# 警惕意外：双模污点分析

Yiu Wai Chow 斯图
加特大学 斯图加特，德国
victorc
wai@gmail.com

Max Schäfer GitHub
牛津，英国
m
ax-schaefer@github.com

Michael Pradel 斯图加特
大学 斯图加特，德国
mic
hael@binaervarianz.de

arXiv:2301.10545v1 [cs.SE] 25 Jan 2023

## 摘要

静态分析是检测安全漏洞和其他编程问题的有力工具。尤其是全局污点跟踪能够发现由多个函数间复杂数据流引发的漏洞。然而，要精确识别哪些数据流存在问题颇具挑战，有时还取决于纯程序分析能力范围之外的因素，例如约定和非正式知识。举例来说，得知某个 API 函数 locale 的参数名称最终会进入文件路径，令人意外且可能存在问题。相比之下，传递给 API 函数 execaCommand 的参数 command 最终被解释为操作系统命令的一部分，则完全不足为奇。本文介绍了 Fluffy，这是一种双模污点分析方法，将推理数据流的静态分析与通过概率判断哪些数据流可能存在问题的机器学习相结合。其核心思路是让机器学习模型根据污点流中涉及的自然语言信息（例如 API 名称）预测该数据流是预期 还是意外，并且只将后者告知开发人员。我们提出了一个通用框架，并通过四种学习得到的模型对其进行实例化；这些模型在训练数据标注需求与预测准确率之间提供了不同的权衡。我们基于 CodeQL 分析框架实现了 Fluffy，并将其应用于 25 万个 JavaScript 项目。在五种常见漏洞类型上进行评估后，我们发现，Fluffy 在多种数据集中的四种类型上均取得了至少 0.85 的 F1 分数。

**ACM 引用格式：**
Yiu Wai Chow、Max Schäfer 和 Michael Pradel。2023。《当心意外：双模态污点分析》。载于*ACM SIGSOFT* 国际软件测试与分析研讨会论文集（*ISSTA 2023*）。ACM，纽约，纽约州，美国，12 页。
https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 引言

污点分析是一种强大的技术，可用于检测各种编程错误，包括安全漏洞和其他类型的程序错误。污点分析会追踪信息从 源 流向汇点 的过程；如果这种流违反了某项预定义策略，就会发出警告。例如，此类策略可能规定，来自未知且可能怀有恶意的用户发送的 HTTP 请求的数据（源）不得流入用于将字符串作为操作系统命令执行的 API（汇点），以防止远程代码执行，这是一种称为命令注入的漏洞。再举一例，某项策略可能规定，未加密密码或其他机密信息不得流入日志语句，以防止通过明文日志记录意外泄露这些信息。污点分析是热门静态分析解决方案的重要组成部分，例如 Facebook Infer 或 GitHub Code Scanning，它们会定期扫描大量专有软件和开源软件。

研究人员和工程师在开发适用于各种语言的精确、可扩展的跨过程污点跟踪算法方面投入了大量工作，确保现代污点分析能够扩展到最大的代码库。然而，归根结底，污点分析的效果取决于其所执行的策略：如果缺少某个源或汇点，分析就不会报告涉及它的有问题的数据流，从而产生假阴性，导致漏洞被遗漏。另一方面，如果纳入了不应包含的源或汇点，分析就会报告假阳性，导致开发人员和安全研究人员做不必要的工作并感到沮丧。

在某些情况下，例如上文的命令注入示例，可以通过仔细检查被分析库的 API 来确定合适的源和汇。这个过程定义明确（尽管可能相当繁琐），但由于需要大量人工投入，只能覆盖有限的一组 API。另一方面，对于明文日志记录，解决方案就不那么显而易见了。虽然识别汇（对日志记录库的调用）很容易，但源的定义则远不那么明确，因为敏感数据的存储和处理方式通常高度取决于具体应用程序。

再举一个例子，考虑如何检测“不卫生”的 API，也就是那些在安全敏感上下文中使用客户端应用程序数据，却没有明确告知客户端这一点的库 API（例如通过恰当的命名或文档说明）。根据这种安全敏感用法是否出于有意，此类 API 要么存在漏洞，要么文档说明不足，而这两种情况都会使客户端应用程序面临风险。举例来说，用于处理时间和日期的 JavaScript 库 moment 的 API 中有一个区域设置函数，用于设置美化输出时所使用的区域设置。在 2.29.2 版本之前，该函数对其参数名称（要使用的区域设置名称）的处理并不安全；如果向其传入不可信数据，就会导致远程代码执行。[1] 函数或参数的名称都没有任何迹象表明该参数会直接影响所执行的命令；事实上，这种行为并非有意为之，而是草率编码实践的后果。依赖 API 名称和文档说明的客户端可能会将从不可信第三方获得的数据传给该 API，从而使自己面临

允许免费制作本作品全部或部分内容的数字版或纸质副本，供个人或课堂使用，但不得为牟利或商业利益而制作或分发副本，且副本首页须附有本声明和完整引文。对于由 ACM 以外的其他方拥有版权的作品组件，必须尊重其版权。注明出处后允许摘录。若要以其他方式复制、重新出版、发布到服务器或分发给他人，须事先获得明确许可和/或支付费用。请通过 permissions@acm.org 申请许可。*ISSTA 2023*，*2023*年*7*月*17*日至*21*日，美国西雅图© 2023 计算机协会。ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1] CVE-2022-24785。

漏洞风险——事实上，在 moment 开发人员修复这一问题之前，确实有一些客户端因此受到影响。[2]

识别这类不卫生 API 似乎很适合采用污点分析，但虽然汇点很容易定义（它们与命令注入的汇点相同），源的集合却较难确定。仅将所有 API 入口点都视为源并不可行：例如，库 execa 提供了更完善的跨平台支持，可用于在 Node.js 上生成子进程；它导出了函数 execaCommand，其参数 command 会流入一个命令注入汇点。然而，将这种数据流标记为漏洞毫无意义，因为这恰恰是预期功能。

这两个例子的共同点是，我们要寻找的数据流在某种意义上是出乎意料的，因为它们与开发人员的意图和预期相悖：在生产代码库中，人们不会期望敏感信息以明文形式记录在日志中；同样，人们也不会期望某个库在安全敏感上下文中使用数据，却不明确告知客户端这一情况。

为了解决检测此类意外污点流的问题，我们提出了 Fluffy（“标记意外数据流，提升安全性”），一种双模污点分析，它利用了这样一个事实：源代码通过编程语言语义以及嵌入代码中的自然语言信息（例如标识符名称）[1, 26, 45]来传达含义。Fluffy 支持实施安全策略，其中源集合部分依据自然语言信息确定。[3]

该方法包含两个步骤。首先，我们运行现成的单模态污点分析，以过度近似源集合，使其能够完全通过代码结构来描述，而无需进行任何自然语言推理。例如，对于明文日志记录，我们将所有变量视为源；对于不卫生 API，则将所有参数视为源。其次，使用机器学习模型对第一步得到的候选数据流进行筛选，预测候选数据流中的源是否确实是真正的源。我们的方法结合了基于逻辑的静态污点分析和机器学习模型的优势：前者推理数据流和控制流依赖，后者则以概率方式“理解”自然语言信息的含义。

Fluffy 方法是一个通用框架，包含四种具体实现，它们采用不同方式构建预测任务、基于不同的机器学习模型，并对人工标注数据提出不同要求。这四种实现分别是：(i) 使用人工标注样例训练的神经网络二元分类器；(ii) 使用通过静态分析自动提取的数百万条污点流训练的神经模型；(iii) 基于单类支持向量机的新颖性检测技术；以及 (iv) 通过小样本学习查询大语言模型（Codex [14]）的技术。

我们的工作与将程序分析和机器学习模型相结合来解决其他分析问题的研究相关，例如调用图剪枝 [33, 51] [30] 和过滤空指针解引用警告。Fluffy 则不同，它解决的是识别意外污点数据流的问题。另一项相关研究是纯神经网络污点分析 [50], , 它训练神经模型，以比实际动态分析更高效的方式“模拟”动态污点分析。相比之下，我们的工作利用静态分析和机器学习的互补优势。最后，Fluffy 也与此前使用训练好的模型自动将 API 标注为源和汇的研究有关 [48]。这项研究依赖一组手工编码特征，例如方法是否返回特定类型，并对源 API 和汇 API 作出预测。相比之下，我们利用预训练神经网络模型，避免手动进行特征工程，并提出能够预测已识别源和汇之间特定数据流的模型。总之，我们的工作首次针对污点跟踪问题提出了双模态分析。

我们在 CodeQL 分析框架中实现了该方法 [22]，并将其应用于 25 万个 JavaScript 和 TypeScript 项目。我们的评估考察了五种污点分析，旨在检测完整性问题（例如命令注入）和机密性问题（例如明文日志记录）。结果表明，Fluffy 能有效识别意外的污点流，根据所用分析的不同，其精确率为 81%–97%，召回率为 80%–100%，F1 分数为 76%–97%。为评估该方法的实用性，我们将 Fluffy 应用于过去已确认的 131 个漏洞，并成功检测出其中 117 个。最后，我们向相应的开发人员报告了 16 个新发现的漏洞；截至目前，他们已确认其中 8 个。

