PennState

CSE543- Computer Security

Module: Security Analysis Techniques

Asst. Prof. Syed Rafiul Hussain

Computer Science and Engineering Department
Pennsylvania State University

Security Analysis Techniques

- Testing/Fuzzing
- Static Analysis (Already covered in software vulnerability)
- Symbolic Execution
- Concolic Execution
- Formal Verification

Testing

- Testing: the process of running a program on a set of test cases and comparing the actual results with expected results (according to the specification).
  - For the implementation of a factorial function, test cases could be {0, 1, 5, 10}. What is missing?
  - Can it guarantee correctness?
    - Correctness: For all possible values of n, your factorial program will provide correct output.
    - Verification: High cost!

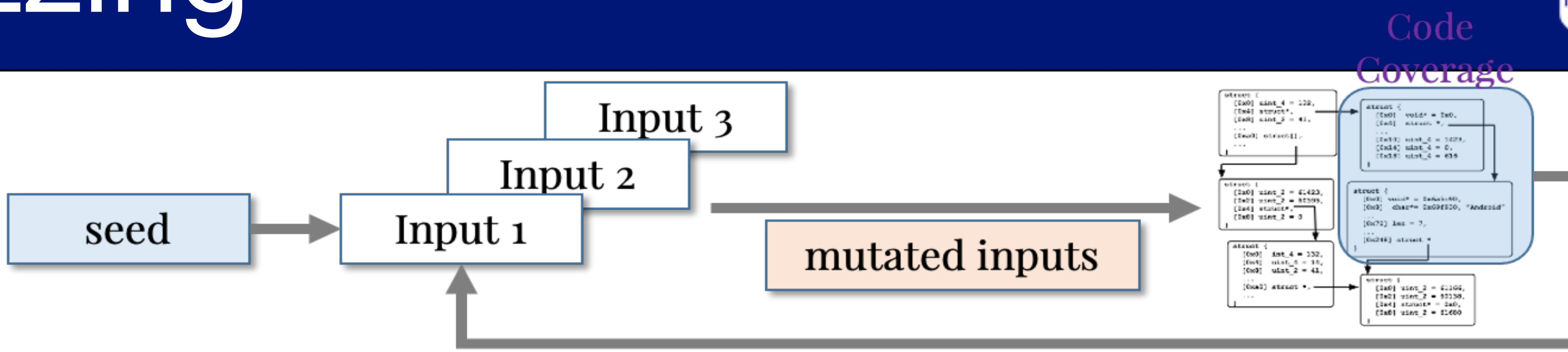
Fuzz Testing

Fuzz Testing

- Idea proposed by Bart Miller at Wisconsin in 1988 after experiencing an unusual crash while accessing a Unix utility remotely

```
format.c (line 276):

... while (lastc != '\n') { //reading line

     rdc(); }


input.c (line 27):
   rdc() {
       do { //reading words
         readchar(); } while (lastc == ' ' || lastc == '\t');
return (lastc);
}
```

