"Continuous Integration" (CI) is now a standard software engineering practice, automating the execution of large test suites in the cloud. By leveraging on-demand computing utilities, CI can execute hundreds of tests in parallel, transforming what might have once been a three-day test suite into a one-hour test suite. Whereas these test suites (including complex integration and end-to-end tests) may have previously been run on a weekly or monthly basis, they now are run as frequently as on *every single change*. Engineers benefit from faster feedback, identifying bugs and performance regressions sooner and optimizing productivity, reducing development costs overall. But, CI is also a relatively new process, and it brings many challenges.

I design novel approaches to address the problems that open-source developers face when building software with CI, and validate those solutions using these open-source projects. I release my tools and datasets under open-source licenses and continue to maintain them. My dynamic taint tracking system for Java, PHOSPHOR [1] has been directly used and extended by at least six external research groups in nine publications [18]–[26]. My flaky test dataset, FLAKEFLAGGER [2] has been adopted by at least three external research groups to support follow-up research [27]–[29]. I have also integrated my findings into popular open-source software used for testing including Apache Maven [15], [16] and Pitest [17]. My work is done in collaboration with colleagues and students who are identified on my C.V.

Improving the Reliability of CI

ICSE 2021, ISSTA 2020, ISSTA 2019, ASE 2018, ICSE 2018, ICST 2018, FSE 2015, IEEE SW 2015, ICSE 2014 Distinguished Paper

NSF CCF-1763822 "SHF: Medium: Collaborative Research: Enhancing Continuous Integration Testing for the Open-Source Ecosystem" with Darko Marinov and Lingming Zhang ($1.2m total)

One major problem faced by developers who adopt CI arises from tests that can seemingly fail entirely unexpectedly and often randomly, so called "flaky tests." Flaky tests undermine efficiency with CI, because developers cannot easily determine when a test failure is due to their recent changes or due to flakiness. Microsoft has reported 5% of test failures in Windows and Dynamics are caused by flaky tests [30], and Google has reported that flaky tests accounted for 73K of the 1.6M (4.56%) *daily* test failures in the Google TAP system for regression testing [31]. There are many underlying causes of non-determinism in tests such as shared files, state on an external server, or device configuration — not to mention non-determinism from thread scheduling and asynchronous events. Prior to CI, tests were run sufficiently infrequently that manually examining infrequent flaky failures was tolerable. However, with CI, a single test might automatically be run *hundreds* of times per day, making an otherwise unlikely nuisance a common complaint. Flaky tests are unavoidable, and developers need new tools and techniques for detecting and mitigating that flakiness.

Test order dependencies are one cause of flaky tests, and occur when one test's execution can influence the behavior of another (to inadvertently fail). Test order dependencies are a nuisance to developers who want to run only a subset of their tests (as in the context of existing work in test selection or test minimization) or want to re-order tests (as in the context of existing work in test prioritization). My contributions in the space of test order dependencies include approaches to very efficiently isolate and remove test-order dependencies [3], as well as approaches to precisely detect which tests depend on each other to improve test reordering approaches [4], [5]. Most recently in this line of work, my analysis of state-of-the-practice approaches to isolate Java test cases [6], led to a contribution to the popular open-source platform *Maven* that reduced the time needed to clean up from each test by 20x [15].

When a test failure occurs in CI, one popular strategy to determine if the failure was due to flakiness or due to a recently introduced bug is to re-run the test. If a test first fails, then is re-executed (without any other changes) and passes, then it is quite likely that the failure is due to flakiness. If the test repeatedly fails when re-executed, then it may be more likely that the failure is due to a defect, and developers should attempt to debug it. However: rerunning tests can be time consuming, and rerunning all failing tests is not feasible for large software companies running millions of tests per-day. My first effort towards automati-

cally determining if a test failure is flaky, DeFlaker, uses extremely lightweight program instrumentation to, in most cases, determine if a failure is due to flakiness or not [7]. A key design constraint in this system is that the instrumentation must be sufficiently lightweight to avoid any performance impact during test execution: otherwise it would be faster to simply rerun failing tests. I have also studied how to integrate flaky failure detection approaches into software analyses that rely on testing, like mutation analysis [8]. My ongoing research in determining whether a test failure is due to flakiness or not aims to make this prediction without requiring any program instrumentation, relying instead only on log files.