总之，本文的贡献如下：

- 一种双模污点分析方法，结合静态分析与机器学习来识别有问题的数据流。
- 一种包含四种实例化方案的通用框架，可在标注工作量与预测准确率之间进行权衡。
- 将该方法集成到 CodeQL 中，并通过实证证明其在实践中有效且实用。

为促进未来研究，我们公开了实现代码和实验结果。[4]

## 2 背景：使用 CodeQL 进行污点分析

CodeQL [22] 是一种开源静态程序分析系统，为 LGTM.com 和 GitHub CodeScanning 提供支持。遵循代码即数据范式，代码被表示为关系数据，分析则以 QL 查询的形式表达。QL [5], 是一种面向对象的 Datalog 扩展。CodeQL 提供了丰富的标准库，用于执行常见的分析任务，其中包括全局污点分析框架，该框架为一系列用于发现常见漏洞的独立分析提供支持，例如命令注入或路径遍历。

安全策略也以 QL 表达，通常会将特定 API 的使用识别为污点源和汇点。例如，在 JavaScript 中，用于识别命令注入漏洞的 CodeQL 查询[5] 会将各种可能包含不可信用户输入的来源（例如通过 express npm 包访问的 HTTP 请求参数）识别为源，并将对命令执行 API 的调用（例如 Node.js

[2]例如，请参阅安全公告 GHSA-7v28-g2pq-ggg8。[3]汇点的情况与此对称，但本文仅关注源。

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

子_进程模块中的调用）识别为汇点。路径遍历查询[6] 使用相同的源集合，但将文件系统操作作为汇点。

这两者都是完整性违规的例子，即在安全敏感上下文中使用了不可信数据。CodeQL 查询还涵盖了其他此类违规，例如代码注入，[7] 即使用不可信数据构造要执行的代码；以及反射型跨站脚本（简称“反射型 XSS”），[8] 即将不可信数据嵌入 HTTP 响应中。

如上所述，有些分析并不适合采用这种表述方式。一个例子是查找敏感信息明文日志记录的问题，这是一种机密性违规，其中安全敏感数据暴露在不可信上下文中。用于查找明文日志记录的 CodeQL 查询[9] 采用手工编写的正则表达式，来识别名称暗示其可能包含敏感数据的变量和属性，并将它们视为分析的源。正如我们将在下文展示的，这种方法容易出错；每当发现多余或缺失的源时，都需要手动更新库。

另一个例子是，CodeQL 查询由库输入构造的不安全 *shell* 命令[10] 尝试通过检测从 API 参数到 shell 命令的污点流，识别不卫生的库 API，同时排除名称暗示其用途是命令的参数。同样，这里也使用正则表达式来实现这一目的，并存在上述相同的缺点。

最近，CodeQL 增加了实验性支持，可利用机器学习技术发现标准安全策略 [21]未涵盖的新源和汇。虽然这有助于在不常用或专有 API 中识别源和汇，但在需要自然语言信息的场景下是否有任何优势尚不明确，因为目前该功能仅适用于四个查询，而这些查询均不属于这一类别。

## 3 方法

本节将更详细地介绍我们的双模污点分析 Fluffy。在定义该方法所要解决的问题（第 3.1 节）之后，第 3.2 节将概述我们的通用框架。接下来的小节将介绍该框架的四种实例化形式，探讨设计空间中的不同权衡。

### 3.1 问题陈述

Fluffy 的输入是一个污点式安全策略 P，以及一个需要根据该策略进行检查的代码库 $C$。该安全策略指定了一组有问题的数据流，对于 $C$，这些数据流以源-汇对的形式表示（$l_{src}$，$l_{sink}$）；在 $C$中，数据不应从源位置 $l_{src}$ 传播到汇位置 $l_{sink}$。

此规范由两个部分组成：$\mathrm{P}_F$ 和 $\mathrm{P}_U$。第一部分 $\mathrm{P}_F$ 定义得精确但过度近似，用于确定一组

[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.

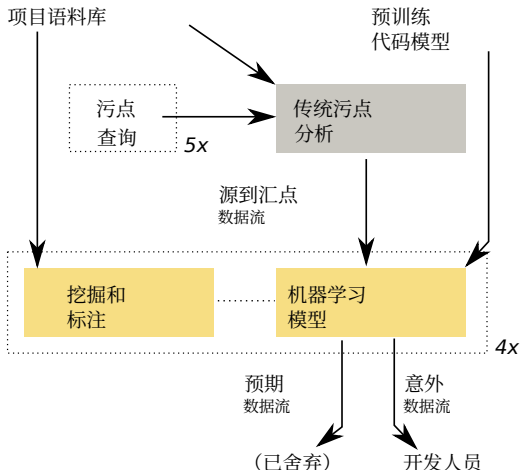



**图1：方法概览。黄色组件是本文的贡献。**

$F$潜在问题数据流，这些数据流可仅根据代码结构计算得出，例如通过标准污点分析。第二部分 $\mathrm{P}_U$的定义不精确，用于描述一组 $U$ 意外数据流。判断某个数据流是否意外，本质上是一个模糊的问题，因为开发人员的预期取决于常见的编码惯例和非正式信息，例如标识符名称和 API 文档。

本文要解决的问题是，识别给定代码库中满足安全策略两个部分的所有数据流，即找出集合 $\{(l_{src}, l_{sink}) \mid l_{src} \in C, l_{sink} \in C, (l_{src}, l_{sink}) \in F \cap U\}$ 中实际存在问题的数据流，并将其呈现给开发人员。

### 3.2 概述

鉴于该问题具有模糊性，我们采用基于学习的技术来解决它。已有研究表明，这类技术能够有效捕捉内在模糊的信息 [45]。图1概述了该方法。与问题的两部分规范相对应，该方法包含两个主要步骤。给定一个待分析的项目，第一步是应用传统污点分析。我们以 CodeQL 为基础，它提供静态代码分析引擎和一套标准污点分析，如第2节所述。污点分析 $T\colon C \mapsto F$识别给定代码库中源-汇对之间的所有数据流 $C$，这些数据流属于 $F$。第二步是将污点分析检测到的源-汇数据流交给机器学习模型 $M\colon F \mapsto U$，由模型预测 $F$中的哪些数据流可能会让开发人员感到意外，因此应报告给开发人员。该机器学习模型基于现有的预训练模型构建，例如，这些模型可以将代码词元嵌入连续向量；同时，模型还依托挖掘与标注组件，该组件提供用于学习的训练数据。

图1所述的整体框架可以通过多种方式实例化。其中一个维度是所采用的污点分析；我们重点关注针对 JavaScript 的五种 CodeQL 分析：代码注入、命令注入、反射型跨站脚本攻击、路径遍历和明文日志记录。另一个维度是所采用的机器学习模型；我们介绍了四种模型，以探索设计空间中的不同权衡。

**表 1：通用框架的四种实例化。**

| 方法 | 训练数据 | 模型 |
|---|---|---|
| 二元分类 | 数百条人工标注的流 | 神经网络二元分类器，预测一条流是否符合预期 |
| 汇点预测 | 数百万条未标注的流 | 神经模型，预测最可能的汇点 |
| 新颖性检测 | 每种汇类型的相关术语少于十个 | 单类支持向量机，用于识别异常数据流 |
| Codex 语言模型 | 提示中给出十个示例 | 生成式语言模型，用于预测某个数据流是否符合预期 |

表1概述了这四种机器学习模型，具体内容详见第 3.3 节和第 3.4 节。

## 3.3 收集和表示自然语言信息

对代码库运行上述查询，可得到一组 $F$ 数据流，即源-汇对，其中数据从源传播到汇点。为了判断某个数据流是否可能令开发人员感到意外，Fluffy 会利用与该数据流相关的自然语言信息。具体而言，该方法会收集每个数据流的以下信息：

- 源的标识符名称 $n_s \in N$；对于完整性查询，源是 API 函数 $f$ 的参数 $p$；对于机密性查询，源是一个源变量 $v$。
- 此外，对于完整性查询，还会收集 $f$ 的名称 $n_f \in N$ $\subseteq$ 函数 以及关于 $p$ 的任何文档。 $N$ $d \in D$

为了让机器学习模型能够有效地推理提取出的自然语言信息，我们将其嵌入为连续向量表示。为此，Fluffy 使用一个预训练嵌入函数 $e: N \to \mathbb{R}^k$，它将 $N$ 中的自然语言字符串映射为一个 $k$ 维向量，使语义相似的字符串具有相似的向量表示。我们使用 VarCLR [15], 作为嵌入函数，原因有二。首先，IdBench 基准测试 [53] 的结果表明，在将标识符名称映射为能够保留语义相关性和相似性的连续向量这一任务上，VarCLR 达到了最先进水平。其次，在初步实验中将 VarCLR 与其他预训练嵌入（例如 FastText [9], ）进行比较时，我们发现 VarCLR 的总体效果最佳。

