Detecting Flaky Tests in Probabilistic and Machine Learning Applications

Saikat Dutta
University of Illinois
Urbana, IL, USA
saikatd2@illinois.edu

August Shi
University of Illinois
Urbana, IL, USA
awshi2@illinois.edu

Rutvik Choudhary
University of Illinois
Urbana, IL, USA
rutvikc2@illinois.edu

Zhekun Zhang
University of Illinois
Urbana, IL, USA
zhekunz2@illinois.edu

Aryaman Jain
University of Illinois
Urbana, IL, USA
aryaman4@illinois.edu

Sasa Misailovic
University of Illinois
Urbana, IL, USA
misailo@illinois.edu

ABSTRACT

Probabilistic programming systems and machine learning frameworks like Pyro, PyMC3, TensorFlow, and PyTorch provide scalable and efficient primitives for inference and training. However, such operations are non-deterministic. Hence, it is challenging for developers to write tests for applications that depend on such frameworks, often resulting in *flaky tests* – tests which fail non-deterministically when run on the same version of code.

In this paper, we conduct the first extensive study of flaky tests in this domain. In particular, we study the projects that depend on four frameworks: Pyro, PyMC3, TensorFlow-Probability, and PyTorch. We identify 75 bug reports/commits that deal with flaky tests, and we categorize the common causes and fixes for them. This study provides developers with useful insights on dealing with flaky tests in this domain.

Motivated by our study, we develop a technique, *FLASH*, to systematically detect flaky tests due to assertions passing and failing in different runs on the same code. These assertions fail due to differences in the sequence of random numbers in different runs of the same test. FLASH exposes such failures, and our evaluation on 20 projects results in 11 previously-unknown flaky tests that we reported to developers.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**.

KEYWORDS

Probabilistic Programming, Machine Learning, Flaky tests, Randomness, Non-Determinism

ACM Reference Format:
Saikat Dutta, August Shi, Rutvik Choudhary, Zhekun Zhang, Aryaman Jain, and Sasa Misailovic. 2020. Detecting Flaky Tests in Probabilistic and Machine Learning Applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3395363.3397366

1 INTRODUCTION

With the surge of machine learning, randomness is becoming a common part of a developer's workflow. For instance, various training algorithms in machine learning use randomness (in data collection, observation order, or the algorithm itself) to improve their generalizability [21, 33, 60]. As another example, probabilistic programming [9, 10, 16, 25, 66] is an emerging framework for expressive Bayesian modeling with efficient inference. Most existing inference algorithms, such as Markov Chain Monte Carlo [44] and Stochastic Variational Inference [7] are inherently randomized.

The traits of algorithms in various machine learning applications, including inherent randomness, probabilistic specifications, and the lack of solid test oracles [5], pose significant challenges for testing these applications. Recent studies identify multiple classes of domain specific errors in frameworks for both deep learning [53] and probabilistic programming [18]. For instance, probabilistic programs provide distributions as outputs instead of individual values. Developers of projects that use probabilistic programming systems typically have to design tests that run an inference algorithm and check whether the outputs fall within some reasonable range or differ by only a small amount from the expected result. However, determining these thresholds or how many iterations to run is often non-intuitive and subject to heuristics, which may be either too liberal (e.g., assuming independence when one may not exist) or too conservative (e.g., running programs too many times).

As a result of non-systematic testing in this domain, many tests in machine learning and probabilistic programming applications end up being *flaky*, meaning they can non-deterministically pass or fail when run on the same version of code [8, 40, 65]. Furthermore, as developers evolve their code, they rely on regression testing, which is the practice of running tests after every change to check that said changes do not break existing functionality [47, 70]. Regression testing becomes more challenging if the tests can pass and fail even without any changes to the code.

Luo et al. previously investigated flaky tests in traditional software [40]. They identified various reasons for flaky tests and found that some of the most common causes include async wait, concurrency, test-order dependencies, and randomness. However, these studies did not investigate the flaky tests in projects that use probabilistic programming systems or machine learning frameworks, where the inherent probabilistic nature of such systems can yield a different spectrum of reasons for flaky tests in dependent projects, or even new reasons altogether. Furthermore, the way developers fix or mitigate flaky tests in these domains will also differ from how developers of traditional software address them.

Our Work. We present a technique for detecting flaky tests in projects using probabilistic programming systems and machine learning frameworks. While common wisdom would suggest that many problems with programs dealing with randomness could be solved by simply fixing the seed of the random number generator, this paper shows that fixing the seed is not always the best solution. Moreover, fixing the seed may not be sufficient for identifying bugs and can be brittle in the presence of program changes.

We conduct ***the first study of the common causes and fixes for flaky tests in projects that build upon and use probabilistic programming systems and machine learning frameworks.*** We perform an extensive study on 345 projects that depend on four of the most common open-source probabilistic programming systems and machine learning frameworks. We identify 75 bug reports and commits across 20 projects where the developers explicitly fix some flaky tests. We categorize these fixes based on (1) the cause of flakiness and (2) the fix patterns. Unlike Luo et al. [40], we find that projects that depend on probabilistic programming systems and machine learning frameworks have a *larger* percentage of flaky tests whose cause is what Luo et al. would refer to as *randomness.*

We do a more thorough analysis of the flaky tests due to randomness in this domain, breaking them down into more specific categories relevant to probabilistic programming systems and machine learning frameworks. We find that the majority of the causes for flaky tests (45 / 75) are due to Algorithmic Non-determinism, where tests have different results due to the underlying inference mechanism and sampling using probabilistic programming systems and machine learning frameworks. We also find that the most common fix pattern for flaky tests is to adjust the thresholds for the assertions that have flaky failures.

As developers are fixing flaky tests by adjusting their assertions' threshold, we develop **FLASH*, a novel technique for systematically detecting tests that are flaky due to incorrectly set thresholds in the assertions.*** These assertions fail due to differences in the sequence of random numbers in different runs of the same test. FLASH exposes such failures by running the tests using different seeds. FLASH reports which seeds lead to runs where an assertion fails, allowing the user to reproduce such failures.

A distinctive feature of FLASH is its use of statistical convergence tests to identify potentially flaky test executions. FLASH dynamically determines how many times to run each test and assertion with different seeds by sampling the actual values computed for each assertion and running a statistical test to determine if the sampled values have converged, indicating that FLASH has likely seen enough samples to form a distribution of the values. FLASH then reports the distribution of sampled actual values, providing the user with information on how to fix the flakiness in the test.

We run FLASH on the 20 projects that historically had flaky tests, as we found from our study. FLASH detects 11 previously unknown flaky tests on the latest version of these projects, and we submit 4 patches to fix 5 flaky tests and 6 bug reports for the remaining flaky tests to the developers. Developers confirmed and accepted our fixes for 5 flaky tests and confirmed 5 other flaky tests that are still pending fixes. The remaining 1 bug report is awaiting developer response. We also run FLASH on the historical versions of each project to target previously known flaky tests. We are able to detect 11 such flaky tests using FLASH.

Contributions. The paper makes several main contributions:

- We perform the first empirical study on flaky test causes and fixes for projects that depend on probabilistic programming systems and machine learning frameworks. We investigate 75 fixes for flaky tests within 20 projects, where the most common cause for flakiness is due to Algorithmic Non-determinism and the most common fix is to adjust the flaky assertion's threshold.
- We propose FLASH, a technique for systematically detecting flaky tests that fail due to differences in the sequences of random numbers required by computations running through a probabilistic programming system or machine learning framework.
- Our evaluation of using FLASH on 20 projects results in 11 detected flaky tests, and we submit 5 fixes and 6 bug reports for these flaky tests to developers. Developers confirmed and accepted 5 fixes and confirmed 5 other flaky tests. We also detect 11 previously known historical flaky tests using FLASH.

Our datasets and tool for this paper are available for open-access at https://github.com/uiuc-arc/flash.

2 EMPIRICAL STUDY

The goal of our empirical study is to understand the common causes and fixes for flaky tests in projects that depend on probabilistic programming systems and machine learning frameworks.

2.1 Evaluation Projects

Table 1: Project Statistics

	Pyro	PyMC3	TF-Prob.	PyTorch
First Commit	Jun 15 '17	Apr 13 '12	Feb 13 '18	Jan 25 '12
#Contributors	73	227	106	1246
#Commits	1823	7223	2254	23,263
#Dependents	4	24	27	290
Prog. Lang.	Python	Python	Python	Python, C++

Initially, we determine four well-known and commonly used probabilistic programming systems and machine learning frameworks: Pyro [9], PyMC3 [59], TensorFlow Probability [17], and PyTorch [51]. We choose these systems because they are open source, have a relatively long development history, and have a large user base. Table 1 presents the statistics for these four frameworks. The table shows for each framework its first commit date, number of contributors, number of commits up until Dec 20, 2019, number

Table 2: Details of Flaky Test Fixes in Dependent Projects

Project	Dependent Projects	Filtered Bug Reports	Filtered Commits
Pyro	1	8	7
PyMC3	1	0	2
TensorFlow-Prob	9	12	29
PyTorch	65	191	184
Total	76	211	222
With "test" keyword	33	110	105
Manual Inspection	**20**	**38**	**37**

of dependent projects[1] with more than 10 stars on GitHub, and the major programming language. For PyTorch, we list both Python and C++, because the core library is built using C++, but the API and several utilities are designed using Python.

