ABSTRACT
Static analysis has established itself as a weapon of choice for detecting security vulnerabilities. Taint analysis in particular is a very general and powerful technique, where security policies are expressed in terms of forbidden flows, either from untrusted input sources to sensitive sinks (in integrity policies) or from sensitive sources to untrusted sinks (in confidentiality policies). The appeal of this approach is that the taint-tracking mechanism has to be implemented only once, and can then be parameterized with different taint specifications (that is, sets of sources and sinks, as well as any sanitizers that render otherwise problematic flows innocuous) to detect many different kinds of vulnerabilities.

But while techniques for implementing scalable inter-procedural static taint tracking are fairly well established, crafting taint specifications is still more of an art than a science, and in practice tends to involve a lot of manual effort.

Past work has focussed on automated techniques for inferring taint specifications for libraries either from their implementation or from the way they tend to be used in client code. Among the latter, machine learning-based approaches have shown great promise.

In this work we present our experience combining an existing machine-learning approach to mining sink specifications for JavaScript libraries with manual taint modelling in the context of GitHub’s CodeQL analysis framework. We show that the machine-learning component can successfully infer many new taint sinks that are not part of the manual modelling or are not detected due to analysis incompleteness. Moreover, we present techniques for organizing sink predictions using automated ranking and code-similarity metrics that allow an analysis engineer to efficiently sift through large numbers of predictions to identify true positives.

KEYWORDS
Taint Analysis, Machine Learning, JavaScript

CCS CONCEPTS
• Software and its engineering → Automated static analysis.

Permission to make digital or hard copies of all or part 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 components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICSE-SEIP ’22, May 21–29, 2022, Pittsburgh, PA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9226-6/22/05...
https://doi.org/10.1145/3510457.3513048
uses probabilistic inference to identify candidate taint specifications from the way clients use the library. The inference attaches to each candidate a score between zero and one which intuitively indicates how confident we are that the prediction is correct. As a final step, the concrete candidates identified on the training set need to be abstracted into a code-base independent representation that can be used to find candidate taint specifications on other code bases.

In this work, we add to this process a refinement step where the score of a candidate is adjusted using code-similarity metrics, giving greater weight to candidates that appear in a context that is syntactically similar to known taint specifications for which we already possess a manually-written model.

Unlike Seldon, we do not aim to obviate manual models but to improve them. To make this feasible, we need a way of presenting predicted taint specifications to an analysis engineer that makes them easy to triage and efficiently prune away false positives.

We propose three criteria for organizing predictions: by score as determined by the probabilistic inference and refined using code similarity; by generality; and by syntactic similarity. The first one is quite obvious: predictions with a low score are not worth showing to the engineer. For the second one, the idea is that overly general representations that lead to a large number of predicted sinks are unlikely to be true positives. The third one again uses code similarity, this time to allow the engineer to dismiss a false positive along with all other predictions that are syntactically similar to it.

We have implemented our approach in a tool called InspectJS, which is based on the CodeQL analysis framework [7], and can be used to infer sink specifications for JavaScript. In CodeQL, sink specifications tend to be analysis-specific and so require greater generality, this time to allow the engineer to dismiss a false positive along with all other predictions that are syntactically similar to it.

We motivate our work using a concrete example in Section 2, discuss its relationship with Seldon in Sections 3 and 4, and empirically evaluate the quality of the sink predictions produced by InspectJS in Section 5 before surveying related work in Section 6 and concluding in Section 7.

In summary, the main contributions of our work are:

- A novel combination of a probabilistic approach to predicting taint-sink specifications from static data-flow information with code-similarity based refinement to adjust prediction score based on their similarity to known sinks.
- Three techniques for organizing sink predictions based on their score, generality, and similarity to each other, allowing a domain expert to efficiently triage large numbers of automatically generated predictions.
- An implementation of our technique on top of the CodeQL static analysis framework in a tool called InspectJS.
- An empirical evaluation of the quality of the predictions produced by InspectJS on real-world code bases, showing that it correctly identifies taint sinks that are missing from the manually-written models shipping with CodeQL, some of which have since been incorporated into the models.

2 MOTIVATING EXAMPLE

To motivate our work, we will show an example of a missing taint specification in the CodeQL static analysis for JavaScript, which was identified with the help of InspectJS and has since been added to the manually-written model.\(^1\)

Consider the code snippet in Figure 1(a), which is adapted (and slightly simplified) from the WaftEngine project.\(^2\) It shows a route handler from an HTTP server implemented as a JavaScript function slightly simplified) from the WaftEngine project.\(^2\) It shows a route handler from an HTTP server implemented as a JavaScript function accepting three parameters req, res, and nxt. Parameter req is the HTTP request object originating from a client, res is the response object to be filled in by the handler, and nxt is the next handler to be called in case of error.

The route handler extracts the slider_key field from the request body (Line 3), passes it to the slugify function (defined in Lines 13-15) which lower-cases it and replaces all spaces with hyphens (Line 4), and then uses the resulting string to look up and update an entry in a NoSQL database using the findByIdAndUpdate method (Lines 5-8). The first argument to this method is a JavaScript object, which is interpreted as a NoSQL query. For example, the query { id: "myslider" } selects all entries with the id field equal to "myslider". This query is really a short-hand for the query { id: { $eq: "myslider" } }, using the NoSQL operator $eq to compare the field id with the value "myslider".

