Software Lab
Institute of Software Engineering
University of Stuttgart
Universitätsstraße 38, 70569 Stuttgart

Master Thesis

Bimodal Taint Analysis for Detecting Unusual Parameter-Sink Flows

Yiu Wai Chow

Course of study:	Computer Science
Examiner:	Prof. Dr. Michael Pradel
Supervisor:	Prof. Dr. Michael Pradel, Dr. Max Schaefer
Started:	May 15, 2022
Completed:	November 15, 2022

University of Stuttgart
Germany

SOLA
SoftwareLab

Abstract

Finding vulnerabilities is a crucial activity, and automated techniques for this purpose are in high demand. For example, the Node Package Manager (*npm*) offers a massive amount of software packages, which get installed and used by millions of developers each day. Because of the dense network of dependencies between npm packages, vulnerabilities in individual packages may easily affect a wide range of software.

Taint analysis is a powerful tool to detect such vulnerabilities. However, it is challenging to clearly define a problematic flow. A possible way to identify problematic flows is to incorporate natural language information like code convention and informal knowledge into the analysis. For example, a user might not find it surprising that a parameter named `cmd` of a function named `execCommand` is open to command injection. Thus this flow is likely unproblematic as the user will not pass untrusted data to `cmd`. In contrast, a user might not expect a parameter named `value` of a function named `staticSetConfig` to be vulnerable to command injection. Thus this flow is likely problematic as the user might pass untrusted data to `value`, since the natural language information from the parameter and function name suggests a different security context.

To effectively exploit the implicit information in code, we introduce a bimodal taint analysis tool, Fluffy. The first modality is *code*: Fluffy uses a mining analysis implemented in CodeQL to find examples of flows from parameters to vulnerable sinks. The second modality is *natural language*: Fluffy uses a machine learning model that, based on a corpus of such examples, learns how to distinguish unexpected flows from expected flows using natural language information. We instantiate four neural models, offering different trade-offs between manual efforts required and accuracy of predictions. In our evaluation, Fluffy is able to achieve a F1-score of 0.85 or more on four common vulnerability types. In addition, Fluffy is able to flag eleven previously unknown vulnerabilities in real-life projects, of which six are confirmed.

Contents

1 Introduction

Finding vulnerabilities is a crucial activity, and automated techniques for this purpose are in high demand. The Node Package Manager (*npm*) offers a massive amount of software packages, which get installed and used by millions of developers each day. Because of the dense network of dependencies between npm packages, vulnerabilities in individual packages may easily affect a wide range of software [41].

One automated technique is taint analysis. Taint analysis tracks data flowing from a *source* to a *sink*. This is useful for finding vulnerabilities as users can specify a policy such that a flow violating the policy is reported. For instance, one use case is to track if some user input flows into a template engine like Mustache[1], which might be vulnerable to code injection attacks, if the user input is not properly sanitized.

However, precisely defining sources and sinks is not always easy. As an example, consider a data flow between a client and the libraries it uses, i.e., calling a function in the library and passing some data via a parameter. We could treat the source as any parameters in the library, i.e., assuming all parameters in the library receive unsanitised data. Nonetheless, this is not reasonable and gives us an unmanageably large number of results. Instead, we should make a reasonable worst-case assumption about the source. Instead of assuming all parameters in the library receive unsanitised data, we should only consider sources where non-malicious clients might pass in untrusted data. One way to define this source is to ask whether a typical JavaScript programmer would expect data passed into `p` to flow into a sink of kind `s`. If the answer is yes, then it is reasonable to assume that no untrusted data will be passed into `p` (if it is then that is the client's fault), but if the answer is no then we should flag any flows from `p` to sinks of kind `s`.

As a concrete example, consider a command injection vulnerability in the "ps" package.[2] An API function exported by this package expects an argument called `pid`, which gets embedded unsafely into a shell command. If a client passes an untrusted input to this argument, this would expose the client to a command injection vulnerability. A typical JavaScript programmer would likely not expect `pid` to flow into a command injection sink, since `pid` does not sound like a security-relevant parameter. Thus, this flow should be flagged.

In contrast, let us consider an argument called `expression` from a function named `evalExpression` that is vulnerable to code injection.[3] This potential vulnerability is likely not as interesting to a developer since they would already expect the possibility of code injection based on the parameter

[1] https://mustache.github.io/
[2] https://github.com/advisories/GHSA-cfhg-9x44-78h2
[3] https://github.com/baidu/amis/blob/1.9.0/src/utils/tpl.ts#L54

name and the function name, because `expression` implies that it is a JavaScript expression, and `evalExpression` implies that the code will be evaluated. As such, it is reasonable to assume that no untrusted data will be passed into `expression`. Thus, this flow should not be flagged.

Other than the aforementioned param-sink flows, there exist other types of taint flows facing similar problems. For example, a clear-text logging flow checks whether a variable that contains sensitive information flows into logging statements. While the sink is obvious in this case, i.e., calling the logging libraries, it is hard to precisely define the source, as we need to determine whether it contains some sensitive information, like passwords or authentication keys. A possible way to tackle this is to reason about the identifier name of the variable being logged. For example, logging a parameter named `user_password` should be flagged, while logging a parameter named `HTTP_status_code` should not be flagged.

As seen from the above examples, automatically determining whether a data flow is unexpected is non-trivial, because the fact that we consider it a vulnerability depends on the rather implicit meaning of `pid`, `expression`, `user_password`, and `HTTP_status_code` and the expectations a developer has based on them.

To effectively exploit the implicit information in code, we introduce a bimodal taint analysis tool, Fluffy ("FLagging Unexpected Flows For better securitY"). The first modality is *code*: Fluffy uses a mining analysis implemented in CodeQL to find examples of flows from parameters to vulnerable sinks. The second modality is *natural language*: Fluffy uses a classifier that, based on a corpus of such examples, learns how to distinguish unexpected flows from expected flows using natural language information.

The mining analysis runs CodeQL queries to extract flows from parameters to sensitive sinks in npm packages and flows from variables to logging statements on GitHub and LSTM.com. CodeQL queries, written by GitHub researchers and community contributors, can be executed against databases where source code is treated as data. A flow extracted by our customized CodeQL queries consists of the parameter name, the sink type, and other metadata, such as package name, function name, and parameter documentation. Using this method, we obtain a corpus of around 3 million param-sink flows and around 4 million logging flows.

Using the corpus collected from CodeQL queries, we train a machine learning model to help distinguish unexpected flows from typical flows. We develop four models, offering different trade-offs between the manual efforts required and the accuracy of predictions. The first model, Sink Prediction, is a classifier that predicts which sink the flow belongs to, trained by the dataset extracted by CodeQL, i.e., it can be trained without any manual labels. The second model, Novelty Detection, is a One-class Support Vector Machines (OC-SVM) [37] which learns a decision function to separate the expected and unexpected names. It requires some manual efforts as the users have to produce a few (less than ten) seed names to train the OC-SVM. The third model, Binary Classification, is a classifier that directly predicts whether a flow is unexpected, trained with a manually labeled ground truth dataset. This requires the largest manual effort as users have to label hundreds of flows. The fourth model, Codex, is based on the large language model (LLM) from OpenAI with the same name, trained on hundreds of millions of lines of code [9]. Our Codex model uses a few-shot learning technique. Our model first generates a prompt using ten examples

extracted from the manually labeled ground truth dataset. Then our model queries the OpenAI Codex model with the prompt to predict whether a flow is unexpected.

We evaluate Fluffy on a ground truth dataset. Each entry in the ground truth dataset is manually labelled as either expected or unexpected by the authors. The quality of the ground truth dataset has been cross-checked by domain experts from GitHub. On our ground truth dataset, Fluffy achieves a precision of 81%-97%, a recall of 80%-100%, and an F1-score of 76%-97%. Moreover, Fluffy flags eleven vulnerabilities in real life projects, of which six have been confirmed by the developers. Additionally, we apply Fluffy on a dataset with 131 previously known vulnerabilities, of which Fluffy is able to flag 117.

In summary, this thesis contributes the following:

- A bi-modal taint analysis technique: the first approach to utilize identifier names through machine learning and logic-based taint analysis to find unusual flows.
- Four different models are presented with different trade-offs between the manual efforts required and accuracy.

2 Background

2.1 JavaScript/NPM Security

JavaScript is one of the most popular programming languages. NPM is the package manager for JavaScript developers to download third-party libraries. Although convenient, using NPM also imposes security risks [41], as code is freely shared and used without any checks. For example, individual packages could impact large parts of the entire ecosystem. Thus, it is crucial to have an automated testing suite to make sure code is indeed secure. Our work aims to find if the shared library code provides a sensible parameter name and function name, such that the client would understand the purpose and expect the potential vulnerability of using it. Specifically, our work is based on taint analysis [38].

2.2 CodeQL Mining Analysis

CodeQL [1] queries, written by GitHub researchers and community contributors, can be executed against databases where source code is treated as relational data. CodeQL provides a suite of analyses, available as libraries, to make finding common vulnerabilities easier.

2.2.1 Param-sink Flows

In Fluffy, we use CodeQL queries to extract potentially problematic parameter-sink flows from APIs of publicly available npm packages (i.e., our CodeQL queries are function-level analysis). Specifically, we are looking for data flows that flow into any of the four sinks: code injection[2], command injection[3], reflected XSS[4], and path traversal[5].

Code injection occurs when user input is evaluated as code without proper sanitization, leading to arbitrary code execution. In CodeQL, a flow is vulnerable to code injection if the parameter is treated as JavaScript code for evaluation, or passed to a framework that evaluate it as an expression. As an example, the parameter `name` of the function `createDynamicClass`[6] is open to code injection, as `name` is embedded directly into JavaScript code.

