Saikat Dutta Research Statement

My research interests are in **Software Engineering**, with a focus on **Software Testing** and its application in making **Machine Learning**-based systems more reliable. Machine Learning (ML) is rapidly revolutionizing the development of many modern-day systems. However, ensuring the reliability of ML-based systems is challenging due to 1) the presence of non-determinism in internal (e.g., stochastic algorithms) and external (e.g., execution environment) components and 2) the lack of accuracy specifications. Traditional software testing techniques, while widely used to improve software reliability, cannot tackle these challenges since they mostly assume deterministic behavior and lack domain knowledge.

The goal of my research is to develop novel testing techniques and tools to make ML-based systems more reliable. I achieve this goal by 1) automatically detecting bugs and debugging failures in ML libraries that drive such systems [FSE18, FSE19, FASE22, FALCON*], and 2) detecting bugs more effectively and efficiently by improving the quality of developer-written tests in ML libraries [ISSTA20, FSE21, ISSTA21, ICST22, ICSE23]. My research exploits the fundamental principle that we can systematically reason about non-determinism and accuracy using rigorous statistical and probabilistic reasoning. I develop novel static and dynamic analyses for testing ML-based systems that build on this principle. I implement these analyses into scalable tools that allow developers to efficiently navigate the inherent trade-off space between the quality and efficiency of their tests.

To fulfill my research goal, I developed **the first generation of testing tools for ML Libraries**. My tools have uncovered bugs and improved the quality of tests in numerous popular open-source ML libraries (e.g., PyTorch and TensorFlow), including those used in large-scale software ecosystems, such as DeepMind, Google, Meta, Microsoft, and Uber. My significant contributions include:

1. The *first* approaches for detecting bugs [FSE18] and debugging failures [FSE19, FASE22] in Probabilistic Programming Systems – a form of Bayesian Machine Learning. My approaches detected more than **50 previously unknown bugs in Probabilistic Programming Systems** and enabled faster debugging of failures.

2. The *first* approaches for detecting [ISSTA20] and fixing [FSE21] *flaky tests* – tests that randomly fail for the same code version – caused by the randomness of stochastic training/inference algorithms used in ML libraries, and the *first* technique [ISSTA21] to significantly speed-up tests in ML libraries. **My tools have already improved the effectiveness and efficiency of over 200 tests in over 60 open-source ML libraries.**

I also received the **Facebook PhD Fellowship** and the **3M Foundation Fellowship** for my research.

My short-term goal is to develop high-quality testing and debugging techniques for ML-based systems that can be seamlessly integrated into daily developer workflow and enable developers to write more effective and efficient tests. My long-term goal is to redefine the testing of systems with non-deterministic computations, like ML-based systems, by identifying unique domain-specific insights and exploiting them using foundational software engineering, program analysis, statistics, and machine learning techniques. I plan to extend my approaches to testing in other domains characterized by non-deterministic computations, including deep probabilistic programming and hybrid systems driven by ML-based components (e.g., autonomous cars and robots). My experience and insights from handling non-determinism can aid in overcoming the challenges in these emerging domains.

1 Testing Probabilistic Programming Systems

Probabilistic Programming (PP) is an emerging programming paradigm that allows users to easily express various statistical models as simple high-level programs while hiding the complex details of inference. A PP system implements a probabilistic programming language and one or more inference algorithms, such as Markov Chain Monte Carlo (MCMC) and Variational Inference. PP has shown great promise in real-world applications (e.g., financial forecasting, social network recommendations, and COVID-19 modeling) and attracted interest from industry (e.g., Google, Meta, Microsoft, and Uber) and scientific communities. However, PP is a young research area, and there is a need for tools that can improve the reliability and usability of PP systems to increase PP's adoption. Hence, **I developed the first testing and debugging tools for PP systems.** My work in this direction focuses on tackling three main challenges: 1) automatically detecting bugs in PP systems while accounting for the approximate

*: under submission

and probabilistic nature of inference, 2) efficiently reducing fault-revealing probabilistic programs (inputs for PP system) to make it easier for developers to debug failures in PP systems, and 3) helping users efficiently debug programming errors in probabilistic programs.