\(^1\)https://github.com/github/codeql/pull/4753
\(^2\)https://github.com/WaftTech/WaftEngine

---

**Figure 1:** Two uses of APIs relevant to NoSQL injection vulnerabilities: (a) findByIdAndUpdate, and (b) findOneAndUpdate. Circled numbers indicate expressions referenced in the text.
Other operators allow more complicated tests, for example $\neq$ for inequality, $\$regex$ for regular expression matching, and $\$where$ for specifying an arbitrary JavaScript expression. It is because of these more advanced operators that it is not, in general, safe to pass data controlled by an untrusted user to a NoSQL API method expecting a query, since the user might specify a query using $\$regex$ and $\$where$ to access almost any entry, or a query using $\$where$ to execute arbitrary JavaScript code.3

In this example, while req. slider_key is under user control, it is used in a reasonably safe manner: as revealed by its use in the slugify function, slider_key is a string, so it cannot be used to encode potentially problematic conditions. CodeQL recognizes this and does not flag this snippet as problematic: while its models allow it to classify req. body and its properties as taint sources and the first argument to findByIdAndUpdate as a taint sink, it also knows that replace (Line 14) acts as a sanitizer in this case since its result is guaranteed to be a string.

Now consider the code snippet in Figure 1(b), showing a different handler function in the same project. Its structure is very similar: a property of req. body is read (Line 3), processed with replace (Line 4), and then used in a NoSQL query (Lines 5–8), this time with the findByIdAndUpdate method. Prior to our work CodeQL did not recognize this method as a sink, and hence would have failed to flag not just this safe use, but also unsafe uses.

This is not an uncommon problem: manually modelling large APIs, like the Mongoose framework4 being used here, is tedious and error-prone often leading to missing taint sources or sinks. Automated taint-specification mining promises to eliminate or at least alleviate this problem.

Many different approaches have been proposed in the literature for automatically discovering taint-specifications. Our work builds on the flow triple1 approach introduced by Merlin [10] as refined by Seldon [4]. At a high level, this involves three steps.

First, we mine a training set of code bases for triples of program elements (src, san, snk) where taint may propagate from src to snk via san, and src is of a syntactic structure that means it could potentially be a taint source (e.g., the result of a function call or a parameter to a callback), san could be a taint sink (i.e., a function call), and snk could be a taint sink (e.g., a function argument). Note that this is done based on purely syntactic criteria, independent of any semantic modelling.

For the snippets in Figure 1, for example, we would obtain the triples (①, ②, ③) and (④, ⑤, ⑥) representing the flow from the request objects through the sanitizing string replacements into the NoSQL queries.

Second, we perform a probabilistic analysis of these triples based on the following observation: if src is known to be a taint source and san a sanitizer, then it is very likely that among all the nodes snk for which we have observed a flow triple (src, san, snk), at least one is a taint sink, since otherwise there presumably will not be any need for sanitization. Similarly, from known sources and sinks we can infer the presence of a sanitizer, and from known sanitizers and sinks a source.

For example, for the triple (④, ⑤, ⑥) mined from Figure 1(b), we already know that ④ is a source and ⑤ is a sanitizer, suggesting that ⑥ may be a sink, as is indeed the case.

These newly inferred elements can then be plugged into the triples in turn, allowing us to discover even more sources, sinks, and sanitizers. As discussed in more detail below, Seldon associates a score between 0 and 1 with each such prediction which represents the degree of confidence in the correctness of the prediction.

In the third step, we can suitably abstract the concrete elements observed on the training set into a code-base independent representation, discard predictions with low scores, and then use them to improve (“boost”) a taint analysis, allowing it to flag more alerts on any code base, not just the ones in the training set.

Alternatively (and this is the use case we are most interested in) the results of the inference step can be presented to an analysis engineer for further triaging, allowing them to identify lacunae in the hand-written models and improve the analysis accordingly.

One weakness of such purely probabilistic approaches is that they have little built-in knowledge of the semantics of the code being analyzed apart from information about known taint specifications, which can sometimes lead to surprising mispredictions. For example, the call to console.log in Figure 1(b) is not a sink, but based on purely syntactic criteria it looks like a plausible candidate, and so we would add the (④, ⑤, ⑥) to our set of mined flow triples. If there are sufficiently many similar usages, we might then end up wrongly predicting that console.log is a sink.

To prevent this, our approach combines the probabilistic analysis with a post-processing step based on code similarity, whereby the scores of sink predictions are adjusted based on their similarity to known sinks. In our example, the call to console.log does not look similar to a sink, so its score would be decreased, while the call to findByIdAndUpdate is syntactically quite similar to the call to findByIdAndUpdate, which we know to be a sink, causing its score to be increased.

3 BACKGROUND: SELDON

Before we delve into the technical details of our approach, we give a brief overview of Seldon [4], on which the core inference engine in InspectJS is based.

Seldon is a semi-supervised approach for inferring likely taint specifications (source, sanitizer, or sink) for unmodeled or partially modeled library APIs from a large corpus of client code using these APIs. Based on a set of client programs $\mathcal{D}$ with a (small) set of program elements $\mathcal{M}$ already annotated as sources, sanitizers, or sinks, Seldon infers specifications for the remaining (larger) set of un-annotated program elements $\mathcal{A}_U$. This involves four steps described in more detail below: capturing information flow in the form of a propagation graph; representing the nodes of that graph in a code-base independent form; building a constraint system encoding the taint-specification inference problem; and finally solving that system.

While Seldon was originally implemented and evaluated for Python, we adapt the approach for JavaScript as we describe in Section 4.1.

Capturing information flow. For each input program, Seldon builds a propagation graph $G = (V, E)$ where the edges $E$ capture
information flows between program elements $V$ (referred to as “events” in the original Seldon paper, a usage which we will not follow). Program elements represented in the propagation graph include arguments to and return values of function calls, reads and writes of object properties or global variables, and any other construct that propagates information. Seldon uses standard points-to analysis to build such graphs.

