PennState

# CSE 597: Security of Emerging Technologies

## Module: Static analysis and Symbolic Execution

Prof. Syed Rafiul Hussain

Systems and Network Security (SyNSec) Research Group

Department of Computer Science and Engineering

The Pennsylvania State University

# 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 vulnerabilities)*
  - *Can be effective when used carefully*

# Control Flow Analysis

PennState

## Can we detect code with no return check?

```
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);
}
```

- 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 Analysis

PennState

- 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

# Static Analysis Example

- Descriptors represent the sign of a value
  - Positive, negative, zero, unknown
- For an expression, *c = a * b*
  - If *a* has a descriptor *pos*
  - And *b* has a descriptor *neg*
- What is the descriptor for *c* after that instruction?
- How might this help?

# Static Analysis Example

- Descriptors represent the sign of a value
  - Positive, negative, zero, unknown
- For an expression, *c = a * b*
  - If *a* has a descriptor *pos*
  - And *b* has a descriptor *neg*
- What is the descriptor for *c* after that instruction?
- How might this help?

# Descriptors

- Choose a set of descriptors that
  - Abstracts away details to make analysis tractable
  - Preserves enough information that key properties hold
    - Can determine interesting results
- Using *sign* as a descriptor
  - Abstracts away specific integer values (billions to four)
  - Guarantees when *a*b = 0* it will be zero in all executions
- Choosing descriptors is one key step in static analysis

# Buffer Overflow Static Analysis

- For C code where
- `char dest[LEN]; int n, a , b;`
- `if (a > 2*b)`
- `n = input();`
- `else n = 50000;`
- `strncpy(dest, src, n);`
- Static analysis will try all paths of the program that impact variable n and flow to strncpy
  - May be complex in general because
    - Paths: Exponential number of program paths
    - Interprocedural: n may be assigned in another function
    - Aliasing: n's memory may be accessed from many places
- What descriptor values do you care about for n?

# Limitations of Static Analysis

- Scalability
  - Can be expensive to reason about all executions of complex programs
- False positives
  - Over-approximation means that executions that are not really possible may be found
- Accuracy
  - Alias analysis and other imprecision may lead to false positives
  - Sound methods (no false negatives) can exacerbate scalability and false positives problems
- Bottom line: Static analysis often must be directed

