PennState

CSE 597: Security of Emerging Technologies

Module: Testing and Fuzzing

Prof. Syed Rafiul Hussain

Systems and Network Security (SyNSec) Research Group

Department of Computer Science and Engineering

The Pennsylvania State University

Security Analysis Techniques

PennState

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

Automatic test case generation | Static analysis | Program verification

→

Fuzzing | Dynamic symbolic execution

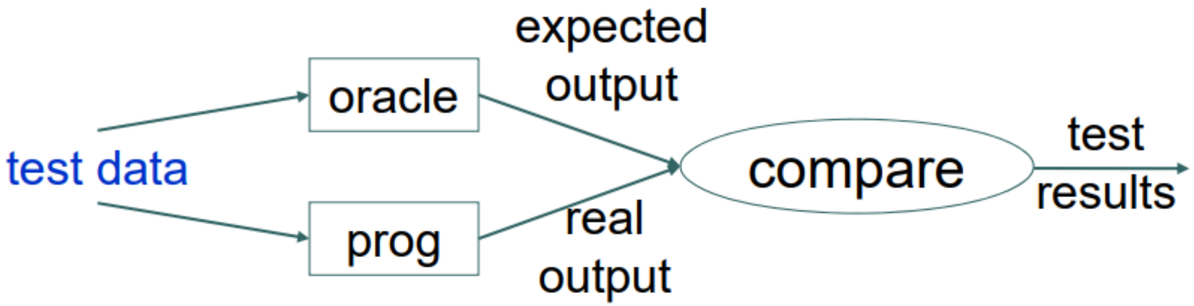
Lower coverage
Lower false positives
Higher false negatives

Higher coverage
Higher false positives
Lower false negatives

Testing

PennState

- 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 { //skipping space and tab
         readchar();
       } while (lastc == ' ' || lastc == '\t');
       return (lastc);
   }
```

Fuzz Testing

Fuzz Testing

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

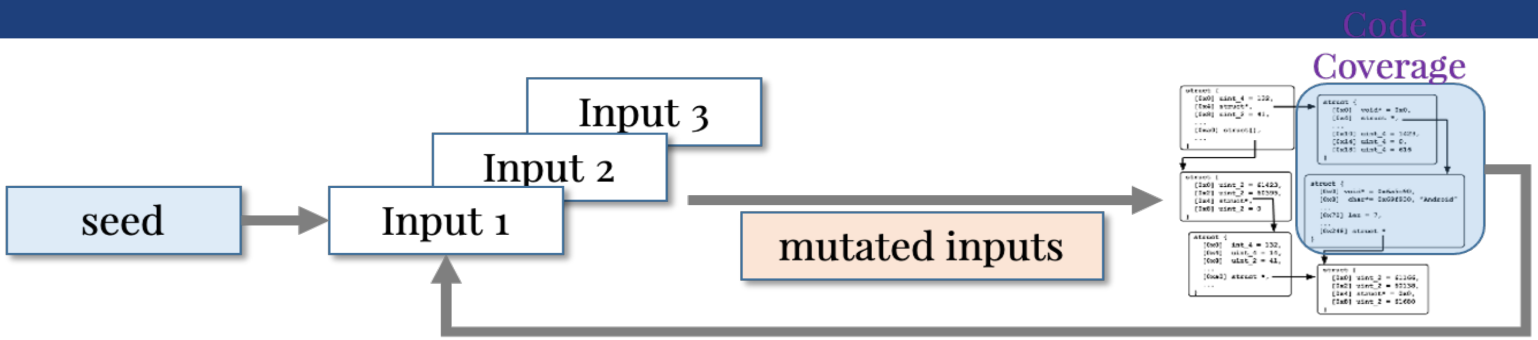
```
format.c (line 276):
... while (lastc != '\n') { //reading line
      rdc(); }

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

When end of file, readchar() sets lastc to be 0; then the program hangs (infinite loop)

Fuzzing

PennState



- 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?).

Blackbox Fuzzing

- Given a program simply feed random inputs and see whether it exhibits incorrect behavior (e.g., crashes)
- Advantage: easy, low programmer cost
- Disadvantage: inefficient
  - Inputs often require structures, random inputs are likely to be malformed
  - Inputs that trigger an incorrect behavior is a a very small fraction, probably of getting lucky is very low

Fuzzing

PennState

- Automatically generate test cases
- Many slightly anomalous test cases are input into a target
- Application is monitored for errors
- Inputs are generally either file based (.pdf, .png, .wav, etc.) or network based (http, SNMP, etc.)