[1] https://codeql.github.com/
[2] https://github.com/github/codeql/blob/main/javascript/ql/src/Security/CWE-094/CodeInjection.ql
[3] https://github.com/github/codeql/blob/main/javascript/ql/src/Security/CWE-078/CommandInjection.ql
[4] https://github.com/github/codeql/blob/main/javascript/ql/src/Security/CWE-079/ReflectedXss.ql
[5] https://github.com/github/codeql/blob/main/javascript/ql/src/Security/CWE-022/TaintedPath.ql
[6] https://github.com/deepkit/deepkit-framework/blob/v1.0.1-alpha.65/packages/core/src/core.ts#L685

Command injection occurs when user input is passed to a command for execution without proper sanitization, leading to arbitrary command execution. In CodeQL, a flow is vulnerable to command injection if the parameter is directly passed to a library routine that executes a command, such as `require('child_process').exec`. The parameter `pid` of the "ps" package mentioned in the introduction is an example of such vulnerabilities.

Reflected XSS occurs when a vulnerable web application returns malicious code provided by the attacker that execute in the client's browser. Reflected cross-site scripting (XSS) specifically refers to the type of XSS attack where the malicious code comes from the current HTTP request. In CodeQL, a flow is vulnerable to reflected XSS when user input are directly written to an HTTP response without proper sanitization. For instance, the parameter `title` of the function `plot`[7] is open to reflected XSS attack, as `title` is unsafely embedded in an HTTP response.

Path traversal occurs when data used for file path construction is uncontrolled, allowing the attacker to access unexpected resource. The CodeQL query checks whether the input to file system API calls (such as `fs.readFileSync`) are properly sanitized. As an example, the parameter `key` of the functions `get` and `set`[8] is vulnerable to path traversal. As `key` is unconstrained, it could traverse out of the cache directory and gains access to unexpected resources.

Beside the four mentioned sinks, we also extract flows where the parameter does not flow into any sink. In this case, we say that the flow has a *None*-sink.

2.2.2 Logging Flows

In addition, we also use a CodeQL query to extract logging flows[9]. This query extracts all variables that flow into a logging statement. This query then uses regular expressions to additionally flag likely sensitive flows, whose parameter names match the regular expressions.

2.3 Machine Learning

To reason about the identifier names, we have to employ machine learning techniques. Machine learning algorithms build a model based on training data to make prediction on previously unseen data. It has shown promising result in various fields, including natural language processing [7, 13] (NLP). Our problem belongs to the NLP field.

2.3.1 Supervised vs Unsupervised Learning

Machine learning algorithms can be classified into supervised learning and unsupervised learning. In supervised learning, labels are required for each data point, and the model learns to map the input features to a label. In unsupervised learning, labels are not required, and the model automatically learns the pattern in the dataset. Our approach includes both: Sink Prediction and Binary Classification are supervised learning, while Novelty Detection is unsupervised learning.

[7] https://github.com/epispot/EpiJS/pull/191
[8] https://github.com/faastjs/faast.js/pull/930
[9] https://github.com/github/codeql/blob/main/javascript/ql/src/Security/CWE-312/CleartextLogging.ql

2.3.2 Neural Network

Neural networks are a subset of machine learning methods, which are inspired by biological neurons. A neural network is separated into different layers, each of which has many neurons. Each neuron computes a signal and passes it to the next layer. The signal is computed by a non-linear function, based on the input, a weight vector, and a bias term. The neural network learns by updating the weight vector and bias via the backpropagation algorithm. In our approach, we use a feedforward network [36] and Long Short-Term Memory (LSTM) network [18]. The main difference between feedforward network and LSTM is that in feedforward network, information only flows in one direction, whereas LSTM uses internal states to "memorize" previously processed information in the same sequence, making it suitable to process sequential data. In the Sink Prediction and Binary Classification model, we use a LSTM network to process the parameter documentation, then we use a feedforward network to learn the label based on all the input features.

2.3.3 One-class Support Vector Machine

For the Novelty Detection model, we use a One-class Support Vector Machine (OC-SVM) [37]. OC-SVM is based on Support Vector Machine (SVM), and is built for novelty detection (i.e., one-class classification). It is an unsupervised algorithm where unlabeled data are used as training data to learns a decision function, which is used to classify new unseen data as similar or dissimilar to the training data. The key idea of OC-SVM is that, unlike traditional SVM classifier which finds and maximizes the margin between the support vectors of different classes, OC-SVM separates the training data from the origin in the feature space with hyperplane. For more details, please consult the original paper [37].

2.3.4 Few-shot Learning

Few-shot learning models are models that perform well by just learning from a few examples. In particular, few-shot refers to having more than one example. Examples of few-shot learning models include large language models (LLMs). For example, the GPT-3 paper is titled "Language Models are Few-Shot Learners" [7], and it is shown that the scaling up in LLMs greatly improves task-agnostic few-shot performance. As another example, GitHub Copilot [10] is based on a LLM trained on code called Codex [9]. Developers can query Copilot via a prompt to generate code or code documentation. One way to utilize few-shot learning in LLMs is prompt engineering. Prompt engineering is about using examples in the prompt to guide a LLM to output a correct prediction. For instance, guiding a LLM to perform question and answering can be done via providing examples in the prompt [11]. Through prompt engineering, our Codex method queries the OpenAI Codex language model to predict if a flow is unexpected.

[10] https://copilot.github.com/
[11] https://beta.openai.com/playground/p/default-qa

3 Approach

3.1 Problem Statement

Running the CodeQL mining analysis of Fluffy (explained in Section 2.2) yields us a set of **potentially** problematic flows F, which is *precisely specified but overapproximate*, computed purely based on code structure. Our goal is to find U, a set of *imprecisely specified* flows that are unexpected to the client, such that we can determine the subset $F \cap U$. Whether a flow is unexpected depends on the context and the interpretation of the client, thus it is fuzzy by nature. Therefore, the approach to find unexpected flows should be able to generalize in different contexts and understand the interpretation of the client. In this thesis, we focus on two kinds of flows, each with a different context: param-sink flows and logging flows. Figure 3.1 shows a high-level overview of our approach.

3.1.1 Unexpected Param-sink Flows

For param-sink flows, the mining analysis gives us an analysis of library code in isolation in the form of parameter-sink flows. However, assuming all parameters in the library receive unsanitised data is not reasonable and gives us an unmanageably large number of results. So we need to make reasonable worst-case assumption about untrusted data being passed in by non-malicious clients.

A reasonable worst-case assumption is to ask whether a JavaScript programmer would expect data passed into p to flow into a sink of kind s. To understand what is "expected", we can ask the following question: *When the programmer looks at the parameter name and context of a flow, do they expect it to flow into its sink?* If the answer is yes, then this flow is expected, and vice versa. For example, while it is expected that the parameter `cmd` of the function `execCommand` is executed as a command[1], it is unexpected that the parameter `value` of the function `staticSetConfig` flows into the command injection sink[2]. We consider this a reasonable worst-case assumption, because if a flow is expected, then we can assume that the clients would not pass untrusted data to it, since the potential vulnerability is expected.

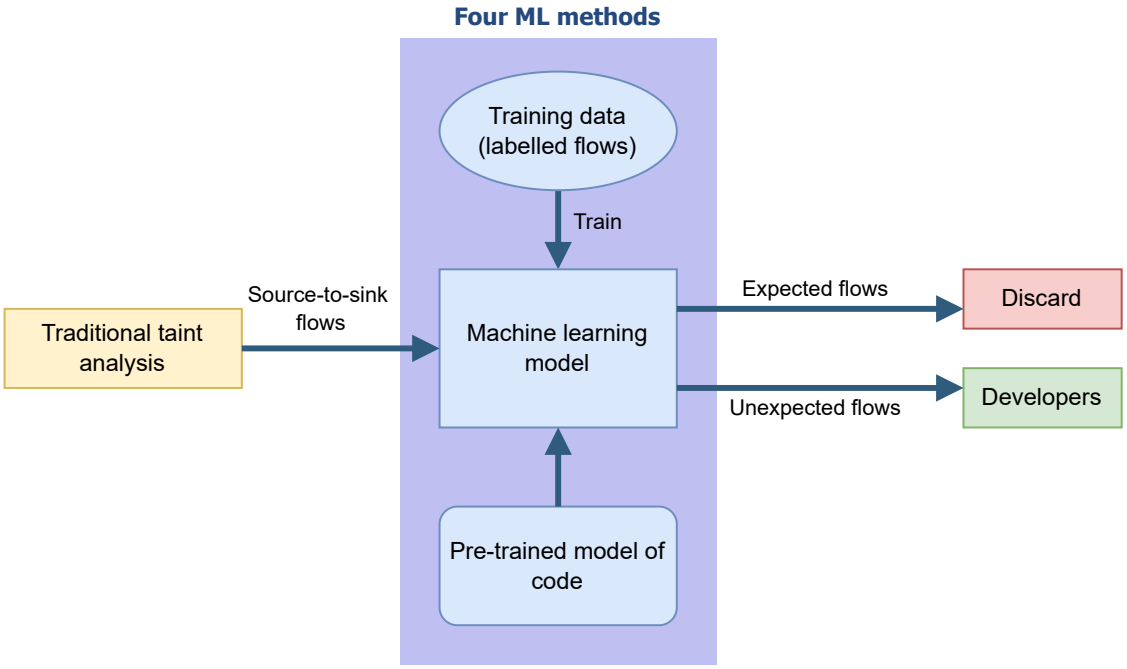
In essence, applying a reasonable worst-case assumption help us filter false positives (i.e., the expected flows) in the flows extracted by our mining analysis. Thus, given these parameter-sink flows extracted by CodeQL, we would like to detect those that are unexpected to a JavaScript programmer.

[1] https://github.com/AxelDeneu/hey/blob/master/lib/index.js#L13
[2] https://github.com/AKASHAorg/ipfs-connector/blob/master/src/IpfsConnector.ts#L164

3.1.2 Unexpected Logging Flows

For logging flows, the mining analysis gives us all variables that flow into a logging sink. In this case, we consider "unexpected" flow to be a flow where the source contains some sensitive data and is exposed due to flowing into a clear-text logging sink. To determine whether this variable contains sensitive data, one possible way is to look at the identifier names, assuming the library developer names the variable correctly. However, this could be challenging as there are subtle differences between similar names. For example, while `USER_ADMIN_PASSWORD`[3] and `forgotPasswordUrl`[4] both contain the word "password", only `USER_ADMIN_PASSWORD` likely exposes sensitive data.

