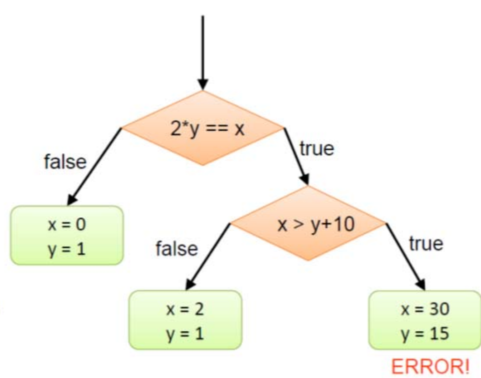
Introduction to Symbolic Execution

Classic Symbolic Execution

```
int twice (int v) {
        return 2*v;
}

void testme (int x, int y) {
        z = twice (y);
        if (z == x) {
                if (x > y+10)
                        ERROR;
                }
        }
}

/* simple driver exercising testme() with
int main() {
        x = sym_input();
        y = sym_input();
        testme(x, y);
        return 0;
}
```



Problem 1: Infinite execution path

```
void testme_inf () {
        int sum = 0;
        int N = sym_input();
        while (N > 0) {
            sum = sum + N;
            N = sym_input();
        }
}
```

Figure 3. Simple example to illustrate infinite number of execution paths.

Problem 2: Unsolvable formulas

```
int twice (int v) {
        return (v*v) % 50;
}
```

Figure 4. Simple modification of the example in Figure 1. The function twice now performs some non-linear computation.

Problem 3: symbolic modeling

- External function calls and system calls are hard to model
- For efficiency, symbolic execution systems often model libc function calls.
  - File system related
  - String operations

Concolic Testing

- Performs symbolic execution dynamically, while the program is executed on some concrete input values.
- Generate some random input: x=22, y=7 and execute the program both concretely and symbolically
- The concrete execution take the “else” branch on Line 7 and the symbolic execution generates the path constraint x != 2y
- Negates a conjunct in the path constraint and solves x==2y and get a new test input x=2, y=1
- Test the program with the new input

Concolic Testing: What is the benefit?

- Solve complex formulas
  - x == (y*y) mod 50, unsolvable if both x and y are symbolic
  - if we concretize y to its concrete value, now solvable
  - Angr does this!
- External library call and system call
  - E.g., fd = open(filename)
  - Set filename to its concrete value "/tmp/abc.txt"
  - Execute the system call concretely
  - Set fd to be concrete after the system call return
  - High level idea of S2E!

Online or Offline?

- Online
  - When encounter a new symbolic branch, solve predicates for both directions
  - If both directions are feasible, fork the execution state (concrete and symbolic)
  - KLEE and S2E take this approach
- Offline (or trace-based)
  - Choose an input and execute the program, collect execution trace
  - Compute path constraints from the trace
  - Negate each conjunct, solve the new path constraint, and get a new input
  - Given the new input to the program and execution again
  - BitBlaze, SAGE and BAP take this approach

Online and Offline: Pros and Cons

	Online	Offline
Efficiency	High	Low
Implementation difficulty	High	Low
Symbolic State	Quickly exploded	No state management

How to execute symbolically?

- Trace based
  - BAP: Use Pintrace to collect execution trace, and then convert the trace into BAP IL (derived from VEX)
  - BitBlaze: Use tracecap plugin to collect execution trace, Convert the trace into Vine IR
  - Low efficiency and possibly very long trace!!
- Dynamic Instrumentation
  - S2E:
    - Run in QEMU with two machines (concrete and symbolic) simultaneously
    - Convert TCG IR to LLVM Bitcode
  - KLEE:
    - Compile C/C++ into LLVM Bitcode
    - Add instrumentation on LLVM Bitcode

How to execute symbolically?

- Complete Interpretation or Simulation
  - Interpret binary execution and add symbolic execution
  - Angr: convert each instruction into VEX, and interpret each VEX statement in Python
  - Pros: full control, easy to implement
  - Cons: low efficiency by nature. All instructions must be interpreted, no matter if symbolic variables are involved or not. For long execution trace, it will take very long time!!

Research Question: how to speed up symbolic execution?

- Most of instructions just need to be executed concretely. We like to execute them natively if possible
- Only a few instructions need to be executed symbolically.
- How to detect if an instruction needs to be executed symbolically
- How to switch between concrete and symbolic execution quickly?

How to deal with state explosion?

- State merging and pruning
- Targeted search
  - Find some interesting target
  - At each branch point, favor the direction closer to the target
  - A fitness function is chosen
- Combine online and concrete re-execution
  - E.g. Mayhem
- Combine symbolic execution with evolutionary fuzzing
  - E.g., Driller

Mayhem: Combine online symbolic execution and concrete re-execution

- Perform online symbolic execution in BFS fashion
- When it reaches a limit, store the symbolic states on disk
- Pick one state to continue. To do so, solve the path constraint, and use it as input to re-execute the program up to the current state
- Start to perform online execution from this state

Driller: Combine symbolic execution with evolutionary fuzzing

- Evolutionary fuzzing drives the path selection
  - AFL
  - Share the seeds with symbolic execution
- Symbolic execution takes each seed and perform a very localized path exploration
  - Angr
  - Generate new inputs and feed them back to the fuzzer
- Problems
  - Most of these new inputs will be unfortunately dropped
  - Some seeds lead to very long trace, take very long time to execute in Angr, and impossible to solve

Path predicate may be over-constrained

- In Dynamic Symbolic Execution,
  - A constraint is computed per execution path
  - A different path may still reach the same point
  - It means some conditions are not necessary
- We can use Max-SMT
  - Specify which clause is hard and which is soft
  - Max-SMT may throw away soft constraint to find a solution

Symbolic execution: A search problem

- BFS, DFS, random, heuristic, etc.
- By nature, similar to Go and Chess
- Can we make an AlphaGo for symbolic execution?