Problem detection

PennState

- See if program crashed
  - Type of crash can tell a lot (SEGV vs. assert fail)
- Run program under dynamic memory error detector (valgrind/purify/AddressSanitizer)
  - Catch more bugs, but more expensive per run.
- See if program locks up
- Roll your own dynamic checker e.g. valgrind skins

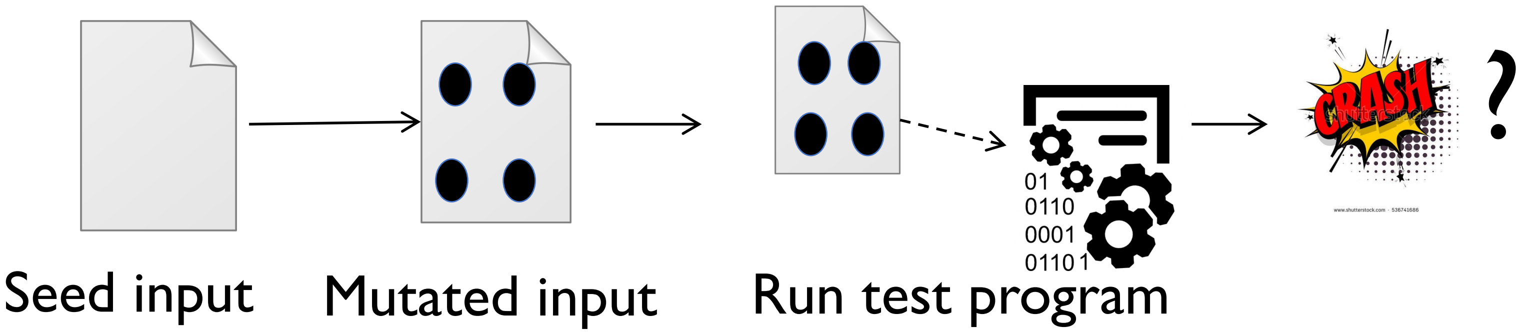
Regression vs. Fuzzing

PennState

	Regerssion	Fuzzing
Definition	Run program on many normal inputs, look for badness	Run program on many abnormal inputs, look for badness
Goals	Prevent normal users from encountering errors (e.g., assertion failures are bad)	Prevent attackers from encountering exploitable errors (e.g., assertion failures are often ok)

Enhancement 1: Mutation-Based fuzzing

- Take a well-formed input, randomly perturb (flipping bit, etc.)
- Little or no knowledge of the structure of the inputs is assumed
- Anomalies are added to existing valid inputs
  - Anomalies may be completely random or follow some heuristics (e.g., remove NULL, shift character forward)
- Examples: ZZUF, Taof, GPF, ProxyFuzz, FileFuzz, Filep, etc.



Example: fuzzing a PDF viewer

PennState

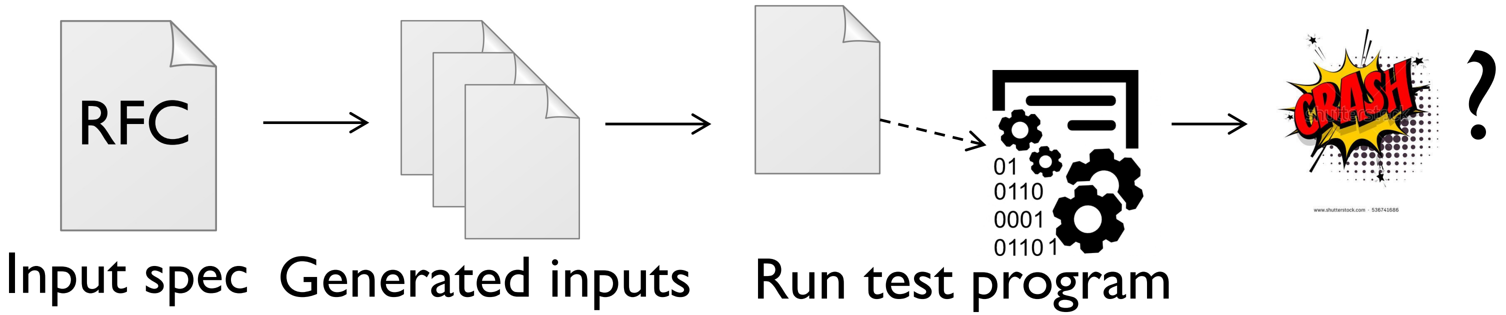
- Google for .pdf (about 1 billion results)
- Crawl pages to build a corpus
- Use fuzzing tool (or script)
  - Collect seed PDF files
  - Mutate that file
  - Feed it to the program
  - Record if it crashed (and input that crashed it)

