What Made This Test Flake? Pinpointing Classes Responsible for Test Flakiness

Sarra Habchi
Ubisoft
sarra.habchi@ubisoft.com

Guillaume Haben
University of Luxembourg
guillaume.haben@uni.lu

Jeongju Sohn
University of Luxembourg
Jeongju.sohn@uni.lu

Adriano Franci
University of Luxembourg
adriano.franci@uni.lu

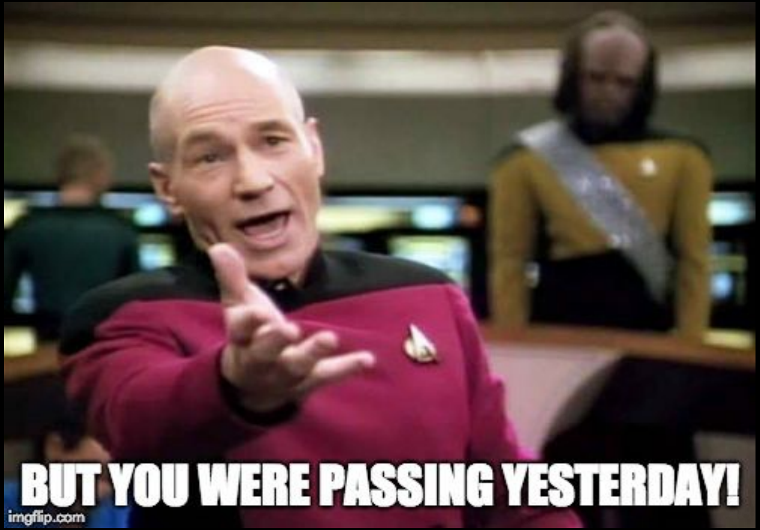
Mike Papadakis
University of Luxembourg
michail.papadakis@uni.lu

Maxime Cordy
University of Luxembourg
maxime.cordy@uni.lu

Yves Le Traon
University of Luxembourg
yves.letraon@uni.lu

2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)

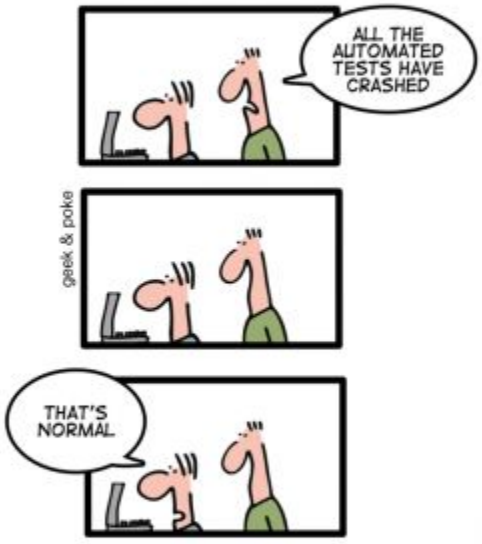
“Find the root cause of flaky behavior by examining just 19% of the classes your tests cover, saving you time and effort!"



Source:https://dieswaytoofast.blogspot.com/2018/07/flaky-teststhe-bane-of-existence.html

Problem

- Flaky Tests - non-deterministic behaviour
- Detection ? - rerun - time and resource consuming
- Affects team productivity and software quality
- Developers waste time in investigating false issues
- Breaks trust in regression testing - faults in production
- Existing Detection methods provide partial solutions
- Categories such as Concurrency and Asynchronous Waits are unaddressed



Source:https://walmyrlimaesilv.medium.com/are-ui-tests-flaky-by-their-nature-3ee24bc45042

Motivation

- Flakiness root cause localisation - important and difficult
- Important: Better control of non-determinism - developers understand source of flakiness
    - Google reported that 16% of their tests manifested some level of flakiness, while more than 90% of their test transitions, either to failing or passing, were due to flakiness
- Difficulty:
    - reproduce failures
    - diversity in potential issues
    - large scope of potential culprits

Proposed Solution and Practical Significance

- Identify classes responsible for non-deterministic behaviour of flaky tests
- Retarget FL techniques to achieve the same
- Specifically SBFL, change history metrics and static code metrics
- Helps with analysis of codebase and flaky tests
- Helps with investigating scenarios causing flakiness

Research Questions

RQ1: Are SBFL-based approaches effective in identifying flaky classes?

RQ2: How do code and change metrics contribute to the identification of flaky classes?

RQ3: How can ensemble learning improve the identification of flaky classes?