Representing program elements. By their very nature, program elements are specific to a single code base, so in order to make sink predictions reusable across programs we need to assign code-base independent representations $\text{Rep}(v)$ to program elements $v \in V$. Seldon uses a variant of qualified names for this purpose; for example, the representation of the result of a function call could be the fully qualified name of the function (most specific), or an unqualified name (least specific). For example, the method $\text{findOneAndUpdate}$ used in Figure 1(b) has the two representations $\text{mongoose}.\text{Model}.\text{findOneAndUpdate}$ and $\text{findOneAndUpdate}$.

Building a constraint system. Seldon frames the problem of inferring taint specifications as a linear optimization task which can be solved using efficient solvers and makes their solution scalable.

For each program element $v \in V$ and each representation $n \in \text{Rep}(v)$ of $v$, Seldon instantiates three variables: $n^{\text{src}}$, $n^{\text{san}}$, and $n^{\text{snk}}$, each of which denotes the likelihood of $v$ being a source, sanitizer, or sink. Seldon adds constraints for: (1) constraining each variable to $[0, 1]$ to enable interpreting them as probabilities, and (2) setting appropriate variables to 1 or 0 for program elements whose specifications are known (from $\mathcal{AM}$).

Seldon then adds various constraints to encode their intuitions about information flow using the propagation graph. Figure 2 presents a visualization of one such constraint. Figure 2a indicates that if there is a flow from a sanitizer to a sink, then it is most likely sanitizing the output from a source. Figure 2b shows a propagation graph capturing such an occurrence. It indicates that if we have a program element $v_{\text{san}}$, classified as sanitizer, and another program element $v_{\text{snk}}$, classified as sink, and there is a flow from $v_{\text{san}}$ to $v_{\text{snk}}$, then we must classify at least one of the program elements $v_i$, $i \in \{1, \ldots, k\}$ which flow into $v_{\text{san}}$ as a source. Figure 2c presents the corresponding constraint that we add to encode this intuition.

$$\begin{align*}
(n^{\text{san}})^{\text{san}} + (n^{\text{snk}})^{\text{snk}} & \leq \sum_{i=1}^{k} (n^{\text{src}})_i \cdot C + \epsilon
\end{align*}$$

Figure 2: Intuition of Information Flow Constraints

Here, $n^{\text{san}} = \text{Rep}(v_{\text{san}})$, $n^{\text{snk}} = \text{Rep}(v_{\text{snk}})$, $n_i = \text{Rep}(v_i)$. $C$ is a fixed constant. Since the programs in the dataset may not always strictly respect the intuitions, Seldon also adds the $\epsilon$ variables (one for each constraint) to allow for minor deviations from the assumptions. Seldon adds one such constraint for each $(\text{san}, \text{snk})$ pair which has at least one source candidate flowing into the sanitizer. Seldon also adds other constraints analogously for pairs of $(\text{src}, \text{snk})$ and $(\text{src}, \text{san})$.

Solving the constraint system. Finally, Seldon solves the optimization problem by minimizing the sum of all relaxation variables ($\epsilon$) and the sum of all variable scores: $\sum_{i} n^t$, for $t \in \{\text{src}, \text{san}, \text{snk}\}$, subject to the specified constraints. Seldon then returns the confidence scores for each representation being a source, sink, or sanitizer. The inferred specifications (e.g., with some minimum confidence) can then be used to boost a taint analyzer. The spec is given by a list of triples $(n, \text{kind}, \text{score})$ where $\text{kind} \in \{\text{src}, \text{snk}, \text{san}\}$ and $n$ is a representation. Note that there can be many program elements with the same representation.

4 OUR APPROACH

We now describe the key technical components of InspectJS. At a high level, InspectJS takes as input a training set of JavaScript code bases $\mathcal{DT}$ and a set of seed specifications $\mathcal{AM}$ that can be used to identify known sinks $\mathcal{SM}$, i.e., program elements in $\mathcal{DT}$ that are known to be sinks, for example, as the result of manual modeling. InspectJS then processes the training set and infers a set of predicted sinks, i.e., program elements in $\mathcal{DT}$ that are likely to be sinks, but were previously unmodeled. Each of these is associated with a score between zero and one indicating the likelihood that the element is, in fact, a sink. As the final step of training, the predicted sinks, which are concrete program elements in $\mathcal{DT}$, are abstracted into code-base independent representations, which together with their associated scores form the predicted sink specifications $\mathcal{AP}$, or predictions for short.

For a given test set of JavaScript code bases $\mathcal{DT}$, these predictions can be instantiated to yield new predicted sinks on those code bases. Since some of the sinks may be false positives, they are not intended to be used directly, but to go through a two-step refinement process: one automated and one based on feedback from an analysis engineer. The final, reviewed set of sinks can then be used to improve manually-written models, or directly to find security vulnerabilities.

Figure 3 presents the overall architecture of InspectJS. These tasks are carried out by a pipeline of four components: Seldon*, GetSinks, Similarity-Based Refiner, and Feedback-Based Refiner.

Seldon* implements our adaptation of Seldon’s approach for predicting likely sinks. It takes $\mathcal{AM}$ and $\mathcal{DT}$ as inputs and produces the predicted sink specifications $\mathcal{AP}$. In general, a sink specification is a 3-tuple of the form $(\text{rep}, \text{kind}, p)$, where $\text{rep}$ is a program element representation, $\text{kind} \in \{\text{src}, \text{san}, \text{snk}\}$ denotes the role of the program element, and $p \in [0, 1]$ is a confidence score indicating the likelihood of the element assuming the given role. In $\mathcal{AM}$, all sinks are known.

