To Seed or Not to Seed? An Empirical Analysis of Usage of Seeds for Testing in Machine Learning Projects

Saikat Dutta, Anshul Arunachalam, Sasa Misailovic
University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
{saikatd2,anshula2,misailo}@illinois.edu

***Abstract*—Many Machine Learning (ML) algorithms are inherently random in nature – executing them using the same inputs may lead to slightly different results across different runs. Such randomness makes it challenging for developers to write tests for their implementations of ML algorithms. A natural consequence of randomness is *test flakiness* – tests both pass and fail non-deterministically for same version of code.**

Developers often choose to alleviate test flakiness in ML projects by setting seeds in the random number generators used by the code under test. However, this approach commonly serves as a "workaround" rather than an actual solution. Instead, it may be possible to mitigate flakiness and alleviate the negative effects of setting seeds using alternative approaches.

To understand the role of seeds and the feasibility of alternative solutions, we conduct the first large-scale empirical study of the usage of seeds and its implications on testing on a corpus of 114 Machine Learning projects. We identify 461 tests in these projects that fail without seeds and study their nature and root causes. We try to minimize the flakiness of a subset of 42 identified tests using alternative strategies such as tuning algorithm hyper-parameters and adjusting assertion bounds and send them to developers. So far, developers have accepted our fixes for 26 tests.

We further manually analyze a subset of 56 tests and study various characteristics such as the nature of test oracles and how the seed settings evolve over time. Finally, we provide a general set of recommendations for both researchers and developers in the context of setting seeds in tests.

I. Introduction

The extensive success of Machine Learning has led to its widespread adoption across several critical domains such as autonomous driving, natural language processing, and medical diagnosis. These domains implement applications that utilize different ML algorithms such as Deep Learning [31], Reinforcement Learning [43], and Probabilistic Programming [32, 35]. Consequently, this has led to the development of a rich ecosystem of Machine Learning libraries and tools that solve tasks at varying levels of specializations. However, bugs in the implementation of these tools can lead to catastrophic consequences for the end users, often amounting to loss of lives and property [33, 68].

Testing implementations of ML algorithms is challenging. Many ML algorithms are *inherently random* in nature – multiple executions of the algorithm with same inputs and configurations may often lead to varying results. Moreover, the lack of proper test oracles further complicates the testing scenario. A natural consequence of randomness is *test flakiness*, i.e., when a test passes and fails non-deterministically for the same version of code. Test flakiness undermines the reliability of test results and puts additional burden on developers for investigating test failures (even in absence of bugs). Flakiness makes it difficult for developers to distinguish real test failures due to programming errors (*bugs*) from noisy executions due to randomness.

To minimize test flakiness, developers need to make various non-trivial choices such as choosing the optimal hyper-parameters for the ML algorithm under test [24] and a reasonable assertion bound [26]. However, without a systematic approach, it is hard for a developer to get these settings right. Hence, to mitigate flakiness, they tend to set the *seeds* for the *random number generators* that are used by the code under test. Setting the seeds can make the test execution deterministic and alleviate the developer from dealing with randomness. However, it is unknown whether this is *always* the best approach or if there are alternative ways to mitigate flakiness. Setting seeds can also lead to unintended consequences. For instance, fixing the seed(s) limits the sequence of computations exercised by the code under test. Hence, developers may potentially miss bugs in code under test that are triggered by other sequences, thus reducing the fault-detecting effectiveness of the test [25].

Prior work [25] has shown that algorithmic randomness is a major contributor to flakiness in ML projects. Further, it is known that tests for ML algorithms (prone to flaky failures) are typically more time-consuming than other tests in the suite, often consuming more than 80% of test time [24]. This makes it important to study such tests – and the role of seeds – in greater depth and scale than previous works.

Our Work. In this work, we conduct the first large-scale and systematic study of the usage of seeds (for random number generators) and its implications for testing in Machine Learning projects. We study several research questions and provide insights that can be useful for both developers and researchers: 1) How prevalent are tests that fail non-deterministically without seeds and how often they fail? (Section IV), 2) Can we use alternative strategies to mitigate flakiness instead of setting seeds for such tests? (Section V), and 3) What are the common characteristics of these failing tests? (Section VI).

We conduct an empirical study on a corpus of 114 Python projects from the Machine Learning domain. We develop a tool, XSEED, which automatically installs each project, runs each test a pre-specified number of times (500 in our evaluation)

in two modes: *with seeds* and *without seeds*, and generates a report summarizing the failed tests, the different types of failures, and the failure rates. Overall, we find 461 unique tests across 32 projects that fail when seeds are removed but always pass when seeds are present.

Although developers commonly try to mitigate the effects of randomness by setting seeds [25], this is often a "workaround" rather than an actual solution. Instead, it may be possible to limit randomness (and test flakiness) and alleviate the negative effects of setting seeds using statistical techniques. To evaluate this intuition, we select a subset of tests that only fail when seeds are removed and attempt to fix them, i.e., minimize their flakiness, using alternative techniques such as tuning the hyper-parameters used for the algorithm under test [24], adjusting the bounds of the assertions in the test [26], or refactoring the test/assertion in other ways. We sent 16 Pull Requests and 7 Issues (for cases where we were unable to fix the test) to the developers. These covered a total of 42 tests. At the time of writing this paper, developers have accepted our changes or fixed 26 tests, while the rest are pending resolution.

We further analyze a subset of 56 tests, discuss and categorize various characteristics such as nature of test oracles, source(s) of randomness, evolution of seeds relative to the tests, and how the seeds are set. We also provide a general set of *recommendations* based on our experience for using (or not using) seeds (more details in Section VI):

- Use *fixed* seeds only for tests checking for exact reproducibility of some functionality in their code.
- Randomize and log the seeds in other tests for non-deterministic algorithms to allow both reproducibility of failures and diverse executions.
- Use Test Re-runs on failure instead of setting seeds to mitigate random failures.
- Determine optimal test settings such as hyper-parameters for the algorithm(s) under test and/or assertion bounds to minimize flakiness.

Finally, we discuss the impact of setting (or not setting) seeds and alternative fixes on the fault-detecting ability of tests on several examples (Sections VII-B,VII-C).

Contributions. We make the following contributions:

- We conduct the first large-scale empirical study of the usage of seeds in tests in 114 Machine Learning projects.
- We analyze the tests that fail without seeds and study important aspects related to the nature of such tests and the root causes for flakiness.
- We apply various alternative strategies (instead of setting seeds) to fix the root causes for 42 tests and mitigate flakiness.
- We provide several insights and implications related to usage of seeds and a general set of recommendations for both developers and researchers.

Our source code and replication package are available at https://github.com/uiuc-arc/xseed.

II. Background

We describe previous research on flaky tests in Machine Learning projects.

A. Common Test Structure in Machine Learning Projects

```
1 def test_MLAlgo():
2     [[setup code]]
3     trainer = MLAlgo( P_1 = v_1, P_2 = v_2, ..., P_k = v_k )
4     trainer.train()
5     metrics = trainer.compute_metrics()
6     for i in range(len(metrics)):
7         assert metrics[i] >= expected[i]
```

Listing 1: Common Test Pattern in ML projects

The tests that developers write for testing the correctness of their implementations of stochastic Machine Learning algorithms typically emulate the training (or fitting) process. Listing 1 presents the common structure of such tests, previously identified by Dutta et al. [24]. In this test, Line 2 contains setup code that performs basic initialization steps such as loading data-set(s), creating the execution environment, or setting up other configurations (such as seeds). Line 3 initializes the Machine Learning algorithm (`MLAlgo`) using a set of values $(v_1, \ldots, v_k)$ for arguments $(P_1, \ldots, P_k)$ known as *hyper-parameters*. These hyper-parameters influence both the accuracy and performance of the ML Algorithm. Lines 4-5 then perform the training step and compute one or more accuracy or performance metrics. Lines 6-7 check if the computed metrics (`metrics[i]`) are greater or equal to expected values (`expected[i]`). In our study, we also find that most tests exhibit similar structure.

