TraceLens: Question-Driven Debugging for Taint Flow Understanding

Burak Yetiştiren
UCLA
Los Angeles, CA, USA
burakyetistiren@cs.ucla.edu

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

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

***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 3rd-party libraries that abstract permissible and impermissible data flows. Furthermore, a tree-view or a list-view used in existing taint-analyzer's visualization 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 TraceLens, a 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. TraceLens 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 TraceLens 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 TraceLens. This shows TraceLens's potential to significantly reduce sensemaking effort.

I. Introduction

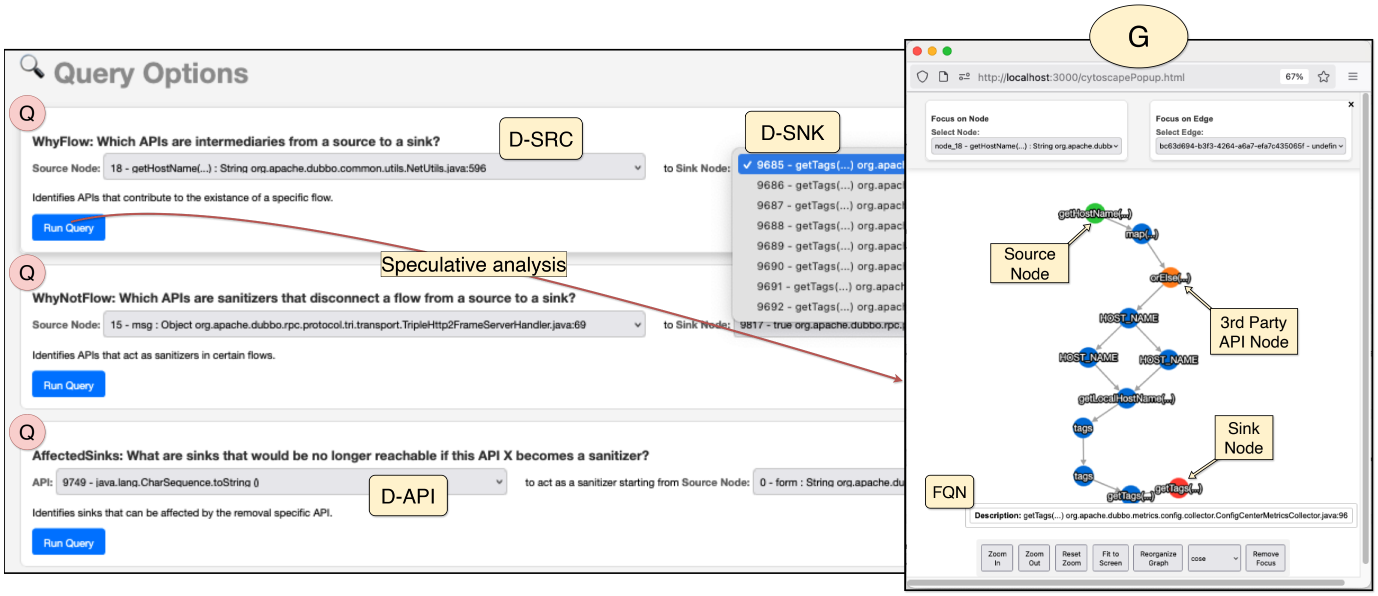
To prevent security vulnerabilities, developers use taint analysis. This technique tracks potentially dangerous data as it moves through a program. Data from untrusted sources, like user input, is marked as "tainted" The analysis then monitors how this tainted data spreads. If it reaches a critical point 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 [1]. 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, 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) [2], [3]. This means it will report potential vulnerabilities that do not actually exist, creating extra, unexpected warnings for the user. 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.

This paper focuses on *end-user debugging* of taint analysis that do not match a user's expectations, in other words, debugging an end-user configuration, as opposed to debugging a taint analysis implementation. To correct configuration errors, developers must understand the impact of configuration 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.

We introduce TraceLens, a novel end-user debugger that brings interactive, question-based sensemaking to taint analysis. Drawing inspiration from interrogative debugging principles, TraceLens enables developers to ask *why*, *why not*, and *what if* questions about their analysis configuration. TraceLens 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, TraceLens 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 don't allow developers to interactively ask questions about these flows. Furthermore, to efficiently narrow


Query Options

WhyFlow: Which APIs are intermediaries from a source to a sink?

Source Node: 18 - getHostName(...) : String org.apache.dubbo.common.utils.NetUtils.java:596

to Sink Node: ✓ 9685 - getTags(...) org.apac

9686 - getTags(...) org.apac

9687 - getTags(...) org.apach

9688 - getTags(...) org.apac

9689 - getTags(...) org.apac

9690 - getTags(...) org.apac

9691 - getTags(...) org.apach

9692 - getTags(...) org.apac

Identifies APIs that contribute to the existance of a specific flow.

WhyNotFlow: Which APIs are sanitizers that disconnect a flow from a source to a sink?

Source Node: 15 - msg : Object org.apache.dubbo.rpc.protocol.tri.transport.TripleHttp2FrameServerHandler.java:69

to Sink Node: 9817 - true org.apache.dubbo.rpc.

Identifies APIs that act as sanitizers in certain flows.

AffectedSinks: What are sinks that would be no longer reachable if this API X becomes a sanitizer?

API: 9749 - java.lang.CharSequence.toString ()

to act as a sanitizer starting from Source Node: 0 - form : String org.apache.du

Identifies sinks that can be affected by the removal specific API.

http://localhost:3000/cytoscapePopup.html 67%

Focus on Node

Select Node: node_18 - getHostName(...) : String org.apache.dubb

Focus on Edge

Select Edge: bc63d694-b3f3-4264-a6a7-efa7c435065f - undefin

Description: getTags(...) org.apache.dubbo.metrics.config.collector.ConfigCenterMetricsCollector.java:96

Figure 1: TRACELENS supports interrogative debugging by enabling a user to ask *why*, *why-not*, and *what-if* questions about taint-analysis. A user can select one among templated queries (shown in `Q`) and contextualize their inquiry with respect to a specific source, sink, and 3rd 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-code annotation.

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 TRACELENS (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 about "which sinks would no longer be reachable if third-party library models were to be configured as a sanitizer from source to sink?"

