Inference for Ever-Changing Policy of Taint Analysis

Wen-Hao Chiang
cwenhao@amazon.com
Amazon Web Services
USA

Peixuan Li
peixuan@amazon.com
Amazon Web Services
USA

Qiang Zhou
zhouqia@amazon.com
Amazon Web Services
USA

Subarno Banerjee
subarno@amazon.com
Amazon Web Services
USA

Martin Schaef
schaef@amazon.com
Amazon Web Services
USA

Yingjun Lyu
yingjunl@amazon.com
Amazon Web Services
USA

Hoan Nguyen
hoanamzn@amazon.com
Amazon Web Services
USA

Omer Tripp
omertrip@amazon.com
Amazon Web Services
USA

ABSTRACT

Identifying correct and complete taint specifications is critical for detecting vulnerabilities in the ever-changing landscape of software security, and an automated scalable and practical solution remains elusive in the field. In this paper, we report our semi-automated scheme for inferring and maintaining taint specifications at industrial scale. *Knowledge graph* is adopted as the core engine to represent the ongoing accumulation of knowledge in the domain of software security: how different functional behaviors in programs relate and manifest in varying contexts of many security vulnerabilities and their defenses. Taint analysis rules are then mapped onto nodes in the knowledge graph to achieve the desired security enforcement. We begin by *mining* from a corpus of existing code analysis tools and code examples from the wild to discover candidate taint specifications, followed by human-in-the-loop *labeling* to assign concrete APIs to nodes in the knowledge graph. To continuously grow the knowledge graph, we propose a novel *inference* algorithm using multi-view active machine learning approach to characterize taint-relevant APIs via collective matrix factorization which combines different aspects of API use-pattern and its naming together. The obtained API embedding is then used as features in a tree-based classifier to expand taint specifications starting from a small list of well-known APIs (seeds). Finally, adequate tooling around the generated taint specifications enables their automatic and uniform deployment in an ensemble of security analysis tools. With the proposed technology, we expand the configurable taint rules used in AWS CodeGuru Reviewer, improving its detection capabilities both in covering novel security scenarios, as well as maintaining a high acceptance rate of its reported findings.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ICSE-SEIP '24, April 14–20, 2024, Lisbon, Portugal
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0501-4/24/04
https://doi.org/10.1145/3639477.3639738

CCS CONCEPTS

• **Software and its engineering** → *Software maintenance tools*; • **Computing methodologies** → *Non-negative matrix factorization*; *Active learning settings*; • **Security and privacy** → *Software security engineering*.

KEYWORDS

Knowledge graph, Mining, Human-in-the-loop, Multi-view usage pattern, Functionality inference, Vulnerability detection

ACM Reference Format:
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 *46th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3639477.3639738

1 INTRODUCTION

A wide class of security vulnerabilities is related to regulating information flows inside a program, and can be formalized in a taint model. In taint analysis, a useful specification must contain sources, sinks and relevant sanitizers. A source specifies the starting point of the taint tracking, while a sink marks the destinations to which tainted data should not flow from the sources, unless the tainted data flows through a sanitizer. For example, to protect confidentiality, we want to prevent any leakage of sensitive data (source) to public holders (sink); for integrity, trusted information (sink) should not be corrupted by untrusted data (source) unless it is verified (sanitizer).

Many analysis tools make efforts to verify information flows of a program, and all these tools rely on taint policies, specification of sources, sinks and sanitizers in the program. The task of providing taint specification is constantly assumed as the primary duty of the tool users, or secondary duty of the tool developers. However, from our observations, even with additional help from domain experts, neither of them have the right resources or capacity to accomplish this task.

Foremost, it requires full-fledged knowledge to identify all the relevant APIs in the under analysis artifact. At least three different

kinds of knowledge are involved. (1) Knowledge on implementations of the artifacts, e.g., what APIs are used inside the under analysis artifact. This knowledge is owned by the developer of the under analysis artifact, which often time is also the tool user. (2) Knowledge on modeling the taint analysis problem, e.g., to use taint analysis to prevent injection attack, one should mark all the APIs that could take user inputs as *sources*. Domain experts, e.g., for injection attacks, possess this knowledge. (3) The syntax and semantics of the specification used by the taint analysis. The developers of the taint analysis owns this knowledge. To analyze real world application, the cost for any party to fully comprehend all the knowledge above is prohibitively high, or maybe disallowed in the first place for security concerns.

What makes this worse is that this is not a one-time effort, but rather the taint specifications requires continuous maintenance over the course of the development cycles of both the under analysis artifact and the taint analysis tool. Changes can be requested to adapt to the new design of the taint analysis. Meanwhile, the under analysis artifact is also evolving, thus, taint specification must be patched to keep up with the updates in the artifact.

It becomes clear that some automated detection of sources, sinks and sanitizers can help alleviate the burden. The key problem of the taint specification inference is to understand the role of APIs, especially on their output data object and input parameters. It is natural to adopt the machine learning tool and model it as a classification task. However, there are two major issues: what are good features in characterizing APIs and how many positive (tainted) API samples could we gather? Compared to the vast number of APIs available today, the taint specifications are very rare, and therefore, modeling it as a classification task faces serious data imbalance issue. It is better to start from the known taint related APIs (i .e., seeds) and identify the similar ones in terms of both the use-patterns and the naming. This can generate a list of high probable taint APIs. However, the known API distributions are also imbalanced for different scenarios, including human-in-loop for labeling findings. And also, iterative expansion of seeds is more likely to yield an effective way in solving the taint specification inference challenge.

In this paper, we aim to improve the recall and the precision of code analysis tools that rely on taint policies to identify security vulnerabilities by automating the process of expanding the list of sources, sinks, and sanitizers, as well as associating them with specific types of security vulnerabilities. The input to our approach is an initial set of taint policies mined from open source tools. Each policy consists of a set of sources, sinks, and sanitizers in the form of APIs. Our approach first generalizes the categorization of sources, sinks, and sanitizers by building a knowledge graph, where each node in the graph represents the categorization of APIs in terms of their functionality. For example, a source API, `HttpServletRequest.getRequestURI()`, is assigned to the node NETWORK_REQUEST_READ in the knowledge graph. After building the knowledge graph for the initial set of sources, sinks, and sanitizers, our approach systematically expands this set using data-flow analysis, mining and labeling, and similarity reasoning. The data-flow analysis is responsible for identifying the potential candidate APIs. For example, it performs forward traversal from the initial set of sources on the data-flow graph, to identify relevant program points that our approach can extract candidate sanitizers or sinks from. Labeling is then used to precisely identify the real candidates, and assign them to nodes in the knowledge graph. To further expand the set of APIs inside each node of the knowledge graph, our approach leverages similarity reasoning and finds more APIs based on the naming convention and usage patterns of the APIs. We demonstrated that our approach is able to precisely identify and categorize a diverse set of taint specifications, which is then deployed onto an industrial security detection tool to improve the frequency and coverage of reported security findings. We also show the efficacy of our approach from our experiments on two large source code datasets, as well as the improvement in coverage and acceptance of security findings in Amazon CodeGuru Reviewer service [1].