B. Minimizing execution time of tests for ML Algorithms

Selecting optimal hyper-parameter values for ML algorithms is non-trivial. Quite often developers end up being conservative – they execute the ML algorithm for long enough cycles so that the test is highly likely to pass (less *flaky*). However, this makes the test more expensive to execute and increases the overall build time. TERA [24] is an approach to optimize such tests (i.e., reduce their execution time) without making the test more flaky or reducing its fault-detecting ability.

TERA formulates this trade-off of test execution time and its flakiness as a Stochastic Optimization problem over the space of hyper-parameter values. In particular, TERA uses Bayesian Optimization – an instance of Stochastic Optimization, for their solution approach. TERA constructs an objective function thta encodes the optimization problem that it needs to solve. More formally, given a test $T : \emptyset \mapsto \{0, 1\}$ and algorithm hyper-parameters $\boldsymbol{\theta} = (P_1, ..., P_k)$, TERA transforms the test to an equivalent variant: $T' : \boldsymbol{\theta} \mapsto \{0, 1\}$. TERA defines a function $\mathit{TPP} : (T', \boldsymbol{\theta}) \mapsto [0, 1]$ that computes the passing probability of T' when run with given hyper-parameters $\boldsymbol{\theta}$. It also defines function $\mathit{Time} : (T', \boldsymbol{\theta}) \mapsto \mathbb{R}^+$ that returns the execution time of the test using the selected hyper-parameters. Finally, it optimizes the following objective function:

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta} \in U_1 \times \ldots \times U_k}{\operatorname{argmin}} \mathit{Time}(T', \boldsymbol{\theta})$$
$$\text{s.t. } \mathit{TPP}(T', \boldsymbol{\theta}) \geq \alpha$$

where U_i is the domain of P_i, $i \in 1, \ldots, k$ and α is the user provided threshold that specifies the minimum test passing

probability. The authors apply TERA on 160 tests across 15 ML projects and obtain a geo-mean run-time improvement of 2.23x for $\alpha = 0.99$.

C. Fixing Flaky Tests in Machine Learning Projects

An alternative way to fix the flaky tests is by adjusting the assertion bounds, such as the expected values in Listing 1, Line 6. *FLEX* [26] is an approach for systematically selecting such bounds using techniques from statistical extreme value theory (EVT). EVT is a popular approach used for modeling extreme events such as floods or market crashes. EVT models the tail distribution of the observed samples. The user then queries the tail distribution to derive extreme values such as 99th or 99.99th percentile. The crucial property of EVT is that, in the limit, the tail distribution converges to a specific group of probability distributions.

III. Methodology

A. Selection of projects

For our study we require projects that test various stochastic/non-deterministic algorithm implementations. Such projects are more likely to use seeds during testing to avoid flakiness. Hence, we select projects from the domains of Machine Learning and Probabilistic Programming. For selecting projects in these domains, we follow a similar methodology as Dutta et al. [26]. We select two Machine Learning frameworks: PyTorch [50, 56] and TensorFlow [63], and four Probabilistic Programming libraries: Pyro [12], NumPyro [49], TensorFlow-Probability [19], and PyMC3 [53]. We search for Python projects that depend on these six main libraries. For this task, we use GitHub's API to search for the dependent projects. We only select projects that can be installed as a Python library (known as "packages") and have at least 10 stars on GitHub. This allows us to eliminate toy projects and select projects that are more likely to have good test suites, relatively more popular, and have an active developer base. To limit our study to a reasonable number of projects, we only select top 100 dependent projects per library for our study.

Using this methodology, we selected 305 unique projects. We use a general installation script to install these Python libraries [26]. This script installs a general set of system-level packages. It processes the project files and creates a list of required dependencies for the project. It then creates a virtual Python environment using Anaconda [18] and installs the project and all its dependencies. However, in many cases the installation may fail due to incomplete specifications in the project files. For each project, we try to install the project using this script and check if we can run the tests successfully using `pytest`. Overall, we were able to run 114 projects.

B. Running and detecting flaky tests

In this work we aim to study tests that are affected by the seeds set for various random number generators that are used by the code under test. To find such tests, we run the existing tests in each project in two modes: *with seeds* (using the original version of the test) and *without seeds* (by removing all seed-setting statements). We then identify the tests that always pass when run with seeds but fail (at least once) without seeds.

To automate this task, we developed a tool: XSEED. XSEED takes as input the GitHub slug of the project, the number of times (N) to run each test, the number of threads to run in parallel (K), and a timeout (T) for executing the entire test-suite. XSEED then performs the following tasks. XSEED installs the project in a new Conda environment using the setup script described in Section III-A. It runs each test N times in the project using `pytest` and collects the test execution logs. XSEED parallelizes the runs by using K threads, with each thread running all the tests in the default order. XSEED uses the specified timeout T to limit the maximum allowable running time for the test-suite (i.e., for a single thread).

XSEED then searches for all seed-setting code in the project and replaces them with *pass* (equivalent of *skip* in Python). In particular, XSEED searches for the API calls across various libraries that provide random number generators. Some libraries like PyTorch and TensorFlow provide multiple APIs to set seeds. Such APIs may also change across library versions. Hence, XSEED searches for invocations to all such APIs in the project. Table I presents the list of all APIs used by XSEED. XSEED then re-runs the tests using the same settings but the seeds *removed* and collects the execution logs.

XSEED parses the test execution logs from the runs with and without seeds and returns a summary report containing the list of tests that failed and frequency of each kind of failure (e.g., `AssertionError`, `ValueError`) per test per project. We use this report for further analyses.

TABLE I: Seed Setting APIs

Library	API
Numpy	`random.seed`
TensorFlow	`random.set_seed`
TensorFlow	`set_random_seed`
TensorFlow	`random.set_random_seed`
TensorFlow	`compat.v1.random.set_random_seed`
PyTorch	`manual_seed`
PyTorch	`cuda.manual_seed_all`
PyTorch	`seed`
Random (Python)	`seed`

C. Analyzing results

We select a subset of tests that always pass with seeds but fail at least once without seeds for manual analysis. For each selected test, we try to identify various characteristics such as the source of randomness, nature of the code under test, and how the seeds are set. We also study historical features of each test such as when were the seed(s) set relative to the test creation date and how often were the test settings (such as seeds or assertions) changed by the developers. For each studied characteristic, we determine appropriate categorizations.

For this analysis, one author independently analyzes each test and determines appropriate categorizations. Then another author double-checks each test and its categorizations to mitigate any inaccuracies. Finally, the authors discuss together and collate all the results. We discuss the results of this analysis in Section VII.

D. Fixing failing tests

We select a subset of tests that never fail when the seeds are set but fail at least once when the seeds are removed. We try to fix such tests using alternative strategies:

1) Adjusting Hyper-parameters: The flakiness of a test can be minimized by tuning the hyper-parameters of the stochastic ML algorithm under test. We manually identify such hyper-parameters for the algorithm under test, select appropriate value ranges for each, and run TERA to find optimal values that minimizes the flakiness.
   TERA originally uses minimization of the test execution time as the objective function. We adopt their approach to minimize the flakiness of the test (instead of execution time). We adjust the objective function used by *TERA* as:

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta} \in U_1 \times \ldots \times U_k}{\operatorname{argmin}} \; (1 - \mathit{TPP}(T, \boldsymbol{\theta}))$$

2) Changing Assertion Bounds: A test can also be fixed by adjusting the *range* of acceptable values used in the test for comparing the end-result – the assertion *bound*. For this strategy, we use FLEX [26] to fix the flaky tests that we find. We use this strategy either when no suitable hyper-parameters are available or when *TERA* fails to find suitable hyper-parameters that can reduce the flakiness to an acceptable level.
3) Refactoring the Assertion or Test: In some cases, the test may be fixed by refactoring the assertion (other than changing the bounds). For instance, developers may use *exact* equality checks in their assertions that are only likely to pass when using fixed seeds. These tests can be fixed by refactoring the assertion to instead check whether the results fall into some bounded range.
   The test may also be fixed by refactoring the test in other ways such as changing a non-deterministic input for a model/algorithm into a fixed input.
4) Fix Code under Test: Removing the seeds may also reveal a bug in the code under test. In these scenarios, we need to debug and determine fixes based on the context.