Figure 3: InspectJS: System Overview
representations are assigned a confidence score of 1 (highest) since we already know their true roles.

GetSinks then instantiates the predicted sinks specifications \( \mathcal{A}_p \) on test projects \( D_E \) to produce new sinks \( \mathcal{S}_p \), which are tuples of the form \((v, p)\). Here, \( v \) is a program element from one of the test projects in \( D_E \) and \( p \in [0, 1] \) is the confidence score of the representation of \( v \) in \( \mathcal{A}_p \).

The Similarity-Based Refiner takes the inferred sinks \( \mathcal{S}_p \) and a set of precomputed embeddings of known sinks \( E_K \) as inputs. It implements a code-similarity based technique to adjust the confidence scores of inferred sinks according to how similar they are to known sinks, and returns the refined set of sinks \( \mathcal{S}_R \), which contains the same program elements as \( \mathcal{S}_p \), but with adjusted scores, and the embeddings for predicted sinks \( E_P \).

The role of the Feedback-Based Refiner is to validate these predictions computed over the test projects \( D_E \), presenting the predicted sinks along with their confidence scores to an analysis engineer to provide feedback about false positives, which the Feedback-Based Refiner eliminates, leaving a final set of refined sinks \( \mathcal{S}_F \). This module uses the embeddings of predicted sinks \( E_P \) provided by the Similarity-Based Refiner to identify sinks that may be similar to a false positive, allowing the engineer to efficiently eliminate groups of similar false positives at once.

### 4.1 Seldon*

Seldon* implements our Seldon-based approach for inferring likely taint sinks by framing the problem as a linear optimization task. The main differences with Seldon are in the way we compute the candidate triples, the use of one canonical representation per event instead of many candidates, and a more compositional approach to dealing with multiple projects where we build one constraint system per project and compose the optimization result. We now explain each of these in detail.

**Computing program elements and triples.** Just like Seldon, we start by extracting triples of the form \((v_{src}, v_{san}, v_{sink})\) from the training projects \( D_T \), denoting information flow from \( v_{src} \) to \( v_{san} \) and from there to \( v_{sink} \), where the three program elements are of the appropriate syntactic structure to potentially act as source, sanitizer, and sink, respectively, as explained in Section 3.

For capturing information flow between program elements, we use the standard inter-procedural taint tracking framework of CodeQL instead of the points-to analysis used by Seldon. The reason behind this choice is that we found that building propagation graphs for JavaScript using Seldon’s approach leads to both spurious and missing flows. We can also avoid ourselves of the support CodeQL provides for tracking flow through templating libraries and frameworks like Angular and React, which is not adequately captured by a simple Seldon-style points-to analysis.

For scalability reasons, we further restrict the set of sink candidates we build triples for by focussing on candidates from the most popular libraries as determined by the number of usages in JavaScript projects on LGTM.com.

**Program element representations.** We represent program elements using partial access paths [11], which are a generalization of qualified names to the setting of JavaScript with its highly dynamic object system and free use of higher-order functions. Access paths are build from three basic operators: property access \( p.q \), representing property \( q \) of the object represented by the base path \( p \); parameter access \( p(i) \), representing the parameter \( i \) of the function represented by the base path \( p \); and result access \( p() \), representing the return value of the function represented by the base path \( p \).

For instance, the first argument of the invocation of \( \text{findByIdAndUpdate} \) on line 5 of Figure 1(a) can be represented by the following three access paths:

1. \( \text{findByIdAndUpdate}(\theta) \), referring to it as the first argument to a method called \( \text{findByIdAndUpdate} \);
2. \( \text{getquerySendResponse}(\theta).*(\theta) \), referring to it as the first argument to some method (the name being left unspecified) of an object that is passed as the first argument to \( \text{getquerySendResponse} \);
3. \( \text{getquerySendResponse}(\theta).\text{findByIdAndUpdate}(\theta) \), which is similar but makes the name of the method concrete.

Instead of Seldon’s approach of allowing a program element to have multiple representations, we only select one canonical representation per program element, which reduces the complexity of the constraint system. For canonical representation we aim to find representations that are general enough to be common across different projects, but still specific enough to capture semantic differences. We choose the canonical representation by extracting features of the representation such as its length and the number of occurrences of the different kinds of accesses, and then assigning a score based on these features.

The weights for computing the score were determined semi-automatically by computing representations of known sinks on a large set of training projects and prioritizing common features. In our example above, the first representation is chosen as canonical because it provides enough information to obtain the program element, but disregards other details in favor of generality.

**Inferring new sink specifications using constraint solving.**

Next, we construct a constraint system using the same approach as Seldon and solve it using CBC [6]. However, while Seldon combines the constraints extracted from propagation graphs of all programs and builds a single optimization objective to feed into the solver, we found this approach difficult to scale. Instead, we solve constraints on a per-project basis, obtaining one set of sink prediction specifications per project. Then we average prediction scores across all projects to obtain sink prediction specifications \( \mathcal{A}_p \).

