Beware of the Unexpected: Bimodal Taint Analysis

Yiu Wai Chow
University of Stuttgart
Stuttgart, Germany
victorcwai@gmail.com

Max Schäfer
GitHub
Oxford, UK
max-schaefer@github.com

Michael Pradel
University of Stuttgart
Stuttgart, Germany
michael@binaervarianz.de

ABSTRACT

Static analysis is a powerful tool for detecting security vulnerabilities and other programming problems. Global taint tracking, in particular, can spot vulnerabilities arising from complicated data flow across multiple functions. However, precisely identifying which flows are problematic is challenging, and sometimes depends on factors beyond the reach of pure program analysis, such as conventions and informal knowledge. For example, learning that a parameter `name` of an API function `locale` ends up in a file path is surprising and potentially problematic. In contrast, it would be completely unsurprising to find that a parameter `command` passed to an API function `execaCommand` is eventually interpreted as part of an operating-system command. This paper presents Fluffy, a bimodal taint analysis that combines static analysis, which reasons about data flow, with machine learning, which probabilistically determines which flows are potentially problematic. The key idea is to let machine learning models predict from natural language information involved in a taint flow, such as API names, whether the flow is *expected* or *unexpected*, and to inform developers only about the latter. We present a general framework and instantiate it with four learned models, which offer different trade-offs between the need to annotate training data and the accuracy of predictions. We implement Fluffy on top of the CodeQL analysis framework and apply it to 250K JavaScript projects. Evaluating on five common vulnerability types, we find that Fluffy achieves an F1 score of 0.85 or more on four of them across a variety of datasets.

1 INTRODUCTION

Taint analysis is a powerful technique for detecting various kinds of programming mistakes, including both security vulnerabilities and other kinds of bugs. A taint analysis tracks the flow of information from a *source* to a *sink* and reports a warning if the existence of this flow violates some pre-defined policy. For example, such a policy might specify that data from an HTTP request sent by an unknown and potentially malicious user (the source) must not flow into an API that executes strings as operating-system commands (the sink) to prevent remote code execution, a type of vulnerability known as *command injection.* As another example, a policy might state that unencrypted passwords or other secrets must not flow into logging statements to prevent accidentally leaking them through *clear-text logging.* Taint analysis is an integral part of popular static analysis solutions, such as Facebook Infer or GitHub Code Scanning, that regularly scan large amounts of proprietary and open-source software.

A lot of research and engineering has gone into developing precise and scalable algorithms for inter-procedural taint tracking for various languages, ensuring that modern taint analyses can scale to even the largest of code bases. However, in the end a taint analysis is only as good as the policies it enforces: If a source or sink is missing, the analysis will not report problematic flows involving it, leading to false negatives and hence missed vulnerabilities. If, on the other hand, a source or sink is included that should not be, the analysis will report false positives, leading to unnecessary work and frustration for developers and security researchers.

In some cases, such as the command injection example above, suitable sources and sinks can be identified by carefully examining the APIs of the analyzed libraries, which is a well-defined if perhaps tedious process that can cover only a limited set of APIs due to the high human effort. For clear-text logging, on the other hand, the solution is not so obvious. While it is easy to identify the sinks (calls to logging libraries), it is much less clear what constitutes a source, since the way sensitive data are stored and handled is usually highly application-specific.

As another example, consider the problem of detecting "unhygienic" APIs, that is, library APIs that use data from a client application in a security-sensitive context without clearly indicating to the client that this is the case (for example by appropriate naming or documentation). Depending on whether the security-sensitive use is intentional or not, such APIs either suffer from a vulnerability or provide insufficient documentation, both of which put client applications at risk. As an example, the API of the JavaScript library `moment` for handling times and dates contains a `locale` function for setting the locale used for pretty-printing purposes. Before version 2.29.2, this function treats its parameter `name` (the name of the locale to use) unsafely, leading to remote code execution if untrusted data is passed to it.[1] Nothing about the name of the function or the parameter suggests that the parameter directly impacts an executed command, and indeed, this behavior was not intentional but a consequence of sloppy coding practices. A client relying on API names and documentation, might well pass data obtained from an untrusted third party to the API and thereby open themselves up to

---

[1]CVE-2022-24785.

a vulnerability–as some, indeed, did before the moment developers fixed the problem.[2]

Identifying such unhygienic APIs seems to be a natural fit for taint analysis, but while it is easy to define the sinks (they are identical to the sinks for command injection), the set of sources is harder to pin down. Just treating all API entry points as sources would not do: For example, the library execa, which provides improved cross-platform support for spawning child processes on Node.js, exports a function execaCommand with a parameter command that flows into a command-injection sink. However, flagging this flow as a vulnerability makes no sense, since it is precisely the intended functionality.

The common denominator in both of these examples is that the flows we are looking for are in some sense *unexpected* in that they are at odds with the intentions and expectations of developers: In a production code base, one would not expect to see sensitive information being logged in clear text, and likewise one would not expect a library to use data in a security-sensitive context without clearly indicating to the client that this is the case.

To address the problem of detecting such unexpected taint flows, we present Fluffy ("Flagging unexpected flows for better security") a *bimodal taint analysis* that exploits the fact that source code conveys meaning both through the programming language semantics and through natural language information embedded in code, e.g., in the form of identifier names [1, 26, 45]. Fluffy allows for implementing security policies where the set of sources is partially determined using natural language information.[3]

The approach consists of two steps. First, we run an off-the-shelf, monomodal taint analysis that *overapproximates* the set of sources so it can be captured purely in terms of code constructs without any natural language reasoning. For example, for clear-text logging we consider *all* variables to be sources, and for unhygienic APIs all parameters. Secondly, the *candidate flows* resulting from the first step are filtered by a machine learning model that predicts whether the source in a candidate flow is, in fact, a true source. Our approach combines the power of logic-based static taint analysis, which reasons about data flow and control flow dependencies, and machine learning models, which probabilistically "understand" the meaning of natural language information.

The Fluffy approach is a general framework with four concrete instantiations that use different ways of formulating the prediction task, build on different machine learning models, and impose different demands for manually labeled data. The four instantiations are (i) a neural, binary classifier trained on manually labeled examples, (ii) a neural model trained on millions of taint flows extracted automatically via static analysis, (iii) a novelty detection technique based on a one-class support vector machine, and (iv) a technique that queries a large language model (Codex [14]) via few-shot learning.

Our work relates to combinations of program analysis and ML models for other analysis problems, such as call graph pruning [33, 51] and filtering null dereference warnings [30]. Fluffy differs by addressing the problem of identifying unexpected taint flows. Another related line of work is a purely neural taint analysis [50], which

trains a neural model to "emulate" a dynamic taint analysis more efficiently than an actual dynamic analysis would be. In contrast, our work leverages the complementary power of static analysis and machine learning. Finally, Fluffy also relates to prior work on automatically annotating APIs as sources and sinks using a trained model [48]. Their work relies on a set of hand-coded features, such as whether a method returns a specific type, and it makes predictions about source and sink APIs. In contrast, we exploit pre-trained neural models that avoid manual feature engineering and propose models that make predictions about specific flows between already identified sources and sinks. Overall, our work is the first to present a bimodal analysis for the problem of taint tracking.

We implement the approach in the CodeQL analysis framework [22] and apply it to 250k JavaScript and TypeScript projects. Our evaluation considers five taint analyses aimed at detecting integrity problems, such as command injections, and confidentiality problems, such as clear-text logging. We find Fluffy to be effective at determining unexpected taint flows, with 81%–97% precision, 80%–100% recall, and 76%–97% F1-score, depending on the analysis. To assess the usefulness of the approach, we apply Fluffy to 131 confirmed vulnerabilities from the past, of which it successfully detects 117. Finally, we report 16 newly detected vulnerabilities to the respective developers, who have so far confirmed eight of them.

