to measure effect size using the pooled standard deviation, where $|d| > 0.8$ indicates a large effect [15].

We test the following null hypotheses:

- **RQ1 (Accuracy):** H_0: There is no difference in the distribution of participants' accuracy (correct, partial, or empty responses) between WhyFlow and CodeQL.
- **RQ2 (Cognitive Load):** H_0: There is no difference in the distributions of participants' mental demand, hurriedness, perceived success, effort, or stress between WhyFlow and CodeQL.
- **RQ3 (Usability):** H_0: There is no difference in the distributions of participants' confidence or ease-of-use ratings between WhyFlow and CodeQL.

We reject each null hypothesis when $p < 0.05$, indicating that observed differences are statistically significant and unlikely due to chance. Each participant is denoted as P#.

5.1 RQ1. Accuracy

Table 4 shows the participants' accuracy in answering questions for the problem sets (*Task 1* and *Task 2*) when using WhyFlow and the CodeQL Visualizer. Accuracy outcomes showed a clear advantage for WhyFlow, where users had more correct answers than using the CodeQL visualizer ($p < 0.05$), alongside fewer empty responses ($p < 0.05$).

These differences are statistically significant according to the Mann-Whitney U test with a large effect size (*Correctness:* Cohen's $d = 1.33$, *Empty answers:* Cohen's $d = 1.8$). We therefore reject the null hypothesis for RQ1: WhyFlow significantly improves accuracy compared to CodeQL. Participants using WhyFlow provided more correct answers, especially when needing to reason about multiple

Problem Set #1	Problem Set #2
(1) Explain why a taint flow is permitted from `getRequestURI(...)` in `[...].PageServlet.java` to charAt(...) in `[...].StringUtils.java`. Name a third-party API permitting the flow. **Answer:** `java.lang.String.substring(int beginIndex, int endIndex)`	(1) Explain why a taint flow is permitted from msg: String in `[...].TelnetProcessHandler.java` to json: String in `[...].FastJsonImpl.java`. Name a third-party API permitting the flow. **Answer:** `java.lang.String.substring(...)`
(2) Explain why no taint flow is permitted from `msg: HttpRequest` to `warn(...)` in `[...].HttpProcessHandler.java`. **Answer:** `io.netty.handler.codec.http.HttpRequest.method(...)`	(2) Explain why no taint flow is permitted from `msg: Http2StreamFrame` in `[...].TripleHttp2ClientResponseHandler.java` to release(...) in `[...].TripleClientStream.java`. **Answer:** `io.netty.handler.codec.http2. Http2DataFrame.isEndStream()`
(3) Explain what sinks would no longer be reachable, if `io.netty.handler.codec.http.HttpRequest.uri()` were modeled as a sanitizer, starting from source `msg: HttpRequest` in `[...].HttpProcessHandler.java`.[1ex] **Answer:** `valueList` in `[...].HttpCommandDecoder.java` and `msg` in `[...].Log4jLogger.java`	(3) Explain what sinks would be no longer reachable, if `io.netty.handler.codec.http2.Http2HeadersFrame.headers()` is marked as a sanitizer, starting from source `msg: Object` in `[...].TripleHttp2FrameServerHandler.java`. **Answer:** path: `[...].TriplePathResolver.java` and headers: `[...].TripleIsolationExecutorSupport.java`
(4) Identify the program point that affects multiple taint flows ending at two sinks: *sinks:* `path: String` in `[...].TriplePathResolver.java` at line 41 and `path: String` in `[...].TriplePathResolver.java` at line 46, *source:* `msg: Http2StreamFrame` in `[...].TripleHttp2ClientResponseHandler.java`. **Answer:** `toString(...): String in [...].TripleServerStream.java`	(4) Identify the program point that affects multiple taint flows ending at two sinks: *sinks:* `path: String` in `[...].TriplePathResolver.java` – line 41 and `path: String` in `[...].TriplePathResolver.java` – line 46, *source:* `msg: Object` in `[...].TripleHttp2FrameServerHandler.java`. **Answer:** `toString(...): String in [...].TripleServerStream.java`
(5) Identify the intermediary program point that influences multiple flows originating from `input: ByteBuf` in `[...].NettyCodecAdapter.java` and in: ByteBuf in `[...].NettyPortUnificationServerHandler.java`, ending at `buffer: ByteBuf` in `[...].NettyBackedChannelBuffer.java`. **Answer:** `parameter this : NettyBackedChannelBuffer [buffer]: ByteBuf in [...].NettyBackedChannelBuffer.java`	(5) Identify the intermediary program point that influences multiple flows originating from `msg: Http2StreamFrame` in `[...].TripleHttp2ClientResponseHandler.java` and `msg: Object` in `[...].TripleHttp2FrameServerHandler.java`, ending at `path: String` in `[...].TriplePathResolver.java`. **Answer:** `headers: Http2Headers in [...].TripleServerStream.java`
(6) Which third-party APIs could have the most influence on the taint path from `msg: Object` in `[...].TripleHttp2FrameServerHandler.java` to `path: String` in `[...].TriplePathResolver.java`? Rank in the order of importance. **Answer:** (1) `io.netty.handler.codec.http2.Http2HeadersFrame.headers()` (2) `java.lang.CharSequence.toString()` (3) `io.netty.handler.codec.http2.Http2Headers.path()`	(6) Which third-party APIs could have the most influence on the taint path from `msg: Object` in `[...].NettyClientHandler.java` to `key: String` in `[...].TraceFilter.java`? Rank in the order of importance. **Answer:** (1) `java.lang.String.trim()` (2) `java.lang.String.replace(...)` (3) `java.lang.String.substring(...)`
(7) Determine the number of pass-through API points from `getRequestURI(...)` in `[...].PageServlet.java` to `charAt(...)` in `[...].StringUtils.java`. **Answer:** 21	(7) Determine the number of pass-through API points from `msg: Http2StreamFrame` in `[...].TripleHttp2ClientResponseHandler.java` to `path: String` in `[...].TriplePathResolver.java`. **Answer:** 21
(8) Count how many different dataflow paths exist from `in: ByteBuf` in `[...].NettyPortUnificationServerHandler.java` to `buffer: ByteBuf` in `[...].NettyBackedChannelBuffer.java`. **Answer:** 2	(8) Count how many different dataflow paths exist from `msg: Http2StreamFrame` in `[...].TripleHttp2ClientResponseHandler.java` to `path: String` in `[...].TriplePathResolver.java`. **Answer:** 4

