Rohan Padhye - Research Statement

Today, the vast majority of software is written by humans. Since much of our society depends on software systems, the consequences of inadvertently introduced software bugs can be devastating. With new application domains emerging faster than the mechanisms to produce provably good software automatically, software bugs are here to stay for the foreseeable future. The predominant form of ensuring quality in practice is via *software testing*: a 50 billion USD market by some estimates, which is expected to keep growing.

Now, software developers have considerable domain expertise and are adept at writing functional test suites. However, handcrafted test cases often fail to catch corner-case bugs. My research focuses on algorithms and tools to *automatically* identify software bugs that do not surface during manual testing.

I develop *lightweight automated testing* techniques using *dynamic program analysis* and *random fuzzing*. Although these approaches are well known, their effectiveness as push-button tools is limited to the availability of good test inputs or to discovering a narrow class of software faults respectively. My research targets scenarios where the test program, the input format, and the testing objective becomes complex. In such settings, the traditional approach of analyzing or automatically testing a program in isolation either does not scale or produces inadequate results. A key insight of my work is that *we can make automated testing tools smarter by drawing upon artifacts incorporating the domain expertise of software developers*; thus, challenging testing problems can be made tractable. My solutions utilize a variety of data sources from existing unit tests to explicitly provided specifications. My research tools have uncovered hundreds of new bugs—affecting correctness, reliability, security, and performance—across widely used open-source and commercial software that runs on billions of devices. These tools have been adopted by large tech firms (e.g. Netflix and Samsung) and have been commercialized by security-oriented startups (e.g. FuzzIt and Pentagrid AG).

For example, JQF+Zest [**ISSTA'19a**,**b**] find semantic bugs deep within software that processes complex inputs—such as compilers—with the help of QuickCheck-like *generator functions* and user-provided *validity predicates*; FuzzFactory [**OOPSLA'19**] enables rapid customization of fuzzing tools for *domain-specific testing objectives* (e.g. regression testing and finding memory consumption bugs); Travioli [**ICSE'17**] identifies algorithmic complexity bottlenecks by *analyzing only functional unit tests*; PerfFuzz [**ISSTA'18**] automatically synthesizes inputs showcasing *worst-case complexity*; TSVD [**SOSP'19**] finds *thread-safety violations* when running unit tests in large-scale continuous integration; PARTEMU [**USENIX-Sec'20**] uses full-system emulation to perform fuzz testing of *trusted execution environments*. **Four of these papers have won awards**: *SOSP Best Paper*, *ACM SIGSOFT Distinguished Paper*, *ACM SIGSOFT Distinguished Artifact*, and *Best Tool Demo*.

Background on *Fuzz Testing*

Fuzz testing, or simply *fuzzing*, is a random test-input generation approach. Its key advantage over systematic techniques such as symbolic execution is scalability: a randomized search can explore many program behaviors quickly and can be easily parallelized. Fueled by the increasing availability of cheap computing resources, fuzz testing has become the predominant automated testing method used in practice. For example, Google's ClusterFuzz system claims to have found ~16,000 bugs in the Chrome project alone. Popular fuzzing tools such as *AFL* perform an *evolutionary search*: test inputs are generated by performing *random mutations* on the binary representation of previously saved inputs (e.g. bit flips and splicing of multi-byte chunks). These tools are *coverage-guided*: new inputs are saved if their execution on the test program leads to a new program location being visited. Such code coverage is collected using *lightweight program instrumentation*. A bug is found when a generated input causes the test program to *crash* (e.g. due to a segfault or assertion violation).

