Test Flakiness

CS453 Automated Software Testing

Shin Yoo | COINSE@KAIST

Push On Green

- A DevOps concept popularised by Google, more commonly and also known as: Continuous Deployment (as in CI/CD)
- Newest version of your software is automatically deployed whenever all tests pass
- Test results are critical
  - False Negative (i.e., test passes when code is incorrect): you end up releasing an incorrect software
  - False Positive (i.e., test fails when code is correct): slows the development down

Making "Push On Green" a Reality: Issues and Actions Involved in Maintaining a Production Service

- A USENIX LISA 2014 presentation given by Daniel Klein, Google: https://www.usenix.org/conference/lisa14/conference-program/presentation/klein
- LISA stands for Large Installation System Administration Conference: the talk is very practical and informative :)

Test Flakiness

- We call a test case to be “flaky” when it changes outcome against the same code.
- This creates a huge problem for Pass on Green philosophy: when a test transitions from pass to fail, is it flaky or is it actually a real problem?

Analysis of Test Results at Google

- Analysis of a large sample of tests (1 month) showed:
  - 84% of transitions from Pass -> Fail are from "flaky" tests
  - Only 1.23% of tests ever found a breakage
  - Frequently changed files more likely to cause a breakage
  - 3 or more developers changing a file is more likely to cause a breakage
  - Changes "closer" in the dependency graph more likely to cause a breakage
  - Certain people / automation more likely to cause breakages (oops!)
  - Certain languages more likely to cause breakages (sorry)

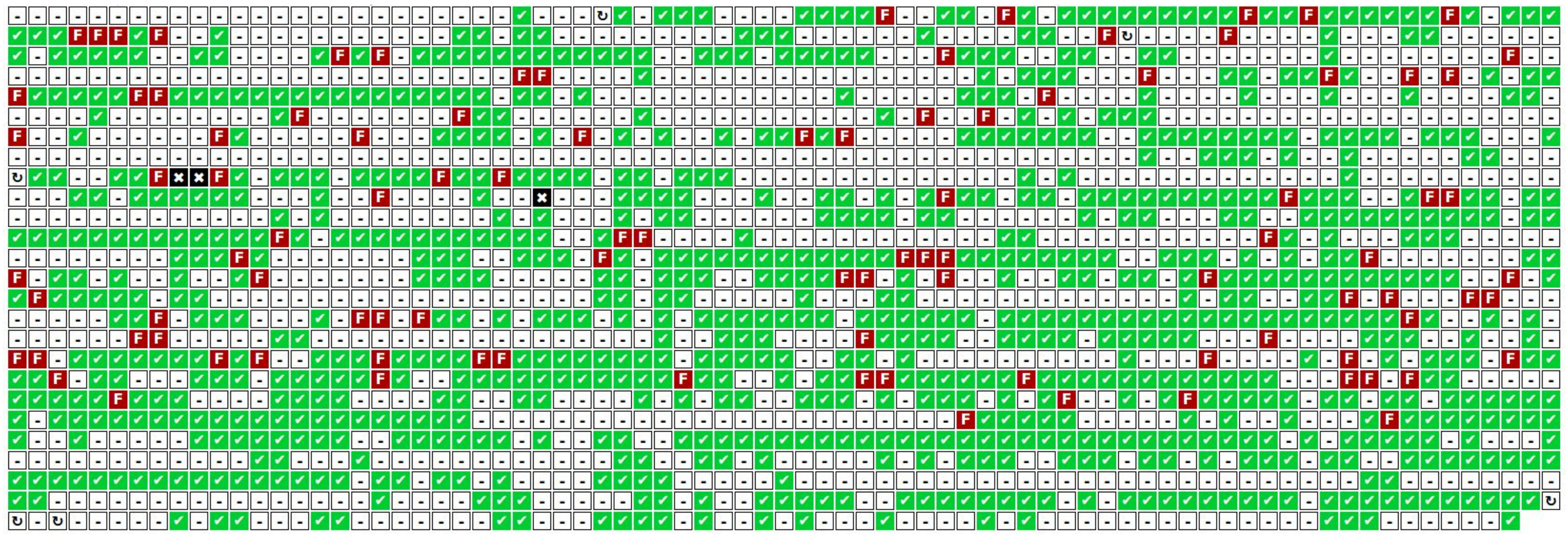
Google

See: prior deck about Google CI System, See this paper about piper and CLs

"The State of Continuous Integration Testing at Google", John Micco, ICST 2017 Keynote (https://research.google/pubs/pub45880/)