Figure 3.1: Overview of our approach.



3.2 Main Challenges

How can we detect unexpected flows? As a first step, we can consider a naïve method: using regular expressions. We can ask JavaScript security experts to come up with a list of expected or unexpected names, then simply search the identifiers in the code base using regex. As a concrete example, the CodeQL query *UnsafeShellCommandConstruction*[5] first detects tainted flow from parameter to shell command execution, then uses regular expressions to exclude parameters whose name suggests they are meant to be a command. Nonetheless, this approach is far from perfect and

[3] https://lgtm.com/projects/g/JuZiSang/blog/snapshot/ed71c667c2a1084e06ff1741cf173c76a3394ccf/files/server/src/main.ts?sort=name&dir=ASC&mode=heatmap#xbce1407e3058eee6:1

[4] https://lgtm.com/projects/g/ibm-cloud-security/appid-serversdk-nodejs/snapshot/f5229c2ee55d7e0b61177777c076505b03241036/files/lib/strategies/webapp-strategy.js#x17b1c3315fa08247:1

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

cannot possibly capture every unexpected names. A more general approach is to train a machine learning model that, given a dataset of flows, learns what names are unexpected. However, doing so comes with certain challenges.

Firstly, the model has to predict "unexpected", which is only interpretable by human. As mentioned in the last section, whether a flow is unexpected depends on what a programmer thinks about the parameter names and context. So it is not possible to use handwritten rules to automatically classify which flows are unexpected. Thus, manual labelling is required. However, labelling would require human experts which are familiar with both JavaScript and common security vulnerabilities. Therefore, we should propose an approach that requires minimal human efforts. Another challenge to this is to find a consensus on what is considered unexpected, as there are no concrete rules that can determine such a thing.

Secondly, we need an effective representation of identifier names. Our model relies on the ability to reason about the relationships of identifier names, i.e., distinguishing between similar and dissimilar names. There are several challenges related to it. Firstly, the relationship should be expressed in the realm of software engineering. As an example, Levenshtein distance can be used to measure the edit distance between two names. However, such naïve method does not adequately capture the meaning of names, for instance, while "folder" and "directory" shares the similar meaning in the realm of software engineering, it has a high Levenshtein distance of eight. Secondly, the embeddings should preserve the order of the names, such that we could distinguish between "array_of_maps" from "map_of_arrays". Thirdly, identifier names could have any combination of words, meaning it is susceptible to out-of-vocabulary (OOV) problem, where the embeddings cannot store all possible names.

Thirdly, for each sink, we have different data distribution. In reality, not all queries have the same level of precision. For some sinks, the majority of the flows might be unexpected. However, for some other sinks, unexpected flows might instead be the anomalies. Using anomaly detection with unsupervised clustering to detect unexpected flows seems sensible, but it will not work if the majority of flows in the sink are unexpected. Therefore, to properly generalize, we need a model that can handle both cases.

3.3 Learning-based Unusual Flows Detection

For each flow extracted by our mining analysis, we obtain the associated metadata, such as package name, function name, and parameter documentation (i.e., JSDoc `@param tag`[6]). We extract these associated metadata because they give programmers the context of the library, i.e., a programmer should be able to understand what this API does and how the parameter is used by reading these metadata. We do not include a lot of the details, as we want to keep the metadata short and precise. Since in real life, developers likely will not read through a lot of documentation or code before calling the function, and documentation in longer form could be noisy (e.g., README file). Note that we do not include the JavaScript code of the API, as the implementation (except for the fact it flows to a sink) is irrelevant in regard to flow being expected or not. The parameter name

[6] https://jsdoc.app/tags-param.html

and metadata should sufficiently convey the intention of the API, otherwise, this flow should also be flagged.

Formally, we define a flow as a tuple:

Definition 1 (Flow)**.** A flow is a tuple (p, pkg, f, d, s), where p is the parameter name, pkg is the package name, f is the function name, d is the parameter documentation, and s is the sink p flows into.

We develop four models aided by machine learning to detect unexpected flows from our dataset. The four models differ by the amount of manual efforts required.

All the models utilize the VarCLR [10] encoder to encode the variable names and function names in our flows. VarCLR helps address the challenge of obtaining an effective representation of identifier names. VarCLR is an encoder trained with contrastive learning, using a renaming dataset mined from GitHub commits. In essence, VarCLR (i) provides an embedding that can reason about the relationship of names in the realm of software engineering, (ii) processes the names as a list of tokens instead of a single string, and (iii) uses Byte-Pair Encoding (BPE) to handle OOV problem.

3.3.1 Method 1: Sink Prediction

The first model can be trained automatically without manual labelling. We train a model which takes p, f, and d of a flow as input and predict the sink it flows to. The model classifies a flow as unexpected if the probability of it flowing into its sink s is very low. In this model, we want the model to learn from the dataset to determine the actual sink the flow should flow to. In particular, we are interested in learning from the large set of None-flows we have in the dataset, as these are names that should not flow into any sink.

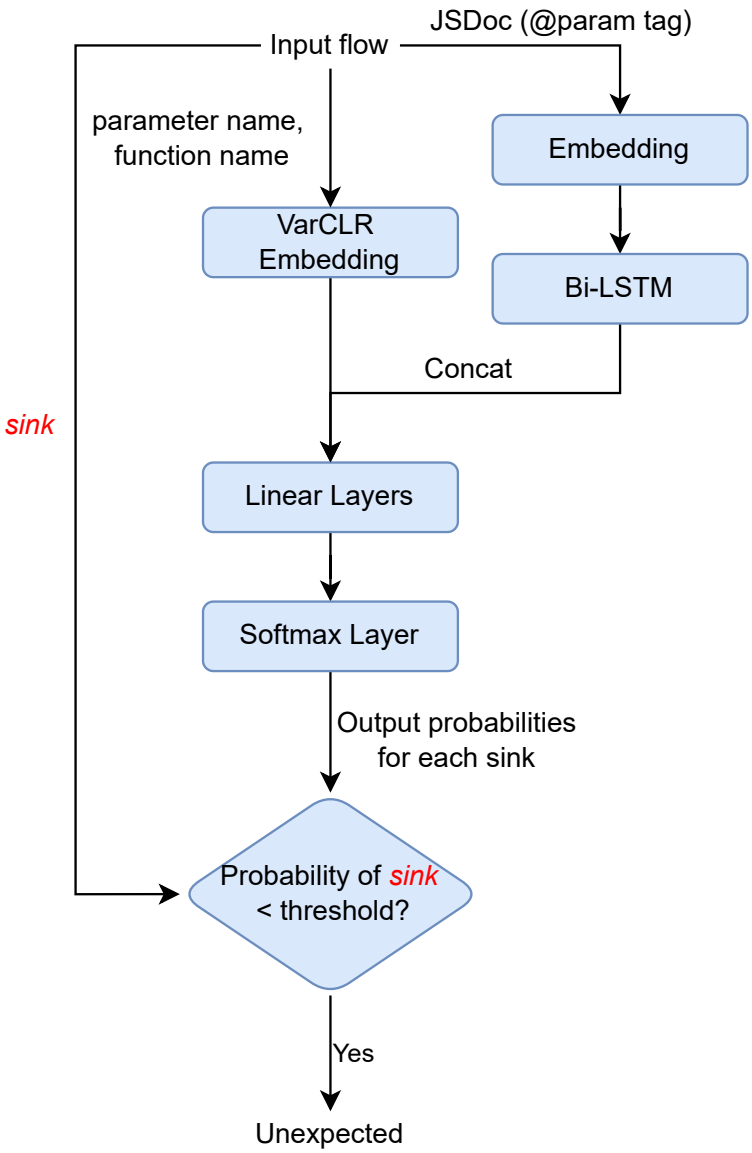
The model is a multi-class feed-forward network (see Figure 3.2 for illustration). We use VarCLR embeddings for p and f (i.e., p and f are encoded by the VarCLR encoder). For d, we use another randomly initialized embedding and a Bi-LSTM layer to process it. We then concatenate the embeddings of p, f, and d together to pass through a hidden layer. Finally, the hidden layer output will be fed into a softmax output layer to output the probabilities of the flow flowing into each sink. For param-sink flows, we use weighted loss function to tackle the imbalanced dataset problem, since most of the flows has a None-sink. For logging flows, we predict the flow to either flow into a *logging* sink or a None-sink, where flowing into *logging* sink means that it is likely insensitive data (and thus unproblematic) and flowing into None-sink means that it is likely sensitive data (and thus problematic).

Note that this approach is fully automatic in the sense that training this model involves no manually labeled data: p, f, d, and s are automatically extracted via CodeQL queries.

3.3.2 Method 2: Novelty Detection

The second model is semi-automatic: users provide parameter and function names as examples (*seed names*) to guide our model. Our model compares a flow with these seed names via a similarity function. Based on the similarity score, the model will classify them as unexpected.

Figure 3.2: Neural network architecture for method 1: sink prediction.



In this approach, we formulate our task as a novelty detection problem [19]. While other approaches use no labels (unsupervised learning) or data labelled as normal or abnormal (supervised learning), novelty detection model only normality, i.e., it only requires pre-classified normal data. Novelty detection defines a boundary of normality, where if a data point lies outside the boundary, it is considered a novelty. In comparison, the approach that learns from both normal and abnormal data would only work if abnormal data can generalize well. But for novelty detection, as long as the abnormal data is far from the normality boundary, it can be detected.

We treat the seed names as normal instances. Thus, our dataset extracted by CodeQL queries are novelties to be classified. Note that the anomalies do not have to be a minority in the dataset for this approach to work.