Results summary. Our mining and inference technique is effective–starting from a small set of seeds ranging from 3 to 8 expands upto 62 to 162 candidate APIs in 13 out of 18 categories, and it is precise–with > 90% inference precision in each category. Overall, we expanded the number of distinct API specifications in one of our internal taint analysis tools by 4.1×. More importantly it improved the diversity in our specifications by adding taint relevant APIs for several scenarios for which we initially had no or few API specifications. Using these inferred specifications for security detections yielded ∼ 2× more findings with 73% true positive rate on a large code dataset, and upto 100× more findings for some taint rules in production with no loss in developer acceptance. This high precision inference allows us to expand and maintain taint specifications in an automated and scalable way to keep up with the demands of ever-changing coding standards and security threat models.

2 OVERVIEW

In this paper, we propose a semi-automated inference algorithm to facilitate the onboarding and maintenance of taint specifications. To see why automation is needed in this process, we first discuss some challenges in maintaining taint specifications:

Challenge 1: Available specifications are limited to reuse. The limitations are two fold: quantity and compatibility. Taint specifications are rare resources. Only limited amount of specifications are made available. Notably Plume [9] and CodeQL [10] provide a generous set of candidates. However, their taint specification are neither compatible in syntax nor semantics, so it is impossible to directly reuse them.

Challenge 2: Numerous existing APIs and evermore new APIs. To come up with one's own taint specification, people have to identify taint relevant APIs from numerous existing APIs. It is usually not difficult to find a few relevant APIs, however, the question is how to determine we've included enough APIs to cover (high recall) the needs of the analysis. Meanwhile, new APIs are released all the time. For example, in the use case to prevent SQL injection attack [6], the under-analysis artifact initially covers only APIs for executing SQL commands from AWS Redshift and RDS, the two services that were launched in 2009 and 2012. As time went by, AWS launched more services with APIs that can run SQL queries, such as AWS Athena, launched in 2016, and AWS Timestream, launched in 2020. The under-analysis artifact must be kept updated with recent APIs provided by these services to protect against SQL injection.

Putting them together, to really obtain a complete set of relevant APIs, it requires endless effort to enumerate and keep monitoring state-of-art relevant APIs.

Challenge 3: Changing threat models. Based on the needs, users of the taint analysis can change their definitions on sources, sinks or sanitizers. For example, a new Log4J vulnerability, commonly known as Log4Shell (CVE-2021-44228) [16] with 10.0 out of 10.0 severity score, was uncovered in late 2021. The CVE revealed that the commonly used logging APIs are vulnerable to untrusted inputs. Before this vulnerability, the users of taint analysis have been using the logging APIs for just detecting logging of sensitive data, where the source represents sensitive data. Now users would want to come up with a new specification quickly where the source represents untrusted data, and the sink represents logging API from the Log4j library to detect this vulnerability. If a taint analysis has hardcoded the specification in a way that does not allow modification or has hardcoded it in a way that makes it difficult to reuse, the analysis would not be able to keep up with the ever-changing requirements.

To adopt changes in taint specification, on the surface, it only requires low operational cost to read, query or edit the specification. But in practice, this also implicitly requires a low cost to justify the meaning of specification and to verify the changes are intended to serve the new interest and it will not result in duplicated, incomplete or conflicting specifications.

Our semi-automated inference algorithm takes into consideration the ever-changing nature of the taint specification. It aims to generate taint specification with high precision and recall, and minimizes operational cost to adopt changes. Our approach includes four major components:

Functionality-based Specification Format. To endure changes, we formalize taint policy by an unchanged property– its functionality. We use *knowledge graph* to describe the functionality of APIs. Based on the knowledge label, rather than concrete APIs, we define *taint rules* to summarize taint policy. When new relevant APIs are added (Challenge 2), ideally, they will be associated with the same knowledge labels, thus, they are immediately used by the taint rules based on those labels. When a new security threat presents (Challenge 3), new taint rules can be created to replace the old rules. On one hand, the new rule can reuse the knowledge label to quickly find relevant APIs; on the other, this confines the change into these two rules, and guarantees other rules are left untouched.

With the functionality-based format, we know a *relevant* knowledge graph, that is all APIs are relevant to the functionality described by its label, leading to high precision inference outcome; meanwhile, a *complete* knowledge graph, that is all relevant APIs of a functionality are included in the knowledge graph, leads to high recall inference result. Therefore, the taint inference problem boils down to how to compute a relevant and (nearly) complete knowledge graph.

Seed Mining. To start with, we employ mining to obtain high quality sample APIs (we refer them as seeds) for each knowledge label. This is a semi-automated step, where we first use mining techniques in combination with data-flow and control-flow analysis to scan a large amount of code bases to find potential seed candidates, and then ask domain experts to review and label the candidate with a knowledge label. This design allows us to quickly find good amount of potential candidates, and at the same time assures the precision of the labeling results.

Similarity Labeling. With high-quality sample APIs (seeds) of each knowledge label, we follow a Machine Learning approach to train a classifier to label the functionality of a given API. Under the hood, this is based on its similarity to the seeds, in terms of naming conventions and their usage patterns in code.

By now, we have taint rules, describing the functionality used as source, sink and sanitizer, and a classifier that can map APIs to their functionality. Putting them together, we are able to generate the taint policy specification for sets of APIs that are sources, sinks or sanitizers.

Active Refinement. To keep the knowledge labeling up to date and accurate (Challenge 2), we also adjust the seeds by learning from the similarity labeling result via manual review.