Table 3: Study Tasks: Problem Sets #1 and #2 with their correct answers. Participants were assigned one of the two tasks to complete with CodeQL visualizer and the other with WhyFlow.

flows and global impact; on the other hand, participants using the CodeQL visualizer felt overwhelmed [49].

> Participants gave more correct and complete answers using WhyFlow than the baseline. Their answers were over 50% more accurate and significantly better ($p < 0.05$). WhyFlow users also left fewer questions unanswered.

5.2 RQ2. Cognitive Load and Confidence

After each task, participants filled in the NASA-TLX questionnaire, which measures five dimensions: mental demand, hurriedness, perceived success, effort, and stress. Figure 9 reports the average scores for the participants. Other than perceived success (where higher is better), lower values are better. All of the NASA-TLX dimensions are rated on a scale from 1 to 7. Overall, participants found completing the task with WhyFlow less mentally demanding (3.25 compared to 5.92), less hurried (2.83 compared to 6.17), and less stressful (2.33 compared to 5.17), while reporting higher perceived success (6.17 compared to 3.08) and requiring less effort (2.92 compared to 6.08). On all NASA-TLX dimensions, the improvements were statistically significant ($p < 0.05$) with a large effect size (Cohen's $d > 2$ for all five dimensions). We therefore reject the null hypothesis for RQ2: WhyFlow significantly reduces cognitive load and improves perceived success compared to CodeQL.