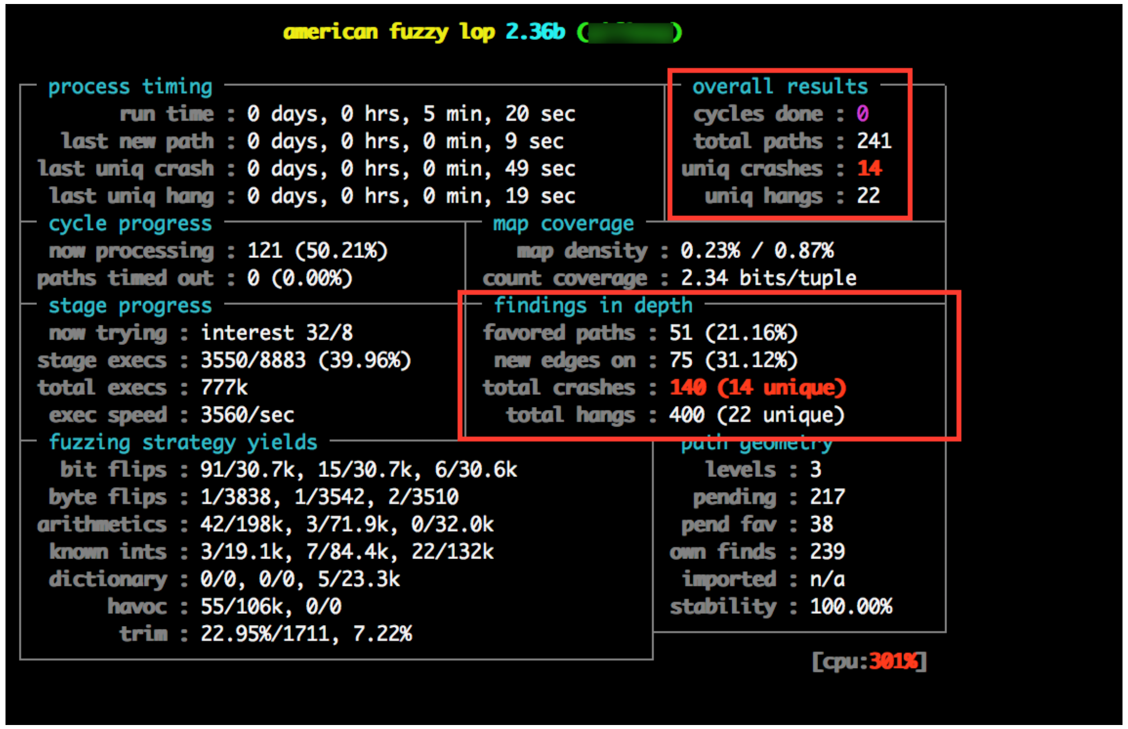
Fuzzing



- Fuzzing is an automated form of testing that runs code on (semi) random and (abnormal) input.
  - Black Box (based on specification): e.g., input is non-negative
  - White Box (source/binary): e.g., if(x>y and y>z) then … else .
- Mutation-based fuzzing generates test cases by mutating existing test cases.
- Generation-based fuzzing generates test cases based on a model of the input (i.e., a specification). It generates inputs "from scratch" rather than using an initial input and mutating.
- Any inputs that crash the program are recorded.
  - Crashes are then sorted, reduced, and bugs are extracted. Bugs are then analyzed individually (is it a security vulnerability?).

American Fuzzy Lop (AFL)

- American Fuzzy Lop is a security-oriented fuzzer hat employs a novel type of compile time instrumentation and genetic algorithms to automatically discover clean, interesting test cases that trigger new internal states in the targeted binary.
- Low overhead and low initialization cost (i.e., fast forward to interesting points in binary before you start fuzzing).
- Different different fuzzing strategies and switches on demand.



Static Analysis

PENNSTATE

- Limitation of dynamic testing:
  - We cannot find all vulnerabilities in a program
- *Can we build a technique that identifies *all* vulnerabilities?*
  - *Turns out that we can: static analysis*
    - Explore all possible executions of a program
      - All possible inputs
      - All possible states
  - *But, it has its own major limitation*
    - *Can identify many false positives (not actual vulnerabilitiies)*
  - *Can be effective when used carefully*

Static Analysis

- Provides an approximation of behavior
- "Run in the aggregate"
  - Rather than executing on ordinary states
  - Finite-sized descriptors representing a collection of states
- "Run in non-standard way"
  - Run in fragments
  - Stitch them together to cover all paths
- Various properties of programs can be tracked
- Control flow, Data flow, Types
- Which ones will expose which vulnerabilities

Control Flow Analysis

PENNSTATE 1855

Can we detect code with no return check? From original Miller fuzzing paper.

```
format.c (line 276):
while (lastc != '\n')
{ //reading line
 rdc();
}
```

```
input.c (line 27):
rdc() {
 do { //reading words
     readchar(); }
while (lastc == ' ' || lastc
== '\t');
   return (lastc);
}
```

- Compute the control flow of a program, i.e., possible execution paths.
- To find an execution path that does not check the return value of a function
  - ❑ That is actually run by the program
  - ❑ How do we do this? Control Flow Analysis

Static vs. Dynamic

- Dynamic
  - Depends on concrete inputs
  - Must run the program
  - Impractical to run all possible executions in most cases
- Static
  - Overapproximates possible input values (sound)
  - Assesses all possible runs of the program at once
  - Setting up static analysis is somewhat of an art form
- Is there something that combines best of both?
  - Can't quite achieve all these, but can come closer

Symbolic Execution

- Symbolic execution is a method for emulating the execution of a program to learn constraints
  - Assign variables to symbolic values instead of concrete values
  - Symbolic execution tells you what values are possible for symbolic variables at any particular point in your program
- Like dynamic analysis (fuzzing) in that the program is executed in a way – albeit on symbolic inputs
- Like static analysis in that one start of the program tells you what values may reach a particular state

Background: SAT

PENNSTATE

1855

Given a propositional formula in CNF, find if there exists an assignment to Boolean variables that makes the formula true:

literals

clauses

$$\omega_1 = (b \vee c)$$

$$\omega_2 = (\neg a \vee \neg d)$$

$$\omega_3 = (\neg b \vee d)$$

$$\varphi = \omega_1 \wedge \omega_2 \wedge \omega_3$$

$$A = \{a=0, b=1, c=0, d=1\}$$

SATisfying assignment!

