Owolabi Legunsen

Research Statement

My research interests are in Software Engineering and Applied Formal Methods, with a focus on Software Testing and Runtime Verification. As software now controls many areas of our daily lives, the risk posed by software bugs has increased significantly. The goal of my research is to improve software reliability by helping developers find more bugs, find bugs faster, and find bugs more reliably. I use static and dynamic program analysis to create scalable techniques and tools that help to (1) *find more bugs* from existing tests through runtime verification of test executions against formal specifications [1, 2], (2) *find bugs faster* through evolution-aware runtime verification [3, 4] and regression test selection (RTS) [5, 6, 7, 8, 9], (3) *find bugs more reliably* by proactively finding flaky tests which nondeterministically pass or fail for the same code [10, 11, 12], and (4) *find bugs in emerging application domains* like probabilistic programming systems [13], approximate computing [14] and RTS tools [15].

My research challenges the conventional notion that dynamic analyses and formal methods are not ready for finding bugs during day-to-day software development, because they induce high overheads. I exploited the key insight that software evolves in small increments to make several techniques scale. Specifically, I created the first evolution-aware runtime verification techniques, the first practical static RTS approach, and the first lightweight flaky test detection technique that only analyzes code changes. My techniques significantly reduced accumulated overheads across multiple versions, and helped discover over 450 bugs in more than 90 open-source projects. My RTS approach can select 10× fewer tests than the RTS used in ultra-large software ecosystems (e.g., at Facebook, Google, and Microsoft), where thousands of interconnected projects evolve rapidly together. I also developed the first approaches for testing probabilistic programming systems and RTS tools, and for using approximate computing transformations to evaluate test suite quality.

In the future, I plan to generalize evolution-awareness to *all* program analyses so they can be included in developers' day-to-day workflow, and used for proactive bug detection in different development settings, e.g, in personal workspaces, in continuous integration (CI) systems, and in ultra-large ecosystems. I also plan to investigate automated debugging and fixing of bugs that my evolution-aware techniques find.

Runtime Verification During Testing

In runtime verification (RV), formal specifications of program behavior (specs, for short) are monitored at runtime. Executions that violate a spec can be corrected on the fly and/or reported. While previous research typically considered RV for production runs, I envision a future where RV is used during testing, to find more bugs from existing tests during day-to-day software development [3].

Contributions. I showed that RV can help find many serious bugs from tests that developers already have. In a large-scale empirical study [1, 2], I found over 198 bugs by monitoring 199 specs while executing almost 20 thousand tests in 218 open-source projects. Developers confirmed 74 of 95 bugs that I reported so far. My paper on this study won an **ACM SIGSOFT Distinguished Paper Award**.

My study revealed high RV overhead in machine time and developer time to inspect violations [1]. I observed that all previous RV techniques do not take software evolution into account, despite the fact that code typically evolves in very small increments [3]. However, during continuous integration, which is widely practiced by developers today, code changes many times per day and each change triggers execution of regression tests. Therefore, using previous RV techniques during continuous integration would wastefully re-incur the entire RV overhead after each code change.

I developed the first evolution-aware RV techniques to reduce overhead and adapt RV to regression testing and continuous integration. My evolution-aware techniques focus RV on code changes and reduce accumulated overhead across multiple program versions [3, 4]. One of my techniques relates specs to parts of the code where they may be violated. Then, after a code change, only specs that can be violated in affected parts of the code are re-monitored. My evolution-aware techniques reduced the accumulated runtime overhead of RV from 9.4× to 1.8×, did not miss any new violations, and showed 540× fewer violations. My analysis also showed that most RV computations during testing are redundant for bug finding—98% of execution traces processed by RV cannot lead to finding any more bugs than the other 2%. This finding is intuitive: many tests execute the same portion of code multiple times. Therefore, I am working on a new class of RV

algorithms and techniques to eliminate these redundant computations to achieve lighter-weight RV that is specialized for bug-finding during testing.

Regression Test Selection (RTS)

Regression testing is the most popular software quality assurance technique in practice, but it is very expensive. RTS reduces regression testing costs by re-running only tests that are affected by code changes. RTS computes test dependencies and re-runs only tests whose dependencies changed. Safe RTS selects all affected tests while precise RTS selects only affected tests.