Mutation-based fuzzing

- Super easy to setup and automate
- Little or no file format knowledge is required
- Limited by initial corpus
- May fail for protocols with checksums, those which depend on challenge

Enhancement II: Generation-Based Fuzzing

- Test cases are generated from some description of the input format: RFC, documentation, etc.
  - Using specified protocols/file format info
  - E.g., SPIKE by Immunity
- Anomalies are added to each possible spot in the inputs
- Knowledge of protocol should give better results than random fuzzing



Mutation-based vs. Generation-based

PennState

- Mutation-based fuzzer
  - Pros: Easy to set up and automate, little to no knowledge of input format required
  - Cons: Limited by initial corpus, may fall for protocols with checksums and other hard checks
- Generation-based fuzzers
  - Pros: Completeness, can deal with complex dependencies (e.g., checksum)
  - Cons: writing generators is hard, performance depends on the quality of the spec

How much fuzzing is enough?

- Mutation-based-fuzzers may generate an infinite number of test cases. When has the fuzzer run long enough?
- Generation-based fuzzers may generate a finite number of test cases. What happens when they're all run and no bugs are found?

Code coverage

- Some of the answers to these questions lie in *code coverage*
- Code coverage is a metric that can be used to determine how much code has been executed.
- Data can be obtained using a variety of profiling tools. e.g. gcov, lcov

Line coverage

PennState

- **Line/block coverage**: Measures how many lines of source code have been executed.
- For the code on the right, how many test cases (values of pair (a,b)) needed for full(100%) line coverage?

```
if( a >  2  )
      a =  2;
if( b >2  )
      b =  2;
```

Branch coverage

PennState

- Branch coverage: Measures how many branches in code have been taken (conditional jmps)
- For the code on the right, how many test cases needed for full branch coverage?

```
if( a >  2  )
      a =  2;
if( b >2  )
      b =  2;
```

Path coverage

- Path coverage: Measures how many paths have been taken

- For the code on the right, how many test cases needed for full path coverage?

```
if( a >  2  )
     a =  2;
if( b >2  )
     b =  2;
```

Benefits of Code coverage

- Can answer the following questions
  - How good is an initial file?
  - Am I getting stuck somewhere?

```
if (packet[0x10] < 7) { //hot path
} else { //cold path }
```

