Project 1: 4 days left

Prevention-Based Cyber Security: Program Testing

CS 459/559: Science of Cyber Security
11th Lecture

Instructor:

Guanhua Yan

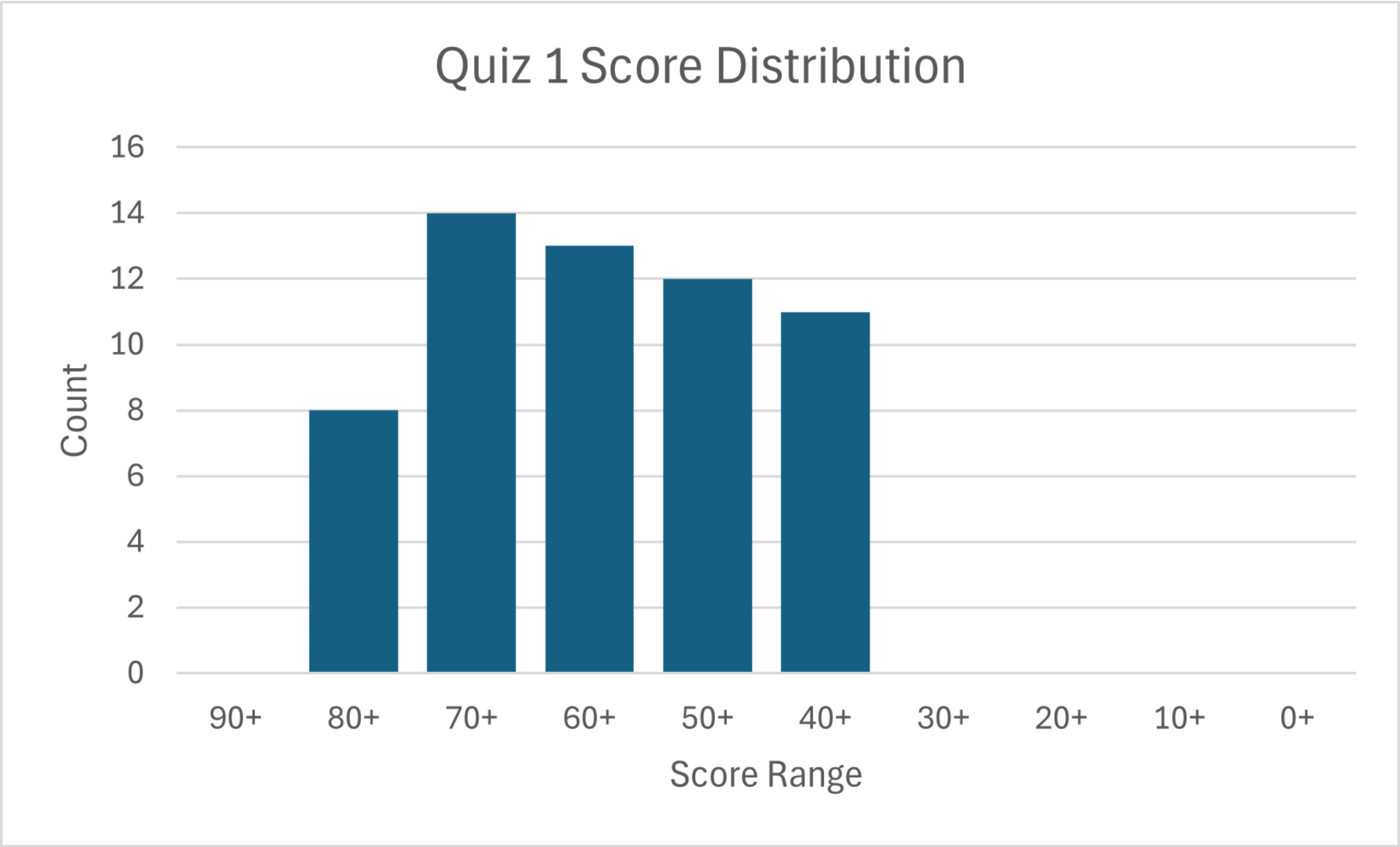
Agenda

- ~~Quiz 1: September 29 (closed book)~~
- Project 1 (offense): October 10
- Project 2 (defense): December 5
- Presentations: 11/17, 11/19, 11/24, 12/1, 12/3
- Final report: December 15

How do you think of first quiz?

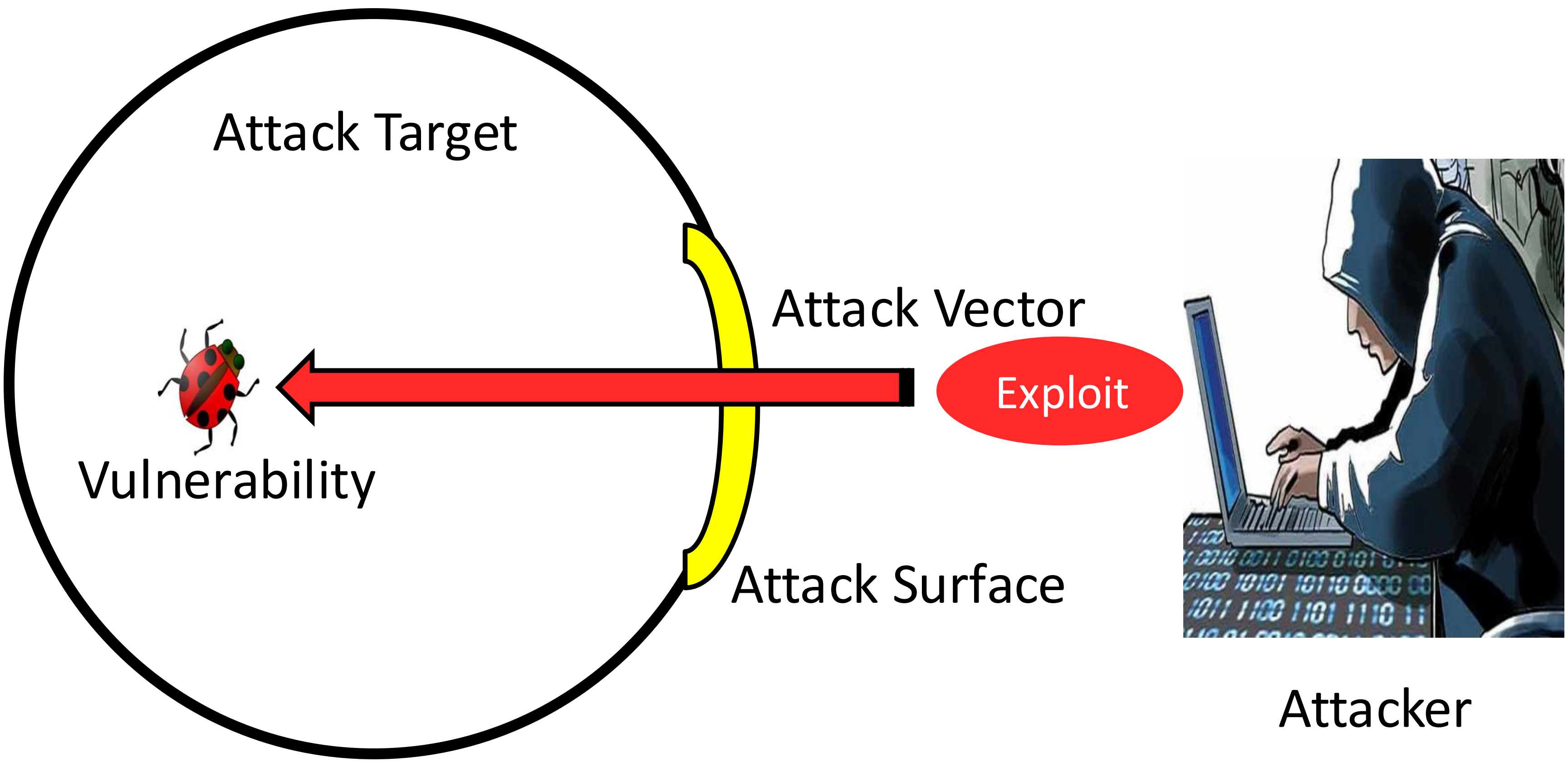
- A: Too difficult
- B: Too easy
- C: About right
- D: Refuse to comment

Quiz 1 score distribution



Goal of program testing

- **Develop techniques to discover and patch vulnerabilities before they are exploited**



Outline

- Fuzzing
- Symbolic execution
- Concolic execution

Fuzzing

Fuzz Testing

- Run program on many random, abnormal inputs and look for bad behavior in the responses
  - Bad behaviors such as crashes or hangs
- What are the benefits of fuzz testing over regular testing?