## 3.4 预测意外数据流

基于传统污点分析提取的流 $F$ 以及与每个流相关联的自然语言信息，Fluffy 的核心步骤是确定意外数据流的子集 $U \cap F$，并将其报告给开发人员。下文介绍了四种用于此目的的基于机器学习的技术。这些技术在所需带标签训练数据的类型和数量、所采用的模型类型，以及（如第 4 节所示）有效识别意外数据流的能力方面各不相同。

*3.4.1 方法 1*：二元分类。我们训练了一个神经模型 $M$，用于预测给定流属于意外情况的概率。模型以源的标识符名称的嵌入作为输入；如果有的话，还会输入与该流相关的其他自然语言信息。也就是说，该模型是通过学习得到的以下函数：

$$M: \quad N \times Nfct \times D \to \{\text{预期，意外}\}$$

该模型采用标准的前馈神经网络实现，包含两个隐藏层，并在输出层应用 softmax 函数；我们使用交叉熵损失对其进行训练。我们使用嵌入函数 $e$ 对名称 $n_s$ 和 $^{n}f$ 进行嵌入。对于文档 $d$，我们使用联合训练的嵌入层对每个词元进行嵌入，并使用基于 LSTM 的双向循环神经网络对嵌入后的词元序列进行编码。我们针对五个污点查询分别训练一个模型，使用数百条人工标注的流（详见第 4 节）。

*3.4.2 方法 2*：汇点预测。二元分类方法需要大量人工标注的训练数据。为避免这项人工投入，以下方法仅使用自动挖掘的训练数据进行训练。其基本思路是训练一个神经分类模型，针对给定的源预测它应当流向哪个汇点。

F对于完整性查询，这对应于函数 tion

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

其中，前四个输出类别对应四个完整性查询的汇点，而 *None* 表示源位置不会传播到这四个汇点中的任何一个。同样，对于机密性查询，该模型是以下学习得到的函数：

$$M: N \to \{Logging, None\}$$

其中，日志记录 表示记录源内容没有问题，而 *None* 表示不应记录这些内容。

该模型采用与二元分类相同的架构，但其输出层长度适合分类问题。训练数据通过 CodeQL 收集，如第 4.1.1 节所述，无需人工标注。

模型训练完成后，Fluffy 会向模型查询此前未见过的数据流，以判断该流是否意外。如果该流涉及汇类型 $s \neq None$，且模型预测 $s$ 是最有可能的汇，那么我们就认为该流是预期的。其理由是，如果具有特定名称的源通常会流向此汇类型，那么开发人员很可能对此有所了解。否则，如果模型预测的是其他汇 $s\prime \neq s$（其中 $s\prime$ 可能为 *None*），我们就认为该流是意外的，并将其报告出来。

*3.4.3 方法 3*：新颖性检测。前两种方法要求训练数据已标注为属于不同的类别。相比之下，我们的第三种方法使用仅基于某一类别示例训练的新颖性检测技术。我们使用单类支持向量机（单类支持向量机）[49]，预测之前未见过的示例是否属于该类别，或者是否为“新颖”示例。

**表 2：用于新颖性检测的种子名称。**

| 汇类型 | 种子名称 |
|---|---|
| 完整性： | |
| 命令注入 | 执行，命令 |
| 代码注入 | 求值，执行，编译，渲染，回调、函数、函数 |
| 反射型跨站脚本攻击 | 已发送、内容 |
| 路径遍历 | 文件，目录，路径，cwd，源，输入 |
| 机密性： | |
| 明文日志记录 | 身份验证密钥，密码，密码，口令 |

对于每种汇点类型，我们使用预期“种子”name 的示例训练一个单类支持向量机模型，如表 2 所示；这些示例是根据我们对该领域的理解选取的。对于完整性查询，这些 name 表示我们预期会流向特定汇点的数据；而对于机密性查询，它们表示不预期流向该汇点的敏感数据。由于 name 集合很小（每种汇点类型少于十个示例），获取这些训练数据所需的工作相对较少。为了训练模型，我们使用预训练嵌入 $e$对自然语言信息进行嵌入。模型训练完成后，我们让模型预测之前未见过的流是否为意外流；同样，我们使用 $e$对自然语言信息进行嵌入。

*3.4.4 方法 4*：*Codex*语言模型。受 Codex [14], 等大语言模型在各类软件工程任务 [6, 27, 29, 30], 中取得的出色成果启发，我们也实现了一种基于 Codex 的方法。为了查询语言模型，我们设计了一个提示，其中提供有关流的信息，并以一条要求模型补全的注释结尾。按照少样本方法 [10], ，提示中包含十个我们希望模型执行的任务示例，并以第十一个未完成的示例结尾，由模型补全。

对于完整性查询，提示由以下内容组成：

- API 函数的签名，以及与其关联的文档注释（如果有）。
- 一条 JavaScript 注释，内容为：“在上述函数 $f$ 中，参数 $p$ 流入了 $s$ 汇点（$e$），它是 $c$”，其中– $f$是函数的名称，– $p$ 是形式参数的名称，– $s$ 是查询的缩写名称，例如“CommandIn- jection”，– $e$ 是对查询的简要说明，例如“在路径表达式中使用了不受控的数据”，而– $c$ 是“预期”或“意外”。

对于机密性查询，提示由以下内容组成：

- 一个桩函数 $f$，它对参数调用 console.log $p$。
- 一条 JavaScript 注释，内容为：“在上面的函数 $f$ 中，参数 $p$ 正在被记录到日志中，这可能会暴露 $c$ 数据”，其中 $c$ 可以是“敏感”或“不敏感”。

对于前十个少样本示例，$c$ 中会提供这个词；对于第十一个示例，则不提供。随后，模型通过预测缺失的词来补全提示。对于完整性查询，我们会检查根据模型判断，“预期”还是“意外”更可能是补全结果；只有当模型预测数据流是意外的时，我们才会向开发人员报告这些数据流。同样，对于机密性查询，只有当模型认为用“敏感数据”补全提示比用“不敏感数据”补全提示更可能时，我们才会向开发人员报告数据流。

## 4 评估

为了评估我们的方法的有效性和实用性，我们提出了以下研究问题：

**研究问题 1** Fluffy 在识别意外数据流方面有多有效？**研究问题 2** Fluffy 在发现漏洞方面有多有效？**研究问题3** 标注工作量与预测准确率之间需要如何权衡？**研究问题 4** Fluffy 扩展到大型代码库的效果如何？

现在，我们将介绍评估设置、方法和结果，并讨论这些内容如何回答我们的研究问题。

### 4.1 实验设置

*4.1.1* 数据收集。我们为评估收集了三个数据集。前两个（参数汇流和 日志流）是通过在 LGTM.com 上所有 JavaScript/TypeScript 项目中运行专用 CodeQL 查询编译而成的（撰写本文时约有 250,000 个项目），第三个数据集则源自 *SecBench.js* [7], ，这是一个包含服务器端 JavaScript 现实世界漏洞的语料库。

参数汇流。第一个数据集关注不卫生 API。我们希望收集从 API 参数流向已知汇点的示例。对于此数据集，API 参数是指 npm 包导出的函数的参数，或者（稍微宽泛地使用这个术语）此类参数的属性。我们的汇点取自四种 CodeQL 安全分析，这些分析可识别四种常见的完整性违规：代码注入、命令注入、反射型跨站脚本攻击和路径遍历。

针对每种完整性违规，我们都使用标准污点跟踪框架编写自定义 CodeQL 查询。这些查询会收集形如（$p$，$s$）的数据流元组，其中 $p$ 是一个 API 参数，$s$ 是前面提到的四种汇点类型之一；当污点从 $p$ 流入类型为 $s$的汇点时，就会收集该元组；如果未发现从 $p$ 流向任何已知汇点的数据流，则汇点类型为特殊类型 *None* 。[11] 除了 API 参数的名称，我们还会收集其文档注释，以及（如果有）其所属函数的文档注释。最后，我们会过滤掉参数名称少于两个字符的数据流，因为单字符名称不太可能向人类或模型传达多少语义信息。

总体而言，该查询在 61,123 个项目中发现了 3,245,860 条数据流，其中 3,228,034 条流向 无；1,123 条流向代码注入汇点；1,498 条流向命令注入汇点；70 条流向反射型 XSS 汇点；15,135 条流向路径遍历汇点。请注意，汇点类型的数量极不平衡，这是由项目的选择所导致的，而项目选择

[11]请注意，这意味着同一个参数 $p$可以对应两条数据流：（$p$，$s$）和（$p$，$s'$），且它们的汇点类型分别为 $s$和 $s'$；但前提是 $s$和 $s'$都不是*None*。