In summary, this paper contributes the following:

- A bimodal taint analysis, combining static analysis with machine learning to identify problematic flows.
- A general framework with four instantiations, offering trade-offs between labeling effort and prediction accuracy.
- An integration into CodeQL and empirical evidence that the approach is effective and useful in practice.

To foster future work, our implementation and experimental results are publicly available.[4]

2 BACKGROUND: TAINT ANALYSIS WITH CODEQL

CodeQL [22] is an open-source static program analysis system, which powers LGTM.com and GitHub CodeScanning. Following the code-as-data paradigm, code is represented as relational data and analyses are expressed as queries written in QL [5], an object-oriented extension of Datalog. CodeQL offers extensive standard libraries for performing typical analysis tasks, including a framework for global taint analysis that underpins a suite of individual analyses for finding common vulnerabilities, such as command injection or path traversal.

Security policies are also expressed in QL, typically identifying uses of particular APIs as sinks and sources of taint. In the case of JavaScript, for instance, the CodeQL query for identifying command-injection vulnerabilities[5] identifies various sources of potentially untrusted user input, such as HTTP request parameters accessed through the express npm package, as sources, and calls to command execution APIs, such as those in the Node.js

---

[2]For example, see advisory GHSA-7v28-g2pq-ggg8.
[3]The case of sinks is symmetric, but in this paper we focus on sources only.

[4]https://figshare.com/s/1ab456424bfb5a2ead5e
[5]https://github.com/github/codeql/blob/main/javascript/ql/src/Security/CWE-078/CommandInjection.ql.

`child_process` module, as sinks. The path-traversal query[6] shares the same set of sources, but uses file-system operations as its sinks.

Both of these are examples of *integrity violations* whereby untrusted data is used in a security-sensitive context. Other examples of such violations also covered by CodeQL queries include code injection,[7] where untrusted data is used to construct code to be executed, and reflected cross-site scripting ("reflected XSS" for short),[8] where untrusted data is embedded in an HTTP response.

As mentioned above, some analyses are not amenable to this kind of formulation. One example is the problem of finding clear-text logging of sensitive information, an example of a *confidentiality violation* in which security-sensitive data is exposed in an untrusted context. The CodeQL query for finding clear-text logging[9] employs hand-crafted regular expressions to identify variables and properties whose names suggest they might contain sensitive data, which are then treated as sources for the analysis. As we will demonstrate below, this is error-prone and a manual update to the libraries is required every time a spurious or missing source is discovered.

As another example, the CodeQL query *Unsafe shell command constructed from library input*[10] attempts to identify unhygienic library APIs by detecting taint flow from API parameters to shell commands, excluding parameters whose name suggests they are meant to be commands. Again, a regular expression is used for this purpose, with the same drawbacks as above.

Recently, CodeQL has been extended with experimental support for leveraging machine-learning techniques to discover new sources and sinks that are not covered by the standard security policies [21]. While this helps with identifying sources and sinks in rarely-used or proprietary APIs, it is not clear whether it offers any advantages in situations where natural language information is needed since it is currently only available for four queries, none of which fall into this category.
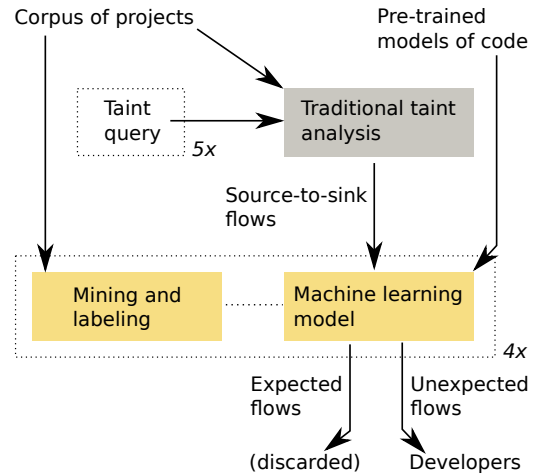
3 APPROACH

This section presents Fluffy, our bimodal taint analysis, in more detail. After defining the problem addressed by the approach (Section 3.1), Section 3.2 gives an overview of our general framework. The remaining subsections then present four instantiations of our framework, which explore different trade-offs in the design space.

3.1 Problem Statement

The input to Fluffy is a taint-style security policy $\mathcal{P}$ and a code base C to check against that policy. The security policy specifies a set of problematic flows for C as a set of source-sink pairs (l_{src}, l_{sink}) for which no data should propagate from the source location l_{src} to the sink location l_{sink} in C.

This specification consists of two parts $\mathcal{P}_F$ and $\mathcal{P}_U$. The first part $\mathcal{P}_F$ is *precisely specified but overapproximate*, determining a

Figure 1: Overview of the approach. The yellow components are the contributions of this paper.

set F of potentially problematic flows that can be computed purely in terms of code structure, for instance by a standard taint analysis. The second part $\mathcal{P}_U$ is *imprecisely specified*, describing a set U of unexpected flows. Whether a flow is unexpected is an inherently fuzzy problem, because the expectations of developers depend on common coding conventions and informal information, such as identifier names and API documentation.

The problem addressed in this paper is to identify all flows in a given code base that fulfill both parts of the security policy, i.e., to find the set $\{(l_{src}, l_{sink}) \mid l_{src} \in C, l_{sink} \in C, (l_{src}, l_{sink}) \in F \cap U\}$ of actually problematic flows to show to the developer.

3.2 Overview

Motivated by the fuzziness of the problem, we address it with learning-based techniques, which have been shown to effectively capture inherently fuzzy information [45]. Figure 1 gives an overview of the approach. In line with the two-part specification of the problem, the approach consists of two main steps. Given a project to analyze, the first step is to apply a traditional taint analysis. We build upon CodeQL, which offers a static code analysis engine and a suite of standard taint analyses as described in Section 2. The taint analysis $T: C \mapsto F$ identifies all flows between source-sink pairs in the given code base C that are in F. In the second step, the source-sink flows detected by the taint analysis are then given to a machine learning model $M: F \mapsto U$ that predicts which flows in F are likely unexpected by the developers, and hence should be reported to the developers. The machine learning model builds on existing, pre-trained models, e.g., to embed code tokens into a continuous vector, and is supported by a mining and labeling component, which provides training data to learn from.

The overall framework described in Figure 1 can be instantiated in several ways. One dimension is the taint analysis to use; we focus on five CodeQL analyses for JavaScript: code injection, command injection, reflected XSS, path traversal, and clear-text logging. The other dimension is the machine learning model to use; we present four models that explore different trade-offs in the design space.

---

[6]https://github.com/github/codeql/blob/main/javascript/ql/src/Security/CWE-022/TaintedPath.ql.

[7]https://github.com/github/codeql/blob/main/javascript/ql/src/Security/CWE-094/CodeInjection.ql.

[8]https://github.com/github/codeql/blob/main/javascript/ql/src/Security/CWE-079/ReflectedXss.ql.

[9]https://github.com/github/codeql/blob/main/javascript/ql/src/Security/CWE-312/CleartextLogging.ql.

[10]https://github.com/github/codeql/blob/main/javascript/ql/src/Security/CWE-078/UnsafeShellCommandConstruction.ql.

Table 1: Four instantiations of our general framework.