Fuzz Testing (Bart Miller, U. Of Wisconsin)

- A night in 1988 with thunderstorm and heavy rain
- Connected to his office Unix system via a dial up connection
- The heavy rain introduced noise on the line
- Crashed many UNIX utilities he had been using everyday
- He realized that there was something deeper
- Asked three groups in his grad-seminar course to implement this idea of fuzz testing
  - Two groups failed to achieve any crash results!
  - The third group succeeded! Crashed 25-33% of the utility programs on the seven Unix variants that they tested

Fuzz Testing

- Approach
  - Generate random inputs
  - Run lots of programs using random inputs
  - Identify crashes of these programs
  - Correlate random inputs with crashes
- Errors found: Not checking returns, Array indices out of bounds, not checking null pointers, …

Fuzz Testing Overview

- Black-box fuzzing
  - Treating the system as a blackbox during fuzzing; not knowing details of the implementation
- Grey-box fuzzing
- White-box fuzzing
  - Design fuzzing based on internals of the system

Black Box Fuzzing

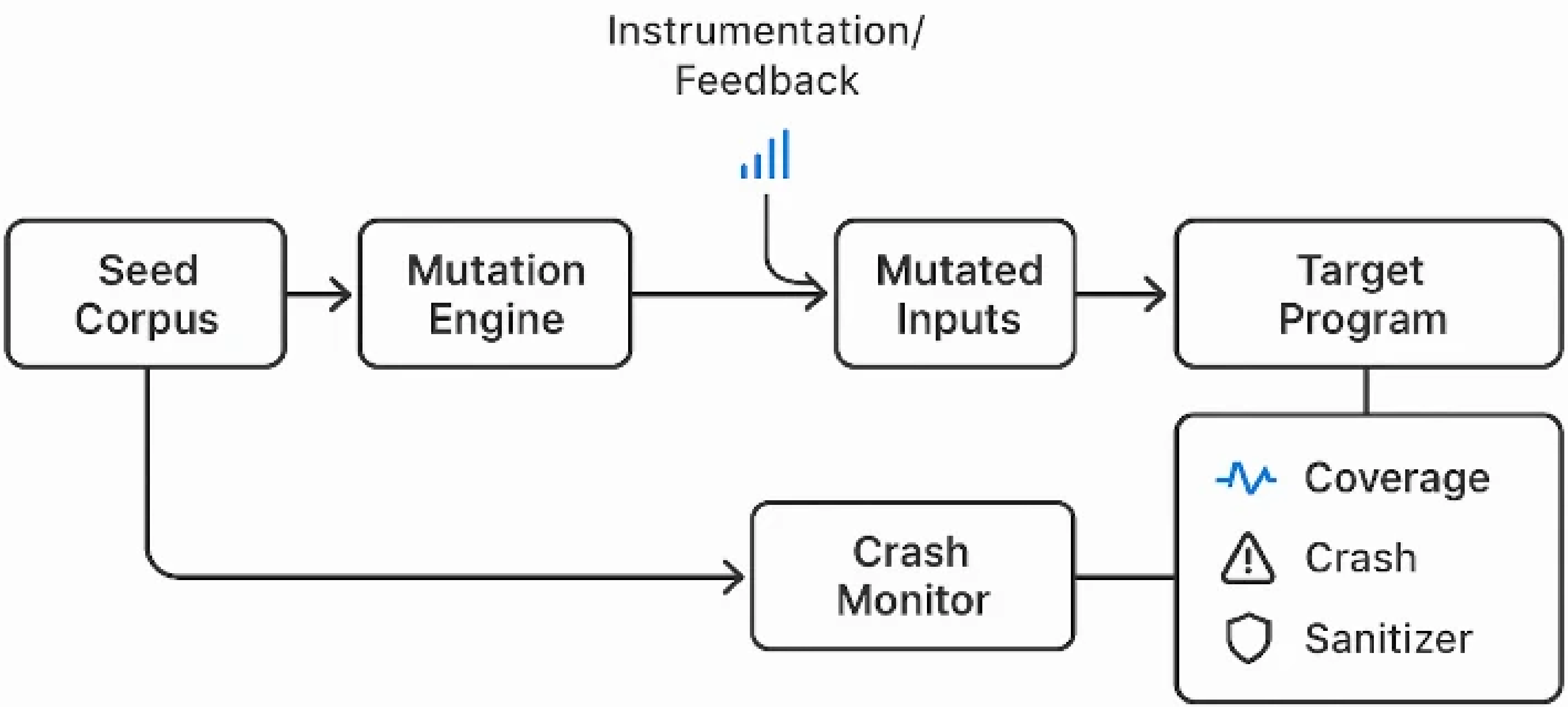
- Example that would be hard for black box fuzzing to find the error

```
function( char *name, char *passwd, char *buf )
{
      if ( authenticate_user( name, passwd )) {
            if ( check_format( buf )) {
                  update( buf ); // crash here
            }
      }
}
```

Mutation-Based Fuzzing

- User supplies a well-formed input
- Fuzzing: Generate random changes to that input
- No assumptions about input
  - Only assumes that variants of well-formed input may be problematic
- Example: zzuf
  - http://sam.zoy.org/zzuf/
  - Reading: The Fuzzing Project Tutorial

MUTATION FUZZING WORKFLOW



Mutation-Based Fuzzing

- The Fuzzing Project Tutorial
  - zzuf -s 0:1000000 -c -C 0 -q -T 3 objdump -x win9x.exe
  - Fuzzes the program `objdump` using the sample input `win9x.exe`
  - Try 1M seed values (-s) from command line (-c) and keep running if crashed (-C 0) with timeout (-T 3)

```
zzuf [-AcdimnqSvxX] [-s seed|-s start:stop] [-r ratio|-r
min:max] [-f fuzzing] [-D delay] [-j jobs] [-C crashes] [-B
bytes] [-t seconds] [-T seconds] [-U seconds] [-M mebibytes]
[-b ranges] [-p ports] [-P protect] [-R refuse] [-a list] [-l
list] [-I include] [-E exclude] [-O opmode] [PROGRAM
[ARGS]...]
zzuf -h | --help
zzuf -V | --version
```

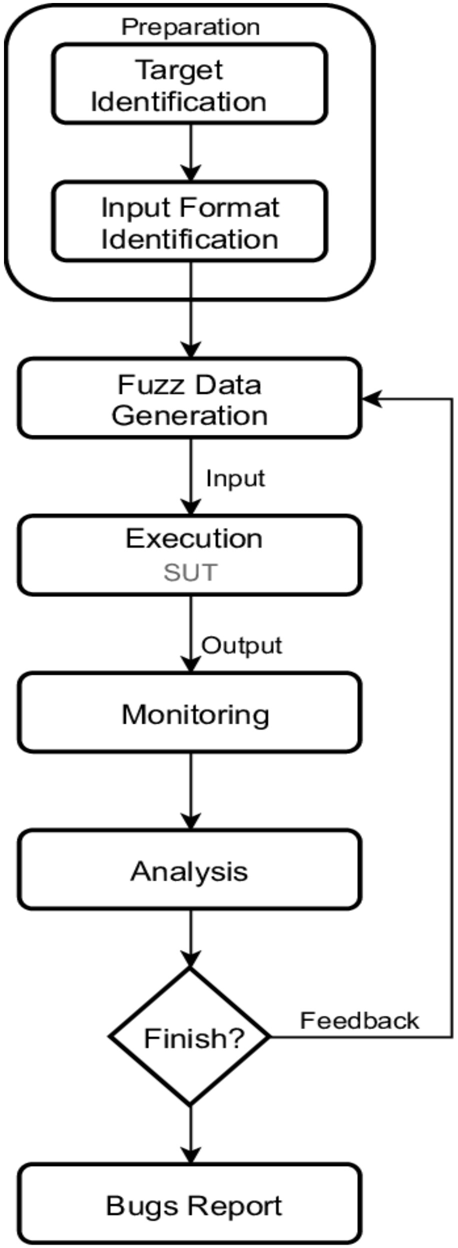
Mutation-Based Fuzzing

- Easy to setup, and not dependent on program details
- But may be strongly biased by the initial input
- Still prone to some problems
  - May re-run the same path over again (same test)
  - May be very hard to generate inputs for certain paths (checksums, hashes, restrictive conditions)