If we are able to fix the test using one of the strategies discussed above and minimize the flakiness to an acceptable level (e.g., $< 1\%$), we send a pull request to the developers. Otherwise, we send an issue (bug report) to discuss possible solutions with the developers. For each project, we initially start by sending one pull request or issue for a single test to gauge developer interest and not bother them with multiple requests if they are not willing to accept such changes. If we get a positive response, we raise further pull requests/issues for remaining tests in the project (if any). For each case, we provide them additional information regarding how we found the flaky test, how we fixed it, and how many times we ran the test to verify the fix. In each case, we also discuss with developers if they are willing to remove the original seed settings. If developers agree, we update the pull request accordingly and also incorporate additional suggestions they provide.

E. Research Questions

We address the following research questions in this work.

RQ1: How many and how often do tests fail without seeds?
RQ2: What kind of assertions are used in the failing tests?
RQ3: How do we fix the tests to mitigate the (flaky) test failures?
RQ4: How do developers respond to our fixes?

We address RQs 1-2 in Section IV and RQs 3-4 in Section V.

F. Experimental Setup

We run all experiments on Azure machines (`Standard_F32s_v2` configuration) with 3.4GHz Xeon CPUs with 32 cores and 64GB memory. For each project, we run XSEED with 20 threads (K) in parallel. We run each test-suite 500 times (N) with a timeout of one hour (T) for each complete test-suite run.

We run *TERA* to find optimal hyper-parameter values to minimize flakiness for several tests. For *TERA*'s convergence tests we use a threshold of 0.1 for the Geweke Diagnostic, maximum iterations of 500, and batch sizes of 30. For the optimization, we set the maximum iterations at 5000. We use the tool implementation provided by the authors [67].

We also run *FLEX* to find assertion thresholds to fix the flaky tests, where applicable. We use similar experiment settings as the authors with 50 `MIN_TAIL_SAMPLES` and 0.05 `SIGNIFICANCE_LEVEL`. We use the tool implementation provided by the authors [29].

IV. Empirical Results

A. RQ1: Test behavior without seeds

We run the tests in 114 projects both with and without all seed-setting code using XSEED (Section III-B). Out of these, in 30 projects the tests timed-out. We exclude those projects from our study and use the results from the remaining 84 projects. Table II presents the details of results for a subset of these projects. Each row of this table represents one project. Column **Project** is the name of the project. Column **#Tests** is the number of tests in each project. Column **Total Failures** is the total number of tests that failed (at least once out of 500) in the project with or without seeds. Sub-column **ws** is the number of failed tests when run with seeds. Sub-column **wos** is the number of failed tests when run without seeds. Sub-column **wos-uniq** is the number of tests that always passed with seeds but failed at least once with seeds. Table II only presents the results of projects with at-least one such failing test. Overall, there are 32 such projects. In 52 projects, we did not find any additional test failures when seeds are removed.

Columns 6-10 present the failure rate statistics for tests in **wos-uniq** category. Column **[0%, 5%)** is the number of tests with failure rate between 0-5 (exclusive)%. Similarly, columns **[5%, 10%)**, **[10%, 50%)**, and **[50%, 100%)** are the number of tests with failure rates of 5-10%,10-50%, and 50-100% respectively. Column **100%** is the number of tests with a failure rate of 100%. The second last row presents the totals per column for 32 projects. The last row presents the summary for all 84 projects that did not time-out. Columns

TABLE II: Running Tests With and Without Seeds

Project	#Tests	Total Failures			[0%, 5%)	[5%, 10%)	[10%, 50%)	[50%, 100%)	100%
		ws	wos	wos-uniq					
Accenture/AmpliGraph	86	11	11	8	3	1	1	1	2
quantumlib/Cirq	10101	27	49	22	3	0	11	7	1
GPflow/GPflow	1003	0	13	13	13	0	0	0	0
ziatdinovmax/GPim	7	0	1	1	0	0	0	0	1
google/TensorNetwork	9224	37	186	149	7	1	4	0	137
SeldonIO/alibi-detect	1166	12	34	25	18	2	5	0	0
bambinos/bambi	88	16	33	17	15	2	0	0	0
pytorch/captum	769	0	102	102	23	9	31	30	9
thinkingmachines/christmAIs	32	5	5	1	1	0	0	0	0
autorope/donkeycar	65	3	5	2	2	0	0	0	0
google/dopamine	137	11	13	2	0	0	0	0	2
RaRe-Technologies/gensim	0	12	13	4	2	0	2	0	0
tensorflow/graphics	2200	17	18	1	0	0	0	0	1
learnables/learn2learn	57	6	1	1	1	0	0	0	0
magenta/magenta	354	4	6	2	2	0	0	0	0
Unity-Technologies/ml-agents	36	17	18	1	1	0	0	0	0
uber/orbit	246	0	25	25	0	0	0	0	25
josejimenezluna/pyGPGO	13	0	1	1	0	1	0	0	0
quantopian/pyfolio	80	8	9	1	0	0	1	0	0
exoplanet-dev/pymc3-ext	93	6	11	5	4	1	0	0	0
pymc-devs/pymc4	1334	20	24	4	4	0	0	0	0
jettify/pytorch-optimizer	346	0	25	25	23	1	1	0	0
tensorflow/ranking	502	23	24	2	0	0	0	0	2
refnx/refnx	227	6	16	10	1	0	0	0	9
datamllab/rlcard	208	5	6	2	1	0	0	0	1
YosefLab/scvi-tools	69	4	5	1	1	0	0	0	0
snorkel-team/snorkel	250	22	32	10	2	4	3	1	0
danielegrattarola/spektral	90	7	4	2	1	0	1	0	0
autonomio/talos	8	3	3	1	1	0	0	0	0
explosion/thinc	21	1	4	4	0	0	0	0	4
EpistasisLab/tpot	7	0	1	1	0	0	0	0	1
lmcinnes/umap	139	1	17	16	14	1	1	0	0
Total/Avg (32 projects)	28958	284	715	461	143	23	61	39	195
Overall (84 projects)	43783	1800	2226	461	236	49	82	54	1805

6-10 in last row present the breakdown of all failing tests without seeds (2226).

Overall, we observe that 1800 tests fail when run with seeds whereas 2226 tests fail when seeds are not used across all 84 projects. In the smaller subset of 32 projects, the number of such tests are 284 and 715 respectively. 461 tests fail in these projects when the seeds are removed but always pass with seeds. Out of these tests, 195 tests consistently fail (i.e., 500 failures out of 500 runs) whereas 266 tests are *flaky* (i.e., they non-deterministically pass or fail). Out of the failing tests, 227 of them have a failure rate of less than 50% whereas 39 of them have a failure rate of more than 50% (but less than 100%). These results show that a significant number of tests depend on seeds set in their random number generators to control the randomness during testing and avoid test failures.

