FASER: Balancing Effectiveness and Flakiness of Non-Deterministic Machine Learning Tests

Chunqiu Steven Xia
University of Illinois Urbana-Champaign
chunqiu2@illinois.edu

Saikat Dutta
University of Illinois Urbana-Champaign
saikatd2@illinois.edu

Sasa Misailovic
University of Illinois Urbana-Champaign
misailo@illinois.edu

Darko Marinov
University of Illinois Urbana-Champaign
marinov@illinois.edu

Lingming Zhang
University of Illinois Urbana-Champaign
lingming@illinois.edu

***Abstract*— Testing Machine Learning (ML) projects is challenging due to inherent *non-determinism* of various ML algorithms and the lack of reliable ways to compute reference results. Developers typically rely on their intuition when writing tests to check whether ML algorithms produce accurate results. However, this approach leads to conservative choices in selecting *assertion bounds* for comparing actual and expected results in test assertions. Because developers want to avoid false positive failures in tests, they often set the bounds to be too loose, potentially leading to missing critical bugs.**

We present FASER – the first systematic approach for balancing the trade-off between the fault-detection effectiveness and flakiness of non-deterministic tests by computing optimal *assertion bounds*. FASER frames this trade-off as an optimization problem between these competing objectives by varying the assertion bound. FASER leverages 1) statistical methods to estimate the flakiness rate, and 2) mutation testing to estimate the fault-detection effectiveness. We evaluate FASER on 87 non-deterministic tests collected from 22 popular ML projects. FASER finds that 23 out of 87 studied tests have conservative bounds and proposes tighter assertion bounds that maximizes the fault-detection effectiveness of the tests while limiting flakiness. We have sent 19 pull requests to developers, each fixing one test, out of which 14 pull requests have already been accepted.

I. Introduction

Machine Learning (ML) and Artificial Intelligence (AI) are revolutionizing critical fields like autonomous driving [1] and healthcare [2]. A key challenge in such fields is to avoid *inaccurate computations* that may lead to faulty predictions and consequently pose a risk to society [3], [4]. ML libraries form an integral component of the ML infrastructure that developers use to build ML applications. Subtle accuracy bugs (i.e., non-crashing bugs that may produce incorrect results) in such libraries can remain unnoticed but lead to wrong results in user-facing applications. Hence, developers must take appropriate measures to eliminate such bugs.

ML libraries primarily implement various algorithms such as Deep Learning [5], Probabilistic Programming [6], and Reinforcement Learning [7]. A common trait of such ML algorithms is that they are *non-deterministic* – separate executions can produce different results. As such, it is difficult to obtain reference solutions to compare against the algorithms' results. Hence, developers often make conservative estimates when selecting thresholds comparing the actual and expected results (typically measured by metrics such as model accuracy). This choice diminishes the fault-detection effectiveness of tests checking for accuracy of such ML algorithms.

```
1 def test_MLAlgorithm():
2     train_data, val_data = generate_data()
3     model = MLAlgorithm(train_data)
4     model.train()
5     val_error = model.evaluate(val_data)
6     assert val_error < exp_max_error
```

Listing 1: A Common Pattern for Tests in ML Projects

Listing 1 shows a *common pattern* for many tests used for checking the accuracy of ML algorithms. The test (i) generates random training and validation data on Line 2, (ii) trains a model using an ML algorithm on the training data on Lines 3-4, (iii) evaluates the trained model on validation data and computes a metric (validation error) on Line 5, and (iv) asserts that the metric is below an acceptable constant value, `exp_max_error`, on Line 6. Such test assertions that compare the result against a pre-determined fixed threshold are known as *approximate assertions* [8], [9], [10], [11] while the threshold itself (`exp_max_error`) is known as the *assertion bound*.

Recent studies [10], [12] have found a significant number of non-deterministic tests in ML projects, which account for a major portion of the total test-suite running time (>31%) [9]. However, improving such tests requires careful analysis and deep understanding of the code/algorithm under test. On one hand, if the developers select a bound that is *too tight* (or *liberal*) it may make the test *flaky* – pass and fail non-deterministically – due to the randomness of the underlying ML algorithm. Test flakiness [13] hurts developer productivity and reduces the reliability of test results. On the other hand, if the bound is *too loose* (or *conservative*), the test may miss detecting accuracy bugs, thereby reducing its effectiveness. Developers typically set these bounds based on their intuition and are inclined to choose conservative bounds to avoid flakiness, but can compromise on its fault-detection effectiveness.

Due to their importance in avoiding accuracy bugs, improving the fault-detection effectiveness of such tests is very important. Hence, the key question becomes: *how can we systematically determine assertion bounds that maximize the fault-detection effectiveness of the test without making it unacceptably flaky?*

Our Work. To address this question, we propose *FASER* – the first approach that systematically balances the trade-off between the fault-detection effectiveness and flakiness of non-deterministic regression tests for ML algorithms with the goal of determining optimal assertion bounds. FASER frames this trade-off as an *optimization problem* of two competing objectives of 1) *maximizing* fault-detection effectiveness and 2) *minimizing* flakiness, by varying the assertion bound. Since it is practically not possible to exactly solve these two objectives, the main challenge is to obtain a reliable estimate of the objectives, in the presence of non-determinism.

FASER employs two key techniques:

- To estimate the passing probability (or conversely flakiness) of the test, we leverage *concentration inequalities* from probability theory [14], [15], [16], [17]. Concentration Inequalities provide conservative *probabilistic bounds* on *how much* a random variable deviates from a given value (e.g., its mean). We use them to estimate the probability that the variable (in the example `val_error`, sampled over multiple runs) does not exceed a specific assertion bound (`exp_max_error`). They key advantages of using concentration inequalities are that they are non-parametric, hold under very mild assumptions, and can be effectively used to reason about a wide class of distributions.
- To determine the fault-detection effectiveness of the test, we apply Mutating Testing [18] – a classic testing methodology originally proposed for evaluating test effectiveness by generating artificial bugs (to simulate real bugs). In addition to standard mutation operators, to simulate developer mistakes, we design *domain-specific* mutations that substitute commonly used APIs with other APIs that share the same input/output specifications and perform similar functions. For FASER, we require mutants that simulate accuracy bugs in code, i.e., mutants that 1) do not crash and 2) produce a distribution of the assertion variable, X, that is *sufficiently different* than the original distribution. We define such mutants as *effective mutants*.

For a given test, FASER constructs an objective function as the weighted sum of its passing probability (estimated using concentration inequalities) and its fault-detection effectiveness (*mutation score* – the average probability of killing a mutant). FASER allows the developer to choose a test's minimum passing probability, $\gamma \in [0, 1]$. FASER then solves the optimization problem and computes the optimal bound, such that the fault-detection effectiveness of the test is maximized while ensuring the test passing probability is at least γ.

Results. We evaluate FASER on 87 non-deterministic tests collected from the latest versions of 22 ML projects, chosen from dependent projects of popular probabilistic programming systems and machine learning frameworks such as PyTorch [19], TensorFlow [20], Pyro [21], and PyMC [22]. These projects are popular, have a wide user base, and provide various ML functionalities. For each project, we only select tests that are non-deterministic due to randomness of the ML algorithm under test and contain an *approximate assertion*.

FASER is able to improve the fault-detection effectiveness for 26.4% (i.e., 23/87) of the studied tests by tightening the assertion bounds while maintaining a high passing probability (γ) of over 99%. Overall, FASER increases the mutation score of such tests by 15.76 percentage points on average. To date, we have sent 19 pull requests, each improving one test, to the developers. Developers have accepted 14 pull requests, while the rest are pending response (no rejected pull requests). The feedback from developers has been largely positive: they appreciated the extensive analysis done in determining the proposed bound, demonstrating the practical value of FASER.

Contributions. This paper makes the following contributions:

- ⋆ **Concept.** This paper opens up a new dimension for balancing test flakiness and effectiveness of ML projects, and can inspire more future works in this important direction.
- ⋆ **Implementation.** We implement FASER, a technique for improving the fault-detection effectiveness of non-deterministic tests in ML projects by effectively combining statistical techniques and mutation testing. FASER is available at: https://github.com/ise-uiuc/FASER
- ⋆ **Evaluation.** We evaluate FASER on 22 popular ML projects, and demonstrate that FASER can tighten the assertion bounds in 12 studied projects and 23 out of 87 studied non-deterministic tests for improving their test effectiveness without incurring new flakiness issues.
- ⋆ **Practical Impacts.** To date, we have sent 19 pull requests fixing the tests with loose bounds, of which 14 have already been accepted. We have received overwhelming positive feedback from the developers, and the first two authors have even been invited to a developer meeting for one of the studied popular ML projects.

II. Example

```
1 def test_adv_example_success_rate_linf(self,
      model, **kwargs):
2   x=torch.randn(100, 2)
3   x_adv=self.attack(model_fn=model,x=x,**kwargs)
4   _, ori_label=model(x).max(1)
5   _, adv_label=model(x_adv).max(1)
6   adv_acc=adv_label.eq(ori_label).sum().to(torch.
        float)/x.size(0)
7   self.assertLess(adv_acc, 0.5)
```

Listing 2: Test in *cleverhans*

Listing 2 presents an example non-deterministic test, `test_adv_example_success_rate_linf`, from *cleverhans-lab/cleverhans* project [23]. *Cleverhans* provides implementations of various adversarial attacks for machine learning models and algorithms. The test contains an assertion with a very conservative bound (Line 7), which allows many potential faults to remain undetected. In this work, we aim to improve the fault-detection effectiveness of such tests without making them *flaky*. We describe the test next.

