Symbolic execution as search, and the rise of solvers

Search and SMT

- Symbolic execution is appealingly **simple and useful**, but **computationally expensive**
- We will see how the effective use of symbolic execution **boils down to a kind of search**
- And also take a moment to see how its feasibility at all has been aided by **the rise of SMT solvers**

Path explosion

- Usually can't run symbolic execution to exhaustion
  - Exponential in branching structure

```
int a = α, b = β, c = γ;    // symbolic
if (a) ... else ...;
if (b) ... else ...;
if (c) ... else ...;
```

    - Ex: 3 variables, 8 program paths
  - Loops on symbolic variables even worse

```
int a = α;    // symbolic
while (a) do ...;
…
```

    - Potentially 2^31 paths through loop!

Compared to static analysis

- Stepping back: Here is a **benefit of static analysis**
  - Static analysis will actually **terminate** even when considering **all possible program runs**
- It does this by approximating multiple loop executions, or branch conditions
  - Essentially **assumes all branches**, and **any number of loop iterations**, are **feasible**
- But can **lead to false alarms**, of course

Basic (symbolic) search

- Simplest ideas: algorithms 101
  - Depth-first search (*DFS*) — **worklist = stack**
  - Breadth-first search (*BFS*) — **worklist = queue**
- Potential drawbacks
  - **Not guided** by any higher-level knowledge
    - Probably a bad sign
  - **DFS could easily get stuck** in one part of the program
    - E.g., it could keep going around a loop over and over again
  - Of these two, BFS is a better choice
    - But more intrusive to implement (can't easily be concolic)

Search strategies

- Need to **prioritize search**
  - Try to steer search towards paths more likely to contain assertion failures
  - Only run for a certain length of time
    - So if we don't find a bug/vulnerability within time budget, too bad
- Think of **program execution as a DAG**
  - Nodes = program states
  - Edge(n_1,n_2) = can transition from state n_1 to state n_2
- We need a kind of **graph exploration algorithm**
  - At each step, pick among all possible paths

Randomness

- We don't know *a priori* which paths to take, so adding some randomness seems like a good idea
  - Idea 1: **pick next path to explore uniformly at random** (*Random Path*, or RP)
  - Idea 2: **randomly restart search** if haven't hit anything interesting in a while
  - Idea 3: **choose among equal priority paths at random**
    - All of these are good ideas, and randomness is very effective
- One drawback of **randomness**: reproducibility
  - Probably good to use pseudo-randomness based on seed, and then record which seed is picked
    - Or bugs may disappear (or reappear) on later runs

Coverage-guided heuristics

- **Idea**: Try to **visit statements we haven't seen before**

- Approach
  - Score of statement = # times it's been seen
  - Pick next statement to explore that has lowest score

- Why might this work?
  - Errors are often in hard-to-reach parts of the program
  - This strategy tries to reach everywhere.

- Why might this *not* work?
  - Maybe never be able to get to a statement if proper precondition not set up

Generational search

- Hybrid of **BFS and coverage-guided**
  - *Generation 0*: pick one program at random, run to completion
  - *Generation 1*: take paths from *gen 0;* negate one branch condition on a path to yield a new path prefix; find a solution for that prefix; then take the resulting path
    - Semi-randomly assigns to any variables not constrained by the prefix
  - *Generation n*: similar, but branching off *gen n-1*
- Also uses a coverage heuristic to pick priority

Combined search

- Run **multiple searches at the same time**
  - Alternate between them; e.g., Fitnext
- Idea: no one-size-fits-all solution
  - Depends on conditions needed to exhibit bug
  - So will be as good as “best” solution, within a constant factor for wasting time with other algorithms
  - Could potentially use different algorithms to reach different parts of the program

SMT solver performance

- SAT solvers are at core of SMT solvers
  - In theory, could reduce all SMT queries to SAT queries
  - In practice, SMT-level optimizations are critical
- Some example extensions/improvements
  - Simple identities ($x + 0 = x$, $x * 0 = 0$)
  - Theory of arrays ($read(x, write(42, x, A)) = 42$)
    - 42 = array index, A = array, x = element
  - Caching (memoize solver queries)
  - Remove useless variables
    - E.g., if trying to show path feasible, only the part of the path condition related to variables in guard are important

Popular SMT solvers

- **Z3** - developed at Microsoft Research
  - http://z3.codeplex.com/
- **Yices** - developed at SRI
  - http://yices.csl.sri.com/
- **STP** - developed by Vijay Ganesh, now @ Waterloo
  - https://sites.google.com/site/stpfastprover/
- **CVC3** - developed primarily at NYU
  - http://www.cs.nyu.edu/acsys/cvc3/

But: Path-based search limited

```
int counter = 0, values = 0;
for (i = 0; i<100; i++) {
  if (input[i] == 'B') {
    counter++;
    values += 2;
  }
}
assert(counter != 75);
```

- This program has 2^{100} possible execution paths.
- Hard to find the bug:
  - $\binom{100}{75} \approx 2^{78}$ paths reach buggy line of code
  - *Pr(finding bug)* = $2^{78} / 2^{100} = 2^{-22}$