Generation-Based Fuzzing

- Generational fuzzer generate inputs “from scratch” rather than using an initial input and mutating
- However, require the user to specify a format or protocol spec to start
  - Equivalently, write a generator for generating well-formated input
- Examples include
  - SPIKE, Peach Fuzz
- However format-aware fuzzing is cumbersome, because you'll need a fuzzer specification for every input format you are fuzzing

Generation-Based Fuzzing Workflow

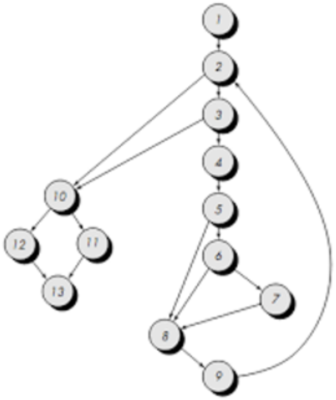


Generation-Based Fuzzing

- Can be more accurate, but at a cost
- Pros: More complete search
  - Values more specific to the program operation
  - Can account for dependencies between inputs
- Cons: More work
  - Get the specification
  - Write the generator – ad hoc
  - Need to do for each program

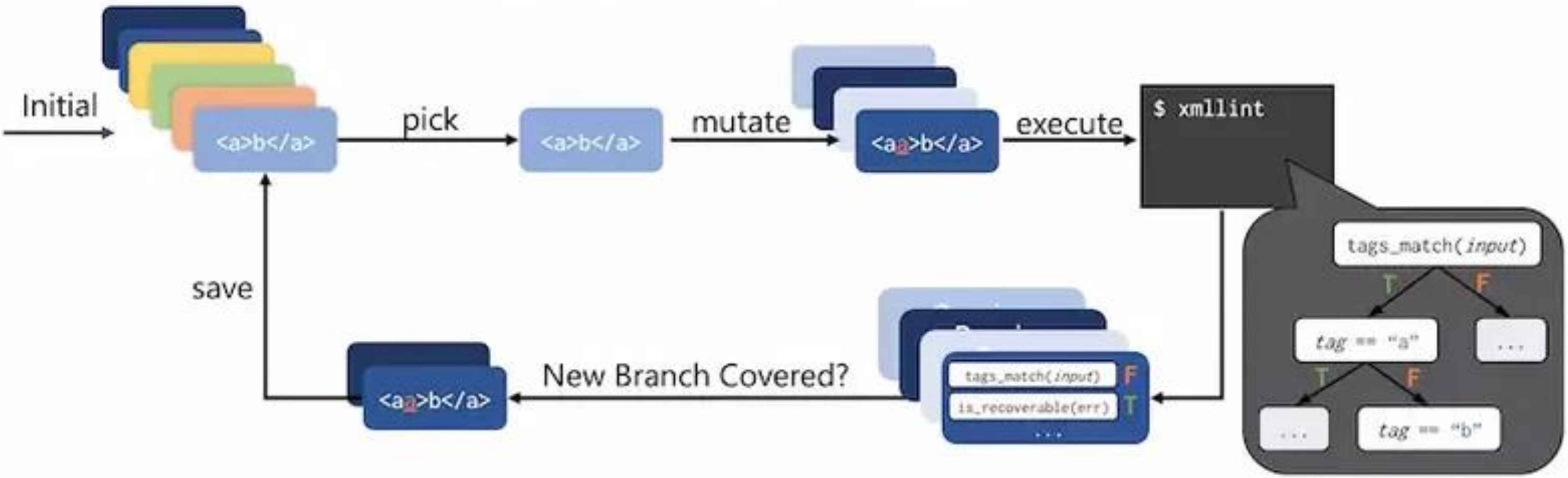
Coverage-Based Fuzzing

- AKA grey-box fuzzing
- Rather than treating the program as a black box, instrument the program to track coverage
  - E.g., the edges covered
- Maintain a pool of high-quality tests
  - Start with some initial ones specified by users
  - Mutate tests in the pool to generate new tests
  - Run new tests
  - If a new test leads to new coverage (e.g., edges), save the new test to the pool; otherwise, discard the new test



Coverage-Guided Fuzzing

AFL, libFuzzer, honggfuzz

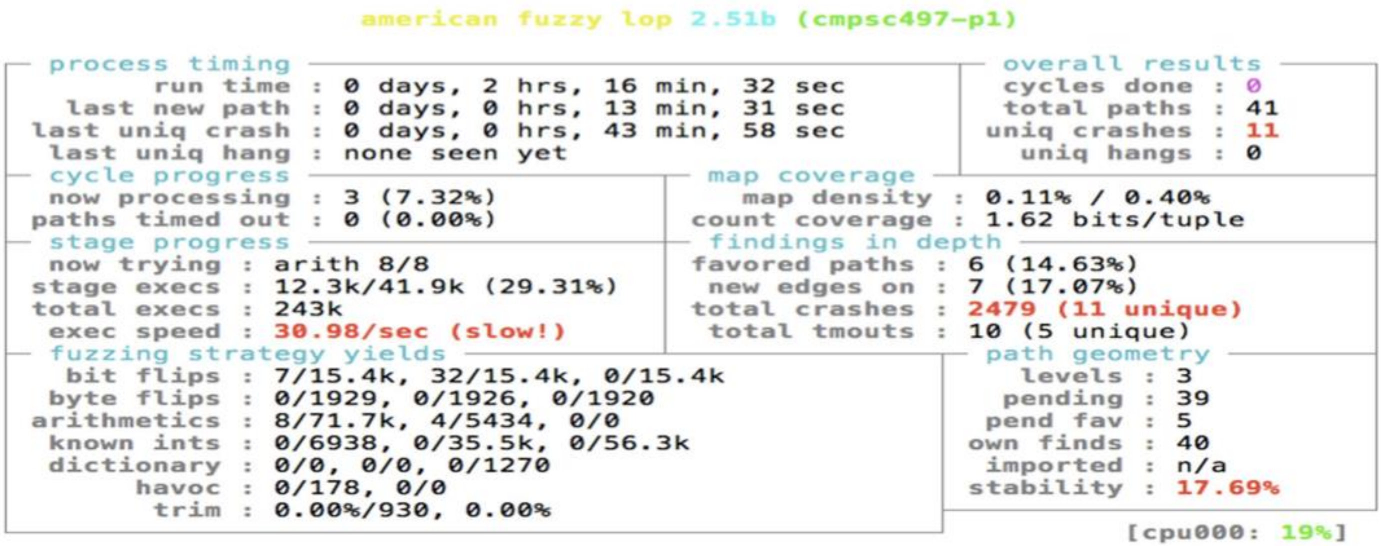


AFL

- Example of coverage-based fuzzing
  - American Fuzzy Lop (AFL)
  - "State of the practice" at this time

AFL Display

- Tracks the execution of the fuzzer



- Key information are
  - "total paths" – number of different execution paths tried
  - "unique crashes" – number of unique crash locations