由 LGTM.com 决定，因此不受我们控制。过滤掉单字符名称后，移除了 233,091 条数据流，即过滤前所有数据流的 6.7%。

日志流。第二个数据集涉及明文日志记录。我们再次编写了一个自定义的 CodeQL 污点跟踪查询，这次用于查找从变量或属性 $v$ 流向日志记录函数调用的数据流；该函数由 CodeQL 标准库中现有的 API 建模确定。除了记录 $v$的名称外，我们还记录随 CodeQL 一同提供的明文日志分析是否认为 $v$ 可能包含敏感信息。与上述情况一样，我们会过滤掉单字符名称；这类名称占所有数据流的 8.9%。最终，我们在 112,765 个项目中获得了 4,535,851 条数据流。

*SecBench.js*。对于最后一个数据集，我们检查 SecBench.js 语料库中的所有代码注入、命令注入和路径注入漏洞，以找出由不卫生 API 导致的漏洞。过滤后，这三类漏洞的数量分别为 33、101 和 1。[12]

*4.1.2* 非神经网络基线。 除了 Fluffy 的四种基于神经网络的实现之外，我们还考虑了两种非神经网络基线方法进行比较。第一种基线称为频率计数，是一种统计方法，用于确定在参数-汇点流数据集中，给定参数名称流入特定汇类型的频率。如果该频率低于给定阈值，我们就认为该汇类型对于此参数而言是意外的。需要注意的是，这种方法不涉及对名称的语义推理，且只考虑名称完全匹配的情况。由于数据集的构成，这种方法仅适用于完整性违规，因为日志记录流数据集只考虑单一汇类型。

第二种基线称为正则表达式，用于日志记录流。这是一个现有的 CodeQL 查询（第 2 节），它使用正则表达式标记可能存在问题的数据流：首先应用两个正则表达式，识别可能表示敏感内容的名称，例如密码和身份验证密钥。[13] 然后，将匹配任一正则表达式的名称与第三个正则表达式进行匹配，以筛除那些表明该值已加密或经过哈希处理的名称。[14]

我们还考虑过将其与本研究之前已有的、用于查找机密性违规的 CodeQL 查询直接比较（第 2 节）。然而，这些查询回答的问题与 Fluffy 并不相同。已有查询查找从受污染数据的源到敏感汇的端到端数据流，而 Fluffy 处理的是从 API 参数到敏感汇的数据流。已有查询不会将这些 API 参数视为源（反过来，Fluffy 也不考虑来自非参数源的数据流），因此无法直接比较二者。

[12]我们不予考虑一个代码注入漏洞和两个命令注入漏洞，因为受影响的 npm 包已不再可用，无法确定涉及的参数。另有一个命令注入漏洞也不予考虑，因为相关安全公告已撤回，且其详细信息已无法获取。[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* 数据标注。为了在评估中获得真实标注，我们对收集到的数据进行子集标注，以指出该数据流是否可能导致或促成漏洞。

对于 param-sink 数据集，作者手动标注了 1,058 条数据流。为检验标注的可靠性，我们向四位具有安全相关分析经验的程序分析工程师发放了问卷。问卷请分析专家根据参数名称、函数名称、包名称、汇点、包描述、参数文档和函数文档，各自标注随机选取的 30 条数据流。具体而言，我们提出的问题是：“作为该包的使用者，您是否预期传入此参数的数据会流入相关汇点，且不会经过进一步净化？”我们使用克里彭多夫 α 系数计算评审者间一致性，结果显示，我们给出的标签与分析专家给出的标签之间的一致性分数为 0.74。也就是说，分析专家总体上认同我们真实标注中的标签。

标注后的数据流总集由两个相互重叠的子集组成。第一个是随机集，包含随机选取的 272 条数据流。尽管该集合没有偏差，但其中意外数据流并不多，因为在真实世界代码中观察到的大多数数据流都没有问题。因此，我们将随机集扩展为第二个集合，即平衡集，并从频数统计方法的结果中额外选取数据流纳入其中。

对于日志数据集，我们人工标注了其中 340 条数据流。表4显示了根据我们的标注，各汇类型对应的数据流总数和意外数据流数量。

请注意，我们的人工标注表明，如果简单地将所有数据流都标记为意外，将会产生许多假阳性，这说明了我们的方法为何重要。

最后，对于 SecBench.js 数据集，我们审查了所有数据流，并发现一个代码注入漏洞和七个命令注入漏洞。尽管这些漏洞已被报告为漏洞，但根据我们的判断，它们都是预期的。有关这些漏洞的完整详情，请参阅补充材料。举一个例子，SecBench.js 收录了 CVE-2020-7784，该漏洞指出，npm 包 ts-process-promises 导出的函数 exec 的参数 command 会被解释为 Shell 命令。我们认为，根据上下文，这条数据流显而易见，因此并非真正的漏洞。这些漏洞均未得到修复；其中一个案例的安全公告已被撤回，这表明相关软件包的作者并不认为这些问题构成问题。其余 127 个漏洞均被我们标注为意外。

*4.1.4* 模型超参数。我们的所有模型都使用 768 维向量，对应于 VarCLR 嵌入 [15]的输出维度。唯一的例外是 Codex，我们按照第 3.4.4 节所述，通过 HTTP API 间接调用它。在汇点预测和二元分类方法中，隐藏层维度分别为 500 和 250。我们使用学习率为 0.001 的 Adam优化器。对于汇点预测，训练批次大小为 256；当验证损失连续两个训练轮次未下降时，我们便提前停止训练。我们留出 10% 的训练数据作为验证集。对于二元分类，训练批次大小为 32；当验证损失连续 50 个训练轮次未下降时，模型便停止训练。我们使用

**表 3：不同方法的比较。**

| 方法 | 训练数据 | 模型数量 | 需要阈值 | F1 分数的计算 |
|---|---|---|---|---|
| 汇点预测 | 参数汇流 | One | Yes | 基于精确率-召回率曲线 |
| 新颖性检测 | 种子名称 | 每种汇类型一个 | Yes | 根据精确率-召回率曲线 |
| 二元分类 | 平衡集 | 每种汇类型一个 | No | 来自 k 折交叉验证 |
| Codex | 平衡集 | One | No | 直接 |
| 频率 | 参数汇流 | One | Yes | 来自精确率-召回率曲线 |

**表4：真实标签。**

| | 随机集 | | 平衡集 | |
|---|---|---|---|---|
| 完整性： | 意外 | 总计 | 意外 | 总计 |
| 代码注入 | 16 | 27 | 113 | 340 |
| 命令注入 | 15 | 29 | 144 | 168 |
| 反射型跨站脚本攻击 | 19 | 28 | 29 | 46 |
| 路径遍历 | 8 | 188 | 105 | 504 |
| 机密性： | | | 意外 | 总计 |
| 记录敏感数据 | | | 245 | 340 |

在新颖性检测方法中，我们使用径向基函数（RBF）作为单类支持向量机的核函数。超参数伽马和 nu 分别设为 0.05 和 0.01。对于 Codex 语言模型，我们将温度、频率惩罚和存在惩罚设为零。

## 4.2 RQ1：识别意外数据流

本节将说明各方法的评估过程，然后讨论参数汇和日志流的结果。我们将这些方法彼此进行比较，并与前文讨论的两个基线（频率计数和正则表达式）进行比较。

*4.2.1* 评估过程。我们使用 F1 分数作为主要指标，考察各方法对意外数据流的分类效果。表 3 比较了四种神经网络方法与频率计数基线的特征。对于所有需要阈值的方法，我们都会绘制精确率-召回率曲线（PR 曲线），以展示不同阈值下精确率和召回率之间的关系，并记录可达到的最佳 F1 分数。我们针对源-汇对计算精确率和召回率。

对于汇点预测，我们使用所有尚未标注的数据流训练模型，其中 90% 用于训练，10% 用于验证。已标注的真实标注用作测试集。由于模型输出的是概率，因此需要设置阈值才能作出预测。

对于新颖性检测，我们使用相应的种子名称，为每种汇类型训练一个单类支持向量机（OC-SVM）（即参数汇流训练四个，日志流训练一个）。我们再次使用真实标注作为测试集，其中可能包含涉及种子名称的数据流，因为这些名称是独立选取的。模型会为每条数据流输出一个分数，表示该数据流与单类支持向量机边界之间的距离；同样，我们需要设置一个阈值来得出预测结果。

对于二元分类，我们也针对每个汇点，使用平衡集中相关的数据流训练一个模型。由于模型会直接对数据流进行分类，因此无需设置阈值。我们使用五折交叉验证，并进行五次不同的 80/20 划分；对于每个模型，报告五折的精确率、召回率和 F1 分数的平均值。

对于 Codex，对于每条待评估的数据流，我们会从平衡集中随机抽取十个具有相同汇点的示例，放入提示中。我们确保要求它评估的数据流不在提示中嵌入的十个示例之列。由于该模型会直接分类判断数据流是否符合预期，因此我们同样不需要阈值，可以根据预测结果直接计算精确率、召回率和 F1 分数。