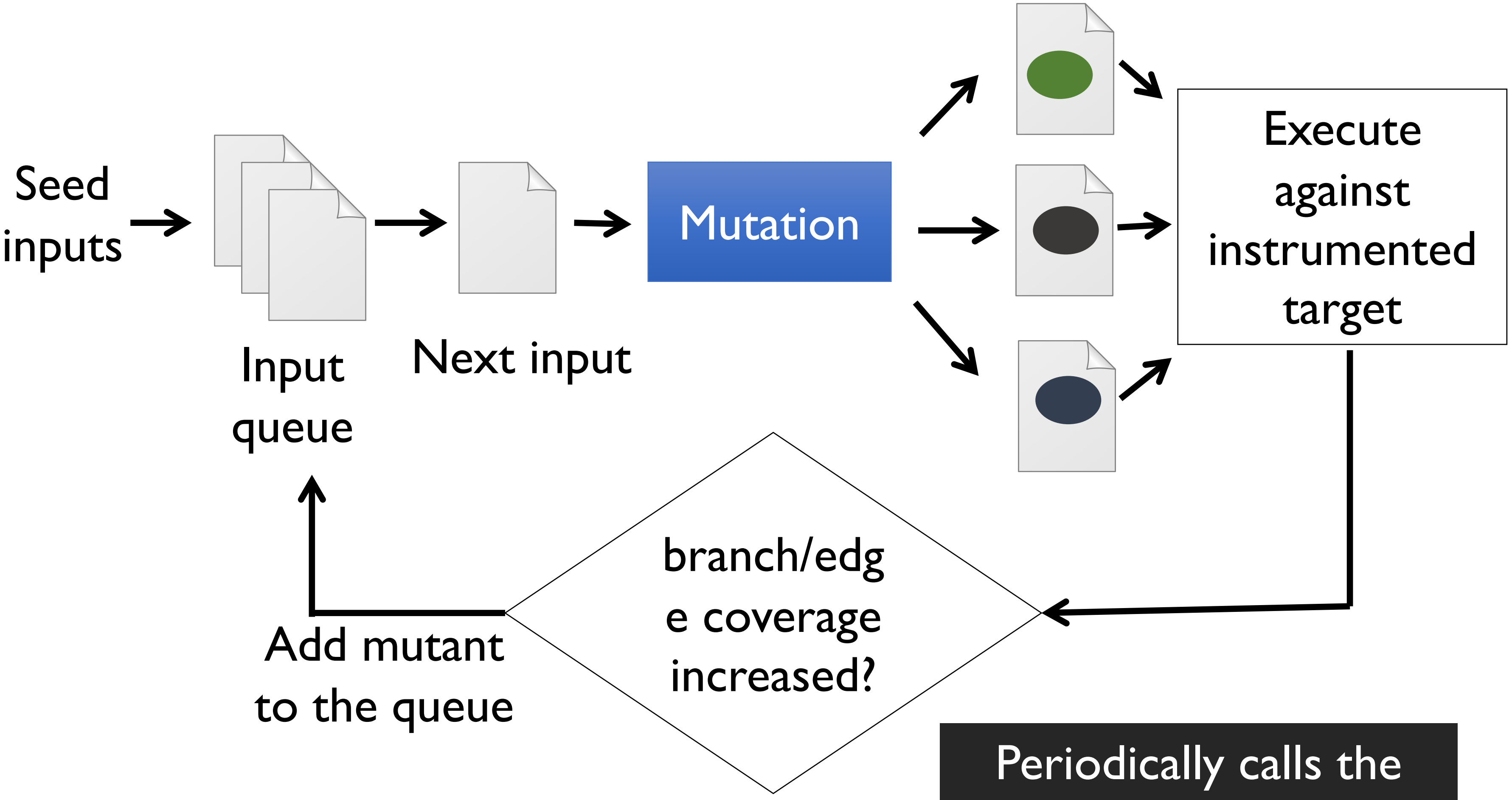
  - How good is fuzzerX vs. fuzzerY
  - Am I getting benefits by running multiple fuzzers?

Enhancement III: Coverage-guided gray-box fuzzing

- Special type of mutation-based fuzzing
  - Run mutated inputs on instrumented program and measure code coverage
  - Search for mutants that result in coverage increase
  - Often use genetic algorithms, i.e., try random mutations on test corpus and only add mutants to the corpus if coverage increases
  - Examples: AFL, libfuzzer

American Fuzzy Lop (AFL)

PennState



Periodically calls the queue without affecting total coverage

AFL

PennState

- Instrument the binary at compile-time
- Regular mode: instrument assembly
- Recent addition: LLVM compiler instrumentation mode
- AFL-fuzz is the driver process, the target app runs as separate process(es)

Data-flow-guided fuzzing

PennState

- Intercept the data flow, analyze the inputs of comparisons
  - Incurs extra overhead
- Modify the test inputs, observe the effect on comparisons
- Prototype implementations in libFuzzer and go-fuzz

Static Analysis

- 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*

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

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 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

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?

