Research Statement

Vijay Ganesh

My research focus is methods and tools for improving software reliability. There is a clear need for effective tools to augment manual software testing. There are, however, two key challenges to greater efficacy and adoption of software reliability tools. First, many tools do not effectively scale to large programs. Second, many methods are not completely automatic. To address these scalability and automation issues, I have designed novel techniques and implemented them in the following two tools.

- **STP Constraint Solver:** STP is an efficient and robust constraint solver [1], that has been used as the core reasoning component in dozens of automated bug finders, scalable program analysis, and model-checking tools [2]. STP has been shown to scale successfully in solving large constraint problems that arise when reasoning about the behavior of real-world programs. STP was the co-winner of the 2006 SMTCOMP competition for such solvers (bit-vector category), held annually at the Computer Aided Verification conference (CAV).
- **BuzzFuzz Automatic Bug Finder:** BuzzFuzz is a dynamic taint-based tool that automatically constructs error-revealing inputs for large C programs [3]. BuzzFuzz has successfully exposed several previously unknown errors in programs with complex input formats (e.g., movie players and document readers). BuzzFuzz requires only the source code of the program-under-test, and a set of test inputs on which the program behaves correctly. BuzzFuzz outputs a set of error-revealing inputs for the program-under-test.

  A key feature of BuzzFuzz is its ability to generate error-revealing inputs that make it past input validation code, and penetrate deep into the semantic core of large programs. Also, BuzzFuzz constructs error-revealing inputs where multiple disjoint regions of the input (e.g., multiple fields in a movie file) take specific values together to trigger an error.

In my research, I have focused on developing innovative algorithms, and implementing them as well-engineered, practical, and useable tools. I am motivated by the need to find effective solutions to key general sub-problems, such as constraint solving, that in turn enable solutions to other problems in software reliability.

The STP Constraint Solver

Many program analysis, formal methods, and automated reliability techniques are easily and effectively reduced to a constraint solving problem over logics such as Boolean logic. Usually, such a reduction, followed by invocation of an efficient external constraint solver, scales much better and is significantly faster than special-purpose algorithms for these methods. To better support this approach, I designed and implemented the STP constraint solver [1], an efficient, robust, and easy-to-use solver. STP employs several new algorithms, key among them is its use of *abstraction-refinement* based techniques.

The input language of STP is the quantifier-free theory of bit-vectors and arrays. The terms and formulas (constraints) in this theory are essentially expressions from standard programming languages such as C or Java. For example, an expression of type *char* in C would correspond to an 8 bit bit-vector term in STP. Arrays are the same as C/Java arrays, and are often used to represent memory. STP accepts a formula in this theory, and determines if it is satisfiable or not. If the input formula is satisfiable, STP generates a satisfying assignment. The largest constraint solved by STP has 2.12 million 32-bit variables, a large number of linear equalities, and array operations that are nested thousands of levels deep. STP solves this constraint in approximately 2 minutes on a 3.2GHz box.

I originally built STP in the context of a system for automatic generation of error-revealing inputs [4]. Since then STP has found much wider and more general application in dozens of research projects in program analysis, model checking, theorem proving, hardware verification, network configuration and as a backend for other solvers. The experience of these different research projects show that STP's efficiency and robustness make it a good choice for solving such constraints.

STP's architecture is *layered*, i.e., a set of novel algorithms pre-process the input constraints, followed by translation to Boolean SAT. STP employs two kinds of pre-processing steps. First, important fragments of STP's input theory can be solved in polynomial time, although the satisfiability problem for the entire theory is NP-complete. For example, I came up with a new polynomial time algorithm for systems of linear equations over bit-vector terms. Thus, constraints heavy with linear equations can be solved much faster using this linear solver. Second, STP employs an abstraction-refinement paradigm for other fragments of the input theory. At a high level, STP *abstracts* away parts of the input constraint that typically do not influence the answer. This step usually makes the problem more tractable. If the abstracted constraints give the correct answer, then STP terminates. Otherwise, STP *refines* its abstraction to get

a little closer to the original constraint, and repeats. In the worst case, STP may refine until it gets the whole original input constraint. In practice, STP rarely hits the worst-case.

Automatic Security Exploit Generation. Equations over regular expressions are generated by bug finding tools that automatically construct exploits like SQL injection and cross-site scripting attacks in PHP scripts. In this context, I worked on Hampi, a constraint solver for equations over regular expressions [5]. Hampi solves such equations to automatically generate security exploits.

