CMPSCI 630 Systems **Spring 2023**

Lecture 17

Lecturer: Emery Berger *Scribe: Allison Poh*

17.1 Correctness

- performance was last class (QoS - quality of service); correctness is this class
- correctness is always important (e.g., compilers, query optimizers, etc.,)
- 2 kinds of correctness: total correctness and partial correctness
  - you can argue there's a third type: no correctness

17.2 Total Correctness

- assumes there is a complete spec (usually written in logic)
- you are trying to show that the original program is behaviorally equivalent to the spec (i.e., $P \equiv spec$)
- what you're really trying to do is show that the original program is a subset of the spec (i.e., $P \subseteq spec$)
- that is, show that $\forall i : O(spec) \equiv O(P)$, where O is output
- considered strong correctness

17.3 Partial Correctness

- there's a set of implicit specs for programs, however partial correctness does not have a complete spec
- implicit specs may include:
  - program never dereferences null
  - no uninitialized variables
  - no use after free (C/C++ problem)
  - no buffer overflow
  - exceptions handled (e.g., no file not found errors)
- error checkers: tools that look for the errors in the implicit specs
- linters: pattern checkers that report bad programming practices (e.g., if (x = 0) vs. if (x == 0))
  - disadvantages of linters:
    - false positives (e.g., 1000s of warning messages but they're not all errors)
    - false negatives (e.g., bug is not part of pattern so not flagged)
- weaker than total correctness

17.4 Recall v. Precision

- recall: measures completeness of positive predictions (bug ⇒ report)
- precision: measures accuracy of positive predictions (report ⇒ bug)
- precision-recall tradeoff: people now prefer high precision in general (use to be the other way)

17.5 The Spec Problem

- writing specs is hard
- spec is a program, just in another language (so spec is long and complicated as well)
- is the spec correct? we don't know ... (who guards the guardians?)
- some say program is the spec! (i.e., there's no real spec)
- Bugs as Deviant Behavior paper says that, if a program acts weird sometimes, then there's probably a bug

17.6 Static Analysis

- static: looks at code
- Coverity is a static analysis tool:
  - collects factual information by running a program in abstract interpretation (i.e., execute symbolically)
  - infeasible paths: paths that no set of inputs will cause (static analysis does not know about these paths)
  - state space explosion: as the number of state variables increases, the size of the system state space increases exponentially
    - example: if, by static analysis, we know the bounds are $x \geq 12$ and $x \leq 130$, then there are many states between the start and end bounds, and thus we lose precision
    - another example: if there's a while(true) loop, we lose precision because we don't know how many iterations
    - therefore, there's a must analyses v. may analyses tradeoff
  - Coverity ranks errors based on state space, precision, etc., (giving a ranked list is better than giving 1mil errors and having someone try to figure out which are important)
- there are also static analysis tools that focus on security over correctness (called security-based analysis):
  - exploitable bug ⇒ $$$
  - classic bugs: use after free, tainted, and writing to a file (e.g., overwrite password file)

17.7 Dynamic Analysis

- dynamic: looks at execution
- Valgrind is a dynamic analysis tool:
  - shadow memory: version of the memory (identical or compact representation)
  - precision is super high (not much recall though)
  - if no error is found, then the particular execution path doesn't have an error (we do not say that there are no errors in the program)
  - generally do not generalize (although static analysis can generalize)
  - Valgrind uses memory check, cache grind, helgrind (race detection)

17.8 Testing

- if people don't write specs, will they write tests?
  - unit tests
  - assertions
  - regression tests
  - integration tests
  - end-to-end tests
- as the number of tests grows, so do the costs (time between commits grows)
- but people hate writing tests, and written tests aren't great (people can't anticipate the unanticipated; edge cases go untested)
- moral hazard of developers v. testers: if testers are there, then developers will write worst code (this turned out to be true!)
- coverage: need tests to increase coverage (e.g., line coverage, branch coverage, full path coverage)
- flaky tests: fail to produce the same outcome with each run (basically a heisenbug in tests)
- fuzzing:
  - discovered by accident (found that all Unix tools crashed on random inputs)
  - no one anticipated randomness!
  - it is easy to generate randomness, but it's not easy to generate random structured inputs (e.g., random JSONs)
  - we now have great fuzzers that have found tons of bugs
  - DART: concrete and symbolic analysis using randomness
  - AFL: American Fuzzy Lop; mutates inputs to see if any further converge using genetic algorithms (one of the most effective fuzzers)