Caroline Lemieux -- Research Statement

As software becomes more pervasive in all aspects of society, the consequences of bugs and other software defects are 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 help 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 [7].

In my PhD research, I improved fuzz testing over three main dimensions. First, while performance and resource consumption errors can be exploited by attackers to deploy denial-of-service attacks, fuzz testing was not able to consistently find such errors. I developed a multi-objective maximization search algorithm that could readily find such bugs. Second, modern mutational fuzzers are great at stress-testing input validation, but are rarely able to generate structurally sound inputs that explore the core logic of programs. I developed a mutation "masking" algorithm and a way to get high-level mutations from hand-written input generators to generate more structurally sound inputs. Third, I realized that generator-based fuzz testing, which uses a hand-written input generator to create inputs, is too tied to a particular sample distribution. I developed methods to automatically adapt this distribution to the program under test, and even tackled program synthesis by smartly altering these distributions. My research on fuzzing has won a **Distinguished Paper Award**, **Distinguished Artifact Award**, **Tool Demonstration Award**, and **Best Paper Award (Industry Track)**.

Going forward, to help developers improve the quality of software, I will innovate far beyond the details of these fuzz testing algorithms. Consider first all the infrastructure problems that prevent developers from using fuzz testing. At Google, I developed FUDGE [14], which conducts large-scale code analysis to synthesize fuzzing-friendly code entry points for a target library. In ongoing work trying to help developers fuzz test software with complex input structure, I developed a new algorithm for input-grammar inference. 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 methods.

Further, I will expand my work beyond the fuzz testing notion of "bug-inducing input". Bug-inducing inputs which expose buggy control-flow paths 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 software systems, 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 search algorithms to develop tools which can 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][4], the only *critical* vulnerability to date found in OpenSSL was found by CGF a day after it was released, and fixed two days later [2]. Although this second vulnerability was more potentially severe than Heartbleed, thanks to the adoption of CGF, it had no remarkable security impacts.

CGF's 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 Feedback: 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. Unfortunately, this signal is not so helpful for finding data-dependent bugs, like floating point errors, integer overflows, algorithmic complexity errors, or out-of-memory errors. These last two classes of 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.

As such, it is critical that developers have inputs showing the breadth of performance behavior of their program. In PerfFuzz, I introduced a multi-objective maximizing search algorithm in order to find such inputs automatically. Unlike prior state-of-the-art which tried to maximize a single performance objective, PerfFuzz worked to independently maximize hit counts of different components of the program under test. 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. Prior work got stuck at inputs that hit a local maxima in their performance objective, and never got to the true worst case. PerfFuzz won a **Distinguished Paper Award** at ISSTA'18.

This algorithm has power beyond finding algorithmic complexity issues. We properly generalized it in FuzzFactory [9] by decoupling the maximizing search algorithm from the instrumentation which creates the feedback to maximize. This enables developers to easily combine different feedbacks to create fuzzers tuned to find unusual types of errors. For example, FuzzFactory enabled us to create a fuzzer that consistently found new memory usage "bombs" (e.g. a 21 byte input causing a 4GB memory allocations), even in heavily-fuzzed software like *libarchive*.

Mutating to Retain Structure: Covering Core Logic

Because CGF models inputs as byte-sequences (e.g. files, standard input), its mutations can easily corrupt high-level input structure. CGF's 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. Structurally unsound inputs like these cannot exercise the core logic of software.

In FairFuzz [10], 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 was able to achieve 10.6% higher branch coverage, a widely-used testing metric, than AFL on benchmark programs with highly nested structure. Thanks to an efficient way of computing this mask, FairFuzz continues to be popular in the fuzzing community. Users actively enquired about its integration into AFL++[15], and said that best practice involves using FairFuzz in a fuzzing deployment [16].

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.

Smart Control 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 [11], we brought together generator-based fuzzing and CGF. This combination relied on the core observation that small changes to the stream of random numbers used by the generator resulted in well-structured mutations in the generator-returned input. Effectively, Zest performed CGF, but with higher-level mutations on the inputs. As such, Zest was able to find bugs in the core logic stages of programs, like a logic error in the code optimization stages of the Google Closure compiler. Zest won a **Distinguished Artifact Award** at ISSTA'19.

By hijacking the random number steam, Zest adaptively controls the generator's probabilistic distribution to produce more valid inputs. RLCheck [12] has the same goal, but does this more explicitly. RLCheck replaces uses of the source of randomness in the generator with reinforcement learning agents, which make the ``choices' ' about the input instead of the pseudo-random backend. RLCheck trains these agents using only blackbox information about the program, 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 [13], 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

Automating Infrastructure for Useful Fuzzing

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. A major roadblock is the infrastructure required to use fuzzing tools and interpret fuzzing results. Currently, a large amount of human effort goes towards building this infrastructure, and this upfront effort lessens the perceived utility of fuzzing tools. I believe we can greatly reduce this effort.

