《软件测试与验证》课程论文解读报告

L4-01: 王力 10225101434，武泽恺 10225101429，李鹏达 10225101460

论　　文：Do Automatic Test Generation Tools Generate Flaky Tests?[1]
实验仓库：https://doi.org/10.6084/m9.figshare.22344706.v3

1　背景介绍

Flaky Test[2]是指在测试环境完全没有变化的情况下，有时通过有时失败的测试用例。*Flakiness* 是指 Flaky Test 的如上所述的产生不稳定结果的特性。Flaky Test 的广泛存在，对软件开发者和研究者都构成了挑战。虽然针对开发者编写的 Flaky Test 已有较多研究，但对于自动测试生成工具生成的 Flaky Test 的研究较少。

2　实验目标

(1) 探究自动生成的测试中 Flaky Test 的普遍性 (Prevalence)。
(2) 验证 EvoSuite 的抑制 Flaky Test 功能的有效性。
(3) 探究自动生成的 Flaky Test 与开发者编写的 Flaky Test 在根本原因上的差异。

3　实验过程

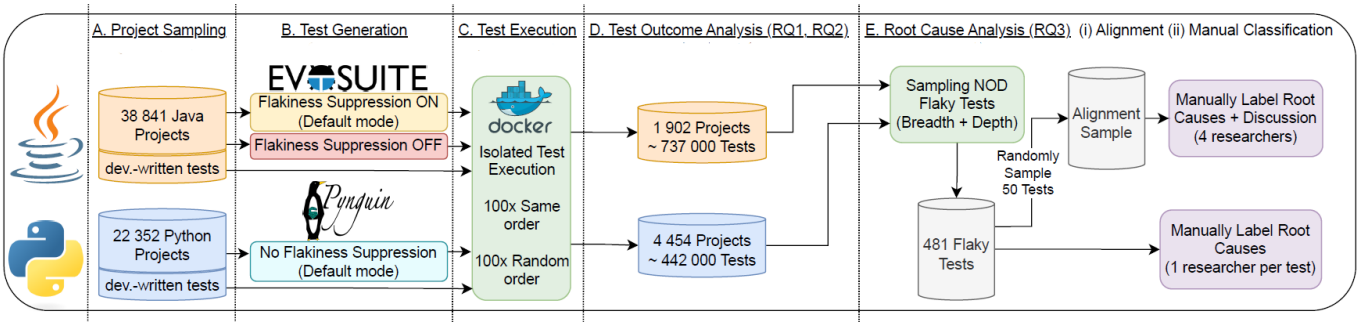


图 1: 实验过程

图 1 展示了实验的主要流程。

[1]Martin Gruber, Muhammad Firhard Roslan, Owain Parry, Fabian Scharnböck, Phil McMinn, and Gordon Fraser. 2024. *Do Automatic Test Generation Tools Generate Flaky Tests?* In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24). Association for Computing Machinery, New York, NY, USA, Article 47, 1–12. https://doi.org/10.1145/3597503.3608138

[2]Flaky Test 暂时没有统一的中文翻译，因此在本报告中称英文原文

3.1 工具选择

- 针对 Java 和 Python 两种语言，分别选择了 EvoSuite 和 Pynguin 两种流行的基于搜索的测试生成工具。
- 使用 Flapy 以在隔离的环境中执行测试并管理依赖。

3.2 项目选择

- *Java 项目*：使用 Maven 中央仓库索引作为数据来源。限定选择的项目必须由 Maven 构建且源代码托管于 GitHub 上，最终筛选出 38,841 个项目。
- *Python 项目*：使用 Python 官方第三方软件仓库 Python Package Index (PyPI) 作为数据来源，随机抽取了 22,352 个项目。每个项目至少包含一个可以通过 pytest 执行的测试用例且项目源代码必须托管在 GitHub 上。

3.3 测试生成

- *Java 项目*：EvoSuite 有抑制 Flaky Test 产生的机制，为了衡量这种机制的影响，实验采用 EvoSuite$_{FSOn}$(启用) 和 EvoSuite$_{FSOff}$(禁用) 两种配置分别进行测试用例生成。测试生成范围则选择为每个项目的每个可测试类生成测试，可测试类的含义为至少包含一个公共方法的类。生成预算为每个可测试类设置 2 分钟的搜索时间。
- *Python 项目*：Pynguin 没有可选的抑制 Flakiness 的参数，所以只在一种配置 (默认配置) 之下进行测试生成。生成范围为对每个项目中的每个模块生成测试。生成预算为每个模块 10 分钟的搜索时间。