The test generates random input data from a normal distribution (Line 2). It generates adversarial input using SPSA adversarial attack algorithm [24] on the original input data (Line 3). It obtains the model output for both the input data and the adversarial input (Lines 4-5). It computes the adversarial accuracy by comparing the adversarial output with the original output of the model (Line 6). Finally, the test checks that the adversarial accuracy is less than the assertion bound of 0.5 (Line 7). Here, lower adversarial accuracy is better since it means the attack is successful in altering the model's output.

Sources of randomness. There are several sources of randomness in this test. Listing 3 shows how the attack algorithm randomizes the starting state. The initial perturbation of the adversarial attack is sampled uniformly between -0.5 and 0.5 (Line 3, with `eps = 0.5`). Further, the input data is also generated randomly in the test (Listing 2, Line 2). Due to the randomness present, the adversarial accuracy (`adv_acc`) will be different across runs. We present the samples of `adv_acc` across 100 test executions in Figure 1 (blue bars). We observe that the distribution of samples falls below the assertion bound of 0.5 (red dotted line). This is a problem since a bug in the code can potentially reduce the adversarial attack effectiveness and produce an output that is higher than the maximum legal value but lower than the initial bound (0.5), allowing the bug to remain undetected.

```
1 def spsa(model_fn, x, eps, norm, ...):
2     ...
3   perturbation=(torch.rand_like(x)*2-1)*eps
4   _project_perturbation(perturbation,norm,eps,...)
```

Listing 3: Random initialization in *cleverhans*

Example bug. Listing 4 shows an example bug that can be introduced in the source code. We modify - to a + operator,

```
perturbation=(torch.rand_like(x)*2 - 1)*eps
perturbation=(torch.rand_like(x)*2 + 1)*eps
```

Listing 4: Example bug in source code

which changes the initial perturbation generation to sample between [0.5, 1.5] instead of [-0.5, 0.5]. We present the output samples of the adversarial accuracy when run under the buggy version in Figure 1 (blue bars). The distribution of the buggy values is below the original developer bound. However, the buggy distribution has a higher mean value compared to the original distribution – it means the bug is undesirably reducing the attack effectiveness. However, due to the loose bound (red line), the test cannot detect the bug. Bugs like these represent subtle *accuracy bugs* that can remain undetected if the tests are not properly designed. These tests cannot be fixed by the tools that loosen the bound to reduce the flakiness [8] (they move the red line to the right). Furthermore, making the test deterministic by fixing the random seed may conceal faults that can only be exposed from a different random input [12].

Instead, we want an assertion bound that is *loose enough* to allow all valid executions to pass (and minimize flakiness) but *tight enough* to catch all such buggy executions (i.e. move the red line to the left).

Our solution. To find an optimal assertion bound, we need to carefully consider the trade-off between test passing probability and its fault-detection effectiveness. To solve this problem, FASER performs several steps. First, to estimate the fault-detection effectiveness of the test, FASER uses mutation testing to generate mutants on code lines covered by the test. Since the test is non-deterministic, FASER runs both versions of the test – with and without mutations – multiple times to obtain the output samples of the assertion value. The mutants that significantly change the output distribution are most relevant since they represent bugs that the original assertion bound may fail to detect. We refer to these mutants as *effective mutants*. To identify effective mutants, FASER compares the output distribution of each mutant with the original distribution using Kolmogorov-Smirnov (KS) test and obtain a set of effective mutants whose distribution is sufficiently different from original. For a given assertion bound, FASER computes the mutation score as the average probability of killing such mutants. For this test, FASER generated 62 mutants and identified 13 effective mutants.

Second, to estimate the passing probability of the test FASER collects samples of the assertion variable (`adv_acc`) by executing the test multiple times. FASER then applies concentration inequalities to compute the probability that the assertion variable does not exceed the given bound, which approximates the passing probability of the test.

Finally, FASER determines the optimal bound that maximizes both the passing probability and mutation score of the test. In this case, FASER estimates that the optimal bound is 0.29. FASER estimates that the passing probability of the test using this bound is greater than 99% and it also increases the mutation score of the test by 11%.

III. Problem Formulation

We describe the problem FASER solves. Consider a test T and an approximate assertion A of the form: `assert X <` θ. We will refer to such θ as the *assertion bound*. The assertion bound set by a developer *may be loose*, i.e., the bound, θ, may be much higher than the possible legal values of `X`. The problem with such a loose bound is that it may miss detecting faults that produce erroneous values in the region between the original maximum of `X` and bound θ. Our goal is to determine an optimal bound, θ^*, that *maximizes both the fault-detection effectiveness* and *passing probability* of the test while keeping *passing probability* above a developer-selected threshold γ. Formally,

$$\begin{aligned} \textbf{max} \quad & [\mathit{Effectiveness}(T,\theta), \mathit{PassingProb}(T,\theta)] \\ \textbf{\textit{s.t.}} \quad & \mathit{PassingProb}(T,\theta) > \gamma \end{aligned} \tag{1}$$

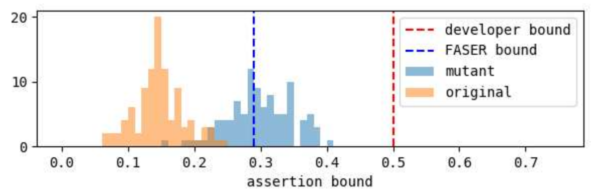


Fig. 1: Original and mutant samples of test in *cleverhans*

We need to tackle multiple challenges to solve this task. First, since the nature of the underlying distribution of X is not known, it is difficult to predict *how likely* is the test to pass for a given bound: *PassingProb*(T, θ). We address this challenge using several *concentration inequalities* from probability theory (Section III-A) to estimate the probability of the given variable (X) exceeding a given bound (θ). We describe our solution approach using the inequalities in Section III-B.

Second, estimating the fault-detection effectiveness of the test, *Effectiveness*(T, θ), is challenging because we need to reason about how the distribution of values (for the assertion variable X) shifts in presence of a fault in the code under test. To solve this challenge, we extend traditional mutation testing techniques to generate mutants that produce such distribution shifts. We also propose a new metric for computing mutation score for such non-deterministic tests (Section III-C).

Finally, since these two objectives are competing, we use a multi-objective optimization technique to solve the problem. We show how we can easily solve this constrained multi-objective problem (Eq. 1) by transforming it into an unconstrained single-objective version whilst satisfying the developer-selected minimum passing probability threshold, γ. The threshold, γ, allows us to prune the search space and choose an optimal solution from the Pareto frontier. We provide more details in Section III-D.

A. Concentration Inequalities

Chebyshev's inequality [15] is a well-known result in probability theory that guarantees that, for a broad class of distributions, no more than a certain proportion of its values will exceed a certain distance from the mean. More formally, let X be a scalar random variable with a finite mean μ and variance σ^2. Then for any real number $k > 0$: $\Pr(|X - \mu| \geq k\sigma) \leq \frac{1}{k^2}$.

Chebyshev's Inequality can be applied to any arbitrary distribution, assuming known mean and variance. However, it often provides a conservative estimate.

Dasgupta's Inequality. Dasgupta [16] proposed a tighter bound if the distribution is known to be Gaussian:

$$\Pr(|X - \mu| \geq k\sigma) \leq \frac{1}{3k^2} \tag{2}$$

Dasgupta's Inequality can also be used with estimated mean ($\overline{X}$) and variance (S^2) [16], i.e., computed from available samples. This provides a conservative estimate but is typically tighter than Chebyshev. Since the distribution is Gaussian, it is symmetric about mean. Hence, we can also use the one-sided variant of this inequality (using estimated mean and variance):

$$\Pr(X - \overline{X} \geq kS) \leq \frac{1}{2} \cdot \frac{1}{3k^2} \tag{3}$$

Cantelli's inequality with estimated mean and variance. Often the mean and variance of a distribution are unknown. Tolhurst [25] proposed a variant of Cantelli's equation [17], which is the one-sided version of Chebyshev's Inequality. It uses mean and variance estimated from available samples:

$$\Pr(X - \overline{X} \geq kQ_N) \leq \frac{1}{N+1} \left\lfloor \frac{N+1}{g^2+1} \right\rfloor \tag{4}$$

where N is number of samples, $g^2 = \frac{Nk^2}{N-1+k^2}$, $Q_N^2 = \left[\frac{N+1}{N}\right] S^2$, and $\lfloor . \rfloor$ is the floor function. Here, $\overline{X}$ and S^2 are the sample mean and variance respectively. This inequality holds for $k > 1$ and $N \geq 2$. This bound converges to Cantelli's inequality as $N \to \infty$.

Sub-Gaussian Distributions. A Sub-Gaussian distribution [26] has a tail that decays (or converges) at least as fast as a Gaussian distribution. Intuitively, the tail of a Sub-Gaussian distribution is dominated by a Gaussian distribution. This is a useful property since the tail properties of Gaussians can be extended to such distributions directly. For instance, if we know that the underlying distribution of a random variable X is sub-gaussian, then we can directly use the Dasgupta inequality (Eq. 3) to estimate the tail probabilities.

