What Do Developer-Repaired Flaky Tests Tell Us About the Effectiveness of Automated Flaky Test Detection?

Owain Parry[1], Gregory M. Kapfhammer[2], Michael Hilton[3], Phil McMinn[1]

[1]University of Sheffield, UK
[2]Allegheny College, USA
[3]Carnegie Mellon University, USA

What is a *flaky test*?

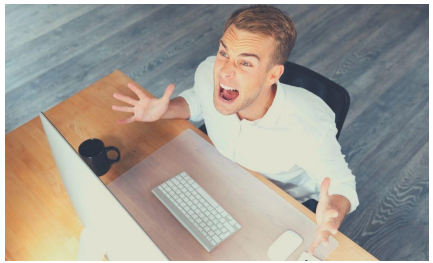
- A test case that can pass and fail without any code changes.
- They disrupt continuous integration, cause of a loss of productivity, and limit the efficiency of testing *[Parry et. al. 2022, ICST]*.
- A recent survey found that nearly 60% of software developer respondents encountered flaky tests on at least a monthly basis *[Parry et. al. 2022, ICSE:SEIP]*.

+

JUnit 5

pytest

=



What has been done about flaky tests?

- The research community has presented a multitude of automated detection techniques.
- Many methodologies for evaluating such techniques do not accurately assess their usefulness for developers.
- Some calculate recall against a baseline of flaky tests detected by *automated rerunning*.
- Others simply present the number of detected flaky tests.

What did we do?

- We performed a study to demonstrate the value of a *developer-based* methodology for evaluating automated detection techniques.
- It features a baseline of developer-repaired flaky tests that is more suitable for assessing a technique's usefulness for developers.
- The fact that developers allocated time to repair the flaky tests in this baseline implies they were of interest.

Our research questions

RQ1: What is the recall of automated rerunning against our baseline?

RQ2: What causes the flaky tests in our baseline and how did developers repair them?

Methodology: Baseline

- We searched for commits among the top-1,000 Python repositories on GitHub (by number of stars) using the query: "`flaky OR flakey OR flakiness OR flakyness OR intermittent`".
- Upon finding matches, we checked the commit messages and code diffs to identify each individual developer-repaired flaky test.
- We ended up with a baseline of **75 flakiness-repairing commits** from **31 open-source Python projects**.

Methodology: RQ1

- We developed our own automated rerunning framework called *ShowFlakes*.
- It can introduce four types of noise into the execution environment during reruns.
- For each of the 75 commits, we used ShowFlakes to rerun the developer-repaired flaky tests at the state of the parent 1,000 times with no noise and 1,000 times with noise.
- We considered a commit to be “detected” if ShowFlakes could detect at least one of its developer-repaired commits.

Methodology: RQ2

- We manually classified the causes of the flakiness and the developer's repairs in the 75 commits.
- For the causes, we used the same ten cause categories introduced by Luo et. al. in their empirical study on flaky tests *[Luo et. al. 2014, FSE]*.
- For the repairs, we followed a more exploratory approach to allow for a set of repair categories to emerge.

Results: RQ1

- Table shows, for how many of the 75 commits, could rerunning detect at least one flaky test.
- Rerunning with noise performed better than without noise, but still only achieved a recall of 40%.

		Detected Commits	
GitHub Repository	**Commits**	**No Noise**	**Noise**
home-assistant/core	6	3	3
HypothesisWorks/hypothesis	6	1	2
pandas-dev/pandas	6	1	2
quantumlib/Cirq	5	1	2
apache/airflow	4	2	3
pytest-dev/pytest	4	-	-
scipy/scipy	4	-	2
python-trio/trio	4	1	2
urllib3/urllib3	4	1	2
+22 others	32	6	12
Total	**75**	**16 (21%)**	**30 (40%)**

Results: RQ2

Cause	Add Mock	Add/Adjust Wait	Guarantee Order	Isolate State	Manage Resource	Reduce Random.	Reduce Scope	Widen Assertion	Misc.	Total
Async. Wait	1	6	-	-	-	-	-	2	-	**9**
Concurrency	-	2	-	-	2	-	-	2	2	**8**
Floating Point	-	-	-	-	-	-	-	3	-	**3**
I/O	-	-	-	-	-	-	-	-	-	**-**
Network	3	3	-	-	1	-	-	-	1	**8**
Order Dependency	-	-	-	2	-	-	1	-	-	**3**
Randomness	-	-	-	-	-	6	-	4	1	**11**
Resource Leak	-	-	-	-	2	-	1	1	-	**4**
Time	5	-	-	-	-	-	1	1	2	**9**
Unordered Collection	-	-	3	-	-	-	-	-	-	**3**
Miscellaneous	2	-	1	-	1	-	-	6	7	**17**
Total	**11**	**11**	**4**	**2**	**6**	**6**	**3**	**19**	**13**	**75**

Implications

- We found that the recall of automated rerunning was low against our baseline.
- This suggests that, for **developers**, the usefulness of this technique is limited.
- For **researchers**, this implies that a baseline provided by automated rerunning would be unsuitable for assessing developer usefulness.
- We found that automated rerunning with noise performed significantly better than without.
- Therefore, if **developers** are going to use rerunning, we recommend doing so with noise.