Approach	Training data	Model
Binary Classification	Hundreds of manually labeled flows	Neural binary classifier, predicts if a flow is expected
Sink Prediction	Millions of unlabeled flows	Neural model, predicts the most likely sink
Novelty Detection	Fewer than ten relevant terms for each sink type	One-class SVM, identifies unusual flows
Codex Language Model	Ten examples given in the prompt	Generative language model, predicts if a flow is expected

Table 1 gives an overview of the four machine learning models, which are described in detail in Sections 3.3 and 3.4.

3.3 Gathering and Representing Natural Language Information

Running the queries described above against a code base yields a set F of flows, i.e., source-sink pairs where data from the source propagates to the sink. To determine whether a flow is likely unexpected by developers, Fluffy exploits natural language information associated with the flow. Specifically, the approach gathers the following information about each flow:

- The identifier name $n_s \in N$ of the source, which is the parameter p of an API function f for the integrity queries and a source variable v for the confidentiality query.
- Additionally, for the integrity queries, the name $n_f \in N_{fct} \subseteq N$ of f and any documentation $d \in D$ for p.

To allow a machine learning model to effectively reason about the extracted natural language information, we embed it into a continuous vector representation. For this purpose, Fluffy uses a pre-trained embedding function $e : N \rightarrow \mathbb{R}^k$ that maps a natural language string in N to a k-dimensional vector in such a way that semantically similar strings have similar vector representations. As the embedding function, we use VarCLR [15], which we select for two reasons. First, VarCLR has been shown on the IdBench benchmark [53] to be the state of the art for the task of mapping identifier names to continuous vectors in a way that preserves semantic relatedness and similarity. Second, while comparing VarCLR against other pre-trained embeddings, e.g., FastText [9], in preliminary experiments, we found VarCLR to produce the best overall results.

3.4 Predicting Unexpected Flows

Based on the flows F extracted by traditional taint analysis and the natural language information associated with each flow, the core step of Fluffy is to determine the subset $U \cap F$ of flows that are unexpected and hence should be reported to developers. The following presents four machine learning-based techniques for this purpose, which differ in the kind and amount of labeled training data they require, the kind of model they build on, and (as shown in Section 4) their ability to effectively identify unexpected flows.

3.4.1 Approach 1: Binary Classification. We train a neural model M that predicts the probability that a given flow is unexpected. As its input, the model receives embeddings of the identifier name of the source, and if available the other natural language information associated with the flow. That is, the model is the following learned function:

$$M\colon N \times N_{fct} \times D \rightarrow \{Expected, Unexpected\}$$

The model is implemented using a standard feed-forward neural network with two hidden layers and the softmax function applied to the output layer, which we train with cross-entropy loss. The names n_s and n_f are embedded using the embedding function e. For the documentation d, we embed each token using a jointly trained embedding layer and encode the sequence of embedded tokens using an LSTM-based, bidirectional recurrent neural network. We train a separate model for each of the five taint queries, using hundreds of manually labeled flows (details in Section 4).

3.4.2 Approach 2: Sink Prediction. The Binary Classification approach requires a significant amount of manually labeled training data. To avoid this human effort, to following approach is trained only on automatically mined training data. The basic idea is to train a neural classification model that predicts for a given source what sink it is supposed to flow into.

For the integrity queries, this corresponds to the function

$$M\colon N \times N_{fct} \times D \rightarrow \{CmdInj, CodeInj, XSS, PathTrav, None\}$$

where the first four output classes correspond to the sinks of the four integrity queries, and *None* means that a source location does not propagate to any of the any four sinks. Likewise, for the confidentiality query, the model is the following learned function:

$$M\colon N \rightarrow \{Logging, None\}$$

where *Logging* means that it is unproblematic for the source contents to be logged, while *None* means that it should not be logged.

The model uses the same architecture as for the Binary Classification, but with an output layer that has the appropriate length for the classification problem. Training data is collected using CodeQL, as explained in Section 4.1.1, without any need for manual labeling.

Once the model has been trained, Fluffy queries it with previously unseen flows to determine whether the flow is unexpected. If the flow involves a sink type $s \neq None$ and the model predicts s as the most likely sink, then we consider the flow to be expected. The rationale is that if sources with particular names commonly flow this sink type, then developers are likely aware of this. Otherwise, if the model predicts some other sink $s' \neq s$ (where s' may be *None*), we consider the flow to be unexpected and report it.

3.4.3 Approach 3: Novelty Detection. The previous two approaches require training data labeled to belong to different classes. Our third approach instead uses a novelty detection technique trained only on examples of one class. We use a one-class support vector machine (OC-SVM) [49] that predicts whether a previously unseen example belongs to this class or is "novel".

Table 2: Seed names used for Novelty Detection.

Sink type	Seed names
Integrity:	
Command injection	execute, command
Code injection	eval, execute, compile, render, callback, function, fn
Reflected XSS	sent, content
Path traversal	file, directory, path, cwd, source, input
Confidentiality:	
Clear-text logging	authkey, password, passcode, passphrase

For each sink kind, we train an OC-SVM model with examples of expected "seed" names, as shown in Table 2, which we select based on our understanding of the domain. For the integrity queries these names indicate data that we would expect to flow to the specific sink, while for the confidentiality query they indicate sensitive data that is *not* expected to flow to the sink. As the set of names is small (less than ten examples per sink kind), obtaining this training data imposes relatively little effort. To train the model, we embed the natural language information using the pre-trained embedding e. Once trained, we let the model predict whether a previously unseen flow is unexpected, again embedding the natural language information using e.

3.4.4 Approach 4: Codex Language Model. Motivated by the impressive results obtained with large language models like Codex [14], for different software engineering tasks [6, 27, 29, 30], we also implement a Codex-based approach. To query the language model, we design a prompt that provides information about the flow and ends with a comment that we ask the model to complete. Following a few-shot approach [10], the prompt includes ten examples of the task we want the model to perform and ends with an unfinished eleventh example for it to complete.

For the integrity queries, the prompt consists of the following:

- The signature of the API function and, if available, the doc comment associated with it.
- A JavaScript comment with the sentence "In the above function f, the parameter p flows into the s sink (e), which is c", where
  - f is the name of the function,
  - p is the name of the formal parameter,
  - s is an abbreviated name of the query, e.g., "CommandInjection",
  - e is a brief explanation of the query, e.g., "uncontrolled data used in a path expression", and
  - c is either "expected" or "unexpected".

For the confidentiality query, the prompt consists of:

- A stub function f calling console.log on its parameter p.
- A JavaScript comment with the sentence "In the above function f, the parameter p is being logged, which likely exposes c data", where c is either "sensitive" or "insensitive".

The word in c is provided for the ten few-shot examples but left undefined for the eleventh example. The model then completes the prompt by predicting the missing word. For the integrity queries, we check whether "expected" or "unexpected" is the more likely completion according to the model, and report flows to the developers only when the model predicts them as unexpected. Likewise, for the confidentiality query, we report a flow to developers only when the model considers completing the prompt with "sensitive data" as more likely than completing it with "insensitive data".

4 EVALUATION

To assess the efficacy and practicality of our approach, we pose ourselves the following research questions:

RQ1 How effective is Fluffy at identifying unexpected flows?
RQ2 How effective is Fluffy at finding vulnerabilities?
RQ3 What is the trade-off between labeling effort and prediction accuracy?
RQ4 How well does Fluffy scale to large code bases?

We now describe our evaluation setup, methodology, and results, and discuss how they answer our research questions.

4.1 Experimental Setup

4.1.1 Data Collection. We collect three datasets for our evaluation. The first two (*param-sink flows* and *logging flows*) are compiled by running special-purpose CodeQL queries across all JavaScript/TypeScript projects on LGTM.com (around 250,000 at the time of writing), while the third one is derived from *SecBench.js* [7], a corpus of real-world vulnerabilities in server-side JavaScript.