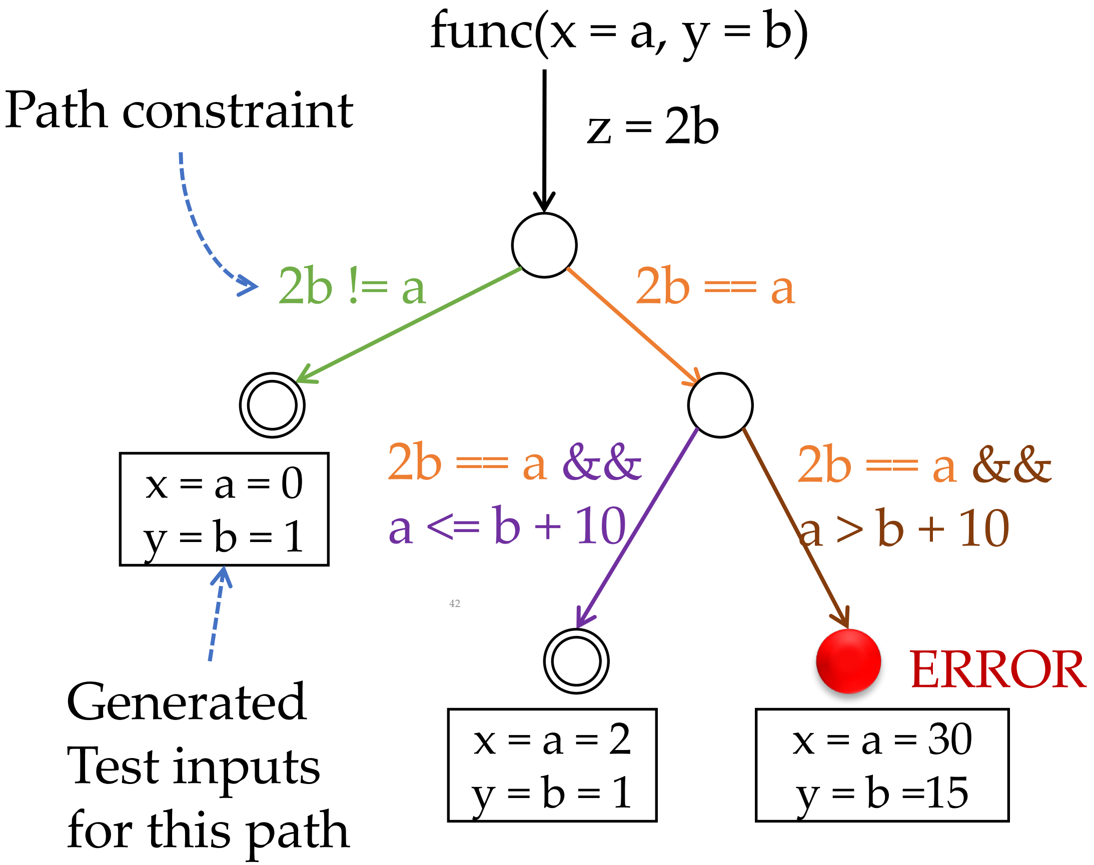
For most SMT solvers: φ is a ground formula

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

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?