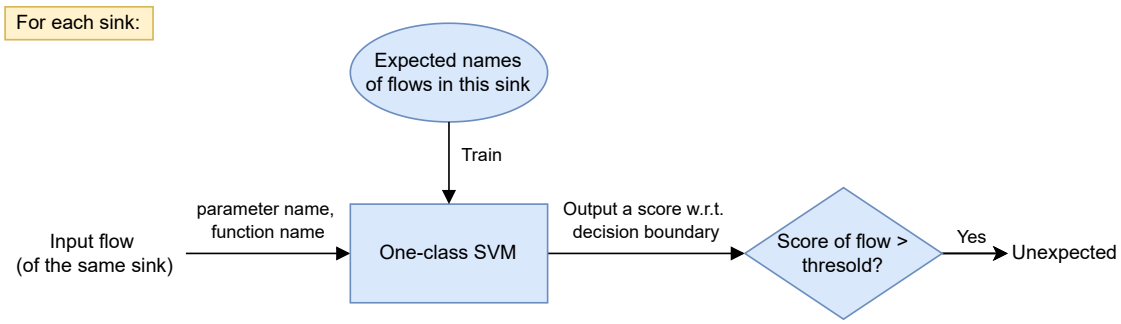
We fit a One-class Support Vector Machine (OC-SVM) with the user-provided seed names as the training data. This help us construct a soft boundary which we can use to detect names that

are unexpected. OC-SVM is suitable for novelty detection because it is sensitive to outliers and can be trained with only normal (seed) examples. Given these normal examples, OC-SVM learns a decision boundary for the seed names. We compute the signed distance to this decision boundary of OC-SVM as thresholds for the classification.

For each sink, we train the OC-SVM using the VarCLR embeddings of p and f of the provided expected names of this sink. In other words, flows with a None-sink are discarded. An illustration of this method can be seen in Figure 3.3. For param-sink flows, we use "expected" names as our seed names, i.e., a flow is considered unexpected if the parameter name is **dissimilar** to the seed names provided. For logging flows, we use "unexpected" names as our seed names, i.e., a flow is considered sensitive if the parameter name is **similar** to the seed names provided.

This model is semi-automatic as the developers need to manually provide a few (less than ten) expected names as examples.

Figure 3.3: Machine learning model for method 2: novelty detection.



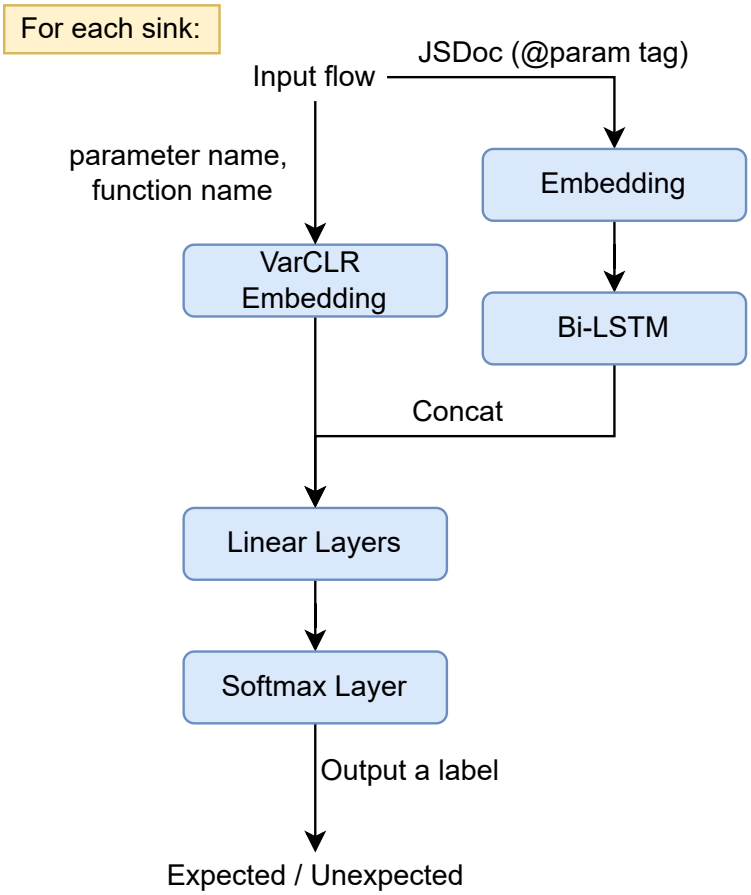
3.3.3 Method 3: Binary Prediction

The third model requires the most manual efforts: we train our model using the ground truth dataset that we have manually labeled. We directly predict whether a flow is unexpected using our manual labels as the training targets under supervised learning. In this method, we use a straight forward approach to explore whether it can achieve a better performance than the Sink Prediction model, by directly using hundreds of labelled examples as training data.

The model is the same as the model in 3.3.1, but with a different output layer. Instead of predicting which sink a flow belongs to, the model is predicting directly whether a flow is indeed unexpected or not. However, unlike the model in 3.3.1, we train one model per each sink using the labelled ground truth of the corresponding sink. In other words, flows with a None-sink are discarded.

This model demands more manual labeling than the model in 3.3.2, which could be challenging in real-life, as it expects significantly more efforts from domain experts.

Figure 3.4: Neural network architecture for method 3: binary prediction. Notice that sink is no longer an input to the model, and the output is directly the label.



3.3.4 Method 4: Codex

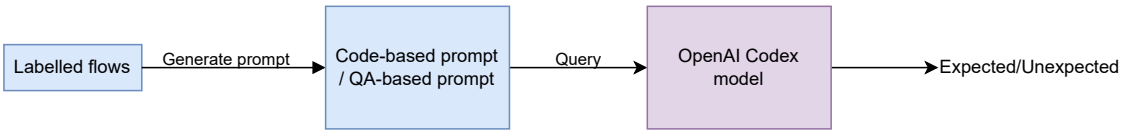
The fourth model is few-shot learning using Codex [9]. Codex is a GPT language model fine-tuned on source code mined from GitHub. It is the model that powers GitHub Copilot [7]. The goal of Codex is to synthesis programs, and it is shown to perform better than the original GPT model in this task.

Using Codex allows us to leverage source code-based large language model as Codex is trained on hundreds of millions of lines of code, as such, we expect Codex to have an effective representation of identifier names and documentation in the latent space. Although our downstream task is different from the original task (classification rather than token generation), Codex can understand what to do if a few examples are given in the prompt. The fact that the base GPT model has been trained with classification tasks before also help Codex understand the classification task.

We query the OpenAI Codex model by sending a prompt to it. For each flow, we construct one prompt with ten examples (randomly selected from our manually labeled ground truth) for Codex to learn from. Then Codex will output a sequence of tokens according to the prompt. The output generated by Codex is the label prediction. A high-level overview can be seen in Figure 3.5.

[7] https://copilot.github.com/

Figure 3.5: High-level overview of method 4: Codex.



In our experiment, we give Codex two types of prompts, one for param-sink flows and one for logging flows. Both prompt are in code-form, a structure that is familiar to the Codex model.

Given a param-sink flow, we construct a function the with empty implementation using p, f, and d. Then we add a comment below that explains whether p is expected to flow into s, i.e., it directly predicts whether this flow is unexpected or not. See Figure 3.6 for a concrete example.

Given a logging flow, we construct a stub function calling `console.log` on its parameter p. Then we add a comment below that explains p is being logged in clear-text by the stub function and states whether it exposes sensitive data, i.e., it directly predicts whether it exposes sensitive data or not. See Figure 3.7 for a concrete example.

Figure 3.6: An example of param-sink flow prompt. For brevity, only the first two examples are shown. Notice that the last line is unfinished, as this is where Codex makes the prediction.

```
/**
* @param path - The path to check for existence *
*/
function exists(path){
}
// In the above function "exists", the parameter "path" flows into
    TaintedPath sink (uncontrolled data used in path expression), which is
    expected.
/**
* @param arr -
*/
function asyncForEach(arr){
}
// In the above function "asyncForEach", the parameter "arr" flows into
    TaintedPath sink (uncontrolled data used in path expression), which is
    expected.

......

/**
* @param arch -
*/
function cacheDir(arch){
}
// In the above function "cacheDir", the parameter "arch" flows into
    TaintedPath sink (uncontrolled data used in path expression), which is
```

Figure 3.7: An example of logging flow prompt. For brevity, only the first two examples are shown. Notice that the last line is unfinished, as this is where Codex makes the prediction.

```
function f(passwordChanged) {
  console.log(passwordChanged);
}
// In the above function f, the parameter "passwordChanged" is being logged
    , which likely exposes insensitive data.

function f(DEFAULT_ADMIN_PASSWORD) {
  console.log(DEFAULT_ADMIN_PASSWORD);
}
// In the above function f, the parameter "DEFAULT_ADMIN_PASSWORD" is being
     logged, which likely exposes sensitive data.

......

function f(torHashedPassword) {
  console.log(torHashedPassword);
}
// In the above function f, the parameter "torHashedPassword" is being
    logged, which likely exposes
```

4 Evaluation

We evaluate all four models mentioned in our approach, focusing on the following research questions:

- RQ1: How effective is Fluffy at flagging vulnerabilities?
- RQ2: How effective is Fluffy in a real-life scenario?
- RQ3: What is the trade-off between human efforts and model performance?
- RQ4: How reliable are the ground truth labels?
- RQ5: How scalable are the neural models of Fluffy?

4.1 Experimental Setup

4.1.1 Model Hyperparameters

The input dimension of each method except Codex is 768, which is the output dimension of VarCLR encoder.

In the Sink Prediction and Binary Classification methods, the hidden layer dimensions are 500 and 250 respectively. Adam optimizer is used with a learning rate of 0.001. For the Sink Prediction method, 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 and use it as the validation set. For the Binary Classification method, the training batch size is 32, and the model early stops when validation loss does not decrease in 50 epochs.

We use Radial Basis Function (RBF) as kernel for the OC-SVM in the Novelty Detection method. 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.

4.1.2 Dataset Preparation

We have two types of flows for evaluating Fluffy: param-sink flows and logging flows.

