Beware of the Unexpected:
Bimodal Taint Analysis

Wai Chow ¹, Max Schäfer ², Michael Pradel ¹

¹ University of Stuttgart, ² GitHub
Motivation

- Static analysis: Only as good as its specs
- E.g., taint analysis
  - Need policy that specifies insecure source-sink pairs
  - Problematic flow if both
    - data flows from source to sink and
    - the flow is unexpected by developers
Example: Command Injection

Want: Untrusted data does not flow to `exec()`
(Otherwise, command injection vulnerability)
Example: Command Injection

Want: Untrusted data does not flow to `exec()`
(Otherwise, command injection vulnerability)
Example: Command Injection

Want: Untrusted data does not flow to `exec()`
(Otherwise, command injection vulnerability)
Example: Command Injection

Want: Untrusted data does not flow to `exec()`
(Otherwise, command injection vulnerability)

Client code

```
execaCommand(command)
```

Library “execa”

```
exec(...)
```

Expected → No need to warn developer
Example: Command Injection

Want: Untrusted data does not flow to `exec()`
(Otherwise, command injection vulnerability)
Example: Command Injection

Want: Untrusted data does not flow to `exec()`
(Otherwise, command injection vulnerability)

Client code

```
locale(name)
```

Library “moment”

```
exec(...)
```

Unexpected $\rightarrow$ Should warn developer
This Talk

Bimodal program analysis

Program analysis: Reason about PL semantics

Machine learning: Reason about NL embedded in code
This Talk

Bimodal program analysis

Program analysis:
Reason about PL semantics

Machine learning:
Reason about NL embedded in code

Overapproximate relevant flows

Fluffy = Flagging unexpected flows for better security

Taint analysis

Identify unexpected flows
Overview

Corpus of projects

Pre-trained models of code

Taint query

5x

Source-to-sink flows

Traditional taint analysis (CodeQL)

Mining and labeling

Machine learning

4x

Expected flows

Unexpected flows
Overview

Corpus of projects

Pre-trained models of code

Taint query

Mining and labeling

Integrity:
- Command injection
- Code injection
- Reflected XSS
- Path traversal

Confidentiality:
- Clear-text logging

Expected flows

Unexpected flows

Traditional taint analysis (CodeQL)

Machine learning

Integrity:

Confidentiality:
Overview

Corpus of projects → Taint query → Mining and labeling

Integrity:
- Command injection
- Code injection
- Reflected XSS
- Path traversal

Confidentiality:
- Clear-text logging

Pre-trained models of code → Taint analysis (CodeQL) → Expected flows
→ Machine learning → Unexpected flows

Integrity:
- Command injection
- Code injection
- Reflected XSS
- Path traversal

Confidentiality:
- Clear-text logging
Overview

Corpus of projects

Pre-trained models of code

Taint query

5x

Traditional taint analysis (CodeQL)

Source-to-sink flows

Mining and labeling

Machine learning

4x

Expected flows

Unexpected flows
Overview

Corpus of projects

Pre-trained models of code

Taint query 5x

Source-to-sink flows

Mining and labeling

Machine learning

Four approaches:
- Binary classification
- Sink prediction
- Novelty detection
- Large language model (Codex)

Expected flows

Unexpected flows

4x

2x
Overview

Corpus of projects

Pre-trained models of code

Source-to-sink flows

Taint query 5x

Mining and labeling

Machine learning

Four approaches:
- Binary classification
- Sink prediction
- Novelty detection
- Large language model (Codex)

Traditional taint analysis (CodeQL)

4x

Expected flows

Unexpected flows

***Four approaches:***
- Binary classification
- Sink prediction
- Novelty detection
- Large language model (Codex)
Approach 1: Binary Classification

Goal: Predict whether a flow is expected

\[ M : N \times N_{fct} \times D \rightarrow \{ \text{Expected, Unexpected} \} \]
Approach 1: Binary Classification

Goal: Predict whether a flow is expected

\[ M : N \times N_{fct} \times D \rightarrow \{ \text{Expected, Unexpected} \} \]

- Name of the source (e.g., parameter)
- Name of the API function
- Documentation of the API function
Approach 1: Binary Classification

Goal: Predict whether a flow is expected

\[ M : N \times N_{fct} \times D \rightarrow \{ \text{Expected}, \text{Unexpected} \} \]

Model:
- Bi-directional RNN with LSTMs
- Input tokens embedded with pre-trained model
- Training data: 1,398 manually labeled examples (total across five taint queries)
Approach 3: Novelty Detection

- **Goal:** Predict whether a source/sink is unusual
- **One-class support vector machine** applied to embedded names of source/sink
Approach 3: Novelty Detection

- **Goal:** Predict whether a source/sink is unusual
- **One-class support vector machine** applied to embedded names of source/sink

### Integrity (names expected to flow to sink):

<table>
<thead>
<tr>
<th>Sink type</th>
<th>Seed names</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command injection</td>
<td>execute, command</td>
</tr>
<tr>
<td>Code injection</td>
<td>eval, execute, compile, render, callback, function, fn</td>
</tr>
<tr>
<td>Reflected XSS</td>
<td>sent, content</td>
</tr>
<tr>
<td>Path traversal</td>
<td>file, directory, path, cwd, source, input</td>
</tr>
</tbody>
</table>

### Confidentiality (names not expected to flow to sink):

- **Clear-text logging**
  - authkey, password, passcode, passphrase
Evaluation

- **Datasets**
  - 250k JavaScript projects → 7.5M taint flows
  - SecBench.js [ICSE’23] → 131 known vulnerabilities

- **Baselines**
  - Simple, frequency-based approach
  - Regular expressions
Evaluation

- **Datasets**
  - 250k JavaScript projects → 7.5M taint flows
  - SecBench.js [ICSE’23] → 131 known vulnerabilities

- **Baselines**
  - Simple, frequency-based approach
  - Regular expressions

- 1,398 manually labeled flows
- Validated by four external experts ($\alpha = 0.74$)
Effectiveness

How effective is Fluffy at identifying unexpected flows?

- 81%–97% precision and 80%–100% recall
- 117/131 known vulnerabilities found
Effectiveness

How effective is Fluffy at identifying unexpected flows?

- 81%–97% precision and 80%–100% recall
- 117/131 known vulnerabilities found
Effectiveness

How effective is Fluffy at identifying unexpected flows?

- 81%–97% precision and 80%–100% recall
- 117/131 known vulnerabilities found

Effectiveness varies depending on taint query and ML model.
Effectiveness

How effective is Fluffy at identifying unexpected flows?

- 81%–97% precision and 80%–100% recall
- 117/131 known vulnerabilities found
Effectiveness

How effective is Fluffy at identifying unexpected flows?

- 81%–97% precision and 80%–100% recall
- 117/131 known vulnerabilities found

Fluffy outperforms the baseline
Real-World Vulnerabilities

- Found and reported **17 previously unknown vulnerabilities**
  - 10/17 confirmed and fixed so far
- Example: CVE-2022-24785 in Moment.js

```javascript
locale(name)
```
Key Take-Aways

- **Bimodal program analysis**
  - Program analysis: Reason about PL semantics
  - Machine learning: Reason about NL embedded in code

- **Concrete application: Detecting unexpected taint flows**
  - Five kinds of vulnerabilities, four machine learning models
  - 81%–97% precision, 80%–100% recall
  - https://github.com/sola-st/fluffy