Several participants commented that CodeQL visualizer forced them to "manually inspect the nodes in each path" (P2) and "go back and forth to visualize in [their] head" (P10). In contrast, P5 described WhyFlow's approach as "very clear," and P11 noted it was "easier to track the dataflow." These comments reinforce our findings that WhyFlow's inquiry-based debugging reduced the cognitive overhead of analyzing multiple interconnected flows. Most participants preferred WhyFlow's graph-based interface over the default list-view of CodeQL visualizer. The participants identified

Q	WhyFlow		CodeQL visualizer	
	Task 1 (C/E)	Task 2 (C/E)	Task 1 (C/E)	Task 2 (C/E)
(1)	100%/0%	100%/0%	100%/0%	67%/0%
(2)	83%/0%	100%/0%	83%/0%	50%/0%
(3)	33%/0%	50%/0%	17%/17%	0%/17%
(4)	100%/0%	100%/0%	83%/0%	83%/17%
(5)	67%/0%	50%/0%	50%/50%	83%/17%
(6)	67%/0%	83%/0%	17%/67%	33%/17%
(7)	100%/0%	67%/17%	83%/17%	17%/83%
(8)	100%/0%	67%/17%	67%/33%	17%/83%

Table 4: User Study Results: Percentage of participants providing (C)orrect/(E)mpty answers when using WhyFlow vs. CodeQL visualizer.

areas for improvement for WhyFlow. P9 mentioned that the reliance on node IDs might be unrealistic in real-world scenarios where users do not have those IDs at hand. P1, while feeling "more confident using [WhyFlow]," pointed out that some "general taint analysis questions" might still be faster in CodeQL visualizer's tabular format. P7 suggested "showing fully qualified names when hovering on a node" to reduce confusion in large, complex graphs.

Participants valued the ease of tracing flows, especially when there are multiple source-sink relationships. They envisioned advanced filtering options, improved naming (to avoid codebase ambiguities), and possible integration of textual or tabular summaries.

> Using WhyFlow, participants reported significantly lower mental demand, effort, stress, and hurriedness, and felt more successful. They found the visually clear graphs and template queries helpful in supporting a smoother and more confident sensemaking process.

5.3 RQ3. Usability and Workflow

5.3.1 Interface Features. The usefulness of WhyFlow's interface was rated by the participants, beyond its core queries. Participants rated color-coding with the highest average score (4.83/5). P9 said, *"Imagine not having the coloring and having to click each node to figure out what they are... The visualization hides the textual noise."*

Some participants underutilized the expandable taint paths (with an average score of 3.33/5) as they were unaware of the feature. Meanwhile, P7 wanted explicit labeling of fully qualified names on the graph to avoid switching to the IDE. Participants found clicking on nodes to jump into code less useful (with an average score of 3.92/5), mainly using it to confirm ambiguous method names.

5.3.2 Confidence and Ease of Use. Table 5 shows participants' ratings of WhyFlow and CodeQL visualizer in terms of confidence (in the answers to the questions) and overall ease of use. WhyFlow scored 4.25/5 on both measures, substantially higher than CodeQL visualizer (2.08/5 for confidence, 1.92/5 for ease). These improvements were statistically significant ($p < 0.05$, Cohen's $d > 2$ for both measures). We therefore reject the null hypothesis for RQ3: WhyFlow significantly improves user confidence and ease of use.

Multiple participants appreciated WhyFlow's *"dedicated queries"* (P2) for analyzing the results, noting that *"it makes it easier to handle higher-level tasks"* (P4). However, some participants (P1, P9) commented that CodeQL visualizer's table-based listings would still be useful in simpler scenarios.

	WhyFlow	CodeQL Visualizer
Confidence (1–5)	4.25	2.08
Ease of Use (1–5)	4.25	1.92

Table 5: The participants were confident in answering questions with WhyFlow and found it easy to use.

5.3.3 Future Adoption. Participants gave an average rating of 4.83/5, when asked if they would like to use WhyFlow in future taint-flow inspections. Many emphasized that despite needing refinements, the specialized queries, graph-based results, and color-coded visualization saved significant time. P5 commented, *"For a large program, it's difficult for the programmer to check each path by hand—WhyFlow's approach can save time and reduce errors."*

> Participants valued WhyFlow's color-coding and graph-based sensemaking features.