Contributions. I developed a static RTS approach, STARTS, which uses only compile-time information to select a subset of tests to re-run; it does not require runtime instrumentation [6, 7]. I was the first to show that in several open-source projects, static RTS has similar end-to-end time as dynamic RTS; they both reduced regression testing time by around 35%. STARTS was often more imprecise than dynamic RTS, but its analysis was very fast, and it was rarely more unsafe. These results show that STARTS can be used when dynamic RTS is challenging, e.g., dynamic dependency computation may be too expensive in resource-constrained settings or ultra-large ecosystems, and flaky tests can cause dynamic RTS to miss dependencies. Further, reflection was the only reason why STARTS was unsafe in my experiments, so I developed a family of reflection-aware techniques to make STARTS safer [8].

I showed that a class-level RTS technique that computes test dependencies at the class level selects an order of magnitude fewer tests than project-level RTS that is currently used in ultra-large software ecosystems, like those at Google, Facebook, Microsoft, CRAN and npm [9]. Such ecosystems contain thousands of rapidly evolving, interconnected projects where client projects transitively depend on library projects. While research showed that class-level RTS achieves better overall time savings than RTS at lower granularity levels (e.g., methods or statements), project-level RTS is even coarser-grained—it selects *all* tests in a changed library and *all* its clients. It was not known whether class-level RTS can outperform project-level RTS at such scale. Yet, not all client tests will depend on changed library classes. My results, using 13,961 change sets in 168 libraries and their 580,876 clients, showed that class-level static RTS selects 10× fewer tests than project-level RTS in ultra-large ecosystems.

Proactive Detection of Flaky Tests

Flaky tests nondeterministically pass or fail on the same code, making test outcomes unreliable. Current approaches to dealing with flaky tests in practice *reactively* find flaky tests after they manifest. I work on *proactively* finding and debugging flaky tests earlier, ideally as soon as they are written.

Contributions. I developed DeFlaker to determine whether new test failures are flaky or due to code changes [12]. DeFlaker uses a novel differential coverage analysis to check if a newly failing test covers changed code. The test is likely flaky if it did not cover changes. DeFlaker helps avoid wasting developer time to debug new test failures by confirming early that the test is flaky. DeFlaker helped find 87 previously unknown flaky tests in 10 of 96 Java projects in my study; it had a low false alarm rate, and higher recall (95.5% vs. 23%) than a state-of-the-art reactive flaky test detector.

I also developed NonDex to find and debug flaky tests caused by wrong developer assumptions about implementations of APIs with underdetermined specifications; such specifications allow multiple implementations to return different results for the same input [10, 11]. For example, in Java, the order of elements in a `HashSet` is underdetermined, so different JDKs can return elements in different orders. NonDex finds flaky tests by randomly exploring different behaviors of underdetermined APIs during test execution. A test that fails during exploration is flaky. NonDex debugs the root cause by searching for the API invocation that caused the failure. NonDex helped find 75 previously unknown flaky tests in 24 open-source applications and precisely debugged 74 of them. NonDex was adopted by CheckStyle, a popular static analysis tool for Java.

Testing in Emerging Application Domains

I am interested in how to test applications in emerging domains, and how to use techniques in these domains to improve software testing. I developed the first approaches for systematically testing probabilistic programming systems (PP systems) and RTS tools. I also studied how approximate computing transformations can help in evaluating test suite quality.

Contributions. PP systems automate inference tasks in probabilistic programming, which recently emerged

to help programmers implement Bayesian inference problems. I performed the first study of PP system bugs and developed ProbFuzz, the first systematic approach for finding PP system bugs [13]. ProbFuzz is similar to compiler fuzzing, but it incorporates domain knowledge to generate *both* programs and input data, and it reasons about result accuracy. Developers so far accepted 51 of 67 bugs that ProbFuzz discovered in three PP systems and their underlying systems, PyTorch and TensorFlow.

To cope with the continually growing costs of regression testing, several RTS tools were recently proposed, with at least eight RTS tools released since 2015. However, there was no systematic approach to find bugs in RTS tools. I developed RTSCheck for testing that RTS tools select tests according to the implemented RTS techniques. Testing an RTS tool requires evolving programs, so RTSCheck uses three components to obtain evolving programs: automatic generation, bug databases, and project histories. RTSCheck uses a rule-based approach to check the output of an RTS tool on an evolving program, and has so far found 27 bugs in the three RTS tools that we evaluated: Clover, Ekstazi, and STARTS.

Approximate transformations were introduced in the emerging area of approximate computing for changing program semantics to trade result accuracy for improved energy efficiency or performance. I showed that approximate transformations can be effective mutation operators—they change program behavior differently than conventional mutation operators and the survival of mutants from approximate transformations helped find several bad tests and code that is amenable to approximation [14].