Param-sink flows. The first dataset concerns unhygienic APIs. We want to collect examples of flows from API parameters to known sinks. For purposes of this dataset, an API parameter is a parameter of a function exported by an npm package, or (by a slight abuse of terminology) a property of such a parameter. Our sinks are collected from four CodeQL security analyses that identify four common types of integrity violations: code injection, command injection, reflected XSS, and path traversal.

For each kind of integrity violation, we write a custom CodeQL query using the standard taint-tracking framework. The queries collect flow tuples of the form (p, s), where p is an API parameter and s is one of the four types of sinks mentioned, such that taint flows from p into a sink of type s, or the special type *None* if no flow from p to any known sink is found.[11] Apart from the name of the API parameter, we also collect its doc comment and the doc comment of the enclosing function, if any. Finally, we filter out flows where the parameter name has less than two characters, since single-character names are unlikely to convey much semantic information to either a human or a model.

Overall, the query found 3,245,860 flows on 61,123 projects, where 3,228,034 are flows to *None*; 1,123 to code-injection sinks; 1,498 to command-injection sinks; 70 to reflected-XSS sinks; and 15,135 to path-traversal sinks. Note the grave imbalance of sink types, which is a consequence of the selection of projects on

---

[11]Note this means that we can have two flows (p, s) and (p, s') for the same parameter p with different sink types s and s', but only if s and s' are not *None*.

LGTM.com, and hence not under our control. Filtering out single-character names removed 233,091 flows, i.e., 6.7% of all flows before filtering.

Logging flows. The second dataset concerns clear-text logging. We again write a custom CodeQL taint-tracking query, this time looking for flow from a variable or property v into a call to a logging function, as determined by existing API modeling in the CodeQL standard libraries. Apart from the name of v, we also record whether the clear-text logging analysis shipping with CodeQL considers v to be potentially sensitive. As above, we filter out single-character names, which account for 8.9% of all flows, yielding 4,535,851 flows in 112,765 projects.

SecBench.js. For our final dataset, we examine all code-injection, command-injection, and path-injection vulnerabilities in the SecBench.js corpus to find the ones caused by unhygienic APIs, of which there are 33, 101, and 1, respectively after filtering.[12]

4.1.2 Non-Neural Baselines.
In addition to the four neural network-based instantiations of Fluffy, we also consider two non-neural baseline approaches to compare against. The first baseline, called Frequency Counting, is a statistical approach based on determining the frequency with which a given parameter name is observed to flow into a particular sink type in our param-sink flows dataset. If this frequency is below a given threshold, we consider this sink type to be unexpected for the parameter. Note that no semantic reasoning about names is involved and only exact name matches are considered. Due to the composition of the datasets this approach only works for the integrity violations since the logging-flows dataset considers only a single sink type.

The second baseline, called Regular Expressions, is for the logging flows. It is an existing CodeQL query (Section 2) that uses regular expressions to flag likely problematic flows, first applying two regular expressions to identify names that may indicate sensitive content such as passcode and authkey.[13] Names matching either of these regular expressions are then checked against a third regular expression that filters out names suggesting that the value has been encrypted or hashed.[14]

We also considered a direct comparison with CodeQL queries that search for confidentiality violations and that existed prior to this work (Section 2). However, these queries do not answer the same question as Fluffy. The preexisting queries find end-to-end flows from sources of tainted data to sensitive sinks, whereas Fluffy works on flows from API parameters to sensitive sinks. The preexisting queries would not consider these API parameters to be sources (and conversely, Fluffy does not consider flows from non-parameter sources), so the two cannot be compared directly.

---

[12]We exclude one code-injection vulnerability and two command-injection vulnerabilities from consideration since the affected npm package is no longer available, making it impossible to determine the parameters involved. We also exclude one further command injection vulnerability for which details are no longer available since the corresponding advisory has been withdrawn.

[13]https://github.com/github/codeql/blob/499f20f6e8a3a91e394c30e05a340fe10b9ecec7/javascript/ql/lib/semmle/javascript/security/internal/SensitiveDataHeuristics.qll#L69

[14]https://github.com/github/codeql/blob/499f20f6e8a3a91e394c30e05a340fe10b9ecec7/javascript/ql/lib/semmle/javascript/security/internal/SensitiveDataHeuristics.qll#L104

4.1.3 Data Labeling.
To obtain ground truth for our evaluation, we label a subset of the collected data to indicate whether the flow may cause or contribute to a vulnerability.

For the param-sink dataset, the authors manually label 1,058 flows. To check the reliability of our labeling, we send out a survey to four program analysis engineers with experience in security-related analyses. The survey asks the analysis experts to label 30 randomly selected flows each, based on the parameter name, function name, package name, sink, package description, parameter documentation, and function documentation. Specifically, the question we ask is: "As a client of the package, would you expect data passed into this parameter to flow into the sink in question without further sanitization?" Using Krippendorff's alpha to compute the inter-rater agreement, the labels we assign and those given by the analysis experts have an agreement score of 0.74. That is, the analysis experts by and large agree with the labels in our ground truth.

The total set of labeled flows consists of two overlapping subsets. The first is the *random set*, which consists of 272 randomly selected flows. While this set is unbiased, it does not contain many unexpected flows since most flows observed in real-world code are unproblematic. Hence, we extend the random set into a second set, which we call the *balanced set*, additionally including flows selected from results of the Frequency Counting approach.

For the logging dataset, we manually label a subset of 340 flows. Table 4 shows the number of total and unexpected flows according to our labeling for each sink type.

Note that our manual labeling shows that simply flagging all flows as unexpected would result in many false positives, demonstrating the importance of our approach.

Finally, for the SecBench.js dataset, we review all flows, and identify one code-injection vulnerability and seven command-injection vulnerabilities that, despite having been reported as a vulnerability, are expected according to our judgment. Full details about these vulnerabilities are included in the supplementary materials. To give just one example, SecBench.js includes CVE-2020-7784, which identifies the fact that the parameter command of the function exec exported by the npm package ts-process-promises is interpreted as a shell command. We consider this flow obvious from context and hence not a genuine vulnerability. None of the vulnerabilities have been fixed, and in one case the advisory has been withdrawn, suggesting that the authors of the packages involved do not consider them to be a problem. All remaining 127 vulnerabilities are labeled as unexpected by us.

4.1.4 Model Hyperparameters.
All our models operate on vectors with 768 dimensions, corresponding to the output dimension of the VarCLR embedding [15]. The only exception to this is Codex, which we access indirectly through an HTTP API as explained in Section 3.4.4. In the Sink Prediction and Binary Classification approaches, the hidden layer dimensions are 500 and 250, respectively. We use the Adam optimizer with a learning rate of 0.001. For Sink Prediction, the training batch size is 256, and we stop the training early when validation loss does not decrease in two epochs. We reserve 10% of the training data for use as a validation set. For Binary Classification, the training batch size is 32, and the model stops when validation loss does not decrease in 50 epochs. We use

Table 3: Comparison of the different approaches.

Approach	Training data	Number of models	Needs threshold	Computation of F1 score
Sink Prediction	Param-sink flows	One	Yes	From precision-recall curve
Novelty Detection	Seed names	One per sink type	Yes	From precision-recall curve
Binary Classification	Balanced set	One per sink type	No	From k-fold cross-validation
Codex	Balanced set	One	No	Direct
Frequency	Param-sink flows	One	Yes	From precision-recall curve

Table 4: Ground truth labels.