5.4 Follow-up Study

To validate our findings from an industry perspective, we conducted studies with two **security** professionals with 5/5 familiarity in static taint analysis and 7-10 years of experience, using the protocol from Section 4.1. Both provided more correct answers with WhyFlow than CodeQL and preferred WhyFlow's workflow. P101 noted, *CodeQL requires lots of manual labor, whereas WhyFlow's visual components make it easier to navigate. [The workflow] was very intuitive, the tooling eases manual inspection load completely."* P102 added, *With CodeQL, I wonder whether I'm missing another row that describes the same location."* Both achieved high task success with both tools (6/7) but reported higher confidence and ease of use with WhyFlow. P102 experienced equally low mental demand but felt more rushed with WhyFlow (4/7 vs. 2/7), despite lower effort (2/7 vs. 3/7) and no additional frustration. P101 reported reduced mental demand (2/7 vs. 7/7), hurriedness (1/7 vs. 7/7), and effort (2/7 vs. 5/7) with WhyFlow, plus slightly less frustration. Overall, WhyFlow matched CodeQL's perceived success while reducing cognitive burden.

6 Discussion

```
.decl edge(id: number, src: number, dst: number)
.input edge

.decl branch(n: number)
.output branch
branch(n) :-
    edge(_, n, _),
    c = count : edge(_, n, _),
    c > 1.
```

Listing 2: A *divergent path* query flagging *branch points* where flow diverges to multiple targets.

Improved Accuracy with Visual Queries. Using WhyFlow, participants achieved better accuracy, especially when analyzing multiple flows. In particular, the AffectedSinks and GlobalImpact queries require the analysis of multiple flows. Surprisingly, participants were able to answer questions related to "why-not" using both tools. Participants using the CodeQL visualizer frequently had empty submissions due to lack of time. This shows that an interactive, inquiry-based debugger has the potential to improve sensemaking for large result sets, which is known to hinder adoption [29].

Suggestions for Enhancement. While participants found WhyFlow "less cumbersome" (P2) for analysis involving multiple flows, they offered suggestions for improving WhyFlow. They recommended adding an on-screen legend to clarify colors and dotted edges to avoid visual confusion. Additional interactive features, such as "click-to-edit" functionality for enabling or disabling sanitizers were also proposed. This suggests that participants would appreciate real-time feedback through interaction on the graph.

Extensibility of Template Queries. While WhyFlow supports six template questions from Table 1, adding more template queries is easy since WhyFlow's template questions are Datalog-style queries. As a case study, adding a new template query called 'divergent path' query to WhyFlow (Listing 2) took under ten minutes. This query identifies the *branch points*, which are locations where the flow branches into multiple targets. Note that WhyFlow's queries support any taint analysis results from CodeQL.

Supporting Interactivity. WhyFlow supports interrogative and speculative debugging built on a rich history of using logic programming for software comprehension [19, 23, 24, 26, 40, 54]. WhyFlow does not compete with underlying taint analysis engines but instead emphasizes *sensemaking* support.

7 Threats to Validity

Construct Validity. We designed questions based on realistic "why," "why-not," and "what-if" debugging scenarios [35], requiring reasoning about multiple flows rather than simple lookup tasks. We employed the NASA-TLX questionnaire [25] and the technology acceptance model [36]. However, the self-reported metrics may be influenced by subjective interpretations.

Internal Validity. We employed a within-subject crossover design with counterbalanced tool order and task assignment to mitigate learning effects and order bias [55]. However, participants may experience fatigue, and time pressure. The counterbalanced assignment controls for difficulty differences between problem sets, but residual variations could affect results.

External Validity. Although our study included only 12 participants, this aligns with typical sample sizes in within-subject SE and HCI studies [3, 21, 27, 28, 31, 52, 55], which require fewer participants than between-subject designs. Most participants were students, but prior work shows that students perform comparably to professionals on security tasks [1, 2, 34, 43, 44, 47, 48], and a follow-up study with two security professionals (Section 5.4) further corroborated our findings.

While our study evaluated WhyFlow only against CodeQL on Apache Dubbo, WhyFlow operates on generic dataflow facts (nodes, edges, sources, sinks) extractable from any analyzer, template queries are Datalog rules independent of the analysis engine, and the approach is extensible without core modifications (Section 6).