Visualization of Global Impact. Once a user initiates a query within TRACELENS, specifying their target source, sink, and any relevant external API calls, TRACELENS 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 facilitate sensemaking, TRACELENS employs a color-coded scheme: source nodes are highlighted in green, sink nodes in red, and external library nodes in orange. This color differentiation draws attention to the impact of configuration choices. Furthermore, templated queries are designed to illuminate how which third-party library's model can impact multiple sinks, multiple sources, etc, making global impact salient through visualization.

User study. We conducted a within-subject study with a factorial crossover on 12 participants (graduate students and professional developers), who inspected taint-analysis warnings generated by CodeQL [4], [5]. Each participant answered eight questions per task (16 total) designed to reflect a realistic sensemaking process of tool-generated warnings. During these tasks, a user had to answer questions about the impact of third-party library models on taint analysis results, by identifying pass-through API calls, APIs that serve as sanitizers, etc. Participants increased the average question completion rate from 71% with the baseline CodeQL visualizer to 92% with TRACELENS, especially on questions that require global reasoning of multiple taint flows. When using TRACELENS, users were more accurate in identifying multiple sinks affected by the same third-party library model. For example, 50% of answers prepared with TRACELENS are correct, whereas only 17% with the baseline.

We measured cognitive load via NASA-TLX questions. We found that TRACELENS led to improvement on the participants self-reported mental demand, stress, and success rate. Considering the technology acceptance model [6], participants rated TRACELENS higher on both *Confidence* (mean 4.3 vs. 2.1) and *Ease of Use* (4.3 vs. 1.9) than the baseline CodeQL. Qualitative feedback highlighted that color-coded graphs and dedicated template queries helped alleviate *analysis paralysis* [7]. Overall, these results underscore that TRACELENS substantially enhances users' ability to make sense of complex taint paths and reduce their mental workload through QA-based end-user 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, TRACELENS is the first interactive end-user debugger that enables the investigation of how different user configuration choices in modeling third-party libraries impact the taint analysis results.
2) TRACELENS 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. TRACELENS's graphical interface eases the examination of 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 TRACELENS's effectiveness in sensemaking CodeQL-generated taint analysis warnings [4], [5]. TRACELENS 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 case study with two professionals corroborates this improvement in accuracy, confidence, and ease of use.

The remainder of this paper is organized as follows. Section II introduces a motivating example and the design goals of TRACELENS. Section III presents TRACELENS's approach. Section IV provides the study design and results from our within-subject user study. Section V reports the results we obtained from our user study. Section VI discusses implications of our findings, and possible threats to validity. Section VII presents related work. Finally, we draw the conclusions of our work in Section VIII.

II. MOTIVATION

When using modern static taint analyzers such as CodeQL [5], 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 expectations. This paper is concerned with the 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 to easily make sense of unexpected results.

First, traditional tools lack support for effective 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. A robust QA process, focused on sensemaking, is therefore essential to bridge this gap to enable active investigation rather than passively viewing the flow.

Second, beyond isolated flow views, developers require support for reasoning about the model's impact on multiple dataflows. Current tools fall short, lacking the ability to easily query these complex relationships, or predict the ripple effects of model changes. This forces developers to manually piece together disparate flows, hindering them from understanding the broader consequences of modifications and leaving unreported flows completely opaque.

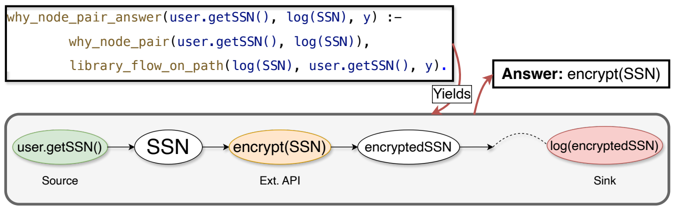
Prior debugging introduced *WhyLine* [8] to let users ask *"why did"* or *"why didn't"* on a program trace, where the context of debugging is on runtime events (e.g., runtime errors). However, *"why and why-not"* questions remain largely unaddressed in the context of modern taint analysis tools. Developers want customizability over taint analysis [9], visual outputs of warnings and code [10], and the ability to inspect intermediate pass through nodes of the analysis [8]. Yet, typical dataflow or taint analysis interfaces do not systematically address 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 I 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) [9], [10], [8]. She suspects the culprit is an imprecise third-party library model; however, manually tracing a long chain of calls is overwhelming. The WHYFLOW query highlights in one view 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 then notices an intermediate API call is `encrypt(SSN)`. She then discovers that this API

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

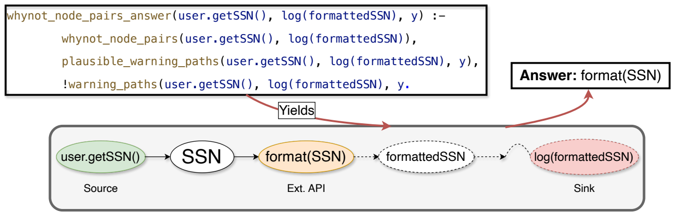
Query Type	Plain-English Question	Logic Query Interpretation
WhyFlow	"Why is there a taint flow from a source X to a sink Y?"	"Which third-party library models (or assumptions) currently allow data to propagate from X to Y?"
WhyNotFlow	"Why is there *no* taint flow from a source X to a sink Y?"	"Which 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?"



```
why_node_pair_answer(user.getSSN(), log(SSN), y) :-
        why_node_pair(user.getSSN(), log(SSN)),
        library_flow_on_path(log(SSN), user.getSSN(), y).
```

Answer: encrypt(SSN)

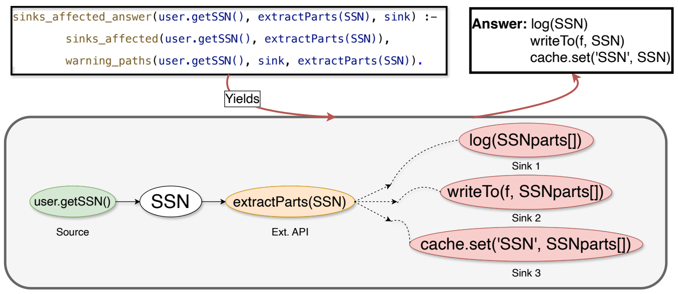
Figure 2: WhyFlow: "Why is there a taint flow from a source to a sink?"



```
whynot_node_pairs_answer(user.getSSN(), log(formattedSSN), y) :-
        whynot_node_pairs(user.getSSN(), log(formattedSSN)),
        plausible_warning_paths(user.getSSN(), log(formattedSSN), y),
        !warning_paths(user.getSSN(), log(formattedSSN), y.
```

Answer: format(SSN)

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



```
sinks_affected_answer(user.getSSN(), extractParts(SSN), sink) :-
        sinks_affected(user.getSSN(), extractParts(SSN)),
        warning_paths(user.getSSN(), sink, extractParts(SSN)).
```

Answer: log(SSN)
writeTo(f, SSN)
cache.set('SSN', SSN)

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

is a sanitizer, and thus this warning should have *not* reported and `encrypt()`'s model should be debugged.