	Random Set		Balanced Set	
Integrity:	Unexpected	Total	Unexpected	Total
Code injection	16	27	113	340
Command injection	15	29	144	168
Reflected XSS	19	28	29	46
Path traversal	8	188	105	504
Confidentiality:			Unexpected	Total
Logging sensitive data			245	340

Radial Basis Function (RBF) as the kernel for the OC-SVM in the Novelty Detection approach. The hyperparameters gamma and nu are 0.05 and 0.01, respectively. For the Codex language model, we set the temperature, frequency penalty, and presence penalty to zero.

4.2 RQ1: Identifying Unexpected Flows

In this section, we explain the evaluation process for each approach and then discuss the results for param-sink and logging flows. We compare the approaches to each other, as well as to the two baselines (Frequency Counting and Regular Expressions) discussed above.

4.2.1 Evaluation Process. We investigate how well each approach can classify unexpected flows, using the F1 score as our main metric. Table 3 compares the characteristics of the four neural approaches and the Frequency Counting baseline. All approaches requiring a threshold are evaluated by plotting the precision-recall curve (PR curve) showing the relationship between precision and recall for different thresholds, and recording the best F-1 score obtainable. We compute precision and recall w.r.t. source-sink pairs.

For Sink Prediction, we train the model with all flows that have not been labeled, using of these flows for 90% training and 10% for validation. The labeled ground truth serves as the test set. Since the model outputs probabilities, a threshold is required to make a prediction.

For Novelty Detection, we train one OC-SVM for each sink type (i.e., four for param-sink flows, and one for logging flows) using the corresponding seed names. We again use the ground truth as our test set, which may contain flows involving the seed names as these were chosen independently. The model outputs a score for each flow, representing the distance between the flow and the boundary of the OC-SVM, which again requires a threshold to derive a prediction.

For Binary Classification, we also train one model per sink based on the relevant flows from the balanced set. No threshold is needed since the model directly classifies flows. We evaluate using 5-fold cross-validation with five different 80/20 splits, reporting average precision, recall, and F1-score across the five folds for each model.

For Codex, for each flow to evaluate, we randomly draw ten examples of the same sink from the balanced set to put in the prompt. We ensure that the flow we ask it to evaluate is not among the ten examples embedded in the prompt. As this model directly classifies whether the flow is expected, we again do not need a threshold and can directly compute the precision, recall, and F1-score based on the predictions.

Finally, Frequency Counting is similar to Sink Prediction in that it is trained on all param-sink flows and requires a threshold.

4.2.2 Param-Sink Flows. For param-sink flows, all four approaches are evaluated on the random set and the balanced set, with Frequency Counting as the baseline. The summary results for the random set are shown in Table 5, with the best scores for each sink type highlighted in bold. We observe that Fluffy's approaches outperform the baseline for all sinks, with Novelty Detection as the overall winner followed by Binary Classification. The approaches perform especially well on command injection and reflected XSS with F1 scores of 0.9–1.0, and around 0.8 for code injection. Effectiveness on path traversal, on the other hand, is significantly worse with widely spread scores peaking at 0.45. This is likely due to the random set having only eight examples of unexpected path-traversal flow.

The result on the balanced set are shown in Table 6, with PR curves in Figure 2. In this setup, Binary Classification performs best, surpassing Novelty Detection by a significant margin on code injection and path traversal. This could be due to the balanced set containing more diverse names, giving Binary Classification an opportunity to learn from the dataset, while Novelty Detection relies on a fixed set of seed names. For the path traversal sink, all approaches perform better in the balanced set than in the random set, which is likely due to having many more (105) unexpected cases. Frequency Counting performs very well for command injection, but this has to be interpreted with caution since the balanced set was partly constructed from results of this very approach. Another interesting finding is that Sink Prediction performs much worse on code injection flows in the balanced set than in the random set, as the F1-score drops from 0.82 to 0.53. This is due to a decline in precision: while it is able to capture most of the unexpected flows (108), it also includes many false positives (194). The rest of the results are similar to the random set.
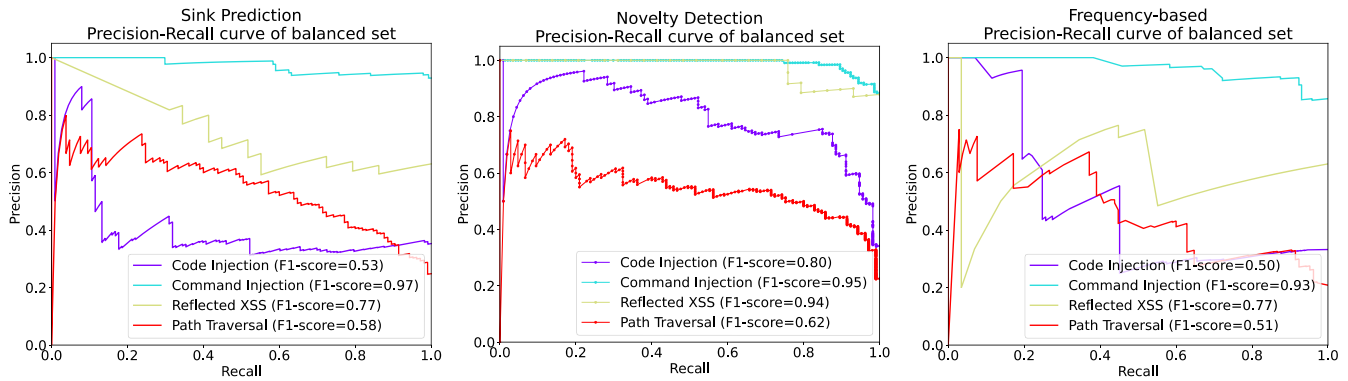
Table 5: Effectiveness of Fluffy on the random set of param-sink flows.

| | Random Set | | | | | | | | | | | |
| | Code injection | | | Command injection | | | Reflected XSS | | | Path traversal | | |
Approach	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Precision	Recall	F1-Score
Sink Pred.	0.78	0.88	0.82	1.00	1.00	**1.00**	0.68	1.00	0.81	0.18	0.75	0.29
Novelty Det.	0.82	0.88	**0.85**	1.00	1.00	**1.00**	0.86	1.00	**0.93**	0.36	0.63	**0.45**
Binary Class.	0.83	0.82	0.81	1.00	1.00	**1.00**	0.95	0.90	0.91	0.33	0.30	0.31
Codex	0.65	0.69	0.67	0.63	1.00	0.77	0.94	0.89	0.92	0.08	0.13	0.10
Frequency	0.59	1.00	0.74	0.91	0.67	0.77	0.68	1.00	0.81	0.22	0.25	0.24

Table 6: Effectiveness of Fluffy on the balanced set of param-sink flows.

| | Balanced Set | | | | | | | | | | | |
| | Code injection | | | Command injection | | | Reflected XSS | | | Path traversal | | |
Approach	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Precision	Recall	F1-Score
Sink Pred.	0.36	0.98	0.53	0.94	0.99	**0.97**	0.63	1.00	0.77	0.53	0.64	0.58
Novelty Det.	0.74	0.88	0.80	0.92	0.97	0.95	0.88	1.00	**0.94**	0.50	0.80	0.62
Binary Class.	0.90	0.88	**0.88**	0.97	0.97	**0.97**	0.96	0.94	**0.94**	0.81	0.73	**0.76**
Codex	0.76	0.76	0.76	0.90	0.97	0.94	0.90	0.93	0.92	0.62	0.42	0.50
Frequency	0.33	1.00	0.50	0.93	0.92	0.93	0.63	1.00	0.77	0.42	0.63	0.51

Figure 2: PR curves on the balanced set for Sink Prediction, Novelty Detection, and Frequency Counting (left to right).