Kurtosis Test [27] is a statistical test that is used to check whether the distribution is heavy-tailed or light-tailed (sub-gaussian) relative to a Gaussian distribution. We use the Kurtosis test to verify if the samples are drawn from a Sub-Gaussian distribution and apply the Dasgupta Inequality (Eq. 3) to obtain the one-sided tail probability. This provides a tighter tail bound than the more general estimate (Eq. 4).

B. Computing Passing Probability of Test

To compute the passing probability of test T, we first execute the test several times and obtain the samples of the variable, say X, in the assertion. Let N be the number of samples. Now, we want to compute the probability: $\Pr(X < \theta)$, where θ is a given *assertion bound*. One potential solution is to compute the empirical probability, i.e., the proportion of samples that fall below the assertion bound. However, this may not be a reliable estimate when N is small and when the nature of distribution is not known. Further, collecting a large number of samples is expensive and hence may not always be feasible.

To overcome these challenges, we apply the inequalities described in Section III-A. Algorithm 1 describes the steps for computing passing probability. First, we check if the underlying distribution of X is Gaussian (Line 5) using Shapiro-Wilk Test [28]. If the distribution is Gaussian, we use the one-sided Dasgupta inequality (Eq. 3) to estimate the passing probability of the test for the given bound (Line 6). Given a bound θ, we compute $\Pr(X < \theta) = 1 - \Pr(X - \overline{X} \geq kS)$ as the passing probability of the test, where $k = (\theta - \overline{X})/S$ (Line 4). Here, $\overline{X}$ is the estimated mean and S is the estimated variance of the samples.

Second, if the distribution is not Gaussian, we check if it is Sub-Gaussian (Line 5) using the Kurtosis Test (Section III-A). If the test passes, we apply the one-sided Dasgupta inequality (Eq. 3) in the same manner as the previous case (Line 6). This provides a more conservative bound than the actual underlying distribution but it is tighter than the general estimate.

Third, if the distribution is neither Gaussian nor Sub-Gaussian, we apply Cantelli's inequality using estimated mean and variance (Eq. 4) to compute the passing probability of the test (Line 8). This inequality gives us a conservative estimate of the actual passing probability – i.e., in the limit, the actual passing probability is guaranteed to be equal or greater than the estimated passing probability. This property is desirable in our case because it forces us to choose a slightly higher bound than what may be required and avoid flaky failures.

Algorithm 1 Passing Probability Algorithm

```
Input: Samples $\mathcal{D}$, Bound $\theta$
Output: Passing probability $P_\theta$
 1: procedure PASSINGPROB($\mathcal{D}$, $\theta$)
 2:     $\overline{X} = mean(\mathcal{D})$
 3:     $S = std(\mathcal{D})$
 4:     $k = \frac{\theta - \overline{X}}{S}$
 5:     if ISGAUSSIAN($\mathcal{D}$) or ISSUBGAUSSIAN($\mathcal{D}$) then
 6:         $P_\theta = 1 -$ DASGUPTAINEQ($\mathcal{D}, k$)            ▷ Using Eq. 3
 7:     else
 8:         $P_\theta = 1 -$ CANTELLIINEQEST($\mathcal{D}, \overline{X}, S, k$)   ▷ Using Eq. 4
 9:     return $P_\theta$
10: end procedure
```

C. Estimating Fault-Detection Ability of Test

We use mutation testing to determine the fault-detection effectiveness of the test. We generate mutants and select the subset of mutants that produce distributions of the assertion variable that are *sufficiently different* from the original distribution. These mutants represent *accuracy bugs* in code, i.e., bugs that lead to wrong results (e.g., lower model accuracy). We define such mutants as *effective mutants*.

Let $m_1, \ldots, m_K$ be the generated set of effective mutants. For a given bound, θ, we define the probability that a mutant, m_i is killed as:

$$\Pr_\theta(m_i \text{ is killed}) = \Pr(X_{m_i} > \theta) = \frac{1}{|D_{m_i}|} \sum_{x \in D_{m_i}} \mathbb{1}_{[x>\theta]} \quad (5)$$

where D_{m_i} is the set of samples obtained and X_{m_i} is the assertion variable when T is executed with mutation m_i.

We define the *mutation score* (MS) of the test, T, for a given θ, as the average mutant kill rate:

$$\text{MS}(T, \theta) = \frac{1}{K} \sum_{i=1 \ldots K} \Pr_\theta(m_i \text{ is killed}) \quad (6)$$

Our definition of mutation score is different than that used in traditional mutation testing where each mutant is *deterministically* either killed (1) or the mutant survives (0). In contrast, for non-deterministic tests, we define whether a mutant is killed as a *probability*, $\Pr_\theta \in [0, 1]$, and the mutation score as the *average probability of killing a mutant*. For deterministic tests, it reduces to standard mutation score metric, i.e., $\Pr_\theta \in \{0, 1\}$.

D. Finding Optimal Assertion Bound

Our goal is to find an assertion bound that has a high fault-detection effectiveness and is not flaky. However, in practical scenarios, there is a trade-off between these two properties of a test. To select an optimal assertion bound θ^*, we transform the constrained optimization problem (Eq. 1) into an unconstrained version by using the weighted sum method [29]:

$$\theta^* = \operatorname*{argmax}_\theta \alpha \cdot p(T, \theta) + (1 - \alpha) \cdot f(T, \theta) \quad (7)$$

where $p(T, \theta)$ represents the probability that the test T passes for a given bound θ and $f(T, \theta)$ represents the fault-detection effectiveness of the test for the given bound θ. Here, we approximate the probability of passing by estimating the probability of the assertion variable X not exceeding θ using Algorithm 1 while incorporating developer-specified threshold, i.e., $p(T, \theta) = \Pr(X < \theta)$ `if` $(\Pr(X < \theta) > \gamma)$ `else` $-\infty$. We approximate the fault-detection effectiveness of the test using mutation score (Eq. 6). Hence, $f(T, \theta) = MS(T, \theta)$.

Here, $\alpha \in [0, 1]$ is a co-efficient that determines relative importance of the two factors during optimization. For instance, an $\alpha > 0.5$ lays greater emphasis on the passing probability of the test while $\alpha < 0.5$ prioritizes the fault-detection effectiveness of the test over test flakiness. Finally, we solve this problem using the standard optimization algorithm (basinhopping [30]) and obtain the new bound θ^*.

IV. FASER

We propose FASER, an approach for improving the fault-detection effectiveness of non-deterministic tests in Machine Learning projects. Figure 2 presents the architecture of FASER. At a high level, FASER takes a non-deterministic test T with an assertion A of the form `assert (X < θ)`, and minimum passing probability γ as inputs. FASER then determines a new assertion bound θ^* such that the mutation score of the test is maximized while also ensuring that the test passes with at least γ passing probability.

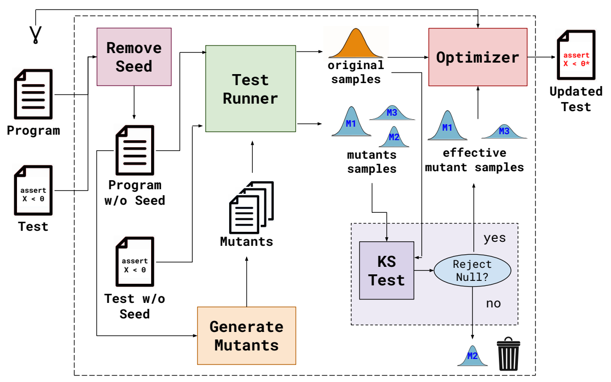


Fig. 2: Overview of the FASER tool

A. FASER Algorithm

Algorithm 2 describes the main algorithm of FASER. First, FASER removes all seed setting code from common library APIs (Python's Random, NumPy, TensorFlow and PyTorch) used in tests from the source project P (Line 2) and creates a new version P^*. Setting seeds makes the test execution deterministic but prevents FASER from collecting the legal range of values that X can take. Without knowing the legal range of values for X, FASER may not be able to identify erroneous values produced by a mutation.

FASER executes the test T using P^* multiple times to collect samples of the actual assertion values: D_O (Line 3). Then FASER computes code coverage of the test (Line 4) and generates a set of mutants, M, by mutating covered lines of code (Line 5). FASER generates each mutant using several mutation operators (Section IV-B) that change a part of the line. FASER initializes an empty set D_M that we use to

Algorithm 2 FASER Algorithm

```
Input: Test $T$, Project $P$, Assertion $A$, Minimum Passing Probability $\gamma$
Output: Updated test $T^*$, Original Mutation Score $MS_\theta$, New Mutation Score $MS_{\theta*}$
 1: procedure FASER($T$, $P$, $A$)
 2:     $P^* \leftarrow RemoveSeed(P)$
 3:     $D_O \leftarrow TestRunner(T, P^*, A, \texttt{ORIGINAL\_SAMPLES})$
 4:     $coverage \leftarrow RunTestCoverage(T, P^*)$
 5:     $M \leftarrow GenerateMutants(P^*, coverage)$
 6:     $D_M \leftarrow \emptyset$
 7:     for $m$ in $M$ do
 8:         if $!TestCrash(T, m, A)$ then
 9:             $d_m \leftarrow TestRunner(T, m, A, \texttt{MUTANT\_SAMPLES})$
10:             $p \leftarrow KSTest(d_m, D_O)$
11:             if $p < \rho$ then                    ▷ rejects null hypothesis
12:                 $D_M \leftarrow D_M \cup d_m$
13:     $MS_\theta, MS_{\theta^*}, \theta^* \leftarrow Optimizer(T, A, D_O, D_M, \gamma)$
14:     return $Patcher(T, \theta^*), MS_\theta, MS_{\theta*}$
15: end procedure
```

aggregate samples from the mutants that generate distributions of X that are sufficiently different than the samples produced by the original version of the test, D_O (Line 6).