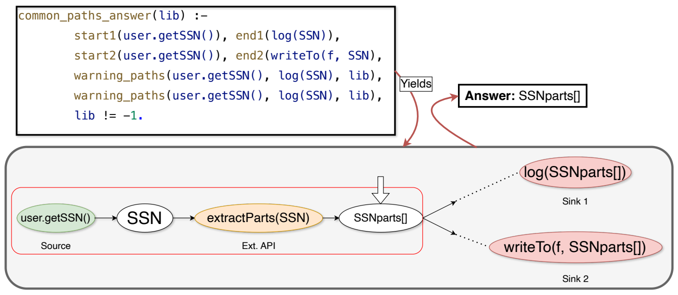
2. WhyNotFlow: Why is there no taint flow from a source X to a sink Y? Suppose that Alice needs to investigate a missing taint flow (i.e., a bug arises at a sink, yet CodeQL does not issue a corresponding warning). She suspects that a third-party library is mistakenly modeled as a *sanitizer*. In Figure 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.

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 `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 [11]. 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* TRACELENS, 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. Figure 4 shows how AFFECTEDSINKS automates this what-if analysis, immediately revealing all red sinks that would be "killed.".

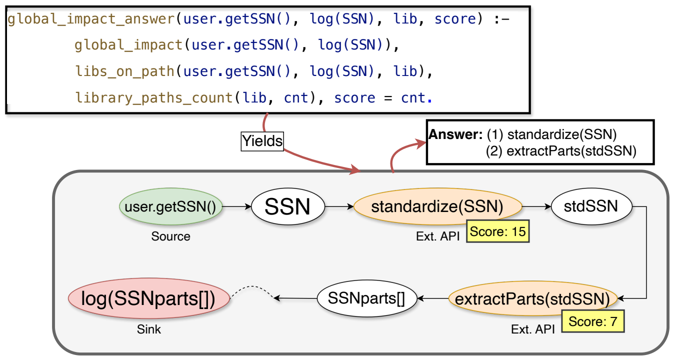
4. DivergentSinks & 5. DivergentSources: Which third party library model could influence multiple taint flows reaching the same sink (or originating from the same source)?

Suppose that Alice would like to know whether a known vulnerability reaching multiple *sinks* could be fixed at once [11], [12]. 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, developers may not 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).



```
common_paths_answer(lib) :-
        start1(user.getSSN()), end1(log(SSN)),
        start2(user.getSSN()), end2(writeTo(f, SSN),
        warning_paths(user.getSSN(), log(SSN), lib),
        warning_paths(user.getSSN(), log(SSN), lib),
        lib != -1.
```

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



```
global_impact_answer(user.getSSN(), log(SSN), lib, score) :-
        global_impact(user.getSSN(), log(SSN)),
        libs_on_path(user.getSSN(), log(SSN), lib),
        library_paths_count(lib, cnt), score = cnt.
```

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

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, TRACELENS provides a visual representation of where these multiple taint paths intersect, which makes pinpointing the culprit model significantly easier.

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 missing warning, developers often prefer a conservative fix that impacts the fewest flows [13]. *Without* TRACELENS, identifying the frequency of each API in *all* taint paths would require her 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.

III. SPECULATIVE ANALYSIS FOR INQUIRY-BASED DEBUGGING

TRACELENS enables inquiry-based debugging for sense-making to debug spurious flows or missing flows. As presented in Figure 7, we execute both the CodeQL taint query as well as a less restricted version of this query (Sections III-A), convert the taint analysis results into *Soufflé* facts and store them in a database (Section III-B). Subsequently, TRACELENS queries the database to provide answers to the questions posed using TRACELENS, according to the user’s choice (Section III-C). The answers are presented as an interactive graph visualization. In this section, we present the components of TRACELENS, and how each of these components help the overall workflow. Before we elaborate on our approach, we state several assumptions in our design.

Assumption 1. We presume that a user of TRACELENS is familiar with the subject program that she is inspecting, meaning that they may recognize that some flows are spurious, unexpected flows and some expected flows are missing, serving a starting point for end-user debugging. This way, running an interrogative debugger can help narrow down the third-party library’s model after selecting a relevant template query for ‘why’ or ‘why-not’ questions.