Grey Box Fuzzing

- Finds flaws, but still does not understand the program
- Pros: Much better than black box testing
  - Essentially no configuration
  - Lots of crashes have been identified
- Cons: Still a bit of a stab in the dark
  - May not be able to execute some paths
  - Searches for inputs independently from the program
- Need to improve the effectiveness further

White Box Fuzzing

- Combines test generation with fuzzing
  - Test generation based on static analysis and/or symbolic execution – more later
  - Rather than generating new inputs and hoping that they enable a new path to be executed, compute inputs that will execute a desired path
    - And use them as fuzzing inputs
- Goal: Given a sequential program with a set of input parameters, generate a set of inputs that maximizes code coverage

Symbolic Execution

Symbolic execution

1976: *A system to generate test data and symbolically execute programs* (Lori Clarke)

1976: *Symbolic execution and program testing* (James King)

2005-present: practical symbolic execution

- Using SMT solvers
- Heuristics to control exponential explosion
- Heap modeling and reasoning about pointers
- Environment modeling
- Dealing with solver limitations

Classic symbolic execution

```
def f (x, y):
  if (x > y):
    x = x + y
    y = x - y
    x = x - y
    if (x - y > 0):
      assert false
  return (x, y)
```

Classic symbolic execution

```
def f (x, y):
   if (x > y):
      x = x + y
      y = x - y
      x = x - y
      if (x - y > 0):
         assert false
   return (x, y)
```

Execute the program on *symbolic values*.

Classic symbolic execution

$x \mapsto X$
$y \mapsto Y$

```
def f (x, y):
   if (x > y):
      x = x + y
      y = x - y
      x = x - y
      if (x - y > 0):
         assert false
   return (x, y)
```

Execute the program on *symbolic values*.

Symbolic state maps variables to symbolic values.

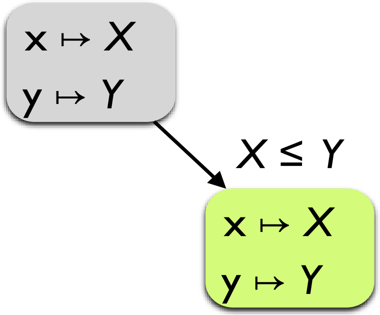
Classic symbolic execution

```
def f (x, y):
  if (x > y):
    x = x + y
    y = x - y
    x = x - y
    if (x - y > 0):
      assert false
  return (x, y)
```

Execute the program on *symbolic values*.

Symbolic state maps variables to symbolic values.

Path condition is a quantifier-free formula over the symbolic inputs that encodes all branch decisions taken so far.



Classic symbolic execution

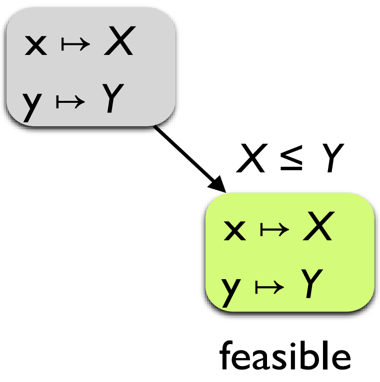
```
def f (x, y):
  if (x > y):
    x = x + y
    y = x - y
    x = x - y
    if (x - y > 0):
      assert false
  return (x, y)
```

Execute the program on *symbolic values*.

Symbolic state 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*.



Classic symbolic execution

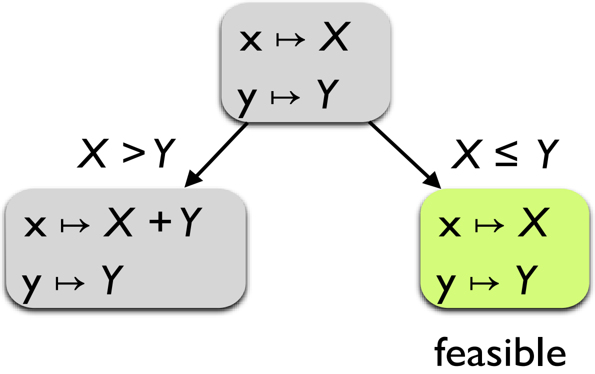
```
def f (x, y):
  if (x > y):
    x = x + y
    y = x - y
    x = x - y
    if (x - y > 0):
      assert false
  return (x, y)
```

Execute the program on *symbolic values*.

Symbolic state 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*.



Classic symbolic execution

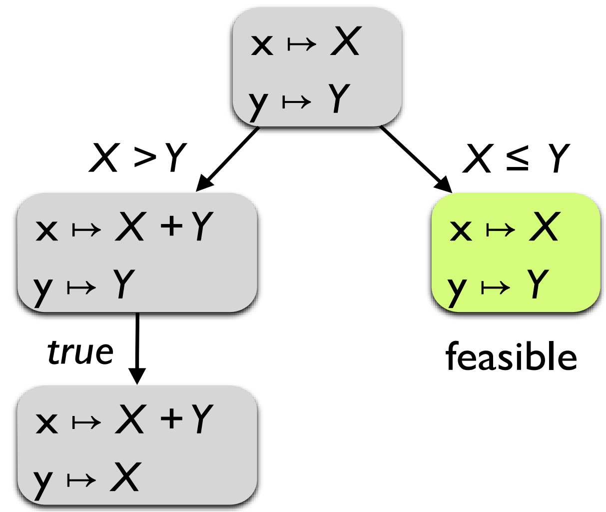
```
def f (x, y):
  if (x > y):
    x = x + y
    y = x - y
    x = x - y
    if (x - y > 0):
      assert false
  return (x, y)
```

Execute the program on *symbolic values*.

Symbolic state 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*.



Classic symbolic execution

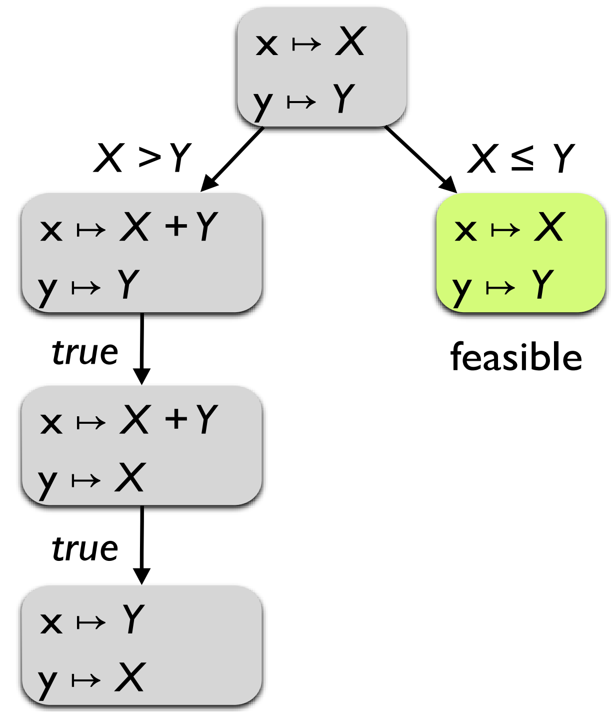
```
def f (x, y):
  if (x > y):
    x = x + y
    y = x - y
    x = x - y
    if (x - y > 0):
      assert false
  return (x, y)
```

Execute the program on *symbolic values*.

Symbolic state 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*.



