NC STATE UNIVERSITY

CSC 574
Computer and Network Security

Symbolic Execution

Alexandros Kapravelos
kapravelos@ncsu.edu

Let's find some bugs

- We have a potentially vulnerable program
- The program has some inputs which can be controlled by the attacker
- What should we do as developers?
  - Add checks (assertions)
  - Write tests
  - Make sure the checks do not fail

- Is this enough?

Concrete Execution

```
void foo(int x, int y) {            x=0, y=0
    int z = 0;
    if (x>y) {                      False
        z = x;
    } else {
        z = y;                      z = 0
    }
    if (z < x) {                    False
        assert false;               // not reached
    }
}
```

Concrete Execution

`void foo(int x, int y) {`	x=1, y=0
`    int z = 0;`	
`    if (x>y) {`	True
`        z = x;`	z = 1
`    } else {`	
`        z = y;`	
`    }`	
`    if (z < x) {`	False
`        assert false;`	// not reached
`    }`	
`}`	

Pros/Cons

- Testing intended functionality
- Testing for known bugs
- Unintended functionality
- Unknown bugs
- Complete coverage

Can we automate this part?

Symbolic Execution

```
void foo(int x, int y) {
    int z = 0;
    if (x>y) {
        z = x;
    } else {
        z = y;
    }
    if (z < x) {
        assert false;
    }
}
```

x=α, y=β

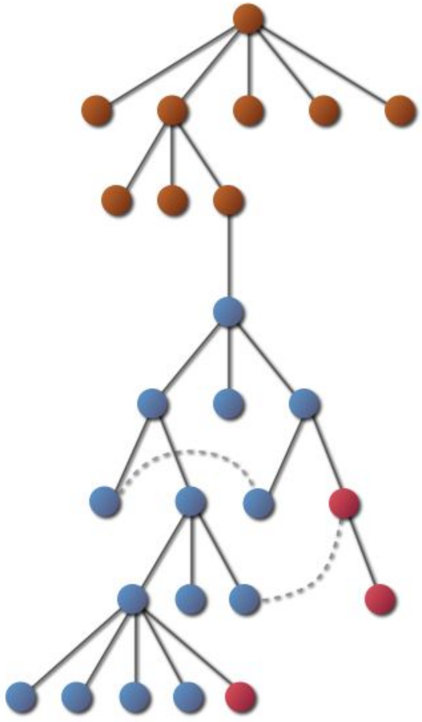
z = α, α > β

z = β, α <= β

1. α < α -> False
2. β < α, α <= β, False

Feasible and Infeasible Paths

- A path is a particular route in the control-flow graph of the program
- A **feasible** path is the path covered for a particular input
- An **infeasible** path is the path that no input can cover



Infeasible Paths

- Dead code => infeasible path
- Infeasible path !=> dead code

- It is normal in a large program to have a large of infeasible paths

- This makes automatic testing based on the input to the program incredibly hard

Constrains

- $\alpha > b \wedge \alpha + \beta <= 10$

- α, β are called **free variables**
- Solution: a set of variable assignments that makes the constraint satisfiable
- $\{\alpha = 3, \beta = 2\}$ is a solution
- $\{\alpha = 6, \beta = 5\}$ is not a solution

- Decision procedure: is the constraint satisfiable?
- Constraint solver: if is satisfiable, find assignments
- Undecidable problem

Symbolic Execution

- Execute the program differently, “symbols” as input
- Take all feasible paths
- Program state is different:
  - No stack/heap
  - Symbolic values for memory locations
  - Path condition
- Path condition: input constraints so that a certain path is feasible
- A solution to a path condition is a test input that covers the desired path

History of Symbolic Execution

James C. King

Symbolic execution and program testing

Communications of the ACM 19, 7

(July 1976)

Why are we talking about it now?