Given that flaky tests are a relatively new phenomena (an FSE 2014 article is generally regarded as the first academic literature on the topic [31]), there are also many foundational questions unanswered. For example, to develop effective new techniques to help *prevent* flakiness, it would help to understand: "What kinds of code changes cause tests to become flaky?" and: "Are flaky tests flaky starting from when they are first written, or do they become flaky later on in development?" To answer these questions, I created "software archaeology" experiments, building software to automatically examine thousands of revisions of dozens of open-source software projects, repeatedly executing each test suite to identify and profile flaky tests [9]. Prior work has built datasets of flaky tests by re-running test suites dozens or hundreds of times in order to find tests that can both pass and fail for the same code under test. To provide a richer dataset that incorporates flaky tests that fail less frequently than $1/100$ runs, I extended my experimental methodology to re-run test suites $10,000$ times [2]. I found that fewer than half of the flaky tests that were detected in the experiment were detected after just 100 re-runs, underscoring just how difficult it is to study this phenomena. Using this dataset, we proposed a set of features that could be used as input to a classifier to predict which tests are likely to be flaky. In just the past few years, this research area has grown, with new works building on my dataset [27]–[29]. My ongoing research studies the causes of these flaky test failures, allowing us to create new approaches to help developers understand and repair them faster.

Finding More Bugs Without Bugging Developers.

ICSE 2022, TOSEM 2021, ICSE 2020, ECOOP 2018, OOPSLA 2014

NSF CNS-1844880 "CAREER: Amplifying Developer-Written Tests for Code Injection Vulnerability Detection" ($500k)

Continuous Integration is said to be a "force multiplier" for developers' time by using (relatively) cheap computing resources to execute large test suites more frequently, providing developers with faster feedback. However: Simply employing CI does not solve testing problems if developers' test suites are not sufficiently thorough, and manually enhancing these test suites is time consuming and challenging. For example: code injection vulnerabilities have become increasingly common and are ranked as #1 on OWASP's most recent list of critical web app vulnerabilities, and were used in the high-profile 2021 Log4J vulnerability and the 2017 Equifax breach. In order to detect these vulnerabilities before releasing their software, developers need techniques to *enhance* their existing test suites to generate more inputs and detect when those inputs violate correctness or safety properties of their software.

As a first step to detect code injection vulnerabilities in Java code, I created RIVULET [10]. RIVULET is an approach that uses existing developer test cases along with an analysis called *dynamic taint tracking* (as implemented by my tool, Phosphor [1]) in order to detect information flows vulnerable to an attack. When a flow is witnessed that could be vulnerable, RIVULET derives a follow-up input that can confirm the presence of a vulnerability. When applied to the version of Apache Struts exploited in the 2017 Equifax attack, RIVULET quickly identifies the vulnerability (with no false positives). I compared RIVULET to the state-of-the-art static vulnerability detector *Julia* on benchmarks, finding that RIVULET outperformed it in both false positives and false negatives. RIVULET also detected previously unknown vulnerabilities in Jenkins and iTrust. However, a significant limitation of RIVULET is that it requires developers to provide automated tests for their web application, which reveal these vulnerable data-flows.

In order to improve the reach of existing test cases, we are turning to fuzz testing. Fuzz testing is an automated testing technique that generates inputs with the goal of revealing otherwise untested behaviors. An emerging approach for generating inputs for Java applications relies on *property tests* in the style of

QuickCheck [32]. With property testing, developers write generators that create domain-specific inputs for their program, and specify properties that are expected to hold over all inputs created by those generators. Then, the property testing framework invokes the generator with random inputs which in turn, generate new inputs to the program under test. Leveraging insights from coverage-guided fuzzing, this random generation can be biased to evolve a corpus of inputs that reveal more interesting behavior than the random approach. CONFETTI, my recently published work in input generation examines how to most effectively combine dynamic analysis (e.g. concolic execution) with generators in fuzzing. CONFETTI achieved better branch coverage than the state-of-the-art fuzzer and revealed 15 previously unknown bugs in the open-source projects BCEL (Apache Foundation) and Closure Compiler (Google). We are now studying the feasibility of using CONFETTI with web apps, combining it with RIVULET to identify user-controlled inputs that flow to vulnerable functions and generate exploits that demonstrate the presence of a vulnerability.