Additional Constraint Solving Work. I have worked on the problem of efficiently solving constraints that are not in *clausal normal form* (CNF) [6]. This is an important problem since many reliability tools generate constraints that are typically not in CNF. However, many constraint solvers require the translation of such constraints into CNF. Such translations, though polynomial time, destroy the Boolean structure of the input formula. In this work [6], we leverage the inherent Boolean structure of the constraints for greater efficiency.

BuzzFuzz: Taint-based Directed Fuzzing

There is a clear need for tools that can automatically expose errors in programs. To this end, I built BuzzFuzz [3], a *dynamic taint*-based directed *fuzzing* tool, that constructs error-revealing inputs for large programs with complex input formats, such as movie players and document readers. Unlike many fuzzing techniques, that randomly mutate or generate inputs, BuzzFuzz is directed, i.e., it uses dynamic taint tracing to selectively modify few regions of the input that reach buggy code, and leaves the larger structure of the input intact.

BuzzFuzz is designed to generate tests that make it past input validation code, and penetrate deep into the semantic core of the program-under-test. Another key feature of BuzzFuzz is that it can construct error-revealing inputs where multiple disjoint regions of the input take specific values together to expose complex program errors.

In many large real-world programs, the vast majority of lethal errors (e.g., program crashes) occur when an input exercises some *corner case* behavior at the interface between the developer's code and library/system calls (e.g., overflow in pointer expressions passed to a library call). Furthermore, for any given error, only very few, often disjoint regions or *fields* of the error-revealing input actually exercise the buggy developer code at the code/library interface. Typically, such fields in corner case inputs contain extremal values.

Despite the need for corner case testing for such programs, developers often limit themselves to writing tests for the *common case* or expected behaviors. Even if the developers want to do corner case testing, the task of manually constructing such inputs with complex formats can be forbidding, since many distinct fields in the input may need to take specific values together to trigger errors in such programs. It may be hard to know such fields and values a priori. Hence, automatic techniques that expose such corner case errors can be very useful.

Leveraging the above mentioned features of lethal corner case errors at the code/library interface, I developed a technique which combines dynamic taint-based tracing with fuzzing to generate error-revealing inputs, and implemented it as the BuzzFuzz tool [3]. BuzzFuzz takes as input the following artifacts: 1) source of the program-under-test, and 2) a set of test cases or seed inputs on which the program behaves correctly. BuzzFuzz also needs a set of program locations, called *attackpoints*, chosen automatically to be code/library interface and/or (optionally) provided by the BuzzFuzz user. Any program point can be an attackpoint. However, library calls are a good choice, since there is evidence of many errors at the code/library interface. The output of BuzzFuzz is a set of tests that can crash the program-under-test. BuzzFuzz imposes minimal requirements on its users, and is completely automatic. BuzzFuzz has so far been used to generate several error-revealing inputs for two large C programs [3], namely Swfdec, a Flash movie player, and MuPDF, a PDF reader. Many of the bugs revealed were previously unknown.

BuzzFuzz works as follows: First, BuzzFuzz's compilation system instruments the source of the program-under-test for *dynamic taint tracing*. The instrumented program is then executed on one of the seed inputs provided. BuzzFuzz's dynamic taint tracing engine maintains a map between any value computed by the program-under-test, and the set of input bytes that influence that value. In particular, BuzzFuzz records the input bytes that influence attackpoints. In the second step, BuzzFuzz's *fuzzing* engine inserts extremal values at these fields, leaving the larger complex structure of the seed input intact. As mentioned before, standard fuzzing randomly generates or mutates program inputs, often with extremal values. However, the main drawback with this approach is that random mutation often produces ill-formed inputs that do not get past input validation code. BuzzFuzz's fuzzer, on the other hand, is directed, thanks to the identification of relevant input regions by the dynamic taint tracer. Thus, inputs generated by BuzzFuzz do not get rejected by input validation code, and manage to penetrate deeper into the program. In the third and final step, BuzzFuzz executes the original program on tests thus generated. If an error is exposed (in the form of a program crash), it is then reported back to the user.

Future Directions

I am very interested in continuing to work on both the scalability and automation challenges in making software more reliable, in particular, exploring the following ideas.