Flaky Tests

- Test Flakiness is a huge problem
- Flakiness is a test that is observed to both Pass and Fail with the same code
- Almost 16% of our 4.2M tests have some level of flakiness
- Flaky failures frequently block and delay releases
- Developers ignore flaky tests when submitting - sometimes incorrectly
- We spend between 2 and 16% of our compute resources re-running flaky tests



Google

"The State of Continuous Integration Testing at Google", John Micco, ICST 2017 Keynote (https://research.google/pubs/pub45880/)

Sources of Flakiness

- Parallelism: interference or poor synchronization
- Execution time: something takes too long and times out
- State management: poorly managed or not controlled
- Data management: poorly managed or not controlled
- Assertions: incorrect assertions
- Algorithm: nondeterministic choices

Solutions (?)

- Better synchronization
- Threadsafe code + independent execution environment
- Break-down long sequences + step-wise synchronization
- Explicit pre-condition setup for both state and data + avoid dependencies between test executions

“Your Tests Aren’t Flaky”

- A talk given by Alister Scott (Automattic) at GTAC 2015
- https://www.youtube.com/watch?v=hmk1h40shaE
- https://docs.google.com/presentation/d/1L9hGYqCAgjZyXE9ch4Toh4ziuYYkB2OiMCdFpgfTko0/pub?slide=id.gd8d3f5279_0_0 (slides)

Research on Test Flakiness

- Detection: is this test failure real, or a result of flakiness?
- Prediction: how likely is this test case to be flaky?
- Repair: automatically remove flakiness? (probably the most ambitious goal)

Detection

- A test fails. How do you determine whether it is flaky or not? (Recall Regression Test Case Selection)
- A test case that transitions from pass to fail but does not cover any of the changed part is likely to be flaky (because the changed behavior is caused by the changed code)
- DeFlaker: Automatically Detecting Flaky Tests, Jonathan Bell; Owolabi Legunsen; Michael Hilton; Lamyaa Eloussi; Tifany Yung; Darko Marinov, ICSE 2018 (https://ieeexplore.ieee.org/abstract/document/8453104)

Table 1: Number of flaky tests found by re-running 5,966 builds of 26 open-source projects. We consider only new test failures, where a test passed on the previous commit, and report flakes reported by each phase of our RERUN strategies. DEFLAKER found more flaky tests than the Surefire or Fork rerun strategies: only the very costly Reboot strategy found more flaky tests than DEFLAKER.

