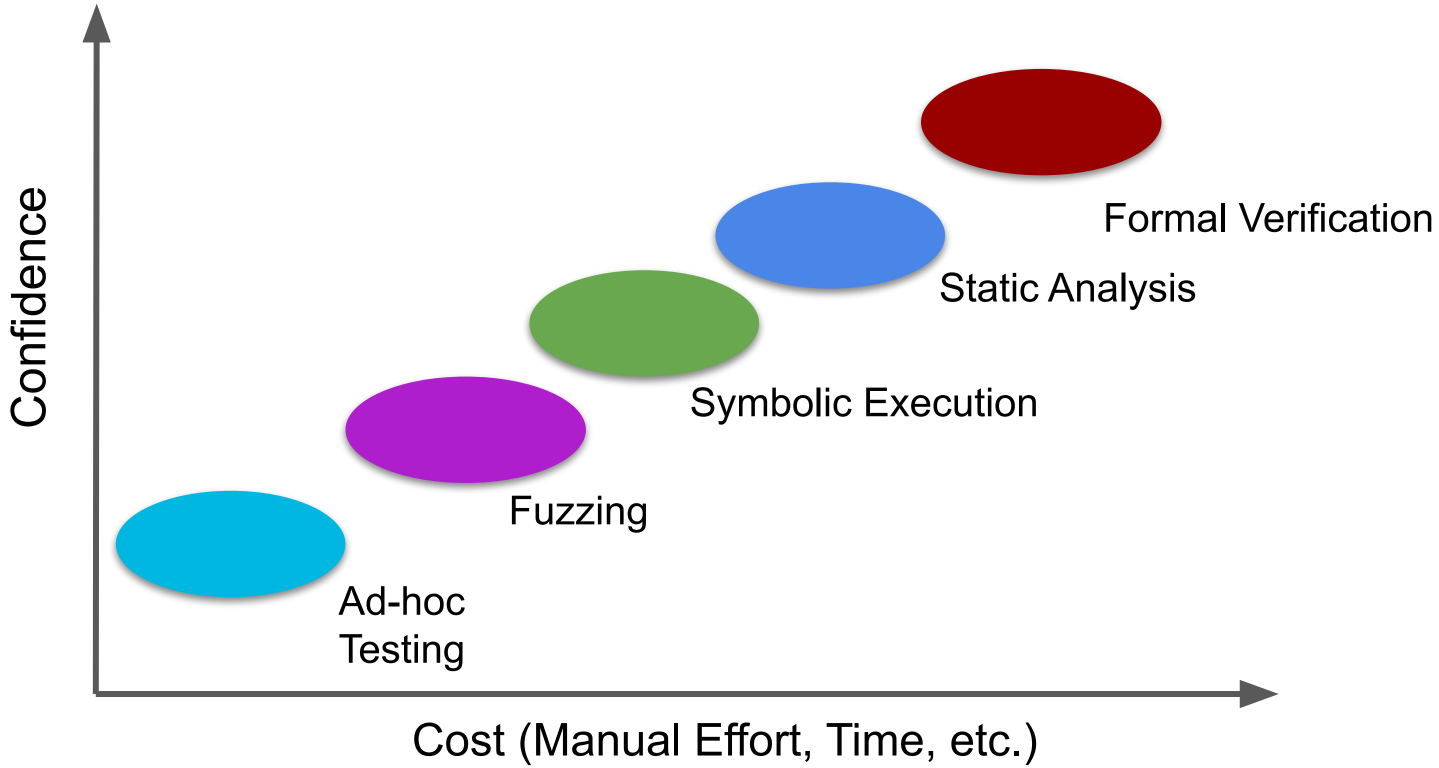
Symbolic Execution

Announcements

- No class 4/21 and 4/30
- Lab8 - due last night
- HW3 - due tonight
- Project part 1 - Resubmission deadline 4/13 at midnight
    - Part 2 deadline extended to 4/21

Landscape of Program Analysis Techniques



Motivation

How would the existing techniques we've seen attempt to find this bug?