2.2 Extracting Bug Reports and Commits

We start with the 345 projects (summing up the row #Dependents in Table 1) that depend on a probabilistic programming system or machine learning framework . We collect each project's bug reports (both the Issue Requests and Pull Requests on GitHub) and commits. We filter these bug reports/commits by searching for the following keywords on the bug reports' conversation text and commit messages: `flaky|flakey|flakiness|intermit|fragile|brittle`. Next, we filter out the bug reports and commits that do not reference the word `test`. This step removes most irrelevant bug reports and commits that do not deal with a flaky test directly. Finally, we manually inspect the remaining bug reports/commits to filter out false positives – bug reports/commits that actually are not related to a flaky test/fix but still match our keyword search. We inspect the bug reports first, because they usually contain a good description of the flaky test and a possible fix. On the other hand, commit messages often leave out details concerning why or how flaky tests are flaky. In some cases, the bug reports also have related fix commits for flaky tests. Afterwards, when inspecting the commits, we ignore the commits already referenced by the bug reports.

Table 2 shows the breakdown of our filtering process. We find 75 bug reports/commits related to fixing a flaky test across 20 projects.

2.3 Analyzing the Bug Reports and Commits

We divide the filtered bug-reports and commits among the authors of the paper. For each test fixed in a bug report or commit, we aim to classify the cause for the flaky test and the type of fix for the flaky test. First, an author independently reasoned in detail about each assigned bug to determine categories for the cause and fix. After that, another author double-checks each bug-report/commit for any incorrect classifications. Finally, we discuss together to determine the distinct categories and merge all the results. We describe these categories next.

2.4 Causes of Flaky Tests

From our manual inspection, we create eight categories for the causes of the flaky tests in our study. Table 3 shows the breakdowns.

[1]We use the dependent "packages" as reported by the GitHub API, which are projects that can compile into libraries to be used by others. We use packages because they are more likely to be actively maintained by developers and have reasonable test suites.

Table 3: Causes of Flakiness

CauseCategory	# of Bug Reports/Commits
Algorithmic Non-determinism	45
Floating-point Computations	5
Incorrect/Flaky API Usage	4
Unsynced Seeds	2
Concurrency	2
Hardware	1
Other	12
Unknown	4
Total	75

2.4.1 *Algorithmic Non-determinism.*

We classify a bug report/commit in this category when the cause of flakiness is due to inherent non-determinism in the algorithm being used in the test. Probabilistic programming systems and machine learning frameworks provide functionality for computations that are inherently non-deterministic. For instance, Bayesian inference algorithms like Markov Chain Monte Carlo (MCMC) [44] compute the posterior distribution of a given statistical model by conditioning on observed data. MCMC guarantees convergence to the target posterior distribution in the limit. When designing a test, the developer typically chooses a simple model and a small dataset, which ideally converges very fast. The developer then adds an assertion checking whether the inferred parameter (mean) is close to the expected result. However, there is always a non-zero probability that the algorithm may not converge even with the same number of iterations. Hence, developers need to choose a suitable assertion that accounts for this randomness but does not let any bugs slip through. This behavior is also common for other systems using deep learning algorithms and natural language processing. Given all the parameters a developer has to consider when designing such tests, it is not surprising that the majority of flaky tests in this domain are due to this reason. There are five bug reports/commits related to the randomness of input data, three due to non-determinism in model sampling, and 37 due to non-determinism of algorithms during the training process. Out of those 37 bug reports/commits related to non-determinism of training algorithms, 14 of them involve NLP training algorithms [6], 12 involve deep learning algorithms [61], six involve deep reinforcement learning algorithms [23], and five involve weak supervision training algorithms [72].

Example. The *rlworkgroup/garage* project provides a toolkit for developing and evaluating Reinforcement Learning (RL) algorithms. It also includes implementations of some RL algorithms. Listing 1a shows an example of a test (simplified) for the DQN (Deep Q-Network) model [43], which is used for learning to control agents using high-dimensional inputs (like images). First, the test chooses model training parameters (Lines 21-25). Second, it initializes a game environment (Line 26), which the model should learn to play and chooses an exploration strategy (Line 27). Third, it initializes the DQN algorithm with several parameters, including the exploration strategy and number of training steps (Lines 30-31). Fourth, it trains the model using specified training parameters and returns the average score (`last_avg_ret`) obtained by the model (Lines 32-35). Finally, the assertion checks whether the average score is greater than 20 (Line 58). The algorithm is, however, non-deterministic in nature – at any step of the game (when the method `get_action` in Listing 1b is called), the algorithm chooses a random action (shown in Listing 1b - Line 64-65) with some probability. Hence, the score obtained by the algorithm varies across runs (even for same parameters), and it sometimes is less than the asserted 20, causing the test to occasionally fail, i.e., it is a flaky test. In Section 2.5.1, we discuss how developers fix such flaky tests.

```
   # test_dqn.py
18 def test_dqn_cartpole(self):
19     """Test DQN with CartPole environment."""
20     with LocalRunner(self.sess) as runner:
21         n_epochs = 10
22         n_epoch_cycles = 10
23         sampler_batch_size = 500
24         num_timesteps = n_epochs * n_epoch_cycles *
25         sampler_batch_size
26         env = TfEnv(gym.make('CartPole-v0'))
27         epilson_greedy_strategy = EpsilonGreedyStrategy(
28             env_spec=env.spec,
29             total_timesteps=num_timesteps, ...)
30         algo = DQN(env_spec=env.spec,
           ↪   exploration_strategy=epilson_greedy_strategy, ...)
31         runner.setup(algo, env)
32         last_avg_ret = runner.train(
33             n_epochs=n_epochs,
34             n_epoch_cycles=n_epoch_cycles,
35             batch_size=sampler_batch_size)
36         assert last_avg_ret > 20
```

(a) Flaky test in `test_dqn.py`

```
   # epsilon_greedy_strategy.py
61 def get_action(self, t, observation, policy, **kwargs):
62     opt_action = policy.get_action(observation)
63     self._decay()
64     if np.random.random() < self._epsilon:
65         opt_action = self._action_space.sample()
66     return opt_action, dict()
```

(b) Source of randomness in `get_action`

Listing 1: A Flaky Test caused due to Algorithmic Non-Determinism

2.4.2 Floating-point Computations. We classify a bug report/commit in this category if the test is flaky due to incorrect handling of floating-point computations such as not handling special values (such as NaN) or having rounding issues. However, the floating point computations only result in these erroneous conditions for certain sequences of values, which can differ across executions due to randomness. Hence, tests sporadically fail due to incorrect handling of these special values.

2.4.3 Incorrect/Flaky API Usage. We classify a bug report/commit in this category if the related code uses an API incorrectly, or if the API is known to be flaky but is not handled appropriately in source/test code. As an example, in the *rlworkgroup/garage* project, there are two tests that are testing some functionality of the project that involves training a TensorFlow computation graph (which persists across tests). However, neither of the tests reset the graph after use. Hence, when run back to back, TensorFlow crashes due to duplicate variables in the graph, so the test that runs after fails.

2.4.4 Unsynced Seeds. We classify a bug report/commit in this category if the test is setting seeds for random number generators inconsistently. Many of the libraries that we study use multiple modules that rely on a notion of randomness. For example, *tensorflow/tensor2tensor* uses `tensorflow`, `numpy`, and python's `random` module. Using different seeds (or not setting seeds) across these systems can trigger different computations, therefore leading to different test results than expected.

2.4.5 Concurrency. We classify a bug report/commit in this category when the flakiness is caused due to interaction among multiple threads. Different runs of the same test leads to different thread inter-leavings, which in turn lead to different test results. As an example, in the *tensorflow/tensor2tensor* project, there was a control dependency on an input for a computation in the RNN. However, when the tests run in parallel, the order of computations is uncertain, which causes some tests to fail when the input has not yet been computed.

2.4.6 Hardware. We classify a bug report/commit in this category if the flakiness is from running on some specialized hardware. We find *one commit* that disables a test on TPU (Tensor Processing Unit), which is a specialized accelerator for deep learning. TPUs are efficient in doing parallel computations like matrix multiplications on a very large scale. However, these computations can be relatively non-deterministic due to different orderings of floating point computations. The randomness in the order of collecting partial results from several threads can sometimes amplify the errors, leading to different results.

2.4.7 Other. We group several bug reports/commits that are likely flaky due to causes related to flaky tests in general software into this category. These flaky tests include test failures due to flakiness in the pylint checker, file system, network delays, and iterating over an unordered key set in a python dictionary. There are 12 bug reports/commits in this category.