Background: SMT

SMT: Satisfiability Modulo Theories

Input: a first-order formula φ over background theory

Output: is φ satisfiable?

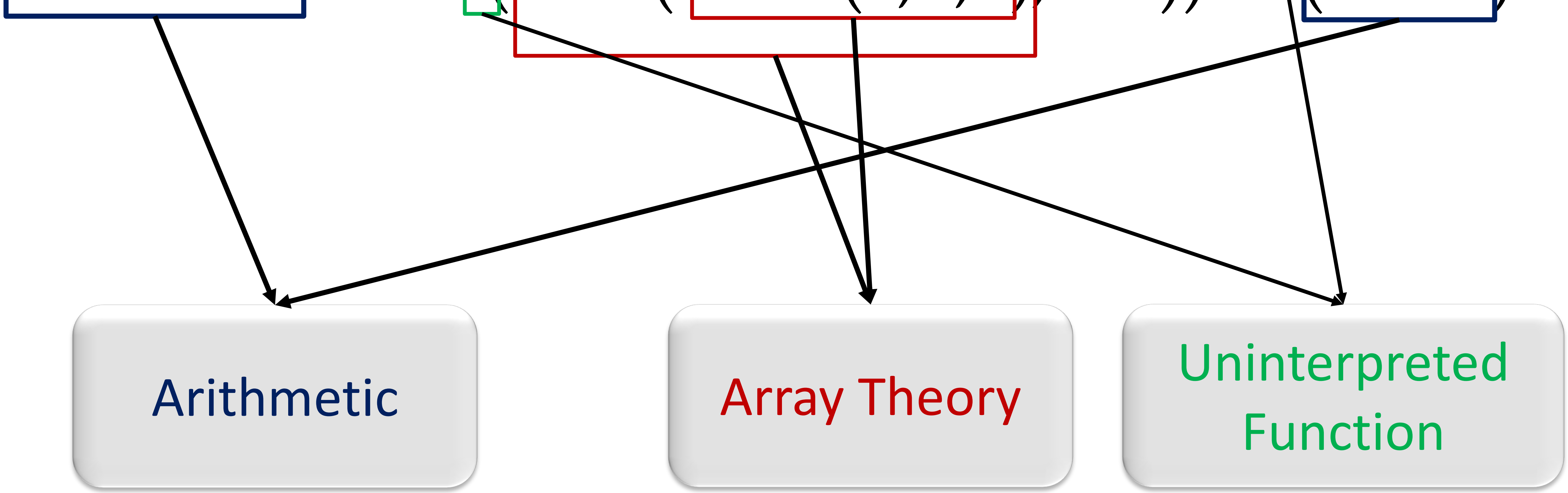
- does φ have a model?
- Is there a refutation of φ = proof of ¬φ?

For most SMT solvers: φ is a ground formula

- Background theories: Arithmetic, Arrays, Bit-vectors, Algebraic Datatypes
- Most SMT solvers support simple first-order sorts

Background: SMT

- $b + 2 = c$ and $f(read(write(a,b,3), c-2)) \neq f(c-b+1)$



Example SMT Solving

- b + 2 = c and f(read(write(a,b,3), c-2)) ≠ f(c-b+1)

[Substituting c by b+2]

- b + 2 = c and f(read(write(a,b,3), b+2-2)) ≠ f(b+2-b+1)

[Arithmetic simplification]

- b + 2 = c and f(read(write(a,b,3), b)) ≠ f(3)