Common Failure Types. We observe that the majority of tests fail due to `Assertion Error` (394 out of 461), which is expected since most of these tests contain approximate assertions that compare the result(s) of non-deterministic computations against a fixed value or range. When the seeds are removed, such assertions are more likely to fail. Other common failure types include `ValueError`(40) and `FileNotFoundError`(17). We present a detailed analysis of the tests that fail due to assertion errors in Section VI.

Tests with 100% failure rates. We observe that 195 tests fail 100% of the time across 12 projects. We investigate a subset of such tests and determine the most common causes:

- *Exact reproducibility*: Some tests check if two successive calls to the same/similar API produce the *exact* same result when starting from the same seed. Such tests are intended to test whether certain computations that depend on random number generators are reproducible.

Listing 2 shows an example of such a test in *google/TensorNetwork*. TensorNetwork [34, 57] is a library that provides implementations of high dimensional data-structures (called Tensor Networks) used in domains like quantum mechanics. TensorNetwork provides a wrapper API for backends like Numpy so that it can handle custom data types defined in TensorNetwork. In Listing 2, Line 2 initializes the Numpy backend. Lines 3 and 4 then make two successive calls to the `randn` API to create a random matrix of size 4×4 for the specified data type (`dtype`). Both use the same seed (10). Line 5 checks whether two matrices are close up to a certain precision level. The `randn` API internally calls *Numpy* random number generator. Hence, the test checks whether generating two matrices starting from the same seed and data type are equal. Evidently, this test is likely to almost always fail if the seeds are removed since the probability of producing

```
1 def test_randn_seed(dtype):
2     backend = numpy_backend.NumPyBackend()
3     a = backend.randn((4, 4), seed=10, dtype=dtype)
4     b = backend.randn((4, 4), seed=10, dtype=dtype)
5     np.testing.assert_allclose(a, b)
```

Listing 2: Example test in *google/TensorNetwork*

the same random matrix in successive calls is very low.

- *Testing for exact equality*: Since seeds make the end result deterministic, developers sometimes add assertions checking whether the final result is *exactly* equal to the expected value. However, without seeds, the computations could lead to slightly different results that causes these assertions to fail. In these scenarios, the test may be fixed by using an approximate assertion that checks whether the result is *close enough* to the expected values.
- *Too strict assertion bounds*: Developers may sometimes specify a very strict assertion bound/tolerance level in their tests that only works when specific seeds are used but not for most cases when the data/sequence of computations are non-deterministic. Such tests may be fixed by choosing a *looser* bound or tolerance level.
- *Bug*: A test that always fails without seeds may also be indicative of a bug. We identify one such scenario in *google/TensorNetwork* [64] where the test often produces an empty array as the end result whereas the expected result is an array of size 1.

Tests with low failure rates. We observe that several tests have low failure rates: 143 tests have a failure rate of less than 5%, 23 tests have a failure rate of 5-10%. The low failure rates indicate that many of these tests can likely be fixed using minor adjustments such as adjusting some hyper-parameters (like iterations) or by updating assertion bounds by a modest amount (Section III-D). In Section V, we describe how we fix some of such tests.

Insights and Implications. We discover a large number of tests that fail without seeds, many of which have high failure rates (195 tests always fail). This indicates developers often use seeds to suppress highly unstable tests instead of properly *fixing* them. The only exception to this are tests that check for exact reproducibility where setting seeds is necessary. We later explore alternative ways to fix such flakiness (Section V).

B. RQ2: Assertions used in the failing tests

Table III presents the number of each kind of assertion that are used in the tests that failed without seeds. For this analysis, we only consider the tests that failed due to an assertion failure. Some tests may have more than one type of assertion that fails, hence the total number of assertions is more than the number of failing tests.

Overall, the most common assertion is Python's `assert` that is used across 137 tests. The developers also use assertion APIs provided by other frameworks like *numpy* (5 APIs used across 135 tests), *unittest* (9 APIs used across 123 tests), and *tensorflow* (5 APIs used across 7 tests).

Insights and Implications. We observe that the failing tests use both approximate (such as Numpy's assert_allclose) and exact equality assertions (such as Unittest's assertEqual). This indicates that developers need to transform both kinds of assertions to reduce flaky failures. For instance, they can change exact equality assertions into approximate ones. On the other hand, they can lower the strictness of approximate assertions (e.g., by reducing precision level).

TABLE III: Distribution of Assertions Used in Failing Tests

API Name	Framework	# of Assertions
assert	Python	137
assert_allclose	Numpy	112
assertAlmostEqual	Unittest	94
assertTrue	Unittest	13
assert_almost_equal	Numpy	12
assert_array_almost_equal	Numpy	9
assertEqual/assertNotEqual	Unittest	6
assertGreater/assertGreaterEqual	Unittest	5
assertLess/assertLessEqual	Unittest	5
assertEqual/assertAllEqual	TensorFlow	3
assertAllClose	TensorFlow	2
assertAlmostEqual	TensorFlow	1
assertIn	Unittest	1
assertLen	TensorFlow	1
assert_	Numpy	1
assert_array_equal	Numpy	1

V. Fixing failing tests

A. RQ3: Determining fix strategies for failing tests

We randomly sub-sample tests that only fail without seeds. For each selected test, we investigate and determine appropriate fix using the methodology described in Section III-D.

TABLE IV: Pull Requests/Issues Sent for Fixing Tests

Project	Tests	PRs	Issues	A	R	P
Accenture/AmpliGraph [11]	1	1	0	0	0	1
GPflow/GPflow [1]	1	1	0	0	0	1
SeldonIO/alibi-detect [10]	1	0	1	0	0	1
danielegrattarola/spektral [6]	1	0	1	1	0	0
google/TensorNetwork [64, 65, 66]	5	0	3	1	0	2
google/dopamine [21]	1	1	0	0	0	1
jettify/pytorch-optimizer [54, 55]	11	2	0	2	0	0
josejimenezluna/pyGPGO [4]	1	1	0	0	0	1
learnables/learn2learn [2]	1	0	1	0	0	1
lmcinnes/umap [8, 9]	2	2	0	0	0	2
pytorch/captum [13, 14, 15]	11	3	0	3	0	0
quantopian/pyfolio [52]	1	1	0	0	0	1
quantumlib/Cirq [16, 17]	2	2	0	1	0	1
snorkel-team/snorkel [60, 61]	2	2	0	0	0	2
ziatdinovmax/GPim [37]	1	0	1	0	0	1
Total	42	16	7	8	0	15

Table IV presents the details of all the Pull Requests (PRs) and Issues that we sent. Each row presents details of one project. Column **Tests** is the number of tests per project that we send PR/Issues for. Column **PRs** is the number of PRs that we send. Column **Issues** is the number of issues that we send. Column **A** is the number of PRs accepted (fixed by us) or Issues resolved (fixed by developers). Column **R** is number of PRs/Issues rejected. Column **P** is the number of PRs/Issues pending resolution. Overall, we send 16 PRs and 7 issues for 42 tests. So far, developers have accepted or fixed 8 (covering 26 tests), rejected none while 15 are pending resolution.

Table V presents the number of tests fixed (or not fixed) using each kind of strategy we applied for 42 tests. Overall, we fix most tests (27) by changing hyper-parameters. We fix 6 tests by updating the assertion bounds. In one case [1] we used a combination of both approaches for fixing the test, i.e., changing both hyper-parameters and assertion bounds since only tuning the hyper-parameters did not reduce the flakiness to an acceptable level.

For tests that we cannot automatically fix by adjusting hyper-parameters or assertion bounds, we manually investigate and determine if a different fix is possible. We fix 2 tests by only refactoring the assertion. In 1 case, we refactor the test to mitigate the failures. In 5 cases, we are not able to determine a fix. Hence, we only send issue (bug reports) to the developers for those cases.