RQ4: How does an SBFL-based approach perform for different flakiness categories?

Approach - Data collection

- **Search** -
  - First set - 902 Java projects selected on the basis of number of commits, contributors, stars, releases, issues, and files
  - Second set - 187 projects available in the IDFLAKIES dataset
  - Gathered 16501 commits with the word 'flaky'
- **Inspection** - Filtered commits based on the keywords: fix, repair, solve, correct, patch, prevent.
  - filter further to keep the ones where fix affects the code under test and atomic changes
- **Test execution** - select commits that are usable for SBFL (runnable test suite to extract coverage matrix)
- **Extraction** - For each collected flakiness fixing commit, retrieve the source code, the test suite, the fixed flaky test, and the flaky class.

TABLE I: Collected Data. *ffc:* number of flakiness-fixing commits. *all:* number of commits in the project.

Proj.	**#Commits**		**#Tests**		**#Classes**	
	ffc	all	min - max	avg	min – max	avg
Hbase	8	18,990	138 - 2,089	1,113	734 – 1366	1053.4
Ignite	14	27,903	15 - 1,018	174	72 – 1767	1262.3
Pulsar	10	8,516	194 - 1,326	626	171 – 422	259.7
Alluxio	3	32,560	315 - 694	473	131 – 817	360.3
Neo4j	3	71,824	21 - 5,782	2,139	40 – 1663	581.3
Total	38		15 - 5,782	905	40 – 1767	820.2

Study Design - RQ1

- Separate tests into 2 groups - flaky and stable
- Classes covered by more flaky tests and fewer stable tests - higher chance of test flakiness
- Run test suits for each commit to obtain SBFL suspicion scores for Ochiai, Barinel, Tarantula and DStar
- Use genetic programming to evolve to a new formula to combine all four SBFL formulae
- Fitness function - average ranking of flaky classes
- Individual - candidate formula generated using six arithmetic operators

Study Design - RQ2

Augment SBFL with metrics

- Metric collection: calculate flakiness, change and size metrics using commits, change history, classes, code analysis results etc.
- Ranking model: GP to generate models that combine:
  - SBFL + flakiness
  - SBFL + change
  - SBFL + size
- Input: SBFL scores + metrics, Output: ranking for each candidate class
- Compare performance

TABLE III: Code and change metrics used to augment SBFL.

	Metric	Definition
Flakiness	#TOPS	Number of time operations performed by the class.
	#ROPS	Number of calls to the `random()` method in the class.
	#IOPS	Number of input/output operations performed by the class.
	#UOPS	Number of operations performed on unordered collections by the class.
	#AOPS	Number of asynchronous waits in the class.
	#COPS	Number of concurrent calls in the class.
	#NOPS	Number of network calls in the class.
Change	Changes	Number of unique changes made on the class.
	Age	Time interval to the last changes made on the class.
	Developers	Number of developers contributing to the class.
Size	LOC	The number of lines of code.
	CC	Cyclomatic complexity.
	DOI	Depth of inheritance.

Study Design - RQ3

Ensemble method:

- Voting between models: SBFL + change metrics(30), SBFL + size metrics(30)

- Candidate selection: all models compute suspicion scores for individual class and select Top N

- Voting: Each model votes for their own Top N candidates

- Votes are aggregated and most voted candidates are obtained

Study Design - RQ4

- Investigate performance of SBFL among different Flakiness categories
- Two authors manually analyse commits separately and assign them one of the flakiness categories derived by Luo et al.[1]
- Adopted existing taxonomy as reference

[1] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, vol. 16-21-November-2014, nov 2014, pp. 643–653.

Evaluation Metrics

Metrics: Accuracy and wasted effort

- Accuracy (acc@n) calculates the number of cases where the flaky classes were ranked in the top n. (n= 1,3,5 and 10)

- Wasted effort (wef), allows to measure the effort wasted while searching for the flaky class:

$$wef = |susp(x) > susp(x*)| + |susp(x) = susp(x*)|/2 + 1/2$$

- in addition to the two absolute metrics, they measure the relative effort as:

$$R_{wef} = \frac{100 \times (wef + 1)}{\text{\# of classes covered by flaky tests}}, \; 0 < R_{wef} \leq 100$$

Study Results - RQ1

TABLE IV: RQ1: Effectiveness of SBFL formulæ. (#) denotes the total number of flaky commits for each project. The row *Perc* contains the percentage of flaky commits whose triggering flaky classes are ranked in the top n; these values are computed only for *acc@n*.