3 FUNCTIONALITY-BASED FORMAT

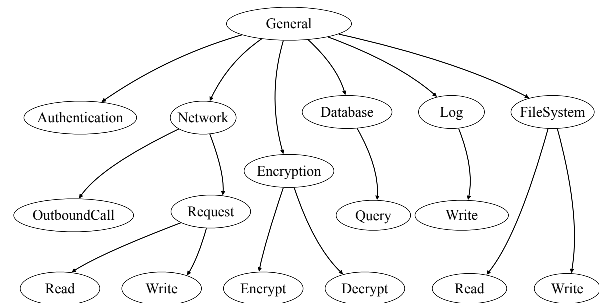


Figure 1: Knowledge Graph

3.1 Knowledge Graph

The structure of the knowledge graph represents our holistic understanding of program functionalities. It starts with a tree structure (extendable to a DAG), and each node stores well-known APIs under that category. A leaf node in the graph represents the most accurate description for a functionality. An example knowledge graph is shown in Figure 1. Each API can be categorized into one or more nodes in the graph. Here are some example knowledge graph

labels, each with an example API:

NETWORK_REQUEST_READ :
– javax.servlet.ServletRequest.getInputStream
CREDENTIALS :
– java.io.Console.readPassword
LOG_WRITE :
– org.apache.commons.logging.Logger.info
COMMAND :
– java.lang.ProcessBuilder.command
COMMAND_ESCAPE :
– org.apache.commons.text.StringEscapeUtils.escapeXSI
CRYPTO_ENCRYPT :
– javax.crypto.Cipher.doFinal

3.2 Taint Rules

With the knowledge graph describing the functionality of the APIs, we formalize taint tracking problems as a set of taint rules, where each rule consists of three sets of knowledge graph labels for sources, sinks and sanitizers. A taint rule is a standalone specification for a taint tracking use case. Multiple taint rules are used to separate the interests of different concerns. From the example above, we distill two rules:

sensitive-info-leak = { *source* : {CREDENTIALS},
sink : {LOG_WRITE},
sanitizer : {CRYPTO_ENCRYPT} }
command-injection = { *source* : {NETWORK_REQUEST_READ},
sink : {COMMAND},
sanitizer : {COMMAND_ESCAPE} }

For justification, with the knowledge graph to explain the functionality, now we can easily justify for each rule. For *sensitive-info-leak*, its goal is prevent credentials (in plain text) being leaked to logging information unless it is encrypted. For *command-injection*, untrusted user input coming from network requests should never be used directly to construct commands unless it is safely escaped.

For configurations, a user can select or add taint rules that are interesting to them. The knowledge graph helps provide complete and correct set of APIs for taint rules.

4 SEED MINING

The initial set seeds from open source tools are limited and imbalanced, i.e., some knowledge labels only contains a list of two to three APIs. If they were used directly for our later-on step, similarity labeling, the resulting set would be also limited. To address this, we employ mining to obtain high quality sample APIs (we refer them as seeds) for each knowledge label. This is a semi-automated step, where we first use mining techniques in combination with data-flow and control-flow analysis to scan a large amount of code bases to find potential seed candidates, and then ask domain experts to review and label the candidate with a knowledge label.

4.1 Mining

As discussed in subsection 3.1, each node in the graph represents certain functionality, and our goal is to assign every taint-related API to a node or a set of nodes (certain API can exhibit different functionality under different context). In order to get there, we need to determine whether an API is taint related or not. The challenge lies in the fact that taint APIs are rare and the majority of the APIs are not taint-related, which makes any classification-based approach under-performing due to this high imbalance in the data. On the other hand, taint APIs typically occur together, e.g., as (source, sanitizer, sink) triplet, and this property allows us to design mining style approaches that starts with known taint specification (seed) and discovers interesting APIs in its neighborhood. Starting from the seeds and following data-flow edges either forward or backward, there are high chances we will reach other taint related APIs. If the starting API is a source, then forward traversal is performed, and if the starting seed is a sink, then backward traversal is executed. Starting from a sanitizer, we perform both forward and backward traversal. Our approach adopted MU graphs [13] which represents programs at statement and expression levels that capture both control and data flows between program elements. (The MU graphs will be used again later and we will explain the details in 5.1). Along the path, if the APIs belong to certain non-interesting types, such as `string.append`, `alpha.upper` etc, the traversal passes them through and moves on until it reaches some interesting APIs. This procedure is different from the constrained optimization setting adopted in Seldon [4], although they are based on the same principle, if there are some known seed occurring in the data-flow edges, the chances of other taint specifications occurring coherently are high. However, one of the biggest issues with Seldon is that the impact of the size of seeds is significant, with very few seeds the results obtained by Seldon exhibit high precision but low recall, and most of the APIs will be resolved as non-taint related. This happens because without enough number of seeds, most of the constraints are relaxed and majority of the nodes are pushed to very small likelihood of being interesting. With more seeds, the information propagation becomes more effective, and the algorithm discovers taint API. The mining approach is simpler and does not suffer from the sparseness of the seeds, mainly because mining collects all possible trajectories, and we then refine the process leveraging human-in-the-loop labeling.

4.2 Labeling

During mining, the code snippet associated with the traversed path is collected and later presented to the experts for labeling. The purpose of labeling is to leverage the experts' domain knowledge to confirm whether the discovered API of interest is taint related, and further what type of taint specification it belongs to. Figure 2 presents the labeling interface we developed to collect expert feedback. It displays the mined code snippet, providing link to the original source code in case more context is needed to reason. The type of seed used for traversal provides additional information, such as the example in Figure 2 is "source-centric", meaning that the source is a seed and feed-forward traversal is performed to discover potential sanitizers and sinks. We find such information usually helps the labeler to perform labeling more confidently. Besides the

broad categories of taint specification, i.e. source, sanitizer or sink, we also attempt to provide sub-category level information which corresponds to the nodes in the knowledge graph. For example the API `DriverManager.getConnection()` is a candidate for sink but it is "SQL connection" type of sink. This allows us to accumulate the knowledge to build the knowledge graph. The labelers are able to change the provided sub-category when they think the inferred one is not correct. On top of the single API level information collected during the labeling, pairs or triplets that form more complete use case of taint flow are also gathered. This allows us to discover what types of sources, sanitizers and sinks usually occur together, and these pairs and triplets together with the sub-category labels form the candidate taint rules. Note that during the mining process, we start from one type of taint specification and may discover taint specifications belonging to a different category. In the next section, we focus our approach on discovering the taint specifications within the same category.