1. Randoop
2. EvoSuite
3. AFL
4. Formal Verification - SAT solving
5. Formal Verification - deductive reasoning

```
public static boolean foo(String[] args) {
   int a = Integer.parseInt(args[0]);
   int b = Integer.parseInt(args[1]);

   boolean flag;

   if (a > 10 && b < 100) {
      int x = 2 * a + 3 * b;
      int y = 5 * a - b;

      if (x == 245 && y == 111) {
         System.out.println("You triggered the secret logic!");
         flag = true; //bug
      } else {
         System.out.println("Not quite the magic numbers.");
      }

   } else {
      System.out.println("Input out of range.");
   }

  return flag;
}
```

Symbolic Execution

- A middle ground between random testing and formal verification
- Treats the program as *symbols* rather than concrete values
- Instead of running the program with specific inputs like `x = 5`, use a placeholder value $x = \alpha$ where alpha is a placeholder representing any value
- As the program “executes”, keep track of the symbolic expression for each value

Concrete Execution

```
assume (N >= 0);
X := 0;
Y := 1; //P1
while X < N do {
  X := X + 1;
  Y := Y * X;
}
assert (Y = N!);
```

Inputs are concrete values

- A concrete state: maps from variables to concrete values

  when N=3, and after P1, we have the state

  {X=0, Y=1, N=3}

- Execution of a program statement
  - Go from an input concrete state to an output concrete state
  - "`X=X+1`" goes from state {X=0, Y=1, N=3} to {X=1,Y=1, N=3}

Concrete Execution

JVM holds LVT values for

`a, b, flag, x, y, args`

During execution of each program statement, the LVTs are updated.

Requires concrete inputs for args

```
public static boolean foo(String[] args) {
    int a = Integer.parseInt(args[0]);
    int b = Integer.parseInt(args[1]);

    boolean flag;

    if (a > 10 && b < 100) {
        int x = 2 * a + 3 * b;
        int y = 5 * a - b;

        if (x == 245 && y == 111) {
            System.out.println("You triggered the secret logic!");
            flag = true; //bug
        } else {
            System.out.println("Not quite the magic numbers.");
        }

    } else {
        System.out.println("Input out of range.");
    }

  return flag;
}
```

Symbolic Execution

Inputs are represented symbolically

$\alpha 1$, $\alpha 2$, $\alpha 3$, …

- Variables get symbolic values
- A symbolic value is either a:

  constant (e.g., an integer constant),

  symbol (αi)

  expression formed from αi and constants ($\alpha 1 + \alpha 2$, $3\alpha 3$)

Symbolic Execution

args = [α1, α2]

```
public static boolean foo(String[] args) {
   int a = Integer.parseInt(args[0]);
   int b = Integer.parseInt(args[1]);

   boolean flag;

   if (a > 10 && b < 100) {
      int x = 2 * a + 3 * b;
      int y = 5 * a - b;

      if (x == 245 && y == 111) {
         System.out.println("You triggered the secret logic!");
         flag = true; //bug
      } else {
         System.out.println("Not quite the magic numbers.");
      }

   } else {
      System.out.println("Input out of range.");
   }

  return flag;
}
```

Symbolic States

A symbolic state consists of:

- A variable state:
  - Map from variable to symbolic values
  - {X: $2\alpha1 + 3\alpha2$, Y: $5\alpha1 - \alpha2$ }
- A path condition (PC):
  - A boolean condition that must hold when the program reaches this point
  - $(2\alpha1 + 3\alpha2 = 245) \land (5\alpha1 - \alpha2 = 111)$

Symbolic Execution Rules

For each program statement, we need to define a “rule” for how the values get updated

Similar to deductive reasoning rules

Before and after each rule statement we have a **variable state (VS)** and a **path condition (PC)**

Notation

VS_e = variable state at the entry of statement S

VS_x = variable state at the exit of statement S

PC_e = path condition at the entry of statement S

PC_x = path condition at the exit of statement S

Init: every input variable is assigned a symbol and PC = True

Assignment `(X := E)`

$VS_x = VS_e\ [X \rightarrow VS_e(E)]$

$PC_x = PC_e$

The new value of X is the symbolic value of E

Path condition is unchanged

A Simple Example

// input variables: A,B,X,Y,Z

{A:α1, B:α2, X:α3, Y:α4, Z:α5}, True

X := A + B;

{A:α1, B:α2, X:α1+α2, Y:α4, Z:α5} , True

Y := A - B;

{A:α1, B:α2, X:α1+α2, Y:α1-α2, Z:α5} , True

Z := X + Y

{A:α1, B:α2, X:α1+α2, Y:α1-α2, Z:(α1+α2)+(α1-α2)} , True

{A:α1, B:α2, X:α1+α2, Y:α1-α2, Z: 2α1} , True

Assume B

- Variable state unchanged

$$VS_x = VS_e$$

- Path condition adds the assumption

$$PC_x = PC_e \land VS_e(B)$$

Assert B

- If PC_e implies $VS_e(B)$

  $VS_x = VS_e$

  $PC_x = PC_e$

- If PCe does not imply VSe(B)

  print “assertion failed“

  terminate the evaluation

Example

//Inputs A, B, X, Y, and Z

{A:α1, B:α2, X:α3, Y:α4, Z:α5}, True

assume (A>B);