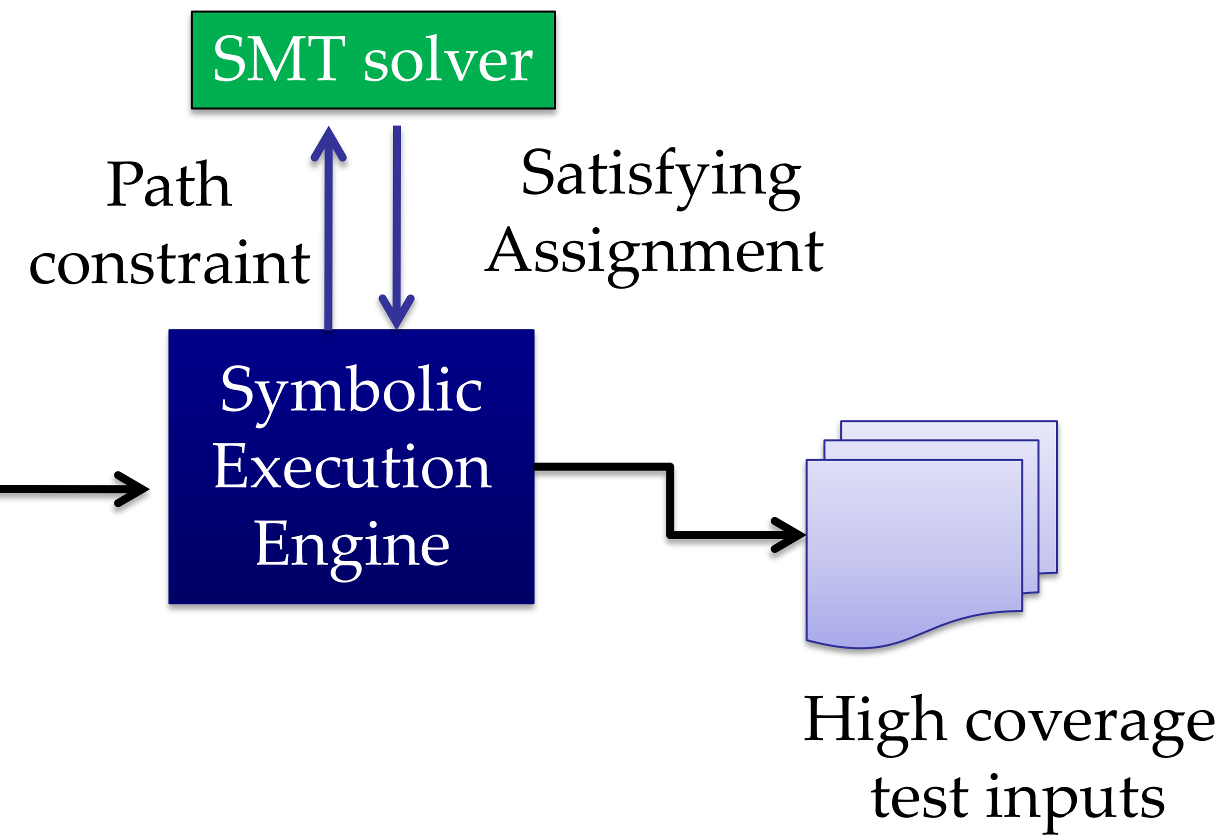
[Applying array theory axiom– 15

forall a,i,v:read(write(a,i,v), i) = v]

- b+2 = c and f(3) ≠ f(3) [NOT SATISFIABLE]

Symbolic Execution

```
Void func(int x, int
   y){
   int z = 2 * y;
   if(z == x){
    if (x > y + 10)
            ERROR
   }
}
int main(){
   int x = sym_input();
   int y = sym_input();
   func(x, y);
   return 0;
}
```



Symbolic Execution

- Execute the program with symbolic valued inputs **(Goal: good path coverage)**
- Represents *equivalence class of inputs* with first order logic formulas **(path constraints)**
- One path constraint abstractly represent all inputs that induces the program execution to go down a specific path
- Solve the path constraint to obtain one representative input that exercises the program to go down that specific path