Future Directions

My broad research vision is to generalize evolution-awareness to *all* program analyses during continuous integration and in ultra-large software ecosystems, while providing support for automated debugging and repair of the bugs that these analyses find. I plan to make RV *even more* effective for finding bugs during testing, and extend the idea of evolution awareness to other dynamic analyses and formal methods, e.g, race detection, atomicity violation detection, etc. I also plan to explore evolution awareness in ultra-large software ecosystems, where the sheer scale, rapid rate of changes and high frequency of releases pose unique challenges to the adoption of different program analysis techniques. Long term, I want to develop a general theory of evolution-aware program analysis (both static and dynamic) during testing.

Improving Runtime Verification during Regression Testing. There are four aspects to my immediate plans for making RV even more effective for finding more bugs during testing. First, I plan to investigate whether specs that are mined from one project can have lower false alarm rates on that same project than the API-level specs that I used. High false alarm rates from API-level specs is often due to lack of program context and specs mined from a project could capture more context and generate fewer false alarms. Second, I plan to distill common bug patterns from literature and open source into monitorable specs. Violations of such specs could have lower likelihood of being false alarms. Third, I plan to investigate dynamic instrumentation composition so RV can be run on more open-source projects—I often had to omit projects and tests because they trigger instrumentation that clashed with RV instrumentation. Instrumentation composition could apply more broadly for using any two dynamic analysis techniques together. Lastly, I plan to deal with the effects of nondeterminism on RV; such nondeterminism leads to traversing different program paths, causing violations to be missed or falsely generated.

Evolution-awareness at Scale. Most of the software that affect our daily lives are in ultra large ecosystems, where it is no longer sustainable to reduce testing time by simply buying more hardware, and project-level RTS is too costly to run all affected tests after every change. The huge scale, very high rate of changes and frequency of releases in these ecosystems pose unique challenges. For example, at Google, more than 5,000 projects are under active development, having at least 20 code changes per minute, leading to more than 100 million test cases being run daily[1]. Yet, there is no class-level RTS technique that works at that scale. My initial results of evaluating class-level RTS in ultra-large ecosystems are very promising, but there are still many challenges to be addressed to make class-level RTS work. I plan to deal with the size explosion of dependency graphs that are constructed, analyzed, and updated after each change—in my study, class-level graphs had 77× more nodes than project-level graphs, translating to millions of nodes at the class-level, which took as much as 90 minutes to construct. This opens up a research opportunity that I plan to explore on applying recent advances on large-scale graph processing to RTS. I plan to investigate how to make RV,

[1] John Micco. Continuous Integration at Google Scale. Google Tech Talk. 2012

flaky test detection, and other program analyses work efficiently at this scale. Finally, I plan to investigate culprit finding. At this scale, it will be critical to efficiently find root cause(s) of test failures, beyond just reporting violations or test failures.

Other Evolution-Aware Analyses. I plan to investigate how to make other dynamic analyses evolution-aware so that they cost less during software evolution, and help developers find more bugs earlier in the software development process. Initial candidate techniques include dynamic race prediction, use-after-free detection, atomicity violation detection, specification mining, fuzzing, etc. These techniques can help find critical bugs, but incur high overheads. More importantly, till date, all these techniques only analyze one program version and would re-incur their entire overheads after each code change. None of these techniques work in the same way as RV, but I believe that the lessons that I learned in the developing evolution-aware RV techniques will help in making other analyses evolution aware.

From Bug Detection to Bug Fixing. In my experience with over 450 bugs, it took humans significantly more time to debug and patch the bugs found by my techniques, compared with the time for automated bug detection. Yet, the tools that found the bugs encountered much of the information needed for debugging during analysis. I plan to explore how to efficiently expose and leverage information that tools used in bug finding to help developers debug, and suggest patches automatically. Some of the time saved by evolution-awareness can be used to collect more information to assist in debugging violations. Automated techniques for suggesting fixes for RV violations would be a novel use of the error recovery concept in RV. I expect program repair of RV violations to be tractable since RV already accurately pinpoints the code location of each violation. Runtime information from monitoring and examples of previous manual fixes for similar violations can help narrow the search space of possible patches. My manual fixes for bugs that I found in my previous study [1] were quite simple and usually involved adding (not modifying) no more than four lines of code. Finally, I would like to explore automatic fixes for flaky tests that NonDex already finds and debugs.