For each mutant, FASER first checks if the mutation does not crash the test (Line 8). Then FASER collects samples from assertion value over several test executions, d_m (Line 9). Using Kolmogorov-Smirnov (KS) test [31], [32], FASER compares the mutant samples with the original samples (Line 10) to see if their distributions are sufficiently different. This allows us to select mutants that represent accuracy bugs. KS test returns a p-value for testing the null hypothesis: the underlying mutant distribution is identical to the original distribution. If the p-value is less than the threshold ρ, we can reject the null hypothesis, in which case FASER adds the mutant samples to D_M (Line 12). We refer to the mutants that produce a different distribution of samples than the original as *effective mutants*.

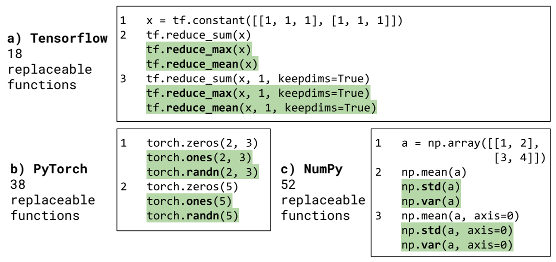
Finally, FASER solves the optimization Equation 1 using the original and effective mutant samples (Line 13) to obtain the optimal bound with at least γ passing probability. FASER returns the mutation score of the original bound and the optimal bound along with the updated test (Line 14).

B. Mutant Generation

For each test, FASER generates mutations on code the test covers. It first runs the test using Python's *Coverage.py* [33] to obtain a list of Python files and lines covered by the test. It filters out all *test* files and uses only source files. To generate mutations, FASER uses traditional mutation operators from the *mutmut* [34] library and also implements new *domain-specific* mutation operators for Machine Learning projects.

Simple mutation operators. FASER uses all *mutmut* mutation operators, such as the ones on numeric constants, keywords (e.g., `break` → `continue`), arithmetic operators, and conditional expressions (`and` → `or`). The only one we did not use is the string mutator (e.g., `p="test"` → `p="XXtestXX"`) because it most often leads to code crashes. Details of each operator can be found on project page [34].

Domain-specific mutation operators. Many Machine Learning projects utilize popular open-source libraries such as TensorFlow or PyTorch as they provide efficient implementations



```
1   x = tf.constant([[1, 1, 1], [1, 1, 1]])
2   tf.reduce_sum(x)
    tf.reduce_max(x)
    tf.reduce_mean(x)
3   tf.reduce_sum(x, 1, keepdims=True)
    tf.reduce_max(x, 1, keepdims=True)
    tf.reduce_mean(x, 1, keepdims=True)
```

```
1   torch.zeros(2, 3)
    torch.ones(2, 3)
    torch.randn(2, 3)
2   torch.zeros(5)
    torch.ones(5)
    torch.randn(5)
```

```
1   a = np.array([[1, 2],
                  [3, 4]])
2   np.mean(a)
    np.std(a)
    np.var(a)
3   np.mean(a, axis=0)
    np.std(a, axis=0)
    np.var(a, axis=0)
```

Fig. 3: Code snippets of replaceable APIs

of many common functionalities (e.g., random data generation, matrix multiplication, and array/tensor manipulation). Many APIs in these libraries share similar input/output specifications, i.e., they have same types of inputs and outputs and perform similar functions. Hence, we can interchange such API calls without causing a program crash. For example, in PyTorch, tensor initialization functions such as `torch.zeros` and `torch.ones` have the same input parameters (including tensor shape and data type) and outputs a tensor that has the specified shape with either all zeros or ones respectively. We can replace one of them with another without causing the program to crash. Actually, developers often make such mistakes of choosing syntactically correct but semantically incorrect APIs for their use-case [35], [36], [37]. By leveraging this insight, we design a mutation operator that swaps such commonly used interchangeable APIs from the same library. We next describe how we collect such APIs.

We consider three popular open-source libraries: NumPy, TensorFlow, and PyTorch as majority of projects in our dataset have at least one of them as a dependency. We first extract all public APIs from each library. We use *pydoc*, a built-in python documentation generator, to obtain developer documentation of each API. Since Python is not a strongly typed language, we cannot directly compare the method signature to determine APIs that have same input and output types. Instead, we use the developer documentation that includes example code snippets listing the usages for each API.

For each API (say A), we test whether replacing it with another API (say B) causes a crash. If it does not crash, we add the API B to A's list of replaceable APIs and vice-versa. We perform this for all APIs and obtain sets of replaceable APIs for each API in the library. It is possible that developer written examples do not cover all use-cases, which means the replaced APIs might fail in some special cases.

Figure 3 shows code snippets of various APIs and their potential replacements (in green) for the 3 open-source libraries we used. In total, we obtain 108 replaceable APIs with an average of 1.87 replacements for each. FASER leverages these API specifications to generate mutations where applicable.

C. Estimating Optimal Assertion Bound

FASER uses the original and effective mutant samples as inputs to the optimization equation (Equation 7), and obtains the optimal assertion bound. Recall that the optimization is parameterized by α – representing the relative importance between passing rate and the mutation score. A high α means we value the passing rate higher (i.e., low flakiness) compared

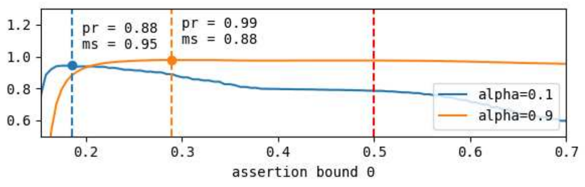


Fig. 4: Optimization graph of test in *cleverhans*

to the fault-detection effectiveness of the test. A low α means we are willing to sacrifice the passing rate in favor of catching more potential bugs in the source code.

To calculate the bound, FASER uses the optimization equation by varying α. We start with $\alpha = 0.5$ (represents equal emphasis on pass rate and fault-detection effectiveness). In practice, developers favor tests that have a high pass rate to minimize failures caused by flaky tests. FASER does a grid-search over higher α values, solves the optimization problem, and finally yields the bound that results the maximum fault effectiveness and passing probability above γ.

Figure 4 presents the optimizing equation graph for the example test described in Section II. The two solid lines present the values of the optimization equation (y-axis) for α of 0.9 and 0.1 as we vary the assertion bound (x-axis). The vertical dotted lines denote the corresponding argmax values of θ. We chose a minimum passing probability γ of 0.99. This passing probability is a conservative theoretical estimate (Section III-A), and the empirical passing probability would be higher – which we demonstrate in our evaluation (Section VI-C). The optimal value using an α of 0.1 is $\theta = 0.18$ – while this gives us the best mutation score of 95% (i.e., best fault-detection effectiveness), the test may become very flaky (0.84 pass rate). An α of 0.9 gives an optimal bound: $\theta = 0.29$ – this has a high probability of passing (0.9903 pass rate) and also maintains a high mutation score of 88%. Hence, FASER selects θ = 0.29 as the optimal bound.

D. Applicability of Assertion Bounds

In Section IV-C, we described FASER's strategy for estimating a bound in the general case. However, some tests may already have a tight bound or a high fault-detection effectiveness. Hence, FASER must decide on how and if the calculated bound should be applied.

FASER starts by computing the optimal bound as described in Section IV-C. FASER then compares the fault-detection effectiveness (mutation score) of the optimal bound and developer bound and decides whether the bound should be updated. There are two possible scenarios: 1) If the increase in mutation score is significant (greater than 5 percentage points), FASER labels this test as a **loose bound test** and uses the proposed bound suggested by FASER to improve this test. 2) If the increase in mutation score is not significant (less than 5 percentage points), this means the developer bound is already tight enough. As a result, the new bound proposed by FASER cannot significantly improve the fault-detection effectiveness. FASER labels this test as a **tight bound test** and does not change the developer bound.

V. Methodology

Project and Test selection. To select tests for evaluation, we use Python projects and tests used in previous work on testing in ML projects [9], [8], [12]. Previous work find such projects by first searching for the dependent projects of popular ML Frameworks (PyTorch and TensorFlow) and Probabilistic Programming Systems (Pyro, NumPyro, TensorFlow-Probability) using GitHub's API. Out of these, they select projects that can be installed as a Python library and have at least 10 stars to eliminate projects that are not actively developed and toy projects. Due to the potential time cost of running FASER, we randomly select 25 out of these 123 projects for our evaluation. We only consider projects that were reported to contain at least one non-deterministic test in previous works.

