Richard Rutledge

Research Statement

My areas of research interests are software engineering and program analysis. Computer software is now ubiquitous to our daily lives. It monitors or controls a multitude of formerly mundane devices previously governed by simple mechanics. For example, multiple manufacturers now provide mass-market light bulbs equipped with Wi-Fi and sufficient computational ability to support a TCP/IP protocol stack and an embedded web server based configuration. As further evidence of the proliferation of software, consider the growth of GitHub public repositories from 12 million in 2015 to 128 million in 2020. Although such ubiquitous computing brings many advantages in convenience and efficiency to modern, everyday life, the software driving this innovation must be reliable. Fault localization of the mundane light bulb generally began with removing the questionable bulb from its socket and ended with a gentle shake. The analogous smart bulb should be as simple, without resorting to rebooting the device or resetting it to factory defaults. This is not a trivial requirement; essentially requiring the software failure rate to be much lower than the hardware failure rate. My research goals encompass automated software testing techniques and tools in support of the developers producing this expanding body of software.

My specific research interests lie in applying over-approximate techniques in symbolic execution to achieve scalable and usable results. Proposed by James King in 1976, symbolic execution (SymEx) is not new. However, applying SymEx to a realistic program was impractical at that time. The defining characteristics of the approach is to assign symbolic values to program inputs, execute the program statements while maintaining a set of internal states relative to the symbolic inputs, and maintain a path condition (PC) for each state as a conjunction in first-order logic. The approach begins with a single trivial state (empty) and path condition (true). When execution reaches a branch instruction, the state is cloned into two, with the PC of one conjoined with the branch condition and the PC of the other conjoined with the branch condition's negation. SymEx allows a systematic exploration of all paths through a program and constructs a first-order logic predicate whose solution yields an input that would reproduce that path. Multiple refinements have been published, such as dynamic and on-line SymEx (hereafter referred to as classic), to improve the tractability of the technique. However, these do not address SymEx's fundamental scalability issue; the total number of resulting states is exponential in the number of branch statements. The advent of efficient satisfiability modulo theory (SMT) solvers capable of solving the PC predicates allowed the implementation of SymEx and the closely related model checking tools for more than just experimental use. For example, they have since been used to verify Windows device drivers and microprocessor execution models. Nevertheless, the state explosion problem mentioned above together with solver theory limitations still limits the general applicability of SymEx to non-trivial, real-world programs.

In earlier work, I used progressively over-approximate symbolic execution to generate program state to execute all acyclic path segments within a program. In contrast to classic SymEx, over-approximate SymEx assigns symbolic values to internal program state in addition to program inputs. Combined with lazy initialization, this allows the symbolic execution to begin at arbitrary program points, not just the main entry point. Though much more scalable than classic SymEx, the approach

2091 Stanrich Ct, Marietta, GA
678.467.6600 • rrutledge@gatech.edu
https://cc.gatech.edu/~rrutledg/

over-approximates feasible behavior since it can explore program state that cannot be realized from the program input. In this prior work, we replayed the generated states on the targeted program fragments while recording the processor's electromagnetic emanations (EM). The recordings were then used to populate an EM model for the program. Finally, the completed model was used to infer the execution path of the program when given arbitrary inputs.

My latest published work applied over-approximate symbolic execution to differential testing. Given two versions of a program, the technique generates inputs for equivalent functions present in both versions of the program that can reach changed code. The inputs that execute a modified program statement are then replayed on both versions while periodically checking for a divergence in internal program state. Because the resulting behavioral differences can contain many intentional and unfeasible differences, the final step ranks them based upon heuristics to emphasize the likely unintentional ones.

My current work continues with approximate techniques in symbolic execution to improve scalability with respect to floating point arithmetic. Prior approaches have concentrated upon augmenting existing SMT language specification and solver theories with precise floating point support. But, this prior work has not gained much traction. For example, floating point support has existed for, but not been adopted into, a current state-of-the-art symbolic execution tool, KLEE. Instead, I propose to use approximated floating point results to reduce solver overhead and yield scalable performance.