TABLE V: Types of Fixes

Fix Type	#Tests
Changing Hyper-Parameters	27
Updating assertion bound	6
Refactoring Assertion	2
Refactoring Test	1
Fix code under test	2
Cannot fix/Unknown	5

```
1  def test_convergence(self):
2    random.seed(123)
3    np.random.seed(123)
4    torch.manual_seed(123)
5    dataloaders = create_data(N_TRAIN, N_VALID)
6    model = MultitaskClassifier(tasks=["task1", "task2"])
7   -trainer = Trainer(lr=0.001, n_epochs=10)
7   +trainer = Trainer(lr=0.00238, n_epochs=50)
8    trainer.fit(model, dataloaders)
9    for idx, task_name in enumerate(["task1", "task2"]):
10     val_loss = compute_val_loss(idx, task_name)
11     self.assertLess(val_loss, 0.05)
```

Listing 3: Fixed test in *snorkel-team/snorkel*

Example: Fixing by Changing Hyper-Parameters. Listing 3 presents an example flaky test (simplified) from *snorkel-team/snorkel* that we fixed by adjusting hyper-parameters. This is a test for *Multitask Classifier* model that simultaneously trains a model on multiple tasks. The test first sets seeds in three random number generators: *Random*, *Numpy* and *PyTorch* (Lines 2-4). The test initializes the data, model, and the trainer object (Lines 5-7). The trainer object uses two hyper-parameters: number of epochs (*default*: 10) and learning rate (*default*: 0.001). Then it trains the model (Line 8) and checks whether the validation loss is less than 0.05 (Line 11). However, when the seed settings (Lines 2-4) are removed, the test fails very frequently: 195 failures out of 500 runs. Using TERA, we find optimal hyper-parameters (`n_epochs`: 50, `lr`: 0.00238) that minimizes the failure probability of the test to 0.08%. We raise a PR for the fix [60].

Example: Fixing by updating assertion bounds. Listing 4 presents an example flaky test (simplified) from *quantumlib/Cirq*. Cirq is python library used for developing quantum circuits and running them using quantum simulators and quantum computers. `test_clifford_circuit_3` is a test for Cirq's implementation of *Clifford Simulator*, which is a special simulator used for efficient simulation of Clifford circuits (quantum circuits that contain Clifford gates [36]). This test, first, builds a random quantum circuit (Lines 3-5). Then, it initializes the Clifford simulator (Line 6) and the standard simulator (Line 7). Finally, it checks whether the end result (i.e., the final state vector) produced by the two simulators are close enough (Line 8). It also sets a seed (Line 4) to make the input circuit deterministic. However, when the seed is removed, the test turns out to be *flaky*: fails 61/500 times. We observed that reducing the precision of the comparison of the end result from 7 (default) to 6 decimal places reduced the failure rate to 0%. Our fix (including the seed removal) was accepted by the developers [17].

```
1  def test_clifford_circuit_3():
2    (q0, q1) = (cirq.LineQubit(0), cirq.LineQubit(1))
3    circuit = cirq.Circuit()
4   -np.random_seed(0)
5    circuit = create_random_circuit(circuit,
         ↪ random_clifford_gate(), q0, q1)
6    clifford_sim = cirq.CliffordSimulator()
7    std_sim = cirq.Simulator()
8    np.testing.assert_almost_equal(
9      clifford_sim.simulate(circuit).final_state_vector,
10     std_sim.simulate(circuit).final_state_vector, +6 )
```

Listing 4: Fixed test in *quantumlib/Cirq*

Example: Fixing by refactoring assertion. Listing 5 presents an example of a test (simplified) in *Accenture/AmpliGraph* project where we refactor the assertion for fixing the test. This is a test for a model selection algorithm (`select_best_model_ranking`) that uses random search to find optimal hyper-parameters for the given model and dataset. As input, it takes the model (`model`), the validation data (`X`), the specification of all parameters (`param_grid`), and the maximum parameter combinations to evaluate (`max_combinations:10`). Then it

```
1  def test_select_best_model_ranking_random():
2    best_model,history=select_best_model_ranking(model,X)
3    assert best_params['k'] in (2, 50)
4    assert np.log(1.00001)<=best_params['lr']<=np.log(100)
5   -assert set(i["param"]["k"] for i in history)=={2,50}
6   +assert all(i[param"]["k"] in [2, 50] for i in history)
```

Listing 5: Fixed test in *Accenture/AmpliGraph*

checks whether the results of the algorithm are within expected ranges (Lines 3-4). In Line 5, it checks whether all the values of parameter "k" were evaluated. However, since they use random search and bound it to `10` evaluations, this property cannot be guaranteed when the seed is not set (not shown here) – which makes the test flaky (failed 3/500 times). We refactor the assertion to instead check whether the evaluated values of "k" belong to the set: (`[2,50]`) and raise a PR [11].

B. RQ4: Developer Response

The developers mostly responded positively to our pull requests and issues. In general, they agreed that they typically set seeds when they are not able to stabilize the results of the computations in their tests or when computing the stable result makes the test unacceptably expensive. For instance, the developers of *GPflow/GPflow* commented on one of our proposed changes: "*Thanks for looking into this...! ... we ended up setting the seeds because, without setting a seed, we might get a test failing for no reason other than "bad luck"... but of course that means we risk missing issues like this one!*".

For fixing a test in *quantumlib/Cirq*, we initially discussed a solution with the developers and proposed a fix [16] by replacing a loop (of size 50), which was creating a quantum circuit with random gates, with a deterministic list of 50 gates. However, the developers responded that this reduces the readability/interpretability of the test. After another round of

```
1 def test_model_loss(self):
2     label_model = LabelModel(cardinality=2, verbose=False)
3     label_model.fit(data, n_epochs=1)
4     init_loss = label_model._loss_mu().item()
5     label_model.fit(data, n_epochs=10)
6     next_loss = label_model._loss_mu().item()
7     self.assertLessEqual(next_loss, init_loss)
```

Listing 6: Test Oracle: Comparison against Same Model

discussions, we updated the PR to just use a fixed local random seed using Numpy's *RandomState* API instead of setting the global seed for Numpy. The advantage of using this API is that it has guaranteed legacy support, i.e., its implementation will not be changed in future Numpy versions [3]. This reduces the chances of the test failing due to any implementation changes.

For a test in *snorkel-team/snorkel* [60], we updated the hyper-parameters to reduce flakiness. The developers agreed with our changes but rejected our proposal of removing seeds since they wanted to keep the test deterministic for better reproducibility.

In *google/TensorNetwork* project, we discovered a test (`test_max_truncation_error`) that always failed when the seeds were not set. However, the test did not contain any suitable hyper-parameters that we could adjust. The test's assertion always failed due to a dimension mismatch between the computed tensor values. Hence, updating the assertion bound also was not a solution. We reported this to the developers [64]. The developers acknowledged that this was due to an incorrect logic in a branch in the code under test that was leading to wrong results in the test. However, this bug was not discovered since the fixed data input (due to the seed) in the test did not trigger this branch. The developers fixed the underlying code and also updated the test to use and log different seeds instead of the previously fixed seed. This instance demonstrates that *setting seeds can sometimes hide buggy behavior causing the developers to miss them.*

VI. Analysis

We select a subset of 56 tests across 21 projects that only failed without seeds, for deeper analysis. We describe the various categories for each characteristic that we analyze.

A. Nature of Test Oracles

We characterize the nature of oracle used in the tests:

1) Comparing against same model but different state or configuration: This category includes the tests that compare the results of running the same model with and without some changes. For instance, Listing 6 shows the test `test_model_loss` in *snorkel-team/snorkel*. This test fits the same model (`LabelModel`) twice on the same data-set `data` and checks if training loss after 10 epochs is less than that after 1 epoch (Line 7). Overall, 7 tests fall in this category.

