Caroline Lemieux – Research Statement

As software becomes more pervasive in all aspects of society, the consequences of bugs and other software defects have become all the more severe. The Consortium for IT Software Quality estimated that in 2018, the cost of poor software quality amounted to *$2.84 trillion* in the US alone.[1] In my research, I work to alleviate these costs by building tools that help developers improve the *correctness*, *security*, and *performance* of software.

A recently successful class of such tools are *fuzz testers*. **Fuzz testing** uses random search methods to find bug-inducing inputs in software. These inputs help developers reproduce bugs and reason about their root causes. Modern fuzz testing tools have been successfully applied to a broad array of programs: Google's OSS-Fuzz project alone has found over 20,000 bugs in 300 open source projects.[2]

In my PhD research, I improved fuzz testing over three main algorithmic dimensions:

1. Coverage-guided fuzz testing does not consistently **find performance and resource consumption errors**, even though attackers can exploit these to deploy denial-of-service attacks. I developed a multi-objective maximization search algorithm that readily finds such bugs.
2. Mutational fuzz testing is great at stress-testing input validation, but rarely generates structurally-sound inputs that **explore the core logic of programs**. I developed algorithms to filter mutations and obtain higher-level mutations which generate more structurally-sound inputs.
3. Generator-based fuzz testing, which uses a hand-written input generator to create inputs, ties its search to a particular input distribution. I developed methods to **automatically adapt these distributions** to the program under test, and even used these methods to tackle program synthesis.

My research on fuzzing has won a **Distinguished Paper Award**, **Distinguished Artifact Award**, **Tool Demonstration Award**, and **Best Paper Award (Industry Track)**.

Going forward, I will build tools that reduce the upfront cost of deploying fuzz testing on new software systems. I will aim to *help developers write test drivers* with higher code coverage, and *parameterize or mock external components* that prevent efficient fuzz testing. Relatedly, I will work on automatically *inferring the structure of inputs*, enabling users to run more effective generator-based fuzzing. I also want to help developers *reason about the relevance of fuzzer-found bugs*, and will explore specification languages for bug relevance, as well as automated patching. I will tackle these problems using insights from my work on test driver generation, program synthesis, and specification mining.

Further, I will target software on which existing fuzz testing tools *cannot* be used. As fuzz-testing-found inputs expose buggy control-flow paths, they are most useful to developers *when control-flow is the most complex-to-reason-about* aspect of the program. This is not the case for many modern programs, like distributed systems, mobile applications, and neural networks. However, there *are* artifacts that help developers reason about the complexities of these systems. For example, groups of inputs in the test data which fail in a similar manner can help ML developers reason about biases in their training methodology. I will leverage my knowledge of the strengths and limitations of fuzz-testing-like search algorithms to automatically find such artifacts.

Improving Coverage-Guided Mutational Fuzzing

Modern **coverage-guided mutational fuzzing (CGF)** tools—AFL, libFuzzer, honggfuzz—have improved the quality of many widely-used software projects. While the well-known Heartbleed bug was present in public version of OpenSSL for two years before it was fixed,[3] the only critical vulnerability to date found in OpenSSL was found by CGF a day after it was released, and fixed two days later.[4]

[1] The Cost of Poor Quality Software in the US: A 2018 Report (p. 5)

[2] `https://github.com/google/oss-fuzz`

[3] `https://www.openssl.org/news/secadv/20140407.txt`, `https://www.openssl.org/source/old/1.0.1/`

[4] `https://www.openssl.org/news/secadv/20160926.txt`

Although this second vulnerability was potentially more severe than Heartbleed, thanks to the adoption of CGF, it had no remarkable security impacts. In my work, I developed algorithms which further expand the applicability of CGF.

CGFs main innovations are (1) a pseudo-genetic algorithm for input generation which (a) uses byte-level mutation operations to create new inputs and (b) determines the fitness of inputs by whether they achieve new coverage; as well as (2) low-overhead instrumentation in order to quickly collect this coverage feedback. Paired with an efficient, empirically-verified implementation, this method enabled CGF to scale to many large software projects.

Generalizing Fuzzer Guidance: ***Finding Resource Consumption Errors***

CGF relies on *branch coverage* to guide its bug-finding. Branch coverage tracks, for each conditional statement in the program, which side (branch) of the statement was exercised (covered) by an input. As it prioritizes broad program exploration, this signal is not helpful for finding algorithmic complexity errors or out-of-memory errors. Unfortunately, these errors can have serious security implications: an attacker can cause a denial-of-service attack by sending inputs that consume an unreasonable amount of compute resources.

Prior state-of-the-art tried to use fuzzing to find such inputs by aiming to maximize a single performance objective. However, I observed it got stuck at inputs that hit a local maxima in their performance objective, and never got to the true worst case. In PerfFuzz [3], I introduced a multi-objective maximizing search algorithm which overcame this problem. Thanks to this, PerfFuzz was able to find inputs that exemplified the worst-case algorithmic complexity of several programs: quadratic blowup in (1) a regex library, (2) a linked-list hash table, and (3) error processing in an XML parser. PerfFuzz won a **Distinguished Paper Award** at ISSTA'18.