		Test Methods in Project		Total New	Confirmed flaky by RERUN strategy			DEFLAKER labeled as: Flaky		DEFLAKER labeled as: Not Flaky	
Project	**#SHAs**	**Total**	**Failing**	**Failures**	**Surefire**	**+Fork**	**++Reboot**	**Confirmed**	**Unconf.**	**Confirmed**	**Unconf.**
achilles	227	337	77	242	13	14	230	225	4	5	8
ambari	500	896	7	75	52	71	74	74	0	0	1
assertj-core	29	6,261	2	3	2	2	2	2	0	0	1
checkstyle	500	1,787	1	1	0	0	0	0	0	0	1
cloudera.oryx	332	275	23	29	5	5	5	5	20	0	4
commons-exec	70	89	2	22	22	22	22	21	0	1	0
dropwizard	298	428	1	60	60	60	60	55	0	5	0
hadoop	298	2,361	365	1,081	284	865	1,054	1,028	25	26	2
handlebars	27	712	7	9	3	7	7	6	2	1	0
hbase	127	431	106	406	62	242	390	383	12	7	4
hector	159	142	12	87	0	74	79	72	4	7	4
httpcore	34	712	2	2	2	2	2	1	0	1	0
jackrabbit-oak	500	4,035	26	34	10	33	34	32	0	2	0
jimfs	164	628	7	21	21	21	21	15	0	6	0
logback	50	964	11	18	18	18	18	18	0	0	0
ninja	317	307	37	122	37	77	110	94	2	16	10
okhttp	500	1,778	129	333	296	305	310	231	0	79	23
oozie	113	1,025	1,065	2,246	42	2,032	2,244	2,234	0	10	2
orbit	227	86	9	86	84	85	85	73	0	12	1
oryx	212	200	38	46	14	14	46	14	0	32	0
spring-boot	111	2,002	67	140	73	107	135	135	3	0	2
tachyon	500	470	4	5	3	5	5	5	0	0	0
togglz	140	227	21	28	5	14	28	28	0	0	0
undertow	7	340	0	0	0	0	0	0	0	0	0
wro4j	306	1,160	114	217	39	96	99	80	8	19	110
zxing	218	415	2	15	15	15	15	15	0	0	0
26 Total	5,966	28,068	2,135	5,328	1,162	4,186	5,075	4,846	80	229	173

Table 2: Number of reruns required to confirm the flakes from Table 1, and the percent of flakes confirmed by reruns at each tier also confirmed by DeFlaker without any reruns required. If a flake was confirmed, we stopped rerunning it; we executed the three rerun strategies in the order listed.

	# Reruns to Find Flaky						% Also Found
Strategy	**1**	**2**	**3**	**4**	**5**	**Total**	**by DeFlaker**
Same JVM	994	90	38	24	16	1, 162(22.9%)	87.6%
New JVM	2, 913	32	32	19	28	3, 024(59.6%)	98.4%
Reboot	889	0	0	0	0	889(17.5%)	95.8%
All						5, 075(100.0%)	95.5%

Prediction

- Can we build a predictive model that can tell us whether a test case is likely to be flaky?
- What would be a good feature set?
- "FlakeFlagger: Predicting Flakiness Without Rerunning Tests", Abdulrahman Alshammari, Christopher Morris, Michael Hilton, and Jonathan Bell, ICSE 2021 (https://ieeexplore.ieee.org/abstract/document/9402098)

TABLE II: Complete list of features captured for test flakiness prediction. The Covered Lines Churn feature is represented in multiple forms based on the h values (number of the past commits). In our evaluation, we considered $h = 5, 10, 25, 50, 75, 100, 500$ and $10,000$

	Feature	Description
Test Smells	Indirect Testing	True if the test interacts with the object under test via an intermediary [24]
	Eager Testing	True if the test exercises more than one method of the tested object [24]
	Test Run War	True if the test allocates a file or resource which might be used by other tests [24]
	Conditional Logic	True if the test has a conditional if-statement within the test method body [25]
	Fire and Forget	True if the test launches background threads or tasks. [26]
	Mystery Guest	True if the test accesses external resources [24]
	Assertion Roulette	True if the test has multiple assertions [24]
	Resources Optimism	True if the test accesses external resources without checking their availability [24]
Numeric Features	Test Lines of Code	Number of lines of code in the test method body
	Number of Assertions	Number of assertions checked by the test
	Execution Time	Running time for the test execution
	Source Covered Lines	Number of lines covered by each test, counting only production code
	Covered Lines	Total number of lines of code covered by the test
	Source Covered Classes	Total number of production classes covered by each test
	External Libraries	Number of external libraries used by the test
	Covered Lines Churn	h-index capturing churn of covered lines in past 5, 10, 25, 50, 75, 100, 500, and 10,000 commits. Each value h indicates that at least h lines were modified at least h times in that period.