### 4.2 GetSinks

Once we get the sink prediction specifications \( \mathcal{A}_p \) we proceed to predict concrete sinks by instantiating the specifications on the test set \( D_E \). That is:

\[
S_U = \{ (v, p) \mid v \in \text{elements}(D_E) \land (\text{repr}(v), \text{snk}, p) \in \mathcal{A}_p \\
\land (\text{repr}(v), \text{snk}_{\_\_}) \notin \mathcal{A}_M \}
\]

Since our goal is to predict new sinks, we furthermore remove known sinks that are already modelled by the CodeQL library from this set. The GetSinks module implements both these steps using CodeQL library.
4.3 Similarity-Based Refiner

Like Seldon, Seldon+ assigns confidence scores based on representation, so all program elements with the same canonical representation are assigned the same score. However, sometimes representations can be too coarse, representing both true sinks and false positives. One way to minimize false positives is to use more specific representations. However, this can lead to the opposite problem, where real sinks are missed. Further, it also impacts scalability since it increases the number of constraint variables. The Similarity-Based Refiner tackles this problem by comparing the syntactic context of predicted sinks with a corpus of known sinks collected across many projects using code embeddings. Code embeddings are numerical vector representations of code components such as tokens or functions [3]. By mapping discrete data (code snippets) into a continuous vector space, code embeddings allow one to easily measure semantic similarity between code components.

Our intuition behind using code similarity is that a sink candidate which is used in a similar context as a known sink is more likely to be true sink than one which is used in a different context. Hence, we refine the predicted sinks by combining the confidence scores computed by Seldon+ and the code-similarity based scores.

Algorithm 1 describes our approach conceptually. It takes the predicted sinks $S_p$, and a set of embeddings of known sinks $E_K$ indexed by representations. These embeddings can be (but do not have to be) computed on the same set of projects. The algorithm then computes, for each prediction, the maximum similarity score with a known sink that has the same canonical representation, using cosine similarity of code embeddings as similarity metric as detailed in Algorithm 2 below. This is done both for the enclosing statement ($Z_{stmt}$) and the enclosing function ($Z_{func}$). The final score of the sink is computed on Line 6 as a combination of the original score and the two similarity-based scores.

In practice, we pre-compute the set of embeddings, $E_K$, from a set of known sinks for performance reasons. To obtain $E_K$, we build a training set comprising projects relevant to various queries (such as NoSqlInjection, XSS, and TaintedPath) and extract known sinks from them using CodeQL’s pre-defined models. Our intuition here behind this approach is that the sinks that are currently unmodeled for a query may be used in syntactic contexts similar to not only the known sinks of that query but also to the ones in different queries as well.

Algorithm 1 Refining predictions using Code Similarity

Input: Predicted sinks $S_p$, Known Sink Embeddings $E_K$
Output: Refined sinks $S_R$

1: procedure SIMILARITYBASEDREFINE($S_p, E_K$)
2: $S_R = \emptyset$
3: $E_P = \emptyset$
4: for $(s, p) \in S_p$ do
5:   $Z_{stmt}, Z_{func} = \text{COMPUTESIMILARITYSCORE}(s, E_K)$
6:   $S_R = S_R \cup (s, (p + (Z_{stmt} + Z_{func})/2)/2)$
7:   $stmt, func = \text{getEnclosingCode}(s)$
8:   $E_P(s) = (\text{Emb}(stmt), \text{Emb}(func))$
9: end for
10: return $S_R, E_P$
11: end procedure

Computing similarity using code embeddings. Given a predicted sink, we compute its similarity to known sinks using two kinds of code embeddings: 1) based on enclosing statement and 2) based on enclosing function. We use GraphCodeBERT [9] to compute these embeddings. GraphCodeBERT is a transformer-based model pre-trained using large corpus of programs on a general task. GraphCodeBERT can be fine-tuned to solve many downstream tasks in programming languages. For our work, we use a publicly available pre-trained GraphCodeBERT model used for clone detection. Algorithm 2 describes our approach.

Algorithm 2 Computing Max Similarity Score of a Sink

Input: Sink $s$, Embeddings $E_K$
Output: Statement similarity $Z_{stmt}$, Function similarity $Z_{func}$

1: procedure COMPUTESIMILARITYSCORE($s, E_K$)
2: $stmt, func = \text{getEnclosingCode}(s)$
3: $Z_{stmt} = \max \{ \theta(\text{Emb}(stmt), e_{stmt}) | (e_{stmt}, e_{func}) \in \text{embeddings} \}$
4: $Z_{func} = \max \{ \theta(\text{Emb}(func), e_{func}) | (e_{stmt}, e_{func}) \in \text{embeddings} \}$
5: return $Z_{stmt}, Z_{func}$
6: end procedure

The algorithm takes a sink program element $s$ and an embeddings map of known sinks $E_K$ (indexed by representations) as inputs and returns the maximum statement-based similarity $Z_{stmt}$ and function-based similarity $Z_{func}$ for $s$ to any known sink. The algorithm extracts the statement and function enclosing sink $s$ (Line 2). Then, it obtains the embeddings for the representation of $s$ (Line 3). Note that embeddings is a set of embeddings of sink program elements which have same representation as sink $s$. Each element in this set is a tuple containing the statement and function embeddings of a sink program element.

The algorithm then computes the similarity of the embedding of sink $s$ to each embedding in the embeddings set and stores the maximum similarity scores in $Z_{stmt}$ and $Z_{func}$ (Lines 4-5). Here, $\theta$ is a function which computes the embedding of a given code snippet (statement or function) using GraphCodeBERT, and $\theta$ is a function which computes the cosine similarity of two input vectors, i.e., the embeddings in this case. Finally, it returns the two maximum similarity scores (Line 6).

Combining the scores. Once, we compute the similarity scores, we combine it with the confidence scores in Algorithm 1, Line 6. This step improves the scores of predicted sinks which are similar to one or more known sinks and penalizes the scores of predicted sinks which are dissimilar.

Computing similarity for sink predictions. In addition to ranking the sinks prediction the Similarity-Based Refiner also computes embeddings $E_P$ for the sinks predictions $S_p$. Algorithm 1 instantiates $E_P$ as empty set (Line 3) and updates the sink embeddings in each iteration (Line 8).