2.4.8 Unknown. We classify a bug report/commit into the Unknown category when we do not find enough information to properly categorize the cause for flakiness. These bug reports/issues do not provide enough description of the cause of flakiness, neither in the commit message nor in the developer conversations.

2.5 Fixes for Flaky Tests

Table 4 shows the common categories of fixes we observe for flaky tests in our evaluation projects. Note that the total number differs by four from the total in Table 3 (79 versus 75), because four of the bug reports/commits includes two fixes each that we classify into two separate categories.

2.5.1 Adjust Thresholds. In many cases, the developers reason that the threshold for the assertion in a test is too tight, causing the test to fail intermittently. In such scenarios, the developers prefer loosening the threshold by some amount to reduce the test flakiness.

One example test is `test_mnist_tutorial_tf` in the *tensorflow/cleverhans* project. The test runs various ML algorithms on the MNIST dataset, perturbed by adversarial attacks. It then checks various accuracy values generated in the report. Originally, the test's assertion compares the computed value with 0.06.

However, the developers observed that, on adversarial examples, the accuracy occasionally exceeds 0.06. To fix [12] this flaky test, the developers used a higher threshold of 0.07.

```
self.assertTrue(report.clean_train_adv_eval < 0.06 )
self.assertTrue(report.clean_train_adv_eval < 0.07 )
```

2.5.2 Fix Test and Fix Code. We classify 13 bug reports/commits where developers fix a bug in the *test code* and 9 bug reports/commits where developers fix a bug in *source code*, to mitigate flakiness completely in the test instead of reducing the chances of flaky failure (e.g., by adjusting thresholds in the assertion).

These two categories each cast a fairly wide net, as there is no *one* kind of fix in this category that is common across all the projects. For instance, Issue #727 of the *allenai/allennlp* project [3] exposes the issue that their tests were creating modules during execution, but python's `importlib` could not find them. To fix this, developers added `importlib.invalidate_caches()` (as recommended by the `importlib` documentation) to the *test code* so that the newly created modules can be discovered. In another case, the loss computation in the *cornellius-gp/gpytorch* project was buggy since it was not handling `NaN`s gracefully, leading to intermittent failures during execution. In Pull Request #373 [31], the developers added a check in the *source code* to account for `NaN`s during execution and provide warning messages to the user instead of crashing.

2.5.3 Fix Seed. Fixing a seed for a non-deterministic algorithm makes the computations deterministic and easier for the developers to write assertions that compare against a fixed value.

An example from Pull Request #1399 in the project *PySyft* [55] illustrates this strategy. Originally, the developers had a test called `test_federated_learning` that creates a dataset and a neural network model, and finally performs a few iterations of back-propagation. Then, the test checks whether the loss in the last iteration is less than the loss in the first iteration.

Table 4: Fixes for Flaky Tests

Fix Category	# of Bug Reports/Commits
Adjust Thresholds	15
Fix Test	13
Fix Seed	12
Remove Test	10
Mark Flaky Test	10
Fix Code	9
Adjust Test Params	8
Upgrade Dependencies	1
Other	1
Total	79

The total number differs by four from the total in Table 3 (79 versus 75), because four of the bug reports/commits include two fixes each that we classify into two separate categories.

```
for iter in range(6):
for iter in range(2):
 for data, target in datasets:
   model.send(data.owners[0])
   model.zero_grad()
   pred = model(data)
   loss = ((pred - target)**2).sum()
   loss.backward()
   ...
    if(iter == 0):
      first_loss_loss = loss.get().data[0]
assert loss.get().data[0] < first_loss
   if(iter == 1):
      final_loss = loss.get().data[0]
assert final_loss == 0.18085284531116486
```

This test was problematic due to non-deterministic floating point computations of gradients. As a result, there is always a possibility that the loss may not reduce in a small number of steps (6 in this case). To resolve the flakiness, the developers instead fixed the seed earlier in the code using `torch.manual_seed(42)`, changed the test to perform just *two* iterations of back-propagation, and finally changed the assertion to check the *exact* value of the loss.

This test, while seemingly fragile, is now deterministic due to the fixed seed. In general, fixing seeds can make the test deterministic. However, the test can potentially fail in the future if the sequence of underlying computations (which deal with randomness) changes due to modifications of the code under test or if the implementation of the random number generator changes. We provide a more detailed qualitative discussion on the trade-offs of setting seeds in the test code through a few case studies in Section 5.

2.5.4 Remove Test. Developers may remove or disable a flaky test if they cannot fix the flakiness or tune the various parameters to control the results. While such a solution does indeed "fix" a flaky test, it is a rather extreme measure as a developer is deciding that the flakiness in the test is more trouble than the benefit the test provides. However, we still found 10 fixes where the developer removes or disables a flaky test.

2.5.5 Mark Flaky Test. The *Flaky* plugin [22] for pytest allows the developers to mark a test as flaky. The developer can annotate a test function with the `@flaky` decorator, and then pytest by default re-runs the test once in the case of a failure. The developer can also set the maximum number of re-tries (`max_runs`) and minimum number of passes (`min_passes`). pytest re-runs the test until it passes `min_passes` times or runs the test `max_runs` of times. The developers usually prefer this setup when the test fails in some corner cases that are expected to occur rarely but hard to handle specifically.

For example, in the *allenai/allennlp* project, the developers added the `@flaky` decorator to `test_model_can_train_save_and_load` [2], forcing pytest to rerun it in case of future failures.

2.5.6 Adjust Test Params. The accuracy of a machine learning or Bayesian inference algorithm depends on the number of iterations/epochs on which the model is trained. If the number of iterations is not enough, then the results may not be stable and can occasionally cause the assertions in tests, which check for accuracy,

Table 5: Distribution of Fixes for Each Cause of Flakiness

Cause\ Fix	Adjust Thresholds	Fix Test	Fix Seed	Remove Test	Mark Flaky Test	Fix Code	Adjust Test Params	Upgrade Deps	Other	Total
Algorithmic Non-det	14	2	11	5	9	0	8	0	0	49
FP Computations	1	1	0	0	0	3	0	0	0	5
Incorrect/Flaky API Usage	0	2	0	0	0	2	0	0	0	4
Unsynced Seeds	0	0	1	1	0	0	0	0	0	2
Concurrency	0	1	0	0	0	1	0	0	0	2
Hardware	0	0	0	1	0	0	0	0	0	1
Other	0	6	0	1	1	3	0	1	0	12
Unknown	0	1	0	2	0	0	0	0	1	4
Total	15	13	12	10	10	9	8	1	1	79

to fail. Apart from number of iterations/epochs, there are other test parameters that a user can configure, like batch size, data set generation parameters (size and distribution), and number of reruns (for known flaky tests).

For example, in the *allenai/allennlp* project, developers noted that the NumericallyAugmentedQaNetTest, from naqanet_test.py, was flaky and can cause failures when run on GPU. So, in commit 089d744 [1], the developers increased the maximum number of runs and specified the minimum number of passes needed for the test to be marked successful:

```
@flaky (max_runs=3, min_passes=1)
def test_model_can_train_save_and_load(self):
    self.ensure_model_can_train_save_and_load(self.param_file)
```

The max_runs and min_passes parameters are part of the *Flaky* plugin [22].

2.5.7 Upgrade Dependencies. Developers sometimes have to upgrade their existing dependency versions, such as a dependency on a probabilistic programming system or machine learning framework. We observed 1 bug report from the project *azavea/raster-vision*, where developers upgraded the version of TensorFlow on which the project depends.

In Issue #285 in the *azavea/raster-vision* project [57], the developers were observing an intermittent failure that occurred roughly ten percent of the time. The failure was triggered by using the command export_inference_graph from the TensorFlow Object Detection API, which would intermittently give a Bad file descriptor error. The developers resolved this issue through upgrading their dependency on TensorFlow 1.5 to TensorFlow 1.8.

2.5.8 Other. In Issue #167 of the *tensorflow/cleverhand* project [13], a developer reported that some tests were both slow to run and flaky. The developers applied a number of fixes to update the test configurations and updated various test parameters to improve the runtime of the tests. We classified this fix as "Other" as it did not directly fit into any other category.

2.6 Distribution of Fixes for Each Cause

Table 5 presents the distribution of fixes for each cause of flakiness. For the biggest cause of flakiness, Algorithmic Non-determinism, we see that the most common developer fixes are to adjust the accuracy thresholds in assertions, fix seeds, mark the tests as flaky (equivalent to re-running), or even remove the test. This analysis brings out two main observations: (1) Fixing flaky tests in the Algorithmic Non-determinism cause category requires an intimate understanding of the code under test and hence is non-trivial, sometimes even for developers familiar with the code base. (2) We observe that adjusting accuracy thresholds in assertions is the most common fix developers choose overall.

These two observations motivated us to develop FLASH, a novel technique for detecting flaky tests caused by Algorithmic Non-determinism that focuses on tests with assertions that do approximate comparisons (using developer-specified accuracy thresholds) between actual and expected values in tests.