# Static vs. Dynamic

PennState

- 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

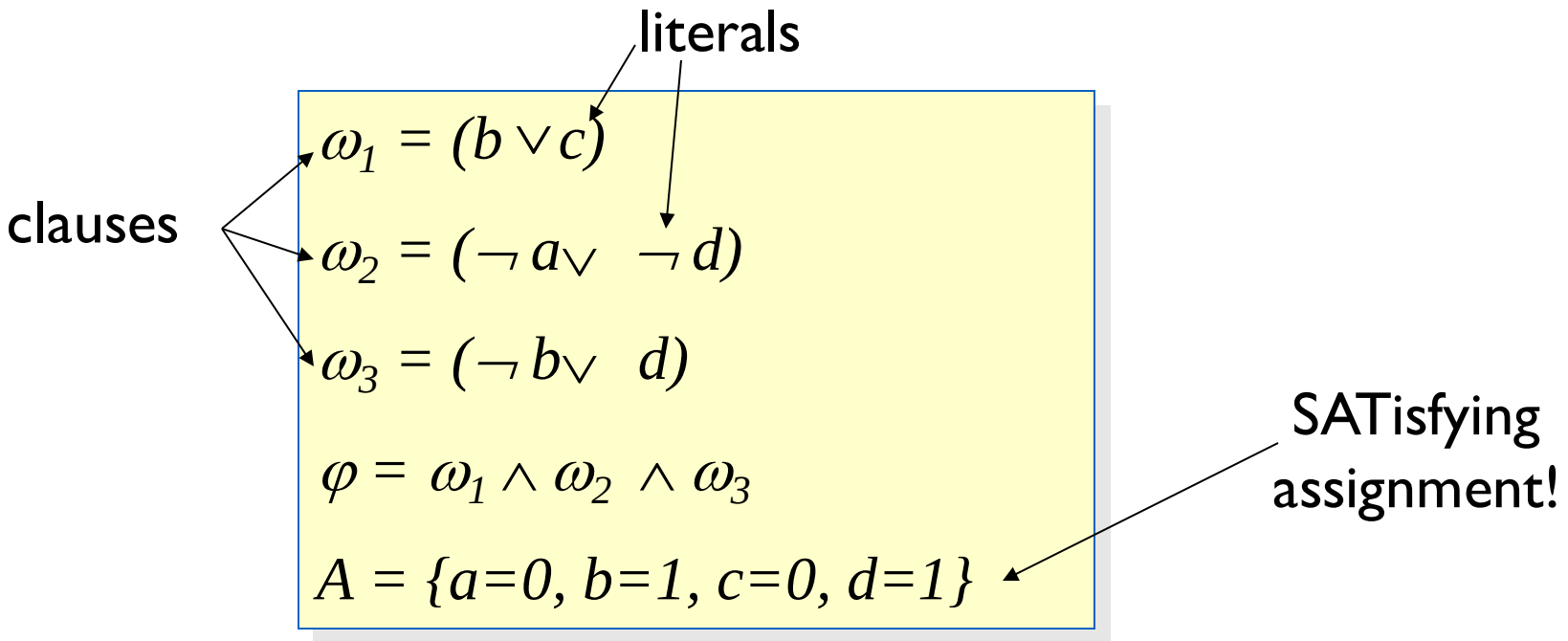
# Symbolic Execution

PennState

- 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

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



# Background: SMT

PennState

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?

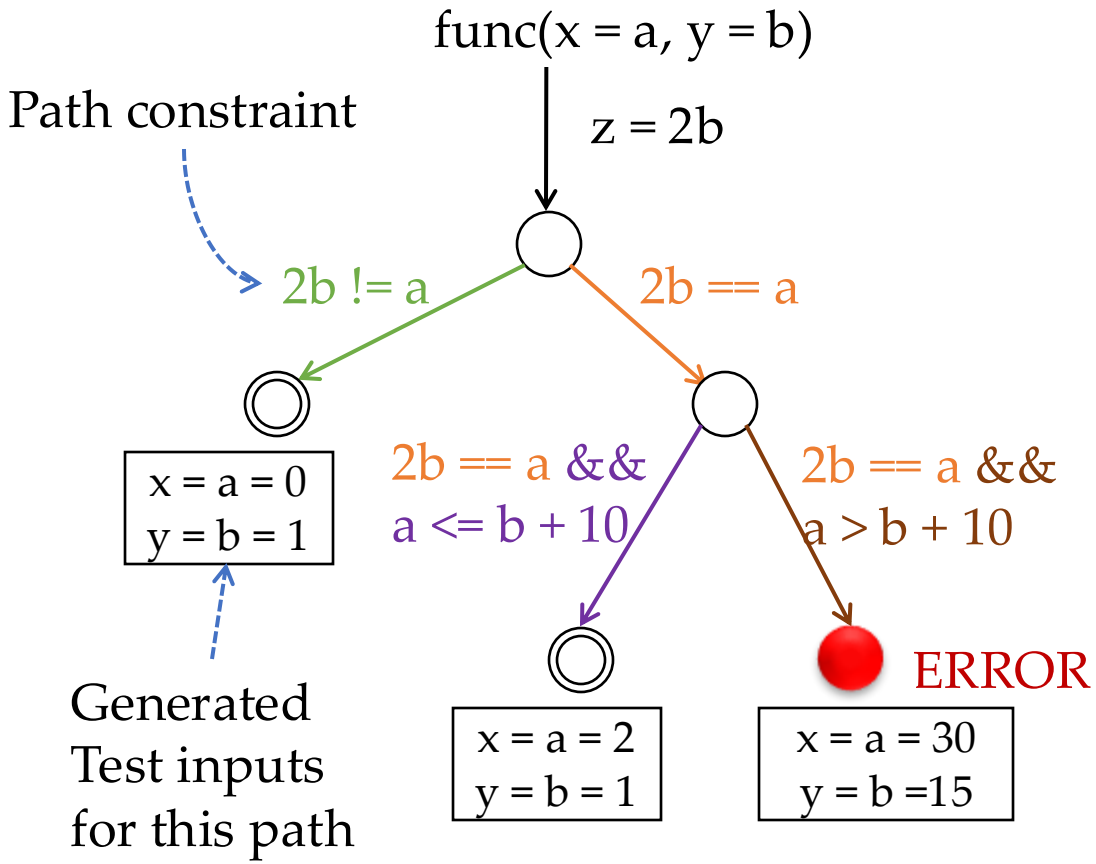
For most SMT solvers: φ is a ground formula