4.4 Feedback-Based Refiner

The Feedback-Based Refiner allows domain experts to further refine the predictions by incorporating their feedback. The Feedback-Based Refiner provides a User Interface (UI) which displays all the predicted sinks sorted by confidence scores.

Figure 4 shows a screenshot of the UI. It provides two options with each prediction: “ban”, which hides the corresponding sink;
and “ban similar”, which hides the corresponding sink as well as other sinks which are similar (up to some pre-selected threshold). The UI also shows the list of representations sorted by the number of sinks each corresponds to. Algorithm 3 describes how we obtain the set of sinks that are similar to given sink \( s \). The algorithm computes the similarity score of each sink \( s' \) which has the same representation as given sink \( s \) (Lines 3-13) using the predicted sink embeddings \( E_p \), computed by the Similarity-Based Refiner. It then selects sinks that have a similarity score above user-defined threshold \( \alpha \) and returns the selected sinks to the caller.

The goal of our UI is to allow an experienced developer to quickly and efficiently triage the list of predicted sinks. The developer can remove individual (or similar) sinks that they consider to be false positives. They can also filter the predictions by de-selecting representations. For instance, a representation which matches too many sinks may indicate that they are too coarse and mostly generate false positives. The developer can easily hide the corresponding predictions for such representations.

**Algorithm 3 Filter similar sinks using Code Similarity**

```
Input: Sink \( s \), Predicted sinks \( S_p \), Sink embeddings \( E_p \), Similarity Threshold \( \alpha \)
Output: similar sinks \( S \)
1: function GetSIMILAR(s, \( S_p \), \( E_p \), \( \alpha \))
2:     \( S = \emptyset \)
3:     for \( (s', \theta) \in S_p \) do
4:         if \( \theta(s) = \theta(s') \wedge s \neq s' \) then
5:             \( \epsilon_{\text{sim}}(s, \theta_{\text{sim}}(s')) = E_p(s) \)
6:             \( \epsilon_{\text{norm}}(s, \theta_{\text{norm}}(s')) = E_p(s') \)
7:             \( Z_{\text{sim}} = \theta(\epsilon_{\text{sim}}, \epsilon_{\text{norm}}) \)
8:             \( Z_{\text{norm}} = \theta(\epsilon_{\text{norm}}, \epsilon_{\text{norm}}) \)
9:             if \( (Z_{\text{sim}} + Z_{\text{norm}})/2 > \alpha \) then
10:                \( S = S \cup s' \)
11:         end if
12:     end if
13: end for
14: return \( S \)
15: end function
```

5 EVALUATION

To evaluate the practical usefulness of InspectJS we pose ourselves the following research questions:

**RQ1:** Does InspectJS find new sinks that are as yet not covered by hand-written models?

**RQ2:** How much effort does it take to triage InspectJS results?

**RQ3:** How important are the different components of InspectJS?

**RQ4:** Can the predicted sinks be used to highlight new security alerts?

**Choosing JavaScript security queries.** For all research questions, we focus on three representative CodeQL security queries addressing some of the top 25 software vulnerabilities identified by MITRE CWE Top 25:

- TaintedPath
- XSS
- NoSQLInjection

TaintedPath detects path-traversal vulnerabilities where a potentially malicious user can control the path of a file being read or written; XSS detects client-side cross-site scripting vulnerabilities where a potentially malicious JavaScript code can be injected into the DOM of a webpage; and NoSQLInjection detects NoSQL-injection vulnerabilities where a user can insert JavaScript code into a NoSQL query.

<table>
<thead>
<tr>
<th>Query</th>
<th># Results</th>
<th># TPs</th>
<th>Min TP score</th>
<th>Coarsest TP repr</th>
<th>Max TP/FP similarity</th>
</tr>
</thead>
<tbody>
<tr>
<td>TaintedPath</td>
<td>4,614</td>
<td>56</td>
<td>0.58</td>
<td>3%</td>
<td>0.91</td>
</tr>
<tr>
<td>XSS</td>
<td>10,504</td>
<td>436</td>
<td>0.75</td>
<td>7%</td>
<td>1.00</td>
</tr>
<tr>
<td>NoSQLInjection</td>
<td>1,473</td>
<td>187</td>
<td>0.58</td>
<td>16%</td>
<td>0.93</td>
</tr>
</tbody>
</table>

Finding representative JavaScript projects per query. To empirically evaluate the effectiveness of InspectJS and answer our research questions, we need a corpus of JavaScript code to train our model on and produce new predictions. For this purpose, we choose open-source projects from GitHub. While there is no shortage of such projects, selecting projects at random would most likely have left us with projects that do not use any APIs relevant to the three queries we focus on. Instead, we choose projects where the existing CodeQL query produces at least one alert (and hence the existing library models identify at least one sink), the intuition being that these projects perhaps also use other API, or as yet unmodeled parts of APIs, relevant to the query.

To select candidate projects, we ran a query on all JavaScript projects on LGTM.com [8], a cloud platform for running CodeQL analyses at scale on large numbers of open-source repositories, in

---

1 https://cwe.mitre.org/top25/archive/2021/cwe_top25.html
2 https://git.io/JrRXW
3 https://git.io/JrRw6
4 https://git.io/JrRy
5 https://git.io/JrRNQ
May 2021. Among the roughly 200,000 projects we queried, we found 562 projects satisfying our criteria for TaintedPath, 2834 for XSS, and 833 for NoSQLInjection.

We conducted two different experiments, one to address the first three research questions, and the other to address the fourth question. We will now describe the setup and outcomes of each experiment in turn, and answer the research questions.

5.1 Experiment 1: Manually labelling sink predictions

For our first experiment, we used InspectJS to automatically identify sinks for the three CodeQL queries, and then inspected the results.