4.1.2.1 Param-Sink flows

Param-sink flows are flows where the parameter is from a public API on npm and flows into a vulnerable sink (discussed in Section 2.2). We filter flows where the parameter name has less than

Table 4.1: Ground truth labels.

	Random Set	Balanced Set
Sinks	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)
		Unexpected (Total)
Logging		245 (340)

two characters, as we consider having only one character very vague and does not show enough information to developers. After filtering, there are 3,228,034 flows with a None-sink, 1,123 code injection flows, 15,135 path traversal flows, 1,498 command injection flows, and 70 reflected XSS flows in our dataset.

For evaluation, the authors have manually labeled two sets of ground truth. The first set is the *random set*, which consists of flows randomly selected from the param-sink flows extracted by our CodeQL queries. The second set is the *balanced set*, which is a union of (i) the *random set*, (ii) randomly selected flows that are classified as unexpected by the frequency-based and an early version of the Sink Prediction classifier, and (iii) flows previously found to be unexpected. The reason we use two datasets is, while the *random set* is unbiased, randomly and directly selecting from our flow dataset would not give us many unexpected flows if the majority of the flows in the sink are false positives. Therefore, we also need the *balanced set* for evaluation as it contains more unexpected flows. We can see the labeled ground truth for each sink and the number of unexpected flows in the ground truth in Table 4.1. As shown in the table, our manual labeling reveals that simply flagging all flows as unexpected would result in many false positives, demonstrating the importance of our approach.

4.1.2.2 Logging Flows

When a parameter is being logged (for example, `console.log(x)`), we say that it flows into a logging sink and thus is a logging flow (discussed in Section 2.2). We record names of all variables that flow into the logging sink. Unlike param-sink flows, we only have one sink (the logging sink), we do not care if the source is from a public API on npm packages, and we do not record other metadata in this case. In total, we have 4,535,851 logging flows.

We compare our approach against a CodeQL query that uses regular expressions to flag likely problematic flows. This query first extracts the logging flows, then uses a "password-related" regular expression[1] to obtain parameter names that are likely password-related, such as `authkey` and `password`. Finally, the query uses a "likely insensitive" regular expression[2] to filter out names that

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

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

Table 4.2: Summary of the difference between each method.

Methods	Training data	Number of models	Need a threshold?	How to compute F1-score?
Fluffy Sink Prediction	All param-sink flows	One	Yes	Find the best F1-score via plotting Precision-Recall curve
Fluffy Novelty Detection	Seed names	One for each sink	Yes	Find the best F1-score via plotting Precision-Recall curve
Fluffy Binary Classification	*balanced set*	One for each sink	No	Average the F1-score from K-Fold cross-validation
Fluffy Codex	*balanced set*	One (from OpenAI)	No	Compute F1-score directly from output
Frequency-based	All param-sink flows	One	Yes	Find the best F1-score via plotting Precision-Recall curve

match the password-related regular expression but are likely insensitive, such as `encryptedPassword`. This query flagged 6,044 flows from all logging flows as problematic.

For ground truth labelling, we have randomly selected and labelled 300 unique names from those 6,044 flows. To include cases that are not considered by the query, we also label all (40) unique names match the "password-related" regular expression **and** the "likely insensitive" regular expression.

4.1.2.3 Seed Names for OC-SVM

In the Novelty Detection method, a list of seed names have to be provided for each sink. Based on our background knowledge on each sink and vulnerability, we come up with a few names. For all sinks except the logging sink, the seed names are the expected names of the sink. For the logging sink, the seed names are the unexpected names. We list the seed names for each sink here:

- Code injection: `eval`, `execute`, `compile`, `render`, `callback`, `function`, and `fn`.
- Command injection: `execute,command`.
- Reflected XSS: `send,content`.
- Path traversal: `file`, `directory`, `path`, `cwd`, `source`, and `input`.
- Logging: `authkey`, `password`, `passcode`, and `passphrase`.

4.2 RQ1: Effectiveness of Fluffy

In this section, we explain the evaluation process for each method and discuss the results. We evaluate our approach by investigating how well it can classify unexpected flows. We use F1-

score as metric to compare the performance of different methods. See Table 4.2 for a summary of difference between different methods. All methods that require a threshold are evaluated by plotting the precision-recall curve (PR curve) on the output. The PR curve shows the relationship between precision and recall for different thresholds. We then record the best F-1 score obtainable in the PR curve.

For Fluffy Sink Prediction, we train the model with all flows that have not been labelled, i.e., flows that are not in the *balanced set*. Since the model outputs probabilities, a threshold is required to make a prediction.

For Fluffy Novelty Detection, we train a OC-SVM for each sink (i.e., four sinks for param-sink flows, and only one sink for logging flows). Each OC-SVM is trained with the seeds of the corresponding sink. The model outputs a score for each flow, which represents the distance between the flow and the boundary of the OC-SVM. This model requires a threshold to make a prediction based on the output scores.

For Fluffy Binary Classification, we train a model for each sink. Each model uses the *balanced set* as the training set (irrelevant flows that belong to a different sink are filtered). We use K-Fold cross validation for evaluation, with K set to five. This model directly classifies whether the flow is expected, a threshold is not needed. We record the averaged F1-score across the five folds for each model.

For Fluffy Codex, for each flow to evaluate, we randomly draw ten examples of the same sink from the *balanced set* to put in the prompt. As this model directly classifies whether the flow is expected, we simply compute the F1-score based on the predictions.

For each kind of flows (param-sink flows and logging flows), we have a non-neural baseline for comparison. For param-sink flows, we use Frequency Counting as our baseline. The Frequency Counting method is a naïve method that simply computes how frequently a sink appears for a given parameter name. For each flow with the same parameter name, we count how often it flows to different sinks. This is also a threshold-based method, as it simply outputs the statistic of how often the parameter flows into a certain sink. For logging flows, we use the aforementioned CodeQL query 4.1.2.2 as our baseline.

4.2.1 Param-sink Flows Evaluation

For param-sink flows, all the methods are evaluated on the *random set* and the *balanced set*.

The result of *random set* is shown in Table 4.3, with PR curve in Figure 4.1 and ROC curve in Figure 4.2. We can observe that all approaches of Fluffy outperform the Frequency Counting baseline for all sinks. Out of all the methods, the Novelty Detection method performs the best in all sinks, followed by the Binary Classification method. The methods perform especially well on the command injection and reflected XSS sinks (0.9 to 1.0 F1-Score), while the code injection sink has around 0.8 F1-score. However, the path traversal sink has a lower and fluctuating F1-score: 0.45 at best. This is likely due to it having very few (8) unexpected flows in the *random set*.

Table 4.3: 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
Freq.	0.59	1.00	0.74	0.91	0.67	0.77	0.68	1.00	0.81	0.22	0.25	0.24

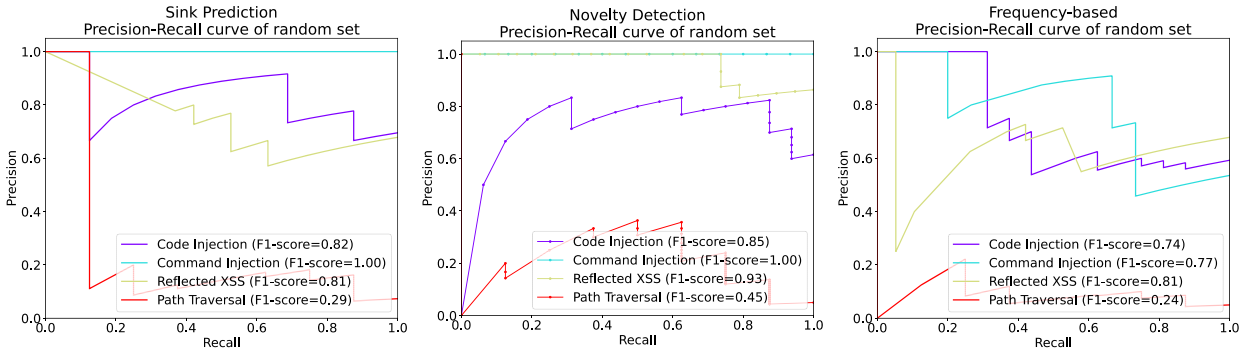


Figure 4.1: PR curves on the random set for Sink Prediction, Novelty Detection, and Frequency Counting (left to right).

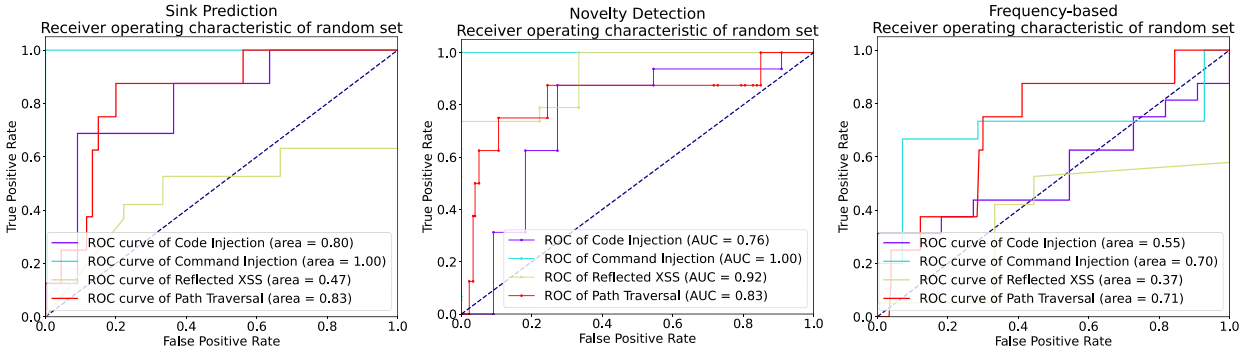


Figure 4.2: ROC curves on the random set for Sink Prediction, Novelty Detection, and Frequency Counting (left to right).

