August Shi

Research Statement

My research interests are in Software Engineering, with a focus on Software Testing. Software has become a crucial part of our daily lives, and *software testing* is widely used to check the quality of software. Developers frequently make changes to improve software and then perform *regression testing* by rerunning their tests to check that changes do not break existing functionality. Despite its widespread use, regression testing still has two key challenges: (1) test suites have *flaky tests* that nondeterministically pass and fail independently of the developers' changes, misleading developers that their changes broke existing functionality, and (2) regression testing takes a long time to run, wasting developer time and compute resources. These challenges are faced by both open- and closed-source communities, including companies such as Facebook, Google, and Microsoft.

My research so far has focused on tackling the two key challenges of regression testing: flaky tests and long regression testing time. More specifically, I developed techniques and tools for (1) *making regression testing more reliable* by detecting flaky tests and mitigating their effects [5, 6, 12, 19, 21, 24], (2) *optimizing test suites for faster run times* through better test placement and through removing redundant tests [1, 20, 22, 25, 26], and (3) *speeding up regression testing* by rerunning only the tests that can be affected by code changes [13, 14, 23, 27, 31]. I also *evaluated test-suite quality* through mutation testing [3, 4, 7, 8, 9, 10].

As long as people write software, there will be bugs, and software testing will remain highly relevant. In the future, I plan to develop testing techniques for emerging software domains where testing is still not mature enough, such as machine learning, probabilistic programming, or Internet of Things. These domains are inherently nondeterministic, leading to challenges in testing that I look forward to overcoming. I also plan to improve testing of code for computational science [2, 11, 18, 28], where reproducibility is currently a big challenge that can be alleviated by software testing techniques.

Detecting and Fixing Flaky Tests

Ideally, a test that fails during regression testing indicates that the developer's changes broke existing functionality, and the developers should fix the changes before continuing development. However, regression testing becomes unreliable in the presence of *flaky tests*, which can nondeterministically pass and fail when run on the same, unchanged code. A test failure no longer indicates that the changes broke existing functionality. Developers then either waste time trying to fix bugs unrelated to their changes, or they start ignoring test failures, potentially releasing software with bugs that they could have fixed. Google in one period reported 73K of 1.6M test failures being due to flaky tests [15]. Furthermore, Google reported using 2-16% of their compute resources rerunning flaky tests [17]. I am the first to develop techniques for automatically fixing flaky tests. My work on flaky tests falls into two categories: (1) mitigating the effects of existing flaky tests in the test suite, including techniques for fixing flaky tests, and (2) proactively detecting flaky tests, ideally as soon as they are written. Next, I describe one project in each category.

Fixing Order-Dependent Flaky Tests. Order-dependent flaky tests pass or fail based on the order in which they are run. The order in which tests are run can vary from one run to another. Conceptually, an order-dependent flaky test fails in one test order but not in others, because it starts in an incorrect system state (e.g., in-memory heap, file system), in a failing order; for example, some other test that ran before it overwrote the initial value to a global variable and did not reset the global variable after running, and the order-dependent flaky test assumes the initial value. Order-dependent flaky tests can be fixed by inserting logic that resets that state before the flaky test runs. While investigating order-dependent flaky tests, I discovered that logic for resetting state is often within the test suite itself! Based on this insight, I developed *iFixFlakies* [24], a framework for automatically fixing order-dependent flaky tests. iFixFlakies first finds "fixer" tests that contain the resetting logic. iFixFlakies then minimizes the code in the fixer tests to find a minimal patch that fixes the order-dependent flaky test. On 110 order-dependent flaky tests from a dataset of flaky tests built in my prior work [12], iFixFlakies automatically fixed 101 flaky tests. I submitted patches for these tests; developers accepted patches for 64 flaky tests, while other patches are pending review.