5 SIMILARITY LABELING

In this section, we propose a novel approach to perform taint specification inference for the same category, i.e., discover more seed APIs under the same node in the knowledge graph. The key concept here is to study the similarity between APIs, and thus we formalize our problem as a collective matrix decomposition setting. In order to quantify the similarities among APIs, we create a vector representation, i.e., embedding, and the distance between two embedding vectors characterize the similarity of two APIs. Such representation is learned from two aspects: (1) use-pattern of the API, and (2) appearance of the API. The problem is then boiled down to how to integrate these two types of information and learn a meaningful vector representation for each API call. One thing worth discussing is that there are many type of features that can be extracted to represent an API. For example, the input of the API can be used to determine the category which the API belongs to. In this work, we focus on the naming of APIs, which is a straightforward signal, and the API interactions, which is a high level and general representation of the API behaviour. To give an example, the input of the API would be captured by the API relationship ('action', 'def', 'data', 'para', 'action'), where the data is processed and pass to the API as an input.

We tackle this representation learning problem through the approach of Collective Matrix Factorization (CMF) [7]. CMF is a model that learns low dimension representations simultaneously through a collection of matrices with shared entities. In our application, it is the joint learning between the matrices that represents API interactions and the matrices that represent API appearance. The idea of CMF is to share the embedding across the matrices (i.e., API-API interactions and API appearance matrices). In this way, we can transfer the information between the appearance and use-pattern.

5.1 Code Representation and Usage Pattern

Other than appearances of the APIs themselves, the usage patterns of APIs with respect to their control-flow and data-flow of the neighboring code reveal the functionality of the APIs. To perform similarity reasoning, we choose MU graphs [13] which represents programs at statement and expression levels that capture both control and data flows between program elements. The control flow represents the order of the execution and the data flow represents the flow of data along the computation. There are 5 different type of nodes in MU graphs: action, data, control, entry and exit. API calls are represented as action nodes while branching and looping structures are represented by control nodes. The interactions among API calls, data elements and control structures are represented by edges between nodes such as definitions (def), parameters (para) and control dependence (dep). Each sub-graph structure involving two or more action nodes describes a particular coding pattern. For example, a sequence of nodes of ('action', 'def', 'data', 'para', 'action') describes an API generates an output and feeds into another API; a sequence of nodes of ('action', 'dep', 'action', 'dep', 'action') describes a sequence of three API calls; and a sequence of nodes of ('action', 'dep', 'control', 'dep', 'action') describes an if and then conditional flow. Beyond these simple chains of nodes, more interesting patterns can be described by more complicated local structures, such as two parallel chains share the same start and end nodes, (('control', 'dep', 'action', 'dep', 'action'), ('control', 'dep', 'action', 'dep', 'action')). This particular structure describes a relationship that can be illustrated by the code snippet in the following code snippet.

```
if (something)
  a.setFoo();
else
  a.setBar();
use(a); // or a.getSomethingElse()
```

Depending on the `something` condition, user might need to set different property (or do different things with the object) before consuming or doing something else. As we can see, each of these sub-structures describes a unique API usage pattern, and simultaneously analyzing them gives multi-view learning opportunities that we believe have not been explored in prior research.

In this paper, we focus on single chain patterns. We first generate MU graphs for java source code in 27k GitHub Repositories. Totally 20 such local patterns are considered and Table 1 lists all these patterns. We infer these local patterns through covering all 1-hop API to API, and the most frequent 2-hop API to API relationships. We believe that the local structure of 1-hop API to API is the most representative given that the APIs are close to each in the graph structure. As for the 2-hop API to API relationships, the number of local structural type grows exponentially. We therefore retain top-12 the most frequent and commonly observed local structures to maintain a short list of 20 local structures. These patterns describe most of the important interactions between two APIs and we consider them together to give a complete view of API usages.

5.2 Co-occurrence Matrices for API Use Embedding

For each type of local structure in Table 1, a co-occurrence matrix is built such that if there is an API pair appearing in this structure, the corresponding cell in the matrix increases by one. The row of the matrix is associated with the first API while the column is with the second API in a pair. The APIs in rows are the API of interest, and a row in the matrix represents how often it interacts with other

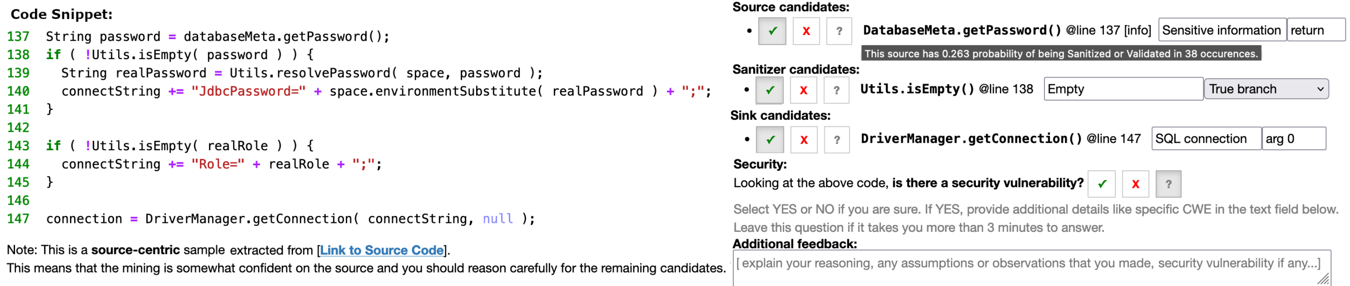


Figure 2: Labeling interface

Table 1: Local Code Patterns

Pattern	Description
1:('action', 'def', 'data', 'para', 'action')	API generates an output which is then consumed by another API
2:('action', 'dep', 'action', 'dep', 'action')	Two API-calls two hops away
3:('action', 'def', 'data', 'receiver', 'action')	API instantiates an object on which another API is then called
4:('action', 'dep', 'action', 'def', 'data', 'para', 'action')	Additional API call before pattern 1
5:('action', 'def', 'data', 'para', 'action', 'dep', 'action')	Additional API call after pattern 1
6:('action', 'dep', 'action', 'dep', 'action', 'dep', 'action')	Two API-calls three hops away
7:('action', 'dep', 'action', 'def', 'data', 'receiver', 'action')	Additional hop before pattern 3
8:('action', 'def', 'data', 'receiver', 'action', 'dep', 'action')	Additional hop after pattern 3
9:('action', 'dep', 'control', 'dep', 'action')	The first API call pre-dominates the execution of the second API call
10:('action', 'dep', 'action', 'dep', 'control', 'dep', 'action')	Additional hop before pattern 9
11:('action', 'def', 'data', 'condition', 'control', 'dep', 'action')	Value returned by the first API call determines the execution of the second
12:('action', 'dep', 'control', 'dep', 'action', 'dep', 'action')	Addition hop after pattern 9
13:('action', 'dep', 'while', 'dep', 'action')	The first API call pre-dominates the repeated executions of the second API
14:('action', 'def', 'data', 'condition', 'while', 'dep', 'action')	Value returned by the first API determines the repeated calls of the second
15:('action', 'dep', 'while', 'dep', 'action', 'dep', 'action')	Additional hop after pattern 13
16:('action', 'dep','action', 'dep', 'while', 'dep', 'action')	Additional hop before pattern 13
17:('action', 'throw', 'catch', 'dep', 'action')	Handling exception thrown by an API call
18:('action', 'may_throw', 'catch', 'dep', 'action')	Handling exception (may be) thrown by an API call
19:('action', 'dep', 'action', 'may_throw', 'catch', 'dep', 'action')	Additional hop before pattern 17
20:('action', 'may_throw', 'catch', 'dep', 'action', 'dep', 'action')	Additional hop after pattern 18