Experimental Setup. To keep the number of predictions manageable, we randomly selected 100 projects per query, and then split each set into a training set of 90 projects and a held-back test set of 10 projects. We trained the InspectJS model on the training set and produced predictions for the test set, filtering out any previously known sinks for which CodeQL already has manually written models. Finally, the fourth author (an experienced CodeQL analysis engineer) manually labelled the predictions as true positives (that is, sinks that are currently not modelled by the CodeQL standard library but arguably should be), or false positives.

Predictions. Table 1 presents the results of this experiment. Each row presents the results of one query. For each query, column #Results presents the total number of sink predictions and column #TPs presents the number of true positives. This data allows us to answer RQ1 in the affirmative: InspectJS does indeed find new sinks. We have reported missing sinks in CodeQL identified by InspectJS to the CodeQL library maintainers on several occasions, which has already led to numerous improvements to the manual models.

However, it is immediately obvious that the raw output of the ML model is too noisy to be useful, with only a few percent of predictions being true positives. This motivates the need for a tool like InspectJS to allow an analysis engineer to efficiently triage the set of predictions and prune out false positives.

As described previously, InspectJS provides three metrics for categorizing predictions: the score of a prediction, the coarseness of its representation (that is, the percentage of all predictions that have this representation), and the similarity of different predictions. The intuition is that a prediction with a low score or high coarseness is likely to be a false positive, and that false positives are likely to be similar to each other, but not to true positives.

Table 1 shows some statistics that allow us to test this claim. Column Min TP score presents the minimum score of a true positive, which is above 0.5 for each query; this suggests that predictions with a score below 0.5 can be disregarded in practice. Column Coarsest TP repr presents the maximum coarseness of a true positive, that is, the percentage of predictions that have the same representation as a true positive. This value varies quite a bit between queries, from 3% for TaintedPath to 16% for NoSQLInjection. Disregarding predictions whose representation accounts for more than 20% of all predictions seems like a reasonably safe thing to do in practice, but the evidence is not clear cut in this case.

Table 2: Metrics for triaging effort, with similarity threshold 0.95 and coarseness threshold 20%

<table>
<thead>
<tr>
<th>Query</th>
<th># Discarded (score + coarseness)</th>
<th># Remaining FPs</th>
<th># Steps to triage</th>
<th>FN</th>
</tr>
</thead>
<tbody>
<tr>
<td>TaintedPath</td>
<td>3,007 (1,025 + 1,982)</td>
<td>1,548</td>
<td>523</td>
<td>0</td>
</tr>
<tr>
<td>XSS</td>
<td>2,136 (2,136 + 0)</td>
<td>7,932</td>
<td>2,874</td>
<td>10</td>
</tr>
<tr>
<td>NoSQLInjection</td>
<td>666 (666 + 0)</td>
<td>628</td>
<td>245</td>
<td>0</td>
</tr>
</tbody>
</table>

Finally, column Max TP/FP similarity shows the maximum similarity between a true positive and a false positive. Recall that our prototype allows a user to dismiss not just a single false positive they have identified, but also all other predictions that are sufficiently similar to it. Here, “sufficiently similar” should be chosen in such a way that it is unlikely that any of the predictions dismissed alongside the false positive are true positives. Unfortunately our experiment shows that this is not achievable with our current similarity metric: for XSS, there are true positives that are indistinguishable from false positives in terms of code similarity, meaning their similarity score is 1. For the other queries, a similarity score of 0.95 looks to be a safe cut-off.

Triaging effort. To estimate the effort required to triage the set of predictions, we count the number of predictions that are discarded due to not meeting the cut-off for score or coarseness, and the number of steps that would be required to triage the remaining predictions, as well as the number of true-positive predictions that would be wrongly discarded during this process. These are, of course, best-case estimates since we are using cut-offs established on the same dataset.

Table 2 presents the results of this computation. Column # Discarded shows the number of predictions that are discarded (with details on how many were discarded due to low score and high coarseness, respectively, in brackets); column # Remaining FPs shows the number of false-positive predictions that are not discarded; column # Steps to Triage shows the number of steps needed to triage the remaining predictions; and column # FNs the number of true results that are missed in this process. We can see that score and coarseness act as a very useful first filter, discarding 65% of predictions for TaintedPath, 20% for XSS, and 45% for NoSQLInjection. After that, the analysis developer still needs to identify and dismiss the remaining false positives, but as the table shows the similarity-based multi-dismissal feature significantly reduces that effort, which each step on average dismissing about three false positives in one go.

For XSS, multi-dismissal results in ten false negatives, since, as we discussed above, there are true positives that are indistinguishable from false positives in terms of code similarity. For the other queries, the number of false negatives is zero.

Figure 5 shows how the number of steps and the number of false positives vary with the similarity threshold: for each of our three queries, we compute both metrics for similarity thresholds between 0.80 and 0.95. As expected, decreasing the similarity threshold makes triaging faster: with a threshold of 0.80, the number of steps is 55 for TaintedPath, 157 for XSS, and 43 for NoSQLInjection, which is about 1/10th to 1/20th the number for a threshold of 0.95 as shown in Table 2. Of course, this comes at the price of missing true
positives: at similarity threshold 0.80, TaintedPath misses eight true positives, XSS 187, and NoSQLInjection 117, which is significantly above the values for 0.95 shown in Table 2.

Our answer to RQ2 is, therefore, nuanced: an analysis engineer using InspectJS needs to be aware that the vast majority of predictions are likely to be false positives, but the various metrics provided by InspectJS can be used to trade off triaging effort against them.