Fuzzers are extremely complex systems to implement, and what might seem to be a simple engineering decision could have a tremendous impact in the overall performance of the system. As my students and I continue to build new fuzzers, we are increasingly focused on the rigorous evaluation of these design decisions. Our ongoing work in fuzzing includes not only guided fuzzing techniques to detect code injection vulnerabilities, but also foundational empirical evaluations of unchallenged assumptions.

Automating Reproducibility for Software Experiments. FSE 2022, ICSE NIER 2023

In preparation: NSF Infrastructure Proposal "Enabling Continuous Large Scale Software Engineering Experimentation" with CMU PIs Christopher Timperley, Michael Hilton and Lauren Herckis (projected $2m total)

The past twenty years of software testing and analysis research has seen a tremendous growth, fueled by the availability of increasingly rich open-source software datasets. These datasets consist of code and associated artifacts like version history, bug information, test suites, and build scripts to dynamically exercise them. These open-source artifacts allow researchers to study (at a large scale) how real bugs are introduced, detected, and repaired, enabling the growth of new and impactful research areas, including automated test generation, automated program repair, fault localization and regression testing. Unfortunately, this increase in experimental datasets has created a tremendous problem for designing, implementing, executing, and reproducing experiments on these datasets, which might require *decades* of CPU time. Designing and parallelizing large experiments requires specialized knowledge of distributed systems that researchers often lack, often resulting in scripts that are hard to share and reuse. This problem affects software engineering, particularly in the areas of automated program repair, fuzzing, and testing. However, similar challenges plague researchers in many other fields such as the computational sciences.

My colleagues and I have studied the artifact evaluation process in Software Engineering in order to better characterize challenges designing and implementing these large-scale evaluations [11]. Artifact evaluation processes have focused on how to create an *evaluation* of artifacts for quality attributes like *portability* across computing resources, *reproducibility* of evaluation results and *reusability* of research tools. However: Portability, reproducibility and reusability are all quality attributes, and, as with most other quality attributes in software engineering, are achieved with the greatest ease when they are considered at each step of the software development lifecycle — not at the end.

Using the lens of software testing, these large-scale experiments can be thought of as performance tests, not unlike those used by large software companies like Facebook to validate their systems. CI could be an excellent solution to these challenges: CI workflows can be adapted and reused in new contexts, and they can be executed on different computing infrastructure. However, building and deploying new CI infrastructure for a project is a significant undertaking — professional software companies often have entire teams dedicated to this role. My vision is to build an open-source ecosystem of reusable components to enable the efficient reuse of experiment scripts. At the same time, this infrastructure will catalyze new research into how to best apply CI to large-scale systems. My reproducibility project and its goals are outlined in a recent ICSE "new ideas" article [12].

Open Source Dependency Management. **ICSE 2023, MSR 2023**

Modern software development relies inextricably on open source package repositories on a massive scale. For example, the NPM repository contains over two million packages and serves tens of billions of downloads weekly, and practically every JavaScript application uses the NPM package manager to install packages from the NPM repository. The core of a package manager is its dependency solver, which tries to quickly find the version to use for each dependency (including transitive dependencies) that a package requires, subject to any and all version constraints. Developers may choose to use the "semantic versioning" scheme, where versions are numbered in the form `major.minor.bug`, where `major` denotes breaking API changes, `minor` denotes non-breaking changes that add new functionality, and `bug` denotes a backwards-compatible fix. Flexible version constraints allow developers of downstream (i.e., dependent) packages to specify which types of updates they are willing to automatically accept.

Unfortunately, NPM uses a greedy algorithm that can duplicate dependencies, fail to include the most recent versions of dependencies, and can even introduce new vulnerabilities while trying to avoid others. Ideally, a developer could specify policies such as "dependencies must not have any critical vulnerabilities," "packages should not be duplicated," and combine these with the basic objective of "select the latest package versions that satisfy all constraints." To address this gap, we created MAXNPM: a complete, drop-in replacement for NPM, which empowers developers to combine multiple objectives. We evaluated MAXNPM on a large dataset of widely-used packages from the NPM repository, finding that MAXNPM outperforms NPM: choosing newer dependencies for 14% of packages, shrinking the footprint of 21% of packages, and reducing the number or severity of vulnerabilities of 33% of packages [13].