EvoSuite 和 Pynguin 生成测试具有不稳定性，每次运行可能生成不同的测试集。以往研究通常通过为每个项目生成多个测试集来应对这种随机性，而在此本研究中，对每个类（EvoSuite）或模块（Pynguin）仅生成一次测试集，以减少计算负担。为保证结果的普遍性，研究通过对大量项目进行抽样来弥补随机性的影响。

3.4 测试执行

为检测 Flaky Tests，研究对开发者编写的测试和生成的测试分别执行 200 次，其中 100 次为固定顺序，100 次为随机顺序。这是为了区分顺序依赖 Flaky Test（OD）和非顺序依赖 Flaky Test（NOD）。测试执行要么直接用 FlaPy 进行，要么仿照其原理进行, 以确保每次测试执行都在一个隔离的环境中并管理依赖。开发者编写的测试和工具生成的测试分开运行，避免交叉影响。

对于项目的依赖安装，Java 项目使用 `mvn dependency:copy-dependencies` 复制项目依赖。Python 项目 FlaPy 使用内置的依赖项安装功能。

对于随机顺序执行，Python 项目通过 pytest 的 “random-order” 插件实现实现随机执行测试。Java 由于 Maven 的 Surefire 插件不支持随机排序，研究团队开发了一个自定义测试运行器。

3.5 测试输出结果分析

3.5.1 数据集分析

作者计算了能够成功执行开发者编写的测试且能够在工具下生成测试的项目数量，这些项目是可以用于后续分析的数据集。除此之外，为了确认是否获得了足够大且多样的项目集，实验采用了两项指标：

(1) 项目的源代码行数 (LOC)
(2) 项目拥有的开发者编写的测试数量

同时，为了确认实验是否正确使用了 EvoSuite 和 Pynguin，作者还比较了自动生成测试的覆盖率和之前应用这些工具的研究所报告的覆盖率。

3.5.2 普遍性

为了研究 Flaky Test 的普遍性，实验分析了

(1) 不使用 Flakiness 抑制机制的情况下生成的 Flaky Test 数量与开发者编写的测试中发现的 Flaky Test 数量。
(2) 顺序依赖 Flaky Test（OD）与非顺序依赖 Flaky Test（NOD）的比例。
(3) 至少包含一个开发者编写或工具生成的 Flaky Test 的项目数量。

3.5.3 Flakiness 抑制机制

为了评估 EvoSuite 的 Flakiness 抑制机制的效果，实验将 EvoSuite$_{FSOn}$ 生成的 Flaky Test 数量与 EvoSuite$_{FSOff}$ 生成的数量 Flaky Test 数量以及开发者编写的 Flaky Test 数量进行了比较。

3.6 根本原因分析

作者团队手动对 Flaky Test 的根本原因进行标注。由于 OD 的 Flaky Test 在随机顺序测试的步骤中能够被自动检测出，手动标注仅针对 NOD 的 Flaky Test。按以下步骤进行检测：

(1) 采样 *(Sampling)*：实验采用了广度采样和深度采样两种采样策略以保证样本的代表性。总计采样 481 个 NOD Flaky Test。其中 329 个来自广度采样，122 个来自深度采样，30 个同时被两种策略选中。

- 广度采样：从每个受影响的项目中随机选择一个 NOD Flaky Test，无论其类型是什么。
- 深度采样：随机选择 21 个 Java 项目和 9 个 Python 项目，采集这些项目中的所有 NOD Flaky Test。这一步选择的项目包含的 Flaky Test 均衡分布于不同类型。

(2) 标注 *(Labeling)*：四名作者根据项目代码、测试代码以及测试失败信息（如堆栈跟踪和错误消息）进行标注。为了确保作者们标注的一致性，作者们事先在 50 个 Flaky Test 上分别独立进行了分类。然后作者们比对他们各自的结果，并对分歧进行了讨论整理，然后对根本原因类别做出了以下调整：

- 将无序集合的类别拓宽，使其一般上也包括未指定的行为。
- 将类别“资源泄漏”的范围扩大，使其也涵盖资源不可用。
- 添加性能 (category performance) 类别，它描述了由于（连续）过程的持续时间不同而间歇性失败的测试。
- 添加非幂等性结果（non-idempotent-outcome NIO）类别，涵盖了自污染和自状态设定测试。