Detecting Bugs in Probabilistic Programming Systems. A PP system is complex, typically consisting of multiple components such as language frontends, a compiler, inference algorithms, and numerical libraries. Additionally, the approximate and probabilistic nature of the inference algorithms makes it hard to reason about correctness in such systems. To tackle these challenges, **I developed ProbFuzz [FSE18] – the first technique that laid the foundation for systematically testing PP systems.** ProbFuzz uses program analysis and domain-specific information about probability distributions to generate semantically valid probabilistic programs (inputs). Such programs can exercise complex logic in PP systems' various components, and expose hard-to-detect bugs. Because inference algorithms do not come with accuracy specifications, ProbFuzz compares the outputs (posterior distributions of model parameters) from multiple PP systems using statistical tests to reason about correctness. If the output of a PP system differs significantly from others, there is a potential accuracy or numerical bug. **Using ProbFuzz, I detected and fixed over 50 bugs in three popular PP systems (Stan, Edward, and Pyro) and their underlying frameworks (PyTorch and TensorFlow).**

Program Reduction for Debugging Probabilistic Programming Systems. PP systems make it easier to write probabilistic programs, but failures in PP systems can be notoriously hard to debug. A developer has to manually analyze the complex interactions between the probabilistic program, its data inputs, and the inference algorithm implemented by the PP system, which is non-trivial. Program Reduction techniques, commonly used in compiler testing, reduce the size of bug-revealing programs so that the reduced program still reproduces the same bug. This reduction makes debugging easier. However, existing program reduction techniques are inefficient when applied to PP: they can produce illegal or semantically invalid programs because they only consider syntactic information. **I developed Storm [FSE19] – the first automated approach for efficiently reducing bug-revealing probabilistic programs while triggering the same failure in the PP system.** Unlike existing approaches, Storm leverages various program analyses and probabilistic reasoning to reduce bug-revealing probabilistic programs efficiently. I evaluated Storm on 47 bug-revealing probabilistic programs. The reduced programs had up to 58% less code and 96% less data and required 99% fewer inference iterations to reproduce the same bugs. The reduced programs run up to 126x faster, significantly speeding up debugging and saving development time.

Debugging Convergence Problems in Probabilistic Programs. Users of probabilistic programs often make programming errors (e.g., using improper distributions) that lead to the non-convergence of inference algorithms. Convergence is necessary for successful inference because non-convergence often causes inaccurate (or wrong) results. The convergence of inference algorithms such as MCMC, although guaranteed *in the limit*, is hard to predict ahead of time (i.e., without executing the program). However, predicting convergence requires a deep understanding of the complicated interaction between the probabilistic program and the inference algorithm. Users often do not have such expertise. To address this problem, **I developed SixthSense [FASE22] – the first data-driven approach to predict whether a probabilistic program will converge in a limited number of steps.** SixthSense encodes the structure of probabilistic programs into a vector representation and trains an ML classifier to learn program features that lead to convergence or non-convergence. SixthSense obtained an average accuracy of over 78% in predicting convergence for a broad class of probabilistic programs using only static program features and 83% when combining dynamic features.

2 Testing Machine Learning Libraries

Testing the correctness of Machine Learning libraries – including those that support Deep Learning, Reinforcement Learning, Probabilistic Programming, and Classical ML – is challenging due to the use of stochastic algorithms and the lack of accuracy specifications. To test such libraries, developers typically write *regression tests* that are run after every code change to check that the change does not break existing functionality. However, when writing regression tests, developers often do not adequately account for the randomness of the algorithm under test and resort to non-systematic techniques, which reduces the reliability, efficiency, and effectiveness of the tests. To address these problems, **I have developed the first set of systematic techniques, grounded in statistical reasoning and probability theory, that improve the quality of regression tests in Machine Learning libraries.** My research focuses on three aspects: 1) improving test reliability by detecting and fixing flaky tests, 2) making regression tests cost-efficient, and 3) improving the fault-detection effectiveness of tests.

Detecting and Fixing Flaky Tests in Machine Learning Libraries. In ML libraries, many regression tests for code involving stochastic computations are *flaky* – they randomly pass or fail on the same code version. Test flakiness is undesirable since it makes the test unreliable and wastes developer time investigating spurious failures. My empirical study [ISSTA20] revelated that the majority of flaky tests in this domain (60%) are tests that involve stochastic computations. To tackle the problem of flaky tests, **I developed FLASH [ISSTA20] – the first technique for automatically detecting tests that are flaky due to stochastic computations.** The key idea behind FLASH is to use statistical convergence testing to model the distribution of values of the variable in the test assertion and identify potential flaky tests. FLASH detected 11 new flaky tests in 7 out of 20 ML libraries that we evaluated. Common wisdom might suggest mitigating such flakiness by *fixing the seed* in random number generators. However, my work has shown that this approach can often hide subtle bugs [ICST22]. As a more systematic approach, **I developed the first technique to fix such flaky tests, FLEX [FSE21]**, which adapts Extreme Value Theory from statistics to reliably model the tail distribution of the variable in the test assertion and estimate a new assertion bound that minimizes flakiness to a desirable level. Using FLEX, I fixed 28 flaky tests across 17 ML libraries.