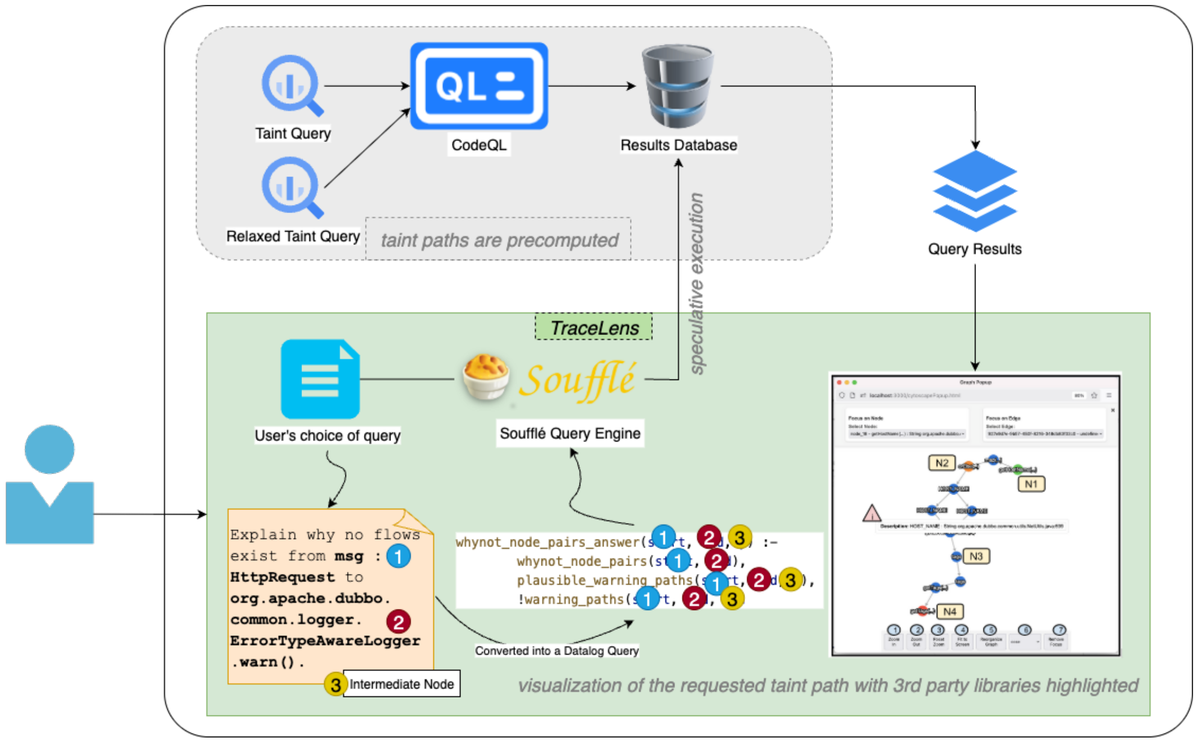
Assumption 2. We leverage CodeQL’s taint analysis as is obtain taint analysis results, and then we utilize Soufflé to run the interrogative debugging queries on top of CodeQL results. Hence, it is important to emphasize that CodeQL and Soufflé are being used together, and there is no modification to the underlying CodeQL’s analysis algorithm. These two tools are not meant to compete.

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. Since these models are plug-in abstractions and third party libraries that may not have source code, we assume that inaccuracies in these plug-in models should be debugged, when a user suspects a missing flow or a spurious flow.

Assumption 4. The goal of TRACELENS is not to frame ‘why’ and ‘why-not’ question asking as a classification problem of false positive or false negative. In fact, TRACELENS never removes or adds any taint warnings reported by CodeQL. Rather, TRACELENS helps users with end-user debugging—how their configuration choices such as source/sink/sanitizer definitions and third-party models may lead to spurious or missing paths. Given that the target users need to maintain and update third-party models, TRACELENS aims to simplify this process.

A. CodeQL

CodeQL is an open-source static analysis framework, developed and maintained by GitHub. Users can use a specialized Domain-Specific Language (DSL) to detect potential security vulnerabilities, code quality issues, or other custom patterns [14]. While CodeQL offers code scanning capabilities out of the box, its extendability via user-defined queries and models is particularly noteworthy.



```
Explain why no flows
exist from msg : 1
HttpRequest to
org.apache.dubbo.
common.logger.      2
ErrorTypeAwareLogger
.warn().
```

```
whynot_node_pairs_answer(s1rt, 2d,3) :-
        whynot_node_pairs(st1t, 2d),
        plausible_warning_paths(s1rt, 2d,3),
        !warning_paths(s1rt, 2d, 3
```

Figure 7: A user selects a *WhyNotFlow* template query to investigate a suspected missing flow `msg:HttpRequest` to `ErrorLogger.warn()`. TRACELENS 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*.

A key factor in CodeQL's precision and 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. Recognizing that manual creation of such models is time-consuming and error-prone, the CodeQL team has explored automated and machine-learning-based techniques to infer library behaviors [9]. Nonetheless, incomplete or incorrect modeling remains a practical challenge, especially in large ecosystems where libraries evolve frequently without thorough documentation of their input-output contracts.

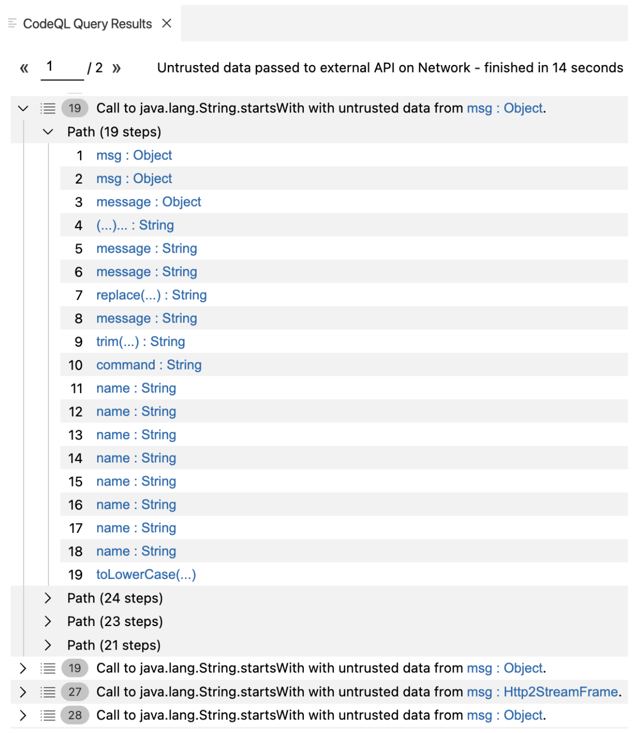
Rather than relying on specific third-party library models and source and sink definitions, a general query may relax these configuration assumptions, revealing the "maximal" set of reachable flows from all possible sinks to all possible sources, ignoring third-party library models. Such an approach 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 sanitizers or third-party methods with pre-defined template queries.

B. Soufflé and Logic Query Implementation

TRACELENS 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, TRACELENS can quickly re-evaluate multiple "secondary" queries, such as those in Table I, 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 [15] to query the precomputed facts in the order of 5 seconds—a speed-up that supports interactive debugging.

a) 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



```
CodeQL Query Results ×
« 1 /2 »   Untrusted data passed to external API on Network - finished in 14 seconds
19 Call to java.lang.String.startsWith with untrusted data from msg : Object.
  Path (19 steps)
    1  msg : Object
    2  msg : Object
    3  message : Object
    4  (...)... : String
    5  message : String
    6  message : String
    7  replace(...) : String
    8  message : String
    9  trim(...) : String
    10 command : String
    11 name : String
    12 name : String
    13 name : String
    14 name : String
    15 name : String
    16 name : String
    17 name : String
    18 name : String
    19 toLowerCase(...)
  Path (24 steps)
  Path (23 steps)
  Path (21 steps)
19 Call to java.lang.String.startsWith with untrusted data from msg : Object.
27 Call to java.lang.String.startsWith with untrusted data from msg : Http2StreamFrame.
28 Call to java.lang.String.startsWith with untrusted data from msg : Object.
```

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 3rd-party library models.