APIs in this type of local structure. The APIs in the columns play supporting roles. This forms a standard matrix factorization setup as traditional document-term analysis. The embedding of the row APIs are the representation of interest, while the embeddings of the column APIs are the basis of the column API space.

In total, we study 20 different type of local structures and thus 20 such co-occurrence matrices. Each of such matrix describes the frequencies of API pairs appearing in that structure, and performing a collective matrix factorization generates a complete use-pattern embedding of APIs. In this study we treat the same API appearing in row and column differently, referring them as row embedding and column embedding respectively. Combining the 20 row embedding of an API together forms the use-pattern signature of an API. Our hypothesis is that two APIs are similar when they interact with the same or similar APIs, i.e., there is high resemblance in terms of their usage patterns, and we are looking at multiple such patterns simultaneously.

5.3 API Appearance Embedding

Besides use-pattern similarity we also consider naming (token level) similarity of two APIs which provide important clues to the intended usage of an API, and we refer to this as appearance similarity. For example, 'getQueryParameter' and 'appendQuery-Parameter' clearly are closely related and likely to be member functions of the same class. The hypothesis is that the appearances of the APIs can also suggest the resemblance of two APIs. Following a good naming convention, developers tend to name APIs and methods with similar purposes in a similar way. For building the co-occurrence matrix for appearance, the rows are individual APIs and columns are their name tokens. The tokens are obtained by decomposing the API name using the traditional camel case splitting scheme, e.g., 'getQueryParameter' is composed of three tokens ('get','Query','Parameter').

The importance of appearance embedding extends beyond capturing the appearance of APIs and it provides "gluing" among

clusters of APIs. A typical document-term matrix is dense, each document contains many words, so that there are many non-zero elements in each row. Removing some words does not change the shared words between two documents that much. However, the API-API co-occurrence matrix is usually sparse, e.g., each API only interacts with a very small number of APIs, and this is especially true for customized class and APIs. If we properly change the order of the rows and columns of the matrix, it will eventually become a large matrix containing many small block matrices along the diagonal line. The direct factorization of a single such shaped matrix generates API embedding that only shows high similarities in a small cluster of closely related APIs but the distance between APIs in different cluster is meaningless. Two things come to help: appearance similarity makes long range connections between API blocks sharing similar appearance; combining multiple co-occurrence matrices such that APIs not connected in one matrix connected in other matrices – these two combined together generates good API embedding.

5.4 Formulation

Given a co-occurrence matrix $X_i \in \mathcal{R}^{m\times n}$, where i stands for i-th relationship between APIs, and two matrices of API appearance representations $U \in \mathcal{R}^{m\times p}$ and $I \in \mathcal{R}^{n\times p}$, collective matrix factorization aims to optimize the following objective

$$\Sigma_i^N ||X_i - AB_i^T|| + \alpha||U - AC^T|| + \beta\Sigma_i^N||I - B_iD^T|| \quad (1)$$

where m is the number of row APIs, n is the number of column APIs; and p is the number of tokens used to represent API names in U and I. Each row in $A \in \mathcal{R}^{m\times k}$ and $B_i \in \mathcal{R}^{n\times k}$ represents the embedding for an API of interest and an interacted API, where k is the dimension of the embedding and i represents the ith type of local structures. Each row in $C \in \mathcal{R}^{p\times k}$ and $D \in \mathcal{R}^{q\times k}$ is the embedding for the corresponding tokens. Note that we don't share C and D since the corresponding A and B_i are representations for APIs in different roles. The information from the API appearances and API interactions are neatly integrated through sharing the embedding across the matrices. If two APIs (i.e., rows API in X_i) which interact with a similar set of APIs (i.e., column API in X) would have similar entries in X_i, and thus they would also have similar embedding (i.e., A). Meanwhile, the API calls having similar appearances should share similar tokens in their names (i.e., U) and as a result, they would share similar embeddings. Very often, the APIs don't interact with exactly the same set of APIs. However, the interacted APIs can share high resemblance in the appearances (i.e., I). This may suggest that the APIs of interested are interacting with APIs with the same functionality. Thus, they should be given a similar embedding (i.e., B_i). Parameters α, β are used to balance the weights between the use-pattern and appearance, and we choose both of them as 1.0 in this paper.

5.5 Active Refinement

Our goal is two folded: to expand the seeds as much as possible to cover most of the vulnerability scenarios, and to build a robust predictive system to better infer the functionality for customized APIs. In this section, we discuss the general strategy for expanding using active learning. Starting with a small number of seed APIs, we design an iterative procedure to include new APIs for training accurate machine learning models. At each step, representative APIs are selected based on different features, such as the API's appearance and its use-pattern to ensure the inclusiveness for the important features with respect to its functionality. The learning process is summarized in Algorithm 1.