We run each test and record their assertion values to verify the results vary across executions. We discard any tests where the value remains the same across executions. We also discard tests that may require special resources to run (e.g., GPU). We do not include any tests that are flaky (i.e., do not have 100% empirical pass rate with original bound), since this implies that the original bound may already be very tight (i.e., close to maximum observable value). From the remaining list of tests, we randomly sample a subset to include in our experiment. Finally, we end up with 87 non-deterministic tests across 22 projects from 25 total projects considered. Table I presents for each selected project a description of its utility (**Description**) and the number of selected tests (**T**).

Reporting Fixes to Developers. We prepare and send pull requests to developers for fixing tests FASER can tighten. We first manually inspect the proposed bound and determine if we need to adjust it. For instance, we may round the proposed bound to the nearest integer (e.g., $4.99 \rightarrow 5$) if the original bound was also an integer (e.g., 20). We then run the test with the proposed bound 500 times to verify that the proposed bound is not flaky (greater than 0.99 pass rate). For each project, we start by only sending one pull request for fixing one test. If the developers accept the initial pull request, we send more pull requests for fixing remaining tests in the project. In each pull request, we explain FASER's methodology, show the distribution of original and mutant samples, and explain trade-offs for pass rate vs. mutation score for the test.

Experimental setup. For our evaluation, we use the latest versions of each project. We develop a custom installation script for each project that creates a virtual environment using Anaconda [38] and installs any dependencies required by each project specified in `requirements.txt` or `setup.py` files in the project. We additionally install two python libraries: *pytest* [39] for running the tests and *Coverage.py* [33] for obtaining the line coverage information.

We implement FASER entirely in Python. We configure FASER to obtain 100 samples for both the original source code (`ORIGINAL_SAMPLES`) and each mutant (`MUTANT_SAMPLES`). We use a minimum passing probability of 0.99 (γ). We use a timeout of 300 seconds for each test execution. We use two-sample KS Test from the *scipy* library [40] with p-value

TABLE I: Running FASER on Non-Deterministic Tests

Project	Description	T	LB	TB
allenai/allennlp [41]	NLP	5	1	4
pytorch/captum [42]	ML Model Interpretability	1	0	1
cleverhans-lab/cleverhans [23]	Adversarial Attacks	5	2	3
coax-dev/coax [43]	Reinforcement Learning	11	0	11
deepchem/deepchem [44]	DL for Natural Science	6	2	4
RaRe-Technologies/gensim [45]	Topic Modelling	1	1	0
GPflow/GPflow [46]	Gaussian Process Modelling	2	0	2
cornellius-gp/gpytorch [47]	Gaussian Process Modelling	7	5	2
kornia/kornia [48]	Computer Vision	2	0	2
learnables/learn2learn [49]	Meta Learning	6	0	6
Unity-Technologies/ml-agents [50]	Training ML agents	3	1	2
pyro-ppl/numpyro [51]	Probabilistic Programming	2	1	1
facebookresearch/ParlAI [52]	Dialog AI modelling	1	1	0
pgmpy/pgmpy [53]	Graph Model	4	0	4
pymc-devs/pymc [22]	Probabilistic Programming	4	1	3
pyro-ppl/pyro [21]	Probabilistic Programming	3	0	3
refnex/refnx [54]	Curve Fitting	2	0	2
stellargraph/stellargraph [55]	Graph Modelling	3	1	2
WillianFuks/tfcausalimpact [56]	Bayesian Optimization	1	0	1
google/trax [57]	Code Generation	6	4	2
lmcinnes/umap [58]	Visualization	9	3	6
zfit/zfit [59]	Model Fitting	3	0	3
Total		87	23	64

threshold of 0.01 (ρ) to compare the original and mutant sample distributions. For solving the optimization equation, we use the basinhopping algorithm [30] from the *scipy* library.

VI. Evaluation

We evaluate FASER on the following research questions:

RQ1 For how many tests can FASER improve their fault-detection effectiveness by tightening assertion bounds?

RQ2 By how much does FASER improve the fault-detection effectiveness of the tests?

RQ3 How do developers respond to the tighter bounds suggested by FASER?

RQ4 What is the cost of running FASER and can it be reduced?

A. RQ1: Tests Improved by FASER

We run FASER on 87 tests collected across 22 projects to improve their fault-detection effectiveness by tightening assertion bounds. Table I presents the results. Each row in the table corresponds to the tests in one project. **Project** is the name of the project, **T** is the number of tests, **LB** is the number of *loose bound* tests – tests for which FASER can improve the mutation score by at least 5 percentage points by tightening the assertion bound, **TB** is the number of *tight bound* tests – tests for which FASER cannot improve the mutation score by at least 5 percentage points by tightening bounds. We observe that approximately one out of four tests contain an assertion bound that is loose (more *conservative* than needed). We find that 12 out of 22 projects contain at least one test(s) that has a loose bound. For these tests, FASER is able tighten the bound and improve the mutation score of the tests.

Cases where FASER improves the mutation score of test. Figure 5 presents a loose bound test with the developer bound of 0.5 (red dotted line) and FASER proposed bound of 0.29 (blue dotted line). The test is `test_adv_example_success_rate_linf` from the *cleverhans-lab/cleverhans* project. We observe that the new bound proposed by FASER can improve the mutation score of the test.

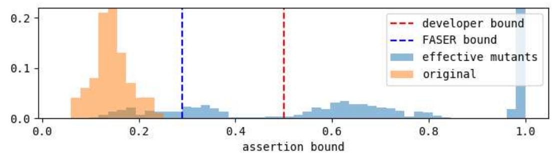


Fig. 5: Normalized output distribution of the original and the effective mutants for a loose bound test in *cleverhans*

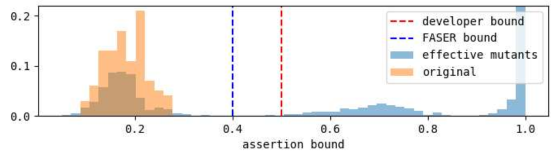


Fig. 6: Normalized output distribution of the original and the effective mutants for a tight bound test in *cleverhans*

FASER generates a significant number of effective mutant samples between the maximum value of the original samples (0.24) and the developer bound (15.5% of all effective mutant samples). Hence, FASER is able to improve the mutation score of the test by 11 percentage points using the new bound, increasing the mutation score from 76.7% to 87.8%. In total, FASER tightens the bounds of 23 out of 87 non-flaky tests.

Cases where FASER cannot improve mutation score. Figure 6 shows the original samples and aggregated mutant samples of `test_adv_example_success_rate_l2` from *cleverhans-lab/cleverhans* project where FASER cannot tighten the bound. We observe that even though FASER proposed a bound of 0.4 (blue dotted line) that is less than the original developer bound of 0.5 (red dotted line), the mutation score of the new bound is not higher than the original. This is because, in contrast to the previous example, there are very few effective mutant samples (only 1.7% of all effective mutant samples) between the original developer bound and the maximum value of the original samples. Therefore, the bound suggested by FASER cannot significantly improve the mutation score (more than 5pp) of the test. Further, to kill 1.7% of effective mutant samples, we need to set the bound to be exactly the max value (0.27), which may decrease the passing probability of the test, making it flaky. The bound set by the developer in this case is already tight enough to kill majority of the mutants (mutation score of 55%) and hence fault-detection effectiveness cannot be improved significantly.

B. RQ2: Improvement in Fault-Detection Effectiveness

We next discuss tests that FASER can tighten and examine by how much it improves the fault detection effectiveness of such tests. Table II presents the details of improvements in fault-detection effectiveness. Each row presents a project with at least one loose bound test. Column **Project** is the project name, **#T** is the number of loose bound tests in the project, **Avg.# Eftv. mutants** is the average number of effective mutants generated by FASER, **Avg.pp MS increase** is the average increase in mutation score (percentage points) obtained by comparing the developer bound and the tight bound proposed by FASER, **Avg.% MS before** is the average mutation score of the original bound, **Avg.% MS after** is the average mutation

TABLE II: Improvement in Fault-Detection Effectiveness

Project	#T	Avg. #Eftv. mutants	Avg.% MS before	Avg.% MS after	Avg.pp MS increase	Avg. α
allennlp	1	13.0	51.08	66.46	15.38	0.9
cleverhans	2	13.0	75.35	86.42	11.08	0.85
deepchem	2	6.5	20.66	25.75	5.08	0.75
gensim	1	29.0	57.55	77.93	20.38	0.9
gpytorch	5	47.8	33.28	57.88	24.6	0.9
ml-agents	1	6.0	33.33	66.17	32.83	0.9
numpyro	1	156.0	23.11	28.89	5.78	0.8
parlai	1	51.0	0.0	25.96	25.96	0.9
pymc	1	58.0	9.67	18.76	9.08	0.9
stellargraph	1	9.0	50.63	55.88	5.25	0.9
trax	4	15.25	15.8	34.44	18.63	0.88
umap	3	12.0	63.0	69.22	6.22	0.8
Total/Avg.	23	30.3	36.35	52.11	15.76	0.86

score using the new bound, **Avg. α** is the average α value FASER uses for determining the optimal bound.