To guide our (and others') future research in dependency management and supply chain security, we conducted a large-scale empirical study of how developers use semantic versioning and the flow of updates through the entire NPM ecosystem [14]. We built an infrastructure to download every version of every package on the NPM, creating a dataset that is continuously updated as new packages are released. We found that most updates to libraries (81%) are released as "bug" updates, and that most dependency constraints (84%) accept both bug and minor updates. Examining the factors that might delay a package from receiving an update (including security patches), we found that transitive dependencies could pose particular problematic cases. Specifically: if a project A depends on B which depends on C, if B specifies a constraint on C, then A may be unable to receive an update to C until B updates its constraint. Future research might examine key choke-points in the flow of updates in the dependency graph.

Our ongoing research in this project is examining techniques to detect breaking changes and malware in open-source dependencies. In order to detect breaking changes when an update for package P is introduced, we are finding all downstream packages D that depend on P and running the test suite for each downstream package D against the new version of P. This testing-based approach will help developers to identify that a change will (or will not) break downstream clients. Beyond breaking functionality, some updates might introduce malware. As a high-profile example: in 2018 an attacker compromised the NPM account of a maintainer of a popular package ("ESLint") and published malicious updates that, upon being installed, exfiltrated sensitive access tokens[1]. Automated detection of malicious updates is a crucial problem in today's dependency-heavy environment. Unfortunately, there are no publicly-available datasets of malware published on NPM, making it challenging for researchers to study approaches to automatically detect and filter out malicious updates. When NPM determines that a package contains malware, that package's contents are removed from NPM and its metadata is annotated with a flag indicating that it was removed due to malware. Our mirror of NPM, however, does not delete the malware, allowing us to build a large dataset that contains every piece of published malware detected by the NPM team. We are analyzing this dataset and designing new tools to efficiently detect these malware, integrating these efforts with our breaking change detectors.

[1] https://eslint.org/blog/2018/07/postmortem-for-malicious-package-publishes/

Referenced Publications

[1] J. Bell and G. Kaiser, Phosphor: Illuminating Dynamic Data Flow in Commodity JVMs, in *OOPSLA*, 2014.

[2] A. Alshammari, C. Morris, M. Hilton, and J. Bell, Flakeflagger: Predicting flakiness without rerunning tests, in *Proceedings of the 2021 International Conference on Software Engineering*, 2021.

[3] J. Bell and G. Kaiser, Unit Test Virtualization with VMVM, in *ICSE*, 2014, pp.550–561.

[4] J. Bell, G. Kaiser, E. Melski, and M. Dattatreya, Efficient dependency detection for safe java test acceleration, in *ESEC/FSE*, 2015, pp.770–781.

[5] A. Gambi, J. Bell, and A. Zeller, Practical test dependency detection, in *Proceedings of the 2018 IEEE Conference on Software Testing, Validation and Verification*, 2018.

[6] A. Celik, P. Nie, M. Coley, A. Milicevic, J. Bell, and M. Gligoric, Experience report: Debugging the performance of maven's test isolation, in *Proceedings of the 2020 International Symposium on Software Testing and Analysis*, 2020.

[7] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, Deflaker: Automatically detecting flaky tests, in *Proceedings of the 2018 International Conference on Software Engineering*, 2018.

[8] A. Shi, J. Bell, and D. Marinov, Mitigating the effects of flaky tests on mutation testing, in *Proceedings of the 2019 ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019.

[9] W. Lam, S. Winter, A. Wei, T. Xie, D. Marinov, and J. Bell, A large-scale longitudinal study of flaky tests, 3, (OOPSLA), 2020.

[10] K. Hough, G. Welearegai, C. Hammer, and J. Bell, Revealing injection vulnerabilities by leveraging existing tests, in *Proceedings of the 2020 International Conference on Software Engineering*, 2020.

[11] S. Winter, C. S. Timperley, B. Hermann, J. Cito, J. Bell, M. Hilton, and D. Beyer, A retrospective study of one decade of artifact evaluations, in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022.