Algorithm 1 Active Learning Procedure

Step 1: Collect commonly used APIs for each scenario and denote them as $\mathcal{S}_i$
Step 2: Expand the training dataset with keyword search from git repositories and denote them as $\mathcal{S}_k$
Step 3: Tokenize API names and build tree-based classifiers to infer on APIs in the test set
Step 4: Perform expert review to increase the training set and denote them as $\mathcal{S}_n$
Step 5: Learn the embeddings for APIs through the collective matrix factorization
Step 6: Build tree-based classifiers on the embeddings learned through multi-view usage patterns and its appearance
Step 7: Review the APIs in the test set to expand the set of seeds and reinforce the classifiers, and denote them as $\mathcal{S}_{\text{CMF}}$

6 EVALUATION

In this section, we report data and experience on running our proposed taint specification inference in several industrial software engineering use cases to answer three key evaluation questions:

EQ1: *What is the precision of our inference algorithm?* Addressing this question is critical to understanding how our technique scales with minimal effort from human experts.
EQ2: *How do the steps of mining and inference improve the coverage and diversity of taint specifications?*
EQ3: *Do the specification obtained using mining and inference actually result in improved quality of security detectors in an industrial setting?*

6.1 Precision of Specification Expansion

Starting from a limit number of seeds that fall into the categories with the functionality tag initially (denoted as $\mathcal{S}_i$), in this section, we will demonstrate the advantage of the proposed approach in expanding the seeds and hence the taint specifications. Following the steps in subsection 5.5, at each step, we make predictions on the test set and select top-50 APIs with the highest prediction scores for review. To verify the accuracy of the expanded seeds, we perform multi-round shadow reviews with a group of security experts to check whether an API belongs to a certain scenario. The reviewing process includes: 1) the APIs sorted by the prediction scores; and 2) the code snippets where the API is used and it's implementation if it's source code is available, e.g., the API is defined in the same repository. During the shadow reviews, each security expert is given top-50 APIs with the highest prediction scores. They are asked to finish such task in one hour during the shadow review. The seed expansion in this step requires human-in-the-loop procedures. However, this is one time effort for us to establish seed pool. After

Table 2: Number of seeds expanded at each step in the active learning procedure and their precision in 5-fold cross validation

Scenario	Number of seeds expanded						Precision (%)		
	$\mathcal{S}_i$	$\mathcal{S}_k$	$\mathcal{S}_n$	$\mathcal{S}_{\text{u/n}}$	$\mathcal{S}_{\text{CMF}}$	total	$\mathcal{S}_{i,k}$	$\mathcal{S}_{i,k,n}$	$\mathcal{S}^{\text{CMF}}_{i,k,n}$
CONSOLE_READ	3	10	45	4	10	72	86.95	87.95	97.40
CONSOLE_WRITE	3	11	39	6	30	87	90.52	90.60	94.24
CREDENTIALS	8	16	43	23	29	119	97.86	96.11	95.51
CRYPTO_ENCRYPT	4	11	49	8	24	96	96.38	92.96	86.63
CRYPTO_DECRYPT	3	16	48	5	14	86	99.72	97.33	93.17
FILESYSTEM_PATH_RESOLVE	3	9	48	15	39	114	96.67	98.36	91.16
FILESYSTEM_READ	14	1	50	4	44	113	86.11	99.71	88.63
FILESYSTEM_WRITE	7	6	48	8	32	101	75.17	90.08	91.34
LOG_WRITE	104	2	23	5	28	162	98.64	92.61	91.25
NETWORK_COOKIE_READ	5	6	38	0	13	62	100.00	94.83	99.12
NETWORK_REQUEST_READ	68	0	50	3	38	159	95.71	99.58	96.29
NETWORK_REQUEST_WRITE	2	9	50	1	42	104	97.50	99.83	98.78
NETWORK_RESPONSE_READ	3	8	48	0	27	86	82.17	92.73	93.83
NETWORK_RESPONSE_WRITE	11	0	47	1	6	65	84.57	94.25	95.50
NETWORK_SESSION_READ	2	10	24	1	15	53	96.12	92.07	65.00
OS_COMMAND	2	8	44	4	22	80	93.19	94.40	93.86
SERIALIZATION_DESERIALIZE_XML	1	11	16	11	40	79	89.31	99.72	86.78
SQL_COMMAND	83	0	32	0	4	116	98.78	94.40	89.02

the shadow review, we select the Random Forest algorithm [15] as the tree-based classifier. We run 5-folds cross validation and select the best estimator with respect to the best average precision. In table 2 we show the expansion results (i.e., Number of seeds expanded) and the precision performance (i.e., Precision %). For example, in the first scenario, we start from only 3 seeds tagged as "CONSOLE_READ". We then use keyword to match and find 10 more seeds. Next, we identify 45 additional seeds based on the API names. With usage pattern, we can find 4 more seeds which can't be mined through just the naming. Finally, we infer 10 more with the classifier trained on the embeddings learned via CMF. In total, we can expand the taint APIs from 3 to 72 in the category.

The precision performance is in general encouraging, and there are a few observations. First, expansion using keywords usually is not enough, e.g., for the "FILESYSTEM_WRITE" tag, the precision is only 75.17% with only $\mathcal{S}_{i,k}$. Second, appearance alone gives a big boost and this is consistent with the intuition that the naming convention typically conveys the intentions of APIs. Third, jointly learning both from naming and the usage pattern shows advantages for majority of the categories. For example, for the "FILESYSTEM_WRITE" tag, the precision can be enhanced to 91.34% with $\mathcal{S}^{\text{CMF}}_{i,k,n}$. Even though the precision is a criteria of the predictive power in terms of the functionality tags, our goal is to be able to mine as many seeds for taint specifications as possible. In the table 2, we are showing that without losing too much of the precision performance, we can expand the seeds by its interactions with other APIs.

We show that our best inference technique expands seed APIs in all of the scenarios achieving > 90% precision overall in filtering taint relevant APIs and correctly inferring their functionality labels. This high precision tool allows us to scale the automated specification inference to keep up with the demands of ever-changing code and libraries, and so in the ideal deployment scenario security engineers need to only review a small set of high-quality specifications.

6.2 Coverage of Specification Expansion

The expanded taint specifications are deployed in a staged manner, with checks and measures along each step. This allows us to better utilize the expertise of security engineers by reviewing high confidence specifications first. Further, the staged expansion enables measuring the individual contributions of mining and inference to improving the coverage and diversity of the taint specifications.

First, we bring all internal security tools on-par by sharing their taint specifications in a canonical format obtained from the knowledge graph. In this "unification" step, we share existing taint specifications among three internal security detectors. Tool_1 is one of the core back-end analysis engines of Amazon CodeGuru Reviewer [1]. which provides code recommendations on internal code commits as well as available externally as a cloud service; Tool_2 is an internal tool used in various security campaigns internally so its specifications are tailored for high precision and coverage on internal service code; and Tool_3 is a compositional taint analysis engine used for various security policy enforcements [2].

