Symbolic Execution

Emina Torlak
ema@cs.washington.edu
Today

Last lecture

• Bounded verification: forward VCG for finitized programs

Today

• Symbolic execution: a path-based translation
• Concolic testing

Reminder

• Project progress reports due today at 11:00 pm.
The spectrum of program validation tools

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

**Cost (programmer effort, time, expertise)**

**Confidence**
The spectrum of program validation tools

Confidence

Cost (programmer effort, time, expertise)

- Static Analysis
- Verification
- Bounded Verification & Symbolic Execution
  - E.g., JPF, Klee
- Extended Static Checking
  - E.g., SAGE, Pex, CUTE, DART
- Concolic Testing & Whitebox Fuzzing
- Ad-hoc Testing
A brief history of 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
Symbolic execution: basic idea

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)
Symbolic execution: basic idea

```
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 execution: basic idea

Execute the program on *symbolic values*.

*Symbolic state* maps variables to symbolic values.

```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)
```

\[
x \mapsto A
\]
\[
y \mapsto B
\]
Symbolic execution: basic idea

Execute the program on \textit{symbolic values}. 

\textit{Symbolic state} maps variables to symbolic values.

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

\begin{verbatim}
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)
\end{verbatim}
Symbolic execution: basic idea

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)
```
Symbolic execution: basic idea

 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)
```
Symbolic execution: basic idea

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)
```
Symbolic execution: basic idea

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)
```
Symbolic execution: basic idea

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)
```
Symbolic execution: basic idea

```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 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 unrolling loops and recursion (bounded verification)
- 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 unrolling loops and recursion (bounded verification)
- Finitize paths by limiting the size of PCs (bounded verification)
- Use loop invariants (verification)

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

Dealing with infinite execution trees:

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

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

```
init;
assert I;
makeSymbolic(targets(B));
assume I;
if (C) {
    B;
    assert I;
} else
    assert P;
```
Loops and recursion

Dealing with infinite execution trees:

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

```c
init;
while (C) {
    B;
}
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
Heap modeling

Modeling symbolic heap values and pointers

- Bit-precise memory modeling with the theory of arrays (EXE, Klee, SAGE)
- Lazy concretization (JPF)
- Concolic lazy concretization (CUTE)
Heap modeling: lazy concretization

class Node {
    int elem;
    Node next;
}

n = symbolic(Node);
x = n.next;
Heap modeling: lazy concretization

class Node {
    int elem;
    Node next;
}

n = symbolic(Node);
x = n.next;
Heap modeling: lazy concretization

class Node {
  int elem;
  Node next;
}

n = symbolic(Node);

x = n.next;

\[
\begin{align*}
n & \mapsto A0 \\
x & \mapsto \text{null}
\end{align*}
\]

\[
\begin{align*}
A0.\text{next} & = \text{null} \\
n & \mapsto A0 \\
x & \mapsto \text{null}
\end{align*}
\]

\[
\begin{align*}
A0 & \\
\text{elem: ?} & \\
\text{next: null}
\end{align*}
\]
Heap modeling: lazy concretization

class Node {
  int elem;
  Node next;
}

n = symbolic(Node);

x = n.next;
```java
class Node {
    int elem;
    Node next;
}

n = symbolic(Node);

x = n.next;
```

Heap modeling: lazy concretization

```
class Node {
    int elem;
    Node next;
}

n = symbolic(Node);

x = n.next;
```
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;
}
Heap modeling: 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;
}

Execute concretely and symbolically.

Concrete

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

PC

\( x > 0 \land p=null \)
Heap modeling: 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;
}

Execute concretely and symbolically.
Negate last decision and solve for new inputs.
Heap modeling: 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;
}

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

Concrete       PC
p ↦ null    x > 0 ∧ p=null
x ↦ 236

Concrete       PC
p ↦ A0    x > 0 ∧ p≠null ∧
v: 634    p.v ≠ 2x + 1
x ↦ 236

Concrete       PC
p ↦ A0    x > 0 ∧ p≠null ∧
next: null    p.v = 2x + 1 ∧
v: 634    p.next ≠ p
x ↦ 3

Concrete       PC
p ↦ A0    x > 0 ∧ p≠null ∧
next: null    p.v = 2x + 1 ∧
v: 3    p.next ≠ p
x ↦ 1
Heap modeling: 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 | PC
---|---
p ↦ null | x > 0 ∧ p=null
v: 634 | x ↦ 236

A0
next: null
v: 634
x ↦ 236

A0
next: null
v: 634
x ↦ 236
p ≠ 2x + 1

A0
next: A0
v: 3
x ↦ 1

A0
next: A0
v: 3
x ↦ 1
p ≠ 2x + 1
p.next ≠ p

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