{A:α1, B:α2, X:α3, Y:α4, Z:α5}, $\alpha\alpha$1>$\alpha\alpha$2

X := A + B;

{A:α1, B:α2, X:α1+α2, Y:α4, Z:α5} , α1>α2

Y := A - B;

{A:α1, B:α2, X:α1+α2, Y:α1-α2, Z:α5} , α1>α2

Z := X + Y

{A:α1, B:α2, X:α1+α2, Y:α1-α2, Z:(α1+α2)+(α1-α2)} , α1>α2

assert (X=A+B ∧ Y=A-B ∧ Z=2*A ∧ Y>0);

α1>α2 → (α1+α2 = α1+α2 ∧ α1-α2 = α1-α2 ∧ α1+α2 +α1-α2 = 2α1 ∧ α1-α2>0) ???

Exercise

```
//inputs A, B, X
assume (A=2*B)
X := A + B;
X := X - B;
X := X - 2*B;
assert (X=0)
```

Conditionals

If B then S1 else S2

Three cases:

1. PC_e -> VS_e (B)
2. PC_e -> ! VS_e (B)
3. Path condition does not imply either the if or else branch

Conditionals

If B then S1 else S2

Case one: Current path condition implies the if condition

1. PC_e -> $VS_e(B)$

Add the if-condition to our path condition

$PC_x = PC_e \land VS_e(B)$

Variable state remains unchanged

$VS_x = VS_e$

Conditionals

If B then S1 else S2

Case two: Current path condition implies the else condition

2. PC_e -> $!VS_e$ (B)

Add the **negation** if-condition to our path condition

$PC_x = PC_e \land !VS_e(B)$

Variable state remains unchanged

$VS_x = VS_e$

Conditionals

If B then S1 else S2

Case three: Current path condition implies neither the if condition or its negation

Need to consider both!

Branching behavior

Our execution splits!

We need to keep track of path conditions and variable states for both options!

Example

```
//inputs X and Y
if X< 0
    Y := -X;
else
    Y := X;
assert (Y>=0)
```

Example

How many total paths are in this program?

Suddenly we have a ton of branches to keep track of and a lot of calls to the SAT solver....

Path explosion problem!

```
public static void main(String[] args) {
        int x = new Scanner(System.in).nextInt();
        int y = 0;
        if (x > 0) {
              y = x * x;
         } else if (x == 0) {
             y = -10;
        } else {
            if (x > -5) y = x + 5;
            else y = x * -1;
        }

        if (x % 2 == 0) y = x;
        Else y = x + 1;

        Assert ( x > y);
}
```

Loops

While B do S

Three cases:

1. PC_e -> VS_e (B)
2. PC_e -> ! VS_e (B)
3. Path condition does not imply either the

Loops

While B do S

Case one: Current path condition implies the loop condition

1. $PC_e \rightarrow VS_e(B)$

Add the loop condition to our path condition

$PC_x = PC_e \land VS_e(B)$

Variable state remains unchanged

$VS_x = VS_e$

Loops

- Other cases follow similarly to if-conditions.
- Note: no loop invariants needed!

- How do we know how many times to execute?
  - We don’t know! Keep a “branch” for each number of possible executions.
  - This would become infeasible quickly...
  - Usually enforce a *loop unrolling limit*
  - “Explore at most 3 iterations of any loop”

What might be difficult to model symbolically?

```
BufferedReader reader = new BufferedReader(new FileReader("input.txt"));
String line = reader.readLine();


if (x > 0)
        If (foo(line) > 100)
else
        x = x + foo(line)
```

Parameterized Unit Tests (PUTs)

- Unit tests where the inputs are left as symbols
- Ideal setup for symbolic execution

```
public static String removeAllSlashes(String input) {
    if (input == null) return null;
    return input.replaceAll("[/\\\\]", "");
}

void testRemove(String input) {
    String output = removeAllSlahses(input);
    assertTrue(!output.contains("/"));
}
```

Summary

- Lab today: running a NASA symbolic execution tool - JavaPathFinder

- Symbolic execution
  - Reasons over *all inputs* to a program by tracking symbolic values and path conditions
  - Leverages SAT solvers
  - Requires less specs than FV

- Next class:
  - Concolic Execution
    - A mix of symbolic execution and concrete execution (Testing!)
    - Also called dynamic symbolic execution (DSE)
  - Techniques to deal with path explosion