Claire Le Goues – Research Statement

I often feel as though the possibilities for modern software are limitless. It's therefore a particular shame that software developers spend so much energy keeping current systems running. Put frankly, software is riddled with defects, new vulnerabilities come to light daily, and bugs cost the economy tens of billions of dollars annually. Researchers have developed techniques for bug prevention, prediction, detection, localization, and triage. However, *actually patching a defect* remains a predominantly manual (and thus expensive) process.

My research is among the first to tackle this problem directly: I devise methods to automatically, generically, and efficiently fix bugs in legacy software. I believe that the complexity of modern software is exciting because it allows us to adapt knowledge about other types of complex systems (e.g., biological immune systems) to software maintenance. At the same time, because my work concerns a practical problem space (maintaining real software), I strive to develop scalable methods and evaluate on real systems of indicative size. I also strongly value reproducible empirical work, and thus release my tools, data, and benchmarks to the community at large. These principles — studying software as a complex system, guided by real-world problems — underlie my previous work and motivate my future research plans.

GenProg: Automatic Program Repair

Imagine that a test case for a large, established codebase has failed. What's a novice developer to do? In many cases, the first step is to use analyses (like "printf debugging") to *localize* the failure to a smaller set of problematic lines. Step two is often to incrementally change the code, perhaps by reordering the buggy code or copying lines from other, likely correct, places. The modified program may be repeatedly re-tested to see if it approaches correctness. The effort required to manually inspect code and devise and test candidate solutions is one of the major reasons that software bugs are so expensive.

My current research focuses on GenProg, an algorithm that automatically repairs bugs in off-the-shelf programs by combining genetic programming with lightweight program analyses. At a high level, GenProg conducts an evolutionary search for a patch that corrects buggy behavior while maintaining required functionality. Throughout my work, I try to retain and simulate important benefits of human ingenuity:

- **Expressive power.** Just as humans are flexible, GenProg is designed to be *generic* and can repair many different types of bugs. This is important because new vulnerability classes regularly gain prominence.
- **Scalability.** Humans regularly repair bugs in large, legacy systems. GenProg similarly applies to programs of millions of lines of code and does not require special coding practices or annotations.

I exploit two key insights to achieve both expressivity and scalability.[1]

First, **external guarantees that a program implementation adheres to its specification provide useful insight into program behavior**. Such guarantees can take the forms of proof obligations or, more commonly, test suites, either developer-written or automatically-generated. These guarantees encode both the bug under repair as well as the other program functionality that a patch must maintain. This enables expressivity because test cases can flexibly capture a broad set of complex behaviors. Additionally, just as developers run their programs using "printf debugging" and other analyses to localize a fault, GenProg uses test cases to identify which statements are more likely to be associated with the bug. This reduces the search space of possible program changes, increasing scalability.

Second, **existing program behavior contains the seeds of many repairs**. GenProg reuses code from non-buggy portions of the program when generating candidate repairs. This approach leverages domain-specific programmer expertise encoded in the rest of the program, increasing the likelihood that GenProg will succeed; fundamentally, most developers program correctly most of the time (e.g., a forgotten bounds check is typically implemented correctly elsewhere in the same program). Code reuse also reduces the space of possible changes.

Real-world evaluation

In addition to being expressive and scalable, GenProg is designed to be *human-competitive* in terms of real-world cost and utility. This last concern is increasingly important, especially in light of some questions

[1] By contrast, most previous automatic repair techniques are limited in their expressivity because they only apply to a predefined set of bug types; most repair memory overflows exclusively. They are also often limited in their scalable legacy applicability because they problematically increase code size, run-time, or both.

commonly posed by industry practitioners: if I gave you the last 100 bugs from my project, how many could GenProg fix? How long would it take, and how much would it cost?

Answering these questions required innovations in both experimental and algorithmic design. We needed a large benchmark set, and we wanted the bugs in the set to be indicative of the ones developers deal with "in the wild." We leveraged version control and regression test suites in large, open-source projects to systematically find 105 reproducible bugs in eight C programs totaling 5.1 MLOC and including over 10,000 test cases. This evaluation is the largest currently available of its kind, and is two orders of magnitude larger than previous work in terms of either code or test suite size or defect count. We also improved the algorithm to run within the demanding resource constraints of a *commodity cloud computing* environment (which are becoming increasingly commonplace). This experimental approach required us to actually pay for the repair experiments, which in turn provided a direct and grounded way to estimate the real-world time and monetary cost of fixing a bug with GenProg.

In a controlled study, GenProg repaired over 50% of the 105 defects for $7.32 each, in 96 minutes, on average. We have confirmed that GenProg can repair at least 8 different bug classes, including standard engineering errors (infinite loops, incorrect output, segmentation faults, etc.) as well as security vulnerabilities taken from public reports. I consider these results strongly competitive in terms of both GenProg's expressive power and its time and monetary cost (by contrast, on average it takes weeks for developers to fix reported bugs in open-source software) and believe that they speak to the promise of automatic repair as a research area.