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

# Symbolic Execution

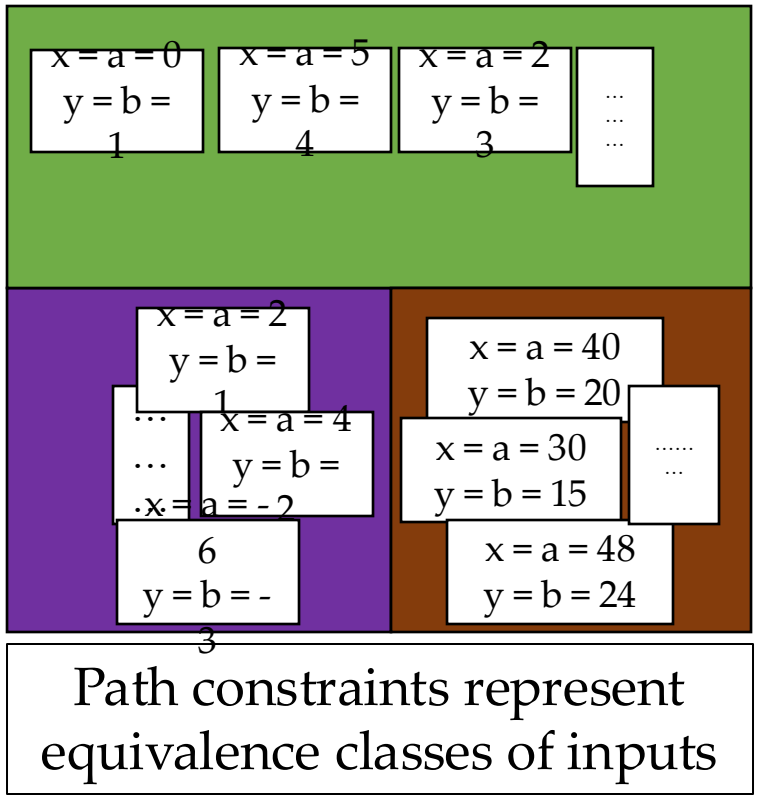
```
Void func(int x, int
  y){
  int z = 2 * y;
  if(z == x){
        if (x > y + 10)
              ERROR
  }
}
int main(){
  int x = sym_input();
  int y = sym_input();
  func(x, y);
  return 0;
}
```