Table 4.4: 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
Freq.	0.33	1.00	0.50	0.93	0.92	0.93	0.63	1.00	0.77	0.42	0.63	0.51

The result of *balanced set* is shown in Table 4.4, with PR curve in Figure 4.3 and ROC curve in Figure 4.4. In this setup, the Binary Classification method performs the best, surpassing the Novelty Detection method significantly on the code injection and path traversal flows. This could be due to the *balanced set* containing many more diverse names, and the Binary Classification method have the advantage of learning from the dataset, while the Novelty Detection method relies on the same 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 in the *balanced set*. For command injection flows, we can see there is a performance boost when using CodeQL queries alone and the frequency-based method. This is likely due to the majority of command injection flows in the *balanced set* are unexpected, i.e., treating them all as unexpected would still give a decent F1-score in this setup. Another interesting finding is that the Sink Prediction method 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 *random set*.

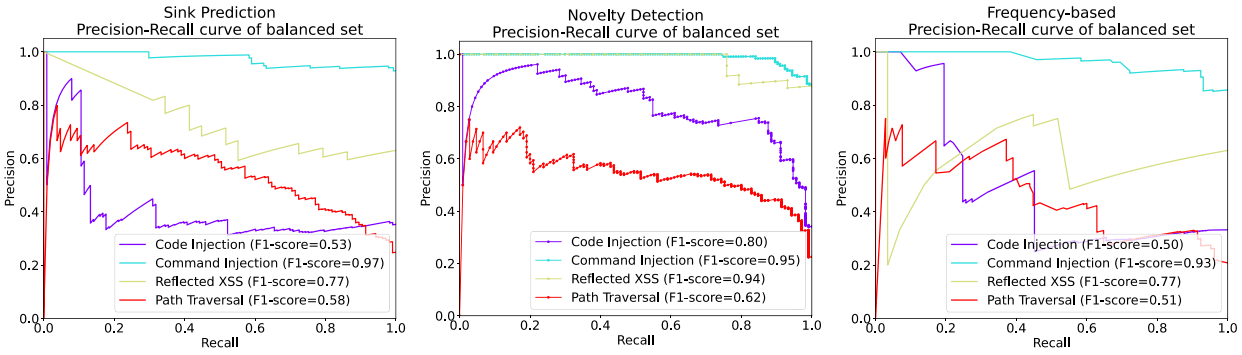


Figure 4.3: PR curves on the balanced set for Sink Prediction, Novelty Detection, and Frequency Counting (left to right).

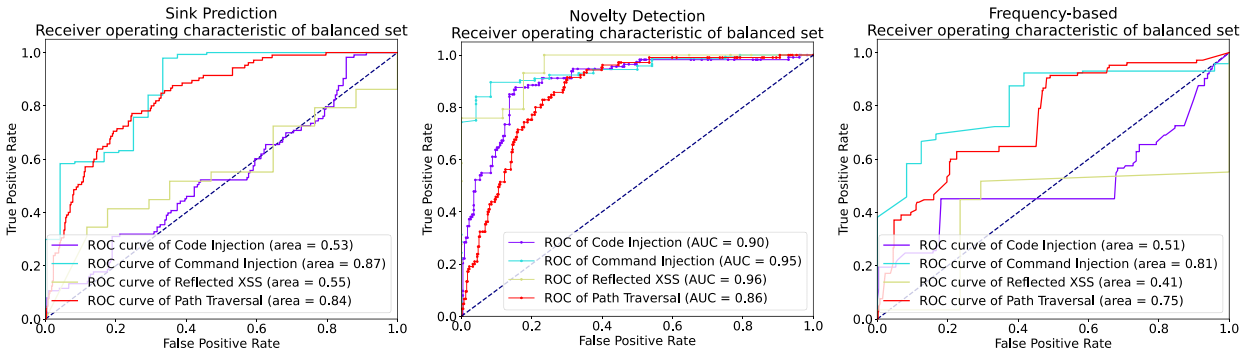


Figure 4.4: ROC curves on the balanced set for Sink Prediction, Novelty Detection, and Frequency Counting (left to right).

In summary, Fluffy approaches prove to be a significant improvement over the non-neural baseline of using a naïve statistical approach. The two best methods are the Novelty Detection method and the Binary Classification method, while the Sink Prediction and the frequency-based method do not work that well. Our interpretation is that how unlikely it is that a source flows to a sink does not help understand how unexpected a flow is, and it requires some form of manual labeling (e.g., providing seed names in Novelty Detection or labeling flows in Binary Classification) to teach the model how to learn what an unexpected flow is. Additionally, the Codex method does not perform as well as the others. We have two possible explanations for this: (i) Codex heavily depends on the examples given in the prompt, for example, if most of the examples given are expected, then Codex will only predict the flow to be expected, or (ii) Codex is not trained on data related to this prediction task.

Path traversal flows are the hardest flows to classify for all the approaches. Upon manual inspection, we discover that the expected names for path traversal are particularly diverse. These expected names include, for example, file operands (such as `move` and `rm`), file extensions (such as `png` and `mp4`), and other names that resemble a file (such as `log` and `config`). It is hard for the models to learn all these files-related names, leading to a poorer performance. In comparison, for other sinks, the expected names are not as diverse. For instance, for command injection, the flow is only expected if the name something similar to `execute,cmd`.

4.2.2 Logging Flows Evaluation

Evaluating logging flows is slightly different from param-sink flows. First, it does not have a frequency-based method: all flows belong to the same sink (the logging sink) in this case. Second, for the Novelty Detection method, we detect unexpected flows using **unexpected flows as seeds**, instead of detecting unexpected flows using expected flows as seeds. In other words, we are trying to flag flows that are the similar to our seed names in this case.

The result is shown in Table 4.5, with PR curve in Figure 4.5 and ROC curve in Figure 4.6. Overall, the Binary Classification method performs the best, followed by the Novelty Detection method and Regular Expressions (CodeQL). Regular Expressions (CodeQL) gives the highest recall, while the Novelty Detection method and the Binary Classification method provide a better precision. Looking at this result, one could argue that the regular expressions used in the CodeQL query already provides a decent performance. However, we should keep in mind that writing such sophisticated queries requires years of efforts and expert knowledge on program analysis and security, whereas using our approach is simplistic and requires much less human efforts (in the case of the Novelty Detection method).

Moreover, even seasoned developers would not be able to write a perfect query, and Fluffy can help with that: our approach is able to flag false negatives that CodeQL query never would have flagged. As a concrete example, our approach is able to flag `passcode` as problematic, while CodeQL query fails to do so. The parameter name `passcode` is filtered, due to a bug in the regular expression, as confirmed by the developer[3]. This shows that our approach is able to not only filter false positives from taint analysis, but also flag false negatives.

[3] https://github.com/github/codeql/issues/11148

Table 4.5: Effectiveness of Fluffy on the logging flows.

	Logging Flows Dataset		
Approach	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

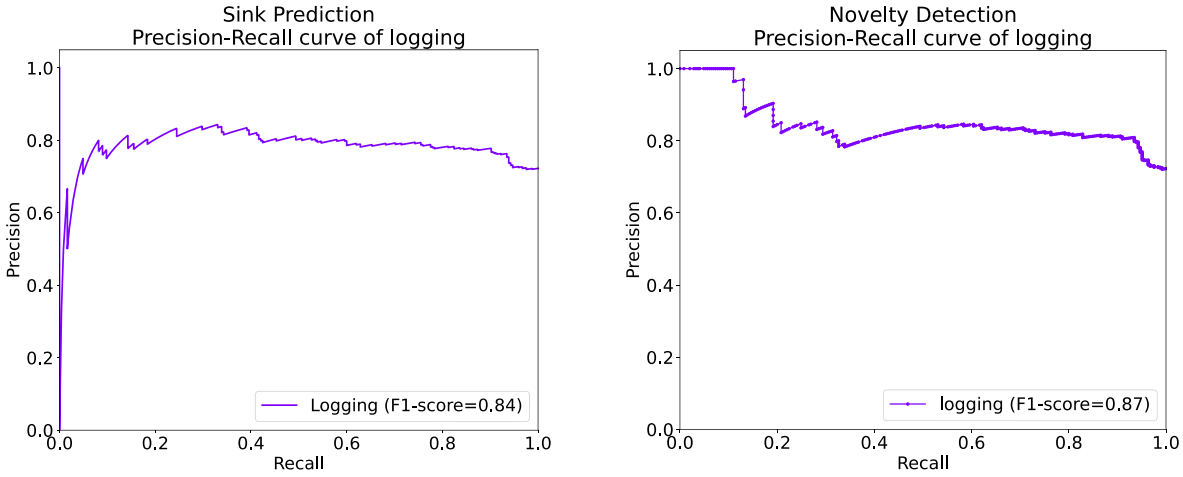


Figure 4.5: PR curves on the logging flows dataset for Sink Prediction, Novelty Detection, and Frequency Counting (left to right).

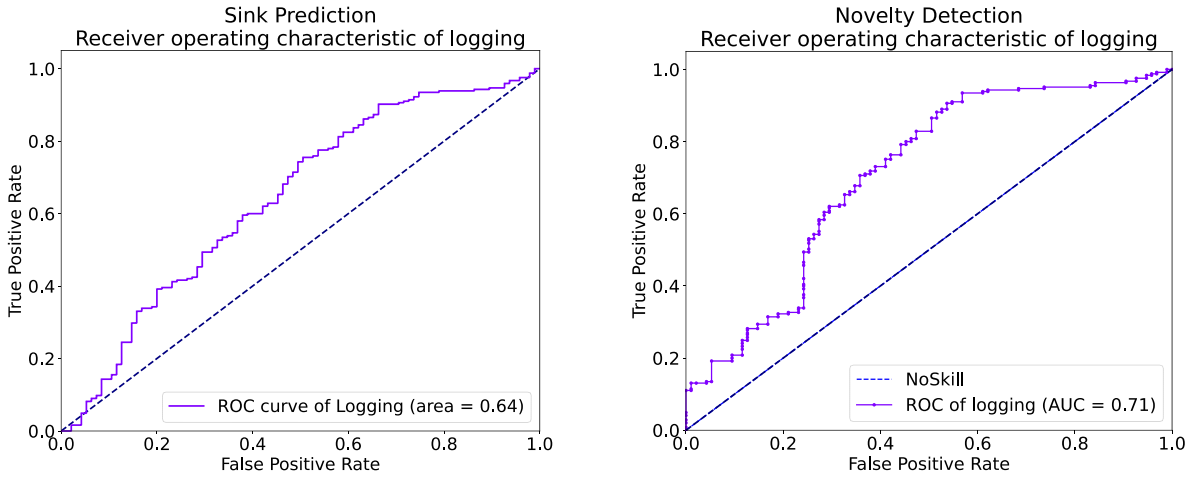


Figure 4.6: ROC curves on the logging flows dataset for Sink Prediction, Novelty Detection, and Frequency Counting (left to right).