As evident from Figure 3b, there is a sizeable opportunity for each tool by sharing their unified specifications, and then in Figure 3d, a significant opportunity to increase coverage by mining and inference. In Table 3, we show how the combined taint specifications evolve starting from the initial rules of **Tool_1**, followed by column **Unification** where sharing specifications from those of Tool_2 and Tool_3 grows its specifications by ~ 1.9×. Next, **Mining** from open-source tools Plume [9] and CodeQL [10] presents a significant increase in coverage by ~ 3.6×, and finally our **Inference** technique further fills in coverage gaps and expands the total set of specifications by ~ 4.1×. The expanded specifications comprise of 1352 distinct API specifications: 406 sources, 854 sinks, and 92 sanitizers. They originate as follows: (i) 161 novel specs inferred as described in section 5, (ii) 569 mined from open-source tools , and (iii) 622 existing specifications used by the two other internal tools not in scope for expansion.

Table 3 further shows that the benefit in coverage varies largely during each stage among the sources, sinks and sanitizers. All increase ratios shown in the table use the first column as the denominator, i.e. with respect to the initial set of specifications. The number of distinct sources, sinks and sanitizers increase by 6.15×, 3.39× and 9.20× respectively. Furthermore, the specifications not only increase in number, but as we see in the bottom section of Table 3, they also evolve in diversity– the number of labels increase from 16 upto 20, and impressively the average number of API specifications belonging to distinct labels improves from 20.5 to 69 indicating that the mining and inference indeed accumulates specifications for security categories that were initially only sparsely available.

Table 3: Increased coverage and diversity of taint specifications with mining and inference

	Tool_1	+ Unification		+ Mining		+ Inference	
# Specs	**328**	**616**	**1.88x**	**1,191**	**3.63x**	**1,352**	**4.12x**
# Sources	66	208	3.15x	353	5.35x	406	6.15x
# Sinks	252	396	1.57x	787	3.12x	854	3.39x
# Sanitizers	10	12	1.20x	51	5.10x	92	9.20x
Diversity of Taint Specifications							
# Labels	16	17	1.06x	20	1.25x	20	1.25x
Min /label	1	1	1.00x	2	2.00x	2	2.00x
Avg /label	20.5	36.1	1.76x	60.9	2.97x	69	3.37x
Max /label	120	166	1.38x	254	2.12x	296	2.47x

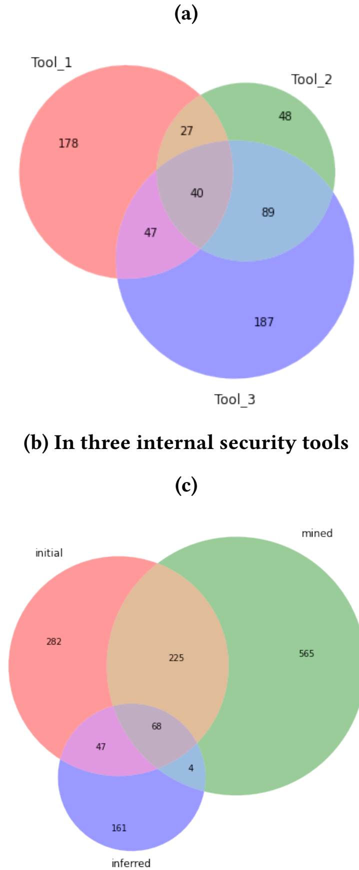
6.3 Impact on Quality of Security Detectors

Next, we report our experience on the impact of security detections reported by `Tool_1` by expanding its taint specifications based on the technologies we presented in this paper. Our evaluation shows that with the proposed taint specification expansion, we significantly improve the recall of the detector while retaining its precision.

Table 4: Improved recall and precision of security detectors using expanded specifications on code datasets

Dataset	#Rules	#Findings Baseline	#Findings Expanded	Findings Increase	Precision TP%
GitHub 145K repos	6/7 fired	7,658	14,856	1.94x	61%
Internal 200K+ repos	4/7 fired	258	479	1.86x	73%

First, we evaluate 7 taint rules on two large code corpus datasets as shown in Table 4. We run two configurations- `Baseline` uses the existing taint specifications, and `Expanded` uses the expanded taint specifications. As evident from Table 4, the taint specification expansion yielded significant increase in the detector's coverage by reporting 94% and 86% additional new findings on the two datasets respectively. We sampled 50 findings from the newly reported findings and manually reviewed them to evaluate the impact on the detector's precision. The tool has stipulated a $> 70\%$ acceptance requirement for its findings. The 73% true positive rate on internal code dataset already exceeds this bar. For the GitHub dataset, we observed the true positive rate declined but only slightly to 61%.



(a)

(b) In three internal security tools

(c)

(d) Among internal tools and expanded via mining and inference

Figure 3: Overlap and sharing opportunity among taint specifications.

Table 5: Improved recall of 7 taint rules with expanded specifications on internal code review system

Rule	Control	Expanded	Acceptance%
cross-site-scripting	3.65x	168.96x	80%
path-traversal	1.63x	71.25x	100%
os-command-injection	0	1.14x	n/a
sql-injection	+	+	100%
untrusted-load	+	+	n/a
ldap-injection	0	+	n/a
code-injection	0	+	n/a

*We cannot disclose exact details around number of recommendations for internal code.

Following this encouraging result, we validated the expanded specifications with manual reviews to exclude low-fidelity specifications. Thereafter, we deployed the expanded specifications and monitored the detector findings on production code review traffic in our internal code review system. Table 5 presents the increase in detection frequency for the 7 taint rules. The column **Expanded**

reports the fraction of findings detected with the expanded rules compared to those detected using the original specifications aggregated over one week period. Since the two detector versions were deployed in different regions serving different code review traffic, the column **Control** additionally presents this data for one week prior to deploying the expanded specifications as the difference in number of findings can vary with the code review traffic being analyzed. A + indicates that findings were reported by the rule when the baseline reported zero findings. Barring the variations in weekly code traffic, we observe significant increase in number of findings. Notably, some of the ultra low frequency rules, e.g. untrusted-load and ldap-injection start yielding many findings.

Finally, the last column in Table 5 reports the acceptance rates of findings yielded by the expanded rules based on developer feedback collected over one month after deployment. The n/a denotes there were not enough developer feedbacks to form a statistically meaningful report, so we exclude them from the acceptance data. To summarize our experiments, we demonstrate significant coverage improvements with the automatic specification expansion, while retaining high true positive rates.

