CSCI426/CSCI926
Software Testing and Analysis

Symbolic Execution

Acknowledgement: Some slides are adapted from Omar Chowdhury, Jeff Foster, and Pezze & Young

Testing

- Fits well with developer intuitions
- In practice, most common form of bug-detection
- But each test explores only one possible execution of the system
- Depends on the quality of the test cases or inputs
- Provides little in terms of coverage

Symbolic Execution

- Key idea: generalize testing by using unknown symbolic variables in evaluation
- Symbolic executor executes program, tracking symbolic state.
- Builds *predicates* that characterize
  - Conditions for executing paths
  - Effects of the execution on program state
- Bridges program behavior to logic
- Finds important applications in
  - program analysis
  - test data generation
  - formal verification (proofs) of program correctness

Let's work through this example

```
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;
}
```

- What are the test cases to cover all the paths in this function?

Obvious Questions?

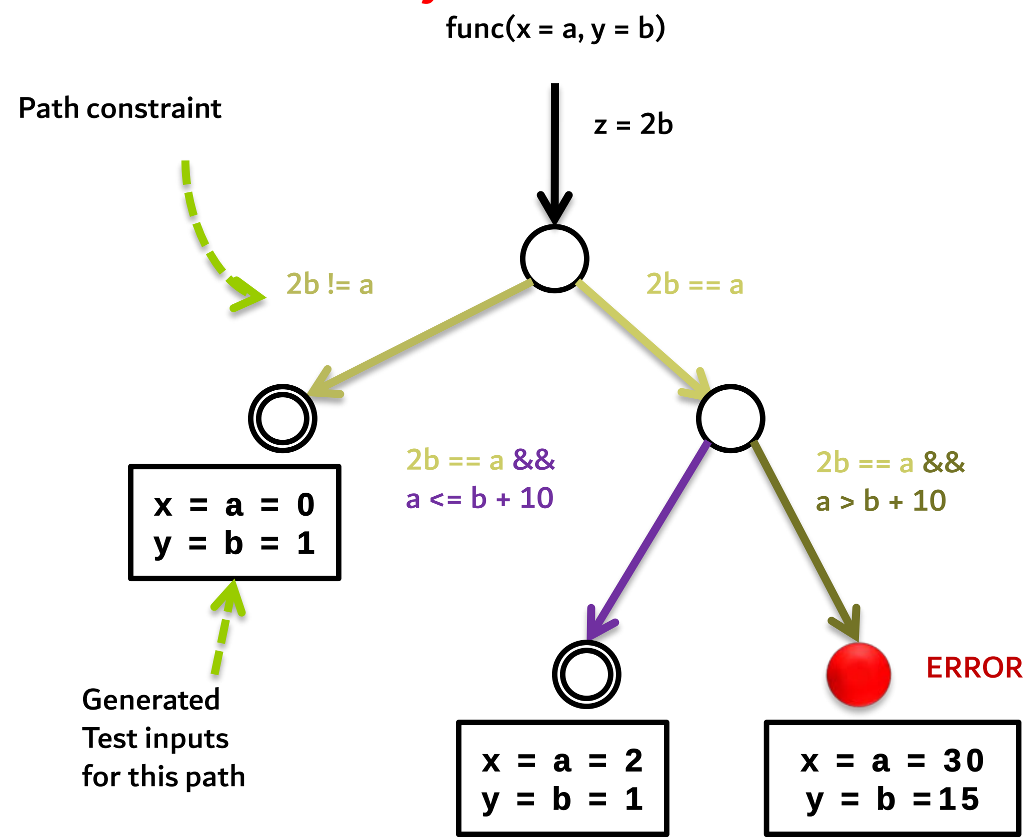
Ca… of test
…me how make it
…tomatic?

Yes, we can.