8 Related Work

Speculative What-If Analysis. Speculative execution [9, 10] allows the investigation of future or alternative actions developers may perform, such as for providing fix suggestions [9] and preventing merge conflicts [10]. Our work is the first to apply it for end-user debugging of taint analysis.

Templated Questions. Developers often ask questions during a variety of developer tasks [8]. In particular, developers need support for understanding code reachability [35], exploring how different vulnerabilities relate and finding similar ones [50], tracking intermediate states of analysis [18], reasoning about system-wide implications when making changes [50]. WhyFlow is the first work providing templated questions for investigating and debugging the results of taint analysis.

Modelling Third-Party Libraries. Existing work [13, 37, 46] automatically infers models of third-party libraries. However, they do not guide users to understand the impact of these models. While Paralib [57] allows comparison between multiple library choices, it does not allow reasoning about how incorrect models cause a mismatch between user expectations and the actual tool results.

End-user debugging. End user debugging [11, 22, 33] focuses on investigating the root causes of tool outcomes, and has been applied to spreadsheet debugging, etc. WhyFlow is an end-user debugger for reasoning how taint flows are affected by the configuration of sources, sinks, and models of third-party libraries.

9 Conclusion

We presented WhyFlow, a tool for end-user debugging for taint analysis. Through speculative analysis, WhyFlow allows developers to ask "why", "why-not", and "what-if" questions about the analysis configuration, and is able to visualize multiple, interconnected flows on a graphical view. WhyFlow enables sensemaking of taint analysis and helps users identify root causes of unexpected flows or missing flows. Our user study confirms that WhyFlow helps users provide 21% more correct answers to questions about taint analysis, while experiencing a lower cognitive load. These results suggest that WhyFlow's inquiry-based approach empowers developers to analyze the outputs of taint analysis.

Data Availability

The replication package and study data are available at https://github.com/UCLA-SEAL/WhyFlow.

Acknowledgments

This work is supported by the National Science Foundation under grant numbers 2426162, 2106838, and 2106404. It is also supported in part by funding from Amazon and Samsung. We thank the anonymous reviewers for their constructive feedback that helped improve the work.

References

[1] Yasemin Acar, Michael Backes, Sascha Fahl, Doowon Kim, Michelle L. Mazurek, and Christian Stransky. 2016. You Get Where You're Looking for: The Impact of Information Sources on Code Security. In *2016 IEEE Symposium on Security and Privacy (SP)*. 289–305. https://doi.org/10.1109/SP.2016.25

[2] Yasemin Acar, Christian Stransky, Dominik Wermke, Michelle L. Mazurek, and Sascha Fahl. 2017. Security Developer Studies with GitHub Users: Exploring a Convenience Sample. 81–95. https://www.usenix.org/conference/soups2017/technical-sessions/presentation/acar

[3] Emily Judith Arteaga Garcia, João Felipe Nicolaci Pimentel, Zixuan Feng, Marco Gerosa, Igor Steinmacher, and Anita Sarma. 2024. How to Support ML End-User Programmers through a Conversational Agent. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 53, 12 pages. https://doi.org/10.1145/3597503.3608130

[4] Deborah Ashby. 1991. Practical statistics for medical research. Douglas G. Altman, Chapman and Hall, London, 1991. No. of pages: 611. Price: £32.00. *Statistics in Medicine* 10, 10 (1991), 1635–1636. https://doi.org/10.1002/sim.4780101015

[5] Pavel Avgustinov, Oege De Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-oriented queries on relational data. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik.

[6] Subarno Banerjee, Siwei Cui, Michael Emmi, Antonio Filieri, Liana Hadarean, Peixuan Li, Linghui Luo, Goran Piskachev, Nicolás Rosner, Aritra Sengupta, Omer Tripp, and Jingbo Wang. 2023. Compositional Taint Analysis for Enforcing Security Policies at Scale. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1985–1996. https://doi.org/10.1145/3611643.3613889

[7] Gabriele Bavota, Bogdan Dit, Rocco Oliveto, Massimilano Di Penta, Denys Poshyvanyk, and Andrea De Lucia. 2013. An empirical study on the developers' perception of software coupling. In *2013 35th International Conference on Software Engineering (ICSE)*. 692–701. https://doi.org/10.1109/ICSE.2013.6606615

[8] Andrew Begel and Thomas Zimmermann. 2014. Analyze this! 145 questions for data scientists in software engineering. In *Proceedings of the 36th International Conference on Software Engineering*. 12–23.