Semantic Fuzzing: Finding Bugs in Programs that Expect Highly Complex Inputs

Coverage-guided fuzzing tools such as AFL can effectively exercise a variety of code paths in programs that process binary data, e.g. media decoders. When testing large software such as compilers and build systems, which process inputs with complex structure (e.g. an XML file describing a build configuration), I observed that most of the bugs reported by conventional fuzzing tools lie in their syntax parsing stages only. This is unsurprising, since random byte-level mutations easily destroy the syntax and semantics of previously valid inputs. I thus asked: *can we provide mechanisms for developers to control the search space of fuzz testing?*

I introduced *Zest* [**ISSTA'19a**], which synthesizes semantically valid inputs with the help of domain expertise that can be provided using simple abstractions. Zest allows developers to specify a *generator* function that randomly samples *syntactically valid* inputs (e.g. XML DOM trees), in a fashion similar to the well-known QuickCheck tool. Unlike QuickCheck, however, Zest controls the "random" choices made in the generators. Zest's key insight is that low-level mutations (e.g. bit-flipping) on the pseudo-random bit-stream used by the generators corresponds to structural mutations in the inputs that they produce (e.g. deleting a DOM sub-tree). With this insight, Zest employs an evolutionary algorithm that searches through the space of syntactically valid inputs (e.g. well-formed XML). The search attempts to increase code coverage while also satisfying user-provided predicates that correspond to *semantic validity* (e.g. conformance to an XML schema). Zest has discovered several complex semantic bugs in very large code bases. For example, Zest found bugs in the Google Closure Compiler's optimization stage by synthesizing JavaScript programs that cause dead-code elimination to fail. Such bugs are far beyond the reach of conventional fuzzing tools such as AFL or Quickcheck, and require much less manual specification than dedicated compiler-fuzzing tools.

Zest is implemented in *JQF* [**ISSTA'19b**], a framework that I developed for guiding QuickCheck-like generators in Java. We are currently using JQF to implement other algorithms for such guidance, including reinforcement learning [**ICSE'20**] and neural networks. Together, JQF+Zest have enabled the discovery of over 40 previously unknown bugs in widely used open source software. The tools have been adopted by projects such as Netflix's Message Security Layer, Apache Tika, and the BouncyCastle crypto library. An independent startup (fuzzit.dev) now provides Zest as one its cloud-based services for use in continuous integration.

Performance Fuzzing: Discovering Algorithmic Complexity Bottlenecks

Profiling tools help developers diagnose performance problems in software. However, profilers are only useful when provided with test inputs that demonstrate pathological performance. Unfortunately, dedicated performance tests are not as commonly available as functional unit tests. I investigated several techniques that addressed the question: *can we discover algorithmic performance bottlenecks using functional test cases?*

Travioli [**ICSE'17**] identifies program functions whose worst-case complexity may be *accidentally* sub-optimal (e.g. quadratic instead of linear), possibly due to the use of the wrong data structures or algorithms. Travioli's key idea is to uncover algorithmic performance bugs by executing readily available functional unit tests and performing *dynamic analysis*, i.e., using lightweight instrumentation to observe program behavior at run-time. Prior work that addressed this class of bugs either assumed the availability of dedicated performance tests, relied on collections libraries with well-defined classes (e.g. `HashMap` in Java), or did not handle recursive functions. Travioli's innovation is in precisely identifying traversals of arbitrary data structures (e.g. ad-hoc graphs in JavaScript), even in the presence of possibly mutual recursion. These traversals are then analyzed to detect redundancies which would indicate super-linear complexity. Travioli helped discover *asymptotic* performance bugs in popular JavaScript projects such as D3 and Express.

Although Travioli identifies potentially buggy program functions, one has to still manually craft test cases that confirm the bugs. *PerfFuzz* [**ISSTA'18**] closes this gap by *automatically generating worst-case inputs*, using fuzz testing. Like Travioli, PerfFuzz starts with off-the-shelf functional tests cases. It then performs an evolutionary algorithm using random mutations to generate new inputs, while saving inputs that are likely closer to worst-case behavior. PerfFuzz's key advantage over prior fuzzing tools is that it saves inputs that maximize the execution count of any branch in the test program; this helps PerfFuzz discover distinct bottlenecks in the test program. In our experiments with real-world C programs, PerfFuzz outperformed its competition by a factor of 5x-69x in exacerbating loop iterations for a fixed input size. PerfFuzz revealed an exponential complexity bug in the parser of the Google Closure compiler. My collaborators and I have recently investigated improvements to this approach by evolving *grammars* that describe a family of inputs [**JPF'19**].