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;
}
```

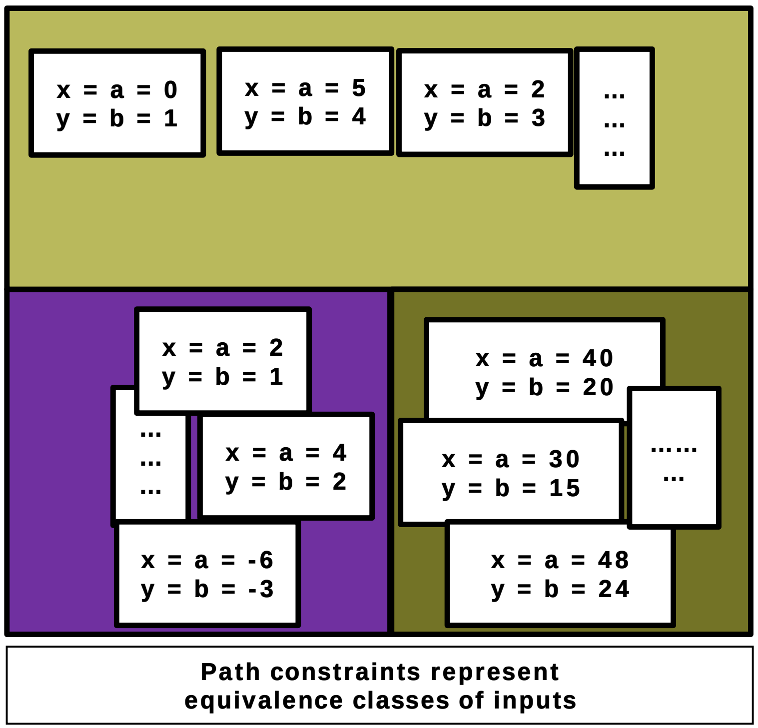
How does symbolic execution work?

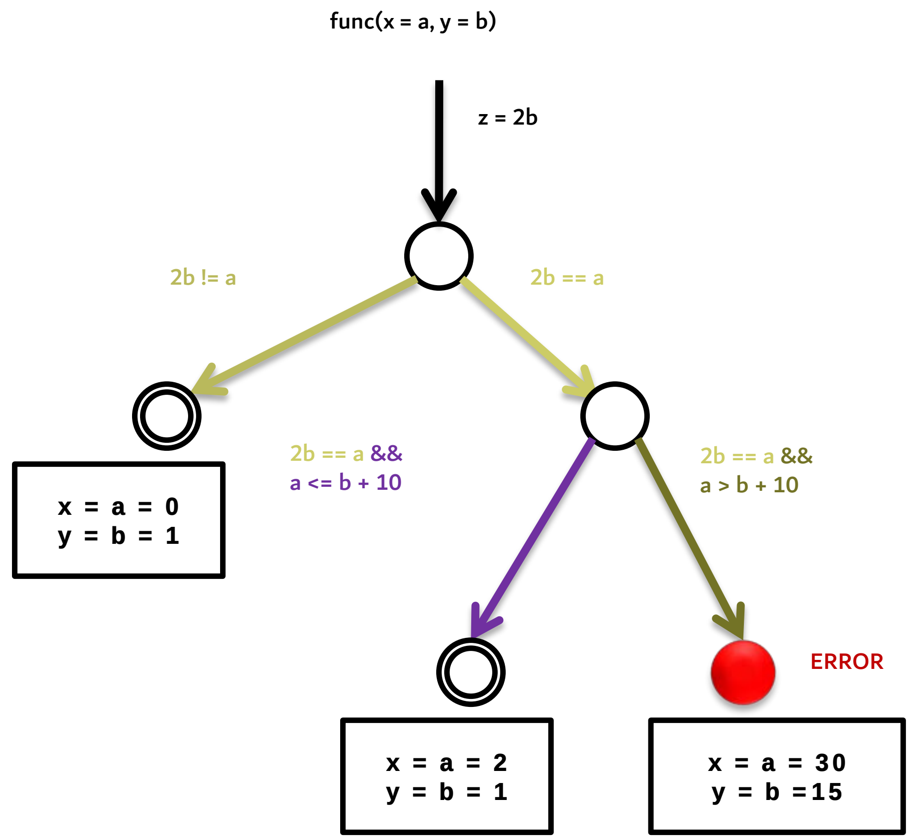


Note: Require inputs to be marked as symbolic

Symbolic Execution

How does symbolic execution work?





Dealing with branching statements

a sample program:

*Binary search for key in sorted array **dictKeys**, returning corresponding value from **dictValues** or null if key does not appear in dictKeys.*

Standard binary search algorithm as described in any elementary text on data structures and algorithms.

```
char *binarySearch( char *key, char *dictKeys[ ],
        char *dictValues[ ],  int dictSize) {

  int low = 0;
  int high = dictSize - 1;
  int mid;
  int comparison;

  while (high >= low) {
    mid = (high + low) / 2;
    comparison = strcmp( dictKeys[mid], key );
    if (comparison < 0) {
      low = mid + 1;
    } else if ( comparison > 0 ) {
      high = mid - 1;
    } else {
      return dictValues[mid];
    }
  }
  return 0;
```

Symbolic state

- Values are expressions over symbols
- Executing statements computes new expressions

Execution with **concrete values**

before

low	12
high	15
mid	-

mid = (high+low)/2

after

low	12
high	15
mid	13

Execution with **symbolic values**

before

low	L
high	H
mid	-

mid = (high+low)/2

after

Low	L
high	H
mid	(L+H)/2

Dealing with branching statements

Branching stmt