最后，频率计数与汇点预测相似，都是在所有参数汇流上训练，并且需要设置阈值。

*4.2.2* 参数汇流。对于参数汇流，四种方法均在随机集和平衡集上进行评估，并以频率计数作为基线。随机集的汇总结果如表 5 所示，每种汇类型的最佳分数以粗体标出。我们观察到，Fluffy 的方法在所有汇点上的表现都优于基线，其中新颖性检测总体表现最佳，其次是二元分类。这些方法在命令注入和反射型跨站脚本攻击上的表现尤其出色，F1 分数达到 0.9–1.0；在代码注入上的 F1 分数则约为 0.8。另一方面，这些方法在路径遍历上的效果明显较差，分数分布很广，最高仅为 0.45。这可能是因为随机集中只有八个意外路径遍历数据流的示例。

平衡集上的结果如表 6 所示，精确率-召回率曲线见图 2。在这种设置下，二元分类表现最佳，在代码注入和路径遍历方面大幅优于新颖性检测。这可能是因为平衡集包含更多样化的名称，使二元分类有机会从数据集中学习，而新颖性检测则依赖固定的一组种子名称。对于路径遍历汇点，所有方法在平衡集上的表现都优于随机集，这可能是因为其中意外情况多得多（105 个）。频率计数在命令注入方面表现非常出色，但对此应谨慎解读，因为平衡集的构建部分依据了该方法自身的结果。另一个有趣的发现是，汇点预测在平衡集的代码注入数据流上的表现远不如在随机集上的表现，F1 分数从 0.82 降至 0.53。这是因为精确率下降了：尽管它能够捕获大多数意外数据流（108 个），但也包含许多假阳性（194 个）。其余结果与随机集上的结果相似。

**表 5：Fluffy 对参数汇流随机集的有效性。**

| | 随机集 | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 代码注入 | | | 命令注入 | | | 反射型跨站脚本攻击 | | | 路径遍历 | | |
| 方法 | 精确率 | 召回率 | F1 分数 | 精确率 | 召回率 | F1 分数 | 精确率 | 召回率 | F1 分数 | 精确率 | 召回率 | F1 分数 |
| 汇点预测 | 0.78 | 0.88 | 0.82 | 1.00 | 1.00 | **1.00** | 0.68 | 1.00 | 0.81 | 0.18 | 0.75 | 0.29 |
| 新颖性检测 | 0.82 | 0.88 | **0.85** | 1.00 | 1.00 | **1.00** | 0.86 | 1.00 | **0.93** | 0.36 | 0.63 | **0.45** |
| 二元分类 | 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 |
| 频率 | 0.59 | 1.00 | 0.74 | 0.91 | 0.67 | 0.77 | 0.68 | 1.00 | 0.81 | 0.22 | 0.25 | 0.24 |

**表 6：Fluffy 在参数汇流的平衡集上的有效性。**

| | 平衡集 | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 代码注入 | | | 命令注入 | | | 反射型跨站脚本攻击 | | | 路径遍历 | | |
| 方法 | 精确率 | 召回率 | F1 分数 | 精确率 | 召回率 | F1 分数 | 精确率 | 召回率 | F1 分数 | 精确率 | 召回率 | F1 分数 |
| 汇点预测 | 0.36 | 0.98 | 0.53 | 0.94 | 0.99 | **0.97** | 0.63 | 1.00 | 0.77 | 0.53 | 0.64 | 0.58 |
| 新颖性检测 | 0.74 | 0.88 | 0.80 | 0.92 | 0.97 | 0.95 | 0.88 | 1.00 | **0.94** | 0.50 | 0.80 | 0.62 |
| 二元分类 | 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 |
| 频率 | 0.33 | 1.00 | 0.50 | 0.93 | 0.92 | 0.93 | 0.63 | 1.00 | 0.77 | 0.42 | 0.63 | 0.51 |

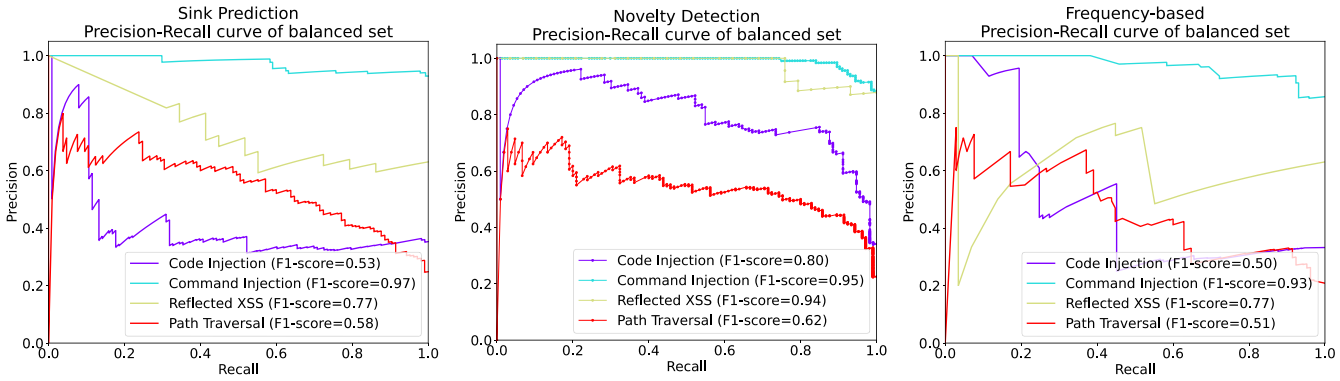
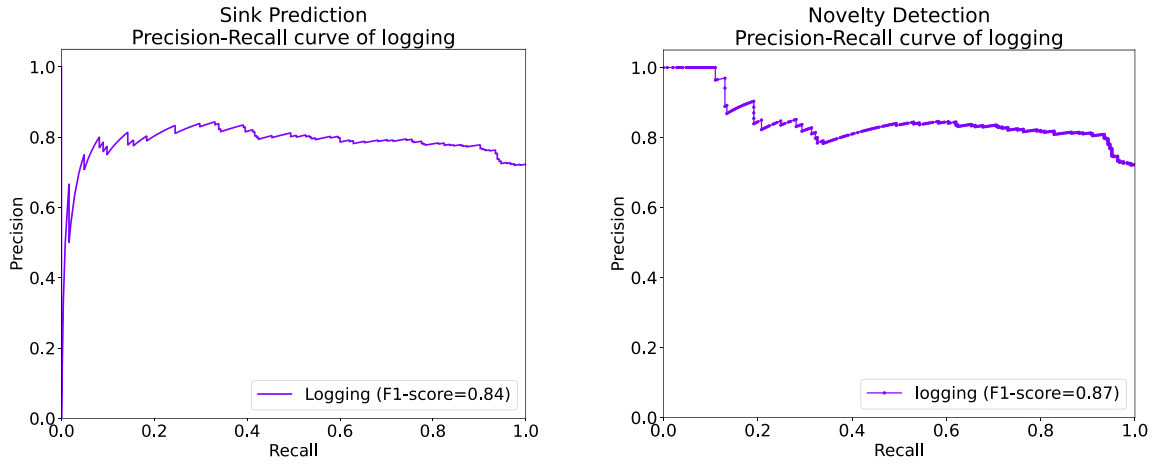



**图 2：平衡集上的汇点预测、新颖性检测和频率计数（从左至右 t).**

总而言之，Fluffy 被证明显著优于基线频率计数方法。表现最好的两种方法是新颖性检测和二元分类，而汇点预测效果不佳。我们的解读是，观察到的从源到汇点的数据流可能性，并不能帮助理解该数据流有多么意外。Codex 的表现也不如其他方法，这可能是因为提示中嵌入的随机抽取样本提供的信息不足。

对于所有方法而言，路径遍历流是最难分类的。这似乎是因为涉及的名称种类非常多样，包括文件操作（例如移动和删除）、文件类型，例如 PNG 和 MP4），以及其他可能代表某些文件类型的名称（例如日志和配置）。模型很难学会所有这些概念，从而降低了其有效性。其他汇点类型的名称则没有那么多样，例如，尽管命令注入流只有两个种子名称，新颖性检测模型仍取得了极佳的分数。

*4.2.3* 日志流。在日志流上评估 Fluffy 的结果如表7所示，图3给出了精确率-召回率曲线。总体而言，二元分类的表现最佳，其次是新颖性检测和正则表达式。后者的召回率最高，




**Fig图3：日志流上的汇点预测和新颖性检测精确率-召回率曲线（从左到右）** .

**表7：Fluffy 在日志流上的有效性。**

| | 日志流数据集 | | |
|---|---|---|---|
| 方法 | 精确率 | 召回率 | F1 分数 |
| 汇点预测 | 0.76 | 0.93 | 0.84 |
| 新颖性检测 | 0.81 | 0.93 | 0.87 |
| 二元分类 | **0.90** | 0.94 | **0.92** |
| Codex | 0.78 | 0.96 | 0.86 |
| 正则表达式 | 0.79 | **0.97** | 0.87 |