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`*`)`.

b) Answering "WhyFlow" and "WhyNotFlow": Once loaded into Soufflé, the questions in Table I 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, effectively terminating any flow.

c) Supporting Divergent Sources/Sinks and Global Impact.: Queries like *DivergentSinks* or *DivergentSources* use logic rules to detect the last common node or first common node in two paths. We do so by computing the intersection of reachable sets for each source/sink pair. Meanwhile, *GlobalImpact* queries pivot on counting how frequently an API appears in distinct `reachable` flows; each API node is assigned a score via an aggregation rule over the factbase. The resulting integer is used to size the node (in the visualization) and thus show its "global impact."

C. TRACELENS's User Interface

To support sensemaking, key design elements in TRACELENS include:

a) 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.

b) Graphical Flows and Expandable Paths.: TRACELENS 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.

c) 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. TRACELENS includes hover popups showing details such as fully-qualified names of identifiers referenced by the intermediate steps along the flow.

d) Customizable Layouts and Node Sizing.: TRACELENS supports multiple layout algorithms (e.g., breadth-first, concentric). Users can reorganize the graph for improved clarity. A *GlobalImpact* query resizes API nodes by their number of occurrences on different flows.

IV. USER STUDY

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

a) 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 in Table III. The first six questions centered on why, why-not, and what-if questions about taint flows and the remaining two centered on quantifying taint flows.

b) Research Questions:

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

A. Study Protocol

The study task is based on 383 taint warnings generated on Apache Dubbo with the CodeQL query shown in Listing 1 [16]. The resulting facts from this query are presented in Table II.

Metric	Value
# of edges	6,901
# of nodes	8,101
# of sources	26
# of sinks	265
# of 3rd-party API functions	85

Table II: Statistics of Taint Analysis Facts for `Apache Dubbo`.

```
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 [16].

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.

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 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 they failed to complete half of both tasks. The average self-reported familiarity with taint analysis was 2.1 out of 5.

a) 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 [17], [18]. Within-subject user studies with 8 to 16 participants are standard practice in both software engineering [19], [20], [21] and HCI research [22], [23], [24].

b) Use of student participants.: While most of our study's participants are students, prior work found that the findings based on student participants often generalize to a broader population as students have comparable performance to professionals in security-related tasks [25], [26], [27], [28]. Other studies [29], [30], [31] also emphasize that students can provide meaningful feedback when their expertise aligns with the tool's target audience. Since TRACELENS is designed for end-user debugging, the use of student participants is methodologically sound. We also conducted a case study with two security professional participants, discussed in Section V-D. Also, the goal of TRACELENS is to help end-users who are not necessarily security experts. TRACELENS helps them to sensemake taint analysis results and debug them with our interrogative queries.

3) Protocol: Each participant took part in a 1.5-hour session. The study involved using both TRACELENS and the baseline CodeQL visualizer and the two tasks. The order of assigned tool (TRACELENS 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 to 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 post-task survey, which included the NASA TLX (5 minutes total). In the final 12-minute post-study survey, we asked participants to compare TRACELENS and CodeQL, and rate TRACELENS's features.

4) Tutorial: We conducted a tutorial session to introduce participants to the functionality of both TRACELENS and the CodeQL VSCode plugin. The tutorials were structured around realistic dataflow scenarios. For TRACELENS, 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 TRACELENS. Participants were encouraged to hover over nodes to view fully qualified names, click on nodes to navigate directly to code locations, and adjust the layout using built-in options. Through these interactions, participants familiarized themselves with TRACELENS's features.

We also conducted a tutorial session on using CodeQL's visualizer. Participants learned how the plugin displays query results in a tabular list. We walked them through the process of expanding or collapsing the flows presented in the visualizer to inspect the intermediate steps between source to sink.

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

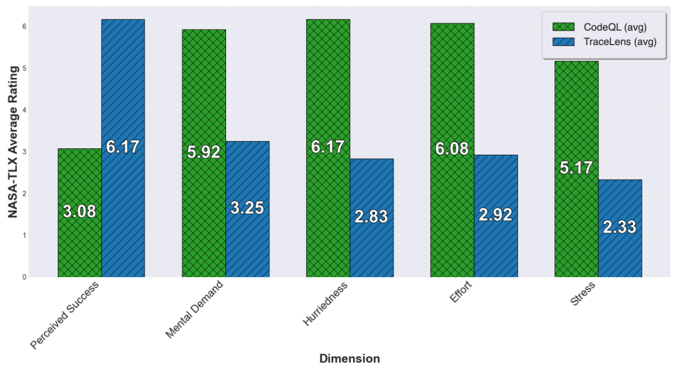


Figure 9: Average NASA-TLX Ratings for TRACELENS vs. CodeQL visualizer.

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 III. To minimize learning effects, we balanced the order of tools that participants used first. Each task contained eight questions about 'why', 'why-not', 'what-if', shown in Table I and the other two about 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 had multiple answers. We granted partial credit if participants selected some correct answers, but missed or added incorrect choices.

6) Criteria for Study Tasks: We designed questions based on realistic scenarios involving the analysis of flows from sources to sinks. 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 external libraries. The tasks require pinpointing of sanitizers and pass-through APIs and assessing how many flows or nodes are present.

a) Post-Study Questionnaire.: After completing each task, participants completed a brief post-task questionnaire, which included the NASA TLX [32] (e.g., mental demand, time pressure, perceived success, effort, and frustration) and described their strategies to find answers to the questions. At the end of the session, participants rated user-interface features, confidence, and ease of use.

V. RESULTS

In this section, we analyze the results of our user study. We assess the statistical significance of our findings with the *Mann-Whitney U test* [33], [34] given the non-normal distribution of our data and its ordinality (e.g., Likert-scale responses), which is common for user studies [35], [36], [37], [38], [39]. Each participant is denoted as P#.

A. RQ1. Accuracy

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

These differences are statistically significant according to the Mann-Whitney U test [40] with a large effect size (*Correctness:* $Cohen's\ d = 1.33$, *Empty answers:* $Cohen's\ d = 1.8$). Participants using TRACELENS provided more correct answers, especially when needing to reason about multiple flows and global impact; on the other hand, participants using the CodeQL visualizer felt overwhelmed [7].

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

B. RQ2. Cognitive Load and Confidence

After each task, participants filled in the NASA-TLX, 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. Overall, participants found completing the task with TRACELENS 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 required 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).

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 TRACELENS's approach as "very clear," and P11 noted it was "easier to track the dataflow." These comments reinforce our findings that TRACELENS's inquiry-based debugging reduced the cognitive overhead of analyzing multiple interconnected flows. Most participants preferred TRACELENS's graph-based interface over the default list-view of CodeQL visualizer. The participants identified areas for improvement for TRACELENS. P9 mentioned 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 [TRACELENS]," 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.

Program Point Set #1	**Program Point 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.meth`	(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`. **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 [...].NettyBackedChannelBuffe`	(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.h` `(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 III: Study Tasks: Program 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 TRACELENS.

	TRACELENS		CodeQL visualizer	
Q	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 IV: User Study Results: Percentage of participants providing (C)orrect/(E)mpty answers when using TRACELENS vs. CodeQL visualizer.

Using TRACELENS, 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.

C. RQ3. Usability and Workflow

1) Interface Features: Participants rated the usefulness of TRACELENS's interface beyond its core queries. Participants rated color-coding with the highest average score (4.83). 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 expandable taint paths (with an average score of 3.33) 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), mainly using it to confirm ambiguous method names.

2) Confidence and Ease of Use: Table V shows participants' ratings of TRACELENS and CodeQL visualizer in terms of confidence (in the answers to the questions) and overall ease of use. TRACELENS scored 4.25 on both measures, substantially higher than CodeQL visualizer (2.08 for confidence, 1.92 for ease). These improvements were statistically significant ($p < 0.05$, Cohen's D > 2).

Multiple participants appreciated TRACELENS'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.

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

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

3) Future Adoption: When asked if they would like to use TRACELENS in future taint-flow inspections, participants gave an average rating of 4.83. 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—TRACELENS's approach can save time and reduce errors."*