All fuzzing tools require a suitable test driver: a side-effect free entry-point which exercises core logic of the system under test. Over 200 drivers synthesized by our FUDGE [14] tool were upstreamed into open-source libraries, and enabled 150 security-improving fixes. However, FUDGE required examples of library usage by client code, which may not exist for more general software systems. I want to explore how we can look at the context of a function within its own code base to generate drivers. Further, FUDGE could not be used to test software where the core functionality is deeply intertwined with the behavior of an external component. In this case, 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.

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. While there is promising work on input grammar inference, it is so-far restricted to particular classes of parsing programs, or lacks the generalization necessary for testing. I will continue my work on input grammar inference, and intend to 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 test input structure may vary. I will develop tools that help developers understand the input space captured by their specifications, and examine the program behaviors covered by this space.

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 an SMT solver or a compiler, where strange behavior on a particularly esoteric formula or program may pose neither a security risk nor ever be encountered by a user. I want to develop specification languages that enable developers to express their model of bug relevance. Then, I will explore how to use the languages to not only filter out bugs, but guide input generation towards those relevant bugs. I also think fuzzer-found bugs are also a compelling target for automatic, partial patch generation. I will investigate the applicability of current patching technologies in this domain.

Rethinking "Test-Input Generation"

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 not just in neural network architectures, but also in the process used to train these neural networks. 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 given 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. I will continue to work on new notions of "test-input generation" which can be just as useful for these systems as fuzz testing has grown to be.

References

[1] Krasner, H. (2018). *The Cost of Poor Quality Software in the US: A 2018 Report* (p. 5). Technical Report. Retrieved October 8, 2020, from https://www.it-cisq.org/the-cost-of-poor-quality-software-in-the-us-a-2018-report/The-Cost-of-Poor-Quality-Software-in-the-US-2018-Report.pdf

[2] OpenSSL. (2016). *OpenSSL Security Advisory [26 Sep 2016]*. Retrieved October 23, 2020, from https://www.openssl.org/news/secadv/20160926.txt

[3] OpenSSL. (2014). *OpenSSL Security Advisory [07 Apr 2014]*. Retrieved October 23, 2020, from https://www.openssl.org/news/secadv/20140407.txt

[4] OpenSSL. (2012). *Old 1.0.1 Releases.* Retrieved October 23, 2020, from https://www.openssl.org/source/old/1.0.1/

[5] Leung, M and Commisso, C. (2014). *Canadians filing taxes late due to 'Heartbleed' bug won't face penalties: CRA.* CTV News. Retrieved October 23, 2020, from https://www.ctvnews.ca/canada/canadians-filing-taxes-late-due-to-heartbleed-bug-won-t-face-penalties-cra-1.1767727

[6] Ogrodnki, I. (2014). *900 SINs stolen due to Heartbleed bug: Canada Revenue Agency.* Global News. Retrieved October 23, 2020, from https://globalnews.ca/news/1269168/900-sin-numbers-stolen-due-to-heartbleed-bug-canada-revenue-agency/

[7] OSS-Fuzz Maintainers. (2020). *OSS-Fuzz Trophies.* Retrieved October 27, 2020, from https://github.com/google/oss-fuzz

[8] 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 2018. ACM/SIGSOFT Distinguished Paper Award.

[9] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, Hayawardh Vijayakumar. FuzzFactory: *Domain-Specific Fuzzing with Waypoints.* OOPSLA 2019.

[10] Caroline Lemieux and Koushik Sen. *FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage.* In Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering, ASE 2018.

[11] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, Yves Le Traon. *Semantic Fuzzing with Zest.* In Proceedings of the 28th International Symposium on Software Testing and Analysis, ISSTA 2019. ACM/SIGSOFT Distinguished Artifact Award. Pdf.

[12] Sameer Reddy, Caroline Lemieux, Rohan Padhye, Koushik Sen. *Quickly Generating Diverse Valid Test Inputs with Reinforcement Learning.* In Proceedings of the 42nd International Conference on Software Engineering, ICSE 2020.

[13] Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, Ion Stoica. *AutoPandas: Neural-Backed Generators for Program Synthesis*. OOPSLA 2019.

[14] Domagoj Babic, Stefan Bucur, Yaohui Chen, Franjo Ivancic, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, Wei 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 (Industry Track).

[15] Denis Kasak (2019). *FairFuzz (afl-rb) integration.* Issue #18. Retrieved November 5, 2020, from https://github.com/AFLplusplus/AFLplusplus/issues/18

[16] van Hauser (2018). Comment on *Questions regarding fuzzing workflow using AFL++ / Best practices?* Issue #258. Retrieved November 5, 2020, from https://github.com/AFLplusplus/AFLplusplus/issues/258#issuecomment-599226036