Domain-Specific Fuzzing: Automated Testing with User-Specified Goals

My experience with PerfFuzz and Zest led me to realize that while there is merit in building domain-specific fuzzing tools, there is no systematic process for doing so. Building ad-hoc derivatives of tools such as AFL requires weeks of effort. Further, such specialized fuzzing tools cannot be composed with each other easily. In response, I designed *FuzzFactory* [**OOPSLA'19**], a framework that generalizes coverage-guided fuzzing to user-specified testing goals. With FuzzFactory, users can customize the evolutionary search to save certain inputs, called *waypoints*, that make progress towards domain-specific testing objectives. We used FuzzFactory to prototype six fuzzing applications, of which four required less than 30 lines of code to implement! This includes three replications of prior work as well as novel test applications such as exacerbating a program's memory usage. A key advantage of FuzzFactory is that it enables *composition*. For example, a *super-fuzzer* that combines a waypoint strategy for generating *magic constants* with a strategy for stressing memory allocation reveals bugs that do not surface when using either strategy independently. The super-fuzzer can synthesize

LZ4 bombs and PNG bombs: tiny inputs which when processed by libarchive and libpng respectively allocate memory corresponding to a compression ratio that is greater than even the theoretical maximum!

Fuzzing Large Systems: Collaborations with Industry

I have also worked on applying random testing to large production systems in industry. I collaborated with Samsung to enable effective fuzzing of four ARM TrustZone-based operating systems that run on over 2 billion devices, mostly Android phones. TrustZone's hardware security features make instrumentation and dynamic analysis very challenging. We used full-system emulation and domain-specific fuzzing to discover security vulnerabilities in 48 *Trusted Applications* extracted from 16 Android smartphones [**USENIX-Sec'20**]. I also collaborated with Microsoft Research to automatically find concurrency bugs in a large-scale continuous integration environment**.** *TSVD* [**SOSP'19**] uses dynamic program analysis and probabilistic delay-injection to find *thread-safety violations* (e.g. concurrent updates to an unsynchronized `List` object). TSVD has discovered 1,000+ bugs when testing ~43,000 modules across active projects at Microsoft. The tool is now open-source.

Future Work: *Data-Driven Automated Testing*

The next generation of program analysis tools will have to deal with an increasing reliance on legacy software, rapid micro-deployments of code changes, client-side code validation, and a heterogeneous mix of application domains (e.g. distributed systems using machine learning and hardware-assisted security features). I envision a future in which automated testing tools can effectively utilize a vast ecosystem of domain-specific data sources as needed. I see a number of exciting research opportunities in this space.

First, most bug-finding tools used today assume that the program under test has never been seen before. In practice, however, critical software such as the GCC compiler have been fuzzed for decades. What can we learn from past fuzzing campaigns? I plan to investigate *adaptive fuzzing* approaches that can automatically adjust their heuristics based on previous executions. I believe that an application of machine learning techniques on generators [**ISSTA'19a**,**b**] will prove useful in isolating interesting input features. For example, we could perhaps learn that most bugs found in GCC relate to use of bitwise arithmetic in C programs. We could then bias C-program generators towards producing a variety of bitwise operators.

Second, a related problem is that of regression testing; that is, quickly finding bugs introduced by code changes. Ideally, we would like to test only the parts of the program that are affected by the change. Such *incremental fuzzing* largely remains an open problem despite several proposals in the literature. With FuzzFactory [**OOPSLA'19**], I proposed a preliminary solution for this problem, but it relies on starting with previously saved inputs which exercise the modified code. Other proposed approaches require too much time to statically analyze the code changes. I plan to investigate *human-in-the-loop* techniques where developers can communicate what input features are desirable to test a particular code component. For example, if a GCC developer updates its vectorizing optimizations, how can they tell their fuzzing tool to generate C programs with lots of nested loops and array accesses? On this front, I am eager to collaborate with HCI experts to study and improve the usability of (semi-)automated testing tools.