Theory of Evolution-Aware Program Analysis. Evolution-Aware RV is an instance of a more general principle of evolution-aware program analysis during regression testing. It is not just about focusing the analyses on code changes, but also discovering and leveraging the peculiarities of these techniques in regression testing settings. Therefore, my plans to make other analyses evolution-aware is a way to obtain more instances of this general principle in order to understand it better. Longer term, I plan to extract and develop this principle into a general theory of evolution-aware program analysis, to provide a solid foundation for all future evolution-aware analyses.

References

[1] **Owolabi Legunsen**, Wajih Ul Hassan, Xinyue Xu, Grigore Roşu, and Darko Marinov. "How Good are the Specs? A Study of the Bug-Finding Effectiveness of Existing Java API Specifications". In: *Automated Software Engineering*. ASE. 2016, pages 602–613.

[2] **Owolabi Legunsen**, Wajih Ul Hassan, Xinyue Xu, Grigore Roşu, and Darko Marinov. "How Good are the Specs? A Study of the Bug-Finding Effectiveness of Existing Java API Specifications". In: *Automated Software Engineering Journal (ASEJ)* (2019), under review.

[3] **Owolabi Legunsen**, Darko Marinov, and Grigore Roşu. "Evolution-Aware Monitoring-Oriented Programming". In: *International Conference on Software Engineering, New Ideas Track*. ICSE NIER. 2015, pages 615–618.

[4] **Owolabi Legunsen**, Yi Zhang, Milica Hadzi-Tanovic, Grigore Roşu, and Darko Marinov. "Techniques for Evolution-Aware Runtime Verification". In: *International Conference on Software Testing, Verification and Validation*. ICST. 2019, in submission.

[5] Milos Gligoric, Stas Negara, **Owolabi Legunsen**, and Darko Marinov. "An Empirical Evaluation and Comparison of Manual and Automated Test Selection". In: *Automated Software Engineering*. ASE. 2014.

[6] **Owolabi Legunsen**, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. "An Extensive Study of Static Regression Test Selection in Modern Software Evolution". In: *Foundations of Software Engineering*. FSE. 2016, pages 583–594.

[7] **Owolabi Legunsen**, August Shi, and Darko Marinov. "STARTS: STAtic Regression Test Selection". In: *Automated Software Engineering*. ASE. 2017, pages 949–954.

[8] **Owolabi Legunsen**, August Shi, Milica Hadzi-Tanovic, Lingming Zhang, and Darko Marinov. "Reflection-Aware Static Regression Test Selection". In: *International Conference on Software Engineering*. ICSE. 2019, in submission.

[9] Alex Gyori, **Owolabi Legunsen**, Farah Hariri, and Darko Marinov. "Evaluating Regression Test Selection Opportunities in a Very Large Open-Source Ecosystem". In: *International Symposium on Software Reliability Engineering*. ISSRE. 2018, pages 112–122.

[10] August Shi, Alex Gyori, **Owolabi Legunsen**, and Darko Marinov. "Detecting Assumptions on Deterministic Implementations of Non-deterministic Specifications". In: *International Conference on Software Testing, Verification and Validation*. ICST. 2016, pages 80–90.

[11] Alex Gyori, Ben Lambeth, August Shi, **Owolabi Legunsen**, and Darko Marinov. "NonDex: A tool for detecting and debugging wrong assumptions on Java API specifications". In: *Foundations of Software Engineering, Tool Demo Track*. FSE Tool Demo. 2016, pages 993–997.

[12] Jon Bell, **Owolabi Legunsen**, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. "DeFlaker: Automatically Detecting Flaky Tests". In: *International Conference on Software Engineering*. ICSE. 2018, pages 433–444.

[13] Saikat Dutta, **Owolabi Legunsen**, Zixin Huang, and Sasa Misailovic. "Testing Probabilistic Programming Systems". In: *Foundations of Software Engineering*. FSE. 2018, pages 574–586.

[14] Farah Hariri, August Shi, **Owolabi Legunsen**, Milos Gligoric, Sarfraz Khurshid, and Sasa Misailovic. "Approximate Transformations as Mutation Operators". In: *International Conference on Software Testing, Validation and Verification*. ICST. 2018, pages 285–296.

[15] Chenguang Zhu, **Owolabi Legunsen**, August Shi, and Milos Gligoric. "A Framework for Checking Regression Test Selection Tools". In: *International Conference on Software Engineering*. ICSE. 2019, in submission.