Symbolic Execution
Symbolic Execution

- Testing works
  - But, each test only explores one possible execution
    - `assert(f(3) == 5)`
  - We hope test cases generalize, but no guarantees
- Symbolic execution generalizes testing
  - Allows unknown symbolic variables in evaluation
    - `y = α; assert(f(y) == 2*y-1);`
  - If execution path depends on unknown, conceptually fork symbolic executor
    - `int f(int x) { if (x > 0) then return 2*x - 1; else return 10; }`
Symbolic Execution Example

1. `int a = α, b = β, c = γ;`
2. `// symbolic`
3. `int x = 0, y = 0, z = 0;`
4. `if (a) {
   5. x = -2;
   6. }
7. if (b < 5) {
   8. if (!a && c) { y = 1; }
   9. z = 2;
10. }
11. assert(x+y+z!=3)
Insight

• Each symbolic execution path stands for *many* actual program runs
  ▪ In fact, exactly the set of runs whose concrete values satisfy the path condition
• Thus, we can cover a lot more of the program’s execution space than testing can
Early work on symbolic execution


• James C. King. Symbolic execution and program testing. CACM, 19(7):385–394, 1976. (most cited)


The Problem

• Computers were small (not much memory) and slow (not much processing power) then
  ▪ Apple’s iPad 2 is as fast as a Cray-2 from the 1980’s

• Symbolic execution is potentially extremely expensive
  ▪ Lots of possible program paths
  ▪ Need to query solver a lot to decide which paths are feasible, which assertions could be false
  ▪ Program state has many bits
Today

• Computers are much faster, memory is cheap
• There are very powerful SMT/SAT solvers today
  ▪ SMT = Satisfiability Modulo Theories = SAT++
  ▪ Can solve very large instances, very quickly
    - Lets us check assertions, prune infeasible paths
  ▪ I’ve used Z3, STP, and Yices
• Recent success: bug finding
  ▪ Heuristic search through space of possible executions
  ▪ Find really interesting bugs
Path explosion

• Usually can’t run symbolic execution to exhaustion
  ▪ Exponential in branching structure
    1. int a = α, b = β, c = γ; // symbolic
    2. if (a) ... else ...;
    3. if (b) ... else ...;
    4. if (c) ... else ...;
  - Ex: 3 variables, 8 program paths
  ▪ Loops on symbolic variables even worse
    1. int a = α; // symbolic
    2. while (a) do ...;
    3.
  - Potentially $2^{31}$ paths through loop!
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(n1,n2) = can transition from state n1 to state n2

• Then we need some kind of graph exploration strategy
  ▪ At each step, pick among all possible paths
Basic search

• Simplest ideas: algorithms 101
  ■ Depth-first search (DFS)
  ■ Breadth-first search (BFS)
  ■ Which of these did we implement?

• Potential drawbacks
  ■ Neither is 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
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, RP)
  - Idea 2: randomly restart search if haven’t hit anything interesting in a while
  - Idea 3: when have equal priority paths to explore, choose next one at random
    - All of these are good ideas, and randomness is very effective

• One drawback: reproducibility
  - Probably good to use pseudo-randomness based on seed, and then record which seed is picked
  - (More important for symbolic execution implementers than users)
Coverage-guided heuristics

• Idea: Try to visit statements we haven’t seen before

• Approach
  ▪ Score of statement = # times it’s been seen and how often
  ▪ 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

• KLEE = RP + coverage-guided
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 path prefix, and then take the resulting path
  - Note will semi-randomly assign to any variables not constrained by the path 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, which 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 and higher-level optimizations are critical

• Some examples
  - Simple identities \(x + 0 = x, \ x \times 0 = 0\)
  - Theory of arrays \(\text{read}(42, \text{write}(42, x, A)) = x\)
    - 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
Libraries and native code

• At some point, symbolic execution will reach the “edges” of the application
  - Library, system, or assembly code calls
• In some cases, could pull in that code also
  - E.g., pull in libc and symbolically execute it
  - But glibc is insanely complicated
    - Symbolic execution can easily get stuck in it
  - \( \Rightarrow \) pull in a simpler version of libc, e.g., newlib
    - libc versions for embedded systems tend to be simpler
• In other cases, need to make models of code
  - E.g., implement ramdisk to model kernel fs code
  - This is a lot of work!
Concolic execution

• Also called *dynamic symbolic execution*

• Instrument the program to do symbolic execution as the program runs
  ▪ I.e., shadow concrete program state with symbolic variables

• Explore one path, from start to completion, at a time
  ▪ Thus, always have a concrete underlying value to rely on
Concretization

• Concolic execution makes it really easy to concretize
  ▪ Replace symbolic variables with concrete values that satisfy the path condition
    - Always have these around in concolic execution

• So, could actually do system calls
  ▪ But we lose symbolic-ness at such calls

• And can handle cases when conditions too complex for SMT solver
  ▪ But can do the same in pure symbolic system
Resurgence of symbolic exection

• Two key systems that triggered revival of this topic:
  ▪ DART — Godefroid and Sen, PLDI 2005
    - Godefroid = model checking, formal systems background
  ▪ EXE — Cadar, Ganesh, Pawlowski, Dill, and Engler, CCS 2006
    - Ganesh and Dill = SMT solver called “STP” (used in implementation)
      - Theory of arrays
    - Cadar and Engler = systems
Recent successes

• SAGE
  ▪ Microsoft internal tool
  ▪ Symbolic execution to find bugs in file parsers
    - E.g., JPEG, DOCX, PPT, etc
  ▪ Cluster of $n$ machines continually running SAGE

• KLEE
  ▪ Open source symbolic executor
  ▪ Runs on top of LLVM
  ▪ Has found lots of problems in open-source software
Figure 6: Relative coverage difference between KLEE and the COREUTILS manual test suite, computed by subtracting the executable lines of code covered by manual tests ($L_{\text{man}}$) from KLEE tests ($L_{klee}$) and dividing by the total possible: $(L_{klee} - L_{\text{man}})/L_{\text{total}}$. Higher bars are better for KLEE, which beats manual testing on all but 9 applications, often significantly.
**KLEE: Coreutils crashes**

<table>
<thead>
<tr>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>paste -d`` abcdefghijklmnopqrstuvwxyz</td>
</tr>
<tr>
<td>pr -e t2.txt</td>
</tr>
<tr>
<td>tac -r t3.txt t3.txt</td>
</tr>
<tr>
<td>mkdir -Z a b</td>
</tr>
<tr>
<td>mkfifo -Z a b</td>
</tr>
<tr>
<td>mknod -Z a b p</td>
</tr>
<tr>
<td>md5sum -c t1.txt</td>
</tr>
<tr>
<td>ptx -F`` abcdefghijklmnopqrstuvwxyz</td>
</tr>
<tr>
<td>ptx x t4.txt</td>
</tr>
<tr>
<td>seq -f %0 1</td>
</tr>
</tbody>
</table>

**Figure 7:** KLEE-generated command lines and inputs (modified for readability) that cause program crashes in COREUTILS version 6.10 when run on Fedora Core 7 with SELinux on a Pentium machine.

Cadar, Dunbar, and Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs, OSDI 2008