In summary, Fluffy proves to be a significant improvement over the baseline Frequency Counting approach. The two best approaches are Novelty Detection and Binary Classification, while Sink Prediction does not work as well. Our interpretation is that the observed likelihood of flow from a source to a sink does not help understand how unexpected such a flow is. Codex also does not perform as well as the others, possibly because the randomly chosen samples embedded in the prompt do not provide enough information.

Path traversal flows are the hardest to classify for all the approaches. This seems to be because the names involved are very diverse, including file operations (such as `move` and `rm`), file types,

e.g., `png` and `mp4`), and other names that could represent certain kinds of files (such as `log` and `config`). It is hard for the models to learn all these concepts, reducing their effectiveness. The names for other sink types are not as diverse, as shown, for example, by the excellent score of the Novelty Detection model for command-injection flows in spite of only having two seed names.

4.2.3 Logging Flows. The results of evaluating Fluffy on the logging flows are shown in Table 7, with PR curves in Figure 3. Overall, Binary Classification performs best, followed by Novelty Detection and Regular Expressions. The latter gives the highest recall,
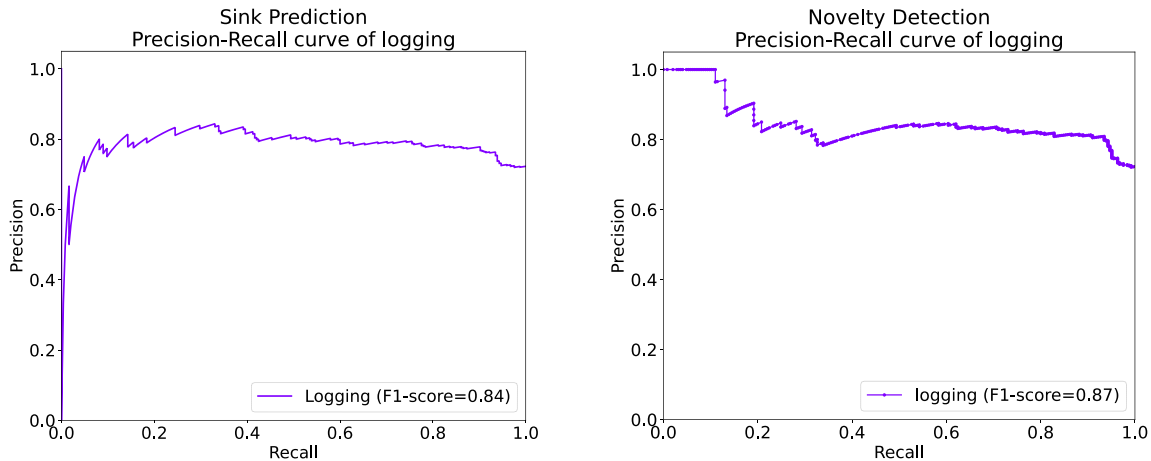
Figure 3: PR curves on the logging flows for Sink Prediction and Novelty Detection (left to right).

Table 7: Effectiveness of Fluffy on the logging flows.

Approach	Logging Flows Dataset		
	Precision	Recall	F1-Score
Sink Prediction	0.76	0.93	0.84
Novelty Detection	0.81	0.93	0.87
Binary Classification	**0.90**	0.94	**0.92**
Codex	0.78	0.96	0.86
Regexps	0.79	**0.97**	0.87

Table 8: Effectiveness of Fluffy on the SecBench.js dataset.

Approach	SecBench.js Dataset					
	Code injection			Command injection		
	Precision	Recall	F1	Precision	Recall	F1
Sink Pred.	0.97	**0.88**	0.92	1.00	0.90	0.95
Novelty Det.	0.96	0.75	0.84	0.99	**0.98**	0.98
Binary Class.	0.97	0.69	0.80	1.00	0.97	0.98
Codex	0.96	0.69	0.80	0.94	0.97	0.95
Frequency	0.94	0.47	0.63	1.00	0.65	0.79

while the latter have better precision. This suggests that regular expressions are fairly effective, but they can be difficult to write and maintain while our learning-based approach requires very little human intervention.

As a case in point, we discover that Fluffy was able to correctly flag clear-text logging of a variable named `passcode` as unexpected, while CodeQL does not. This was surprising since the name matches one of the regular expressions for finding sensitive data. However, it turns out that it *also* matches the regular expression used to filter out encrypted sources. We report this to the CodeQL maintainers, who acknowledge it as a bug.

4.3 RQ2: Real-World Vulnerabilities

The results for RQ1 provide some evidence that Fluffy is effective at spotting unexpected flows, but this does not yet prove that it can be used to find real-world vulnerabilities. To address this question, we evaluate Fluffy on the SecBench.js dataset, and also report some unexpected flows flagged in the param-sink dataset to the maintainers of the corresponding projects to get their input on whether the flows are indeed problematic.

4.3.1 Fluffy on Past Vulnerabilities. For evaluating on SecBench.js, we use mostly the same evaluation methodology as shown Section 4.2.2, except for the threshold-based approaches where we use the threshold that gave the best F-1 score on the balanced set in Section 4.2.2. This allows us to assess if the threshold we set can generalize to a different dataset. We omit the path-traversal sink type, since there is only one flow for this type in the dataset.

We use recall as our main metric (Table 8), since we are mainly interested in whether Fluffy would have been able to flag these known vulnerabilities and since precision is already evaluated in RQ1. Somewhat surprisingly, Sink Prediction has the best recall for code injection. This turns out to be because the best threshold for this approach on the balanced set is very high, favoring the SecBench.js dataset where all but one code injection flows are unexpected. As before, Novelty Detection does very well for command injection, closely followed by Binary Classification.

4.3.2 Fluffy on Previously Unknown Vulnerabilities. In this section, we examine how effective Fluffy is at finding new vulnerabilities. Inspecting the flows flagged by Fluffy on the param-sink dataset, we find and report a flow from a parameter called `name` of a function called `locale` in the popular `moment` library to code-injection sink, as mentioned in the introduction. The developers have acknowledged and fixed this vulnerability, now known as CVE-2022-24785. We furthermore create 16 pull requests on GitHub to fix other unexpected flows that seemed undesirable but less critical. At the time of writing, seven of them have been merged, eight are still open without feedback from developers, and one pull request has been closed, with an alternative fix implemented.
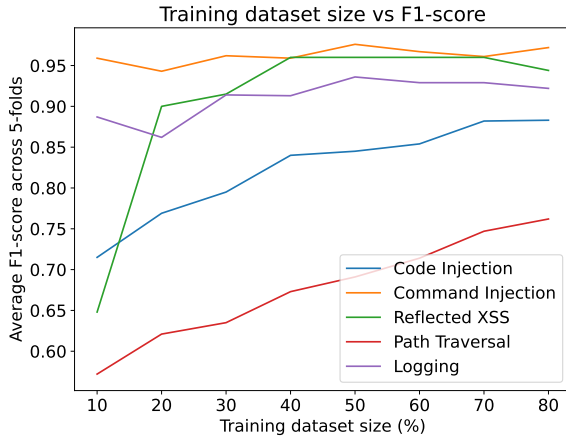
Figure 4: Training set size versus model effectiveness.

Taken together, these results indicate that Fluffy is indeed able to find real-world vulnerabilities.

4.4 RQ3: Human Efforts vs. Model Effectiveness

From Table 5 and Table 6, we can observe that Binary Classification is one of the approaches with the best effectiveness. However, training the Binary Classification model requires a lot of manual labeling, requiring human effort, which is a scarce resource. Therefore, we would like to know how many labels are needed to achieve a good effectiveness for Binary Classification.