(3) 基于标注的根本原因比较：

- 比较本研究中开发者编写测试的 Flakiness 根本原因与先前研究结果。
- 比较未启用 Flakiness 抑制机制的工具生成测试（EvoSuite$_{FSOff}$、Pynguin）与开发者编写的测试的 Flakiness 根本原因。
- 比较启用（EvoSuite$_{FSOn}$）与未启用（EvoSuite$_{FSOff}$）Flakiness 抑制机制的工具生成 Flaky Test 的根本原因。

3.7 有效性威胁分析

作者从以下方面分析了对实验有效性的潜在威胁：

- 外部有效性 *(External Validity)*：外部有效性的威胁主要体现在项目的选取上。Java 项目仅包括使用 Maven 的项目，并排除使用 JUnit 3 的项目。Python 项目基于现有数据集，该数据集已被多项研究使用，但可能继承其潜在缺陷。
- 构建有效性 *(Construct Validity)*：首先，尽管实验顺序和乱序各执行了每个测试 100 次，但有的 Flaky Test 发生概率较低，这将导致对 Flaky Test 数量的低估。其次，为每个测试分配的生成预算是有限的，如果分配更多的时间可能会有不同的结果。

- 内部有效性 *(Internal Validity)*: 首先，因为实验产生的 NOD Flaky Test 的数量过多而不得不采用抽样分析的方法，这个抽样过程对研究结果的有效性构成潜在威胁。其次，对 Flaky Test 的根本原因标注由四位作者合作完成，因作者们对 Flaky Test 的根本原因有不同的经验，这也导致了潜在的威胁。

4 实验结果与分析

4.1 数据集分析

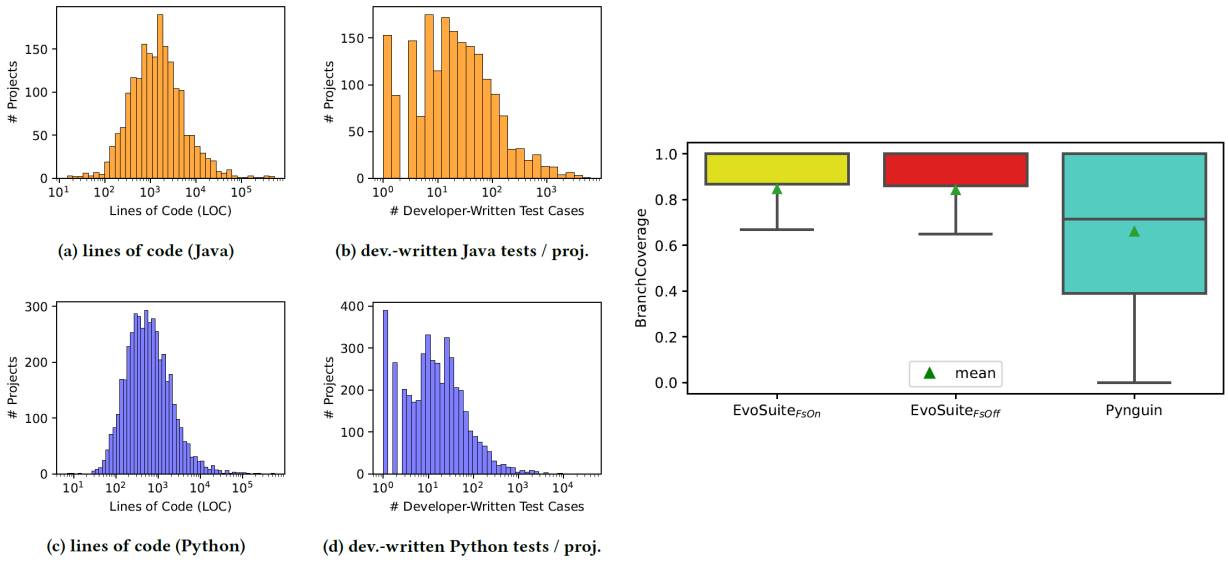


图 2: 数据集分析

图 2 展示了有效数据集 (能够成功执行开发者编写的测试且能够在工具下生成测试的项目) 的统计信息。

- *Java* 项目：代码平均行数为 4,948，涵盖了从小于 100 行到大于 50 万行的范围。其中，共包含 163,305 个开发者编写用例，平均每个项目 85.9 个。EvoSuite 生成测试覆盖率平均达 81%, 与以往研究 (84%) 相符。
- *Python* 项目：代码平均行数为 1,755，涵盖了从小于 100 行到大于 50 万行。共包含 303,711 个开发者编写用例。平均每个项目 68.2 个。Pynguin 生成测试覆盖率平均达 66.0%, 与以往研究 (71.6%) 非常接近。