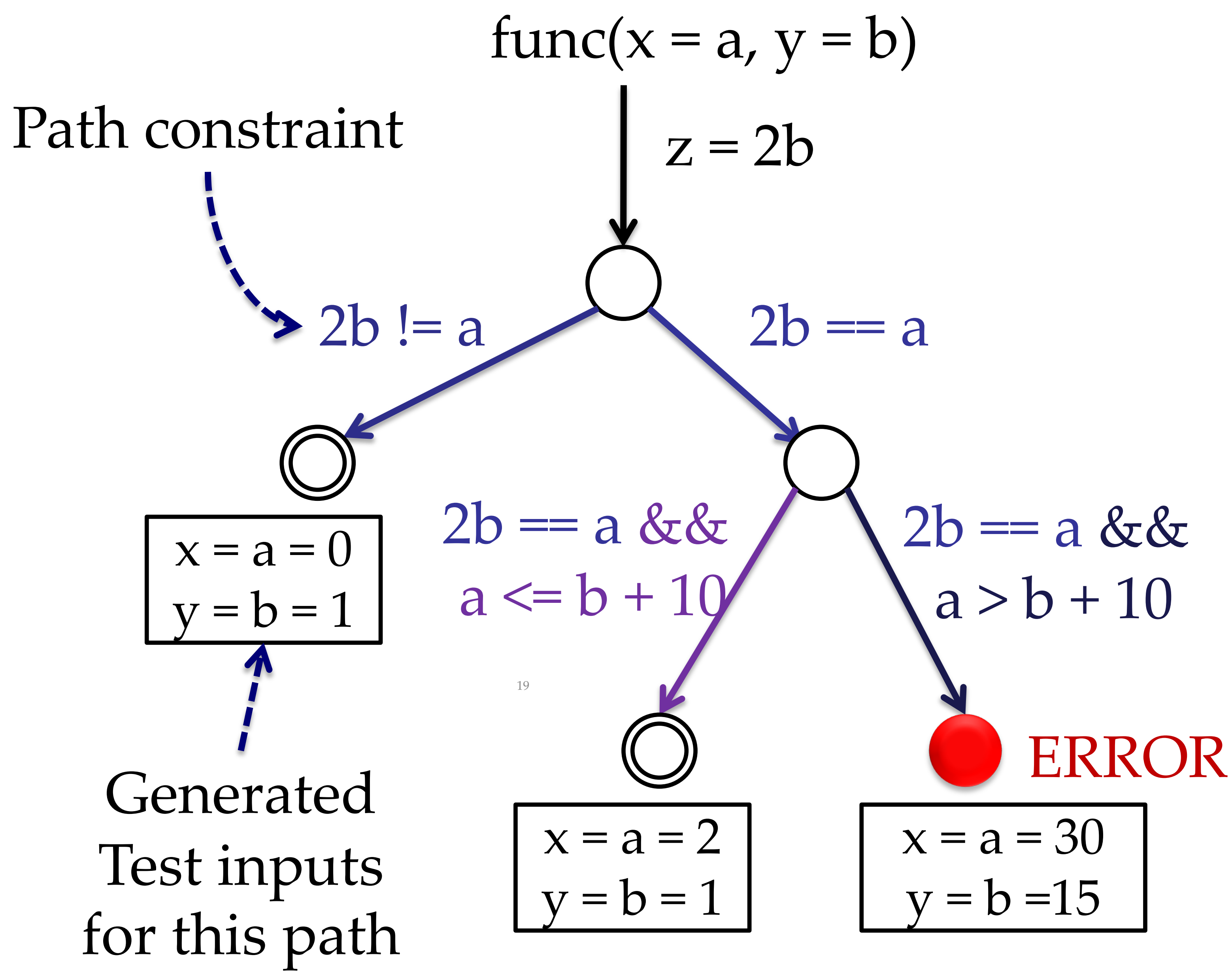
Symbolic Execution

- Instead of concrete state, the program maintains symbolic states, each of which maps variables to symbolic values
- Path condition is a quantifier-free formula over the symbolic inputs that encodes all branch decisions taken so far
- All paths in the program form its execution tree, in which some paths are feasible and some are infeasible

Symbolic Execution

```
Void func(int x, int y){
  int z = 2 * y;

  if(z == x){
   if (x > y + 10)
        ERROR

  }

}

int main(){
  int x = sym_input();
  int y = sym_input();
  func(x, y);
  return 0;

}
```

How does symbolic execution work?