Classic symbolic execution

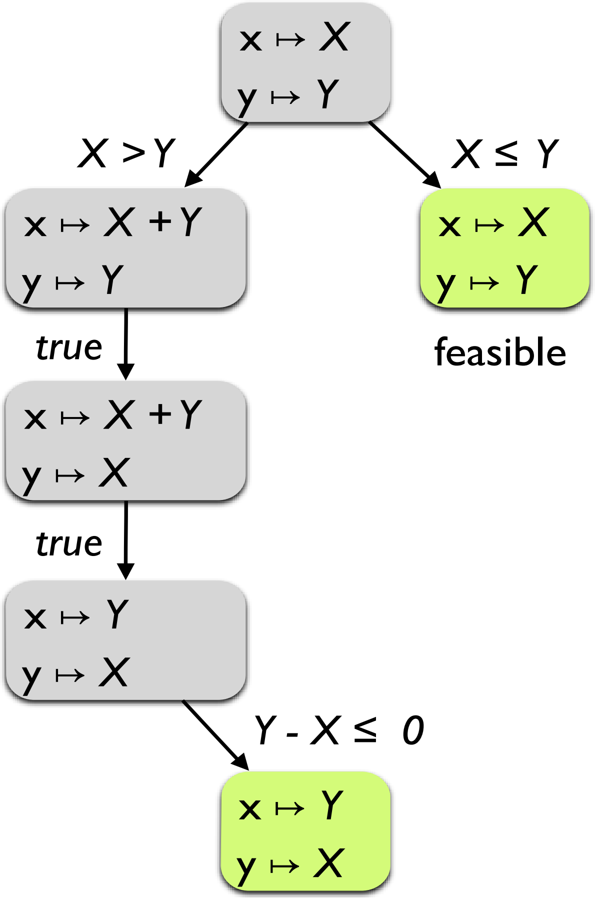
```
def f (x, y):
  if (x > y):
    x = x + y
    y = x - y
    x = x - y
    if (x - y > 0):
      assert false
  return (x, y)
```

Execute the program on *symbolic values*.

Symbolic state 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*.



Classic symbolic execution

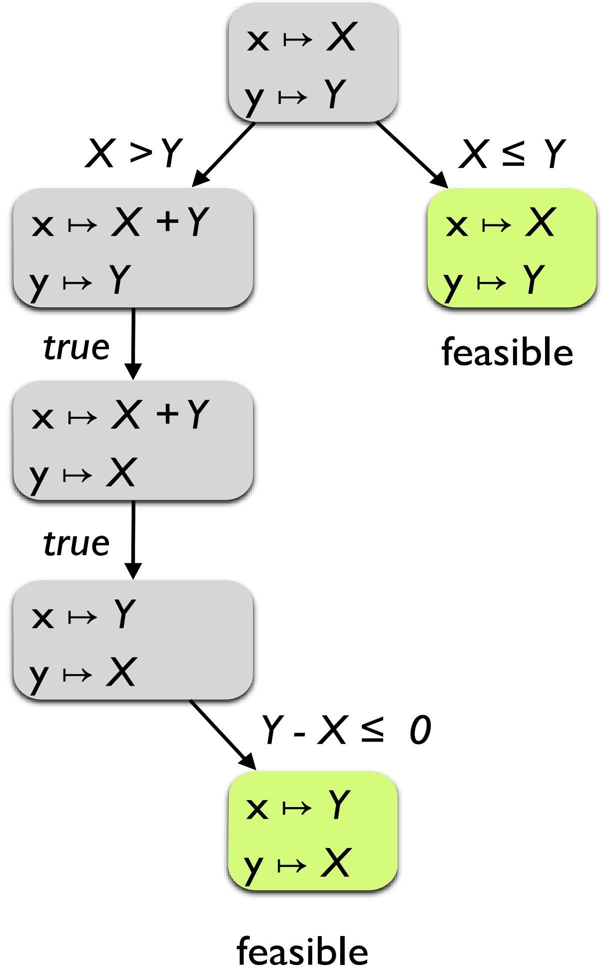
```
def f (x, y):
  if (x > y):
    x = x + y
    y = x - y
    x = x - y
    if (x - y > 0):
      assert false
  return (x, y)
```

Execute the program on *symbolic values*.

Symbolic state 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*.



Classic symbolic execution

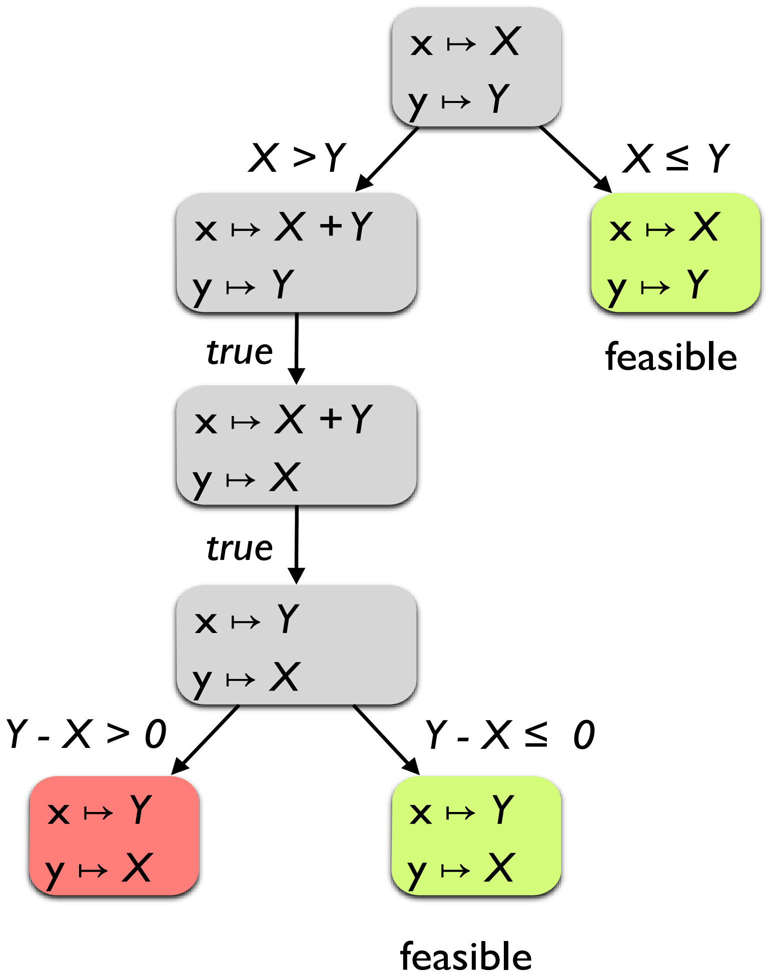
```
def f (x, y):
  if (x > y):
    x = x + y
    y = x - y
    x = x - y
    if (x - y > 0):
      assert false
  return (x, y)
```

Execute the program on *symbolic values*.

Symbolic state 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*.



Classic symbolic execution

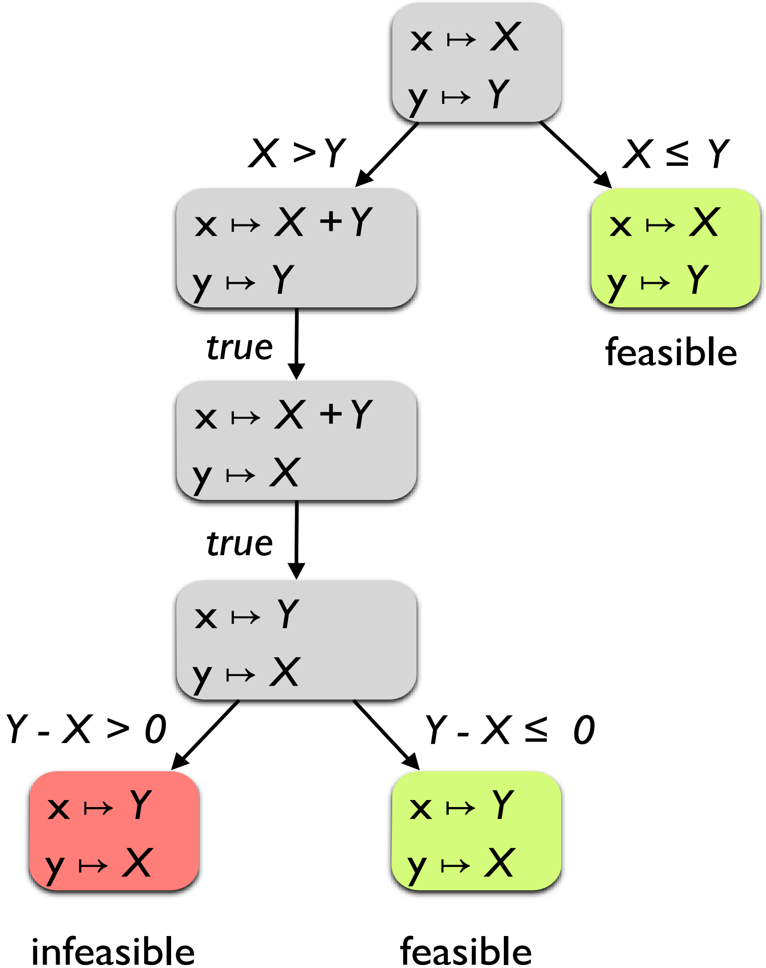
```
def f (x, y):
  if (x > y):
    x = x + y
    y = x - y
    x = x - y
    if (x - y > 0):
      assert false
  return (x, y)
```

Execute the program on *symbolic values*.