Importance of InspectJS components. The novelty of InspectJS lies in combining the triple-mining approach of Seldon with a code-similarity metric to weed out false positives. A natural question lies in combining the triple-mining approach of Seldon with a code-similarity metric to weed out false positives, but the various metrics provided by InspectJS can be used to trade off triaging effort against them.

On one hand, we might conceivably do away with the triple computation altogether, and simply consider all data-flow nodes that could potentially be sinks, relying entirely on code similarity to rank and triage them. However, this is not a viable approach: the number of potential sinks across our ten test projects is 992,035 for TaintedPath, 1,308,150 for XSS, and 105,651 for NoSQLInjection. Overall, this means that the triple-computation step reduces the number of predictions by about two orders of magnitude.

On the other hand, we could discard the code-similarity step, but, as already discussed, Table 2 shows that without similarity-based multi-dismissal the triaging effort would be about three times as big. In summary, then, our answer to RQ3 is that both components of our approach make their own important contribution towards easing the reviewing burden.

5.2 Experiment 2: Analyzing security alerts
To answer RQ4, we take our three queries and boost them, that is, we use InspectJS to predict new sinks on a training set of projects, and then include them among the set of sinks recognized by the query to yield a boosted query. We then run that boosted query on a test set of projects, and consider the new alerts it produces on these projects (compared to the original query), and evaluate whether they are correct.

Experimental Setup. Manually evaluating whether new alerts are correct is labor-intensive, of course, so we use an alternative strategy to evaluate InspectJS’s performance: for each given CodeQL query $Q_{\text{new}}$, we obtain an older version $Q_{\text{old}}$ of the same query from the version history of CodeQL. This query will have the role of $A_M$ and we boost it with InspectJS to produce $A_M^{\text{boosted}}$. We call this query $Q_{\text{boosted}}$. Finally, we compare the alerts generated by the boosted query on a set of test projects with the alerts generated by $Q_{\text{new}}$ on the same set of projects, which we use as ground truth.

Table 3: Results from comparing old versions of queries boosted with InspectJS to the latest version of the same query. Averaged over three runs on 200 projects, with random 50-50 splits to obtain test and training sets in each round.

<table>
<thead>
<tr>
<th>Query</th>
<th>Alerts Recovered</th>
<th>Spurious Alerts</th>
<th>Projects with Alerts to Recover</th>
</tr>
</thead>
<tbody>
<tr>
<td>TaintedPath</td>
<td>58.33</td>
<td>46</td>
<td>19</td>
</tr>
<tr>
<td>XSS</td>
<td>15</td>
<td>14</td>
<td>406</td>
</tr>
<tr>
<td>NoSQLInjection</td>
<td>303</td>
<td>266.67</td>
<td>719.67</td>
</tr>
</tbody>
</table>

To run this experiment, we selected 200 different projects for each of the three queries we consider. Then, for each query we run three rounds of the boosting process described above, randomly splitting the projects into 100 projects for training and 100 for testing in each round.

Results. In Table 3 we present the results averaged over the three rounds for each query. Column Alerts to Recover shows the average number of new alerts produced by $Q_{\text{new}}$ that are not flagged by $Q_{\text{old}}$. Column Alerts Recovered shows how many of these new alerts are found by the boosted query $Q_{\text{boosted}}$ on average. Conversely, Column Spurious Alerts shows how many of the new alerts from the boosted query are not flagged by $Q_{\text{new}}$. We consider them as false positives, even though it is possible that some of them are actual true positives not captured by $Q_{\text{new}}$. Finally, column Projects with Alerts to Recover shows how many of the 100 test projects had any alerts to recover on average.

In response to RQ4, we can say that InspectJS succeeds in predicting sinks that lead to security alerts, and its recall with respect to new query versions is high. The false positive rate is also quite high, however, which agrees with the results of Experiment 1. It is worth noting that in this experiment we do not filter out predictions with very coarse representations, which may exacerbate this problem, and of course (as noted above) our labeling of false positives is pessimistic, so the actual number of false positives is lower.

5.3 Threats to validity
The main threat to the validity of the results from Experiment 1 is bias in the manual labelling. To counter this threat, we randomly selected 20 predictions for each query and gave them to CodeQL experts not involved in this project to label, and compared the results with our own labelling. For TaintedPath we agreed on all 20 predictions, for XSS on 17 (with the external expert marking three predictions as true positives that we had dismissed as false positives).
which are more likely to have similar roles (e.g., sinks). JSNice [14] is a re-implementation of Seldon for JavaScript. However, their approach relies on static program properties for JavaScript. Typilus [1] is a Graph Neural Network (GNN) based approach for predicting variables types for Python. As such, these techniques may also be leveraged by future approaches to improve the precision of InspectJS’s results.

7 CONCLUSION
In this paper, we described our experience combining machine-learning based taint specification inference of sinks along with manual modelling for important CodeQL security queries for JavaScript. We also describe how we leverage code-similarity metrics and user-feedback to help analysis engineers effectively triage the predictions to prune spurious predictions.

In future work, we plan to extend InspectJS to infer source specifications, as well as taint-flow and aliasing specifications. We are also working on incorporating approaches based on abductive inference of library specifications [18]. Finally, it could be interesting to apply our feedback-based refiner to other approaches for inferring taint specifications, since it is not technically tied to Seldon*.

ACKNOWLEDGMENTS
We thank Ian Wright and Madan Musuvathi for many insightful discussions which greatly improved InspectJS. This work is partially funded by ANPCYT PICT 2018-3835, PICT 2019-1442, CONICET PIP 0731CO and UBACYT 2018-0419BA projects and Microsoft Azure research grants.

REFERENCES
[14] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting program properties from big code. In POPL.