Proj. (#)	**Dstar**						**Ochiai**						**Tarantula**						**Barinel**					
	acc				wef (R_{wef})		acc				wef (R_{wef})		acc				wef (R_{wef})		acc				wef (R_{wef})	
	@1	@3	@5	@10	mean	med	@1	@3	@5	@10	mean	med	@1	@3	@5	@10	mean	med	@1	@3	@5	@10	mean	med
Hbase (8)	0	3	4	4	33.0 (17)	7 (5)	2	5	5	5	14.9 (13)	1 (4)	1	4	4	5	11.9 (12)	4 (4)	1	4	4	5	11.6 (12)	4 (4)
ignite (14)	0	2	2	2	214.7 (21)	31 (4)	0	3	3	4	212.0 (20)	20 (4)	0	3	3	4	177.1 (17)	20 (4)	0	3	3	4	175.1 (17)	20 (4)
Pulsar (10)	1	3	6	9	9.9 (21)	4 (6)	3	5	6	9	9.2 (13)	3 (6)	3	5	6	9	9.2 (13)	3 (6)	3	5	6	9	9.2 (13)	3 (6)
Alluxio (3)	0	0	0	1	60.7 (43)	72 (31)	0	0	0	1	71.0 (46)	72 (41)	0	0	0	0	92.7 (59)	73 (58)	0	0	0	0	105.3 (66)	87 (65)
Neo4j (3)	1	2	2	2	12.0 (41)	1 (18)	1	2	2	2	12.0 (41)	1 (18)	1	2	2	2	23.0 (43)	1 (18)	1	2	2	2	23.7 (43)	1 (18)
Total (38)	2	10	14	18	94.4 (24)	**11 (17)**	6	15	16	21	90.2 (21)	7 (6)	5	14	15	20	79.3 (21)	8 (7)	5	14	15	20	79.6 (21)	8 (7)
Perc (%)	5	26	37	**47**	-	-	**16**	**39**	42	**55**	-	-	13	37	39	**53**	-	-	13	37	39	**53**	-	-

Using SBFL, we were able to localise flaky classes by inspecting only 21-24% (6-7%) of classes covered by flaky tests on average (median). With Ochiai, flaky classes are ranked at the top and in the top 10 for 16% and 55% of total flaky commits.

TABLE V: RQ1: The effectiveness of GP evolved formulæ using Ochiai, Barinel, Tarantula, and DStar.

Project	**Total**	**acc**				**wef (R_{wef})**	
		@1	@3	@5	@10	mean	med
Hbase	8	1	4	5	5	13.12 (16)	2.5 (5)
Ignite	14	0	3	3	5	214.93 (21)	20.0 (4)
Pulsar	10	3	5	6	9	9.20 (23)	3.0 (9)
Alluxio	3	0	0	0	1	101.67 (65)	86.0 (83)
Neo4j	3	1	2	2	2	23.33 (43)	1.0 (18)
Total	38	5	14	16	22	94.24 (26)	6.5 (8)
Percentage (%)	100	**13**	**37**	42	**58**	-	-

Study Results - RQ2

TABLE VI: RQ2: The contribution of flakiness, change, and size metrics to the identification of flaky classes.

Proj. (#)	**SBFL & flakiness**						**SBFL & change**						**SBFL & size**					
	acc				wef (R_{wef})		acc				wef (R_{wef})		acc				wef (R_{wef})	
	@1	@3	@5	@10	mean	med	@1	@3	@5	@10	mean	med	@1	@3	@5	@10	mean	med
Hbase (8)	1	4	5	5	11.9 (12)	3 (4)	2	4	4	5	16.9 (13)	4 (4)	2	4	5	5	11.4 (12)	3 (3)
Ignite (14)	0	2	2	4	230.9 (26)	63 (4)	2	4	4	4	222.3 (24)	18 (4)	1	3	3	5	220.1 (24)	43 (4)
Pulsar (10)	2	5	6	8	10.2 (15)	3 (8)	3	5	7	9	8.0 (12)	2 (5)	2	5	7	9	6.9 (13)	2 (6)
Alluxio (3)	0	0	1	1	97.7 (51)	73 (65)	0	0	1	1	75.7 (49)	94 (39)	0	0	1	1	90.7 (49)	77 (58)
Neo4j (3)	1	2	2	2	19.3 (42)	1 (18)	2	2	2	2	6.7 (37)	0 (9)	2	2	2	2	23.0 (40)	0 (10)
Total (38)	4	13	16	20	99.5 (24)	8 (8)	9	15	18	21	94.1 (21)	5 (6)	7	14	18	22	94.3 (22)	5 (7)
Percentage (%)	**11**	34	42	**53**	-	-	**24**	39	47	55	-	-	**18**	37	47	58	-	-

The augmentation of Spectrum-Based Fault Localisation with change or size metrics lets more flaky classes be ranked near the top; by adding change metrics, we can rank 24% flaky classes at the top. In contrast, metrics specific to flakiness categories do not provide any beneficial signals to the identification approach.