Making Regression Tests in Machine Learning Libraries Cost-Efficient. To avoid flakiness, developers of ML libraries are often overtly conservative in their tests – they execute the stochastic algorithm under test for more iterations than necessary to ensure that the outputs are *in the expected ranges*. Naturally, such tests are slow. To tackle this challenge, **I developed TERA [ISSTA21] – the first automated technique for reducing the execution cost of regression tests in ML libraries without making them flaky.** My central insight in TERA is that we can enable faster and highly reliable test execution by relaxing the desired passing probability of the test. TERA formulates this problem of trade-off space exploration between test execution time and its passing probability as an optimization problem over the space of algorithm hyper-parameters. TERA then leverages Bayesian optimization to find the optimal hyper-parameter values that minimize test execution time while ensuring that the passing probability is above a developer-specified threshold. Overall, I obtained a speedup of 2.23x across 160 tests in 15 ML libraries using TERA.

Improving Fault Detection Effectiveness of Regression Tests. Because developers want to avoid flaky failures, they often set loose assertion bounds in tests. This practice reduces the fault-detection effectiveness of the test. **I developed FASER [ICSE23] – the first systematic technique that maximizes the fault-detection effectiveness of the test without making it unacceptably flaky.** FASER frames this problem as an optimization problem with two competing objectives: 1) maximizing fault-detection effectiveness and 2) minimizing flakiness by varying the assertion bound. Since it is not feasible to specify these objectives analytically, FASER uses mutation testing to estimate test effectiveness and various concentration inequalities from probability theory to estimate test flakiness. Using FASER, I improved the effectiveness of 23 tests across 12 ML libraries by more than 15% while limiting flakiness.

3 Other Research Contributions

In addition to my primary research, I made several contributions in probabilistic programming and program analysis.

Symbolic Inference for Probabilistic Programs. I contributed to developing **AQUA** [ATVA21] – a novel symbolic inference algorithm for probabilistic programs. The central idea in AQUA is to approximate and estimate the posterior distribution using a novel quantized symbolic representation. The main benefit of AQUA is that it can provide more accurate and faster results than approximate inference algorithms like MCMC and supports programs that are out of reach of exact inference algorithms. AQUA provides precise results on a diverse set of 24 benchmark probabilistic programs.

Improving Security of Javascript Applications by Inferring Taint Specifications. Taint analysis is a powerful static analysis technique for exposing violations of security policies. However, taint analysis requires taint specifications – program points controlling sensitive data flow. Such specifications are hard to infer automatically. I developed **InspectJS** [ICSE-SEIP22] – a novel Machine Learning-based approach to infer taint specifications for JavaScript automatically. InspectJS found several new taint specifications in real-world JavaScript projects that were missing in GitHub's CodeQL analysis framework.

4 Future Work

As the influence and impact of Machine Learning-based systems continue to gain momentum in society, I believe that the principal themes of my research will only grow in importance and applicability. The recently introduced White House's AI Bill of Rights underscores the importance of "extensive testing" in developing Machine Learning-based systems to avoid undue harm to society. Further, there is growing interest in this area both from industry and federal funding programs (e.g., NSF DASS and NSF SHF) acknowledging the importance of testing ML-based systems.

In the near term, I will focus on tackling the challenges in testing the ever-expanding ML toolset, including tools such as deep learning compilers and ML libraries. In the long term, my goal is to adopt an interdisciplinary approach by synergistically combining cutting-edge techniques from software engineering, programming languages, statistics, and machine learning to solve the challenges in testing emerging domains such as deep probabilistic programming and ML-driven hybrid systems such as robots and autonomous cars. These domains exhibit the common theme of non-determinism in various forms, which I am uniquely positioned to tackle due to my research experience.

Effective Testing and Debugging of Deep Learning Compilers. Deep Learning (DL) compilers automate the compilation and optimization of DL models for heterogenous hardware such as high-end GPUs, FPGAs, and edge devices. DL compilers, like any other software, are prone to bugs. Such bugs can often lead to silent failures in the form of inaccurate results or poor performance, which makes testing important. First, I will develop a comprehensive and cost-efficient testing approach for DL compilers to detect such hard-to-find bugs. My early efforts in this dimension have shown promising results [FALCON*]. Second, I will develop automated bug localization and repair techniques to debug such failures.