**表8：Fluffy 在 SecBench.js 数据集上的效果。**

| | SecBench.js 数据集 | | | | | |
|---|---|---|---|---|---|---|
| | 代码注入 | | | 命令注入 | | |
| 方法 | 精确率 | 召回率 | F1 | 精确率 | 召回率 | F1 |
| 汇点预测 | 0.97 | **0.88** | 0.92 | 1.00 | 0.90 | 0.95 |
| 新颖性检测 | 0.96 | 0.75 | 0.84 | 0.99 | **0.98** | 0.98 |
| 二元分类 | 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 |
| 频率 | 0.94 | 0.47 | 0.63 | 1.00 | 0.65 | 0.79 |

而前者的精确率更高。这表明，正则表达式相当有效，但编写和维护可能较为困难，而我们基于学习的方法几乎不需要人工干预。

举例来说，我们发现，Fluffy 能正确地将名为 passcode 的变量的明文日志记录标记为意外，而 CodeQL 却没有。这令人惊讶，因为该名称符合用于查找敏感数据的某个正则表达式。不过，事实证明，它也符合用于过滤加密源的正则表达式。我们将此问题报告给 CodeQL 维护者，他们承认这是一个错误。

## 4.3 研究问题 2：真实世界漏洞

RQ1 的结果为 Fluffy 能有效识别意外数据流提供了一些证据，但这尚不足以证明它可以用于发现现实世界漏洞。为回答这一问题，我们在 SecBench.js 数据集上评估 Fluffy，并将 param-sink 数据集中的一些被标记为意外的数据流报告给相关项目的维护者，以了解他们认为这些数据流是否确实存在问题。

*4.3.1 Fluffy* 对过去漏洞的表现。在 SecBench.js 上进行评估时，我们基本采用与第 4.2.2 节相同的评估方法，但基于阈值的方法除外：对于这些方法，我们使用在第 4.2.2 节的平衡集上取得最佳 F1 分数的阈值。这样可以评估我们设定的阈值能否推广到不同的数据集。我们省略路径遍历汇点类型，因为数据集中只有一条属于该类型的数据流。

我们以召回率作为主要指标（表8），因为我们主要关心 Fluffy 能否检测出这些已知漏洞，而精确率已在 RQ1 中评估。稍显意外的是，汇点预测在代码注入方面的召回率最高。原因是该方法在平衡集上的最佳阈值非常高，因此更适用于 SecBench.js 数据集；在该数据集中，除了一个代码注入数据流之外，其余都属于意料之外的情况。与之前一样，新颖性检测在命令注入方面表现出色，二元分类紧随其后。

*4.3.2 Fluffy* 针对此前未知的漏洞。在本节中，我们将考察 Fluffy 在发现新漏洞方面的效果。检查 Fluffy 在 param-sink 数据集上标记的数据流后，我们发现并报告了一条从热门 moment 库中名为 locale 的函数的 name 参数流向代码注入汇的数据流，正如引言中所提到的。开发人员已确认并修复了该漏洞，该漏洞现已被命名为 CVE-2022-24785。此外，我们还在 GitHub 上创建了 16 个拉取请求，以修复其他一些出乎意料、看起来不理想但危害较小的数据流。截至撰写本文时，其中 7 个已合并，8 个仍处于开放状态，尚未收到开发人员的反馈；另一个拉取请求已关闭，并已采用替代方案完成修复。

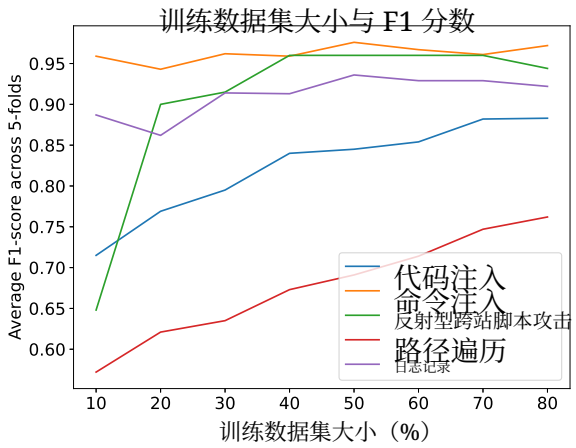



**图 4：训练集大小与模型有效性的关系。**

综合来看，这些结果表明，Fluffy 确实能够发现现实世界中的漏洞。

### 4.4 研究问题3：人工投入与模型有效性

从表 5 和表 6 可以看出，二元分类是有效性最佳的方法之一。然而，训练二元分类模型需要大量人工标注，这需要投入人力，而人力资源十分有限。因此，我们想了解需要多少标签才能使二元分类达到良好的有效性。

在本实验中，我们评估了平衡集上不同训练集规模下的二元分类效果。对于单一汇类型，将测试集固定为数据流的 20 %，并以 10% 为步长将训练集规模从 10% 增加到 80%。图 4 的结果显示，F1 分数通常会随着训练集规模增大而提高，但对于命令注入而言，即使 10% 也已足够。然而，使用 10% 训练数据的二元分类效果不及仅需更少训练数据的新颖性检测（即表 2 中的种子名称）。这表明，如果有足够的标注数据，前者是不错的选择；而后者则更为精简。

### 4.5 研究问题 4：可扩展性

我们的方法可能存在一个潜在问题：底层 CodeQL 查询包含数量庞大的源，其性能可能受到影响。然而，除 73 个项目（> 99.97 %）外，参数-汇点流数据集的查询均在 LGTM.com 默认的四小时超时限制内完成；日志流的查询也除 49 个项目（> 99.98%）外均在限制内完成。这表明，尽管 Fluffy 可能无法扩展到规模极其庞大的项目，但 CodeQL 组件通常不会成为瓶颈。

对于二元分类，我们的 5 折交叉验证训练耗时不到 5 分钟，每个模型的评估耗时约 1.5 秒。汇点预测模型需要一个半小时训练，评估耗时约 20 秒。新颖性检测模型的训练和评估耗时均不到 3 秒。对于 Codex，模型通过受到速率限制的 REST API 访问，平均 1.8 秒即可生成一条补全结果。这表明，除 Codex 外，Fluffy 的神经组件扩展性良好：训练最多只需几个小时，模型便能在几秒内对数百条数据流进行分类。我们在一台服务器上进行实验，该服务器配备 48 个主频为 2.2GHz 的 Intel Xeon 中央处理器核心、250GB 内存和一块 NVIDIA Tesla V100 图形处理器。二元分类和汇点预测模型使用图形处理器，而新颖性检测和 Codex 使用中央处理器。

### 4.6 有效性威胁

尽管上述结果令人鼓舞，但仍有若干有效性威胁需要注意。首先，我们只评估了五种漏洞类型，结果可能无法推广到其他类型。我们所考虑的代码库语料库受托管在 LGTM.com 上的项目选择限制，这可能会引入偏差。同样，我们的评估仅限于 JavaScript，因此无法确定该方法是否适用于其他语言。其次，我们的评估依赖人工标注，而人工标注可能出现人为错误。我们尝试通过引入独立标注者来减轻这一问题，但他们只标注了少部分数据，因此其余部分仍可能存在偏差。第三，新颖性检测所用种子名称的选择基于人工判断，这同样可能引入偏差。最后，平衡数据集包含的意外数据流多于我们预计现实环境中实际出现的数量，这可能导致过拟合。

## 5 相关工作

我们的工作属于神经软件分析这一大类 [45], , 其特点是可用信息模糊、缺乏定义明确的正确性标准，但训练数据量庞大。该领域以往的研究涵盖了各种问题，包括错误检测 [3, 47, 52],、自动程序修复 [16, 34, 35, 39, 54], 、代码补全 [4, 14, 31], 、概率类型推断 [2, 24, 40, 46], 、调用图剪枝 [33, 51], 、模拟动态污点分析 [50],, 以及从数据中学习静态分析器的推理规则 [8]。此外，也有大量研究利用机器学习技术检测漏洞 [20, 23, 36, 37], , 不过这类方法在实践中尚不够精确 [13]。

然而，与许多此类研究不同，我们的目标并非构建一个完全基于神经网络的端到端漏洞检测器。相反，我们借助机器学习技术增强现有的静态分析工具（CodeQL）。在这方面，我们延续了 Merlin [38], SuSi [48], Seldon [17], 和 USpec [19], 等系统的传统：这些系统利用机器学习，通过识别更多源和汇或推断别名行为来改进污点分析。不过，这些系统与我们的工作目标互为补充。上述方法可用于识别会直接产生受污染数据的 API，而 Fluffy 则能找出用户可能会不慎向其传入受污染数据的参数；也就是说，受污染数据并非源自该参数，而是来自其他地方。此外，由于我们使用自然语言信息，即使从未见过受污染数据实际传入此参数的示例，也能做到这一点

从而使我们能够发现潜在但尚不可利用的漏洞。[15]