Proactively Detecting Order-Dependent Flaky Tests. A complementary strategy to iFixFlakies is to detect and address order-dependent flaky tests *proactively*, before they can even fail. Such a flaky test fails due to a "polluter" test that pollutes the state. If that polluter test is introduced into the test suite before

an order-dependent flaky test, developers should be alerted so they can address the polluter test as soon as it is written. I developed *PolDet* [6], a technique for proactively detecting polluter tests. PolDet compares the state (including the in-memory heap and parts of the file system) before and after each test run and reports when the states differ; such a test is a polluter test. I ran PolDet on 26 open-source projects from GitHub and found 194 polluter tests, showing the prevalence of polluter tests and the effectiveness of PolDet.

Optimizing Test Suites

Besides flaky tests, another problem with regression testing is that it is slow, because many tests run after every change and changes occur very frequently. To speed up regression testing, I developed techniques to optimize the test suite itself so that running it in the future would take less time.

Optimizing Test Placement. Developers naturally group their code and tests into modules, especially as the codebases grow very large. In companies that have large codebases, such as Facebook, Google, or Microsoft, the developers take advantage of this modular structure and, along with the developer-specified dependencies among modules, run only the tests within modules that depend on any changed module; only tests within these modules can have different outcomes. However, over time, developers may suboptimally place tests in modules such that these tests are run even when their outcomes are not affected by the code changes. Developers often make suboptimal test placements due to lack of knowledge of the overall project structure. I proposed TestOptimizer [25] for optimizing test placements in modules. TestOptimizer first uses dynamic analysis to determine the precise modules each test depends on. Using this precise dependency information and historical information on the frequency of changed modules, TestOptimizer proposes moving tests into different modules to reduce the number of tests run, while also moving the fewest number of tests. I developed TestOptimizer at Microsoft, leading to an estimated average of over 21 million fewer test runs across five projects during a three-month period. Developers accepted our suggested test movements, where one team even asked for periodic TestOptimizer reports. This work on TestOptimizer won an **ACM SIGSOFT Distinguished Paper Award** at ICSE 2017.

Test-Suite Reduction. Another way to speed up regression testing is to make the test suite smaller, running fewer tests in the future, ideally without losing bug-detection capability. *Test-suite reduction* removes redundant tests from the test suite based on test-suite quality metrics, such as code coverage. While researchers have studied test-suite reduction for decades, I noticed that past evaluations all had a big flaw. Test-suite reduction creates reduced test suites to be run on future versions of software, yet prior work flawly evaluated test-suite reduction on the *single* version of software, where the reduced test suite was computed. I also questioned whether proposed metrics are properly capturing the reduced test suite's effectiveness at detecting future bugs. I evaluated reduced test suites using real failures found from historical test runs of open-source projects [22]. A reduced test suite would ideally include to run any failing test in a future version of the code. My evaluation showed that prior metrics do not correlate well with the actual test failures developers would observe. The reduced test suites also performed a lot worse than suggested by those prior metrics, missing to detect failures in up to 52% of future versions. I proposed a new metric for evaluating test-suite reduction that is more faithful with respect to detecting future bugs.

Regression Test Selection

Regression test selection speeds up regression testing by selecting to rerun after every change only a subset of the tests that depend on the changed code. Prior work proposed that the best way to perform regression test selection is to use dynamic analysis at a coarser granularity level of classes, meaning tracking what classes each test depends on, rather than at the finer granularity level of method- or statement-level test dependencies. My work on regression test selection challenges these established findings by using a *static analysis* for selecting tests and by tracking changes at *even coarser levels* of granularity.

Static regression test selection. I proposed a technique called STARTS [13,14], which uses static analysis to perform regression test selection. STARTS computes a dependency graph among classes in the code, and, given the changed classes between versions, it then selects to rerun only the tests that transitively depend on the changed classes in the dependency graph. Using static analysis to collect dependencies avoids problems with dynamic analysis where broken and incomplete test runs can lead to incorrectly computed dependencies. On the other hand, using static analysis can select more tests than necessary, leading to longer regression testing time. However, my experiments showed that STARTS's end-to-end time is comparable to state-of-

the-art dynamic regression test selection, despite selecting more tests, due to its faster analysis time [13]. I also found that STARTS can miss dynamic dependencies that are due to reflection, so I further developed a range of techniques to make STARTS reflection aware [23].