```
.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. Using the raw three-arity `edge(id,src,dst)` input, we declare `branch(n)` for each n that appears as `src` in more than one `edge(_,n,_)` fact.

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

D. Case Study

To reinforce our findings from the perspective of industry professionals, we conducted studies with two **security** professionals in addition to our within-subject study with 12 participants. We used the same protocol from Section IV-A. Both reported 5/5 familiarity with static taint analysis and had 7 to 10 years of experience.

Both participants (P101 and P102) provided more correct answers with TRACELENS than CodeQL. They liked the workflow of inspecting taint flows using TRACELENS. P101 commented, *"CodeQL requires lots of manual labor, whereas* TRACELENS*'s visual components make it easier to navigate." "That [the workflow for using* TRACELENS*] was very intuitive, the tooling eases manual inspection load completely."* P102 mentioned *"With CodeQL, I wonder whether I'm missing another row that describes the same location."* Both participants reported higher confidence and ease of use using TRACELENS, achieving the same high level of task success with both tools (6 out of 7), while TRACELENS consistently felt easier to use. For P102, mental demand was equally low for both tools but they felt more rushed with TRACELENS (4/7 vs. 2/7), even though it required less effort (2/7 vs. 3/7) and caused no additional frustration. P101 experienced a reduction in mental demand (2/7 vs. 7/7), pace pressure (1/7 vs. 7/7), and effort (2/7 vs. 5/7) using TRACELENS, with a slight decrease in frustration. In both cases, TRACELENS matched CodeQL in perceived success, while lowering cognitive burden.

VI. DISCUSSION

a) Improved Accuracy with Visual Queries.: Participants using TRACELENS achieved better accuracy, especially when analyzing multiple flows. In particular, the AffectedSinks and GlobalImpact queries require the analysis of multiple or interconnected flows. Surprisingly, participants were able to answer questions related to "why-not" using both tools. In contrast, participants using the CodeQL visualizer responses frequently had empty submissions due to lack of time, suggesting viewing results one by one a list view is time-consuming. This shows that an interactive, inquiry-based debugger of taint analysis warnings has the potential to improve sensemaking for large result sets, which is known to hinder adoption [10].

b) Suggestions for Enhancement.: While participants found TRACELENS "less cumbersome" (P2) for analysis involving multiple flows, they offered suggestions for improving TRACELENS. They recommended adding an on-screen legend to clarify colors and dotted edges, and using distinct shapes for different node types (e.g., sources versus sinks) 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.

c) Extensibility of Template Queries.: While TRACELENS supports six template questions from Table I, adding more template queries is easy since TRACELENS's template questions are Datalog-style queries. As a case study, adding a new template query called 'divergent path' query to TRACELENS (Listing 2) took under ten minutes. This query identifies the *branch points*, which are locations where the flow branches into multiple targets. Note that TRACELENS's queries supports any taint analysis results from CodeQL. More extension queries are available in our replication package (https://github.com/tracelens/TraceLens/tree/main/data/extension_queries).

d) Supporting Interactivity.: TRACELENS supports interrogative and speculative debugging built on a rich history of using logic programming for software comprehension [41], [42], [43], [44], [45], [46]. Soufflé is the state of the art logic query engine. TRACELENS does not compete with underlying taint analysis engines but instead emphasizes *sensemaking* support. We pay a one-time cost of fact extraction from CodeQL's result to support TRACELENS's interactive features.

e) Threats to Validity.: A threat to validity is the size of our user study. While our user study involved only 12 participants, it employs a within-subject crossover design, thereby reducing variance and requiring fewer participants [17], [47]. While our study evaluated TRACELENS only against a single baseline (CodeQL) and and a single subject program (Apache Dubbo), our approach is not tied to CodeQL, and can generalize to any taint analysis.

VII. RELATED WORK

Speculative What-If Analysis. Speculative execution allows the investigation of future or alternative actions developers may perform. Brun et al. explored its application for providing fix suggestions [48], as well as identifying conflicts during version merging [49]. Our work is the first to apply it to taint analysis for reasoning about permissible dataflows of sensitive information under different configuration of different dataflow connectivity.

Templated Questions for asking questions. Developers often ask questions during variety of developer tasks [50]. In particular, developers need support for understanding code reachability [8], exploring how different vulnerabilities relate and finding similar ones [11], tracking intermediate states of analysis [51], reasoning about system-wide implications when making changes [11]. TRACELENS is the first work providing templated questions for investigating and debugging the results of taint analysis.