Symbolic state 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*.

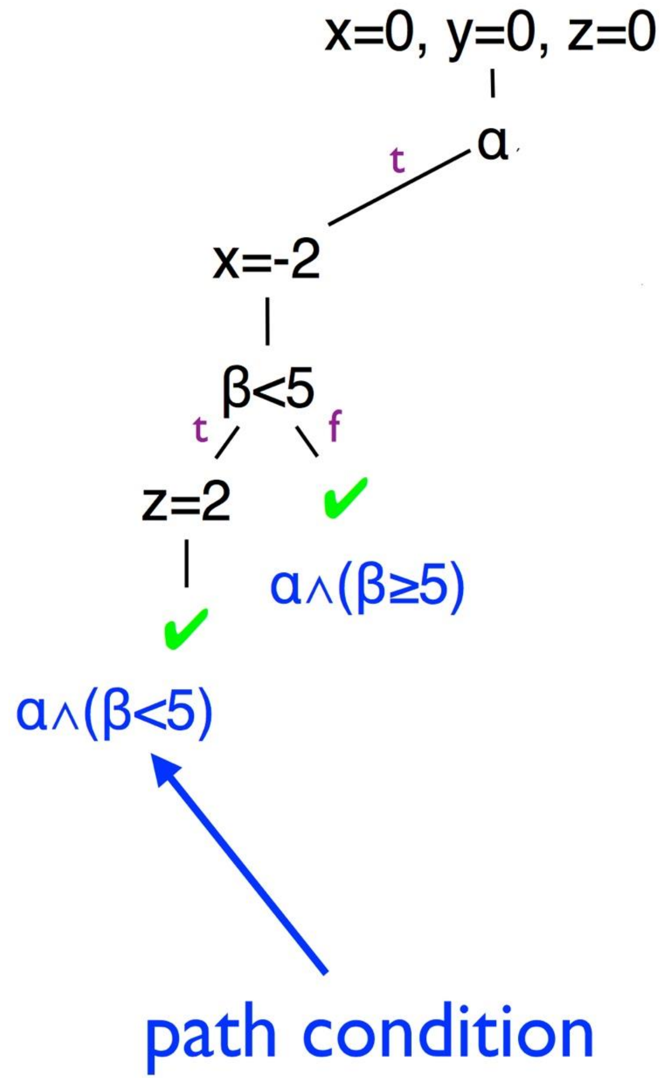


Another symbolic execution example

```
int a = α, b = β, c = γ;
                        // symbolic
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c)  { y = 1; }
  z = 2;
}
assert(x+y+z!=3)
```

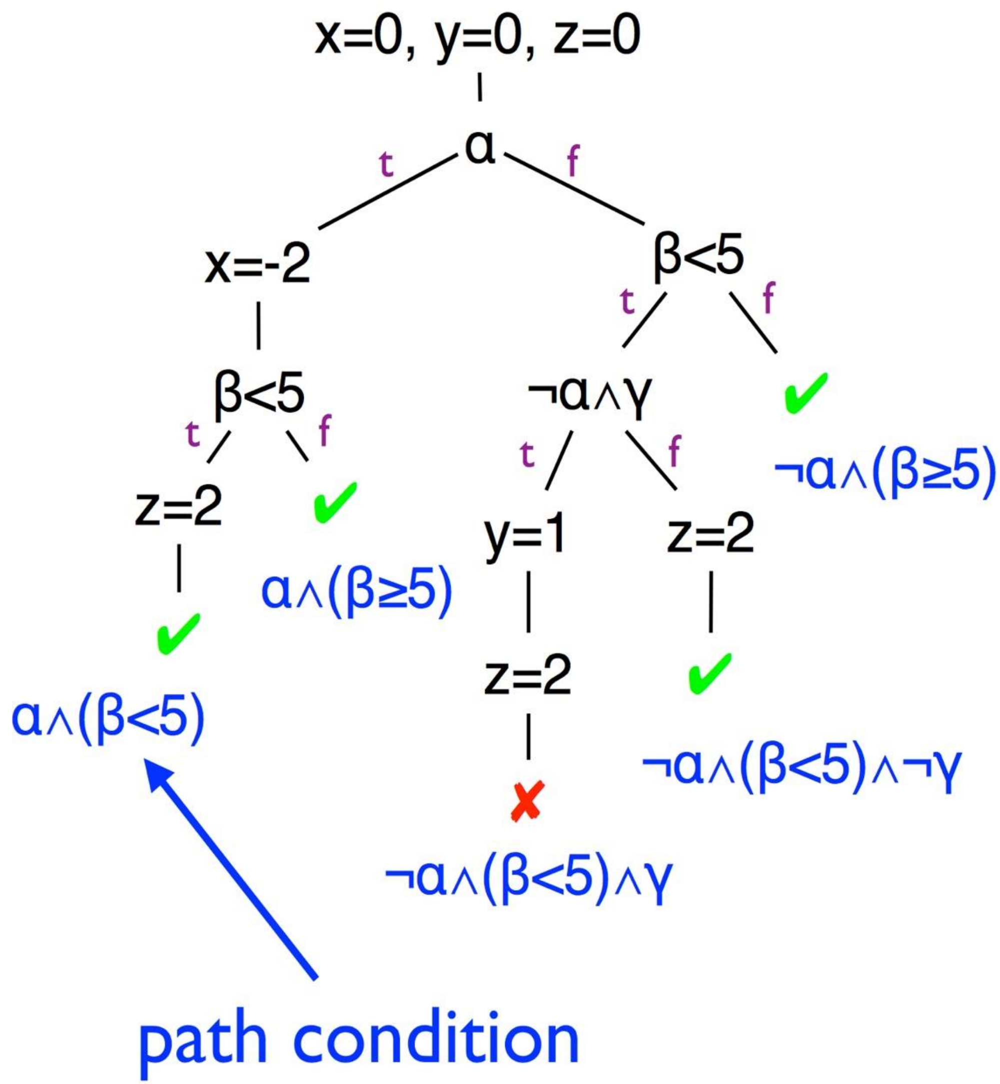
Another symbolic execution example

```
int a = α, b = β, c = γ;
                    // symbolic
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c)  { y = 1; }
  z = 2;
}
assert(x+y+z!=3)
```



Another symbolic execution example

```
int a = α, b = β, c = γ;
                    // symbolic
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c)  { y = 1; }
  z = 2;
}
assert(x+y+z!=3)
```