Comparing module- and class-level selection. Although prior work found that tracking test dependencies at the class level can lead to more effective regression test selection than finer levels of granularity, I investigated tracking dependencies at the even coarser module-level granularity [27], which is used in large companies like Facebook, Google, and Microsoft. Tracking dependencies at the module level can speed up analysis time at the cost of selecting more tests to rerun, but the overall regression testing time may still be comparable. I compared a module-level test selection technique with a class-level test selection technique. I found that the techniques have very similar end-to-end times in a cloud-based continuous integration environment, which is now popular among developers. Furthermore, when I examined the test failures from the test runs, I found that regression test selection did not miss to select any failing test due to a real bug in the changes. Rather, the test failures are almost entirely flaky test failures. These results demonstrate that regression test selection at the module level should be investigated more in research.

Evaluating Test Quality

Researchers have for long used mutation testing as a means of evaluating test-suite quality in terms of bug-finding capability. I developed techniques that further improve existing mutation testing.

Approximate computing and mutation testing. Mutation testing first uses operators that make syntactic changes to the code under test, creating variants of the code called *mutants*. When the test suite does not fail when run on a mutant, the test suite is weak and should be improved. Generating more diverse mutants, such as through different operators, can provide more insights to developers on how to improve the test suite. Taking cues from the emerging domain of approximate computing, I adapted several approximate transformations as new mutation testing operators [10], and I found them effective at generating mutants that help developers improve their test suites in different ways than traditional mutants. I submitted 11 pull requests to fix the developers' tests, with seven of them already accepted.

Mutation testing at different levels. Most mutation testing tools generate mutants by directly transforming the source code, but they could also transform the intermediate representation (IR), e.g., LLVM bitcode. A tool for mutating at the IR level can generate mutants for any programming language that compiles to the same IR. I developed a tool, SRCIROR [7], for mutating both C source code and LLVM bitcode, and my evaluation showed that the two levels result in similar mutation testing scores, but with fewer mutants generated at the C source code level [9]. Furthermore, I investigated the effect of different optimization levels at the IR level and found it best to use the highest optimization level, `O3`. The final mutation testing results were roughly the same across optimization levels, with `O3` running faster [8].

Future Work

My broad research goal is to help developers be more productive in whatever software they develop. Although we have still much to do concerning testing and development of traditional software domains, more challenges await us testing in emerging software domains. These domains, which include machine learning, probabilistic programming, and Internet of Things, are inherently nondeterministic, which is a major reason why the software testing research community has yet to develop satisfactory testing for these domains.

In the short term, I will work on the plethora of challenges that still plague regression testing of conventional software by applying better, data-driven solutions such as using machine-learning techniques for detecting/fixing flaky tests or optimizing test suites; such techniques are finding application in other software engineering tasks [16, 29, 30]. In the longer term, I plan to tackle the challenges in testing emerging domains or systems that are neither well-understood nor well-tested. For example, given the challenges in reproducing results obtained from computational science papers [2, 18, 28], scientists inherently have a problem with nondeterminism and flaky tests. My research puts me in a uniquely strong position to help them.

Fixing Flaky Tests. Conceptually, a flaky test passes and fails due to factors that the test does not control. For example, order-dependent flaky tests fail because they rely on state polluted by tests that run before. My work on iFixFlakies demonstrated the possibility to create simple yet effective fixes for order-dependent flaky tests. These tests, while relatively common, are but one of 10 types of flaky tests identified in a prior

study [15]. I believe that a similar approach as iFixFlakies is also possible for many other types of flaky tests. By understanding the root causes for each type of flaky test, I plan to develop specific techniques for fixing each one. The basis of each technique will be to localize the factor that the flaky test does not control and have the test control it. In addition, I plan to use natural-language processing and data-mining approaches to more efficiently find fixes for flaky tests from both the project's current code base and its history.