Fundamentally, the PerfFuzz algorithm allows us to find inputs which maximize *values* for certain *keys*. We can use it to find inputs that maximize memory allocated (value) at memory allocation locations (keys). Or, to find inputs that get through `if` statements with a strict equality condition—a real challenge for CGF—, by maximizing the number of bits matched (value) for each equality condition (keys). In FuzzFactory [6] we allowed users to customize these key-value maps, and run the PerfFuzz algorithm over a set of such maps. This enabled us to create a fuzzer that consistently found new memory usage "bombs" (e.g. a 21-byte input causing a 4GB memory allocations) in previously heavily-fuzzed software like *libarchive*.

Mutating to Retain Structure: ***Exploring Core Logic***

Because CGF models inputs as byte-sequences (e.g. files, standard input), its mutations can easily corrupt high-level input structure. CGFs typical mutations include flipping random bytes, duplicating sequences of bytes, setting bytes to 0. So, CGF might produce the mutant `<a>b</b</a>` from `<a>b</a>`, ruining the XML format. Such structurally unsound inputs cannot exercise the core logic of software.

In FairFuzz [4], we introduced the concept of a *mutation mask*, which specifies which bytes of an input can be mutated while still exercising important parts of the program. FairFuzz also prioritized mutations on inputs exercising important parts of the program, which led to a more depth-first search. FairFuzz achieved up to 10.6% higher branch coverage (a common metric for testing effectiveness) than state-of-the-art, and continues to be popular in the fuzzing community. Users actively enquired about its integration into AFL++,[5] because "in [their] tests, **its approach was very effective** [emphasis added]", and said that **best practice involves using FairFuzz** in a fuzzing deployment.[6]

[5] `https://github.com/AFLplusplus/AFLplusplus/issues/18`

[6] `https://github.com/AFLplusplus/AFLplusplus/issues/258#issuecomment-599226036`

However, for highly-structured inputs, e.g. XML documents, FairFuzz remains unlikely to mutate an input in a structurally-significant manner, e.g. adding a child or attributes to an existing element. To tackle this problem, we looked to another branch of modern fuzzing.

Automatically Adapting Distributions of Random Input Generators

Unlike CGF, which produces inputs via mutation, *generator-based fuzzing* produces inputs by repeatedly calling a generator. Generator-based fuzzing is the backbone of commercial fuzzing tools such as *Peach*, *beStorm*, *Defensics*, and *Codenomicon*. A *generator* uses calls to some source of randomness to produce a different element from the space of inputs each time it is called. Generators are a natural way for developers to describe a *search space of inputs*. However, they also couple a particular—often non-optimal for testing—*probabilistic distribution* with this search space.

In Zest [5], we brought together generator-based fuzzing and CGF. We observed that small changes in the values returned by the generator-consumed source of randomness resulted in small changes to the generator-returned input. Zest replaced the source of randomness with a controllable stream of "random" numbers. Zest then performed CGF by mutating this stream of numbers, *rather than the generator-produced input*, so the inputs Zest generated were always well-structured. As such, Zest was able to find bugs in the core logic stages of programs, like a logic error in the code optimization stage of the Google Closure compiler. Zest won a **Distinguished Artifact Award** at ISSTA'19.

In RLCheck [7], rather than mutating the stream of random numbers to control the distribution from which inputs are drawn, we explicitly control the distribution from which inputs are drawn. In particular, we replace uses of the source of randomness in the generator with abstract "choice" operators. These operators are backed by reinforcement learning agents, which learn over time which choices are likely to result in the production of a new valid input. RLCheck rewards these agents using only return codes from the program, rather than coverage information, enabling it to find orders-of-magnitude more unique valid inputs than Zest in the same time frame.

I have also leveraged input generators to solve other, non-testing, search problems. For example, we can solve program synthesis with generator-based search: given an input-output example *(I,O)*, use a program generator to generate random programs until we find the program p such that *p(I)=O*. We leveraged this observation in AutoPandas [2], a program synthesis engine for the Python library `pandas`. To achieve good performance, AutoPandas replaced random choice points in the generator with a neural network trained to return the choice most likely to result in a program *p* s.t. *p(I)=O*. Thanks to this, AutoPandas was able to solve program synthesis problems taken from StackOverflow.

Future Work

Looking forward, I will further increase the impact of my work by (1) building tools that enable developers to deploy fuzz testing on more systems and (2) investigating new notions of "test-input generation" for software which does not fit into fuzz testings notion of "test program".

