Patrice Godefroid - Automating Software Testing Using Program Analysis (2008)

Tom Rochette <tom.rochette@coreteks.org>

November 2, 2024 — 36c8eb68

0.1 Context

0.2 Learned in this study

0.3 Things to explore

1 Overview

2 Notes

- Tree main ingredients:
  - Automatic
  - Scalable
  - Check many properties
- Any tool that can automatically check millions of lines of code against hundreds of coding rules is bound to find on average, say, one bug every thousand lines of code
- Given a program with a set of input parameters, automatically generate a set of input values that, upon execution, will exercise as many program statements as possible
- 3 tools developed at Microsoft using techniques from
  - Static program analysis (symbolic execution)
  - Dynamic analysis (testing and runtime instrumentation)
  - Model checking (systematic state-space exploration)
  - Automated constraint solving

2.1 Static versus dynamic test generation

- Static test generation consists of analyzing a program P statically by reading the program code and using symbolic execution techniques to simulate abstract program executions to attempt to compute inputs to drive P along specific execution paths or branches, without ever executing the program
- Cannot reason about constraints outside of the constraint solver's scope of reasoning (external method calls, calls to functions such as hash functions which are mathematically designed to prevent such reasoning)
- Dynamic test generation, consists of
  - executing the program P, starting with some given or random inputs
  - gathering symbolic constraints on inputs at conditional statements along the execution
  - using a constraint solver to infer variants of the previous input to steer the program's next execution toward an alternative program branch
- This process is repeated until a specific program statement is reached
- To solve the x == hash(y) problem, we can execute hash(y) with a given value and then assign x to this value

2.2 SAGE: White-box fuzz testing for security

2.2.1 SAGE architecture

- SAGE repeatedly performs four main tasks.
  - The tester executes the test program on a given input under a runtime checker looking for various kinds of runtime exceptions, such as hangs and memory access violation
  - The coverage collector collects instruction adresses executed during the run; instruction coverage is used as a heuristic to favor the expansion of executions with high new coverage
  - The tracer records a complete instruction-level trace of the run using the iDNA framework
  - Lastly, the symbolic executor replays the recorded execution, collects input-related constraints, and generates new inputs using the constraint solver Disolver

2.3 Pex: Automating unit testing for .NET

- Most fully automatic test-generation tools suffer from a common problem: they don't know when a test fails
- A new testing methodology: the parameterized unit test (PUT)
- Pex uses Z3 as its constraint solver

2.4 Yogi: Combining testing and static analysis

- The Yogi tool verifies properties specified by finite-state machines representing invalid program behaviors

3 See also

4 References

- Godefroid, Patrice, et al. "Automating software testing using program analysis." IEEE software 25.5 (2008): 30-37.