Modelling 3rd party libraries. Existing work [52], [53], [54] automatically infers models of third-party libraries. However, they do not guide users to understand the impact of these models. They do not support end-user debugging and do not allow end-users to ask questions about the impact of modelling choices. While Paralib [55] allows comparison between multiple choices of libraries, 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 [56], [57], [58] is about the problem of investigating the root cause of tool outcomes, and has been applied to spreadsheet debugging, search engine configuration, etc. Similarly, TRACELENS is an end-user debugger for reasoning how taint flows are affected by the configuration of sources, sinks, and models of third-party libraries.

VIII. CONCLUSION

We presented TRACELENS, a tool for end-user debugging for taint analysis. Through speculative analysis, TRACELENS 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. TRACELENS enables sensemaking of taint analysis and helps users identify root causes of unexpected flows or missing flows. Our user study confirms that TRACELENS helps users provide 21% more correct answers to questions about the taint analysis, while experiencing a lower cognitive load. These results suggest that TRACELENS's inquiry-based approach empowers developers to analyze the outputs of taint analysis.

IX. DATA AVAILABILITY

The replication package and study's data are available. at https://github.com/tracelens/TraceLens/tree/main/data/extension_queries.

REFERENCES

[1] 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*, ser. PLDI 2019. New York, NY, USA: Association for Computing Machinery, Jun. 2019, p. 760–774. [Online]. Available: https://dl.acm.org/doi/10.1145/3314221.3314648

[2] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee, "Merlin: specification inference for explicit information flow problems," *SIGPLAN Not.*, vol. 44, no. 6, p. 75–86, Jun. 2009. [Online]. Available: https://dl.acm.org/doi/10.1145/1543135.1542485

[3] S. Banerjee, S. Cui, M. Emmi, A. Filieri, L. Hadarean, P. Li, L. Luo, G. Piskachev, N. Rosner, A. Sengupta, O. Tripp, and J. Wang, "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*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, Nov. 2023, p. 1985–1996. [Online]. Available: https://dl.acm.org/doi/10.1145/3611643.3613889

[4] P. Avgustinov, O. De Moor, M. P. Jones, and M. Schäfer, "Ql: Object-oriented queries on relational data," in *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2016.

[5] T. Szabó, "Incrementalizing production codeql analyses," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1716–1726.

[6] Y. Lee, K. A. Kozar, and K. R. Larsen, "The technology acceptance model: Past, present, and future," *Communications of the Association for information systems*, vol. 12, no. 1, p. 50, 2003.

[7] B. Schwartz, "The paradox of choice," *Positive psychology in practice: Promoting human flourishing in work, health, education, and everyday life*, pp. 121–138, 2015.

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

[9] M. Nachtigall, M. Schlichtig, and E. Bodden, "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*, 2022, pp. 532–543.

[10] 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)*. IEEE, 2013, pp. 672–681.

[11] J. Smith, B. Johnson, E. Murphy-Hill, B. Chu, and H. R. Lipford, "Questions developers ask while diagnosing potential security vulnerabilities with static analysis," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 248–259.

[12] G. Piskachev, L. N. Q. Do, and E. Bodden, "Codebase-adaptive detection of security-relevant methods," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 181–191.

[13] E. Murphy-Hill, T. Zimmermann, C. Bird, and N. Nagappan, "The design of bug fixes," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 332–341.

[14] C. developers, "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.

[15] H. Jordan, B. Scholz, and P. Subotić, "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, 2016, pp. 422–430.

[16] C. developers, "CodeQL query: Untrusteddata-toexternalapi," https://github.com/github/codeql/blob/996bc47ae8d10f6087504413db02c8920243b13e/java/ql/src/Security/CWE/CWE-020/UntrustedDataToExternalAPI.ql.

[17] S. Vegas, C. Apa, and N. Juristo, "Crossover designs in software engineering experiments: Benefits and perils," *IEEE Transactions on Software Engineering*, vol. 42, no. 2, pp. 120–135, 2015.

[18] S. E. Chasins, E. L. Glassman, and J. Sunshine, "Pl and hci: better together," *Commun. ACM*, vol. 64, no. 8, p. 98–106, Jul. 2021. [Online]. Available: https://doi.org/10.1145/3469279

[19] E. J. Arteaga Garcia, J. a. F. Nicolaci Pimentel, Z. Feng, M. Gerosa, I. Steinmacher, and A. Sarma, "How to support ml end-user programmers through a conversational agent," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3608130

[20] H. J. Kang, K. Wang, and M. Kim, "Scaling code pattern inference with interactive what-if analysis," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639193

[21] M. Ganji, S. Alimadadi, and F. Tip, "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*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 1307–1319. [Online]. Available: https://doi.org/10.1145/3611643.3616292

[22] A. Horvath, B. Myers, A. Macvean, and I. Rahman, "Using annotations for sensemaking about code," in *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3526113.3545667

[23] S. Suh, B. Min, S. Palani, and H. Xia, "Sensecape: Enabling multilevel exploration and sensemaking with large language models," in *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3586183.3606756

[24] M. Huh and A. Pavel, "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*, ser. UIST '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3654777.3676369

[25] G. Sandoval, H. Pearce, T. Nys, R. Karri, S. Garg, and B. Dolan-Gavitt, "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*, ser. SEC '23. USA: USENIX Association, 2023.

[26] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky, "You get where you're looking for: The impact of information sources on code security," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 289–305.

[27] Y. Acar, C. Stransky, D. Wermke, M. L. Mazurek, and S. Fahl, "Security developer studies with GitHub users: Exploring a convenience sample," 2017, p. 81–95. [Online]. Available: https://www.usenix.org/conference/soups2017/technical-sessions/presentation/acar

[28] I. Salman, A. T. Misirli, and N. Juristo, "Are students representatives of professionals in software engineering experiments?" in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, pp. 666–676.

[29] A. Naiakshina, A. Danilova, C. Tiefenau, M. Herzog, S. Dechand, and M. Smith, "Why do developers get password storage wrong? a qualitative usability study," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 311–328. [Online]. Available: https://doi.org/10.1145/3133956.3134082

[30] A. Naiakshina, A. Danilova, E. Gerlitz, and M. Smith, "On conducting security developer studies with cs students: Examining a password-storage study with cs students, freelancers, and company developers," ser. CHI '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1–13. [Online]. Available: https://doi.org/10.1145/3313831.3376791

[31] A. J. Ko, T. D. LaToza, and M. M. Burnett, "A practical guide to controlled experiments of software engineering tools with human participants," *Empirical Software Engineering*, vol. 20, no. 1, p. 110–141, Feb. 2015.