TABLE III: Prediction performance for FlakeFlagger, the vocabulary-based approach, and the hybrid combination of both. The hybrid approach builds a model with both FlakeFlagger's and the vocabulary-based approach's features. We show the number of True Positives, False Negatives, False Positives and True Negatives, Precision, Recall, and F1 scores per-project. The AUC value is calculated after each fold where the reported value is the overall averages of AUC values after all folds. Projects with zero F1 values have very low numbers of flaky tests (less than 3 per project), and illustrate known limitations of FlakeFlagger.

		Flaky by	FlakeFlagger							Vocabulary-Based Approach [12]							Combined Approach						
Project	**Tests**	**Reruns**	**TP**	**FN**	**FP**	**TN**	**Pr**	**R**	**F**	**TP**	**FN**	**FP**	**TN**	**Pr**	**R**	**F**	**TP**	**FN**	**FP**	**TN**	**Pr**	**R**	**F**
spring-boot	2,108	160	139	21	15	1,933	90%	87%	89%	134	26	703	1,245	16%	84%	27%	143	17	18	1,930	89%	89%	89%
hbase	431	145	129	16	32	254	80%	89%	84%	89	56	152	134	37%	61%	46%	130	15	33	253	80%	90%	84%
alluxio	187	116	116	0	0	71	100%	100%	100%	108	8	11	60	91%	93%	92%	116	0	0	71	100%	100%	100%
okhttp	810	100	52	48	159	551	25%	52%	33%	79	21	444	266	15%	79%	25%	46	54	104	606	31%	46%	37%
ambari	324	52	47	5	3	269	94%	90%	92%	36	16	121	151	23%	69%	34%	47	5	3	269	94%	90%	92%
hector	142	33	30	3	8	101	79%	91%	85%	13	20	23	86	36%	39%	38%	25	8	11	98	69%	76%	72%
activiti	2,043	32	10	22	43	1,968	19%	31%	24%	12	20	531	1,480	2%	38%	4%	7	25	34	1,977	17%	22%	19%
java-websocket	145	23	19	4	1	121	95%	83%	88%	23	0	74	48	24%	100%	38%	19	4	4	118	83%	83%	83%
wildfly	1,023	23	11	12	27	973	29%	48%	36%	20	3	554	446	3%	87%	7%	17	6	24	976	41%	74%	53%
httpcore	712	22	14	8	23	667	38%	64%	47%	16	6	375	315	4%	73%	8%	15	7	24	666	38%	68%	49%
logback	805	22	3	19	17	766	15%	14%	14%	10	12	259	524	4%	45%	7%	5	17	11	772	31%	23%	26%
incubator-dubbo	2,174	19	8	11	35	2,120	19%	42%	26%	11	8	813	1,342	1%	58%	3%	13	6	23	2,132	36%	68%	47%
http-request	163	18	12	6	6	139	67%	67%	67%	16	2	84	61	16%	89%	27%	12	6	6	139	67%	67%	67%
wro4j	1,135	16	4	12	2	1,117	67%	25%	36%	2	14	101	1,018	2%	12%	3%	0	16	1	1,118	0%	0%	0%
orbit	86	7	1	6	8	71	11%	14%	12%	6	1	32	47	16%	86%	27%	1	6	7	72	12%	14%	13%
undertow	183	7	2	5	8	168	20%	29%	24%	6	1	63	113	9%	86%	16%	3	4	8	168	27%	43%	33%
achilles	1,317	4	2	2	3	1,310	40%	50%	44%	0	4	0	1,313	0%	0%	0%	0	4	0	1,313	0%	0%	0%
elastic-job-lite	558	3	0	3	0	555	0%	0%	0%	0	3	34	521	0%	0%	0%	1	2	0	555	100%	33%	50%
zxing	345	2	0	2	2	341	0%	0%	0%	1	1	144	199	1%	50%	1%	0	2	2	341	0%	0%	0%
assertj-core	6,261	1	0	1	5	6,255	0%	0%	0%	0	1	6	6,254	0%	0%	0%	0	1	0	6,260	0%	0%	0%
commons-exec	55	1	0	1	1	53	0%	0%	0%	1	0	18	36	5%	100%	10%	0	1	1	53	0%	0%	0%
handlebars.java	420	1	0	1	5	414	0%	0%	0%	0	1	91	328	0%	0%	0%	0	1	0	419	0%	0%	0%
ninja	307	1	0	1	3	303	0%	0%	0%	0	1	50	256	0%	0%	0%	0	1	0	306	0%	0%	0%
Total	21,734	808	599	209	406	20,520	60%	74%	66%	583	225	4,683	16,243	11%	72%	19%	600	208	314	20,612	66%	74%	86%
AUC (Average per fold)			86%							75%							86%						

