Symbolic Execution

Emina Torlak
cmina@cs.washington.edu
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

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

- **Static Analysis**
- **Verification**
- **Bounded Verification & Symbolic Execution**
- **Extended Static Checking**
- **Concolic Testing & Whitebox Fuzzing**
- **Ad-hoc Testing**
The spectrum of program validation tools

- **Concolic Testing & Whitebox Fuzzing**
  - E.g., JPF, Klee

- **Bounded Verification & Symbolic Execution**
  - E.g., SAGE, Pex, CUTE, DART

- **Ad-hoc Testing**

- **Verification**
  - Static Analysis

- **Extended Static Checking**
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

```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 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 state maps variables to symbolic values.
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.
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 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: 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.
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.*
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 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: 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

```
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)

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

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

• 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;
Heap modeling: lazy concretization

class Node {
    int elem;
    Node next;
}

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

A0

A0.next = null

A0.next = A0

A0

n ↦ A0
x ↦ null

A0

n ↦ A0
x ↦ A0

elem: ?
next: null

A0

elem: ?
next: A0
Heap modeling: lazy concretization

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

n = symbolic(Node);
x = n.next;
```

A0

A0

A0

A0

A0

A0

A0
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;
}
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.
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

<table>
<thead>
<tr>
<th>p</th>
<th>null</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>236</td>
</tr>
<tr>
<td>x &gt; 0 ∧ p=null</td>
<td></td>
</tr>
</tbody>
</table>

PC

<table>
<thead>
<tr>
<th>p</th>
<th>A0</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>236</td>
</tr>
<tr>
<td>x &gt; 0 ∧ p≠null ∧ p.v ≠ 2x + 1</td>
<td></td>
</tr>
</tbody>
</table>

next: null

v: 634

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.
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.
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

• Angelic execution