Reducing Upfront Costs of Fuzz Testing

In spite of its bug-finding power, fuzz testing is still used primarily as a quality assurance tool for particularly widely-used libraries, rather than a universal tool to help developers improve the quality of code. Upfront costs such as building test drivers, writing specifications of input structure, and understanding fuzzer-found bugs still prevent the wide adoption of fuzzing. By automating these tasks, we help reduce these costs, and thus, improve the quality of software. My work on automating test driver generation in FUDGE [1] shows this concretely: over 200 FUDGE-generated drivers were upstreamed into open source libraries, and **enabled 150 security-improving fixes**.

Fuzz testing more code. To fuzz a piece of code, we need an entry-point to that piece of code, often called a *test driver*. Existing work, including FUDGE, tackles the problem of building test drivers for libraries that expose a public API. However, this work requires examples of library usage by client code, which may not exist for more general software systems. I will explore how we can look at the context of a function within its own code base to generate drivers. Further, in many pieces of software, the core functionality is deeply intertwined with the behavior of an external component. In these cases, we need tools that help developers build *mocks* of these components, which capture the components' core test-relevant behavior. I will develop methods to automatically synthesize such mocks, building on my work in input-output-example-based program synthesis.

Inferring and adjusting input structure specifications. To fuzz systems where the input is not well-modelled by a byte sequence, developers spend a large amount of time writing generators, grammars, or protobufs modeling the input structure. Existing work on input grammar inference, while promising, is so far restricted to particular classes of parsing programs, or does not generalize enough. Utilizing scalable static analysis techniques, like those used in FUDGE, I will expand this to automated generator inference, which captures higher-level relationships in inputs. Further, depending on the nature of the bugs the developer wants to find, or the position of the program in its larger input ecosystem, the relevant space of test inputs may vary. Building on my work on adapting generator distributions, I will develop tools that help developers understand and edit a generator's search space.

Coping with fuzzer-found bugs. Finally, fuzz testing has been most adopted in systems which directly accept user input. In these systems, memory-corruption errors often lead to security vulnerabilities, and so fuzzer-found bugs are viewed as important. This is not necessarily the case for, for example, a compiler, where strange behavior on a particularly esoteric program may pose neither a security risk nor ever be encountered by a user. I will develop specification languages that enable developers to express their model of bug relevance. I will use my background in specification mining to ensure these languages are expressive but understandable. Then, I will explore how to use the specifications to not only filter bugs, but guide input generation towards relevant bugs.

Rethinking "Test-Input Generation"

Test-input generation centers control flow, but the complexity of many modern software systems comes from sources other than control flow. The complexity of networked systems lies in the interaction between different nodes of the system. Mobile applications are complex in part because of their reactive nature, as well as the interaction of GUI elements with a variety of operating systems. Deep learning applications draw their complexity in large part from the process used to train their underlying models. In these systems, test-input generation may not be the right way to help developers improve software.

Throughout 2019, I interacted with students, practitioners, and thought leaders in machine learning to investigate core areas in which programming languages and software engineering research could improve the machine learning development experience. This culminated in a talk I gave at the Workshop on ML Systems at SOSP'19. Amongst other findings, I discovered that tools for single-input generation were not particularly compelling to ML developers, but that tools which could find groups of similar inputs, all causing the same failure mode in the system, could help developers reason about where their training went wrong.

In my work on fuzz testing, I learned how to adjust input search problems to fit particular problem constraints, and developed a deep understanding of the strengths and limitations of fuzz-testing-like search algorithms. I will use this expertise to develop new notions of "test-input generation" for today's hardest programming tasks.

References

[1] D. Babić, S. Bucur, Y. Chen, F. Ivančić, T. King, M. Kusano, C. Lemieux, L. Szekeres, and W. Wang. FUDGE: Fuzz Driver Generation at Scale. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, pages 975–985, New York, NY, USA, 2019. Association for Computing Machinery.

[2] R. Bavishi, C. Lemieux, R. Fox, K. Sen, and I. Stoica. AutoPandas: Neural-Backed Generators for Program Synthesis. *Proc. ACM Program. Lang.*, 3(OOPSLA), Oct. 2019.

[3] C. Lemieux, R. Padhye, K. Sen, and D. Song. PerfFuzz: Automatically Generating Pathological Inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 254–265, New York, NY, USA, 2018. Association for Computing Machinery.

[4] C. Lemieux and K. Sen. FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 475–485, New York, NY, USA, 2018. Association for Computing Machinery.

[5] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. Le Traon. Semantic Fuzzing with Zest. ISSTA 2019, pages 329–340, New York, NY, USA, 2019. Association for Computing Machinery.

[6] R. Padhye, C. Lemieux, K. Sen, L. Simon, and H. Vijayakumar. FuzzFactory: Domain-Specific Fuzzing with Waypoints. *Proc. ACM Program. Lang.*, 3(OOPSLA), Oct. 2019.

[7] S. Reddy, C. Lemieux, R. Padhye, and K. Sen. Quickly Generating Diverse Valid Test Inputs with Reinforcement Learning. In *Proceedings of the 42nd International Conference on Software Engineering*, ICSE 2020. IEEE, 2020.