3 FLASH

We propose FLASH, a technique for detecting flaky tests due to non-deterministic behavior of algorithms. At a high-level, FLASH runs tests multiple times but with different random number seeds, which leads to different sequences of random numbers between executions. FLASH then checks for differences in actual values in test assertions and even different test outcomes. The main challenge to this approach is to determine how many times FLASH should run each test before it can make a decision as to whether the test is flaky. We address this challenge by using a *convergence test* (Section 3.2) to dynamically determine how many times to run a test.

3.1 System Overview

Figure 2 shows the overall architecture for FLASH. FLASH takes as input a project and a set of assertion patterns to check for flakiness.

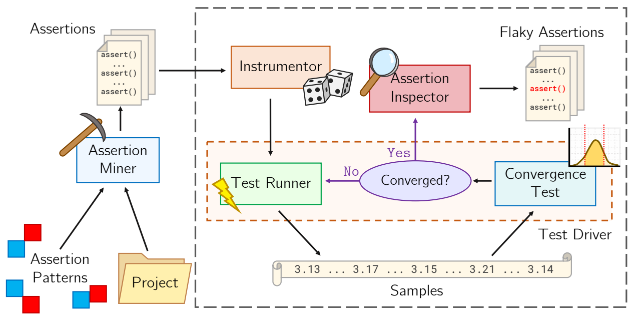

Figure 2: Flash Architecture

FLASH reports the assertions in the project that match the input assertion patterns and that can fail due to sampling random numbers. FLASH consists of four main components: (1) the Assertion Miner collects all the assertions that match the assertion patterns specified by the user, (2) the Test Instrumentor instruments a test assertion to set necessary seeds and log the actual and expected values in the assertion, (3) the Test Driver runs the test several times until the distribution of actual values converge, and (4) the Assertion Inspector compares the actual and expected values and determines the probability that the assertion might fail. Algorithm 1 describes the main algorithm for FLASH and how it uses all four components. While FLASH currently runs on all tests in a project to detect flaky tests (and we run on all for our later evaluation), developers can instead run FLASH for tests only after they fail, or check only newly committed tests, speeding up the flaky test detection over time.

Next, we first discuss the details of the convergence test, and then we discuss each of FLASH's components in detail and how they use the convergence test.

Algorithm 1 FLASH Algorithm

```
**Input**: Project $P$, Assertion patterns $Ap$
**Output**: Set of resulting flaky assertions $FA$
procedure FLASH($P$, $Ap$)
    $FA \leftarrow \emptyset$
    $As \leftarrow AssertionMiner(P, Ap)$
    for $(T, A) \in As$ do
        $T_i \leftarrow TestInstrumentor(T, A)$
        $S \leftarrow TestDriver(T_i)$
        $status, \mathcal{P} \leftarrow AssertionInspector(T, A, S) =$ FLAKY
        if $status =$ FLAKY then
            $FA \leftarrow FA \cup \{(A, S, \mathcal{P})\}$
        end if
    end for
    return $FA$
end procedure
```

3.2 Convergence Test

Given an assertion A in a test function T, we want to compute the probability of failure for the assertion. Computing this probability requires the entire distribution of values that the expression in the assertion can evaluate to. Let us assume we have an assertion of the following form:

$$\text{assert } x < \theta$$

Here, we would like to estimate the distribution for x so that we can compute the probability of x exceeding θ.

We pose this problem in the form of estimating the distribution of an unknown function $\mathcal{F}$, where $\mathcal{F}$ essentially runs T and returns the value of x. One approach for solving this problem is to use a sampling-based approach, wherein we execute $\mathcal{F}$ multiple times to obtain a number of samples, estimate the distribution from the samples, and compute the probability of failure : $P(\mathcal{F} > \theta)$. However, this approach has two main challenges. We need to decide (1) whether *we have seen enough samples* and (2) *how many samples* to collect at minimum.

To tackle the first challenge, researchers have proposed several metrics to measure convergence of a distribution like Gelman-Rubin diagnostic [26], Geweke diagnostic [27], and Raftery-Lewis [56]. In this work, we specifically use the Geweke diagnostic [27] to measure the convergence of a set of samples. Intuitively, the Geweke diagnostic checks whether the mean of the first 10% of the samples is not significantly different from the last 50%. If yes, then we can say that the distribution has converged to the target distribution. To measure the difference between the two parts, the Geweke diagnostic computes the Z-score, which is computed as the difference between the two sample means divided by the standard errors. Equation 1 shows the formula for the Z-score computation for Geweke diagnostic, where a is the early interval of samples, b is the later interval of samples, $\hat{\lambda}$ is the mean of each interval and Var is the variance of each interval of samples.

$$z = \frac{\hat{\lambda}_a - \hat{\lambda}_b}{\sqrt{Var(\lambda_a) + Var(\lambda_b)}} \tag{1}$$

If the Z-score is small, then we can say that the distribution has converged. We choose to use the Geweke diagnostic because it does not involve any assumptions about the independence of the samples and does not require samples from multiple chains like other metrics. To use this metric with our approach, the user can specify the minimum desired threshold. In FLASH, we use a threshold of 1.0 in the algorithm, which intuitively translates to choosing a maximum standard deviation of 1.0 between intervals.

Unfortunately, there is no one way to tackle the second challenge i.e., *how many samples* to collect at minimum. If the minimum sample size is too large, we waste too much time executing $\mathcal{F}$ even when not needed. On the other hand, if the sample size is too small, the conclusions may be questionable. Some studies [58] recommend using a minimum sample size of 30 for statistical significance. We allow the user to specify the minimum sample size, but by default, and for our own evaluation (Section 4), we set it to 30 samples.

Computing probability of failure. Given a set of samples for $\mathcal{F}$, we now need to determine the probability of failure: $P(\mathcal{F} > \theta)$. For this task, we fit an empirical distribution, $\mathcal{D}$, over the samples, and compute the cumulative distribution function (CDF): $CDF_{\mathcal{D}}(\theta) = P(\mathcal{D} < \theta)$. Finally, the probability of failure is given by $1 - P(\mathcal{D} < \theta)$. We can easily transform any other kind of assertion into this form to do this computation as well.

Comparison with hypothesis testing. An alternate approach to using a convergence test is to use statistical hypothesis testing. In this case, we would try to reason about two hypotheses: the null hypothesis: $P(\mathcal{F} > \theta) > k$, and the alternative hypothesis: $P(\mathcal{F} > \theta) \leq k$. A common hypothesis test is the sequential probability ratio test (SPRT) [68], which continuously samples from the given function until it either rejects the null hypothesis or accepts the alternative hypothesis. SPRT is proven to take the optimal number of iterations for a desired level of accuracy. For this approach, we can model $\mathcal{F}$ as a binomial random variable and only record whether it passed or failed after each execution. Each sample of $\mathcal{F}$ is assumed to be independent.

However, there are several practical limitations of using hypothesis testing for our purposes. First, to obtain a desired level of accuracy, one needs to collect a very large number of samples. For instance, for $k = 0.01$, to obtain a Type I error of less than 0.01 and Type II error of less than 0.2, one needs at least 100 samples. For a non-flaky assertion where the values of x are not non-deterministic, hypothesis testing would still require many samples to determine the assertion is not flaky, wasting time. Second, a flaky assertion

Table 6: Assertion Patterns

Source	Assertion	Count
Python	assert expr < | > | <= | >= threshold	62
Unittest	assertTrue	64
Unittest	assertFalse	0
Unittest	assertGreater	168
Unittest	assertGreaterEqual	7
Unittest	assertLess	467
Unittest	assertLessEqual	21
Numpy	assert_almost_equal	252
Numpy	assert_approx_equal	8
Numpy	assert_array_almost_equal	222
Numpy	assert_allclose	278
Numpy	assert_array_less	4
TensorFlow	assertAllClose	1613
Total		3166

may exhibit a large variation in actual observations but may not fail in the first 100 iterations (due to pure chance). The SPRT test would miss detecting this flaky test. The convergence test, however, has a better chance of capturing this behaviour as it monitors the variation in the overall distribution of values. Given these limitations, we choose to use a convergence test to determine the number of times to run a test for our technique FLASH.

3.3 FLASH Components

We now describe the main components of FLASH.

3.3.1 Assertion Miner. We consider only assertion methods that do approximate comparisons between the actual computed value and the expected value rather than checking for exact equality. These assertions are similar to approximation oracles as defined by Nejadgholi and Yang in their prior work [45]. Table 6 shows the assertion patterns that we consider along with what library the assertion comes from and how often each assertion appears within our evaluation projects.

In Algorithm 1, FLASH calls the Assertion Miner to get all the assertions that match the specified assertion patterns in the project (line 3). The Assertion Miner iterates through all the test files in a project and checks each test function to see if it contains one or more assertions of interest. The Assertion Miner creates an *assertion record* for each assertion of interest, which consists of the file name, class name, test function name, and assertion location in the file.

