Basic symbolic execution

Symbolic variables

- Extend the language's support for expressions e to include **symbolic variables,** representing *unknowns*

$$e ::= \alpha \mid n \mid X \mid e_0 + e_1 \mid e_0 \leq e_1 \mid e_0 \mathbin{\&\&} e_1 \mid \ldots$$

  - $n \in N$ = integers, $X \in Var$ = variables, $\alpha \in SymVar$

- Symbolic variables are **introduced** when **reading input**
  - Using `mmap`, `read`, `write`, `fgets`, etc.
  - So if a bug is found, we can recover an input that reproduces the bug when the program is run normally

Symbolic expressions

- We make (or modify) a language interpreter to be able to **compute symbolically**
  - Normally, a program's variables contain *values*
  - Now they can also contain *symbolic expressions*
    - Which are expressions containing symbolic variables
- Example normal values:
  - 5, "hello"
- Example symbolic expressions:
  - α+5, "hello"+α, a[α+β+2]

Straight-line execution

```
→ x = read();
  y = 5 + x;
  z = 7 + y;
  a[z] = 1;
```

Concrete Memory

x ↦ 5

y ↦ 10

z ↦ 17

a ↦ {0,0,0,0}

Overrun!

Symbolic Memory

x ↦ α

y ↦ $5+\alpha$

z ↦ $12+\alpha$

a ↦ {0,0,0,0}

Possible overrun!

We'll explain arrays shortly

Path condition

- Program control can be affected by symbolic values

```
1  x = read();
2  if (x>5) {
3    y = 6;
4    if (x<10)
5      y = 5;
6  } else y = 0;
```

- We represent the influence of symbolic values on the current path using a **path condition π**
  - Line 3 reached when $\alpha>5$
  - Line 5 reached when $\alpha>5$ and $\alpha<10$
  - Line 6 reached when $\alpha\leq5$

Path feasibility

- Whether a path is feasible is tantamount to a path condition being **satisfiable**

```
1  x = read();
2  if (x>5) {
3    y = 6;
4    if (x<3)
5      y = 5;
6  } else y = 0;
```

$\pi = \alpha > 5$

$\pi = \alpha > 5 \wedge \alpha < 3$ *Not satisfiable!*

$\pi = \alpha \leq 5$

- **Solution** to path constraints **can be used as inputs** to a concrete test case that will execute that path
  - Solution to reach line 3: $\alpha = 6$
  - Solution to reach line 6: $\alpha = 2$

Paths and assertions

- Assertions, like array bounds checks, are conditionals

```
1 x = read();
2 y = 5 + x;
3 z = 7 + y;
4 if(z < 0)
5    abort();
6 if(z >= 4);
7    abort();
8 a[z] = 1;
```

$\pi = \text{true}$

$\pi = \text{true}$

$\pi = \text{true}$

$\pi = \text{true}$

$\pi = 12+\alpha<0$

$\pi = \neg(12+\alpha<0)$

$\pi = \neg(12+\alpha<0) \land 12+\alpha\geq 4$

$\pi = \neg(12+\alpha<0) \land \neg(12+\alpha\geq 4)$

- So, if either lines 5 or lines 7 are reachable (i.e., the paths reaching them are feasible), we have found an out-of-bounds access

Forking execution

- Symbolic executors can **fork** at branching points
  - Happens when there are **solutions** to both the **path condition *and* its negation**
- How to systematically explore both directions?
  - **Check feasibility during execution** and queue feasible path (condition)s for later consideration
  - **Concolic execution**: run the program (concretely) to completion, then generate new input by changing the path condition

Execution algorithm

1. Create initial task
   - $pc = 0$, $\pi = \varnothing$, $\sigma = \varnothing$
2. Add task (pc, π, σ) onto worklist
3. While (list is not empty)
   3a. pull some task (pc, π, σ) from worklist
   3b. execute. if it potentially forks at (pc_0, π_0, σ_0)
   3ba. add task $(pc_1, (\pi_0 \wedge p), \sigma_0)$ if $\pi_0 \wedge p$ feasible
   3bb. add task $(pc_2, (\pi_0 \wedge \neg p), \sigma_0)$ if $\pi_0 \wedge \neg p$ feasible

```
pc0   if (p) {
pc1       …
pc2 } else { …
```

Note: Libraries, 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
  - So, pull in a simpler version of libc, e.g., newlib
- In other cases, need to make models of code
  - E.g., implement ramdisk to model kernel fs code

Concolic execution

- Also called *dynamic symbolic execution*
- **Instrument the program** to do symbolic execution as the program runs
  - Shadow concrete program state with symbolic variables
    - Initial concrete state determines initial path
      - could be randomly generated
  - **Keep shadow path condition**
- **Explore one path at a time**, start to finish
  - The next path can be determined by
    - negating some element of the last path condition, and
    - solving for it, to produce concrete inputs for the next test
  - 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**