2) Comparing against different model: Tests in this category compare the results of running a model against a different kind of model (baseline). Listing 4 showed such an example from *quantumlib/Cirq* that compared the results of the specialized simulator (Line 6) with the standard (baseline) simulator (Line 7) and checked if their results are close up to a given precision (Line 8). Overall, 5 tests belong to this category.

```
1 def test_tSP_opt_nograd():
2     tsp = tStudentProcess(squaredExponential())
3     tsp.fit(X, y)
4     assert 0.3 < tsp.params['l'] < 0.5
```

Listing 7: Test Oracle: Comparison against Fixed Values

3) Comparing against fixed values: These tests compare the results of training or fitting a model against a fixed value or value range. Listing 7 shows such a test in *josejimenezluna/pyGPGO* that fits a model (`tStudentProcess`) on a dataset `(X,y)` and checks if the fitted parameter (`l`) fall in the specified range (Line 4). Overall, 44 tests fall in this category.

Insights and Implications. We observe that majority of the failing tests (44) compare against fixed values. This implies that developers often find it difficult to choose these values (also known as assertion bounds) which in turn forces them to use seeds in their tests to avoid flaky failures. Hence, developers should be more careful when choosing assertion bounds or use tools like FLEX to automatically find optimal values.

B. Introduction and Evolution of Seeds Relative to Tests

We look into commits between when the test was added and when it was last modified and study the evolution of seeds relative to the test. Overall, we find that for 23 tests, seeds were introduced in the same commit as the test. For 16 tests, the seeds were added after the test. In 17 tests, seeds were present before the tests were added. Further, in 20 tests, developers also modified the seeds in later commits.

Insights and Implications. We observe that developers often modify the seeds they set. This indicates that setting seeds may not always be the most reliable way of mitigating flakiness.

C. Sources of Randomness

We categorize the tests based on the source of randomness. In 19 cases, the randomness is only due to the algorithm under test. In 33 cases, the randomness is only due to generation of random data. In 4 cases, randomness is due to both.

Insights and Implications. We observe that the nature of the source of randomness does not have a strong correlation with flakiness. Rather, other test settings such as hyper-parameters or assertion bounds have a stronger impact on flakiness.

D. Seed Setting Location

Developers set seeds in tests in different ways, which effects the execution of tests differently. For instance, developers can set seeds at the *global* level, i.e., before executing all tests. Developers can set the seed at *module* level (or file level), i.e., before running tests in a file, or at *class* level, i.e., at the beginning of test class, or at *function* level, i.e., inside the test method. Out of 56 tests we analyze, developers set seeds at global level in 7 cases, at module level in 3 cases, at class level in 17 cases, and at test level in 29 cases.

Insights and Implications. We observe that developers mostly prefer setting seeds at the test level, which minimizes chances of flaky failures. Setting seeds at class level is useful when the class initialization code involves generating some random data that is shared among the tests in the class. Setting seeds

at higher levels (module or global) can potentially introduce implicit order dependencies between tests such that tests only pass for a specific set of orderings but fail for others. Future work may explore this aspect of setting seeds.

VII. Discussion

A. Should Seeds be Used in Tests?

Based on our experience with the tests and projects that we study, we develop a set of general recommendations or best practices for using (or not using) seeds for testing.

When to use fixed seeds. Developers should ideally use seeds when testing for *exact reproducibility* of some functionality in their code. For instance, this may include APIs that implement functionality/wrappers related to random number generators (such as Listing 2 from *google/TensorNetwork*). Another example is a test in *TensorFlow/ranking* [7] that tests a sorting algorithm that randomly shuffles ties. The test uses two different seeds to check whether the algorithm outputs two sequences with two different orderings of tied elements.

Randomize and Log seeds for variability and reproducibility. During our discussions with developers, some mentioned that they use fixed seeds in their tests for *better reproducibility* of test failures. A better approach might be to *randomize* and *log* the seed. This would ensure that the code under test exhibits different sequences of computations and test failures can still be reproduced. Interestingly, we find one such example in *pytorch/serve* [5]: `random.seed(datetime.datetime.now())`. Further, in one case, the developers randomized the seed after our bug report [64]. However, the test may also randomly fail (not due to a bug). We next discuss a strategy for mitigating this risk.

Use Test Re-run on failure instead of setting seeds. Instead of setting seeds, developers can choose to re-run the test on failure (e.g., using Python's *flaky* plugin [28]). This has a few distinct advantages: 1) CI builds will not be blocked due to intermittent failures, reducing the burden on developers, 2) intermittent failures can still be logged allowing developers to investigate them later, and 3) if test still fails after re-run(s), developers can use it as signal for immediate investigation. The expected cost of re-runs will be low if the test rarely fails [24].

Finding optimal test settings to minimize flakiness. In general, developers can use the available tools: TERA and FLEX, to find both optimal hyper-parameters and assertion bounds for their tests. In our study, we were able to fix a majority (more than 78%) of the selected tests using these two techniques (or their combination). The positive response for our fixes also demonstrates that developers welcome such changes. Future research can perhaps look into making these tools more approachable and cost-efficient for developers so that they can easily integrate them into their workflow.

B. Impact of Seeds on Fault-Detecting Ability

Setting seeds minimizes the randomness in the test and consequently the chance of flaky failures. But it can also limit the ability of the test to detect faults that are only exposed by a subset of potential sequences of random numbers. In this work, we found such an instance in *google/TensorNetwork* [64] project, where the fixed seed was hiding a truncation issue in the code under test (described in Section V). On removing the seed, the test failed 519 out of 1000 times exposing the issue. Dutta et al. [25] also reported a similar observation in *geomstats/geomstats* project where the test fails in 42 out of 1000 runs only when the seed is removed and exposes a bug. These instances show seeds can seriously impact the fault-detecting ability of tests. Hence, developers must be careful when dealing with randomness in tests and also consider alternative strategies, such as ones discussed in Section III-D, when possible.

A more comprehensive analysis of fault-detecting ability of the tests with and without seeds can be done using techniques such as mutation testing or by leveraging historical bugs – similar to the methodologies described in TERA [24] (Section 6). This is however beyond the scope of this work.

C. Impact of Fixes on Fault-Detecting Ability

On another hand, without seeds, tests can become flaky – making it difficult for developers to distinguish spurious failures from real ones. In this work, we used two alternative fixes for flakiness previously proposed in literature: tuning algorithm hyper-parameters and adjusting assertion bounds. However, they may also impact the fault-detecting ability of tests. Next we discuss their impact in more details.

Adjusting assertion bounds. Ideally, the assertion bound in a test should be loose enough to allow all valid executions to pass but catch all faulty executions. We explain this trade-off in the context of a test that we fixed by adjusting assertion bounds.

We inject an artificial fault (by changing a multiply operator to divide) in `LabelModel` model (label_model.py, Line 309) in *snorkel-team/snorkel* project. Listing 8 presents a test, `test_labeling_convergence`, for this model. This test fits the model on a small training dataset and checks if the fitting error (`err`) is below a fixed threshold (originally `0.05`). The test also sets seeds in three random number generators (Numpy, PyTorch, and Python's random). In its original form with the seeds set, the test never fails due to this injected fault. But when we remove the seeds, the test fails 76 out of 500 times. However, the test also fails 55 out of 500 times even if we do not inject any faults. This makes it difficult for developers to distinguish real failures from spurious (or noisy) ones.

```
1 def test_labeling_convergence(self) -> None:
2     random.seed(123)
3     np.random.seed(123)
4     torch.manual_seed(123)
5     err=compute_fit_error(LabelModel, data, iters=100,
          ↪ ...)
6     -self.assertLess(err, 0.05)
7     +self.assertLess(err, 0.06)
```

Listing 8: Example test in *snorkel-team/snorkel*

We fix this test using FLEX [26] by updating the assertion bound to 0.06 (the 99.99th percentile) from 0.05 (the 90th percentile). Now, the test only fails 1 in 500 times without any fault injections and 47 in 500 times with the injected fault. This shows that the new bound reduces the flakiness and

also retains the fault-detecting ability of the test. Further, we also verified that the empirical percentiles (calculated using 10,000 samples) are consistent with those calculated by FLEX.