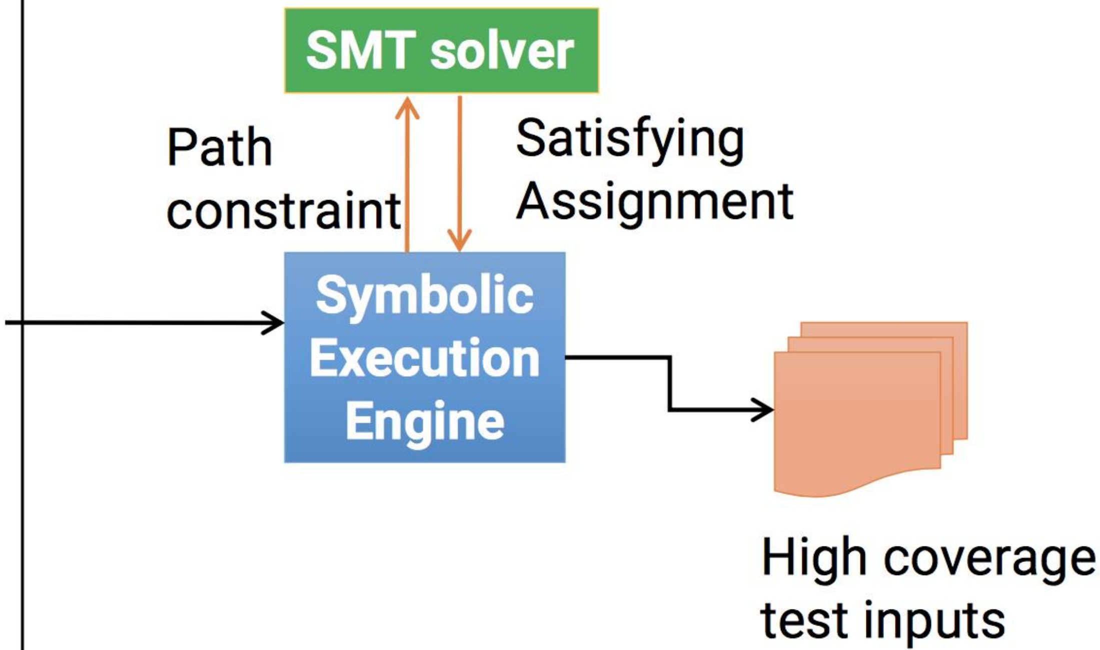


What's going on here?

- During symbolic execution, we are trying to determine if certain formulas are satisfiable
  - E.g., is a particular program point reachable?
    - Figure out if the path condition is satisfiable
  - E.g., is array access a[i] out of bounds?
    - Figure out if conjunction of path condition and $i<0 \lor i > a.length$ is satisfiable
  - E.g., generate concrete inputs that execute the same paths
- This is enabled by powerful SMT/SAT solvers
  - SAT = Satisfiability
  - SMT = Satisfiability modulo theory = SAT++
  - E.g. Z3, Yices, STP

Symbolic execution for software testing

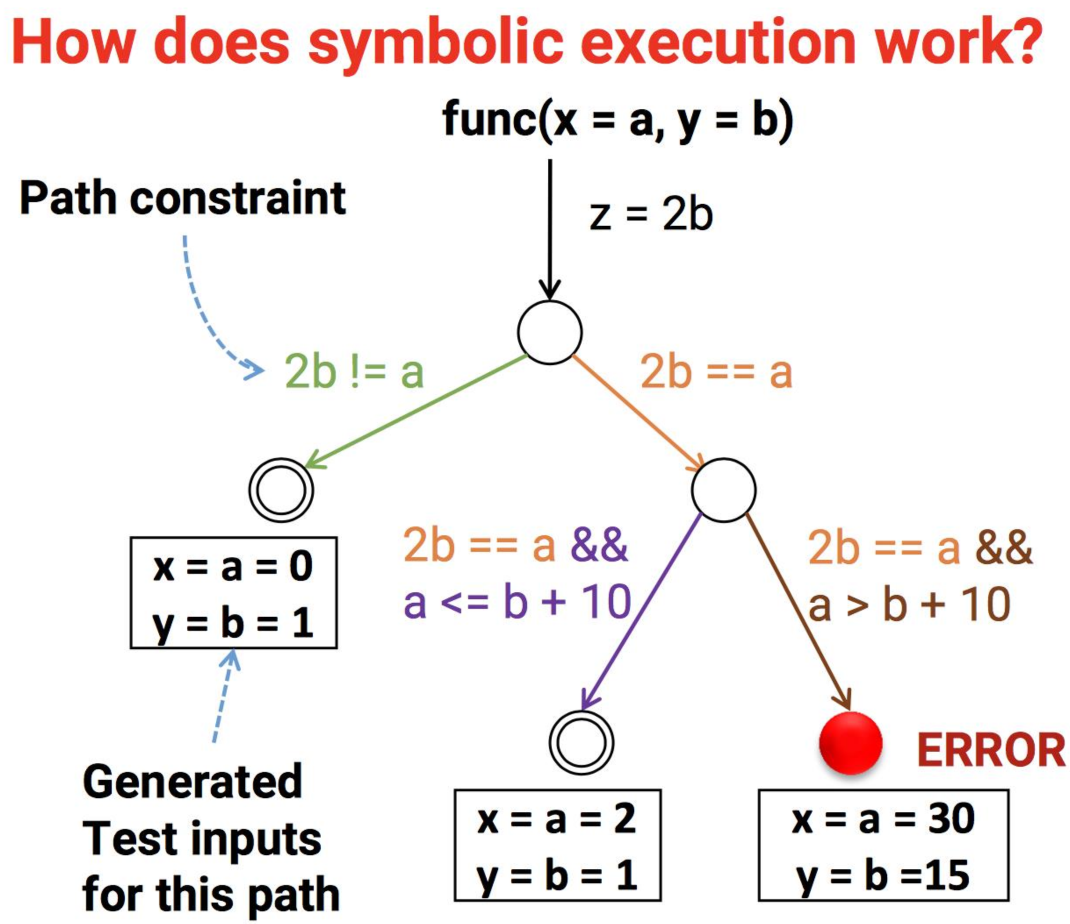
```
Void func(int x, int y){
    int z = 2 * y;
    if(z == x){
            if (x > y + 10)
                ERROR
    }
}
int main(){
    int x = sym_input();
    int y = sym_input();
    func(x, y);
    return 0;
}
```



Symbolic Execution

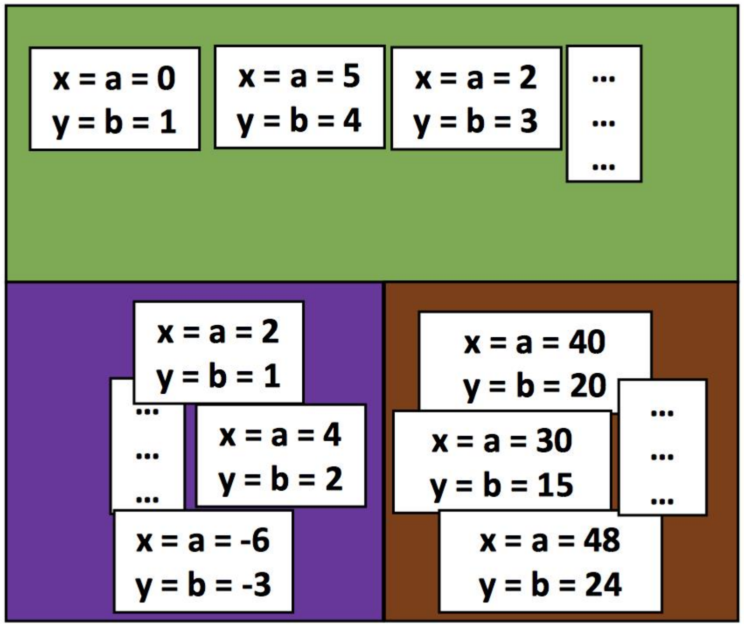
How does symbolic execution work?

```
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;
}
```