[12] E. Barr, J. Bell, M. Hilton, C. Timperley, and S. Mechtaev, Continuously accelerating research, in *Proceedings of the 2023 ACM/IEEE International Conference on Software Engineering, New Ideas and Emerging Results*, 2023.

[13] D. Pinckney, F. Cassano, A. Guha, J. Bell, M. Culo, and T. Gamblin, Flexible and optimal dependency management via max-smt, in *Proceedings of the 2023 International Conference on Software Engineering*, 2023.

[14] D. Pinckney, F. Cassano, A. Guha, and J. Bell, A large scale analysis of semantic versioning in npm, in *Proceedings of the 20th International Conference on Mining Software Repositories*, 2023.

Referenced Open Source Contributions

[15] J. Bell, *Fixes [SUREFIRE-1516]: Poor performance in reuseForks=false*, https://github.com/apache/maven-surefire/pull/253, 2019.

[16] M. Coley and J. Bell, *Surefire-1584: Add option to rerun failing tests for JUnit5*, https://github.com/apache/maven-surefire/pull/245, 2019.

[17] J. Bell, *Enhance block coverage to correctly track coverage in presence of exceptions; Base mutant-test-pairs on blocks (not lines)*, https://github.com/hcoles/pitest/pull/534, 2019.

External References

[18] J. Cox, "Improved partial instrumentation for dynamic taint analysis in the jvm," Ph.D. dissertation, UCLA: Computer Science, May 2016.

[19] S. Amir-Mohammadian and C. Skalka, In-depth enforcement of dynamic integrity taint analysis, in *ACM Programming Languages and Security Workshop (PLAS)*, 2016.

[20] J. Toman and D. Grossman, Staccato: A Bug Finder for Dynamic Configuration Updates, in *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, S. Krishnamurthi and B. S. Lerner, Eds., vol. 56, 2016, pp.24:1–24:25.

[21] J. Singleton, "Advancing practical specification techniques for modern software systems," Ph.D. dissertation, University of Central Florida: Computer Science, 2018.

[22] S. Amir-Mohammadian, "A formal approach to combining prospective and retrospective security," Ph.D. dissertation, University of Vermont: Computer Science, 2017.

[23] M. Velez, P. Jamshidi, N. Siegmund, S. Apel, and C. Kästner, White-box analysis over machine learning: Modeling performance of configurable systems, in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp.1072–1084.

[24] C.-c. Tsai, J. Son, B. Jain, J. McAvey, R. A. Popa, and D. E. Porter, Civet: An efficient java partitioning framework for hardware enclaves, in *29th USENIX Security Symposium (USENIX Security 20)*, Aug. 2020, pp.505–522.

[25] J. Singleton, "Advancing practical specification techniques for modern software systems," Ph.D. dissertation, University of Central Florida, 2018.

[26] J. Jiang, S. Zhao, D. Alsayed, Y. Wang, H. Cui, F. Liang, and Z. Gu, Kakute: A precise, unified information flow analysis system for big-data security, in *Proceedings of the 33rd Annual Computer Security Applications Conference*, 2017, pp.79–90.

[27] Y. Qin, S. Wang, K. Liu, B. Lin, H. Wu, L. Li, X. Mao, and T. F. Bissyande, Peeler: Learning to effectively predict flakiness without running tests, in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2022, pp.257–268.

[28] D. Dell'Anna, F. B. Aydemir, and F. Dalpiaz, Evaluating classifiers in se research: The ecser pipeline and two replication studies, *Empirical Software Engineering*, 28, (1), 3, Nov. 2022.

[29] V. Pontillo, F. Palomba, and F. Ferrucci, Static test flakiness prediction: How far can we go? *Empirical Softw. Engg.*, 27, (7), Dec. 2022.

[30] K. Herzig and N. Nagappan, Empirically detecting false test alarms using association rules, in *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, IEEE Press, 2015, pp.39–48.

[31] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, An empirical analysis of flaky tests, in *22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, Nov. 2014, pp.643–653.

[32] K. Claessen and J. Hughes, Quickcheck: A lightweight tool for random testing of haskell programs, *SIGPLAN Not.*, 35, (9), 268–279, Sep. 2000.