Symbolic and Concolic Testing

Nick Feng

Based on slides by Azadeh Farzan, Caroline Hu, Marsha Chechik and Michael Hicks
Symbolic Execution Summary

• A static analysis technique
• **Symbolic values** instead of concrete inputs.
• At each program location, the state is defined by:
  • **current assignments** to symbolic values and local variables.
  • **a path condition** that must hold for the execution to reach that location (conditions on the inputs to reach the location).

Slide Credit: Azadeh Farzan
Symbolic Execution Summary

• At each branch, both paths are followed.
  • On the true branch: the condition is added to the path constraints.
  • On the false branch: the negation of the condition is added.
• If the branch is infeasible, execution stops.
Function foo (int x, int y):
1   Read x, y
2   if x > 0:
3       y = 2 * x
4   else:
5       y = x
6   endif
7   if y ≥ 0:
8       y = y + 1
9   endif
10  if x * y > 0:
11      return x
12  else:
13      return y
Function foo (int x, int y):
1       Read x, y
2       if x > 0:
3           y = 2 * x
4       else:
5           y = x
6       endif
7       if y ≥ 0:
8           y = y + 1
9       endif
10      if x * y > 0:
11          return x
12      else:
13          return y

Example

• Input variables: x, y
  • Symbolic Names: X, Y
• All paths?
  • 1, 2, 3, 7, 8, 10, 11
  • 1, 2, 5, 7, 8, 10, 11
  • 1, 2, 3, 7, 10, 11
  • 1, 2, 5, 7, 10, 11
  • 1, 2, 3, 7, 8, 10, 13
  • 1, 2, 5, 7, 8, 10, 13
  • 1, 2, 3, 7, 10, 13
  • 1, 2, 5, 7, 10, 13
  • 1, 2, 3, 7, 10, 13
  • 1, 2, 5, 7, 10, 13
Function foo (int x, int y):
1       Read x, y
2       if x > 0:
3           y = 2 * x
4       else:
5           y = x
6       endif
7       if y ≥ 0:
8           y = y + 1
9       endif
10      if x * y > 0:
11         return x
12      else:
13         return y
14      endif
<table>
<thead>
<tr>
<th>line</th>
<th>Assignment</th>
<th>Path Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$x \leftarrow X, \ y \leftarrow Y$</td>
<td>True</td>
</tr>
<tr>
<td>2, 3</td>
<td>$y \leftarrow 2 \times X$</td>
<td>True AND $X &gt; 0$</td>
</tr>
<tr>
<td>7, 8</td>
<td>$y \leftarrow 2 \times X + 1$</td>
<td>True AND $X &gt; 0$ AND $2 \times X \geq 0$</td>
</tr>
<tr>
<td>10, 11</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>True AND $X &gt; 0$ AND $2 \times X \geq 0$ AND $X \times (2 \times X + 1) &gt; 0$</td>
<td></td>
</tr>
</tbody>
</table>

Function $\text{foo}(\text{int} \ x, \text{int} \ y)$:

1. Read $x, y$
2. if $x > 0$:
   3. $y = 2 \times x$
4. else:
   5. $y = x$
6. endif
7. if $y \geq 0$:
   8. $y = y + 1$
9. endif
10. if $x \times y > 0$:
    11. return $x$
12. else:
    13. return $y$

Solvable: $X = 1, Y = 1$, feasible path
Function foo (int x, int y):

1. Read x, y
2. if x > 0:
   3. y = 2 * x
   4. else:
      5. y = x
   6. endif
7. if y \geq 0:
   8. y = y + 1
   9. endif
10. if x * y > 0:
    11. return x
    12. else:
       13. return y
Path: 1, 2, 3, 7, 10, 13

<table>
<thead>
<tr>
<th>line</th>
<th>Assignment</th>
<th>Path Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$x \leftarrow X, \ y \leftarrow Y$</td>
<td>True</td>
</tr>
<tr>
<td>2, 3</td>
<td>$y \leftarrow 2 \times X$</td>
<td>True AND X &gt; 0</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>True AND X &gt; 0 AND $2 \times X &lt; 0$</td>
</tr>
<tr>
<td>10, 13</td>
<td></td>
<td>True AND X &gt; 0 AND $2 \times X &lt; 0$ AND $X \times (2 \times X) \leq 0$</td>
</tr>
</tbody>
</table>

No Solution! Infeasible path

Function foo (int x, int y):
1. Read x, y
2. if x > 0:
   3. $y = 2 \times x$
4. else:
   5. $y = x$
6. endif
7. if $y \geq 0$:
   8. $y = y + 1$
9. endif
10. if $x \times y > 0$:
   11. return x
12. else:
   13. return y
Exploring the Path Tree (DFS)

Function foo (int x, int y):
1   Read x, y
2   if x > 0:
3       y = 2 * x
4   else:
5       y = x
6   endif
7   if y ≥ 0:
8       y = y + 1
9   endif
10  if x * y > 0:
11     return x
12   else:
13     return y

Problem with Symbolic Execution

- Symbolic constraints can be very complex and cannot be solved by the constraint solver.
- The program being analyzed may have black box library functions.

Solution

Replace some of the symbolic values by concrete values available from the concrete state.