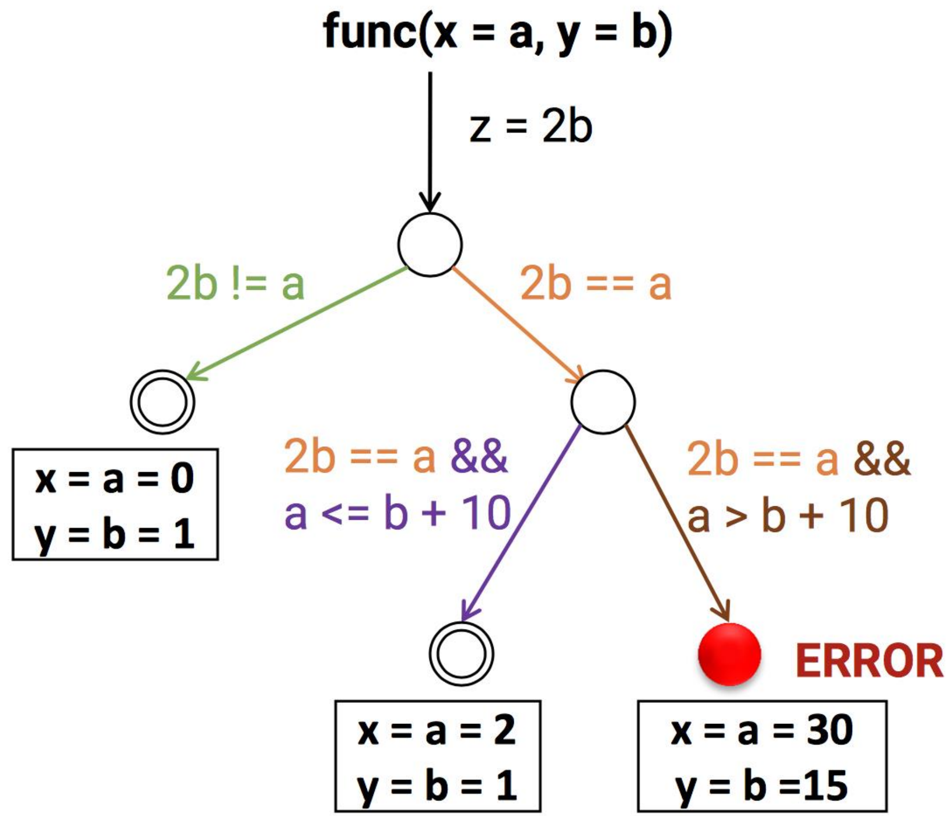
Note: Require inputs to be marked as symbolic

Equivalence classes of inputs



Path constraints represent equivalence classes of inputs

How does symbolic execution work?



Concolic (Concrete + Symbolic) Execution

Fuzz (Random) Testing

- **Very low probability of reaching an error**
- **Problematic for complex data structures**

```
Example ( ) {
    s = readString();
    if (s[0]=='I' && s[1]=='C' &&
        s[2]=='S' && s[3]=='E' &&
        s[4]=='2' && s[5]=='0' &&
        s[6]=='0' && s[7]=='7') {
          printf("Am I here?");
    }
}
```

Input domain = {'0', '2', '7', 'C', 'E', 'I', 'S'}
Probability of reaching printf = $7^{-8} \approx 10^{-7}$

Fast and Inexpensive

Concolic Testing

- **Combine concrete testing (concrete execution) and symbolic testing (symbolic execution)**

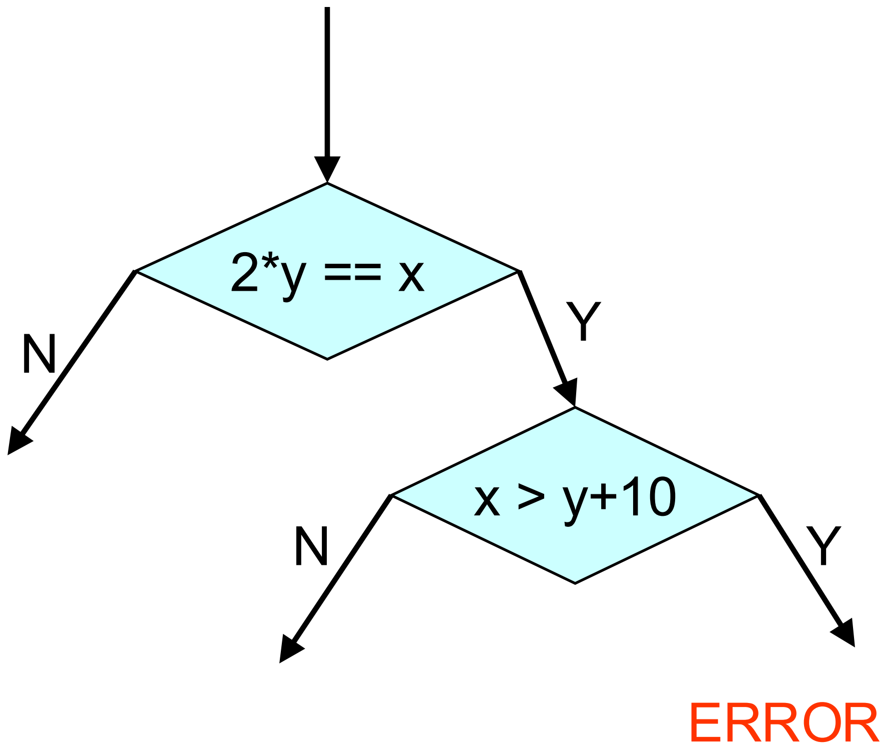
Concrete + Symb**olic** = Concolic

Example

```
int double (int v) {

    return 2*v;
}

void testme (int x, int y) {

    z = double (y);

    if (z == x) {

            if (x > y+10) {

                ERROR;
            }
    }

}
```

Example

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

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



Concolic Testing Approach

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

void testme (int x, int y) {

    z = double (y);

    if (z == x) {

        if (x > y+10) {

            ERROR;
        }
    }

}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 22, y = 7	$x = x_0$, $y = y_0$	

Concolic Testing Approach

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

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

	Concrete Execution	Symbolic Execution	
	concrete state	symbolic state	path condition
	$x = 22$, $y = 7$, $z = 14$	$x = x_0$, $y = y_0$, $z = 2*y_0$	

Concolic Testing Approach

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

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

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
		$2*y_0 \,!= x_0$
x = 22, y = 7, z = 14	$x = x_0$, $y = y_0$, $z = 2*y_0$	

Concolic Testing Approach

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

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

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 22, y = 7, z = 14	$x = x_0$, $y = y_0$, $z = 2*y_0$	$2*y_0 \,!= x_0$

Solve: $2*y_0 == x_0$
Solution: $x_0 = 2$, $y_0 = 1$

Concolic Testing Approach

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

void testme (int x, int y) {

    z = double (y);

    if (z == x) {

        if (x > y+10) {

            ERROR;
        }
    }

}
```

	Concrete Execution	Symbolic Execution	
	concrete state	symbolic state	path condition
	$x = 2, y = 1$	$x = x_0, y = y_0$	

Concolic Testing Approach

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

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

	Concrete Execution	Symbolic Execution	
	concrete state	symbolic state	path condition
	$x = 2$, $y = 1$, $z = 2$	$x = x_0$, $y = y_0$, $z = 2^*y_0$	

Concolic Testing Approach

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

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

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
		$2*y_0 == x_0$
x = 2, y = 1, z = 2	$x = x_0$, $y = y_0$, $z = 2*y_0$	

Concolic Testing Approach

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

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

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
		$2*y_0 == x_0$
		$x_0 <= y_0+10$
$x = 2, y = 1, z = 2$	$x = x_0, y = y_0, z = 2*y_0$	

Concolic Testing Approach

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

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

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
		$2*y_0 == x_0$
		$x_0 <= y_0+10$
$x = 2, y = 1, z = 2$	$x = x_0, y = y_0, z = 2*y_0$	

Solve: $(2*y_0 == x_0)$ **&** $(x_0 > y_0 + 10)$

Solution: $x_0 = 30, y_0 = 15$

Concolic Testing Approach

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

void testme (int x, int y) {

    z = double (y);

    if (z == x) {

        if (x > y+10) {

            ERROR;
        }
    }

}
```

concrete state	symbolic state	path condition
x = 30, y = 15	$x = x_0$, $y = y_0$	

Concrete Execution

Symbolic Execution

Concolic Testing Approach

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

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

Concrete Execution	Symbolic Execution	
	symbolic state	path condition
		$2*y_0 == x_0$
		$x_0 > y_0+10$
x = 30, y = 15	$x = x_0$, $y = y_0$	

Program Error

End of Lecture 11