这些结果表明，实验获得了足够大且多样的项目集，且实验正确使用了 EvoSuite 和 Pynguin 生成测试。

4.2 自动化测试生成工具生成 Flaky Test 的普遍性

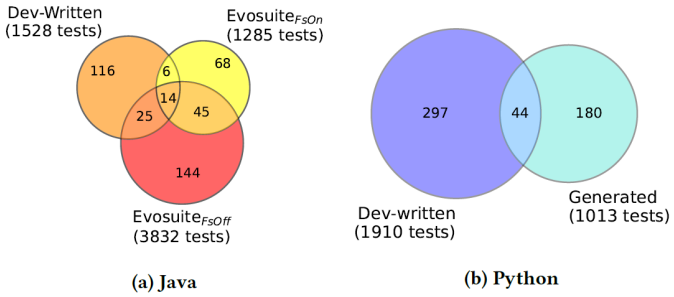
表 1 展示了开发者编写的测试和自动生成的测试中 Flaky Test 的数量。在 120 万次测试中，共发现了 9,568 个 Flaky Test，其中约 2/3 的测试由自动化测试生成工具生成。

与开发者编写的测试相比，自动生成的测试更容易出现 Flaky Test，Java 测试的比例增加了 54%（从 0.94% 增加到 1.45%），Python 测试的比例增加了 16%（从 0.63% 增加到 0.73%），证实了 **Flaky Test 在开发者编写和自动生成的测试中均普遍存在**。顺序依赖性方面，自动生成的 Python 测试与开发者编写的测试类似，顺序依赖性较强；而 Java 测试中开发者编写的测试表现均衡，自动生成的测试却更偏向顺序依赖性。

图 3 展示了 Flaky Test 的分布，由自动化测试生成工具生成的 Flaky Test 与开发者编写的测试交集较小，Java 中仅有 17.1% 的项目同时包含两类 Flaky Test，Python 中为 19.6%。作者建议研究者可以进一步探索。

Language	Test Type	NOD		OD		Flaky (NOD + OD)		All	
		# Tests	# Projects	# Tests	# Projects	# Tests	# Projects	# Tests	# Projects
Java	Developer-Written	698 (0.43 %)	105 (5.52 %)	830 (0.51 %)	104 (5.46 %)	1 528 (0.94 %)	161 (8.46 %)	163 305	1 902
	EvoSuite$_{FSOn}$	175 (0.06 %)	43 (2.26 %)	1 110 (0.35 %)	109 (5.73 %)	1 285 (0.41 %)	133 (6.99 %)	310 193	
	EvoSuite$_{FSOff}$	597 (0.22 %)	111 (5.84 %)	3 235 (1.23 %)	163 (8.57 %)	3 832 (1.45 %)	228 (11.9 %)	264 000	
Python	Developer-Written	182 (0.06 %)	88 (1.98 %)	1 728 (0.57 %)	270 (6.06 %)	1 910 (0.63 %)	341 (7.65 %)	303 711	4 454
	Pynguin	88 (0.06 %)	49 (1.10 %)	925 (0.67 %)	183 (4.11 %)	1 013 (0.73 %)	224 (5.03 %)	138 627	

表 1: 开发者编写的测试和自动生成的测试中 Flaky Test 的数量



(a) Java (b) Python

图 3: Flaky Test 的分布

4.3 自动化测试生成工具中 Flakiness 抑制机制的有效性

如表 1 所示，通过引入 Flakiness 抑制机制，Flaky Test 数量比例降低幅度达 71.7%（从 1.45% 降至 0.41%），也远低于开发者手写测试的 Flaky Test 数量比例（降低幅度达 56.4%），证实了 **EvoSuite 提供的 Flakiness 抑制机制能够有效降低测试的 Flakiness**。同时，即使 Flakiness 抑制机制开启的情况下，仍然有 86.4% 的 flaky test 表现出顺序依赖性。因此，作者不仅建议测试生成工具的维护者可以去借鉴 EvoSuite 的成功经验，也提议研究者对工具生成测试中的顺序依赖性进行进一步探讨。