Third, an increasing number of program analysis tools are being run directly in production, in order to test client-specific platform interactions as well as locally synthesized customization code. In such scenarios, the performance of automated testing tools is critical. The use of program instrumentation for receiving feedback during fuzz testing slows down test execution. In turn, this reduces the number of inputs that can be evaluated per unit time. I recently investigated some VM-level optimizations for improving the performance of code patterns inserted by some instrumentation engines [**VMIL'19**]. Going forward, I plan to investigate the use of *static program analysis* to identify in advance a small subset of program locations whose instrumentation is most likely to help in achieving domain-specific testing objectives. For example, to reveal performance bugs in GCC, static analysis might identify that it is important to instrument the optimizer but not necessarily the parser.

Finally, I am interested in pursuing collaborations to build effective automated testing techniques for domains such as *distributed computing***,** *operating systems***,** and *AI-driven applications*. In such domains, many of the assumptions made by traditional fuzzing tools do not hold, including deterministic and idempotent execution, capabilities to instrument all components, and the availability of good test oracles. I have already started collaborating on a project that utilizes some ideas that I developed in Zest and FuzzFactory for testing object-detection modules in *self-driving cars*.

In this way, I would like to continue pushing the boundaries of automated testing tools by effectively incorporating external data sources as well as the domain-specific expertise provided by humans.

References

[**ICSE’17**] Rohan Padhye and Koushik Sen. Travioli: A Dynamic Analysis for Detecting Data-Structure Traversals. In *Proceedings of the 39th ACM/IEEE International Conference on Software Engineering* (ICSE’17).

[**ISSTA’18**] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. PerfFuzz: Automatically Generating Pathological Inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (ISSTA’18). **ACM SIGSOFT Distinguished Paper Award**.

[**ISSTA’19a**] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Semantic Fuzzing with Zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (ISSTA’19). **ACM SIGSOFT Distinguished Artifact Award**.

[**ISSTA’19b**] Rohan Padhye, Caroline Lemieux, and Koushik Sen. JQF: Coverage-Guided Property-Based Testing in Java. In *Proceedings of the 28th International Symposium on Software Testing and Analysis* (ISSTA’19), Tool-demo paper. **ACM SIGSOFT Tool Demonstration Award**.

[**OOPSLA’19**] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. FuzzFactory: Domain-Specific Fuzzing with Waypoints. In *Proceedings of the ACM on Programming Languages, Volume 3 Issue OOPSLA*. 2019.

[**VMIL’19**] Rohan Padhye and Koushik Sen. Efficient Fail-Fast Dynamic Subtype Checking. In *Proceedings of the 11th ACM SIGPLAN Workshop on Virtual Machines and Intermediate Languages* (VMIL’19).

[**JPF’19**] Xuan Bach D. Le, Corina Pasareanu, Rohan Padhye, David Lo, Willem Visser, and Koushik Sen. SAFFRON: Adaptive Grammar-based Fuzzing for Worst-Case Analysis. In *Proceedings of the 2019 Java PathFinder Workshop* (JPF’19).

[**SOSP’19**] Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath, and Rohan Padhye. Efficient and Scalable Thread-Safety-Violation Detection. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (SOSP’19). **Best Paper Award**.

[**USENIX-Sec’20**] Lee Harrison, Hayawardh Vijayakumar, Rohan Padhye, Koushik Sen, and Michael Grace. PARTEMU: Enabling Dynamic Analysis of Real-World TrustZone Software Using Emulation. To appear in *Proceedings of the 29th USENIX Security Symposium*. 2020.

[**ICSE’20**] Sameer Reddy, Caroline Lemieux, Rohan Padhye, and Koushik Sen. Quickly Generating Diverse Valid Test Inputs with Reinforcement Learning. To appear in *Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering* (ICSE’20).