On average, FASER improves the mutation score of tests by 15.76pp. The original developer bounds have an average mutation score of 36.35%, by using the tighter bound, FASER can improve the average mutation score to 52.11%. For the test in *facebookresearch/ParlAI*, the original developer bound has a zero mutation score - the bound is not able to kill off any effective mutants. This is because the developer bound is extremely loose. FASER proposes a tighter bound for this test and increases mutation score to almost 26%. The significant increase in mutation score indicates that the tests (with the new bound) are more likely to detect potential bugs that would previously remain undetected.

On average, FASER generates 30.3 effective mutants that significantly change the output distribution of the test. Effective mutants represent potential accuracy bugs that the test should detect. We observe that the number of effective mutants vary across different projects and tests. This is because FASER generates mutations based on lines covered by the test. Different tests have different number of coverage lines, while not all lines equally affect the test execution result – leading to a high variance in the number of effective mutants generated. FASER is able to generate a significant number of effective mutants in most cases, which gives us confidence that the computed mutation score is a practical estimate for the fault-detection effectiveness of the test.

Impact of different mutation operators. We further evaluate how each mutation operator (Section IV-B) contributes to the performance of FASER. We perform an ablation study by removing one mutation operator at a time and computing the average increase in mutation score and effective mutants in each case. Table III presents the results of this experiment. Column **Avg.# Eftv. mutants** is the average number of effective mutants generated and **Avg.pp MS increase** is the average increase in mutation score. Each row after the first presents the results when removing one mutation operator with the decrease in performance compared with FASER indicated within the brackets. The first row (None) presents the original results with FASER when using all mutation operators.

Recall that FASER obtained an average of 30.3 effective mutants and mutation score increase of 15.76pp. We observe that each mutation operator helps to generate new effective mutants and improve the mutation score. Our "Domain-specific operators" on average generate more effective mutants than most of the simple mutation operators which shows that leveraging domain knowledge of interchangeable library APIs to produce mutants that swap library APIs can further boost the performance of FASER. By using all operators, we can generate a substantial number of effective mutants per test, increasing confidence that mutation score is a good estimate of the fault-detection effectiveness.

TABLE III: Ablation Study of Mutation Operators

Operator Removed	Avg.# Eftv. mutants	Avg.pp MS increase
None	30.3	15.76
Numeric constants	23.1 (-7.2)	13.40 (-2.36)
Keywords	26.1 (-4.2)	14.22 (-1.54)
Arithmetic operators	21.8 (-8.5)	12.98 (-2.78)
Conditional expressions	28.2 (-2.1)	15.17 (-0.59)
Domain-specific operators	22.0 (-8.3)	13.21 (-2.55)

Choice of optimization co-efficient. FASER uses an average α of 0.86 to solve the optimization equation. Recall that the α value balances the trade-off between mutation score and pass rate of the test. In order to obtain a high pass rate, FASER uses a high α value most commonly α = 0.9. However, we also observed cases where FASER uses an α of 0.8 or lower in order to obtain the bound that has a high mutation score. FASER starts with a lower α and incrementally increases it until the pass rate is at least 0.99. This allows FASER to effectively tighten the bound to obtain the best mutation score while not making the tests flaky.

C. RQ3: Developer Response

We send 19 pull requests (one for each test) to the developers for tightening the assertion bounds based on FASER's results. We follow the methodology described in Section V to first open one pull request (PR) per project and only send the rest if we receive positive feedback from the developer on the first pull request. In each PR, we indicate to the developers that the bound set in the test is too loose and briefly explain the approach of FASER. We further monitor each PR to answer any questions or concerns raised by the developers.

Table IV presents the details of the PRs that we sent. Column **Project** is the name of the project, **#Tests** is the number of loose bound tests in the project, **#PRs** is the total number of PRs we send for each project, **Accepted** is the number of accepted PRs, **Pending** is the number of PRs pending developer response, **Rejected** is the number of rejected PRs, **Unsubmitted** is the number of tests for which we did not submit the PR since we are waiting for feedback on the first PR. Developers have accepted 14 of them, rejected none while 5 PRs are still awaiting developer response.

Overall, we received overwhelmingly positive feedback for the pull requests. Developers are happy about the thorough analysis of FASER in determining the new bound. For instance, developers of *cornellius-gp/gpytorch* commented: *"... this seems very reasonable to me, thanks for the detailed*

TABLE IV: **Details of Pull Requests**

Project	#Tests	#PRs	Accepted	Pending	Rejected	Unsubmitted
allennlp	1	1	1	0	0	0
cleverhans	2	1	0	1	0	1
deepchem	2	2	1	1	0	0
gensim	1	1	1	0	0	0
gpytorch	5	5	5	0	0	0
ml-agents	1	1	0	1	0	0
numpyro	1	1	1	0	0	0
parlai	1	1	1	0	0	0
pymc	1	1	1	0	0	0
stellagraph	1	1	0	1	0	0
trax	4	1	0	1	0	3
umap	3	3	3	0	0	0
Total	23	19	14	5	0	4

profile!". Developers of *facebookresearch/ParlAI* commented: *"Super cool! Awesome analysis. Very happy to try this for a bit and see how it goes."*. As we are not active contributors of the projects, we do not know the details about each developer who responded to the PR. However, we observe that all developers who responded (11/11) have contributed multiple times to their respective projects. Additionally, the developers of *deepchem/deepchem* invited us to their developer meeting on Zoom where the core team of the project discussed about our proposed change (accepted) and potentially incorporating FASER to test more bounds in their project.

The developers of *deepchem/deepchem* indicated to us that their main concern is the flakiness of the tests in the project – which forced them to choose loose bounds in the first place. They mentioned that, in the past, flaky tests have caused their developers to waste time in investigating spurious failures, which is a big concern, especially for smaller teams like theirs. However, they also appreciated our systematic approach, saying – *"I really like this style of analysis and want to see if we can tighten up our tests without blowing up false positive counts"* and accepted our proposed bound and PR. FASER addresses the flakiness concerns by enforcing a high predicted passing probability during optimization.

To demonstrate that the proposed bound from FASER is not flaky, we evaluate the passing probability estimated by FASER by comparing against the empirical pass rate of the proposed bound. We select 10 tests that FASER proposes a tighter bound for and calculate the pass rate with the proposed bound from 10,000 executions. We obtain an average pass rate of 99.88%, with 4 tests achieving 100% pass rate and the lowest pass rate being 99.64%. This confirms our hypothesis that the estimated predicted probability produced by FASER is a conversative measure and the developers can be confident that the tighter bound proposed by FASER is not flaky.

We closely monitor the tests that developers have accepted the new bound by scraping the CI logs of test runs. Out of 14 tests, only 2 tests have seen failures across thousands of CI runs. The single failure from `test_improper_normal` in *pyro-ppl/numpyro* is due to a faulty PR that triggered the CI where many tests in the test suite failed. The failures from `test_sample_after_set_data` in *pymc-devs/pymc* are caused by changes in source code that shifted the distribution of the assertion value in the test. The bound suggested by FASER is calculated based on the old distribution that caused the test to fail on the new source code. This is confirmed by the developers as they have now adjusted the bound to reflect the updated source code. The positive feedback from the developers demonstrates that FASER provides practical value by improving the quality of tests in ML projects by tightening approximate assertion bounds.

TABLE V: **Time, Cost, and Improvement Trade-off when Executing a Fraction of Mutants.**

Project	100%			75%			50%			25%		
	#	T	$	#	T	$	#	T	$	#	T	$
deepchem	1	257	6.68	0	193	5.02	0	129	3.35	0	64	1.66
gensim	1	145	3.77	1	109	2.83	1	73	1.90	1	36	0.94
numpyro	1	112	2.91	0	84	2.18	0	56	1.47	0	28	0.73
parlai	1	133	2.94	1	100	2.60	1	66	1.72	1	33	0.86
pymc	1	232	6.03	1	174	4.52	1	116	3.02	0	58	1.51
umap	3	397	10.32	2	298	7.75	1	199	5.17	0	99	2.57
trax	3	333	8.66	2	250	6.50	2	167	4.34	2	83	2.16
Count/Avg	11	279	7.25	8	209	5.44	7	140	3.62	5	70	1.81

D. RQ4: Efficiency of FASER

We collected 8700 original test samples corresponding to 87 tests in our experiement. Our experiment took around 9000 CPU-hours with majority of the time spent on running and collecting mutant samples. On average, each test required 100 CPU-hours. The corresponding estimated dollar cost of running FASER on a dedicated (more expensive than spot) Amazon EC2 a1.large instance ($0.026 / CPU hour) [60] would be $2.55 on average per test.

We expect developers to run FASER offline (once per test). However, we evaluate if it is possible to further reduce the cost of FASER when developers have a limited budget. We select 11 tests that FASER can tighten and have high run time ($\geq$ 100 CPU-hours). We attempt to reduce their runtime cost by randomly selecting a portion (75, 50, 25%) of mutants to run for these tests. Table V presents the results of this experiment with the number of tests that can be improved **'#'**, run time **'T'** (in CPU-hours) and cost **'$'**. To account for the randomness, we repeat each experiment 10 times with different random seeds and report averages.

We observe that even without running the full set of generated mutants, FASER can still tighten/improve the bounds of majority of these tests and achieve a significant decrease in execution time and cost of running FASER. The results demonstrate that developers can potentially reduce the cost of running FASER by using only a subset of mutants and still achieve close to best results.

