Symbolic Execution

Emina Torlak
emina@cs.washington.edu
Today
Today

Last lecture

• Bounded verification: forward VCG for finitized programs
Today

Last lecture

• Bounded verification: forward VCG for finitized programs

Today

• Symbolic execution: a path-based translation
• Concolic testing
The spectrum of program validation tools

- Ad-hoc Testing
- Concolic Testing & Whitebox Fuzzing
- Bounded Verification & Symbolic Execution
- Extended Static Checking
- Static Analysis
- Verification

Confidence vs. Cost (programmer effort, time, expertise)
The spectrum of program validation tools

Cost (programmer effort, time, expertise)

Confidence

- Concolic Testing & Whitebox Fuzzing
- Bounded Verification & Symbolic Execution
- Extended Static Checking
- Static Analysis
- Verification

E.g., JPF, Klee

E.g., SAGE, Pex, CUTE, DART
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
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

```python
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

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.
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

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*.

```
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**

```python
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.
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*. 
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**.

```python
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.

*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**

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*.

```python
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

```python
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*. 

- *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: practical issues
Classic symbolic execution: practical issues

Loops and recursion: infinite execution trees
Classic symbolic execution: practical issues

Loops and recursion: infinite execution trees

Path explosion: exponentially many paths
Classic symbolic execution: practical issues

**Loops and recursion:** infinite execution trees

**Path explosion:** exponentially many paths

**Heap modeling:** symbolic data structures and pointers
Classic symbolic execution: practical issues

Loops and recursion: infinite execution trees

Path explosion: exponentially many paths

Heap modeling: symbolic data structures and pointers

Solver limitations: dealing with complex PCs
Classic symbolic execution: practical issues

Loops and recursion: infinite execution trees

Path explosion: exponentially many paths

Heap modeling: symbolic data structures and pointers

Solver limitations: dealing with complex PCs

Environment modeling: dealing with native / system / library calls
Classic symbolic execution: practical issues

Loops and recursion: infinite execution trees

Path explosion: exponentially many paths

Heap modeling: symbolic data structures and pointers

Solver limitations: dealing with complex PCs

Environment modeling: dealing with native / system / library calls
Loops and recursion

Dealing with infinite execution trees:

• Finitize paths by limiting the size of PCs (bounded verification)
• Use loop invariants (verification)
Loops and recursion

Dealing with infinite execution trees:

• Finitize paths by limiting the size of PCs (bounded verification)
• Use loop invariants (verification)

```
init;
while (C) {
    B;
}
assert P;
```
Loops and recursion

Dealing with infinite execution trees:

- Finitize paths by limiting the size of PCs (bounded verification)
- Use loop invariants (verification)

```java
init;
while (C) {
    B;
}
assert P;
```

```java
init;
assert I;
makeSymbolic(targets(B));
assume I;
if (C) {
    B;
    assert I;
} else
    assert P;
```
Path explosion

Achieving good coverage in the presence of exponentially many paths:

• Select next branch at random
• Select next branch based on coverage
• Interleave symbolic execution with random testing
Path explosion

Achieving good coverage in the presence of exponentially many paths:

- Select next branch at random
- Select next branch based on coverage
- Interleave symbolic execution with random testing

symbolic execution  random testing  interleaved execution
Heap modeling

Modeling symbolic heap values and pointers

• Segmented address space via the theory of arrays (Klee)
• Lazy concretization (JPF)
• Concolic lazy concretization (CUTE)
General lazy concretization

class Node {
    int elem;
    Node next;
}

n = symbolic(Node);
x = n.next;
General lazy concretization

class Node {
    int elem;
    Node next;
}

n = symbolic(Node);

x = n.next;
General lazy concretization

class Node {
    int elem;
    Node next;
}

n = symbolic(Node);
x = n.next;
General lazy concretization

class Node {
  int elem;
  Node next;
}
n = symbolic(Node);
x = n.next;
class Node {
    int elem;
    Node next;
}

n = symbolic(Node);
x = n.next;
Concolic testing

typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    abort();
        return 0;
}
Concolic testing

```c
typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    abort();
    return 0;
}
```

Concrete

<table>
<thead>
<tr>
<th>p</th>
<th>null</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>236</td>
</tr>
</tbody>
</table>

PC

\[ x > 0 \land p = \text{null} \]

Execute concretely and symbolically.
Concolic testing

typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    abort();
    return 0;
}
Concolic testing

typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0)
        if (p != NULL)
            if (f(x) == p->v)
                if (p->next == p)
                    abort();
    return 0;
}

Concrete

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>A0</td>
</tr>
<tr>
<td>next: null</td>
<td>next: null</td>
</tr>
<tr>
<td>v: 634</td>
<td>v: 3</td>
</tr>
</tbody>
</table>

PC

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x &gt; 0 ∧ p=null</td>
<td>x &gt; 0 ∧ p≠null ∧ p.v ≠ 2x + 1</td>
</tr>
<tr>
<td>x = 236</td>
<td>x = 1</td>
</tr>
</tbody>
</table>

Execute concretely and symbolically.
Negate last decision and solve for new inputs.
Concolic testing

```c
typedef struct cell {
    int v;
    struct cell *next;
} cell;

int f(int v) {
    return 2*v + 1;
}

int testme(cell *p, int x) {
    if (x > 0) {
        if (p != NULL) {
            if (f(x) == p->v) {
                if (p->next == p)
                    abort();
            }
        }
    }
    return 0;
}
```

<table>
<thead>
<tr>
<th>Concrete</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>p ↦ null</td>
<td>x &gt; 0 ∧ p=null</td>
</tr>
<tr>
<td>x ↦ 236</td>
<td></td>
</tr>
<tr>
<td>p ↦ A0</td>
<td>x &gt; 0 ∧ p≠null ∧ p.v ≠ 2x + 1</td>
</tr>
<tr>
<td>v: 634</td>
<td></td>
</tr>
<tr>
<td>x ↦ 236</td>
<td></td>
</tr>
<tr>
<td>p ↦ A0</td>
<td>x &gt; 0 ∧ p≠null ∧ p.v = 2x + 1 ∧ p.next ≠ p</td>
</tr>
<tr>
<td>v: 3</td>
<td></td>
</tr>
<tr>
<td>x ↦ 1</td>
<td></td>
</tr>
<tr>
<td>p ↦ A0</td>
<td>x &gt; 0 ∧ p≠null ∧ p.v = 2x + 1 ∧ p.next = p</td>
</tr>
<tr>
<td>v: 3</td>
<td></td>
</tr>
<tr>
<td>x ↦ 1</td>
<td></td>
</tr>
</tbody>
</table>

Execute concretely and symbolically.
Negate last decision and solve for new inputs.
Solver limitations

Reducing the demands on the solver:

- On-the-fly expression simplification
- Incremental solving
- Solution caching
- Substituting concrete values for symbolic in complex PCs (CUTE)
Environment modeling

Dealing with system / native / library calls:

- Partial state concretization
- Manual *models* of the environment (Klee)
Summary

Today

• Practical symbolic execution and concolic testing

Next lecture

• Basics of model checking