当然，我们并非第一个意识到并利用源代码 [1, 12, 26]“双模”结构的人。例如，标识符名称此前就被用于错误检测 [43, 47], 、错误注入 [42], 、类型细化 [18], 和类型预测 [40]。不过，我们相信，我们是第一个系统地利用自然语言信息来改进污点分析的人。

也可以将我们的方法视为运用神经网络技术来识别假阳性，尽管在我们的案例中，底层污点分析有意保持不精确，并且并不期望它能独立发挥作用。近期在这一方向上的一项研究是 Kharkar 等人 [30],的工作，该研究探索了基于特征的方法和神经网络方法，以识别 Infer 静态分析器 [11]产生的假阳性。他们的思路与我们相关，因为其神经模型隐式地同时利用了代码和自然语言信息。我们的工作则专门聚焦于污点分析，并提出了其他形式的学习任务。

另一类相关工作关注预测静态分析中使用的启发式方法。例如，Jeong 等人学习用于选择方法的启发式方法，以找出可能受益于特定上下文敏感指向分析深度级别的方法 [28]。其他工作则预测在不遗漏任何待报告程序错误的情况下应展开哪些循环 [25] ，并学习状态选择启发式方法，以降低路径敏感性的成本 [32]。这类工作的最终目标是提高静态分析的效率，而 Fluffy 的目标则是确保分析报告的警告确实与开发人员相关。

我们的方法在很大程度上取决于底层的词嵌入。训练此类嵌入的常见方法包括 FastText [9],、word2vec [41] 和 GloVe [44], , 但这些方法是为自然语言设计的。我们处理的是变量名称，其特征截然不同 [53]。因此，我们使用 VarCLR [15], , 它专门针对标识符名称，并且已被证明在这一领域优于其他方法。

## 6 结论

本文提出了 Fluffy，这是一种用于双模污点分析的通用框架，它将用于识别候选数据流的静态污点分析与用于识别其中问题数据流的神经网络组件相结合。我们基于 CodeQL JavaScript 分析实现了这一框架，并采用了四种不同的神经组件（二元分类、汇点预测、新颖性检测和 Codex 语言模型）；随后，我们在大型真实 JavaScript 代码库语料库以及一组已知漏洞上评估了该系统。研究结果表明，Fluffy 在实践中表现良好，能够以较高的准确率识别可能存在问题的数据流，并以较高的召回率标记已知漏洞。

未来的工作中，我们希望研究静态组件与神经组件之间更紧密的集成，并进一步探索该技术在其他语言中的更广泛适用性。

[15]虽然 SuSi 使用的某些特征基于自然语言信息，但这些特征是人工设计的，并依赖非常简单的字符串匹配，例如判断方法名是否以“get”开头。

## 致谢

本研究获得了欧洲研究委员会（ERC，资助协议编号 851895）以及德国研究基金会通过 ConcSys 和 DeMoCo 项目提供的支持。

## 参考文献

[1] Miltiadis Allamanis、Earl T Barr、Premkumar Devanbu 和 Charles Sutton。2018。大规模代码的机器学习与自然性综述。*ACM*计算调查 *(CSUR)* 51, 4 (2018), 81. [2] Miltiadis Allamanis、Earl T. Barr、Soline Ducousso 和 Zheng Gao。2020。Typilus：神经类型提示。载于 *PLDI*。[3] Miltiadis Allamanis、Marc Brockschmidt 和 Mahmoud Khademi。2018。学习用图表示程序。载于 第六届国际学习表征会议（*ICLR 2018*），加拿大不列颠哥伦比亚省温哥华，*2018*年*4*月*30*日至*5*月*3*日，会议论文集。https://openreview.net/forum?id=BJOFETxR- [4] Uri Alon、Shaked Brody、Omer Levy 和 Eran Yahav。2019。code2seq：从代码的结构化表示生成序列。载于 第七届国际学习表征会议，*ICLR 2019, 6 9,* 美国路易斯安那州新奥尔良，*5*月 -。https://openreview.net/forum?id H gKYo tX [5] Pavel Avgustinov、Oege de Moor、Michael Peyton Jones 和 Max Schäfer。2016。QL：关系数据上的面向对象查询。载于 第*30*届欧洲面向对象编程会议，*ECOOP 2016*，*2016*年*7*月*18*日至*22*日，意大利罗马（*LIPIcs*，第*56*卷），Shriram Krishnamurthi 和 Benjamin S. Lerner（编）。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 和 Michael Pradel。2022。几乎免费的代码生成工具？对代码领域少样本预训练语言模型的研究。*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 和 Cristian-Alexandru Staicu。2022。SecBench.js：面向服务器端 JavaScript 的可执行安全基准测试套件。https://github.com/cristianstaicu/SecBench.js [8] Pavol Bielik、Veselin Raychev 和 Martin T. Vechev。2017。利用数据学习静态分析器。载于 计算机辅助验证——第*29*届国际会议，*CAV 2017*，德国海德堡，*2017*年*7*月*24*日至*28*日，论文集，第一部分（计算机科学讲义，第*10426*卷），Rupak Majumdar 和 Viktor Kuncak（编）。施普林格，233 253 10 1007 978 3 319 63387 9_12 – 。https://doi.org/ 。 / - - - [9] - Piotr Bojanowski 、 Edouard Grave 、 Armand Joulin 和 Tomas Mikolov。2017。利用子词信息丰富词向量。*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 和 Dario Amodei。2020。语言模型是少样本学习者。载于 神经信息处理系统进展*33*：神经信息处理系统年度会议*2020*（*NeurIPS 2020*），*2020*年*12*月*6*日至*12*日，线上，Hugo Larochelle、Marc’Aurelio Ranzato、Raia Hadsell、Maria-Florina Balcan 和 Hsuan-Tien Lin（编）。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 和 Dulma Rodriguez。2015。快速推进软件验证。载于 *NASA*形式化方法研讨会。施普林格，3–11。[12] Casey Casalnuovo、Earl T Barr、Santanu Kumar Dash、Prem Devanbu 和 Emily Morgan。2020。双通道约束理论。载于 *2020 IEEE/ACM* 第*42*届软件工程国际会议：新思想与新兴成果（*ICSE-NIER*）。IEEE，25–28。[13] Saikat Chakraborty、Rahul Krishna、Yangruibo Ding 和 Baishakhi Ray。2020。基于深度学习的漏洞检测：我们做到哪一步了？*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 和 Wojciech Zaremba。2021。评估基于代码训练的大语言模型。CoRR abs/2107.03374 (2021)。arXiv: 2107 03374 2107 03374 。https://arxiv.org/abs/ 。Qibin Chen、[15] Jeremy Lacomis、Edward J. Schwartz、Graham Neubig、Bogdan Vasilescu 和 Claire Le Goues。2022。VarCLR：通过对比学习进行变量语义表示预训练。载于 ICSE。[16] Zimin Chen、Steve Kommrusch、Michele Tufano、Louis-Noël Pouchet、Denys Poshyvanyk 和 Martin Monperrus。2019。SequenceR：用于端到端程序修复的序列到序列学习。IEEE TSE (2019)。[17] Victor Chibotaru、Benjamin Bichsel、Veselin Raychev 和 Martin T. Vechev。2019。利用大规模代码进行可扩展的污点规范推断。载于第40届 ACM SIGPLAN 编程语言设计与实现会议论文集（PLDI 2019），美国亚利桑那州菲尼克斯，2019年6月22日至26日，Kathryn S. McKinley 和 Kathleen Fisher（编）。ACM, 760 774 10 1145 3314221 3314648 – 。https://doi.org/ 。[18] / 。Santanu Kumar Dash 、 Miltiadis Allamanis 和 Earl T. Barr。 。RefiNym：使用名称细化类型。载于 ESEC/FSE。[19] Jan Eberhardt、Samuel Steffen、Veselin Raychev 和 Martin T. Vechev。2019。无监督学习 API 别名规范。载于第40届 ACM SIGPLAN 编程语言设计与实现会议论文集（PLDI 2019），美国亚利桑那州菲尼克斯，2019年6月22日至26日。745–759。https://doi.org/10.1145/ 3314221.3314640[20] Michael Fu 和 Chakkrit Tantithamthavorn。2022。LineVul：基于 Transformer 的行级漏洞预测。载于 MSR。[21] Tiferet Gazit。2022。利用机器学习查找安全漏洞。
h
t
t
p
s://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 和 Tomo Lazovich。2018。利用机器学习自动检测软件漏洞。CoRR abs/1803.04497 (2018)。arXiv:1803.04497 http://arxiv.org/abs/ 1803.04497[24] Vincent J. Hellendoorn、Christian Bird、Earl T. Barr 和 Miltiadis Allamanis。2018。深度学习类型推断。载于2018年 ACM 欧洲软件工程会议与软件工程基础研讨会联合会议论文集（ESEC/SIGSOFT FSE 2018），美国佛罗里达州布埃纳维斯塔湖，2018年11月4日至9日，Gary T. Leavens、Alessandro Garcia 和 Corina S. Pasareanu（编）。ACM, 152 162 10 1145 3236024 3236051 – 。https://doi.org/ 。 / 。[25] Kihong Heo、Hakjoo Oh 和 Kwangkeun Yi。 。机器学习引导的选择性不健全静态分析。载于第39届软件工程国际会议论文集（ICSE 2017），2017年28，2017，20，阿根廷布宜诺斯艾利斯，5月 - Sebastián Uchitel、Alessandro Orso 和 Martin P. Robillard（编）。IEEE / ACM, 519 529 10 1109 2017 54 – 。https://doi.org/ 。 / ICSE。 。Abram Hindle、Earl T. Barr、Zhendong Su、Mark Gabel 和 Premkumar T. Devanbu。2012。论软件的自然性。载于第34届软件工程国际会议（ICSE 2012），2012年6月2日至9日，瑞士苏黎世。837–847。[27] Naman Jain、Skanda Vaidyanath、Arun Iyer、Nagarajan Natarajan、Suresh Parthasarathy、Sriram Rajamani 和 Rahul Sharma。2022。Jigsaw：大语言模型与程序合成相遇。载于 ICSE。[28] Sehun Jeong、Minseok Jeon、Sung Deok Cha 和 Hakjoo Oh。2017。用于指向分析的数据驱动上下文敏感性。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 和 Gust Verbruggen。2022。修复几乎等同于生成：使用大语言模型进行多语言程序修复。https://doi.org/10.48550/ARXIV.2208.11640[30] Anant Kharkar、Roshanak Zilouchian Moghaddam、Matthew Jin、Xiaoyu Liu、Xin Shi、Colin Clement 和 Neel Sundaresan。2022。在分析型错误检测器中减少假阳性的学习。载于 ICSE。[31] Seohyun Kim、Jinman Zhao、Yuchi Tian 和 Satish Chandra。2021。将树输入 Transformer 进行代码预测。载于 ICSE。[32] Yoonseok Ko 和 Hakjoo Oh。2023。学习提升析取式静态错误查找器。载于 ICSE。[33] Thanh Le-Cong、Hong Jin Kang、Truong Giang Nguyen、Stefanus Agus Haryono、David Lo、Xuan-Bach D. Le 和 Quyet Thang Huynh。2022。AutoPruner：基于 Transformer 的调用图剪枝。载于 ESEC/FSE。[34] Claire Le Goues、Michael Pradel 和 Abhik Roychoudhury。2019。自动程序修复。Commun. ACM 62, 12 (2019), 56–65。https://doi.org/10.1145/ 3318162[35] Yi Li、Shaohua Wang 和 Tien N. Nguyen。2020。DLFix：用于自动程序修复的基于上下文的代码转换学习。载于 ICSE。[36] Yi Li、Shaohua Wang 和 Tien N. Nguyen。2021。利用细粒度解释进行漏洞检测。载于 ESEC/FSE '21：第29届 ACM 欧洲软件工程会议与软件工程基础研讨会联合会议，希腊雅典，2021年8月23日至28日，Diomidis Spinellis、Georgios Gousios、Marsha Chechik 和 Massimiliano Di Penta（编）。ACM，292–303。https://doi.org/10. 1145/3468264.3468597