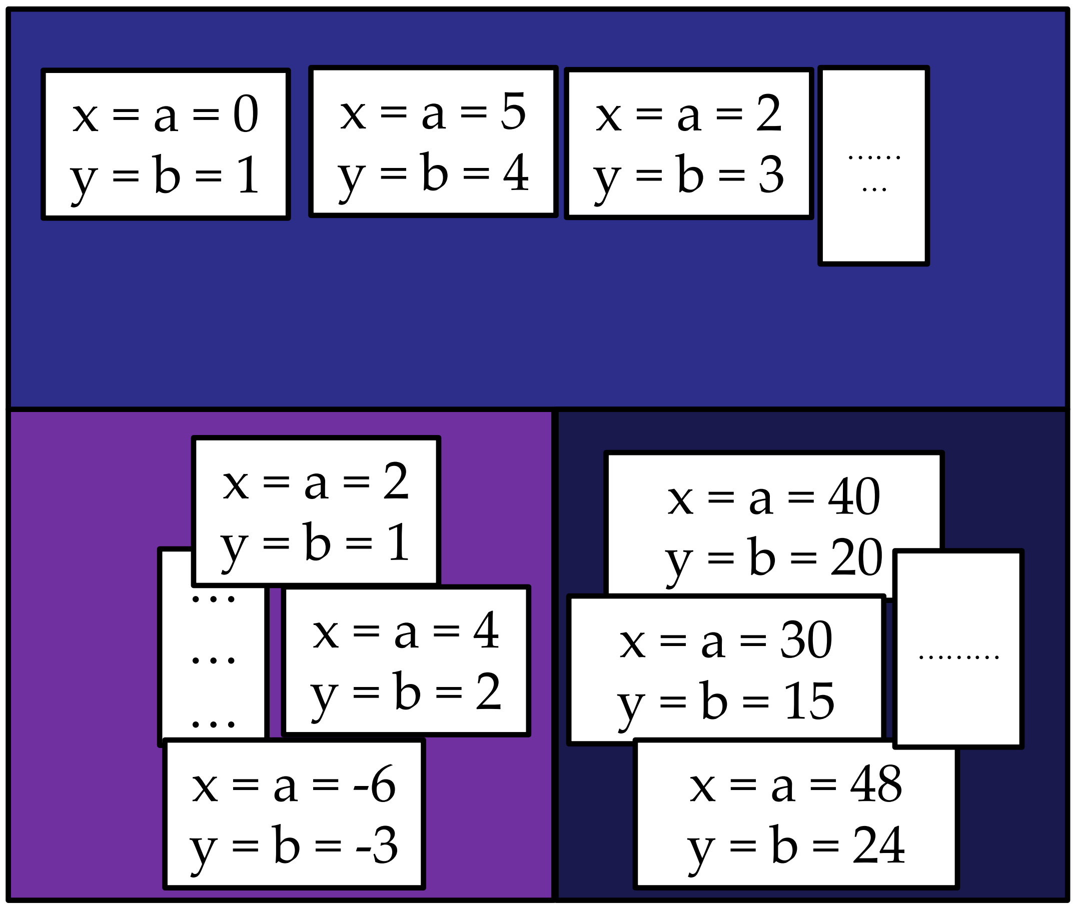


Note: Require inputs to be marked as symbolic

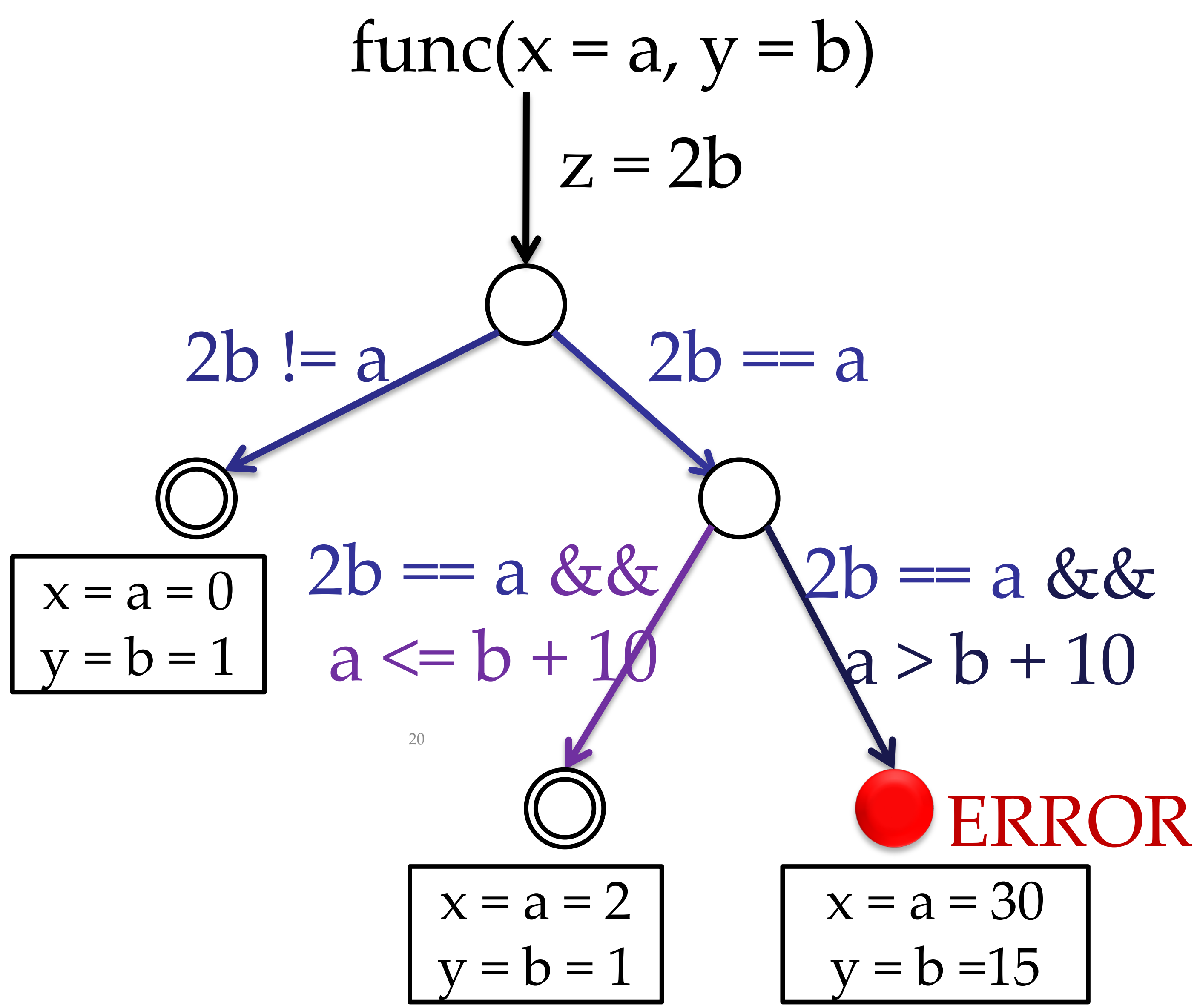
Symbolic Execution

PENNSTATE 1855

How does symbolic execution work?



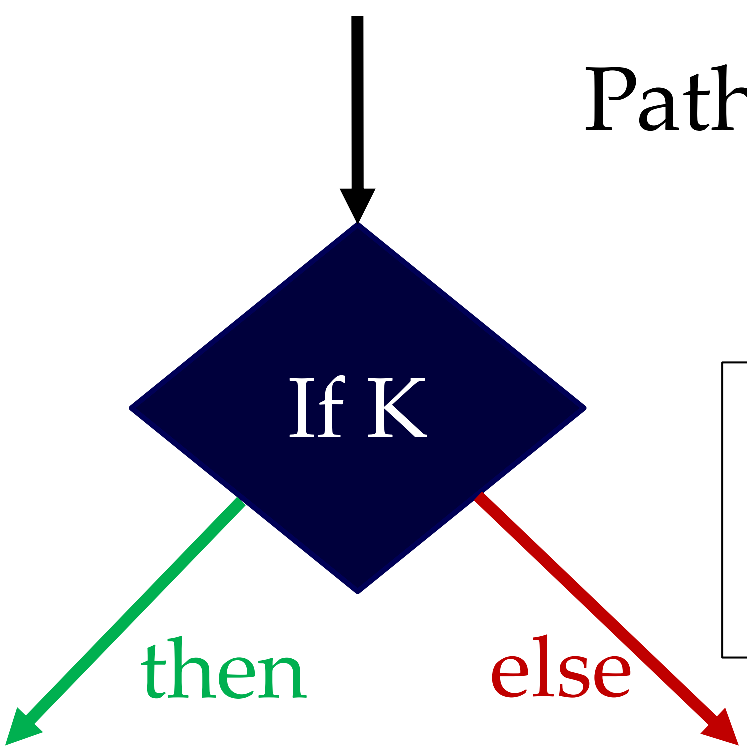
Path constraints represent equivalence classes of inputs



SMT Queries

- Counterexample queries (generate a test case)

- Branch queries (whether a branch is valid)



Path Constraints = $\{C_1, C_2, \ldots, C_n\}$;
SAT

<u>Use queries to determine validity of a branch</u>

else path is impossible: $C_1 \wedge C_2 \wedge \ldots \wedge C_n \wedge \neg K$ is UNSAT

then path is impossible: $C_1 \wedge C_2 \wedge \ldots \wedge C_n \wedge K$ is UNSAT

