## Lecture 12

# Symbolic Execution

Zvonimir Rakamarić
University of Utah

# Symbolic Testing

- Symbolic execution
- Concolic execution

# Past and Present of Symbolic Testing

- Introduced in 1976 by James King from IBM T.J. Watson Research Center
  - Implemented in EFFIGY – symbolic execution for a PL/I-like language
- Still very active area of research
  - SAGE, Pex [MSR]
  - KLEE [Stanford]
  - JDart [NASA, CMU, Utah]
  - BitScope [Berkeley]
  - CUTE [UIUC]
  - Calysto [UBC]
  - Saturn [Stanford]

# Program Paths

- Program path refers to a path in the control-flow graph of the program
- Program path is feasible if there exists an input to the program that “covers” the path
  - When the program is executed with this input, the path is taken
- Program path is infeasible if there exists no input that covers the path

# Infeasible Paths

- Infeasible path does not imply dead code
- Dead code implies infeasible path
- Example:

```
if (x > 0) {…}
else {…}
…
if (x > 10) {…}
else {…}
…
if (x < -10) {…}
else {…}
```

# Traditional Testing

- Real software has lots of infeasible paths
- Traditional testing does not scale when there is a large number of infeasible paths to the target location that needs to be covered

# Symbolic Execution

- Key idea: execution of programs using symbolic input values instead of concrete data
- Concrete vs symbolic
  - Concrete execution
    - Program takes only one path determined by input values
  - Symbolic execution
    - Program can (in theory) take any feasible path
    - Limited by the power of constraint solver
    - Scalability issues when faced with large (exponential) number of paths – path explosion

# Symbolic Program State

- Symbolic values of program variables
- Path condition (PC)
  - Logical formula over symbolic inputs
  - Accumulates constraints that inputs have to satisfy for the particular path to be executed
  - If a path is feasible its PC is satisfiable
- Program location

# Symbolic Execution Tree

- Characterizes execution paths constructed during symbolic execution
- Nodes are symbolic program states
- Edges are labeled with program transitions

# Example I

```
int x, y;
if (x > y) {
  x = x + y;
  y = x – y;
  x = x – y;
  if (x > y)
    assert false;
}
```

# Concrete Execution

- $x = 4$, $y = 3$

# Constructed Symbolic Execution Tree I

# Example II

```
int foo(int a, int b) {
  int k = a - b;
  int t = a + b + 3;
  if (a % 2 == 0) {
    a = b++;
    if (t > 0)
      k = t - 2;
  }
  if (a + 6 > k)
    b = 5;
  if (t + a + b == 20)
    assert false;
  return t + a + b;
}
```

# Constructed Symbolic Execution Tree II

# Path Explosion Problem I

```
int g1, g2;

int init(int x) {
  ... // Lots of paths
}

bool flip(int *data) {
  if (*data < 0) {
    *data = -(*data);
    return true;
  }
  return false;
}
```

```
void scale() {
  g2 = init(g1);
  if (flip(&g2)) {
    if (g2 == 0)
      assert false;
    g1 = g1/g2;
  }
}
```

# Solution: Structural Abstraction

- Key idea: abstract function calls by replacing them with uninterpreted functions
- Algorithm
  - Replace function calls with uninterpreted functions
  - If error is not reachable
    - Done
  - If error is reachable
    - Analyze error path
    - Perform on-demand abstraction refinement by replacing an uninterpreted function with the actual callee

# Path Explosion Problem I

```
int g1, g2;

int init(int x) {
  ... // Lots of paths
}

bool flip(int *data) {
  if (*data < 0) {
    *data = -(*data);
    return true;
  }
  return false;
}
```

```
void scale() {
  g2 = init(g1);
  if (flip(&g2)) {
    if (g2 == 0)
      assert false;
    g1 = g1/g2;
  }
}
```

# Path Explosion Problem II

```
int abs(int x) {
  if (x >= 0) return x;
  else return -x;
}

int sumAbs(int[] a) {
  int sum = 0;
  for (int i = 0; i < 50; i++)
    sum += abs(a[i]);
  if (sum == 13)
    assert false;
  return sum;
}
```

# Solution: Compositional Symb. Execution

- Key idea: compute function summaries to be used at all call sites of the function
  - Function summary encodes path conditions and return values of all paths through the function
  - Potential solution to path explosion problem
  - Only as good as computed function summaries
- Algorithm
  - Symbolically execute all paths of callee function and compute a function summary
  - When symbolically executing paths in the caller function, reuse the summary of the callee instead of repeatedly executing paths in the callee

# Path Explosion Problem II

```
int abs(int x) {
  if (x >= 0) return x;
  else return -x;
}

int sumAbs(int[] a) {
  int sum = 0;
  for (int i = 0; i < 50; i++)
    sum += abs(a[i]);
  if (sum == 13)
    assert false;
  return sum;
}
```

Summary of **abs**:

This is a stupid summary (causes branching in Z3 when unsat).

forall x. $(x \geq 0 \wedge abs(x)=x) \vee (x < 0 \wedge abs(x)=-x)$

This is a better summary:

forall x. (abs(x)>=0)

Path condition leading to error:

$abs(a[0]) + abs(a[1]) + \ldots + abs(a[49]) = 13 \wedge$ (forall x. $(x \geq 0 \wedge abs(x)=x) \vee (x < 0 \wedge abs(x)=-x))$

# Further Reading

- J.C. King: Symbolic Execution and Program Testing, CACM 1976
- D. Babic, A.J. Hu: Structural Abstraction of Software Verification Conditions, CAV 2007
- C. Pasareanu, W. Visser: A Survey of New Trends in Symbolic Execution for Software Testing and Analysis, STTT 2009
- N. Sinha, N. Singhania, S. Chandra, M. Sridharan: Alternate and Learn: Finding Witnesses without Looking All over, CAV 2012

# Next Time

- Concolic (concrete+symbolic) execution