## How does symbolic execution work?



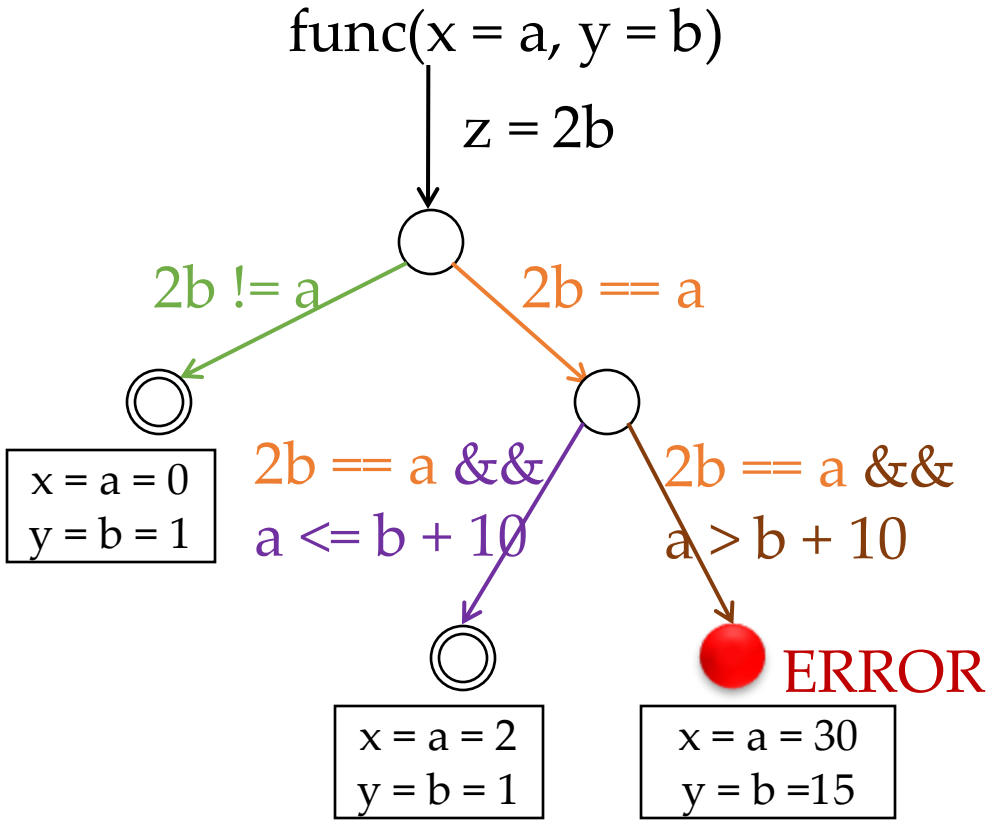

Note: Require inputs to be marked as symbolic

# Symbolic Execution




Path constraints represent equivalence classes of inputs

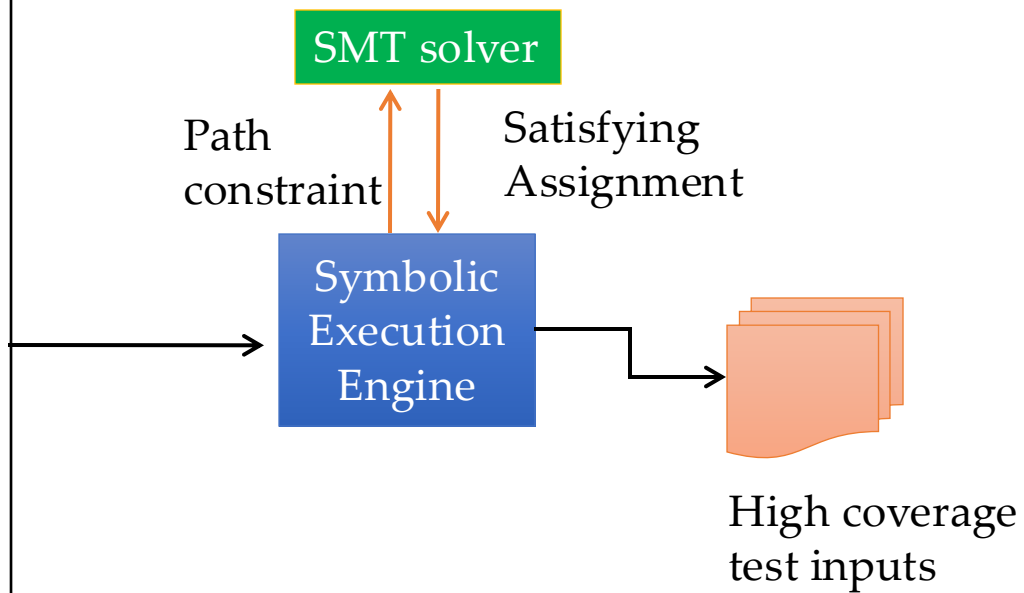
**How does symbolic execution work?**



# Symbolic Execution

PennState

```
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

PennState

- 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 represents 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

PennState

- 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 Tools