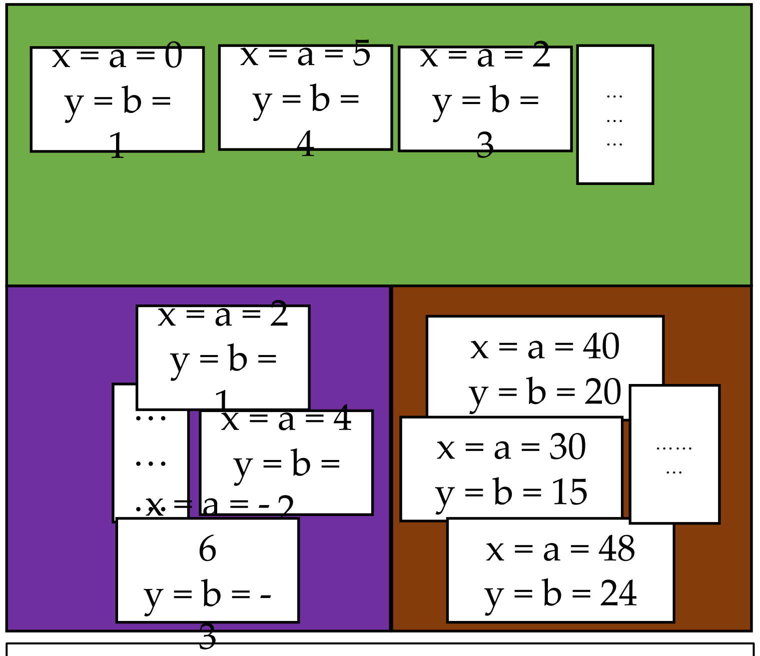


Note: Require inputs to be marked as symbolic

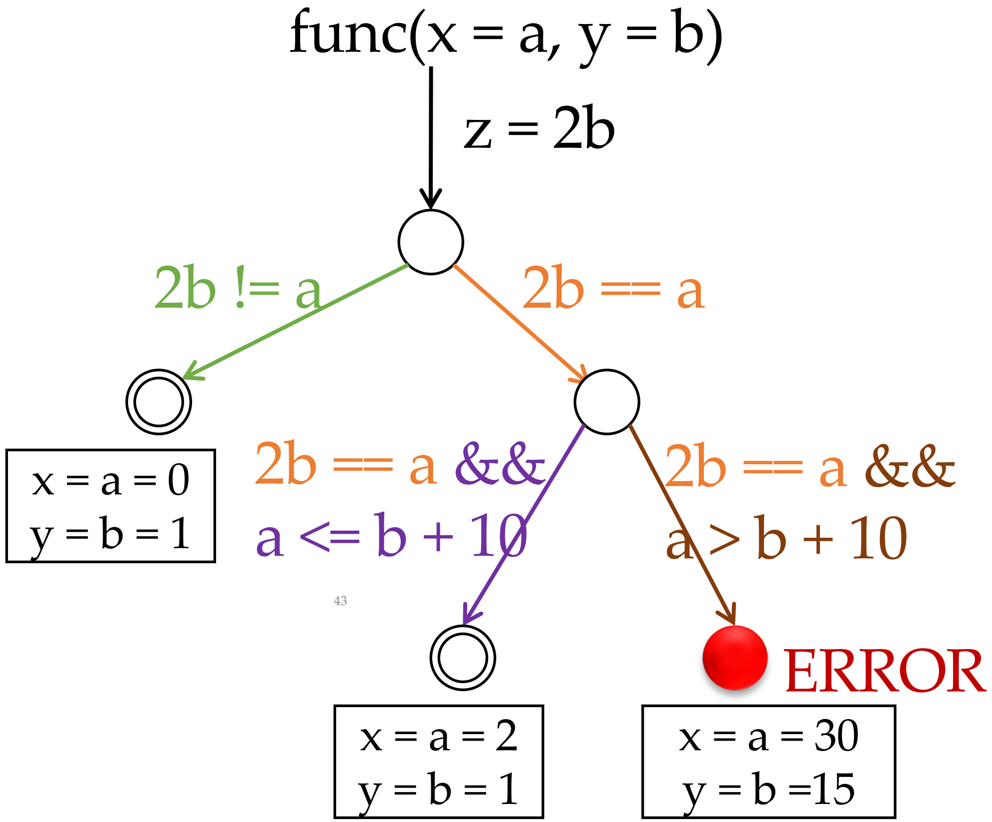
Symbolic Execution

PennState

How does symbolic execution work?

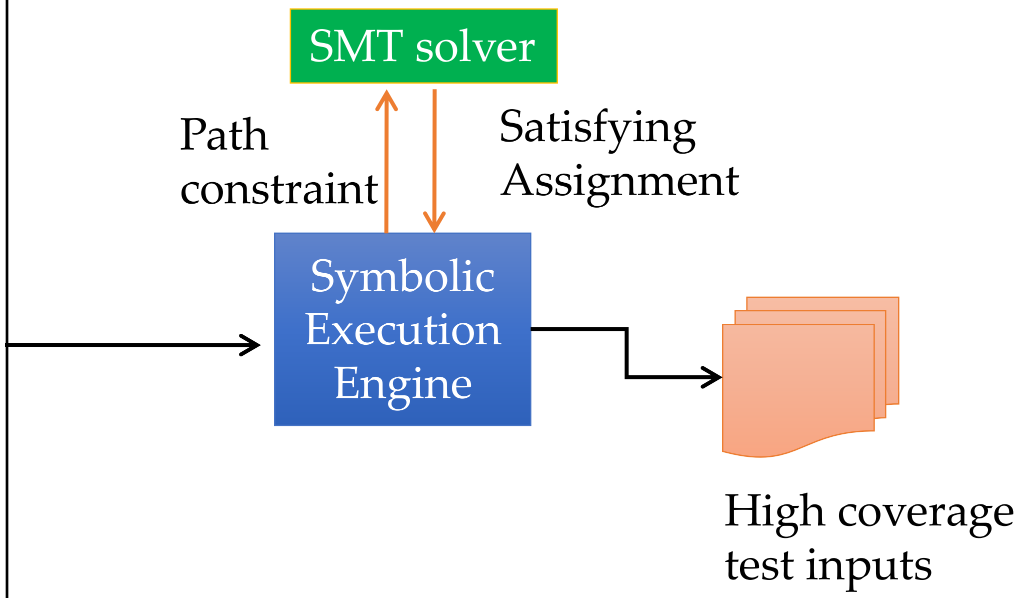


Path constraints represent equivalence classes of inputs



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 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