7 RELATED WORK

Merlin [11] and Seldon [4] model the API relationships as constraints and try to solve a constrained optimization problem. Given a small set of seed specifications, Seldon frames the problem as a semi-supervised learning problem and jointly infers new sources, sinks and sanitizers by encoding data flow assumptions into linear constraints over taint indicator variables. It is based on the assumption that code is mostly free from security vulnerabilities and redundant operations in code are rare, and it tries to infer property of every API occurring in the flow. The quality of the inference results depends on the percentage of available seeds as well as the complexity of the flow. Seldon typically does not work well on scenarios in which sources and sinks are usually far apart and sometime belong to different methods, as it typically involves a large graph with very few seeds. Additionally, Seldon only considers the use-pattern but not the naming and the propagation graphs it constructs mixes different types of relationships together thus lacking the potential to isolate the most relevant types of flows with respect to taint analysis. InspectJS [8] extends Seldon and presents a mining framework including the similarity refiner and manual feedback driver. The similarity adopted is based on the code snippet embedding surrounding the sink instead of the pure API embedding, therefore, it suffers from the noises inherent in the context.

For general API usage study, API2Vec [14] presents a general framework to study API embedding, which derives the embedding following the style of Word2Vec [12]. Chains of API flows are extracted, and the middle API is masked out and predicted by the rest of APIs. As the paper shows, API elements that are frequently used together also cluster closely in the API2Vec space, which shows good performance in characterizing the API usage patterns. However, it only considers the use pattern similarity but not the naming similarity. Names in the source code are important as they often convey program semantics and developers' intention. VarCLR [3] has been shown on the IdBench benchmark to be the state of the art for the task of mapping identifier names to embedding in a way that preserves semantic relations and similarity. Fluffy [5] utilizes the API names as part of the natural language information to predict whether a tainted flow is expected or not. However, there is not much work focusing on integrating both the use-pattern and naming information together to characterize APIs. Besides the works on API embeddings, there are also works like SuSi [17] which directly uses the traditional ML classification techniques to predict the API functionality, it trains a support vector machine (SVM) classifier based on a number of extracted features to identify privacy specifications of Android APIs. While effective for Android, this method is less applicable for general application domains.

8 CONCLUSION

In this paper, we describe a collection of efforts to address the problem of policy specification and inference for taint analysis in an ever-changing landscape of software development and security threats. We adopt the knowledge graph representation to capture the learned functional categorizations of taint-relevant APIs, thereby decoupling from their security contexts which are captured as rules mapping to the functional labels in this graph. Mining and labeling provide an effective way to build the knowledge graph and taint rules, by discovering new candidate taint specifications which we then refine leveraging the expertise of security engineers. To actively maintain the taint specification in a scalable and automated way, we employ a novel multi-view active learning approach to address the taint specification inference problem. To better characterize an API, multi-view aspect of its use-pattern and its appearance are integrated together, and the generated API embedding is used to perform active learning style similarity reasoning for expanding taint specifications from a small set of well-known seeds. The problem is formulated into a collective matrix factorization setting to reuse existing efficient algorithms to achieve high precision needed for a production level use. The experimental results demonstrate the effectiveness of the proposed algorithm in accumulating numerous and diverse taint specifications. Our experience in deploying the mined and inferred specifications into the security detectors of Amazon CodeGuru Reviewer service showed significant improvements in the coverage of the taint rules while maintaining high acceptance rate of reported findings.

REFERENCES

[1] AWS. 2023. *Amazon CodeGuru Reviewer.* Available at https://docs.aws.amazon.com/codeguru/latest/reviewer-ug/welcome.html.

[2] Subarno Banerjee, Siwei Cui, Michael Emmi, Antonio Filieri, Liana Hadarean, Peixuan Li, Linghui Luo, Goran Piskachev, Nicolás Rosner, Aritra Sengupta, et al. 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*. ACM Digital Library, Association for Computing Machinery, 1601 Broadway, 10th Floor, New York, NY 10019-7434, USA, 1985–1996.

[3] Qibin Chen, Jeremy Lacomis, Edward J Schwartz, Graham Neubig, Bogdan Vasilescu, and Claire Le Goues. 2022. Varclr: Variable semantic representation pre-training via contrastive learning. In *Proceedings of the 44th International Conference on Software Engineering*. 2327–2339.

[4] 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*. 760–774.

[5] Yiu Wai Chow, Max Schäfer, and Michael Pradel. 2023. Beware of the unexpected: Bimodal taint analysis. *arXiv preprint arXiv:2301.10545* (2023).

[6] CWE Community. 2023. *CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection').* Available at https://cwe.mitre.org/data/definitions/89.html.
[7] David Cortes. 2018. Cold-start recommendations in collective matrix factorization. *arXiv preprint arXiv:1809.00366* (2018).
[8] Saikat Dutta, Diego Garbervetsky, Shuvendu K Lahiri, and Max Schäfer. 2022. InspectJS: leveraging code similarity and user-feedback for effective taint specification inference for JavaScript. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice.* 165–174.
[9] David B Effendi, Fabian Yamaguchi, and Jung Jaden. 2022. Plume. https://github.com/plume-oss/plume.
[10] Erik K Kristensen and Geoffrey White. 2023. Codeql. https://github.com/github/codeql.
[11] Benjamin Livshits, Aditya V Nori, Sriram K Rajamani, and Anindya Banerjee. 2009. Merlin: Specification inference for explicit information flow problems. *ACM Sigplan Notices* 44, 6 (2009), 75–86.
[12] T Mikolov, I Sutskever, K Chen, GS Corrado, and J Dean. 2013. Distributed representations of words and phrases and their compositionality. *Neural information processing systems* (2013).
[13] Rajdeep Mukherjee, Omer Tripp, Ben Liblit, and Michael Wilson. 2022. Static Analysis for AWS Best Practices in Python Code. In *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany (LIPIcs, Vol. 222).* 14:1–14:28. https://doi.org/10.4230/LIPIcs.ECOOP.2022.14
[14] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N Nguyen. 2017. Exploring API embedding for API usages and applications. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE).* IEEE, 438–449.
[15] Aakash Parmar, Rakesh Katariya, and Vatsal Patel. 2019. A review on random forest: An ensemble classifier. In *International conference on intelligent data communication technologies and internet of things (ICICI) 2018.* Springer, 758–763.
[16] CVE Program. 2021. *CVE-2021-44228.* https://cve.mitre.org/cgi-bin/cvename.cgi?name=2021-44228
[17] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. 2014. A machine-learning approach for classifying and categorizing android sources and sinks.. In *NDSS*, Vol. 14. 1125.

Received 6 October 2023; accepted 20 December 2024