```
char *binarySearch( char *key, char *dictKeys[ ],
     char *dictValues[ ],  int dictSize) {

  int low = 0;
  int high = dictSize - 1;
  int mid;
  int comparison;

  while (high >= low) {
    mid = (high + low) / 2;
    comparison = strcmp( dictKeys[mid], key );
    if (comparison < 0) {
      low = mid + 1;
    } else if ( comparison > 0 ) {
      high = mid - 1;
    } else {
      return dictValues[mid];
    }
  }
  return 0;
```

Executing while (high >= low) {

before

	low = 0
and	high = (H-1)/2 -1
and	mid = (H-1)/2

while (high >= low) {

Add an expression that records the condition for the execution of the branch (PATH CONDITION)

after

	low = 0
and	high = (H-1)/2 -1
and	mid = (H-1)/2
and	(H-1)/2 - 1 >= 0

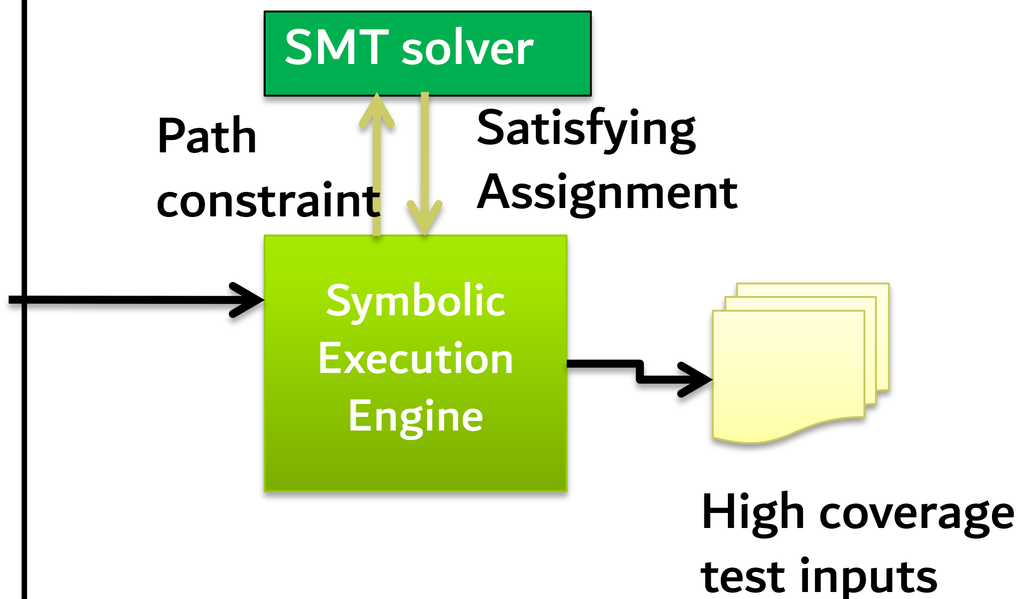
if the TRUE branch was taken

… and not((H-1)/2 - 1 >= 0)

if the FALSE branch was taken

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

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 implementations**: KLEE, Java PathFinder, etc.

Symbolic Execution (cont.)

- 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 (cont.)

- During symbolic execution, we are trying to determine if certain formulas are satisfiable
  - E.g., is a particular program point reachable (feasible vs. infeasible paths)?
- 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

Summary information

- Symbolic representation of paths may become extremely complex
- We can simplify the representation by replacing a complex condition P with a weaker condition W such that

  P => W (P implies W)

- W describes the path with less precision
- W is a *summary* of P

Example of summary information

(Referring to Binary search: Line 17 , *mid = (high+low)/2*)

- If we are reasoning about the correctness of the binary search algorithm, the complete condition:

  low = L
  and high = H
  and mid = M
  and M = (L+H)/2

- Contains more information than needed and can be replaced with the weaker condition:

  low = L
  and high = H
  and mid = M
  and L <= M <= H

- The weaker condition contains less information, but still enough to reason about correctness.

Loops and assertions

- A predicate stating what *should* be true at a given point can be expressed in the form of an **assertion**
- The number of execution paths through a program with loops is potentially infinite
- To reason about program behavior in a loop, we can place within the loop an **invariant**:
  - assertion that states a predicate that is expected to be true each time execution reaches that point.
- Each time program execution reaches the invariant assertion, we can weaken the description of program state:
  - If predicate P represents the program state
  - and the assertion is W
  - we must first ascertain P => W
  - and then we can substitute W for P

Pre- and post-conditions

- Suppose:
  - every loop contains an assertion
  - there is an assertion at the beginning of the program
  - a final assertion at the end
- Then:
  - every possible execution path would be a sequence of segments from one assertion to the next.
- Terminology:
  - Precondition: The assertion at the beginning of a segment,
  - Postcondition: The assertion at the end of the segment

Verifying program correctness

- If for each program segment we can verify that
  - Starting from the precondition
  - Executing the program segment
  - The postcondition holds at the end of the segment
- Then
  - We verify the correctness of an infinite number of program paths

Example