Study Results - RQ3

Project	Total	acc				wef (R_{wef})	
		@1	@3	@5	@10	mean	med
Hbase	8	3	5	6	6	9.62 (12)	1.5 (2)
Ignite	14	2	4	4	4	228.61 (24)	17.5 (4)
Pulsar	10	3	6	7	9	7.30 (12)	2.0 (5)
Alluxio	3	1	1	1	2	61.83 (22)	9.0 (10)
Neo4j	3	1	2	2	2	19.67 (42)	1.0 (18)
Total	38	10	18	20	23	94.61 (19)	3.5 (5)
Perc (%)	100	26	**47**	53	61	-	-

Improvement in the accuracy at the top 3 - reaches 47%

Median drops to 3.5

A voting between models based on SBFL, change, and size metrics, provides the best ranking for flaky classes. 47% of flaky classes are ranked in the top 3 and 26% of them are ranked at the top. The average R_{wef} further reduces to 19, highlighting the practical usefulness of our approach.

Study Results - RQ4

Flakiness Category	acc				wef (R_{wef})	
	@1	@3	@5	@10	mean	med
Concurrency (16)	6 (**38**)	7 (44)	7(44)	8 (**50**)	147.53 (27)	9.5 (9)
Async wait (10)	3 (**30**)	6 (60)	8 (80)	8 (**80**)	21.05 (8)	1.5 (3)
Ambiguous (4)	1 (25)	2 (50)	2 (50)	3 (75)	18.88 (5)	3.5 (5)
Time (3)	0 (0)	0 (0)	0 (0)	1 (**33**)	88.33 (16)	14.0 (10)
Network (2)	0 (0)	2 (100)	2 (100)	2 (100)	1.00 (10)	1.0 (10)
Unordered collections (2)	0 (0)	1 (50)	1(50)	1 (50)	331.5 (33)	331.5 (33)
I/O (1)	0 (0)	0 (0)	0(0)	0 (**0**)	12.50 (3)	12.5 (3)
Random (1)	0 (0)	1 (100)	1 (100)	1 (100)	2.00 (75)	2.0 (75)
Total (39[4])	10	18	20	23	94.47 (19)	3.5 (5)
Perc (%)	26	47	53	61	-	-

The most prominent flakiness categories, Concurrency and Asynchronous Waits, are identified effectively, with 38% and 30% of their flaky classes ranked at the top, respectively. In the Concurrency category, flaky classes are identified by examining 8% of classes covered by flaky tests on average.

Novelty

- Tools such as RootFinder and Flakiness Debugger relied on differences between passing and failing executions of flaky tests to localise flakiness, the researchers here explore a new direction by analysing the differences between flaky and stable tests.

- In this paper, the authors do not focus on any specific flakiness category and their analysis is based on the test coverage instead of environmental factor

- Assessment of the benefits obtained by combining SBFL with change, size and flakiness metrics to locate flaky classes

- First to leverage open-source software to localise flakiness root cause

Assumptions

- Classes covered by more flaky tests and fewer stable tests have a higher chance to be responsible for test flakiness

- Data collection process involves filtering commits using the keyword ‘flaky’ based on the assumption that flakiness-fixing commits identify classes that are responsible for flakiness.

- It cannot be certain that (i) the flakiness fix is effective, and (ii) the modified class is the one responsible for flakiness