Machine Learning for Constraint Solving. It is well known that the performance of many Boolean SAT solvers is very sensitive to certain tunable parameters. In an ongoing collaborative work, I used machine learning techniques to train a classifier that maps select input-constraint features to locally optimal Boolean SAT solver parameter values. When the classifier receives an heretofore unseen input, it produces solver parameters such that the solver is typically significantly faster with these new parameters, than with default parameters. The classifier itself is simple and has negligible performance overhead. The preliminary results suggest rich research directions in using machine learning techniques for tuning constraint solver parameters for individual inputs.

Bug Finding in Numerical Programs. I would like to extend STP with a decision procedure for an interesting fragment of floating point arithmetic. This can be useful in the context of bug finding in numerical programs.

Dynamic Repair of Software. Can software systems be repaired (i.e., code or input corrected at runtime) in the face of an error-revealing input? Many kinds of software, such as Web servers or image viewers, can have security exploits associated with program crashes or other lethal errors. Furthermore, such software tend not to have exacting output requirements, e.g., an image viewer need not render an image perfectly. However, the image viewer clearly must not display exploitable behavior. In such scenarios, it may be better to learn the software's expected behavior in the form of *likely invariants* during testing over a large corpus of inputs, and enforce these likely invariants on the deployed version. The deployed version of the software can be instrumented, such that, the instrumentation detects any violation of learnt invariants, and enforces them if violations occur. I am currently involved in a project exploring exactly this idea.

Concolic Testing and Likely Invariants. In recent years there has been lot of work on the generation of likely invariants, e.g., the Daikon tool. These invariants are generated automatically by analyzing program traces obtained through program execution over a corpus of inputs. Likely invariants have been shown to be useful for many reliability techniques. However, the generality or *scope* of such invariants critically depends on inputs that exercise a large number of distinct program behaviors. Automatic generation of such *diverse* test inputs can improve the scope of likely invariants. In the reverse direction, likely invariants can be used to drive test case generation. Thus, there is a positive feedback loop between likely invariant learning and automated test generation.

However, for the feedback loop to make genuine progress the following is critical: Every new test case generated must *challenge* the likely invariants learnt so far, i.e., the new test case must exercise as yet untested program behavior that violates some likely invariants learnt so far.

One way to make genuine progress is to combine concolic test case generation [4] with likely invariant generation. At a high level, a concolic tester is a program that executes the program-under-test both concretely and symbolically on a given seed input, until some interesting program expression or invariant is reached (e.g., a pointer expression may not be null). The concrete execution serves the purpose of selecting a program path cheaply. The symbolic execution of such a path is converted into a constraint, called a *path constraint*. The path constraint is then conjoined with a query about the negation of interesting likely invariants, and fed to a constraint solver [1] for solution. In general, the solution, in terms of variables representing the input, is a test case that exercises untested behaviors in the program. Thus, in any given iteration of the feedback loop, likely invariants with greater generality may be generated, thanks to an input constructed in the previous iteration that exercises as yet untested behavior. The improved likely invariants thus generated may help construct tests that expose new behaviors.

BuzzFuzz: Taintolic Testing. The success of BuzzFuzz suggests other ways to automatically generate error-revealing inputs. For example, it is possible to use some symbolic path information with taint tracing, and solve the resulting constraints to get a range of values to fuzz seed inputs with. A whole range of different kinds of constraints can be obtained for constructing tests, with taint-tracing based program execution at one end, and concolic execution at the other.

References

[1] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *19th Conf. on Computer Aided Verification (CAV)*, 2007.

[2] Vijay Ganesh. A partial list of tools built using the STP constraint solver. `http://people.csail.mit.edu/vganesh/STP_files/stp-tools.html`, 2008.

[3] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *31st Conf. on Software Engineering (ICSE)*, 2009. To appear.

[4] Cristian Cadar, Vijay Ganesh, Peter Pawlowski, David Dill, and Dawson Engler. EXE: Automatically generating inputs of death. In *13th ACM Conf. on Computer and Communications Security (CCS)*, 2006.

[5] Adam Kiezun, Vijay Ganesh, and Michael Ernst. A solver for equations over regular expressions. In Preparation.

[6] Philippe Suter, Vijay Ganesh, and Viktor Kuncak. Non-clausal satisfiability modulo theories. In Preparation.