[32] S. G. Hart and L. E. Staveland, "Development of nasa-tlx (task load index): Results of empirical and theoretical research," in *Advances in psychology*. Elsevier, 1988, vol. 52, pp. 139–183.

[33] D. Ashby, "Practical statistics for medical research. douglas g. altman, chapman and hall, london, 1991. no. of pages: 611. price: £32.00," *Statistics in Medicine*, vol. 10, no. 10, p. 1635–1636, 1991.

[34] Student, "The probable error of a mean," *Biometrika*, vol. 6, no. 1, p. 1–25, 1908.

[35] G. Bavota, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "An empirical study on the developers' perception of software coupling," in *2013 35th International Conference on Software Engineering (ICSE)*, May 2013, p. 692–701. [Online]. Available: https://ieeexplore.ieee.org/document/6606615

[36] X. Yu, F. R. Cogo, S. McIntosh, and M. W. Godfrey, "Studying the impact of risk assessment analytics on risk awareness and code review performance," *Empirical Software Engineering*, vol. 29, no. 2, p. 46, Feb. 2024.

[37] E. Fast, B. Chen, J. Mendelsohn, J. Bassen, and M. S. Bernstein, "Iris: A conversational agent for complex tasks," in *Proceedings of the 2018*

CHI Conference on Human Factors in Computing Systems, ser. CHI '18. New York, NY, USA: Association for Computing Machinery, Apr. 2018, p. 1–12. [Online]. Available: https://doi.org/10.1145/3173574.3174047

[38] Z. Liu, C. Chen, J. Wang, M. Chen, B. Wu, Y. Huang, J. Hu, and Q. Wang, "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*, ser. CHI '24. New York, NY, USA: Association for Computing Machinery, May 2024, p. 1–20. [Online]. Available: https://dl.acm.org/doi/10.1145/3613904.3642939

[39] N. Warford, C. W. Munyendo, A. Mediratta, A. J. Aviv, and M. L. Mazurek, "Strategies and perceived risks of sending sensitive documents," 2021, p. 1217–1234. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/warford

[40] H. B. Mann and D. R. Whitney, "On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other," *The Annals of Mathematical Statistics*, vol. 18, no. 1, pp. 50 – 60, 1947. [Online]. Available: https://doi.org/10.1214/aoms/1177730491

[41] R. C. Holt, "Structural manipulations of software architecture using tarski relational algebra," in *Proceedings of the Working Conference on Reverse Engineering (WCRE'98)*, ser. WCRE '98. USA: IEEE Computer Society, 1998, p. 210.

[42] K. Mens, T. Mens, and M. Wermelinger, "Maintaining software through intentional source-code views," in *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*, ser. SEKE '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 289–296. [Online]. Available: https://doi.org/10.1145/568760.568812

[43] E. Hajiyev, M. Verbaere, O. de Moor, and K. de Volder, "Codequest: querying source code with datalog," in *Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 102–103. [Online]. Available: https://doi.org/10.1145/1094855.1094884

[44] M. Eichberg, S. Kloppenburg, K. Klose, and M. Mezini, "Defining and continuous checking of structural program dependencies," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 391–400. [Online]. Available: https://doi.org/10.1145/1368088.1368142

[45] Y.-G. Guéhéneuc and G. Antoniol, "Demima: A multilayered approach for design pattern identification," *IEEE Transactions on Software Engineering*, vol. 34, no. 5, pp. 667–684, 2008.

[46] T. Tourwe and T. Mens, "Identifying refactoring opportunities using logic meta programming," in *Seventh European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings.*, 2003, pp. 91–100.

[47] D. Gergle and D. S. Tan, *Experimental Research in HCI*. New York, NY: Springer, 2014, p. 191–227. [Online]. Available: https://doi.org/10.1007/978-1-4939-0378-8_9

[48] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Speculative analysis: exploring future development states of software," in *Proceedings of the FSE/SDP workshop on Future of software engineering research*, 2010, pp. 59–64.

[49] ——, "Proactive detection of collaboration conflicts," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 168–178.

[50] A. Begel and T. Zimmermann, "Analyze this! 145 questions for data scientists in software engineering," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 12–23.

[51] L. N. Q. Do, S. Krüger, P. Hill, K. Ali, and E. Bodden, "Debugging static analysis," *IEEE Transactions on Software Engineering*, vol. 46, no. 7, pp. 697–709, 2018.

[52] Z. Li, S. Dutta, and M. Naik, "Llm-assisted static analysis for detecting security vulnerabilities," *arXiv preprint arXiv:2405.17238*, 2024.

[53] G. Piskachev, L. N. Q. Do, O. Johnson, and E. Bodden, "Swan_assist: semi-automated detection of code-specific, security-relevant methods," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1094–1097.

[54] W.-H. Chiang, P. Li, Q. Zhou, S. Banerjee, M. Schaef, Y. Lyu, H. Nguyen, and O. Tripp, "Inference for ever-changing policy of taint analysis," in *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '24. New York, NY, USA: Association for Computing Machinery, May 2024, p. 452–462. [Online]. Available: https://dl.acm.org/doi/10.1145/3639477.3639738

[55] L. Yan, M. Kim, B. Hartmann, T. Zhang, and E. L. Glassman, "Concept-annotated examples for library comparison," in *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*, 2022, pp. 1–16.

[56] M. Burnett, C. Cook, and G. Rothermel, "End-user software engineering," *Communications of the ACM*, vol. 47, no. 9, pp. 53–58, 2004.

[57] C. Kissinger, M. Burnett, S. Stumpf, N. Subrahmaniyan, L. Beckwith, S. Yang, and M. B. Rosson, "Supporting end-user debugging: what do users want to know?" in *Proceedings of the working conference on Advanced visual interfaces*, 2006, pp. 135–142.

[58] V. Grigoreanu, M. Burnett, S. Wiedenbeck, J. Cao, K. Rector, and I. Kwan, "End-user debugging strategies: A sensemaking perspective," *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 19, no. 1, pp. 1–28, 2012.