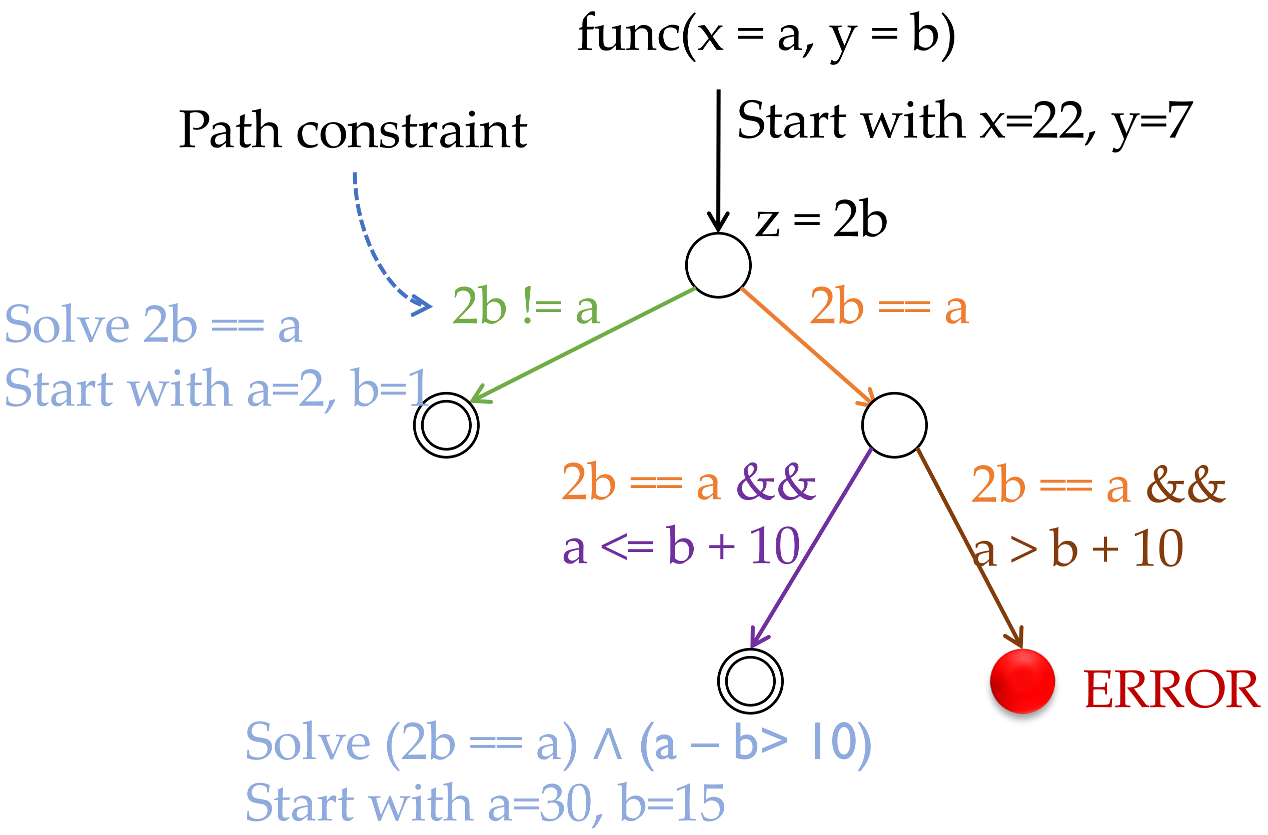
- FuzzBALL:
  - Works on binaries, generic SE engine. Used to, e.g., find PoC exploits given a vulnerability condition.
  - KLEE: Instruments through LLVM-based pass, relies on source code. Used to, e.g., nd bugs in programs.
  - S2E: Selective Symbolic Execution: automatic testing of large source base, 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 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

- 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.