Given these automatically-produced repairs, quality is an important concern, especially with a technique that does not guarantee soundness. We qualitatively and quantitatively evaluated patch quality using indicative workloads and industrial practices for validating security-critical patches (e.g., following Microsoft). We found that the patches truly addressed the underlying defect (instead of just masking symptoms) while maintaining normal functionality. These results held even when inspection showed the repairs differed from the developer fixes. I found this especially interesting, because it suggests that software, like many biological systems, can be robust in the face of random mutations.

I am also excited about the possibilities cloud resources represent for software engineering research, especially in enabling grounded economic arguments. Another positive side-effect of our methodology is push-button reproducibility, and our publicly-available virtual machine images and bug packages allow other investigators to pay the same cloud prices and reproduce our results.

Recognition

My research has been well-received by both the software engineering and evolutionary computation communities. GenProg was the subject of an invited *Research Highlight* in the Communications of the ACM, with a readership of over 90,000. My publications have received four distinguished paper awards: two conference best papers, one workshop best short paper, and one distinguished article designation in a top journal. One of those papers also received the 2009 IFIP TC2 Manfred Paul Award for "excellence in software: theory and practice" (with a prize of 1,024€), which is awarded annually to one paper selected from one of several conferences. We won the gold and bronze awards in the 2009 and 2012 ACM SIGEVO "Humie" awards for human-competitive results produced by genetic and evolutionary computation (with $10,000 and $2,000 prizes, respectively). This critical validation provides additional confidence that the overall approach represents a promising research direction.

Future Work

My long-term research goal is to improve software correctness and reliability by understanding software as a complex adaptive system and leveraging the analogies between it and other such systems. I think one reason GenProg works is that software shares certain underlying principles with the biological systems on which evolutionary algorithms are based: program source code is robust to random mutations, patches are evolvable, and alternative pathways can provide the same program functionality. This line of thinking admits a number of opportunities for future research.

Patch security

I would like to leverage diversity to practically improve software security. Diversity is an important source of robustness in biological systems, providing protection against the spread of disease and alternative pathways that allow functionality to be maintained when one is disrupted. By contrast, most copies of an application on a given platform are exactly the same (leading to a *software monoculture*). I want to develop techniques for automatic *patch diversification* or hardening to counter the ability of hackers to reverse-engineer publicly-released security patches (and attacking unpatched systems). With colleagues, I am currently investigating patch diversification through cryptographic hashing or hoisting and code movement. We are especially interested in protocols that trade off scalability with mathematical guarantees: we seek sufficient computational complexity that a hacker must wait a provable amount of time before learning of the patch vector (giving users time to patch their systems). This line of research also admits investigations into software robustness and N-variant systems, and the types of guarantees they provide in terms of proactive diversity in the face of unknown defects.

Repair templates

I am interested in researching new ways for GenProg to construct patches to admit more natural repairs or repairs of new bug types. GenProg currently explores very coarse-grained, generic edits. Although this generality is good, at the same time, code and source control history often contain examples of repeated or common repairs for particular programs. Crashes in an image manipulation library for example, may often be patched by inserting a particular bounds check. This problem has parallels in computational models of vertebrate immune systems, which successfully balance the tradeoff between remembering past attacks with memory T-cells, and countering new ones with generalist T-cells.

I think program repair can be improved by mining and learning templates of candidate changes from program code and version control histories. I have successfully used information mined from such sources in previous work on specification mining; I believe that a similarly lightweight approach mining approach can be used to learn project-specific change patterns. I hypothesize that learning from the existing system in a grounded way can allay the tradeoff between bug-specific and general mutations.

Medical device safety

I am interested in improving medical device system safety by expanding the scope of existing fault and verification analyses. Unlike many desktop applications, safety critical medical systems are often assumed to operate in very controlled environments (e.g., the software in an insulin pump is not currently expected to interface with possibly-buggy device drivers). As a result, existing verification protocols can make very limiting assumptions about the user and the way the device is integrated into the patient's life. By contrast, the "software system" under study includes not only the software and hardware, but also the user, the decisions made by the user and medical team, and the environment in which the various components operate. As medical devices increase in complexity and move from the clinic into patient homes, we need new verification and validation techniques to mitigate threats to patient safety.

I have therefore begun collaborating with safety-critical systems researchers to extend existing procedures such as fault tree analysis to the entire patient-device interaction. Other researchers have begun initial investigations in this field by applying model checking to medical processes such as blood transfusion, with clinical success. I think this field is wide open, and I look forward to applying what the software engineering community knows about building and verifying safe software to other types of complex, human-centered systems.