3.3.2 Test Instrumentor. For each assertion record, FLASH uses the Test Instrumentor to instrument the relevant assertion for the later steps that runs the test (line 5 of Algorithm 1). The instrumentation involves logging the actual and expected values during a test run and introducing the logic for controlling the seed for the random number generators during the test run.

The Test Instrumentor sets the seed by introducing a pytest fixture function that performs setup before tests run. The fixture sets a concrete seed for the random number generator in different modules (e.g., NumPy, TensorFlow, PyTorch). We configure the pytest fixture to run at the session level, which means that the fixture runs once before any test runs. If the developer has explicitly set some seed for their tests, the fixture would not override that seed.

3.3.3 Test Driver. FLASH takes the instrumented test and passes it to the Test Driver to run and collect samples for the relevant assertion (line 6 in Algorithm 1). The Test Driver relies on a convergence test to determine how many samples it needs to collect (which is the number of times to run the test).

Algorithm 2 Test Driver Algorithm

```
Input: Instrumented test $T_i$
Output: Set of samples $S$
1:  procedure TestDriver($T_i$)
2:      $batch\_size$ = INITIAL_BATCH_SIZE
3:      $i \leftarrow 0$
4:      $S \leftarrow \emptyset$
5:      while $i <$ MAX_ITERS do
6:          $b \leftarrow 0$
7:          while $b <$ batch_size do
8:              $sample \leftarrow ExecuteTest(T_i)$
9:              $S \leftarrow S \cup \{sample\}$
10:             $b \leftarrow b + 1$
11:         end while
12:         $score \leftarrow ConvergenceScore(samples)$
13:         if $score <$ CONV_THRESHOLD then
14:             break
15:         end if
16:         $i \leftarrow i + batch\_size$
17:         $batch\_size \leftarrow$ BATCH_UPDATE_SIZE
18:     end while
19: return $S$
20: end procedure
```

Algorithm 2 shows the high-level algorithm for the Test Driver. Aside from the input instrumented test T_i, the Test Driver requires the user to set up some other configuration options related to the convergence test (Section 3.2). These options include the initial number of iterations to run the test (`INITIAL_BATCH_SIZE`, default 30), the maximum iterations to run if the values do not converge (`MAX_ITERS`, default 500), the maximum threshold for the convergence test score (`CONV_THRESHOLD`, default 1.0), and the additional number of iterations to run at a time if the samples do not converge (`BATCH_UPDATE_SIZE`, default 10). The Test Driver then returns the set of collected samples.

`TestDriver` runs the test multiple times using the `ExecuteTest` procedure (line 8 in Algorithm 2). The `ExecuteTest` procedure runs the test using pytest in a virtual environment setup for the project. After running the test, `ExecuteTest` returns a *sample*, which consists of the actual value from the assertion, the expected value from the assertion, a log of the output from the test run, and the seed set to the random number generators for that one run. The actual and expected value can be either a scalar, array, or tensor. Furthermore, `ExecuteTest` can obtain multiple actual and expected values for a single assertion, if the assertion is called multiple times within a single test run, e.g., the assertion is in a loop. In such cases, the actual and expected values in the sample are arrays containing the values from each time the assertion is executed in the test run.

If the convergence test finds the samples to have converged after the initial batch of runs (lines 7- 12), then the Test Driver continues to collect more samples by iterating in batches of `BATCH_UPDATE_SIZE` iterations (line 17). the Test Driver returns the set of collected samples after convergence or after reaching `MAX_ITERS` iterations.

3.3.4 Assertion Inspector. FLASH uses the Assertion Inspector to determine if an assertion is flaky or not (line 7 in Algorithm 1). The Assertion Inspector first checks if any of the collected samples contain a failure (the assertion was passing in the production environment before running with FLASH, so a failure should indicate flakiness). If not, the Assertion Inspector takes the samples and the assertion, and it computes the probability of the assertion failing (Section 3.2). If the probability of failure is above the user-specified threshold, the assertion is also considered flaky. FLASH records all the assertions the Assertion Inspector reports as flaky, along with collected samples and probability of failure $\mathcal{P}$ for such assertions.

The Assertion Inspector also reports the bounds of the distribution of actual values sampled. The user can use the reports from the Assertion Inspector to make a decision on whether to take action in fixing the assertion or not, along with information that can help with fixing the assertion.

4 EVALUATION

We evaluate FLASH on the same projects where we find flaky tests from historical bug reports/commits from Section 2. Specifically, we answer the following research questions:

RQ1 How many new flaky tests does FLASH detect?
RQ2 How do developers react to flaky tests FLASH detects?
RQ3 How many old, historical flaky tests does FLASH detect?

Experimental Setup. We run all our experiments using a 3 GHz Intel Xeon machine with 12 Cores and 64GB RAM. We implement FLASH entirely using Python. We use the Geweke test implementation provided by the Arviz library [35] in Python.

4.1 RQ1: New Flaky Tests Detected by FLASH

For the 20 projects from our study with flaky test bug reports/commits, we run FLASH on each project's latest commit, collected on Dec 20, 2019. For each project, we create a virtual environment using the Anaconda package management system [14] for Python to run the tests, installing the latest version of each project's specified dependencies. We use a threshold of 1.0 for convergence test using Geweke diagnostic (`CONV_THRESHOLD`). `INITIAL_BATCH_SIZE` is set to 30 and `BATCH_UPDATE_SIZE` is set to 10. We are not able to run FLASH for three projects. Of these three projects, we do not find assertions matching the assertion patterns we support in FLASH for *pytorch/captum* and *azavea/raster-vision*. For the remaining project, *rlworkgroup/garage*, the tests time out when run with FLASH due to the limited GPU support that we have on our machine, which causes those tests to run very slowly.

Table 7 shows our results for running FLASH on the remaining projects. The first column **Projects** shows the name of the project. The second column **Total** shows the total number of assertions FLASH collects samples from for each project. The third column **Pass** shows the number of assertions that always pass across all runs. The fourth column **Fail** shows the number of assertions that fail at least once when FLASH runs them for several iterations. The fifth column **Skip** shows the number of assertions that were skipped due to several reasons (e.g., needing specialized hardware like a TPU). The sixth column **Conv-NZ** shows the assertions that have a convergence score of more than 0 in the first batch. The seventh column **Conv-Z** shows the number of assertions that have a 0 convergence score in the first batch. The eighth column **Avg-Runs** shows the average number of iterations that FLASH runs for the assertions. The last column **Max-Runs** shows the maximum number of iterations that FLASH runs for the assertions.

From the **Conv-Z** column, we see that most of the assertions always have the same actual value, which is why the convergence score is 0 in these cases. As such, the average number of runs is usually quite close to the initial batch size we set (30). The average number of iterations per assertion over all projects is 45.32 (and the average of the maximum number of iterations per assertion is 233.53, suggesting the convergence test is effective at reducing the number of iterations FLASH should run per assertion.

There are 252 cases where an assertion never fails but has a non-zero convergence score. The values of the expressions in these assertions fluctuate, but we do not observe any failure. We rerun FLASH for those assertions with a stricter threshold of 0.5 for the convergence test and observe whether they reveal any more failures. On average, while this configuration option makes FLASH take more iterations for convergence, we still do not observe any failures.

Table 8 shows the details of the failing tests that FLASH detects (we do not find a case where FLASH finds multiple assertions in the same test to fail). The second column lists the total failing tests for each project. We manually inspect each failure and reason about whether the failure indicates a potential flaky test in the project under test. The third column shows the number of tests we determine to be a new flaky test. The fourth column shows the number of tests where the developers confirm the flaky test after we report them. The final column shows the number of tests where we determine that the cause of failure is either due to a mis-configuration in our local test environment or it is an already confirmed flaky test. Overall, FLASH detects 32 failing assertions across 11 projects, of which we determine 11 indicate flaky tests.

4.2 RQ2: Fixing Flaky Tests

For the 11 potential flaky tests FLASH detects, we sent 4 Pull Requests (PRs) and 6 Issue Requests to the developers of the projects on GitHub. Developers accepted 3 PRs that fix four flaky tests, while the other is closed. For the closed PR (in *zfit/zfit*), the developers confirmed that the test is flaky but made a different fix than what we proposed. Out of the 6 Issue Requests, the developers have confirmed 5 and fixed one of them, and the last is pending review. We discuss the flaky tests that are confirmed and fixed in more detail.

FLASH finds four flaky tests in *allenai/allennlp*. In one flaky test, the test randomly generates a list of predictions and corresponding labels, then computes the AUC (Area Under Curve) metric and checks whether the result matches the expected result, which is computed using a similar implementation from the *scikit-learn* package. However, the metric fails when there are no positive labels in the randomly generated list of labels. We could easily reproduce the error using the seed(s) FLASH finds. After discussing with the developers, we conclude that this is an expected behaviour for the AUC implementation. Hence, we fix the test by explicitly setting one label to positive always before calling the metric. In the three other flaky tests, the tests run various training algorithms and compute their F1 scores (harmonic mean of precision and recall). Then, the tests check whether the computed F1 score is greater than 0. FLASH finds at least one PyTorch seed for each test that causes the F1 score to be 0, and hence the assertion fails. After discussing this issue with the developers, we conclude that the best fix is to mark the test as flaky. An F1 score of 0 is expected when there are no positive samples in a dataset.