[9] Yuriy Brun, Reid Holmes, Michael D Ernst, and David Notkin. 2010. Speculative analysis: exploring future development states of software. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*. 59–64.

[10] Yuriy Brun, Reid Holmes, Michael D Ernst, and David Notkin. 2011. Proactive detection of collaboration conflicts. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 168–178.

[11] Margaret Burnett, Curtis Cook, and Gregg Rothermel. 2004. End-user software engineering. *Commun. ACM* 47, 9 (2004), 53–58.

[12] Sarah E. Chasins, Elena L. Glassman, and Joshua Sunshine. 2021. PL and HCI: better together. *Commun. ACM* 64, 8 (July 2021), 98–106. https://doi.org/10.1145/3469279

[13] Wen-Hao Chiang, Peixuan Li, Qiang Zhou, Subarno Banerjee, Martin Schaef, Yingjun Lyu, Hoan Nguyen, and Omer Tripp. 2024. Inference for Ever-Changing Policy of Taint Analysis. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '24)*. Association for Computing Machinery, New York, NY, USA, 452–462. https://doi.org/10.1145/3639477.3639738

[14] Victor Chibotaru, Benjamin Bichsel, Veselin Raychev, and Martin Vechev. 2019. Scalable taint specification inference with big code. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 760–774. https://doi.org/10.1145/3314221.3314648

[15] Jacob Cohen. 2013. *Statistical power analysis for the behavioral sciences* (2 ed.). Routledge, London, England.

[16] CodeQL developers. [n. d.]. CodeQL documentation: Using custom queries with the CodeQL CLI. https://docs.github.com/en/code-security/codeql-cli/using-the-advanced-functionality-of-the-codeql-cli/using-custom-queries-with-the-codeql-cli.

[17] CodeQL developers. [n. d.]. CodeQL query: UntrustedDataToExternalAPI. https://github.com/github/codeql/blob/996bc47ae8d10f6087504413db02c8920243b13e/java/ql/src/Security/CWE/CWE-020/UntrustedDataToExternalAPI.ql.

[18] Lisa Nguyen Quang Do, Stefan Krüger, Patrick Hill, Karim Ali, and Eric Bodden. 2018. Debugging static analysis. *IEEE Transactions on Software Engineering* 46, 7 (2018), 697–709.

[19] Michael Eichberg, Sven Kloppenburg, Karl Klose, and Mira Mezini. 2008. Defining and continuous checking of structural program dependencies. In *Proceedings of the 30th International Conference on Software Engineering* (Leipzig, Germany) *(ICSE '08)*. Association for Computing Machinery, New York, NY, USA, 391–400. https://doi.org/10.1145/1368088.1368142

[20] Ethan Fast, Binbin Chen, Julia Mendelsohn, Jonathan Bassen, and Michael S. Bernstein. 2018. Iris: A Conversational Agent for Complex Tasks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3173574.3174047

[21] Mohammad Ganji, Saba Alimadadi, and Frank Tip. 2023. Code Coverage Criteria for Asynchronous Programs. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1307–1319. https://doi.org/10.1145/3611643.3616292

[22] Valentina Grigoreanu, Margaret Burnett, Susan Wiedenbeck, Jill Cao, Kyle Rector, and Irwin Kwan. 2012. End-user debugging strategies: A sensemaking perspective. *ACM Transactions on Computer-Human Interaction (TOCHI)* 19, 1 (2012), 1–28.

[23] Yann-Gaël Guéhéneuc and Giuliano Antoniol. 2008. DeMIMA: A Multilayered Approach for Design Pattern Identification. *IEEE Transactions on Software Engineering* 34, 5 (2008), 667–684. https://doi.org/10.1109/TSE.2008.48

[24] Elnar Hajiyev, Mathieu Verbaere, Oege de Moor, and Kris de Volder. 2005. CodeQuest: querying source code with datalog. In *Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA) *(OOPSLA '05)*. Association for Computing Machinery, New York, NY, USA, 102–103. https://doi.org/10.1145/1094855.1094884

[25] Sandra G Hart and Lowell E Staveland. 1988. Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research. In *Advances in psychology*. Vol. 52. Elsevier, 139–183.