Tuning algorithm hyper-parameters. Changing algorithm hyper-parameters in a test affects the test runtime, flakiness, and fault-detecting ability. Hence, developers need to consider these trade-offs when choosing suitable hyper-parameters. We use TERA [24] to select optimal values that minimize flakiness.

To evaluate the impact of changing hyper-parameters, we perform mutation testing of 4 tests from *pytorch/captum* (randomly chosen) that we fixed. We use the same methodology as in TERA: comparing the original tests (with seed) against the fixed tests (without seeds). We observed that the mutation score of the fixed tests is 14.98% – almost the same as the original tests, 15.02% – indicating that the fault-detecting ability is not reduced. This observation is in line with the results reported in TERA [24]. We refer the readers to TERA [24] (Sections 6, RQ2 and 7.1) for a more comprehensive discussion of its impact on fault-detecting ability of tests.

In conclusion, we observed that both fixes reduced flakiness while only making a minimal impact on the fault-detecting ability of the tests. Further, they can help developers in more precisely distinguishing real failures from noisy executions. Hence, these fixes serve as reliable alternatives to fixing seeds.

Choosing a fix. In some cases, there may be multiple ways to mitigate flakiness, each with its own trade-offs. For instance, adjusting hyper-parameters may make the test slower (e.g., by increasing iterations). If the increase in run-time is significant, developers may instead loosen assertion bounds since this does not affect test run-time. However, if the chosen bound is too loose (e.g., checking if error is less than 50%), it may reduce test effectiveness. Finally, if neither of the strategies are applicable, developers may choose to set the seeds to mitigate flakiness and retain test effectiveness to some extent.

While fixing the tests, we observed that adjusting hyper-parameters (in 27 tests) did not impact the test run-time significantly. Developers also responded positively to our changes.

D. Threats to Validity

The projects that we use for our empirical study only contain a subset of all machine learning projects. Hence, our results may not generalize well beyond the projects we study. To mitigate this risk, we start with popular ML and probabilistic programming frameworks and select their dependent projects that are also fairly popular (have at least 10 stars). Using this approach we find a large number of projects where seeds are used and tests that are affected when those seeds are removed. Hence, we believe that our results are representative.

Our analysis of tests may contain potential miss-categorizations. To minimize this risk, two authors of the paper jointly analyze the tests and determine the correct characterizations after mutual discussions. Our fixes for the tests using TERA and FLEX may not be optimal. For instance, TERA may find hyper-parameters that are not globally optimal in reducing the flakiness whereas FLEX may sometimes over-estimate the assertion bound required to minimize the flakiness. To minimize this risk, we use a high convergence threshold of 0.1 for TERA and high confidence threshold for FLEX's hypothesis tests. Further, we attach relevant information with each fix to allow developers to make any adjustments.

VIII. Related Work

Flaky Tests. Luo et al. [48] conducted the first systematic study on flaky tests. They studied flaky tests in Java open-source projects and discovered common causes and fixes. Researchers have also studied flaky tests specific to Python [38], Android [69], and Embedded Systems [62]. Researchers have developed techniques to detect flaky tests of specific kinds such as ones due to test-order dependencies [30, 46], concurrency [20], unordered collections [58], and asynchronous wait [44, 45]. Dutta et al. [25] conducted the first study of flaky tests in Machine Learning projects. They observed that the major cause of flakiness in this domain is due to algorithmic randomness. They developed FLASH [25] to detect such flaky tests. Researchers have developed various techniques for fixing flaky tests due to test-order dependencies [59], unordered collections [72], asynchronous waits [45], and algorithmic randomness [26].

Prior works on flaky tests in Machine Learning projects have made brief observations regarding usage and influence of seeds on testing [24, 25, 26]. However, their observations have been mostly limited to small number of projects and they do not directly address the problem of setting seeds or its general impact. In this work, we present the first large-scale study on how seeds are used and how they impact testing in this domain.

Testing non-deterministic or approximate software. Many emerging systems in domains such as ML and Probabilistic Programming exhibit non-deterministic behavior. These systems require specialized testing techniques to detect deep faults. Researchers have proposed methods for testing and debugging various non-deterministic systems such as ML frameworks [39, 41, 51, 70, 71, 73], Probabilistic Programming Systems [22, 23, 27, 47], and Approximation Algorithms [42]. On the other hand, Hariri et al. [40] proposed approximate transformations (e.g., loop perforation) for mutation testing of Java projects. They observed that such mutations can sometimes generate valid approximations (leading to surviving mutants) due to the presence of *approximable code*.

IX. Conclusion

We identified 461 tests across 114 projects that are flaky but are hidden due to developer-set seeds. This demonstrates that setting seeds is a common *workaround* used by many developers. We showed that it is possible to *fix* such tests using alternative strategies and mitigate test flakiness. We hope our study and insights will motivate developers in writing better tests and researchers in improving the fixing techniques and making them more accessible to developers.

Acknowledgements

This research was supported in part by NSF Grants No. CCF-1846354, CCF-1956374, CCF-2008883, USDA NIFA Grant No. NIFA-2024827, a gift from Facebook, a Facebook Graduate Fellowship, and Microsoft Azure Credits.