Lexical Analysis Approach

- If sources of flakiness are limited to a few typical ones (network related latency, external resources not ready, file I/O, etc), do they manifest themselves with specific lexical patterns?
  - G. Pinto, B. Miranda, S. Dissanayake, M. d'Amorim, C. Treude, and A. Bertolino. What is the vocabulary of flaky tests? MSR 2020, pages 492–502
  - Static flaky test prediction essentially becomes text classification

Table 3: Classifier performance

algorithm	precision	recall	F_1	MCC	AUC
Random Forest	**0.99**	0.91	**0.95**	**0.90**	**0.98**
Decision Tree	0.89	0.88	0.89	0.77	0.91
Naive Bayes	0.93	0.80	0.86	0.74	0.93
Support Vector	0.93	**0.92**	0.93	0.85	0.93
Nearest Neighbour	0.97	0.88	0.92	0.85	0.93

Is it test case or test execution?

An et al., ICSME 2024

- What if all (or most of) your tests deal with database connection? Lexical analysis at the test code level will lose precision 🫠
- Instead, we can focus on individual failure, and lexically analyse the **symptoms** (stack traces, error messages…)
- We match observed symptoms to a set of known flaky symptoms - but abstract details (such as IP address) for more accurate matching.

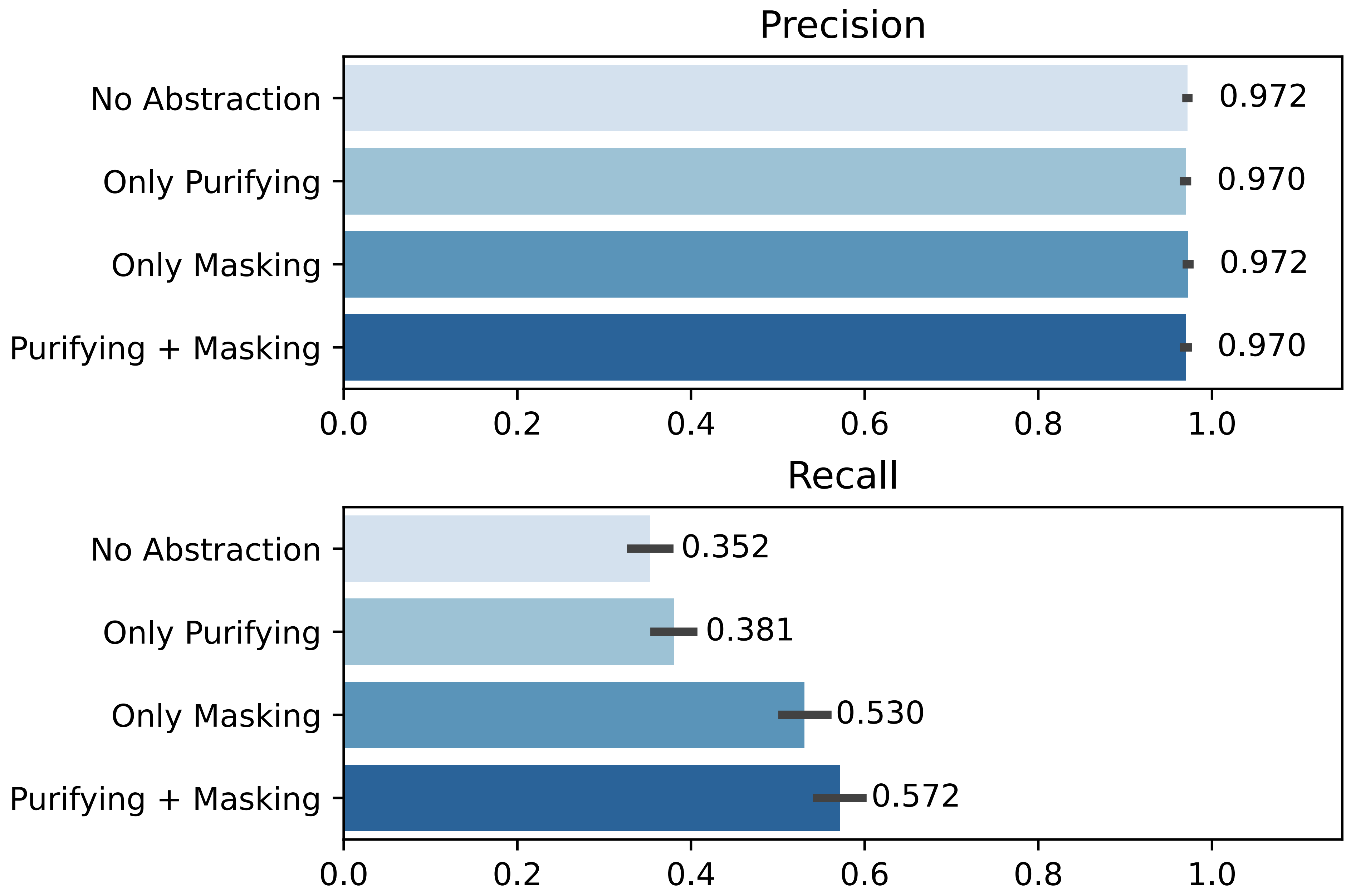


Figure 6: Precision (upper) and Recall (lower) for each abstraction setting: without abstraction, after only purifying stack trace, after only masking numbers, and after both purifying stack trace and masking numbers. The results are averaged over all hyperparameter settings (mean).

Automated Repair of Flakiness

FlakeSync (ICSE 2024)

- Specifically handles asynchronous flakiness by ensuring execution order (see the example on the right: the added lines are the fix by FlakeSync)