In this experiment, we evaluate Binary Classification with different training set sizes on the balanced set. Keeping the test set constant at 20% of the flows for a single sink type, we increase the training-set size from 10% to 80% in 10% steps. The results in Figure 4 show that the F1 score generally increases with training-set size, but for command injection even 10% suffices. Binary Classification with 10% training data is, however, outperformed by Novelty Detection with even less training data (viz, the seed names in Table 2), showing that the former is a good choice if enough labeled data is available, while the latter is more parsimonious.

4.5 RQ4: Scalability

One potential concern with our approach is the performance of the underlying CodeQL query which has a very large set of sources. However, the query for the param-sink flows dataset finished within LGTM.com's default timeout of four hours on all but 73 projects (> 99.97%), and the one for the logging flows on all but 49 (> 99.98%). This suggests that while Fluffy might not scale to truly massive projects, the CodeQL component is not usually a bottleneck.

For Binary Classification, our 5-fold cross-validation takes less than 5 minutes to train and around 1.5 seconds to evaluate each model. The Sink Prediction model takes one and a half hour to train and around 20 seconds to evaluate. The Novelty Detection model takes less than 3 seconds to train and less than 3 seconds to evaluate. For Codex, the model is accessed via a rate-limited

REST API, providing one completion in 1.8 seconds, on average. This indicates that, except for Codex, the neural components of Fluffy scale well, as training takes at most a couple of hours, and the model can classify hundreds of flows within seconds. We conduct our experiments on a server with 48 Intel Xeon CPU cores clocked at 2.2GHz, 250GB of RAM, and one NVIDIA Tesla V100 GPU. The Binary Classification and Sink Prediction model use the GPU, whereas Novelty Detection and Codex use the CPU.

4.6 Threats to Validity

While the results shown above are promising, there are several threats to validity to keep in mind. First, we only evaluate on five vulnerability types, and our results may not generalize to other types. The corpus of code bases we consider is prescribed by the selection of projects hosted on LGTM.com, potentially introducing bias. Similarly, our evaluation is limited to JavaScript, and we cannot say for certain that our approach would work for other languages. Second, our evaluation relies on manual labeling, which is subject to human error. We tried to mitigate this by involving independent labelers, but they only labeled a small fraction of the data, leaving open the possibility of bias in the remainder. Third, our selection of seed names for Novelty Detection is based on human judgment, again introducing a potential source of bias. Finally, the balanced dataset contains more unexpected flows than we expect to actually occur in the wild, which may lead to overfitting.

5 RELATED WORK

Our work fits in the general category of neural software analysis [45], which is characterized by fuzziness of available information and lack of well-defined correctness criteria, but also a large amount of training data. Previous work in this area ranges tackles problems as diverse as bug detection [3, 47, 52], automated program repair [16, 34, 35, 39, 54], code completion [4, 14, 31], probabilistic type inference [2, 24, 40, 46], call graph pruning [33, 51], emulating a dynamic taint analysis [50], and learning inference rules for static analyzers from data [8]. There has also been a lot of work on using machine learning techniques to detect vulnerabilities [20, 23, 36, 37], though such approaches are not yet precise enough in practice [13].

Unlike many of these works, however, our goal is not to produce a fully neural end-to-end vulnerability detector. Instead, we enhance an existing static analysis tool (CodeQL) with the help of machine learning techniques. In this regard we stand in the tradition of systems like Merlin [38], SuSi [48], Seldon [17], and USpec [19], which use machine learning to improve taint analyses by identifying additional sources and sinks or inferring aliasing behavior. However, those systems and our work have complementary goals. While the above-mentioned approaches can be used to identify APIs that produce tainted data directly, Fluffy finds parameters into which an unsuspecting user might pass tainted data, that is, the tainted data does not originate in the parameter, but somewhere else. Moreover, since we use natural-language information we can do this even if we have never seen an example of tainted data actually being

passed to this parameter, thus allowing us to spot potential, not yet exploitable vulnerabilities.[15]

We are, of course, not the first to realize and exploit the "bi-modal" structure of source code [1, 12, 26]. For example, identifier names in particular have previously been used for bug detection [43, 47], bug injection [42], type refinement [18], and type prediction [40]. However, we believe we are the first to systematically use natural language information for the purpose of improving taint analyses.

One could also view our approach as using neural techniques to identify false positives, though in our case the underlying taint analysis is deliberately imprecise and not expected to stand on its own. A recent contribution in this direction is Kharkar et al.'s work [30], which explores both a feature-based and a neural approach to identifying false positives produced by the Infer static analyzer [11]. Their ideas relate to ours in that their neural models implicitly use both code and natural language information. Our work differs by specifically focusing on taint analysis and by presenting other formulations of the learning task.

Another line of related work is on predicting heuristics used in static analyses. For example, Jeong et al. learn heuristics for selecting methods likely to benefit from certain depth-levels of context-sensitive points-to analysis [28]. Others predict which loops to unroll without missing any bugs to report [25] and learn state-selection heuristics to reduce the cost of path-sensitivity [32]. Such work ultimately aims at improving the efficiency of static analyses, whereas Fluffy's goal is to ensure that the warnings reported by the analysis are indeed relevant to developers.

The effectiveness of our approach depends crucially on the underlying word embeddings. Common methods to train such embeddings include FastText [9], word2vec [41] and GloVe [44], but these are meant for natural language. We work with variables names, which have substantially different characteristics [53]. Hence we use VarCLR [15], which specifically targets identifier names, and has been shown to outperform other approaches in this domain.

6 CONCLUSIONS

This paper presents Fluffy, a general framework for bimodal taint analysis combining a static taint analysis identifying candidate flows with a neural component identifying the problematic ones. We instantiate this framework on top of the CodeQL JavaScript analysis with four different neural components (Binary Classification, Sink Prediction, Novelty Detection, and Codex Language Model), and evaluate the resulting system on a large corpus of real-world JavaScript code bases as well as a collection of known vulnerabilities. Our findings show that Fluffy performs well in practice, identifying potentially problematic flows with high accuracy and flagging known vulnerabilities with high recall.

For future work, we would like to investigate a tighter integration between the static and the neural components, and also explore the broader applicability of this technique to other languages.

---

[15]While some features used by SuSi are based on natural language information, these are manually engineered and rely on very simple string matching, such as determining whether a method name starts with "get".

REFERENCES

[1] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 81.

[2] Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. 2020. Typilus: Neural Type Hints. In *PLDI*.

[3] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings.* https://openreview.net/forum?id=BJOFETxR-

[4] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating Sequences from Structured Representations of Code. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019.* https://openreview.net/forum?id=H1gKYo09tX

[5] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-oriented Queries on Relational Data. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy (LIPIcs, Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2:1–2:25. https://doi.org/10.4230/LIPIcs.ECOOP.2016.2

[6] Patrick Bareiß, Beatriz Souza, Marcelo d'Amorim, and Michael Pradel. 2022. Code Generation Tools (Almost) for Free? A Study of Few-Shot, Pre-Trained Language Models on Code. *CoRR* abs/2206.01335 (2022). https://doi.org/10.48550/arXiv.2206.01335 arXiv:2206.01335

[7] Masudul Bhuiyan, Adithya Srinivas, Nikos Vasilakis, Michael Pradel, and Cristian-Alexandru Staicu. 2022. SecBench.js: An Executable Security Benchmark Suite for Server-Side JavaScript. https://github.com/cristianstaicu/SecBench.js

[8] Pavol Bielik, Veselin Raychev, and Martin T. Vechev. 2017. Learning a Static Analyzer from Data. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10426)*, Rupak Majumdar and Viktor Kuncak (Eds.). Springer, 233–253. https://doi.org/10.1007/978-3-319-63387-9_12

[9] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching Word Vectors with Subword Information. *TACL* 5 (2017), 135–146. https://transacl.org/ojs/index.php/tacl/article/view/999

[10] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html

[11] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving fast with software verification. In *NASA Formal Methods Symposium*. Springer, 3–11.