[26] Richard C. Holt. 1998. Structural Manipulations of Software Architecture Using Tarski Relational Algebra. In *Proceedings of the Working Conference on Reverse Engineering (WCRE'98) (WCRE '98)*. IEEE Computer Society, USA, 210.

[27] Amber Horvath, Brad Myers, Andrew Macvean, and Imtiaz Rahman. 2022. Using Annotations for Sensemaking About Code. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology* (Bend, OR, USA) *(UIST '22)*. Association for Computing Machinery, New York, NY, USA, Article 61, 16 pages. https://doi.org/10.1145/3526113.3545667

[28] Mina Huh and Amy Pavel. 2024. DesignChecker: Visual Design Support for Blind and Low Vision Web Developers. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology* (Pittsburgh, PA, USA) *(UIST '24)*. Association for Computing Machinery, New York, NY, USA, Article 142, 19 pages. https://doi.org/10.1145/3654777.3676369

[29] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 672–681.

[30] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On synthesis of program analyzers. In *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II 28*. Springer, 422–430.

[31] Hong Jin Kang, Kevin Wang, and Miryung Kim. 2024. Scaling Code Pattern Inference with Interactive What-If Analysis. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 234, 12 pages. https://doi.org/10.1145/3597503.3639193

[32] Albumen Kevin, Ken Liu, dependabot[bot], Ian Luo, Jermaine Hua, cvictory, Mercy, Huxing Zhang, zrlw, BurningCN, GuoHao, Sean Yang, Gong Dewei, Wang Chengming, wxbty, TomlongTK, qinliujie, Huang YunKun, icodening, earthchen, Pin Xiong, xiaosheng, tswstarplanet, Andy Cheung, Xin Wang, suncairong163, and Jerrick. 2025. apache/dubbo. https://github.com/apache/dubbo. https://github.com/apache/dubbo

[33] Cory Kissinger, Margaret Burnett, Simone Stumpf, Neeraja Subrahmaniyan, Laura Beckwith, Sherry Yang, and Mary Beth Rosson. 2006. Supporting end-user debugging: what do users want to know?. In *Proceedings of the working conference on Advanced visual interfaces*. 135–142.

[34] Amy J. Ko, Thomas D. LaToza, and Margaret M. Burnett. 2015. A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Software Engineering* 20, 1 (Feb. 2015), 110–141. https://doi.org/10.1007/s10664-013-9279-3

[35] Amy J Ko and Brad A Myers. 2004. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 151–158.

[36] Younghwa Lee, Kenneth A Kozar, and Kai RT Larsen. 2003. The technology acceptance model: Past, present, and future. *Communications of the Association for information systems* 12, 1 (2003), 50.

[37] Ziyang Li, Saikat Dutta, and Mayur Naik. 2024. LLM-Assisted Static Analysis for Detecting Security Vulnerabilities. *arXiv preprint arXiv:2405.17238* (2024).

[38] Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Yuekai Huang, Jun Hu, and Qing Wang. 2024. Unblind Text Inputs: Predicting Hint-text of Text Input in Mobile Apps via LLM. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems (CHI '24)*. Association for Computing Machinery, New York, NY, USA, 1–20. https://doi.org/10.1145/3613904.3642939

[39] Benjamin Livshits, Aditya V. Nori, Sriram K. Rajamani, and Anindya Banerjee. 2009. Merlin: specification inference for explicit information flow problems. *SIGPLAN Not.* 44, 6 (June 2009), 75–86. https://doi.org/10.1145/1543135.1542485

[40] Kim Mens, Tom Mens, and Michel Wermelinger. 2002. Maintaining software through intentional source-code views. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering* (Ischia, Italy) *(SEKE '02)*. Association for Computing Machinery, New York, NY, USA, 289–296. https://doi.org/10.1145/568760.568812

[41] Emerson Murphy-Hill, Thomas Zimmermann, Christian Bird, and Nachiappan Nagappan. 2013. The design of bug fixes. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 332–341.