Limitation

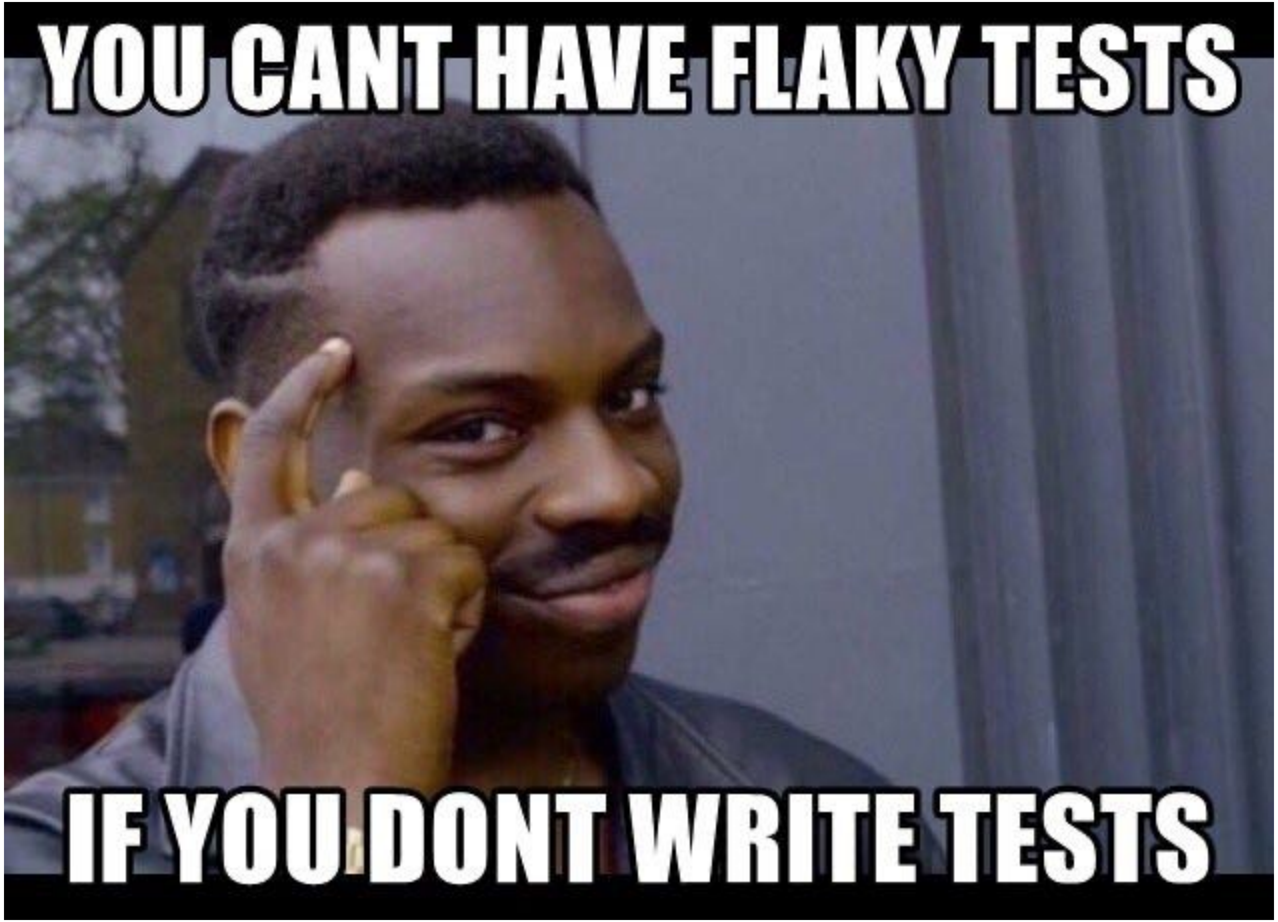
SBFL scores are calculated using the status of the tests (flaky or stable) determined by the developer. However, it is difficult to reproduce flaky behaviour because flaky tests can pass or fail for the same version of the program. Coverage of both the pass and the fail status could result into different SBFL results and therefore, if made available, the evaluation of the study can change.

Discussion

- Since, SBFL suffers from test-suite adequacy issue which is due to the lower DDU (Density, Diversity and Uniqueness) values of the test suite, can we utilise any other FL techniques that do not face the same issue? And help in better localisation of the flaky classes.

- The authors combine flakiness metrics with SBFL that did not improve (infact worsened) the performance of best performing SBFL technique (Ochiai). We feel that the reason it happened was because simply counting operations might miss nuances like timing issues, specific sequences of events, or particular states that lead to flakiness. Some of the metrics that we feel will help are: **number of unhandled exceptions, frequency and duration of lock contention events, latency to external service calls, number of shared variables** etc.

- It would be interesting to observe how machine learning techniques would perform for identifying flaky tests using a comprehensive dataset of code and change metrics, system metrics such as CPU load,memory usage etc. to encompass flakiness root cause across software(CUT) and hardware



Source: https://flexport.engineering/solving-flaky-tests-in-rspec-9ceadedeaf0e

~fin~