4.4 产生 Flaky Test 的根本原因

Language	Java			Python	
Test Type	Dev.-Written	EvoSuite$_{FSOn}$	EvoSuite$_{FSOff}$	Dev.-Written	Pynguin
Total	170 (106)	57 (42)	113 (102)	93 (88)	48 (47)
Async Wait	36 (23)	3 (2)	5 (4)	11 (7)	1 (1)
Concurrency	15 (14)	0 (0)	5 (3)	3 (3)	0 (0)
Floating Point	0 (0)	0 (0)	0 (0)	2 (2)	0 (0)
I/O	9 (9)	0 (0)	0 (0)	6 (6)	3 (3)
Network	36 (9)	1 (1)	13 (13)	28 (27)	6 (6)
NIO	0 (0)	0 (0)	0 (0)	3 (3)	5 (5)
OTHER	0 (0)	34 (22)	0 (0)	0 (0)	3 (3)
Performance	33 (11)	0 (0)	7 (7)	4 (4)	4 (4)
Randomness	13 (12)	2 (2)	23 (22)	16 (16)	11 (11)
Resource Leak / Resource Unavailability	7 (7)	2 (2)	7 (6)	0 (0)	0 (0)
Test Case Timeout	2 (2)	4 (4)	23 (18)	0 (0)	0 (0)
Time	6 (6)	2 (1)	12 (12)	4 (4)	0 (0)
Too Restrictive Range	2 (2)	0 (0)	0 (0)	7 (7)	0 (0)
UNKNOWN	8 (8)	2 (2)	4 (4)	7 (7)	6 (5)
Unordered Collection / Unspecified Behavior	3 (3)	7 (6)	14 (13)	2 (2)	9 (9)

表 2: 对于 NOD Flaky Test 根本原因的分析

表 2 展示了对于 NOD Flaky Test 根本原因的分析。作者将 Flaky Test 的根本原因归纳划分为 15 种。对于 Java 项目，开发人员编写的 Flaky Test 的根本原因主要是异步等待（21.2%）和性能假设（19.4%），而 Python 项目主要由于网络（30.1%）和随机性（17.2%）问题。相比之下，未启用抑制机制所生成 Flaky Test 主要由随机性（约 20%）和未指定的行为（12.4%~18.7%）导致；同时，EvoSuite 生成的测试还会造成测试用例超时（20.4%），而 Pynguin 生成的测试则会导致非幂等结果（10.4%）。启用抑制机制后，传统根本原因（如随机性和超时）的 Flaky Test 比例显著减少，但也引入了新类别问题，主要由运行时优化（如栈溢出错误等）和自动生成测试工具内部资源限制两类根本原因引起（59.6%）。

结合分析，作者建议 EvoSuite 用户开启 Flakiness 抑制机制，并调整标志参数和移除特定脚手架文件，而 Pynguin 用户应设置随机数种子，从而尽可能地减少随机性带来的 Flakiness。这些建议为目前部分已知根本原因的问题提供了有效解决方案。

4.5 总结反思

Flaky Tests 是软件测试中常见且棘手的现象。在此研究中，研究团队证明了不稳定性不仅会影响开发者编写的测试，且 Flaky Test 在生成的测试中比在开发者编写的测试中更为常见。EvoSuite 的 Flakiness 抑制机制能够有效降低测试的 Flakiness，但也可能会引入新类别的 Flaky Test。生成的测试与开发者编写的测试具有相同的根本原因，但这些原因的分布有所不同：开发者编写的 Flaky Test 通常由并发和网络操作引起，而生成的 Flaky Test 则往往是由随机性和未指定行为导致的。

此研究填补了现有研究中对自动生成测试中 Flaky Test 的研究空白，并详细地分析了自动测试生成产生 Flaky Test 的根本原因。研究结果为测试生成工具的维护者和研究者提供了有价值的见解。然而，我们认为以下问题仍然值得考虑：

(1) 研究只使用了 EvoSuite 和 Pynguin 两个工具，覆盖的语言仅限于 Java 和 Python，限制了研究的普适性。实验结果显示，对于不同的语言和工具，Flaky Test 的根本原因可能会有所不同。Flaky Test 的产生的原因是否与语言本身特性有关？同一语言下，不同测试生成工具生成 Flaky Test 的根本原因是否相同？
(2) 研究虽然验证了 EvoSuite 的 Flakiness 抑制机制的有效性，但并未探究其具体原理。EvoSuite 的 Flakiness 抑制机制是如何工作的？是否可以将这种机制应用到其他测试生成工具中？
(3) 研究中对于 Flaky Test 的根本原因的分析是基于手动标注的，这可能会引入主观因素。是否可以通过机器学习等方法自动识别 Flaky Test 的根本原因，以提高分析的准确性？
(4) 作者提供了实验的数据集、测试输出结果和分析测试输出结果的代码，但未提供生成和执行测试的代码，这使得读者难以重现实验的全部过程。