REFERENCES

[1] "Gpflow pr 1720," 2021, https://github.com/GPflow/GPflow/pull/1720.
[2] "Learn2learn issue 251," 2021, https://github.com/learnables/learn2learn/issues/251.
[3] "Numpy randomstate api," 2021, https://numpy.org/doc/stable/reference/random/legacy.html?highlight=randomstate#numpy.random.RandomState.
[4] "Pygpgo pr 34," 2021, https://github.com/josejimenezluna/pyGPGO/pull/34.
[5] "Pytorch/serve example," 2021, https://github.com/pytorch/serve/blob/ccaba6f66a1f4a00a594b1371e42d5749be4e35d/test/benchmark/tests/conftest.py#L120.
[6] "Spektral issue 273," 2021, https://github.com/danielegrattarola/spektral/issues/273.
[7] "Tensorflow ranking example," 2021, https://github.com/tensorflow/ranking/blob/019a7db68d83959b8774bc77bfb6905180504216/tensorflow_ranking/python/utils_test.py#L157.
[8] "Umap pr 770," 2021, https://github.com/lmcinnes/umap/pull/770.
[9] "Umap pr 773," 2021, https://github.com/lmcinnes/umap/pull/773.
[10] "Alibi-detect issue 256," 2021, https://github.com/SeldonIO/alibi-detect/issues/256.
[11] "Ampligraph pr 256," 2021, https://github.com/Accenture/AmpliGraph/pull/256.
[12] E. Bingham, J. P. Chen, M. Jankowiak, F. Obermeyer, N. Pradhan, T. Karaletsos, R. Singh, P. Szerlip, P. Horsfall, and N. D. Goodman, "Pyro: Deep Universal Probabilistic Programming," *Journal of Machine Learning Research*, 2018.
[13] "Captum pr 775," 2021, https://github.com/pytorch/captum/pull/775.
[14] "Captum pr 780," 2021, https://github.com/pytorch/captum/pull/780.
[15] "Captum pr 781," 2021, https://github.com/pytorch/captum/pull/781.
[16] "Cirq pr 4531," 2021, https://github.com/quantumlib/Cirq/pull/4531.
[17] "Cirq pr 4534," 2021, https://github.com/quantumlib/Cirq/pull/4534.
[18] "Conda package management system," 2021, https://docs.conda.io.
[19] J. V. Dillon, I. Langmore, D. Tran, E. Brevdo, S. Vasudevan, D. Moore, B. Patton, A. Alemi, M. Hoffman, and R. A. Saurous, "Tensorflow distributions," *arXiv preprint arXiv:1711.10604*, 2017.
[20] Z. Dong, A. Tiwari, X. L. Yu, and A. Roychoudhury, "Flaky test detection in android via event order exploration," 2021.
[21] "Dopamine pr 185," 2021, https://github.com/google/dopamine/pull/185.
[22] S. Dutta, Z. Huang, and S. Misailovic, "Sixthsense: Debugging convergence problems in probabilistic programs via program representation learning," *FASE*, 2022.
[23] S. Dutta, O. Legunsen, Z. Huang, and S. Misailovic, "Testing probabilistic programming systems," in *FSE*, 2018.
[24] S. Dutta, J. Selvam, A. Jain, and S. Misailovic, "Tera: Optimizing stochastic regression tests in machine learning projects," *ISSTA*, 2021.
[25] S. Dutta, A. Shi, R. Choudhary, Z. Zhang, A. Jain, and S. Misailovic, "Detecting flaky tests in probabilistic and machine learning applications," in *ISSTA*, 2020.
[26] S. Dutta, A. Shi, and S. Misailovic, "Flex: fixing flaky tests in machine learning projects by updating assertion bounds," in *FSE*, 2021.
[27] S. Dutta, W. Zhang, Z. Huang, and S. Misailovic, "Storm: program reduction for testing and debugging probabilistic programming systems," in *FSE*, 2019.
[28] 2021, https://github.com/box/flaky.
[29] "Flex tool," 2021, https://github.com/uiuc-arc/flex.
[30] A. Gambi, J. Bell, and A. Zeller, "Practical test dependency detection," in *ICST*, 2018.
[31] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*. MIT Press Cambridge, 2016.
[32] N. Goodman, V. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum, "Church: a language for generative models," *arXiv preprint arXiv:1206.3255*, 2012.
[33] "A google self-driving car caused a crash for the first time," *The Verge*, '16, https://bit.ly/2CNSeUZ.
[34] "Tensornetwork," 2021, https://github.com/google/TensorNetwork.
[35] A. D. Gordon, T. A. Henzinger, A. V. Nori, and S. K. Rajamani, "Probabilistic programming," in *FoSE*, 2014.
[36] D. Gottesman, "Theory of fault-tolerant quantum computation," *Physical Review A*, vol. 57, no. 1, p. 127, 1998.
[37] "Gpim issue 34," 2021, https://github.com/ziatdinovmax/GPim/issues/34.
[38] M. Gruber, S. Lukasczyk, F. Kroiß, and G. Fraser, "An empirical study of flaky tests in python," in *ICST*, 2021.
[39] Q. Guo, X. Xie, Y. Li, X. Zhang, Y. Liu, X. Li, and C. Shen, "Audee: Automated testing for deep learning frameworks," in *ASE*, 2020.
[40] F. Hariri, A. Shi, O. Legunsen, M. Gligoric, S. Khurshid, and S. Misailovic, "Approximate transformations as mutation operators," in *ICST*, 2018.
[41] Q. Hu, L. Ma, X. Xie, B. Yu, Y. Liu, and J. Zhao, "Deepmutation++: A mutation testing framework for deep learning systems," in *ASE*, 2019.
[42] K. Joshi, V. Fernando, and S. Misailovic, "Statistical algorithmic profiling for randomized approximate programs," in *ICSE*, 2019.
[43] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of artificial intelligence research*, 1996.
[44] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta, "Root causing flaky tests in a large-scale industrial setting," in *ISSTA*, 2019.
[45] W. Lam, K. Muşlu, H. Sajnani, and S. Thummalapenta, "A study on the lifecycle of flaky tests," in *ICSE*, 2020.
[46] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "iDFlakies: A framework for detecting and partially classifying flaky tests," in *ICST*, 2019.
[47] Y. R. S. Llerena, M. Böhme, M. Brünink, G. Su, and D. S. Rosenblum, "Verifying the long-run behavior of probabilistic system models in the presence of uncertainty," in *ESEC/FSE*, 2018.
[48] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *FSE*, 2014.
[49] "Numpyro," 2020, https://github.com/pyro-ppl/numpyro.
[50] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "PyTorch: An imperative style, high-performance deep learning library," in *NeurIPS*, 2019.
[51] H. V. Pham, T. Lutellier, W. Qi, and L. Tan, "Cradle: cross-backend validation to detect and localize bugs in deep learning libraries," in *ICSE*, 2019.
[52] "Pyfolio pr 683," 2021, https://github.com/quantopian/pyfolio/pull/683.
[53] "Pymc3," 2021, https://github.com/pymc-devs/pymc3.
[54] "Pytorch-optimizer pr 368," 2021, https://github.com/jettify/pytorch-optimizer/pull/368.
[55] "Pytorch-optimizer pr 369," 2021, https://github.com/jettify/pytorch-optimizer/pull/369.
[56] "Pytorch," 2021, http://pytorch.org.
[57] C. Roberts, A. Milsted, M. Ganahl, A. Zalcman, B. Fontaine, Y. Zou, J. Hidary, G. Vidal, and S. Leichenauer, "Tensornetwork: A library for physics and machine learning," 2019.
[58] A. Shi, A. Gyori, O. Legunsen, and D. Marinov, "Detecting assumptions on deterministic implementations of non-deterministic specifications," in *ICST*, 2016.
[59] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov, "ifixflakies: A framework for automatically fixing order-dependent flaky tests," in *FSE*, 2019.
[60] "Snorkel pr 1666," 2021, https://github.com/snorkel-team/snorkel/pull/1666.
[61] "Snorkel pr 1671," 2021, https://github.com/snorkel-team/snorkel/pull/1671.
[62] P. E. Strandberg, T. J. Ostrand, E. J. Weyuker, W. Afzal, and D. Sundmark, "Intermittently failing tests in the embedded systems domain," in *ISSTA*, 2020.
[63] "Tensorflow," 2021, https://www.tensorflow.org.
[64] "Tensornetwork issue 943," 2021, https://github.com/google/TensorNetwork/issues/943.
[65] "Tensornetwork issue 945," 2021, https://github.com/google/TensorNetwork/issues/945.
[66] "Tensornetwork issue 946," 2021, https://github.com/google/TensorNetwork/issues/946.
[67] "Tera tool," 2021, https://github.com/uiuc-arc/tera.
[68] "Understanding the fatal tesla accident on autopilot and the nhtsa probe," *electrek*, '16, https://bit.ly/2PtnDCZ.
[69] S. Thorve, C. Sreshtha, and N. Meng, "An empirical study of flaky tests in android apps," in *ICSME*, 2018.
[70] Z. Wang, M. Yan, J. Chen, S. Liu, and D. Zhang, "Deep learning library testing via effective model generation," in *FSE*, 2020.
[71] A. Wei, Y. Deng, C. Yang, and L. Zhang, "Free lunch for testing: Fuzzing deep-learning libraries from open source," in *ICSE*, 2022.
[72] P. Zhang, Y. Jiang, A. Wei, V. Stodden, D. Marinov, and A. Shi, "Domain-specific fixes for flaky tests with wrong assumptions on underdetermined specifications," in *ICSE*, 2021.
[73] Y. Zhang, L. Ren, L. Chen, Y. Xiong, S.-C. Cheung, and T. Xie, "Detecting numerical bugs in neural network architectures," in *ESEC/FSE*, 2020.