Testing in Emerging Domains. An example of an emerging domain of software that is becoming mainstream yet still not well tested is machine learning frameworks. Some of such frameworks rely on underlying approximate-inference based systems, such as probabilistic-programming systems, to provide automation for common inference tasks. Companies such as Facebook, Google, Microsoft, and Uber are developing their own approximate-inference based systems to help with their deep learning tasks. I have experience utilizing techniques from the approximate-inference domain to software testing problems, specifically in adapting approximate computing techniques to help with mutation testing [3, 10], and I am interested to use software testing techniques to improve the quality of approximate-inference based systems. These systems are challenging to test due to their inherent nondeterministic nature. For example, probabilistic-programming systems automate inference tasks through random sampling of different distributions. As such, *all* tests for these systems can be flaky. While tests can be made completely deterministic by setting concrete seeds, such tests would not always check the probabilistic properties that such systems provide. I plan to develop techniques that automatically generate tests for the randomness inherent in such systems while still providing reliable testing outcomes that can indicate bugs in the system without sacrificing the quality of the inference.

Testing Techniques for Computational Science. Computational science is currently going through a period that some call a "reproducibility crisis" [18]. I believe that software testing can go a long way in helping overcome these challenges. Currently, scientists do not employ testing techniques for the software projects that drive their experiments. For example, the problem of flaky tests in software testing corresponds to the problem scientists have in obtaining the same results from the same code and the same inputs. From my previous work on reproducing computational physics results [11], I realized that even if scientists adopt proper software engineering practices, given the kind of experiments they run that model the world, nondeterminism would still cause problems with reproducing results. I envision extending the techniques I develop for mitigating flaky tests in conventional software to help with the reproducibility crisis in computational science. My goal in this domain is to help scientists get more deterministic and reliable computational results, and to make it easier for future researchers to reproduce those results. Furthermore, I envision developing techniques that minimize tests for scientific code such that passing tests would indicate the code can likely reproduce the paper results without requiring to run the full experiments, which would be much more time consuming. With such tests, I can help scientists bring regression testing practices to their development so they can make improvements to their experiments and be confident that they build upon the results they obtained before.

References

[1] M. A. Alipour, A. Shi, R. Gopinath, D. Marinov, and A. Groce. Evaluating non-adequate test-case reduction. In *ASE*, pages 16–26, 2016.

[2] D. L. Donoho, A. Maleki, I. U. Rahman, M. Shahram, and V. Stodden. Reproducible research in computational harmonic analysis. *CiSE*, 11(1):8–18, 2008.

[3] M. Gligoric, S. Khurshid, S. Misailovic, and A. Shi. Mutation testing meets approximate computing. In *ICSE NIER*, pages 3–6, 2017.

[4] A. Groce, J. Holmes, D. Marinov, A. Shi, and L. Zhang. An extensible, regular-expression-based tool for multi-language mutant generation. In *ICSE Demo*, pages 25–28, 2018.

[5] A. Gyori, B. Lambeth, A. Shi, O. Legunsen, and D. Marinov. NonDex: A tool for detecting and debugging wrong assumptions on Java API specifications. In *FSE Demo*, pages 993–997, 2016.

[6] A. Gyori, A. Shi, F. Hariri, and D. Marinov. Reliable testing: Detecting state-polluting tests to prevent test dependency. In *ISSTA*, pages 223–233, 2015.

[7] F. Hariri and A. Shi. SRCIROR: A toolset for mutation testing of C source code and LLVM intermediate representation. In *ASE Demo*, pages 860–863, 2018.