Table 7: Distribution of Passing and Failing Asserts

Projects	Total	Pass	Fail	Skip	Conv-NZ	Conv-Z	Avg-Runs	Max-Runs
allenai/allennlp	428	424	4	0	97	331	43.29	200
cornellius-gp/gpytorch	1180	1179	1	0	126	1054	35.66	200
deepmind/sonnet	185	166	1	18	0	167	30.00	30
geomstats/geomstats	467	464	3	0	0	467	30.00	30
HazyResearch/metal	19	18	1	0	3	16	35.79	80
OpenMined/PySyft	12	11	1	0	8	4	110.83	500
PrincetonUniversity/PsyNeuLink	198	166	0	32	0	166	30.00	30
pytorch/botorch	86	85	1	0	2	84	35.81	500
pytorch/vision	7	7	0	0	3	4	42.86	80
Qiskit/qiskit-aqua	113	75	9	29	7	77	53.33	500
RasaHQ/rasa	19	19	0	0	9	10	85.26	400
snorkel-team/snorkel	58	58	0	0	7	51	42.76	500
tensorflow/cleverhans	79	75	2	2	6	71	31.82	70
tensorflow/gan	108	100	0	8	2	98	31.00	90
tensorflow/magenta	21	21	0	0	0	21	30.00	30
tensorflow/tensor2tensor	157	150	7	0	9	148	38.22	500
zfit/zfit	29	27	2	0	13	16	63.79	230
Sum/Avg	3166	3045	32	89	292	2785	45.32	233.53

Table 8: Failures for Each Project

Project	Failures	Flaky Tests	Confirmed	Other
allenai/allennlp	4	4	4	0
cornellius-gp/gpytorch	1	0	0	1
deepmind/sonnet	1	1	1	0
geomstats/geomstats	3	0	0	3
HazyResearch/metal	1	0	0	1
OpenMined/PySyft	1	1	1	0
pytorch/botorch	1	1	1	0
Qiskit/qiskit-aqua	9	0	0	9
tensorflow/cleverhans	2	2	2	0
tensorflow/tensor2tensor	7	1	0	6
zfit/zfit	2	1	1	1
Total	32	11	10	21

Table 9: Old Flaky Tests Detected by FLASH

Project	Commit	Cause	Failures/Iters	Threshold
allenai/allennlp	5e68d04	FP Computations	2/100	1.0
allenai/allennlp	5dd1997	Algo. Non-det.	28/100	1.0
tensorflow/tensor2tensor	6edbd6b	Algo. Non-det.	2/70	1.0
pytorch/botorch	e2e132d	Algo. Non-det.	2/170	0.5
pytorch/botorch	7ac0273	Algo. Non-det.	6/50	1.0
tensorflow/cleverhans	b2bb73a	Algo. Non-det.	5/40	1.0
tensorflow/cleverhans	4249afc	Algo. Non-det.	19/80	1.0
tensorflow/cleverhans	58505ce	Algo. Non-det.	7/50	1.0
snorkel-team/snorkel	3d8ca08	Algo. Non-det.	1/90	0.5
OpenMined/PySyft	b221776	Algo. Non-det.	1/270	0.5
pytorch/captum	d44db43	Algo. Non-det.	4/40	1.0

FLASH finds one new flaky test in *zfit/zfit*. The test creates four random variables, each of which is assigned a Gaussian distribution. Then, it creates another random variable that is just the sum over those four Gaussian random variables. Finally, the test fetches the dependent random variables from that summed random variable through the call `get_dependents()`, which returns an *ordered set*. The test asserts that the iteration order of the ordered set is different than the iteration order of the regular, unordered set of these dependent random variables. FLASH finds an execution where this test fails, which is expected since there is always a non-zero probability that iterating through the unordered set leads to the same order as the ordered set. We proposed a fix to this test in a PR, where we marked it as flaky using the *@flaky* annotation in pytest. The developers confirmed that the test is flaky (and buggy) but they made a different fix. They reasoned that the test should just be checking that multiple calls to `get_dependents()` return the same ordered set, so they refactored the test to check that the return of `get_dependents()` is the same across subsequent calls.

FLASH finds another flaky test in *pytorch/botorch*. The test creates a probabilistic model, generates data, runs a few steps of gradient computation, and checks whether the gradient is greater than a fixed threshold. However, FLASH finds an execution and the corresponding Torch seed where this checks fails. After reporting this behavior to the developers, they confirm that the test is indeed flaky and adjust the threshold to reduce the chance of failure.

4.3 RQ3: Old Flaky Tests Detected by FLASH

For the same 20 projects, we evaluate how effective FLASH is at detecting the old, already identified flaky tests. We only run FLASH for flaky tests that fail due to the assertions FLASH tracks.

First, for each commit related to a fix for a flaky test from our study, we checkout its immediate parent commit (before the fix) and create a virtual environment to run the test on that commit. Like in Section 4.1, we create this virtual environment using Anaconda, but we ensure that the version of each dependency we install is set to the latest version at the time of the commit, not the current latest, as to better reproduce the environment developers were initially running the tests and encountering flakiness. Once we set up this environment, we run FLASH only on the flaky test in question to see if FLASH can detect this flaky test. We use the same configuration for FLASH as we describe in Section 4.1.

Table 9 shows our results on using FLASH to detect old flaky tests. The table shows the project and old commit SHA where we run FLASH, the cause of the flaky test, the fix for the flaky test by the developers, the number of iterations FLASH runs the test based from using Algorithm 1, the number of times the test fails in those iterations, and the threshold we eventually set for the Geweke test to check for convergence. We are unable to run FLASH on the other old flaky tests, because either we cannot reproduce the exact environment to make any tests pass on the old commit, or the old flaky test in question was not failing due to assertions FLASH supports. For example, there are projects where we could not reproduce the environment due missing old dependencies.

Overall, FLASH requires at most 100 iterations to detect most of these flaky tests (only two require more iterations). For three flaky tests, we have to use a tighter threshold of 0.5 to detect the known flaky test. A tighter threshold is expected in some cases, because it might require longer to converge to the target distribution.

5 DISCUSSION

For many assertions, FLASH obtains the same actual value for every run, despite changing the seeds. We suspect that developers are purposely setting the seed to a specific value tests, overriding FLASH's attempt at changing the seed. Setting the seed guarantees a deterministic execution of the tests, a common fix for flaky tests (Section 2.5). However, setting the seed to a specific value can "lock" the developer into a specific execution for all test runs, and developers can potentially miss bugs if other seeds lead to different values that the code is not handling properly. Furthermore, if the underlying random number generator changes, or if the developer changes their code where only the random number generator's sequence of random numbers changes, the assertions can end up failing, because they are too dependent on the specific sequence of random numbers and are therefore flaky.

To evaluate how flaky existing assertions are when run under different seeds for a project, we disable all explicit seed settings throughout the tests in the project [2]. Then, we run FLASH on this modified code. We experiment on two projects: *PrincetonUniversity/PsyNeuLink* and *geomstats/geomstats*.

We sample several of the assertions that FLASH finds as flaky after running on the modified code. For the assertions we determine to be flaky with our manual inspection, we send PR(s) to the developers for fixing the flakiness. We observe both a positive response and a negative response from the two projects.

Experience with *geomstats/geomstats*. Tests in *geomstats/geomstats* run with different backends, including NumPy, TensorFlow, and PyTorch. We run FLASH on the project using each backend while changing seeds. FLASH finds only one test in *geomstats/geomstats* that fails when run with the TensorFlow backend, but always passes in other backends. The test creates an object that represents an N-dimensional space, samples a random point from the defined space, then checks whether the sample belongs to the defined space using the exposed API. The test uses a predefined tolerance of $1e^{-6}$ for the check. After reporting to the developers, they confirmed that this is likely a precision issue: the TensorFlow backend uses single precision floating point by default, whereas the NumPy backend uses double precision. Hence, the NumPy backend can handle higher tolerance levels than TensorFlow for this test. We send a PR to reduce the tolerance level to $1e^{-4}$, which the developers accepted. This case demonstrates that setting seeds in tests can be problematic and sometimes hide subtle bugs. Hence, developers must be careful about using fixed seeds in their tests and reason judiciously about the entailing risks of fixed seeds.

Experience with *PrincetonUniversity/PsyNeuLink*. FLASH finds several tests in *PrincetonUniversity/PsyNeuLink* to fail when run with different seeds. We send a PR to fix one test to start. The test is testing an integrator mechanism that gives deterministic results modulo noise sampled from a normal distribution from NumPy. The assertion checks that the actual value should be close to the expected value with a very specific tolerance level, which is the actual value from running with the specific seed set by the developers. We also find that developers had to several times update the assertion in the past due to changes in the sequence of random numbers, even though they never change the seed itself. We update the assertion to accept a wider tolerance, representing the distribution of values FLASH reports. The goal is to make the assertion fail less often in the future due to changes in just the sequence of random numbers.