VII. Threats to Validity

Internal. There can be potential implementation issues in FASER. To reduce such threats, the first two authors regularly checked the results and code to eliminate potential bugs.

External. Our results may not generalize beyond selected projects and tests. To reduce this threat, similar to prior work [10], [9], [8], we select from the dependent projects of popular, actively-maintained ML and probabilistic programming libraries. We believe these tests and projects are representative. Further, the number of selected tests is also on a similar scale as previous works in this domain [10], [9], [8], [12]. Similar to these works, we also assume that the code under test is correct, which means that FASER's bounds are based on the intended implementation of the source code.

Construct. Mutants may not be representative of real accuracy bugs. To mitigate this risk, we choose mutation operators commonly used in literature and also design domain-specific mutations that simulate real bugs. It may be possible to further optimize the bounds proposed by FASER. Since the tests are non-deterministic and we employ various statistical heuristics, FASER may suggest inaccurate bounds. To minimize this risk, we collect a large number of samples both for original test and all mutants to improve confidence in our results.

VIII. Related Work

Flaky Tests. Previous works have characterized the common causes of flaky tests in real world open-source projects [13], [61], [62], [63] as well as in industry [64], [62]. Researchers have developed general and specialized approaches for detecting [10], [65], [66], [67] and fixing [68], [69], [8] flaky tests.

Dutta et al. [10] studied the characteristics of flaky tests in Machine Learning and Probabilistic Programming projects. In a subsequent work, they developed an approach TERA [9], which reduces the execution time of such non-deterministic tests without making them unacceptably flaky. In contrast to FASER, TERA changes algorithm hyper-parameters and does not significantly impact fault-detection effectiveness.

FLEX [8] is an approach for fixing flaky tests by leveraging extreme value theory to *loosen* assertion bounds. Similar to FASER, FLEX also updates the assertion bounds of the test. However, unlike FASER, FLEX's approach is based on conservative estimates stemming from applying extreme value theory. Further, FLEX only reduces flakiness, but does not discuss its impact on fault-detection effectiveness of the test. It may often require thousands of samples to reach convergence and provide very conservative bounds when it converges to a heavy tailed distribution. In contrast, FASER's statistical approach aims to identify subtle accuracy differences and requires fewer samples (100 in our case).

Mutation Testing. Mutating testing has been widely studied over past few decades in the context of measuring test effectiveness in detecting faults. Jia and Harman [70] provide a comprehensive survey of mutation testing techniques. Several approaches improve mutation testing through custom mutation operators [71], [72], test prioritization and selection [73], [74], [75], and eliminate duplicate/equivalent mutants [76], [77].

Shi et al. [78] investigated the effects of flaky tests on mutation testing. They proposed techniques to mitigate the effects of flakiness by strategically re-running tests. In contrast, we apply mutation testing to help improve non-deterministic tests. Also, we execute each mutation 100 times. This allows us to use a *probabilistic* version for mutation score unlike traditional mutation testing, which expects deterministic behavior.

Metamorphic Testing. Metamorphic testing [79] has been widely used to leverage the relationships between multiple inputs (metamorphic relations) to address the test oracle problem. Because the ML domain typically lacks reliable test oracles, researchers have identified metamorphic relations for specific applications, including autonomous driving [80], [81], [82] (e.g., insensitivity of the result to weather conditions), search engines [83], [84], and machine translation [85], [86]. FASER targets tests where the metamorphic relationship between outputs is numerical (continuous).

FASER plays a complementary role to metamorphic testing (i.e., it fixes loose bounds in existing tests) and, as a special case, can be used to improve the ML tests generated by applying metamorphic relations. For example, let us consider a scenario where a developer uses metamorphic testing to provide relevant test inputs (e.g., image of an animal and the same image with slightly altered background) and output relationships (e.g., classification likelihoods of images should be *similar*) [87]. However, due to the non-deterministic nature of ML systems, the appropriate *tolerance bound* of the output relationship (e.g., allowable difference between the classification likelihoods) may not be known. FASER can help tackle this problem by running the test with different animal/altered-background images to build a distribution of the differences of the classification likelihood of the true label and estimate the optimal bound (e.g., within 5%) that aims to minimize flakiness and maximize fault-detection effectiveness of the test.

Testing Non-Deterministic Systems. Machine Learning frameworks like TensorFlow [20] and PyTorch [19] are predominantly used to develop machine learning applications. Similarly probabilistic programming systems [88], [89], [90], [91] are also gaining in importance in recent years. Recently, researchers have proposed approaches for testing machine learning frameworks [92], [93], [94], [95], [96], [97], [98], [99], [100], [101], [102], probabilistic programming systems [37], [103], [104], and randomized algorithms [105]. While these approaches focus on detecting new bugs, in this work we focus on improving fault-detection effectiveness of existing regression tests in such projects.

IX. Conclusion

We proposed a novel approach, FASER, for balancing the trade-offs between flakiness and fault-detection effectiveness of non-deterministic tests in ML projects. We found that 23 out of 87 studied non-deterministic tests contain bounds that can be tightened to improve their fault-detection effectiveness. Our observations and the positive feedback from developers reflect that FASER is practically useful.

Acknowledgments

This work was partially supported by NSF grants CCF-1763788, CCF-1956374, CCF-1846354, CCF-2008883, CCF-2131943, and CCF-2141474. We acknowledge support for research on flaky tests and ML systems from Meta and Google.

References

[1] B. Huval, T. Wang, S. Tandon, J. Kiske, W. Song, J. Pazhayampallil, M. Andriluka, P. Rajpurkar, T. Migimatsu, R. Cheng-Yue, F. Mujica, A. Coates, and A. Y. Ng, "An empirical evaluation of deep learning on highway driving," 2015.

[2] T. Davenport and R. Kalakota, "The potential for artificial intelligence in healthcare," *Future Hospital Journal*, vol. 6, pp. 94–98, 06 2019.

[3] "Understanding the fatal tesla accident on autopilot and the nhtsa probe," *electrek*, 2016, https://electrek.co/2016/07/01/understanding-fatal-tesla-accident-autopilot-nhtsa-probe.

[4] "A google self-driving car caused a crash for the first time," *The Verge*, 2016, https://www.theverge.com/2016/2/29/11134344/google-self-driving-car-crash-report.

[5] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*. MIT Press Cambridge, 2016.

[6] N. D. Goodman and A. Stuhlmüller, "The design and implementation of probabilistic programming languages," 2014.

[7] V. Francois-Lavet, P. Henderson, R. Islam, M. G. Bellemare, and J. Pineau, "An introduction to deep reinforcement learning," 2018.

[8] S. Dutta, A. Shi, and S. Misailovic, "Flex: Fixing flaky tests in machine learning projects by updating assertion bounds," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021.

[9] S. Dutta, J. Selvam, A. Jain, and S. Misailovic, "Tera: Optimizing stochastic regression tests in machine learning projects," in *ISSTA*, 2021.

[10] 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.

[11] M. Nejadgholi and J. Yang, "A study of oracle approximations in testing deep learning libraries," in *ASE*, 2019.

[12] S. Dutta, A. Arunachalam, and S. Misailovic, "To seed or not to seed? an empirical analysis of usage of seeds for testing in machine learning projects," in *ICST*, 2022.

[13] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *FSE*, 2014.

[14] M. Mitzenmacher and E. Upfal, *Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis*. Cambridge university press, 2017.

[15] P. L. Chebyshev, "Des valeurs moyennes," *J. Math. Pures Appl*, 1867.

[16] A. DasGupta, "Best constants in chebyshev inequalities with various applications," *Metrika*, 2000.

[17] F. P. Cantelli, *Intorno ad un teorema fondamentale della teoria del rischio*. Tip. degli operai, 1910.

[18] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, 2011.

[19] "Pytorch," 2018, http://pytorch.org.

[20] "Tensorflow," 2020, https://www.tensorflow.org.

[21] "Pyro," 2022, https://github.com/pyro-ppl/pyro.

[22] "Pymc," 2022, https://github.com/pymc-devs/pymc.

[23] "Cleverhans," 2022, https://github.com/cleverhans-lab/cleverhans.

[24] J. Uesato, B. O'Donoghue, A. van den Oord, and P. Kohli, "Adversarial risk and the dangers of evaluating against weak attacks," 2018.

[25] T. N. Tolhurst, "Model-free tests and evidence of bubbles in real markets," Ph.D. dissertation, University of California, Davis, 2020.

[26] V. V. Buldygin and Y. V. Kozachenko, "Sub-gaussian random variables," *Ukrainian Mathematical Journal*, 1980.

[27] F. J. Anscombe and W. J. Glynn, "Distribution of the kurtosis statistic b 2 for normal samples," *Biometrika*, 1983.

[28] S. S. Shapiro and M. B. Wilk, "An analysis of variance test for normality (complete samples)," *Biometrika*, 1965.

[29] R. T. Marler and J. S. Arora, "The weighted sum method for multi-objective optimization: new insights," *Structural and multidisciplinary optimization*, 2010.

[30] D. J. Wales and J. P. K. Doye, "Global optimization by basin-hopping and the lowest energy structures of lennard-jones clusters containing up to 110 atoms," *The Journal of Physical Chemistry A*, 1997.

[31] F. J. Massey Jr, "The kolmogorov-smirnov test for goodness of fit," *Journal of the American statistical Association*, 1951.