PennState

- 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.

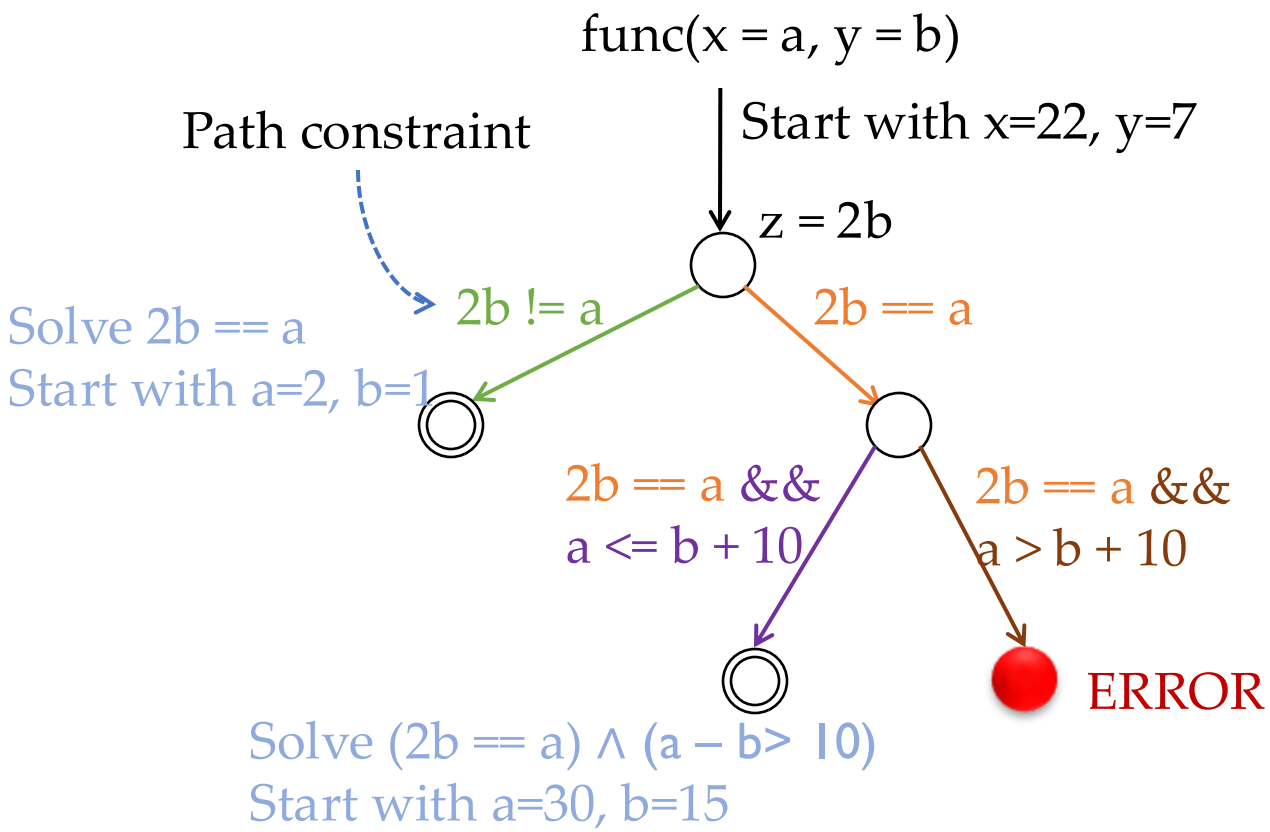
# Symbolic Execution Summary

PennState

- Symbolic execution is a great tool to find 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

PennState

- 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

PennState

- 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 difficult.
- All mechanisms (except testing) run into state explosion.

# Thanks

PennState

Thanks to Omar Chowdhury, Gang Tan, Suman Jana and Baishakhi Ray for some slides.