[12] Casey Casalnuovo, Earl T Barr, Santanu Kumar Dash, Prem Devanbu, and Emily Morgan. 2020. A theory of dual channel constraints. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE, 25–28.

[13] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2020. Deep Learning based Vulnerability Detection: Are We There Yet? *CoRR* abs/2009.07235 (2020). arXiv:2009.07235 https://arxiv.org/abs/2009.07235

[14] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario

Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

[15] 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 *ICSE*.

[16] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair. *IEEE TSE* (2019).

[17] Victor Chibotaru, Benjamin Bichsel, Veselin Raychev, and Martin T. Vechev. 2019. Scalable Taint Specification Inference with Big Code. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 760–774. https://doi.org/10.1145/3314221.3314648

[18] Santanu Kumar Dash, Miltiadis Allamanis, and Earl T. Barr. 2018. RefiNym: Using Names to Refine Types. In *ESEC/FSE*.

[19] Jan Eberhardt, Samuel Steffen, Veselin Raychev, and Martin T. Vechev. 2019. Unsupervised learning of API aliasing specifications. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019.* 745–759. https://doi.org/10.1145/3314221.3314640

[20] Michael Fu and Chakkrit Tantithamthavorn. 2022. LineVul: A Transformer-based Line-Level Vulnerability Prediction. In *MSR*.

[21] Tiferet Gazit. 2022. Leveraging machine learning to find security vulnerabilities. https://github.blog/2022-02-17-leveraging-machine-learning-find-security-vulnerabilities/

[22] GitHub. 2022. CodeQL. https://codeql.github.com/

[23] Jacob A. Harer, Louis Y. Kim, Rebecca L. Russell, Onur Ozdemir, Leonard R. Kosta, Akshay Rangamani, Lei H. Hamilton, Gabriel I. Centeno, Jonathan R. Key, Paul M. Ellingwood, Marc W. McConley, Jeffrey M. Opper, Sang Peter Chin, and Tomo Lazovich. 2018. Automated software vulnerability detection with machine learning. *CoRR* abs/1803.04497 (2018). arXiv:1803.04497 http://arxiv.org/abs/1803.04497

[24] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. 2018. Deep learning type inference. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 152–162. https://doi.org/10.1145/3236024.3236051

[25] Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. 2017. Machine-learning-guided selectively unsound static analysis. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE / ACM, 519–529. https://doi.org/10.1109/ICSE.2017.54

[26] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar T. Devanbu. 2012. On the naturalness of software. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland.* 837–847.

[27] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2022. Jigsaw: Large Language Models meet Program Synthesis. In *ICSE*.

[28] Sehun Jeong, Minseok Jeon, Sung Deok Cha, and Hakjoo Oh. 2017. Data-driven context-sensitivity for points-to analysis. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 100:1–100:28. https://doi.org/10.1145/3133924

[29] Harshit Joshi, José Cambronero, Sumit Gulwani, Vu Le, Ivan Radicek, and Gust Verbruggen. 2022. Repair Is Nearly Generation: Multilingual Program Repair with LLMs. https://doi.org/10.48550/ARXIV.2208.11640

[30] Anant Kharkar, Roshanak Zilouchian Moghaddam, Matthew Jin, Xiaoyu Liu, Xin Shi, Colin Clement, and Neel Sundaresan. 2022. Learning to Reduce False Positives in Analytic Bug Detectors. In *ICSE*.

[31] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2021. Code Prediction by Feeding Trees to Transformers. In *ICSE*.

[32] Yoonseok Ko and Hakjoo Oh. 2023. Learning to Boost Disjunctive Static Bug-Finders. In *ICSE*.

[33] Thanh Le-Cong, Hong Jin Kang, Truong Giang Nguyen, Stefanus Agus Haryono, David Lo, Xuan-Bach D. Le, and Quyet Thang Huynh. 2022. AutoPruner: Transformer-Based Call Graph Pruning. In *ESEC/FSE*.

[34] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65. https://doi.org/10.1145/3318162

[35] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. DLFix: Context-based Code Transformation Learning for Automated Program Repair. In *ICSE*.

[36] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. Vulnerability detection with fine-grained interpretations. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 292–303. https://doi.org/10.1145/3468264.3468597

[37] Zhen Li, Shouhuai Xu Deqing Zou and, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. In *NDSS*.

[38] V. Benjamin Livshits, Aditya V. Nori, Sriram K. Rajamani, and Anindya Banerjee. 2009. Merlin: specification inference for explicit information flow problems. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009.* 75–86.

[39] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: combining context-aware neural translation models using ensemble for program repair. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, Sarfraz Khurshid and Corina S. Pasareanu (Eds.). ACM, 101–114. https://doi.org/10.1145/3395363.3397369

[40] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. 2019. NL2Type: Inferring JavaScript function types from natural language information. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019.* 304–315. https://doi.org/10.1109/ICSE.2019.00045

[41] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. *CoRR* abs/1301.3781 (2013). http://arxiv.org/abs/1301.3781

[42] Jibesh Patra and Michael Pradel. 2021. Semantic bug seeding: a learning-based approach for creating realistic bugs. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 906–918. https://doi.org/10.1145/3468264.3468623

[43] Jibesh Patra and Michael Pradel. 2022. Nalin: Learning from Runtime Behavior to Find Name-Value Inconsistencies in Jupyter Notebooks. In *ICSE*.

[44] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. GloVe: Global Vectors for Word Representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, Alessandro Moschitti, Bo Pang, and Walter Daelemans (Eds.). ACL, 1532–1543. https://doi.org/10.3115/v1/d14-1162

[45] Michael Pradel and Satish Chandra. 2022. Neural software analysis. *Commun. ACM* 65, 1 (2022), 86–96. https://doi.org/10.1145/3460348

[46] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. 2020. TypeWriter: Neural Type Prediction with Search-based Validation. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020.* 209–220. https://doi.org/10.1145/3368089.3409715

[47] Michael Pradel and Koushik Sen. 2018. DeepBugs: A learning approach to name-based bug detection. *PACMPL* 2, OOPSLA (2018), 147:1–147:25. https://doi.org/10.1145/3276517

[48] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. 2014. A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014.* The Internet Society. https://www.ndss-symposium.org/ndss2014/machine-learning-approach-classifying-and-categorizing-android-sources-and-sinks

[49] Bernhard Schölkopf, John C Platt, John Shawe-Taylor, Alex J Smola, and Robert C Williamson. 2001. Estimating the support of a high-dimensional distribution. *Neural computation* 13, 7 (2001), 1443–1471.

[50] Dongdong She, Yizheng Chen, Baishakhi Ray, and Suman Jana. 2019. Neutaint: Efficient Dynamic Taint Analysis with Neural Networks. *CoRR* abs/1907.03756 (2019). arXiv:1907.03756 http://arxiv.org/abs/1907.03756

[51] Akshay Utture, Shuyang Liu, Christian Gram Kalhauge, and Jens Palsberg. 2022. Striking a Balance: Pruning False-Positives from Static Call Graphs. In *ICSE*.

[52] Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. 2019. Neural Program Repair by Jointly Learning to Localize and Repair. In *ICLR*.

[53] Yaza Wainakh, Moiz Rauf, and Michael Pradel. 2021. IdBench: Evaluating Semantic Representations of Identifier Names in Source Code. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021.* IEEE, 562–573. https://doi.org/10.1109/ICSE43902.2021.00059

[54] He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural Program Repair with Execution-based Backpropagation. In *ICSE*.