- Computation intensive
  - Too many paths
  - Program state grows a lot
  - Constraint solver is computationally expensive, but we need to identify the feasible paths

- Powerful computers
- Better constraint solvers

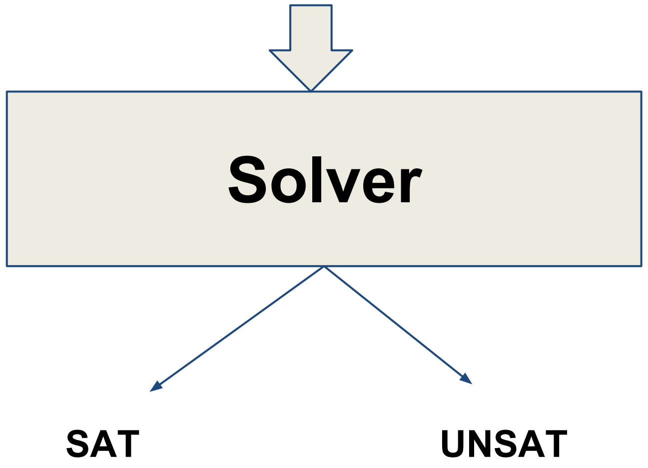
Symbolic Execution Tools

- KLEE
  - Open source symbolic executor
  - Runs on top of LLVM
  - Has found lots of problems in open-source software
- 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

Constraint Solver

- Boolean **SAT**isfiability Problem
- Find values that satisfy a boolean formula
- NP-Complete

$$(I1 \lor I2 \lor x2) \land (\neg x2 \lor I3 \lor x3)$$



SMT Solvers

- Satisfiability modulo theories
- SAT, but with binary variables replaced by predicates over a suitable set of non-binary variables

3x + 2y - z >= 4

$$(\sin(x)^3 = \cos(\log(y) \cdot x) \vee b \vee -x^2 \geq 2.3y) \wedge \left(\neg b \vee y < -34.4 \vee \exp(x) > \frac{y}{x}\right)$$

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/

Forking Execution

- What to do when we reach a branching point?
  - Follow both paths (condition satisfied and negation)
- State explosion *really* fast (exponential)
  - Loops on symbolic variables are problematic
- How can we do this more efficiently?
  - Prune paths by following only feasible ones
  - Concolic execution: run the program concretely and assist the execution with symbolic execution by changing the path conditions

Static analysis

- It will terminate, even if the whole program is taken into account
- Approximation is the key
  - Let's assume every path is feasible
- False alarms
- Less accurate

Symbolic search

- We have to decide on a strategy
  - Depth-first search (DFS)
  - Breadth-first search (BFS)

- Potential drawbacks
  - No smart choices
  - DFS can get easily stuck in one part of the program
    - Literally on a loop
  - BFS is a better choice
    - Harder to implement (think about concolic execution)

Search strategies

- Focus on the paths that matter
  - Assertion failures
  - Time bound

- Improve coverage
  - Program execution as a DAG
    - Nodes = program states
    - Edge(n1, n2) = can transition from n1 to state n2
  - Graph exploration algorithm

Randomness

- In the beginning we know nothing, how do we start?
- Ideas
  - Pick next path at random
  - Randomly restart search
  - Choose randomly among equal priority paths
- But then how do we reproduce our analysis?
  - Pseudo-randomness
  - Record the seed
  - Otherwise bugs can disappear on reruns

Coverage-guided heuristics

- Let's visit statements that we haven't seen before
- Approach
  - Score of statement = # visits
  - Pick the next statement with the lowest score
- Pros
  - Errors are often in hard-to-reach parts of the program
  - This strategy tries to reach everywhere.
- Cons
  - 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
  - Generation n: similar, but branching off gen n-1
- Also uses a coverage heuristic to pick priority

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);
```

- 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) = $2^{78} / 2^{100} = 2^{-22}$

Libraries and native code

- Execution of a program is not solely contained on the programs code
  - Libraries, system calls, assembly code
- We could extend the symbolic execution to those parts
  - Pull in the library and symbolically execute it
  - If library is complicated, then our program state will grow too large
  - Replace the library with a simpler version (libc -> newlib)
- Model the code of the external dependencies

Concolic Execution

- Dynamic symbolic execution
- Concrete execution of the program with assistance by symbolic execution

- Instrument the program
  - Keep a shadow state with symbolic variables
  - Start with a concrete execution that sets an initial path

- Follow one path and use symbolic execution to determine the next one
  - Negate a condition
  - Inputs are concrete values

Concretization

- Use symbolic execution as guidance
  - But replace symbolic variables with concrete values that satisfy the path condition
- This way the program is actually executed
  - Abstract parts that are not in the code (system calls)
  - No symbolic-ness at such calls (we lose information)
- Very useful when conditions get too complex for SMT solver

Conclusion

- Symbolic execution is very powerful and productive
- Not very practical as programs grow large
  - Limited by the power of the constraint solver
  - Bound by the infeasible paths number
- Promising research area!