- How do we find where to insert such a guard? Critical sections are the points where an injected delay can cause test failures :)

```
  public class GrpcServerTest {
    @Test
    public void testGrpcExecutorPool() {
      GRPCMetrics gm = GRPCMetrics.getEmptyGRPCMetrics();

      GrpcThreadPoolExecutor executor =
        new GrpcServer.GrpcThreadPoolExecutor(gm);
      ...
      executor.submit(...);
      ...
+     while (!GGrpcThreadPoolExecutor.hasExecuted) {
+       Thread.yield();
+     }
      Thread.sleep(120);
      double activeThreads = gm.getGaugeMap().get(THREADS);

      assertEquals(2, activeThreads);
      double queueSize = gm.getGaugeMap().get(QUEUE);
      assertEquals(1, queueSize);
      ....
    }
  }

public GrpcThreadPoolExecutor {
    ...
    public GrpcThreadPoolExecutor {
      private final GRPCMetrics gm;
      public GrpcThreadPoolExecutor(GRPCMetrics gm) {
        this.gm = gm;
      }
    @Override
    protected void beforeExecute(Thread t, Runnable r) {
      gm.incGauge(THREADS);
      gm.setGauge(QUEUE, getQueue().size());
+     hasExecuted = true;
      super.beforeExecute(t, r);
    }
  }
```

Figure 1: Example flaky test from apache/incubator-uniffle

Summary

- Test flakiness is a simple yet extremely important problem in industry (especially under CI/CD practice).
- Empirical evidence suggests that, as long as you automate your test, you probably cannot avoid flakiness entirely.
- Solving it will require testable design that considers flakiness from the early development stage.
- There are research that tries to detect, predict, and repair flakiness.