[8] F. Hariri, A. Shi, H. Converse, D. Marinov, and S. Khurshid. Evaluating the effects of compiler optimizations on mutation testing at the compiler IR level. In *ISSRE*, pages 105–115, 2016.
[9] F. Hariri, A. Shi, V. Fernando, S. Mahmood, and D. Marinov. Comparing mutation testing at the levels of source code and compiler intermediate representation. In *ICST*, pages 114–124, 2019.
[10] F. Hariri, A. Shi, O. Legunsen, M. Gligoric, S. Khurshid, and S. Misailovic. Approximate transformations as mutation operators. In *ICST*, pages 285–296, 2018.
[11] M. Krafczyk, A. Shi, A. Bhaskar, D. Marinov, and V. Stodden. Scientific tests and continuous integration strategies to enhance reproducibility in the scientific software context. In *P-RECS*, pages 23–28, 2019.
[12] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie. iDFlakies: A framework for detecting and partially classifying flaky tests. In *ICST*, pages 312–322, 2019.
[13] O. Legunsen, F. Hariri, A. Shi, Y. Lu, L. Zhang, and D. Marinov. An extensive study of static regression test selection in modern software evolution. In *FSE*, pages 583–594, 2016.
[14] O. Legunsen, A. Shi, and D. Marinov. STARTS: STAtic Regression Test Selection. In *ASE Demo*, pages 949–954, 2017.
[15] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. An empirical analysis of flaky tests. In *FSE*, pages 643–653, 2014.
[16] M. Machalica, A. Samylkin, M. Porth, and S. Chandra. Predictive test selection. In *ICSE SEIP*, pages 91–100, 2019.
[17] J. Micco. The state of continuous integration testing at Google, 2017.
[18] National Academies of Sciences, Engineering, and Medicine. *Reproducibility and Replicability in Science*. The National Academies Press, 2019.
[19] A. Shi, J. Bell, and D. Marinov. Mitigating the effects of flaky tests on mutation testing. In *ISSTA*, pages 112–122, 2019.
[20] A. Shi, A. Gyori, M. Gligoric, A. Zaytsev, and D. Marinov. Balancing trade-offs in test-suite reduction. In *FSE*, pages 246–256, 2014.
[21] A. Shi, A. Gyori, O. Legunsen, and D. Marinov. Detecting assumptions on deterministic implementations of non-deterministic specifications. In *ICST*, pages 80–90, 2016.
[22] A. Shi, A. Gyori, S. Mahmood, P. Zhao, and D. Marinov. Evaluating test-suite reduction in real software evolution. In *ISSTA*, pages 84–94, 2018.
[23] A. Shi, M. Hadzi-Tanovic, L. Zhang, D. Marinov, and O. Legunsen. Reflection-aware static regression test selection. In *OOPSLA*, pages 187:1–187:29, 2019.
[24] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov. iFixFlakies: A framework for automatically fixing order-dependent flaky tests. In *ESEC/FSE*, pages 545–555, 2019.
[25] A. Shi, S. Thummalapenta, S. Lahiri, N. Bjorner, and J. Czerwonka. Optimizing test placement for module-level regression testing. In *ICSE*, pages 689–699, 2017.
[26] A. Shi, T. Yung, A. Gyori, and D. Marinov. Comparing and combining test-suite reduction and regression test selection. In *ESEC/FSE*, pages 237–247, 2015.
[27] A. Shi, P. Zhao, and D. Marinov. Understanding and improving regression test selection in continuous integration. In *ISSRE*, pages 228–238, 2019.
[28] V. Stodden, M. Krafczyk, and A. Bhaskar. Enabling the verification of computational results: An empirical evaluation of computational reproducibility. In *P-RECS*, pages 3:1–3:5, 2018.
[29] Z. Wu, E. Johnson, W. Yang, O. Bastani, D. Song, J. Peng, and T. Xie. REINAM: Reinforcement learning for input-grammar inference. In *ESEC/FSE*, pages 488–498, 2019.
[30] J. Zhang, Z. Wang, L. Zhang, D. Hao, L. Zang, S. Cheng, and L. Zhang. Predictive mutation testing. In *ISSTA*, pages 342–353, 2016.
[31] C. Zhu, O. Legunsen, A. Shi, and M. Gligoric. A framework for checking regression test selection tools. In *ICSE*, pages 430–441, 2019.