4.3 RQ2: Fluffy in a real-life scenario

In this section, we evaluate how our model performs in a real-life scenario. We do that by running our models on past known vulnerabilities and previously unknown vulnerabilities.

4.3.1 Fluffy on Past Vulnerabilities

We evaluate our approach on SecBench.js [5], a benchmark suite of server-side JavaScript vulnerabilities. We evaluate Fluffy against three classes of vulnerabilities in SecBench.js: code injection, command injection, and path traversal, of which there are 33, 101, and 1 vulnerabilities respectively. Note that we only include flows where the source is a parameter of a public API.

In addition, we review all flows to see if they are indeed vulnerable. In total, there are 127 vulnerabilities that we consider unexpected. There are eight flows that we label as expected: one code injection and seven command-injection vulnerabilities. For example, SecBench.js includes CVE-2020-7636, where the parameter `command` of function `execADBCommand` of the npm package `adb-driver` is vulnerable to command injection. However, we do not consider this an unexpected (and thus vulnerable) flow, as it is clear from the context that this parameter is meant to be executed as a command.

We use the same evaluation methods as shown in Section 4.2.1, with a slight difference for the threshold-based methods. For the threshold-based methods, we pick a **default threshold** to use, which is the threshold that gives the best F-1 score in the *balanced set* in Section 4.2.1. We do this for two reasons. Firstly, we want to see if the threshold we learned can generalize to a different dataset. Secondly, finding the best threshold for evaluation via PR curve on such a skewed dataset (i.e., most flows are unexpected) is not fair to the non-threshold-based models, as simply picking a large threshold according to the PR curve and flagging all flows as unexpected would return a high F-1 score. In addition, note that we use the frequency-based method as baseline in this case, as these flows are not extracted by a CodeQL query.

We use recall to evaluate this dataset (Table 4.6), as the vulnerabilities here have already been found, and we are interested in whether Fluffy is able to flag them. In this setup, surprisingly, Sink Prediction has the best recall in code injection, followed by Novelty Detection. This is because the default threshold of Sink Prediction is set noticeably high, as shown in Table 4.4 where the precision is low and recall is high in Sink Prediction for code injection flows. A high default threshold is favorable to the SecBench.js dataset, where all but one code injection flows are unexpected. Nonetheless, all of our approaches performs better than the naïve frequency-based method, demonstrating the usefulness of our approach. Note that we omit the single path traversal vulnerability in our evaluation, as it does not have a statistically significant sample size.

4.3.2 Fluffy on Previously Unknown Vulnerabilities

In this section, we examine how effective Fluffy is for flagging vulnerabilities in the present day by looking at projects hosted on GitHub.

We created eleven pull requests on GitHub to fix the vulnerabilities where the unexpected flows are detected by Fluffy. By the time of this writing, five of them have been merged by developers who

Table 4.6: Effectiveness of Fluffy on the SecBench.js dataset.

	SecBench.js Dataset					
	Code injection			Command injection		
Approach	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
Freq.	0.94	0.47	0.63	1.00	0.65	0.79

were generally grateful about the changes. Five are still open without feedback from developers. One pull request is closed by the developer, who provides an alternate fix for the vulnerability.

4.4 RQ3: Manual Labels Required for Fluffy Binary Classification

From Table 4.3 and Table 4.4, we can observe that Binary Classification is one of the methods with the best performance. However, training the Binary Classification model requires manual labelling the dataset. In reality, we must consider the human efforts involved as they are scarce resources. Therefore, we would like to know how many labels are needed to achieve a good performance for the Binary Classification method.

In this experiment, we evaluate the Binary Classification method with different training set size using the *balanced set*. We use the same test set (amount to 20% of the flows belonging to the same sink) for all training set, i.e., we split train/test set as 80/20, 70/20, 60/20, etc.

The result of this experiment is displayed in Figure 4.7. It shows that for all sinks, the performance of the Binary Classification method improves as the dataset size grows. The only exception is command injection flows, where the model does not require many training examples to achieve a high performance.

The important takeaway from this experiment is that, although more manual efforts are required, it is possible to improve the performance of our approach by increasing the training set size. This provides an alternative when low-effort approaches like the Novelty Detection method does not achieve a desirable performance for a particular source or sink.

Comparing the Binary Classification method with 10% training data against the Novelty Detection method, we can see that the Novelty Detection method performs better, using even less training data (i.e., the seed names provided in Table 4.1.2.3). Therefore, with the trade-off between human efforts and model performance in mind, we consider the Novelty Detection method the more cost-effective model.

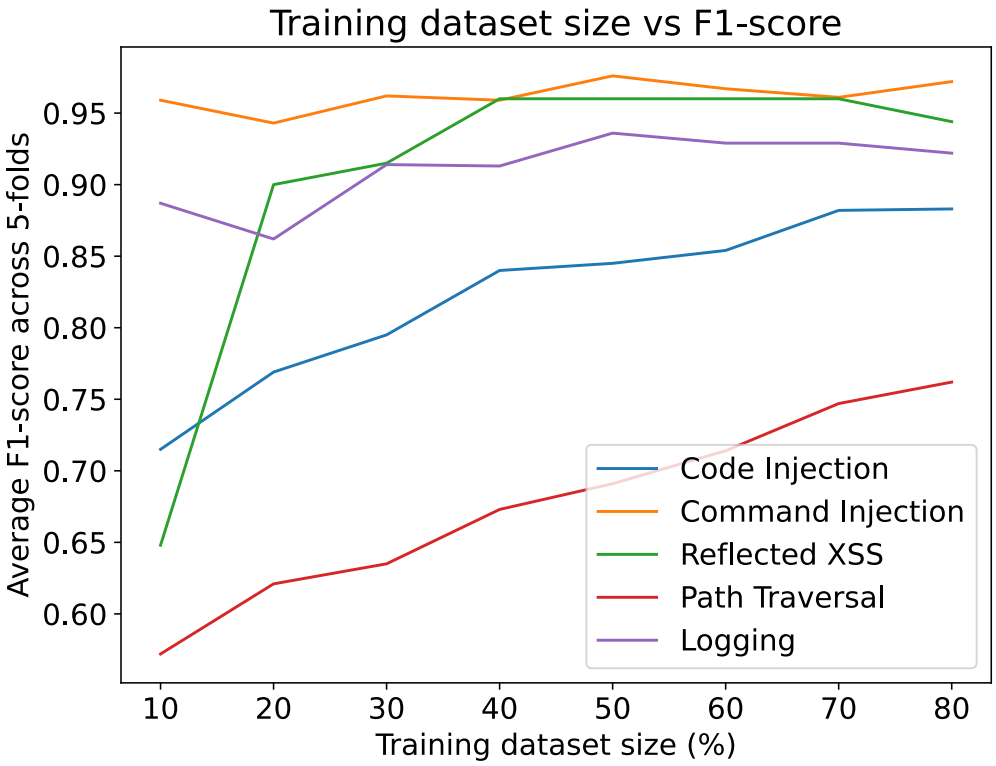


Figure 4.7: Training set size versus model performance.

4.5 RQ4: Ground Truth Reliability

The validity of our evaluation relies on the reliability of our ground truth labeling. In this section, we explain how we verify the reliability of our ground truth via an inter-rater agreement survey.

4.5.1 Ground Truth Inter-rater Agreement Survey

The survey consists of ten code injection flows, five command injection flows, five reflected XSS flows, and ten path traversal flows. Given the parameter name, function name, package name, sink, package description, parameter documentation, and function documentation, the participant has to determine for each flow whether the corresponding sink is expected. We invited four security and program analysis experts on JavaScript at GitHub to participate. In addition, the authors have also completed the survey, who are treated as another participant. Thus, we have five participants in total for the survey.

We evaluate the ground truth labels using inter-rater agreement. Using Krippendorff's alpha as our metric, our ground truth labels have a score of 0.74. It shows that our ground truth labels are reliable as the experts by and large agree with the labels in our ground truth.

4.6 RQ5: Fluffy Neural Model Scalability

To verify that the neural component of Fluffy can scale, we record the training time and evaluation time on our server. 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.

5 Discussion

5.1 Threat to Validity

Firstly, in our experiments, we evaluate our approach on five kinds of vulnerabilities only. Although the result is promising, we lack the empirical evidence that our approach generalizes well to other types of vulnerabilities.

Secondly, the experiment is done with JavaScript only, so our approach might not generalize to other programming languages.

Thirdly, since the ground truth labeling is done manually, it is prone to human errors. We have asked external inspectors to help, although they generally agree with our judgement, only a small subset of the data is verified.

Fourthly, the seed names used in Novelty Detection are selected manually by the authors, which might be biased.

5.2 Limitation

We assume that the natural language information in code is correct, but that might not always be true. For example, in the SecBench.js dataset, there is a code injection vulnerability with `code` as parameter name and `safe-eval` as function name. Despite the names and documentation stating that the code evaluation is safe, this is not the case. In these scenarios, our approach would assume that the natural language information is correct and thus classify this flow as expected, missing the vulnerability.

For the Novelty Detection method, users have to provide seed names for the OC-SVM, which might be difficult for some sinks if there is no clear definition of what names are expected. This is demonstrated in the path traversal sink in our experiment, which has the lowest F1-score out of all vulnerabilities.