[32] V. W. Berger and Y. Zhou, "Kolmogorov–smirnov test: Overview," *Wiley statsref: Statistics reference online*, 2014.

[33] 2009, https://coverage.readthedocs.io/en/6.2/.

[34] "Mutmut: Python mutation tester," 2020, https://github.com/boxed/mutmut.

[35] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang, "An empirical study on tensorflow program bugs," in *ISSTA*, 2018.

[36] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella, "Taxonomy of real faults in deep learning systems," in *ICSE*, 2020.

[37] S. Dutta, O. Legunsen, Z. Huang, and S. Misailovic, "Testing probabilistic programming systems," in *FSE*, 2018.

[38] 2017, https://docs.conda.io.

[39] 2020, https://docs.pytest.org/en/stable.

[40] "Scipy ks-test," 2022, https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.kstest.html.

[41] "Allennlp," 2022, https://github.com/allenai/allennlp.

[42] "Captum," 2022, https://github.com/pytorch/captum.

[43] "Coax," 2022, https://github.com/microsoft/coax.

[44] "Deepchem," 2022, https://github.com/deepchem/deepchem.

[45] "Gensim," 2022, https://github.com/RaRe-Technologies/gensim.

[46] "Gpflow," 2022, https://github.com/GPflow/GPflow.

[47] "Gpytorch," 2022, https://github.com/cornellius-gp/gpytorch.

[48] "Kornia," 2022, https://github.com/kornia/kornia.

[49] "learn2learn," 2022, https://github.com/learnables/learn2learn.

[50] "Ml-agents," 2022, https://github.com/Unity-Technologies/ml-agents.

[51] "numpyro," 2022, https://github.com/pyro-ppl/numpyro.

[52] "Parlai," 2022, https://github.com/facebookresearch/ParlAI.

[53] "pgmpy," 2022, https://github.com/pgmpy/pgmpy.

[54] "refnx," 2022, https://github.com/refnx/refnx.

[55] "stellargraph," 2022, https://github.com/stellargraph/stellargraph.

[56] "tfcausalimpact," 2022, https://github.com/WillianFuks/tfcausalimpact.

[57] "trax," 2022, https://github.com/google/trax.

[58] "umap," 2022, https://github.com/lmcinnes/umap.

[59] "zfit," 2022, https://github.com/zfit/zfit.

[60] "Amazon ec2 on-demand pricing," 2022, https://aws.amazon.com/ec2/pricing/on-demand/.

[61] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine learning testing: Survey, landscapes and horizons," *TSE*, 2020.

[62] M. Harman and P. O'Hearn, "From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis," in *SCAM*, 2018.

[63] A. Romano, Z. Song, S. Grandhi, W. Yang, and W. Wang, "An empirical analysis of ui-based flaky tests," in *ICSE*, 2021.

[64] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta, "Root causing flaky tests in a large-scale industrial setting," in *ISSTA*, 2019.

[65] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "DeFlaker: Automatically detecting flaky tests," in *ICSE*, 2018.

[66] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "iDFlakies: A framework for detecting and partially classifying flaky tests," in *ICST*, 2019.

[67] A. Shi, A. Gyori, O. Legunsen, and D. Marinov, "Detecting assumptions on deterministic implementations of non-deterministic specifications," in *ICST*, 2016.

[68] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov, "ifixflakies: A framework for automatically fixing order-dependent flaky tests," in *FSE*, 2019.

[69] 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.

[70] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE transactions on software engineering*, 2010.

[71] O. L. Vera-Pérez, M. Monperrus, and B. Baudry, "Descartes: a pitest engine to detect pseudo-tested methods: tool demonstration," in *ASE*, 2018.

[72] F. Hariri, A. Shi, O. Legunsen, M. Gligoric, S. Khurshid, and S. Misailovic, "Approximate transformations as mutation operators," in *ICST*, 2018.

[73] L. Zhang, D. Marinov, and S. Khurshid, "Faster mutation testing inspired by test prioritization and reduction," in *ISSTA*, 2013.

[74] M. Gligoric, L. Zhang, C. Pereira, and G. Pokam, "Selective mutation testing for concurrent code," in *ISSTA*, 2013.

[75] L. Zhang, M. Gligoric, D. Marinov, and S. Khurshid, "Operator-based and random mutant selection: Better together," in *ASE*, 2013.

[76] B. J. Grün, D. Schuler, and A. Zeller, "The impact of equivalent mutants," in *2009 International Conference on Software Testing, Verification, and Validation Workshops*, 2009.

[77] M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon, "Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 2015.

[78] A. Shi, J. Bell, and D. Marinov, "Mitigating the effects of flaky tests on mutation testing," in *ISSTA*, 2019.

[79] T. Y. Chen, S. C. Cheung, and S. M. Yiu, "Metamorphic testing: a new approach for generating next test cases," CS Department, Hong Kong University of Science and Technology, Tech. Rep., 1998.

[80] Y. Tian, K. Pei, S. Jana, and B. Ray, "Deeptest: Automated testing of deep-neural-network-driven autonomous cars," in *Proceedings of the 40th international conference on software engineering*, 2018, pp. 303–314.

[81] Z. Q. Zhou and L. Sun, "Metamorphic testing of driverless cars," *Communications of the ACM*, vol. 62, no. 3, pp. 61–67, 2019.

[82] M. Zhang, Y. Zhang, L. Zhang, C. Liu, and S. Khurshid, "Deeproad: Gan-based metamorphic testing and input validation framework for autonomous driving systems," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 132–142.

[83] Z. Q. Zhou, S. Xiang, and T. Y. Chen, "Metamorphic testing for software quality assessment: A study of search engines," *IEEE Transactions on Software Engineering*, vol. 42, no. 3, pp. 264–284, 2016.

[84] T. Y. Chen, F.-C. Kuo, W. Ma, W. Susilo, D. Towey, J. Voas, and Z. Q. Zhou, "Metamorphic testing for cybersecurity," *Computer*, vol. 49, no. 6, pp. 48–55, 2016.

[85] P. He, C. Meister, and Z. Su, "Structure-invariant testing for machine translation," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 961–973.

[86] Z. Sun, J. M. Zhang, M. Harman, M. Papadakis, and L. Zhang, "Automatic testing and improvement of machine translation," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 974–985.

[87] Y. Tian, S. Ma, M. Wen, Y. Liu, S.-C. Cheung, and X. Zhang, "To what extent do dnn-based image classification models make unreliable inferences?" *Empirical Softw. Engg.*, vol. 26, no. 5, sep 2021.

[88] N. D. Goodman, V. K. Mansinghka, D. Roy, K. Bonawitz, and J. B. Tenenbaum, "Church: a language for generative models," in *UAI*, 2008.

[89] B. Carpenter, A. Gelman, M. Hoffman, D. Lee, B. Goodrich, M. Betancourt, M. A. Brubaker, J. Guo, P. Li, A. Riddell *et al.*, "Stan: A probabilistic programming language," *JSTATSOFT*, vol. 20, no. 2, 2016.

[90] T. Gehr, S. Misailovic, and M. Vechev, "PSI: Exact symbolic inference for probabilistic programs," in *CAV*, 2016.

[91] Z. Huang, S. Dutta, and S. Misailovic, "Aqua: Automated quantized inference for probabilistic programs," in *ATVA*, 2021.

[92] 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.

[93] A. Dwarakanath, M. Ahuja, S. Sikand, R. M. Rao, R. J. C. Bose, N. Dubash, and S. Podder, "Identifying implementation bugs in machine learning based image classifiers using metamorphic testing," in *ISSTA*, 2018.

[94] Y. Zhang, L. Ren, L. Chen, Y. Xiong, S.-C. Cheung, and T. Xie, "Detecting numerical bugs in neural network architectures," in *FSE*, 2020.

[95] 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.

[96] 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.

[97] A. Wei, Y. Deng, C. Yang, and L. Zhang, "Free lunch for testing: Fuzzing deep-learning libraries from open source," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022, pp. 995–1007.

[98] J. Liu, Y. Wei, S. Yang, Y. Deng, and L. Zhang, "Coverage-guided tensor compiler fuzzing with joint ir-pass mutation," *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA1, 2022.

[99] Y. Deng, C. Yang, A. Wei, and L. Zhang, "Fuzzing deep-learning libraries via automated relational api inference," in *FSE*, 2022.

[100] C. Yang, Y. Deng, J. Yao, Y. Tu, H. Li, and L. Zhang, "Fuzzing automatic differentiation in deep-learning libraries," in *ICSE*, 2023.

[101] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Fuzzing deep-learning libraries via large language models," *arXiv preprint arXiv:2212.14834*, 2022.

[102] J. Liu, J. Lin, F. Ruffy, C. Tan, J. Li, A. Panda, and L. Zhang, "Nnsmith: Generating diverse and valid test cases for deep learning compilers," in *ASPLOS*, 2023.

[103] S. Dutta, W. Zhang, Z. Huang, and S. Misailovic, "Storm: program reduction for testing and debugging probabilistic programming systems," in *FSE*, 2019.

[104] 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 *FSE*, 2018.

[105] K. Joshi, V. Fernando, and S. Misailovic, "Statistical algorithmic profiling for randomized approximate programs," in *ICSE*, 2019.