Rethinking Regression Testing for ML libraries. While writing regression tests for ML libraries, developers manually choose various configurations (like algorithm hyper-parameters and assertion bounds), often leading to problems like flakiness or reduced fault-detection effectiveness. I envision a test generation approach that, given a test specification, can automatically make these choices while jointly optimizing for one or more goals (such as maximizing fault-detection effectiveness or coverage). I will combine insights from my work on regression tests in ML libraries with novel approaches to determine optimal test configurations from the large combinatorial space of choices.

Testing Deep Probabilistic Programs. Deep probabilistic programming is a promising programming system that enables composing deep learning models with probabilistic programs. Such programming systems allow users to effectively combine the benefits of both domains, such as scalability to billions of parameters, AI accelerators, interpretability, and reasoning about uncertainty. While the domain is still in its infancy, it will require novel testing approaches to enable robust development of the programming platforms. My experience with probabilistic programming and deep learning has helped me gain unique insights to develop novel solutions for this domain. I will investigate this domain's unique challenges and programming errors and develop automated testing approaches.

Improving Reliability of Hybrid Systems with Machine Learning Components. With the proliferation of powerful Machine Learning models, various real-world systems in emerging domains such as autonomous driving, AR/VR, and robotics have started integrating ML-based components. Testing such hybrid systems present challenges that are beyond current testing approaches. First, a testing approach should account for how the ML-based components interact with other ML- and non-ML based components, especially in the presence of non-determinism. Second, the business logic of the ML-based components is often learned from data and hence lacks interpretability. Therefore, we require a test oracle that can reason about the correctness of the component in the context of the entire system. To tackle these challenges, I will develop a testing approach that 1) reasons about component- and system-level non-determinism by modeling their behavior using statistical methods, 2) captures component interactions using probabilistic models, 3) generates test inputs that trigger diverse component behaviors and their interactions, and 4) checks for correctness by adapting system-level specifications to the hybrid setting.

References

[FSE18] **Saikat Dutta**, Owolabi Legunsen, Zixin Huang, and Sasa Misailovic. "Testing Probabilistic Programming Systems". *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 2018.

[FSE19] **Saikat Dutta**, Wenxian Zhang, Zixin Huang, and Sasa Misailovic. "Storm: Program Reduction for Testing and Debugging Probabilistic Programming Systems". *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 2019.

[FASE22] **Saikat Dutta**, Zixin Huang, and Sasa Misailovic. "Sixthsense: Debugging Convergence Problems in Probabilistic Programs via Program Representation Learning". *International Conference on Fundamental Approaches to Software Engineering (FASE)*. 2022.

[FALCON*] **Saikat Dutta** and Daniel Kroening. "Can Canary Datasets be Used Effectively for Fuzzing Deep Learning Compilers?" *Under submission*. 2022.

[ISSTA20] **Saikat Dutta**, August Shi, Rutvik Choudhary, Zhekun Zhang, Aryaman Jain, and Sasa Misailovic. "Detecting Flaky Tests in Probabilistic and Machine Learning Applications". *29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 2020.

[FSE21] **Saikat Dutta**, August Shi, and Sasa Misailovic. "Flex: Fixing Flaky Tests in Machine Learning Projects by Updating Assertion Bounds". *29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 2021.

[ISSTA21] **Saikat Dutta**, Jeeva Selvam, Aryaman Jain, and Sasa Misailovic. "Tera: Optimizing Stochastic Regression Tests in Machine Learning Projects". *30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 2021.

[ICST22] **Saikat Dutta**, Anshul Arunachalam, and Sasa Misailovic. "To Seed or Not to Seed? An Empirical Analysis of Usage of Seeds for Testing in Machine Learning Projects". *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 2022.

[ICSE23] Steven Xia, **Saikat Dutta**, Darko Marinov, Sasa Misailovic, and Lingming Zhang. "FASER: Balancing Effectiveness and Flakiness of Non-Deterministic Machine Learning Tests". *IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2023.

[ATVA21] Zixin Huang, **Saikat Dutta**, and Sasa Misailovic. "Aqua: Automated Quantized Inference for Probabilistic Programs". *International Symposium on Automated Technology for Verification and Analysis (ATVA)*. 2021.

[ICSE-SEIP22] **Saikat Dutta**, Diego Garbervetsky, Shuvendu K Lahiri, and Max Schäfer. "InspectJS: Leveraging Code Similarity and User-Feedback for Effective Taint Specification Inference for JavaScript". *IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 2022.