6 Related Work

6.1 Neural Software Analysis

Neural software analysis [30] means applying machine learning techniques on source code. It becomes popular as machine learning is shown to excel in the field of natural language processing. Neural software analysis has three main characteristics: (1) fuzziness of available information, (2) lack of "well-defined correctness criterion", and (3) large amount of examples. Neural software analysis covers a wide range of applications, including automated program repair [15, 25, 11, 4], probabilistic type inference [31, 2, 26], and code completion [39, 9, 3, 22]. Our problem fits the three characteristics of neural software analysis. Similar to most other approaches, we make use of natural language info embedded in the source code. However, beyond extracting the source, sink, and metadata via taint analysis, our work does not care about the code syntactic structure. Deep learning-based vulnerability detection is a kind of neural software analysis in the security field [16, 23]. However, the accuracy is shown to be lacking in real world situation [8].

6.2 Natural Language Information in Program Analysis

There are previous works that also utilize implicit information embedded in code [1, 17]. In particular, our work is related to name-based program analysis, a type of neural software analysis that reasons about program via identifier names. Names are fuzzy by nature. As such, rule-based analysis cannot easily capture the meaning of names. Instead, machine learning is applied. An early adopter of name-based program analysis is DeepBugs [32], which trains a classifier to determine whether the code has bugs by looking at names. Nalin [28] finds name-value inconsistency by looking at whether the parameter names actually match the values it stores via dynamic analysis. Our approach differs from these previous works, as it is the first to combine natural language information with taint analysis.

6.3 Machine Learning-aided Static Analysis

Our work is related to machine learning-aided static analysis. Note that it is different from neural software analysis, as it is not an end-to-end neural analyzer. Merlin [24] automatically infers information flow specifications via probabilistic inference. Susi [33] and Seldon [12] automatically learns a taint specification via machine learning. USpec [14] performs unsupervised learning of API aliasing specifications. Our work is different as we focus on the natural language information

along with the code, while the previous works mainly focus on syntactic and semantic aspects of the code. One could also view our work as filtering warnings from static analysis [20]. As an example, a transformer-based learning approach can be used to identify false positive bug warnings, improving the precision of the static analyzer [21].

6.4 Word Embeddings

Word embeddings are used to represent text in machine learning models mathematically, usually as a vector [34]. Ideally, word embeddings would group similar words close together in the vector space, such that we can distinguish different words even as a vector. Common methods to train such embeddings include FastText [6], ELMo [35], word2vec [27] and GloVe [29]. However, these methods are meant for natural text. Our work targets variables identifiers used in programming, which are different to natural text [40]. To this end, we use VarCLR embeddings [10], which is specifically trained for identifiers used in programs. Specifically, our work is a downstream task of VarCLR, specifically we target names that are in JavaScript and predict if they are unexpected in different security-contexts.

7 Future Work

One problem in our work is the lack of data labels. We rely on manual labeling, which is prone to human errors and not scalable. This could be improved by automatic label collection. A possible way to do this is to record the parameter name and metadata when a vulnerability is recorded in advisory or vulnerability databases, or we can simply crawl these databases.

An obvious extension to our approach is to extend to other programming languages and other vulnerabilities. This should be possible as long as the names are in English, which is a natural language and does not vary in different programming languages. Our embedding also normalizes the names in source code, such that it can generalize to different coding conventions, e.g., using underscore (`min_length`) is the same as using camel case (`minLength`). Our approach should be able to extend to other vulnerabilities as well, as long as the natural language information is a relevant part of the vulnerability. For example, natural language information is not relevant to the JavaScript vulnerability "access to let-bound variable in temporal dead zone"[1].

In our approach, we have not made use of some metadata, such as package description and README file. This is because they might not always contain information that is helpful to determining whether a flow (precisely, the parameter) is unexpected. In the worst case, they could be noisy and worsen the model. In the future, we could retrieve these metadata as well, and somehow filters the irrelevant noisy content. Note that this might be difficult for the Novelty Detection approach, as we have to come up with an "expected" documentations for these new metadata.

[1] https://codeql.github.com/codeql-query-help/javascript/js-variable-use-in-temporal-dead-zone/

8 Conclusion

To effectively exploit the implicit information in code, we introduce a bimodal taint analysis tool, Fluffy. Fluffy is a bi-modal taint analysis tool: the first approach to utilize identifier names through machine learning and logic-based taint analysis to find unusual flows. The first modality is *code*: Fluffy uses a mining analysis implemented in CodeQL to find examples of flows from parameters to vulnerable sinks. The second modality is *natural language*: Fluffy uses a machine learning model that, based on a corpus of such examples, learns how to distinguish unexpected flows from expected flows using natural language information. We present four different models with different trade-offs between the manual efforts required and accuracy. In our evaluation, Fluffy achieves a high F1-score on four common vulnerability types. We show empirical evidence that Fluffy works in practice: we apply Fluffy on a dataset with 131 previously known vulnerabilities, of which Fluffy is able to flag 117. In addition, Fluffy flags eleven previously unknown vulnerabilities in real life projects, of which six have been confirmed by the developers.

Bibliography

[1] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37, 2018.

[2] M. Allamanis, E. T. Barr, S. Ducousso, and Z. Gao. Typilus: Neural type hints. In *Proceedings of the 41st acm sigplan conference on programming language design and implementation*, pages 91–105, 2020.

[3] U. Alon, S. Brody, O. Levy, and E. Yahav. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*, 2018.

[4] B. Berabi, J. He, V. Raychev, and M. Vechev. Tfix: Learning to fix coding errors with a text-to-text transformer. In *International Conference on Machine Learning*, pages 780–791. PMLR, 2021.

[5] M. Bhuiyan, A. Srinivas, N. Vasilakis, M. Pradel, and C.-A. Staicu. SecBench.js: An Executable Security Benchmark Suite for Server-Side JavaScript, 2022.

[6] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov. Enriching word vectors with subword information. *Transactions of the association for computational linguistics*, 5:135–146, 2017.

[7] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[8] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering*, 2021.

[9] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[10] Q. Chen, J. Lacomis, E. J. Schwartz, G. Neubig, B. Vasilescu, and C. Le Goues. Varclr: Variable semantic representation pre-training via contrastive learning. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 2327–2339. IEEE, 2022.

[11] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, 47(9):1943–1959, 2019.

[12] V. Chibotaru, B. Bichsel, V. Raychev, and M. Vechev. Scalable taint specification inference with big code. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 760–774, 2019.

[13] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[14] J. Eberhardt, S. Steffen, V. Raychev, and M. Vechev. Unsupervised learning of api aliasing specifications. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 745–759, 2019.

[15] C. L. Goues, M. Pradel, and A. Roychoudhury. Automated program repair. *Communications of the ACM*, 62(12):56–65, 2019.

[16] J. A. Harer, L. Y. Kim, R. L. Russell, O. Ozdemir, L. R. Kosta, A. Rangamani, L. H. Hamilton, G. I. Centeno, J. R. Key, P. M. Ellingwood, et al. Automated software vulnerability detection with machine learning. *arXiv preprint arXiv:1803.04497*, 2018.

[17] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu. On the naturalness of software. *Communications of the ACM*, 59(5):122–131, 2016.

[18] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[19] V. Hodge and J. Austin. A survey of outlier detection methodologies. *Artificial intelligence review*, 22(2):85–126, 2004.

[20] H. J. Kang, K. L. Aw, and D. Lo. Detecting false alarms from automatic static analysis tools: How far are we? *arXiv preprint arXiv:2202.05982*, 2022.

[21] A. Kharkar, R. Z. Moghaddam, M. Jin, X. Liu, X. Shi, C. Clement, and N. Sundaresan. Learning to reduce false positives in analytic bug detectors. *arXiv preprint arXiv:2203.09907*, 2022.

[22] S. Kim, J. Zhao, Y. Tian, and S. Chandra. Code prediction by feeding trees to transformers. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 150–162. IEEE, 2021.

[23] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018.

[24] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee. Merlin: Specification inference for explicit information flow problems. *ACM Sigplan Notices*, 44(6):75–86, 2009.

[25] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan. Coconut: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, pages 101–114, 2020.

[26] R. S. Malik, J. Patra, and M. Pradel. Nl2type: inferring javascript function types from natural language information. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 304–315. IEEE, 2019.

[27] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[28] J. Patra and M. Pradel. Nalin: Learning from runtime behavior to find name-value inconsistencies in jupyter notebooks. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 1469–1481. IEEE, 2022.

[29] J. Pennington, R. Socher, and C. D. Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[30] M. Pradel and S. Chandra. Neural software analysis. *Communications of the ACM*, 65(1):86–96, 2021.

[31] M. Pradel, G. Gousios, J. Liu, and S. Chandra. Typewriter: Neural type prediction with search-based validation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 209–220, 2020.

[32] M. Pradel and K. Sen. Deepbugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–25, 2018.

[33] S. Rasthofer, S. Arzt, and E. Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *NDSS*, volume 14, page 1125, 2014.

[34] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Neurocomputing: Foundations of research, 1988.

[35] J. Sarzynska-Wawer, A. Wawer, A. Pawlak, J. Szymanowska, I. Stefaniak, M. Jarkiewicz, and L. Okruszek. Detecting formal thought disorder by deep contextualized word representations. *Psychiatry Research*, 304:114135, 2021.

[36] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[37] B. Schölkopf, R. C. Williamson, A. Smola, J. Shawe-Taylor, and J. Platt. Support vector method for novelty detection. *Advances in neural information processing systems*, 12, 1999.

[38] C.-A. Staicu, M. T. Torp, M. Schäfer, A. Møller, and M. Pradel. Extracting taint specifications for javascript libraries. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 198–209, 2020.

[39] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1433–1443, 2020.

[40] Y. Wainakh, M. Rauf, and M. Pradel. Idbench: Evaluating semantic representations of identifier names in source code. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 562–573. IEEE, 2021.

[41] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel. Small world with high risks: A study of security threats in the npm ecosystem. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 995–1010, 2019.