[42] Marcus Nachtigall, Michael Schlichtig, and Eric Bodden. 2022. A large-scale study of usability criteria addressed by static analysis tools. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 532–543.

[43] Alena Naiakshina, Anastasia Danilova, Eva Gerlitz, and Matthew Smith. 2020. On Conducting Security Developer Studies with CS Students: Examining a Password-Storage Study with CS Students, Freelancers, and Company Developers *(CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3313831.3376791

[44] Alena Naiakshina, Anastasia Danilova, Christian Tiefenau, Marco Herzog, Sergej Dechand, and Matthew Smith. 2017. Why Do Developers Get Password Storage Wrong? A Qualitative Usability Study. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 311–328. https://doi.org/10.1145/3133956.3134082

[45] Goran Piskachev, Lisa Nguyen Quang Do, and Eric Bodden. 2019. Codebase-adaptive detection of security-relevant methods. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 181–191.

[46] Goran Piskachev, Lisa Nguyen Quang Do, Oshando Johnson, and Eric Bodden. 2019. SWAN_ASSIST: semi-automated detection of code-specific, security-relevant methods. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1094–1097.

[47] Iflaah Salman, Ayse Tosun Misirli, and Natalia Juristo. 2015. Are Students Representatives of Professionals in Software Engineering Experiments?. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 666–676. https://doi.org/10.1109/ICSE.2015.82

[48] Gustavo Sandoval, Hammond Pearce, Teo Nys, Ramesh Karri, Siddharth Garg, and Brendan Dolan-Gavitt. 2023. Lost at C: a user study on the security implications of large language model code assistants. In *Proceedings of the 32nd USENIX Conference on Security Symposium* (Anaheim, CA, USA) *(SEC '23)*. USENIX Association, USA, Article 124, 18 pages.

[49] Barry Schwartz. 2015. The paradox of choice. *Positive psychology in practice: Promoting human flourishing in work, health, education, and everyday life* (2015), 121–138.

[50] Justin Smith, Brittany Johnson, Emerson Murphy-Hill, Bill Chu, and Heather Richter Lipford. 2015. Questions developers ask while diagnosing potential security vulnerabilities with static analysis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 248–259.

[51] Student. 1908. The Probable Error of a Mean. *Biometrika* 6, 1 (1908), 1–25. https://doi.org/10.2307/2331554

[52] Sangho Suh, Bryan Min, Srishti Palani, and Haijun Xia. 2023. Sensecape: Enabling Multilevel Exploration and Sensemaking with Large Language Models. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology* (San Francisco, CA, USA) *(UIST '23)*. Association for Computing Machinery, New York, NY, USA, Article 1, 18 pages. https://doi.org/10.1145/3586183.3606756

[53] Tamás Szabó. 2023. Incrementalizing Production CodeQL Analyses. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1716–1726.

[54] T. Tourwe and T. Mens. 2003. Identifying refactoring opportunities using logic meta programming. In *Seventh European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings.* 91–100. https://doi.org/10.1109/CSMR.2003.1192416

[55] Sira Vegas, Cecilia Apa, and Natalia Juristo. 2015. Crossover designs in software engineering experiments: Benefits and perils. *IEEE Transactions on Software Engineering* 42, 2 (2015), 120–135.

[56] Noel Warford, Collins W. Munyendo, Ashna Mediratta, Adam J. Aviv, and Michelle L. Mazurek. 2021. Strategies and Perceived Risks of Sending Sensitive Documents. 1217–1234. https://www.usenix.org/conference/usenixsecurity21/presentation/warford

[57] Litao Yan, Miryung Kim, Bjoern Hartmann, Tianyi Zhang, and Elena L Glassman. 2022. Concept-annotated examples for library comparison. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*. 1–16.

[58] Xueyao Yu, Filipe R. Cogo, Shane McIntosh, and Michael W. Godfrey. 2024. Studying the impact of risk assessment analytics on risk awareness and code review performance. *Empirical Software Engineering* 29, 2 (Feb. 2024), 46. https://doi.org/10.1007/s10664-024-10443-x