```
char *binarySearch( char *key, char *dictKeys[ ],
    char *dictValues[ ], int dictSize) {

 int low = 0;
 int high = dictSize - 1;
 int mid;
 int comparison;

 while (high >= low) {
   mid = (high + low) / 2;
   comparison = strcmp( dictKeys[mid], key
   if (comparison < 0) {
     low = mid + 1;
   } else if ( comparison > 0 ) {
     high = mid - 1;
   } else {
     return dictValues[mid];
   }
 }
 return 0;
```

Precondition: is sorted:

`Forall{i,j} 0 <= i < j < size : dictKeys[i] <= dictKeys[j]`

Invariant: in range

`Forall{i} 0 <= i < size : dictKeys[i] = key => low <= i <= high`

Executing the loop once…

Initial values:
```
low = L
and high = H
```

Precondition
Forall{i,j} 0 <= i < j < size
dictKeys[i] <= dictKeys[j]

Instantiated invariant:
```
Forall{i,j} 0 <= i < j < size :
dictKeys[i] <= dictKeys[j]
and Forall{k} 0 <= k < size :
dictKeys[k] = key => L <= k <= H
```

Invariant
Forall{i} 0 <= i < size :
dictKeys[i] = key =>
low <= i <= high

After executing: `mid = (high + low)/2`

```
low = L
and high = H
and mid = M
and Forall{i,j} 0 <= i < j < size :
dictKeys[i] <= dictKeys[j]
and Forall{k} 0 <= k < size :
dictKeys[k] = key => L <= k <= H
```
and H >= M >= L

Note.
M = (L+H)/2
(possibly rounded to closest smallest integer)

….

Note. low = mid + 1 is executed after the "**if**" condition is true

Note. low = mid + 1 is executed after the "**if**" condition is true

…executing the loop once

After executing the loop

and the "**if**" condition is true

low = M+1
and high = H
and mid = M
and Forall{i,j} 0 <= i < j < size :
dictKeys[i] <= dictKeys[j]
and Forall{k} 0 <= k < size :
dictKeys[k] = key => L <= k <= H
and H >= M >= L
and dictkeys[M]<key

The new instance of the invariant:

Forall{i,j} 0 <= i < j < size :
dictKeys[i] <= dictKeys[j]
and Forall{k} 0 <= k < size :
dictKeys[k] = key **=> M+1 <= k <= H**

Invariant
Forall{i} 0 <= i < size :
dictKeys[i] = key =>
low <= i <= high

If the invariant is satisfied, the loop is correct wrt the preconditions and the invariant

From the loop to the end

If the invariant is satisfied, but the condition is false:

Invariant
Forall{i} 0 <= i < size :
dictKeys[i] = key
=>
low <= i <= high

```
low = L
and high = H
and Forall{i,j} 0 <= i < j <
size :
dictKeys[i] <= dictKeys[j]
and Forall{k} 0 <= k < size :
dictKeys[k] = key => L <= k <= H
and L > H
```

If the condition satisfies the post-condition, the program is correct wrt the pre- and post-condition.

What does the above statement mean?

It means that no such *k* exists.

How does Symbolic Execution Find bugs?

- It is possible to extend symbolic execution to help us catch bugs
- **How**: Dedicated checkers
  - **Divide by zero example** --- y = x / z where x and z are symbolic variables and assume current path condition is ***f***
  - Even though we only fork in branches, we will now fork in the division operator
  - One branch in which z = 0 and another where z !=0
  - We will get two paths with the following constraints:

    z = 0 && ***f***   z != 0 && ***f***
  - Solving the constraint z = 0 && ***f*** will give us concrete input values that will trigger the divide by zero error.

Write a dedicated checker for each kind of bug (e.g., buffer overflow, integer overflow, integer underflow)

How does Symbolic Execution Find bugs? Example

Use symbolic execution to find a test case which reveal a division by zero bug for the following program:

```
int foo(int i) {
        int j = 2 * i;
        i = i+1;
        i = i * j;
        if (i < 1)
                i = -i;
        i = j/i;
        return i;
}
```

Pen and paper exercise

- Apply symbolic execution on the following program. Identify **feasible** paths and **infeasible** paths.

```
void hello(int x, int y) {
        int t = 0;
        if (x > y) {
                t = x;
        } else {
                t = y;
        }

        if (t < x) {
                print("Hello World");
        }
}
```