[37] Zhen Li、Shouhuai Xu Deqing Zou 和、Xinyu Ou、Hai Jin、Sujuan Wang、Zhijun Deng 和 Yuyi Zhong。2018。VulDeePecker：一种基于深度学习的漏洞检测系统。载于*NDSS*。[38] V. Benjamin Livshits、Aditya V. Nori、Sriram K. Rajamani 和 Anindya Banerjee。2009。Merlin：针对显式信息流问题的规范推断。载于*2009*年 *ACM SIGPLAN* 编程语言设计与实现会议论文集，*PLDI 2009*，爱尔兰都柏林，*2009*年*6*月*15*日至*21*日。75–86。[39] Thibaud Lutellier、Hung Viet Pham、Lawrence Pang、Yitong Li、Moshi Wei 和 Lin Tan。2020。CoCoNuT：使用集成组合上下文感知神经翻译模型进行程序修复。载于*ISSTA '20*：第*29*届 *ACM SIGSOFT* 国际软件测试与分析研讨会，线上活动，美国，*2020*年*7*月*18 22*日，101 114-Sarfraz Khurshid 和 Corina S. Pasareanu（编）。ACM，–。https: //doi.org/ 10 1145 3395363 3397369 。/。Rabee [40] Sohail Malik、Jibesh Patra 和 Michael Pradel。2019。NL2Type：从自然语言信息推断 JavaScript 函数类型。载于第*41*届软件工程国际会议论文集，*ICSE 2019*，蒙特利尔，魁北克省，加拿大，*2019*年*5*月 *25 31, 2019* 304 315 10 1109 2019 00045 -。–。https://doi.org/。/ICSE。。[41] Tomas Mikolov、Kai Chen、Greg Corrado 和 Jeffrey Dean。。高效估计向量空间中的词表示。*CoRR* abs/1301.3781 (2013)。http://arxiv.org/abs/ 1301 3781 。Jibesh Patra 和 Michael [42] Pradel。2021。语义错误播种：一种创建真实错误的基于学习的方法。载于*ESEC/FSE '21*：第*29*届 *ACM* 联合软件工程欧洲会议暨软件工程基础研讨会，希腊雅典，*2021*年*8*月*23*日至*28*日，Diomidis Spinellis、Georgios Gousios、Marsha Chechik 和 Massimiliano Di Penta（编）。ACM，906–918。https://doi.org/ 10 1145 3468264 3468623 。/。Jibesh Patra 和 Michael Pradel。。Nalin：从运行时行为中学习，以发现 Jupyter Notebook 中的名称-值不一致。载于*ICSE*。[44] Jeffrey Pennington、Richard Socher 和 Christopher D. Manning。2014。GloVe：用于词表示的全局向量。载于*2014*年自然语言处理实证方法会议论文集，*EMNLP 2014*，*2014*年*10*月*25*日至*29*日，多哈，卡塔尔，*ACL* 特别兴趣小组 *SIGDAT* 会议，Alessandro Moschitti、Bo Pang 和 Walter Daelemans（编）。ACL，1532–1543。https://doi.org/ 10 3115 1 14 1162 。/v /d - Michael Pradel 和 Satish Chandra。2022。神经软件分析。[45] *Commun. ACM* 65, 1 2022 86 96 10 1145 3460348 ( )，–。https://doi.org/。/ Michael [46] Pradel、Georgios Gousios、Jason Liu 和 Satish Chandra。2020。Type-Writer：结合基于搜索的验证进行神经类型预测。载于*ESEC/FSE '20*：第*28*届 *ACM* 联合软件工程欧洲会议暨软件工程基础研讨会，线上活动，美国，*2020*年*11*月*8*日至*13*日。 209 220 10 1145 3368089 3409715 –。https://doi.org/。/。Michael [47] Pradel 和 Koushik Sen。。DeepBugs：一种基于学习的基于名称的错误检测方法。*PACMPL* 2，OOPSLA (2018)，147:1–147:25。https: //doi.org/ 10 1145 3276517 。/ Siegfried Rasthofer、Steven Arzt 和 Eric Bodden。2014。一种用于对 Android 源与汇进行分类和归类的机器学习方法。载于第*21*届年度网络与分布式系统安全研讨会，*NDSS 2014*，美国加利福尼亚州圣迭戈，*2014*年*2*月*23*日至*26*日。互联网协会。[48]
h
t
t
p
s://
w
w
w.
n
d
s
s-
s
y
m
p
o
si
um.org/ndss2014/machine-learning-approach-classifying-and-categorizing-android-sources-and-sinks [49] Bernhard Schölkopf、John C Platt、John Shawe-Taylor、Alex J Smola 和 Robert C Williamson。2001。估计高维分布的支撑集。《神经计算》13，7 (2001)，1443–1471。[50] Dongdong She、Yizheng Chen、Baishakhi Ray 和 Suman Jana。2019。Neutaint：使用神经网络进行高效动态污点分析。*CoRR* abs/1907.03756 ( 2019 1907 03756 1907 03756 )。arXiv: 。http://arxiv.org/abs/。[51] Akshay Utture、Shuyang Liu、Christian Gram Kalhauge 和 Jens Palsberg。2022。取得平衡：从静态调用图中剔除假阳性。载于*ICSE*。[52] Marko Vasic、Aditya Kanade、Petros Maniatis、David Bieber 和 Rishabh Singh。2019。通过联合学习定位与修复，实现神经程序修复。载于*ICLR*。[53] Yaza Wainakh、Moiz Rauf 和 Michael Pradel。2021。IdBench：评估源代码中标识符名称的语义表示。载于第*43*届 *IEEE/ACM* 国际软件工程会议，*ICSE 2021*，西班牙马德里，*5*月*22*日至*30*日 *2021* 562 573 10 1109 43902 2021 00059 。IEEE，–。https://doi.org/。/ICSE。。[54] He Ye、Matias Martinez 和 Martin Monperrus。。通过基于执行的反向传播进行神经程序修复。载于*ICSE*。