However, the developers were against the change, because they indeed want the test to fail even when the failure is solely due to changes in the sequence of random numbers. They want to examine *all* failures, and if they decide the problem is not in their code, they will update the expected value of the assertion accordingly. As such, the developers seem not bothered by the times the test has a flaky failure unrelated to changes in their code under test.

6 THREATS TO VALIDITY

We study only a subset of projects that use probabilistic programming systems and machine learning frameworks, so our results may not generalize to all projects. To mitigate this threat, we study four popular open-source probabilistic programming systems and machine learning frameworks on GitHub, and we consider all of their dependent projects with more than 10 stars (indicating they are somewhat popular and used). Our study on historical bug reports/commits referencing flaky tests may not include all such bug reports/commits. We use a keyword search similar to prior work on studying flaky tests in project histories. We are still able to identify a substantial number of bug reports/commits referencing flaky tests.

FLASH samples seeds to use for random number generators, so the test executions we run through FLASH is only a subset of all possible test executions given different possible sequences of random numbers. The tests FLASH finds to be flaky are for sure flaky, as we confirm the test can pass when run on the code, and we can reproduce the test failure using the seeds FLASH reports. Furthermore, we report these flaky tests to developers and receive confirmation from them that we have detected flaky tests. There may be more flaky tests that FLASH does not detect due to not running enough. As such, our results on flaky tests FLASH detects is an under-count of how may flaky tests exist in these projects.

[2]Interestingly, developers across different projects like to use very similar seeds, such as 0, 1234, or 42.

7 RELATED WORK

Testing of systems dealing with randomness. Machine learning frameworks like TensorFlow [64] and PyTorch [51] have revolutionized the domain of machine learning. A recent surge of interest in probabilistic programming has also led to the development of numerous probabilistic programming systems both in academic research and industry [10, 11, 28–30, 41, 42, 46, 52, 54, 67, 69]. However, tests written by developers in these domains suffer from the problem of inherent non-determinism and lack concrete oracles. Recent work has proposed techniques to systematically test and debug probabilistic programming systems [18, 19], machine learning frameworks [20, 53], and randomized algorithms [34].

Nejadgholi and Yang [45] studied the distribution and nature of approximate assertions in deep learning libraries. They report *adjusting thresholds* as one class of changes that developers typically do for approximate assertions. We also find fixes in the same category in our study (Section 2.5), and we made similar fixes in Section 4.2. However, in other cases in our study, the fixes are more involved and belong to other categories.

Study of Flaky Tests. Recently, there has been much work on studying flaky tests. Harman and O'Hearn [32] reported the problems with flaky tests at Facebook, and they have even suggested that future testing research should adjust to assuming all tests to be flaky. Luo et al. studied the various causes and fixes for flaky tests in open-source software [40]. They studied flaky tests in traditional software, finding that common causes for flaky tests include async wait, concurrency, and test-order dependencies. In this work, we study flaky tests specifically in the domain of software that depends on probabilistic programming systems and machine learning frameworks. In our study, we find that most causes of flaky tests would fall under the category prior work would consider as "randomness", which did not show up as a prominent cause of flakiness in their evaluation. Zhang et al. conducted a survey of machine learning testing and found that flaky tests arise in metamorphic testing whenever floating point computations are involved [71]. In our study, we find floating-point computations to be a major cause for flakiness, such as tests not handling special values (such as NaN) or having rounding issues.

Detecting Flaky Tests. Concerning flaky test detection, Bell et al. proposed DeFlaker [8], a technique for detecting when test failures after a change are due to flaky tests by comparing the coverage of the failing tests with the changed code. There has also been work on techniques that detect specific types of flaky tests. Lam et al. proposed iDFlakies [38], a framework for detecting order-dependent flaky tests, tests that fail when run in different orders, by running tests in different orders. Gambi et al. also detect order-dependent flaky tests using PRADET [24], a technique that tracks data dependencies between tests. Shi et al. proposed NonDex [62], a technique for detecting flaky tests that assume deterministic iteration order over unordered collections by randomly shuffling unordered collections and checking if tests fail. Our tool FLASH also focuses on a specific type of flaky tests that rely on random number generators used through probabilistic programming system and machine learning frameworks. FLASH accomplishes this task through running tests multiple times under different seeds, while using statistical tests to determine the number of runs.

There has also been prior work on analyzing the impact of flakiness. Cordy et al. proposed FlakiMe [15], a laboratory-controlled test flakiness impact assessment and experimentation platform, that supports the seeding of a (controllable) degree of flakiness into the behavior of a given test suite. Our tool FLASH also supports setting seeds for non-deterministic flaky tests and checking the posterior distribution of the testing objects.

Debugging and Fixing Flaky Tests. Recent work has started to focus on how to debug and ultimately fix flaky tests. Lam et al. developed a framework at Microsoft to instrument flaky test executions and collect logs of the traces for both passing and failing executions, which they then find differences between so they can determine the root cause of flakiness [36]. Lam et al. followed up this work with an additional study on how developers at Microsoft attempt to fix flaky tests [37]. Based on their study, they proposed a technique for handling the common case of async waits by modifying wait times in the async waits to reduce flakiness [37]. Shi et al. studied how to fix another type of flaky tests, order-dependent flaky tests, finding that order-dependent flaky tests can be fixed using code taken from other tests in the test suite [63].

Fuzz Testing. Our approach for FLASH is also similar to fuzz testing. Fuzz testing involves generating random data as inputs to software to find bugs in the software. Fuzz testing has been used to find security vulnerabilities [4], performance hot-spots [39], bugs in probabilistic programming systems [18], etc. Many recent fuzz testing techniques are guided to increase coverage of code as to find more bugs [48–50]. While fuzz testing techniques generate random values that are explicitly passed as inputs to the code under test, FLASH generates random values for the seeds to random number generators that code depends on, exploring how different seed values can lead to different test outcomes when existing tests are run on the same version of code.

8 CONCLUSION

Randomness is an important trait of many probabilistic programming and machine learning applications. At the same time, software developers who write and maintain these applications often do not have adequate intuition and tools for testing these applications, resulting in flaky tests and brittle software. In this paper, we conduct the first study of flaky tests in projects that use probabilistic programming systems and machine learning frameworks. Our investigation of 75 bug reports/commits that reference flaky tests in 20 projects identified Algorithmic Non-determinism to be a major cause of flaky tests. We also observe that developers commonly fix such flaky tests by adjusting assertion thresholds. Inspired by the results of our study, we propose FLASH, a technique for systematically running tests with different random number generator seeds to detect flaky tests. FLASH detects 11 new flaky tests that we report to developers, with 10 already confirmed and 6 fixed. We believe that a new generation of software testing tools (like FLASH) based on the foundations of the theory of probability and statistics is necessary to improve the reliability of emerging applications.

ACKNOWLEDGMENTS

This work was partially funded by NSF grants CNS-1646305, CCF-1703637, OAC-1839010, and CCF-1846354.