Symbolic Execution Tools

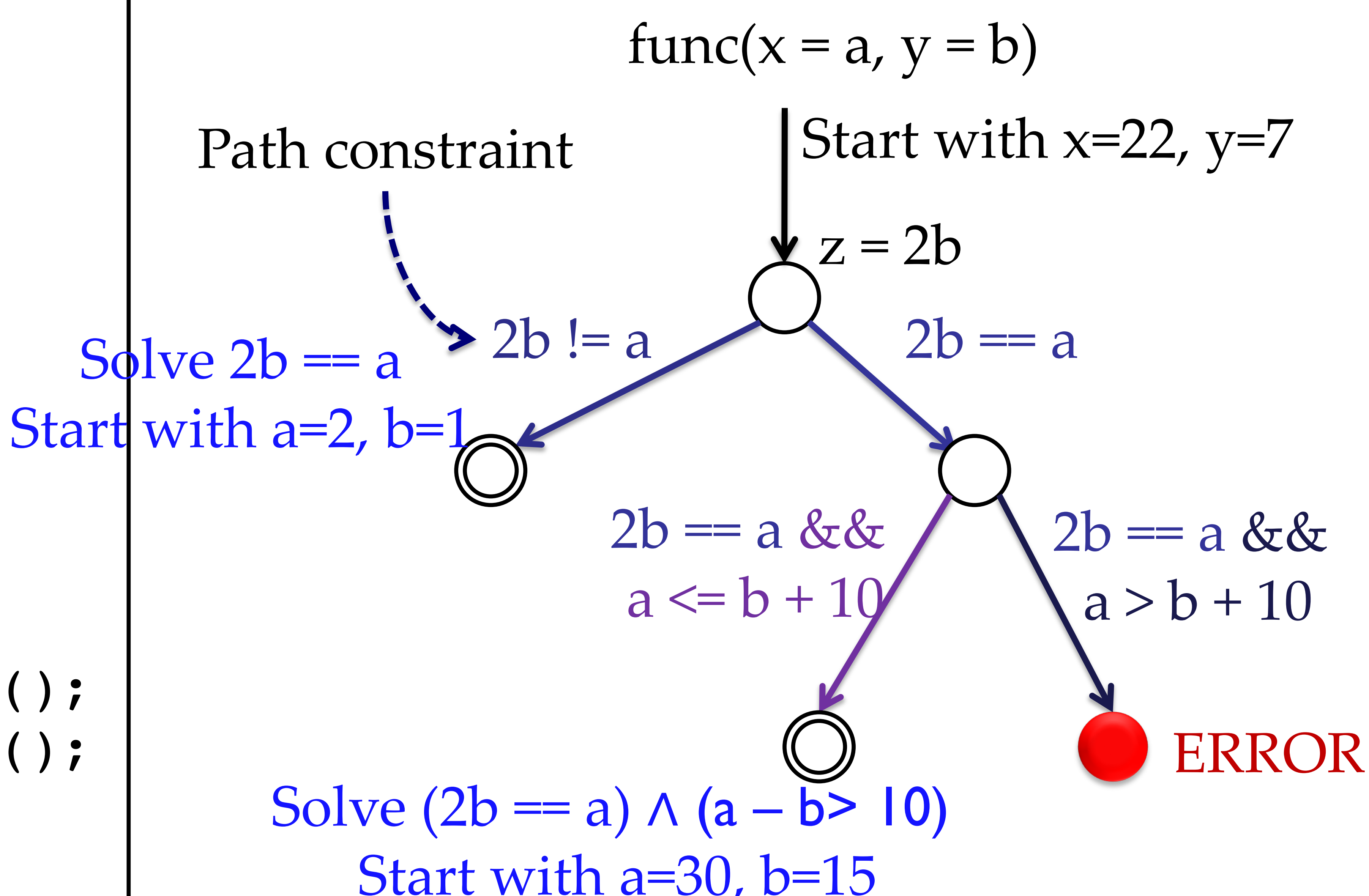
- FuzzBALL:
  - Works on binaries, generic SE engine. Used to, e.g., find PoC exploits given a vulnerability condition.
  - KLEE: Instruments through LLVM-based pass, relies on source code. Used to, e.g., nd bugs in programs.
  - S2E: Selective Symbolic Execution: automatic testing of large source base, combines KLEE with an concolic execution. Used to, e.g., test large source bases (e.g., drivers in kernels) for bugs.
- Efficiency of SE tool depends on the search heuristics and search strategy. As search space grows exponentially, a good search strategy is crucial for efficiency and scalability.

Symbolic Execution Summary

- Symbolic execution is a great tool to nd vulnerabilities or to create PoC exploits.
- Symbolic execution is limited in its scalability. An efficient search strategy is crucial.

Concolic Execution

```
Void func(int x, int
  y){
  int z = 2 * y;
  if(z == x){
   if (x > y + 10)
        ERROR
  }
}

int main(){
  int x = sym_input();
  int y = sym_input();
  func(x, y);
  return 0;
}
```



Formal Verification

- Formal verification is the act of using formal methods to proving or disproving the correctness of a certain system given its formal specification.
- Formal verification requires a specification and an abstraction mechanism to show that the formal specification either holds (i.e., its correctness is proven) or fails (i.e., there is a bug).
- Verification is carried out by providing a formal proof on the abstracted mathematical model of the system according to the specification. Many different forms of mathematical objects can be used for formal verification like finite state machines or formal semantics of programming languages (e.g., operational semantics or Hoare logic).

Takeaways

- Testing is simple but only tests for presence of functionality.
- Fuzzing uses test cases to explore other paths, might run forever.
- Static analysis has limited precision (e.g., aliasing).
- Symbolic execution needs guidance when searching through program.
- Formal verification is precise but arithmetic operations can be diffiucult.
- All mechanisms (except testing) run into state explosion.