REFERENCES

[1] AllenNLP Commit 089d744 2019. https://github.com/allenai/allennlp/pull/2778/commits/089d744.
[2] AllenNLP Commit 53bba3d 2018. https://github.com/allenai/allennlp/commit/53bba3d.
[3] AllenNLP Issue 727 2018. https://github.com/allenai/allennlp/pull/727.
[4] American Fuzzy Loop 2014. http://lcamtuf.coredump.cx/afl.
[5] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The oracle problem in software testing: A survey. *IEEE transactions on software engineering* (2015).
[6] M. Bates. 1995. Models of natural language understanding. *Proceedings of the National Academy of Sciences of the United States of America* (1995).
[7] Matthew James Beal. 2003. *Variational algorithms for approximate Bayesian inference.*
[8] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. DeFlaker: Automatically detecting flaky tests. In *ICSE.*
[9] Eli Bingham, Jonathan P Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D Goodman. 2019. Pyro: Deep universal probabilistic programming. *The Journal of Machine Learning Research* (2019).
[10] Bob Carpenter, Andrew Gelman, Matt Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Michael A Brubaker, Jiqiang Guo, Peter Li, Allen Riddell, et al. 2016. Stan: A probabilistic programming language. *JSTATSOFT* (2016).
[11] Guillaume Claret, Sriram K. Rajamani, Aditya V. Nori, Andrew D. Gordon, and Johannes Borgström. 2013. Bayesian Inference Using Data Flow Analysis. In *ESEC/FSE.*
[12] Cleverhans Commit 58505ce 2017. https://github.com/tensorflow/cleverhans/pull/149/commits/58505ce.
[13] Cleverhans Issue 167 2017. https://github.com/tensorflow/cleverhans/issues/167.
[14] Conda package management system 2017. https://docs.conda.io.
[15] Maxime Cordy, Renaud Rwemalika, Mike Papadakis, and Mark Harman. 2019. FlakiMe: Laboratory-Controlled Test Flakiness Impact Assessment. A Case Study on Mutation Testing and Program Repair. arXiv:1912.03197 [cs.SE]
[16] Marco Cusumano-Towner, Benjamin Bichsel, Timon Gehr, Vikash K. Mansinghka, and Martin Vechev. 2018. Incremental Inference for Probabilistic Programs. In *PLDI.*
[17] Joshua V Dillon, Ian Langmore, Dustin Tran, Eugene Brevdo, Srinivas Vasudevan, Dave Moore, Brian Patton, Alex Alemi, Matt Hoffman, and Rif A Saurous. 2017. Tensorflow distributions. *arXiv preprint arXiv:1711.10604* (2017).
[18] Saikat Dutta, Owolabi Legunsen, Zixin Huang, and Sasa Misailovic. 2018. Testing probabilistic programming systems. In *ESEC/FSE.*
[19] Saikat Dutta, Wenxian Zhang, Zixin Huang, and Sasa Misailovic. 2019. Storm: program reduction for testing and debugging probabilistic programming systems. In *ESEC/FSE.*
[20] Anurag Dwarakanath, Manish Ahuja, Samarth Sikand, Raghotham M Rao, RP Jagadeesh Chandra Bose, Neville Dubash, and Sanjay Podder. 2018. Identifying implementation bugs in machine learning based image classifiers using metamorphic testing. In *ISSTA.*
[21] Eric Jang. Why Randomness is Important for Deep Learning 2016. https://blog.evjang.com/2016/07/randomness-deep-learning.html.
[22] Flaky test plugin 2019. https://github.com/box/flaky.
[23] Vincent Francois-Lavet, Peter Henderson, Riashat Islam, Marc G. Bellemare, and Joelle Pineau. 2018. An Introduction to Deep Reinforcement Learning. arXiv:1811.12560 [cs.LG]
[24] Alessio Gambi, Jonathan Bell, and Andreas Zeller. 2018. Practical Test Dependency Detection. In *ICST.*
[25] Timon Gehr, Sasa Misailovic, and Martin Vechev. 2016. PSI: Exact Symbolic Inference for Probabilistic Programs. In *CAV.*
[26] Andrew Gelman, Hal S Stern, John B Carlin, David B Dunson, Aki Vehtari, and Donald B Rubin. 2013. *Bayesian data analysis.*
[27] John Geweke et al. 1991. *Evaluating the accuracy of sampling-based approaches to the calculation of posterior moments.* Federal Reserve Bank of Minneapolis, Research Department Minneapolis, MN.
[28] Wally R Gilks, Andrew Thomas, and David J Spiegelhalter. 1994. A language and program for complex Bayesian modelling. *The Statistician* (1994).
[29] Noah Goodman, Vikash Mansinghka, Daniel M Roy, Keith Bonawitz, and Joshua B Tenenbaum. 2012. Church: a language for generative models. *arXiv preprint arXiv:1206.3255* (2012).
[30] Noah D Goodman and Andreas Stuhlmüller. 2014. The design and implementation of probabilistic programming languages. http://dippl.org.
[31] GPytorch Pull Request 373 2018. https://github.com/cornellius-gp/gpytorch/pull/373.
[32] Mark Harman and Peter O'Hearn. 2018. From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis. In *SCAM.*
[33] Jason Brownlee. Embrace Randomness in Machine Learning 2019. https://machinelearningmastery.com/randomness-in-machine-learning/.
[34] Keyur Joshi, Vimuth Fernando, and Sasa Misailovic. 2019. Statistical algorithmic profiling for randomized approximate programs. In *ICSE.*
[35] Ravin Kumar, Colin Carroll, Ari Hartikainen, and Osvaldo A. Martin. 2019. ArviZ a unified library for exploratory analysis of Bayesian models in Python. *The Journal of Open Source Software* (2019).
[36] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. 2019. Root Causing Flaky Tests in a Large-Scale Industrial Setting. In *ISSTA.*
[37] Wing Lam, Kıvanç Muşlu, Hitesh Sajnani, and Suresh Thummalapenta. 2020. A Study on the Lifecycle of Flaky Tests. In *ICSE.*
[38] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. 2019. iDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests. In *ICST.*
[39] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. PerfFuzz: Automatically Generating Pathological Inputs. In *ISSTA.*
[40] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *FSE.*
[41] Vikash Mansinghka, Daniel Selsam, and Yura Perov. 2014. Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv preprint 1404.0099* (2014).
[42] T. Minka, J.M. Winn, J.P. Guiver, S. Webster, Y. Zaykov, B. Yangel, A. Spengler, and J. Bronskill. 2013. Infer.NET 2.5. Microsoft Research Cambridge. http://research.microsoft.com/infernet.
[43] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).
[44] Radford M Neal et al. 2011. MCMC using Hamiltonian dynamics. *Handbook of markov chain monte carlo* (2011).
[45] Mahdi Nejadgholi and Jinqiu Yang. 2019. A Study of Oracle Approximations in Testing Deep Learning Libraries. In *ASE.*
[46] Aditya V Nori, Chung-Kil Hur, Sriram K Rajamani, and Selva Samuel. 2014. R2: An efficient MCMC sampler for probabilistic programs. In *AAAI.*
[47] Akira K Onoma, Wei-Tek Tsai, Mustafa Poonawala, and Hiroshi Suganuma. 1998. Regression testing in an industrial environment. *Commun. ACM* (1998).
[48] Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019. JQF: Coverage-Guided Property-Based Testing in Java. In *ISSTA DEMO.*
[49] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic Fuzzing with Zest. In *ISSTA.*
[50] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. 2019. FuzzFactory: Domain-Specific Fuzzing with Waypoints. *Proc. ACM Program. Lang.* OOPSLA (2019).
[51] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. PyTorch: An imperative style, high-performance deep learning library. In *NeurIPS.*
[52] Avi Pfeffer. 2001. IBAL: a probabilistic rational programming language. In *IJCAI.*
[53] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. 2019. CRADLE: cross-backend validation to detect and localize bugs in deep learning libraries. In *ICSE.*
[54] PyroWebPage 2018. Pyro. http://pyro.ai.
[55] PySyft Issue 1399 2018. https://github.com/OpenMined/PySyft/pull/1399.
[56] Adrian E Raftery and Steven M Lewis. 1995. The number of iterations, convergence diagnostics and generic Metropolis algorithms. *Practical Markov Chain Monte Carlo* (1995).
[57] Raster Vision Issue 285 2018. https://github.com/azavea/raster-vision/issues/285.
[58] John A Rice. 2006. *Mathematical statistics and data analysis.*
[59] John Salvatier, Thomas V Wiecki, and Christopher Fonnesbeck. 2016. Probabilistic programming in Python using PyMC3. *PeerJ Computer Science* (2016).
[60] Simone Scardapane and Dianhui Wang. 2017. Randomness in neural networks: an overview. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* (2017).
[61] Jurgen Schmidhuber. 2015. Deep learning in neural networks: An overview. *Neural Networks* (2015).
[62] August Shi, Alex Gyori, Owolabi Legunsen, and Darko Marinov. 2016. Detecting Assumptions on Deterministic Implementations of Non-deterministic Specifications. In *ICST.*
[63] August Shi, Wing Lam, Reed Oei, Tao Xie, and Darko Marinov. 2019. iFixFlakies: A framework for automatically fixing order-dependent flaky tests. In *ESEC/FSE.*
[64] TensorFlowWebPage 2018. TensorFlow. https://www.tensorflow.org.
[65] Swapna Thorve, Chandani Sreshtha, and Na Meng. 2018. An Empirical Study of Flaky Tests in Android Apps. In *ICSME.*
[66] Dustin Tran, Matthew D. Hoffman, Rif A. Saurous, Eugene Brevdo, Kevin Murphy, and David M. Blei. 2017. Deep probabilistic programming. In *ICLR.*
[67] Dustin Tran, Alp Kucukelbir, Adji B. Dieng, Maja Rudolph, Dawen Liang, and David M. Blei. 2016. Edward: A library for probabilistic modeling, inference, and criticism. *arXiv* (2016).
[68] Abraham Wald. 1945. Sequential tests of statistical hypotheses. *The annals of mathematical statistics* (1945).

[69] Frank Wood, Jan Willem van de Meent, and Vikash Mansinghka. 2014. A new approach to probabilistic programming inference. In *AISTATS*.
[70] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification, and Reliability* (2012).
[71] Jie M. Zhang, Mark Harman, Lei Ma, and Yang Liu. 2019. Machine Learning Testing: Survey, Landscapes and Horizons. arXiv:1906.10742 [cs.LG]
[72] Zhi-Hua Zhou. 2017. A Brief Introduction to Weakly Supervised Learning. *National Science Review* (2017).