This is sound because concrete values are instantiations of symbolic values.
Problem with Symbolic Execution and Solution

• However, the approach may lose completeness.
• Nevertheless, this way of replacing some symbolic values by concrete values helps concolic testing scale for large programs for which symbolic testing would have otherwise failed.
Concolic Testing

\[ \text{CONCRETE EXECUTION (random testing)} + \text{SYMBOLIC EXECUTION (symbolic testing)} = \text{CONCOLIC EXECUTION} \]

```c
int foo (int v):
1    return (v*v) % 50

void testme (int x, int y):
1    z = foo (y)
2    if (z == x):
3        if (x > y + 10):
4            Error
```

Slide credit: Marsha Chechik
Concolic Testing

int foo (int v):
1         return (v*v) % 50

void testme (int x, int y):
1         z = foo (y)
2         if (z == x):
3             if (x > y + 10):
4                         Error

Concrete State: x = 22, y = 7
Symbolic State: x = X, y = Y
Path Condition: x = X, y = Y

Slide credit: Marsha Chechik
Concolic Testing

int foo (int v):
1     return (v*v) % 50

void testme (int x, int y):
1     z = foo (y)
2     if (z == x):
3         if (x > y + 10):
4             Error

Concrete Execution
Concrete State
x = 22, y = 7
x = 22, y = 7, z = 49

Symbolic Execution
Symbolic State
x = X, y = Y
x = X, y = Y, z = (Y * Y) % 50

Path Condition
Concolic Testing

int foo (int v):
1    return (v*v) % 50

void testme (int x, int y):
1    z = foo (y)
2    if (z == x):
3        if (x > y + 10):
4            Error

Concrete Execution
Concrete State
x = 22, y = 7
Symbolic State
x = X, y = Y
Path Condition
(Y * Y) % 50 != X

Symbolic Execution
x = X, y = Y,
% 50
z = (Y * Y)

Slide credit: Marsha Chechik
Concolic Testing

Concrete Execution

Symbolic Execution

Concrete State

Symbolic State

Path Condition

Solve: \((Y*Y) \mod 50 = X\)

Solution?

\((Y * Y) \mod 50 \neq X\)

---

```c
int foo (int v):
1    return (v*v) % 50

void testme (int x, int y):
1    z = foo (y)
2    if (z == x):
3        if (x > y + 10):
4            Error
```
Concolic Testing

• Deals with black box library functions
  • Replace symbolic values by concrete values
  • Ex.

```
int foo (int v):
1   return v*v % 50

void testme (int x, int y):
1   z = foo (y)
2   if (z == x):
3     if (x > y + 10):
4       Error

Solve for Y * Y % 50 = X
Solution: X = 49, Y = 7
```

• Deals with complex symbolic constraints: (Y*Y) % 50 = X
Concolic Testing

int foo (int v):
1   return (v*v) % 50

void testme (int x, int y):
1   z = foo (y)
2   if (z == x):
3     if (x > y + 10):
4       Error

Concrete Execution

Symbolic Execution

Concrete State

Symbolic State

Path Condition

Solve: (Y*Y) % 50 = X

Solution?

When the constraint is complex, use concrete state

Replace Y by 7

(Y * Y) % 50 != X
Concolic Testing

Concrete Execution

Symbolic Execution

Concrete State | Symbolic State | Path Condition

Solution: $X = 49, Y = 7$

int foo (int v):
1    return $(v*v) \mod 50$

void testme (int x, int y):
1    $z = \text{foo} (y)$
2    if $(z == x)$:
3        if $(x > y + 10)$:
4            Error

$(Y \times Y) \mod 50 \neq X$

Slide credit: Marsha Chechik
Concolic Testing

Concrete Execution

Symbolic Execution

Concrete State

Symbolic State

Path Condition

```
int foo (int v):
1  return (v*v) % 50

void testme (int x, int y):
1  z = foo (y)
2  if (z == x):
3     if (x > y + 10):
4         Error
```

x = 49, y = 7

x = X, y = Y

Slide credit: Marsha Chechik
Concolic Testing

<table>
<thead>
<tr>
<th>Concrete Execution</th>
<th>Symbolic Execution</th>
<th>Path Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concrete State</td>
<td>Symbolic State</td>
<td></td>
</tr>
<tr>
<td>$x = 49, y = 7$</td>
<td>$x = X, y = Y$</td>
<td>$Z \equiv X$</td>
</tr>
<tr>
<td>$x = 49, y = 7, z = 49$</td>
<td>$x = X, y = Y, z = 49$</td>
<td>$X &gt; Y + 10$</td>
</tr>
</tbody>
</table>

### int foo (int v):
1. return $(v*v) \% 50$

### void testme (int x, int y):
1. $z = \text{foo}(y)$
2. if ($z == x$):
3. \hspace{1em} if ($x > y + 10$):
4. \hspace{2em} Error
Concolic Testing

int foo (int v):
1. return (v*v) % 50

void testme (int x, int y):
1. z = foo (y)
2. if (z == x):
3. if (x > y + 10):
4. Error

Program Error
Concolic Testing

Concrete Execution

<table>
<thead>
<tr>
<th>Concrete State</th>
<th>Symbolic Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = 49, y = 7 )</td>
<td>( x = X, y = Y )</td>
</tr>
<tr>
<td>( x = 49, y = 7, z = 49 )</td>
<td>( x = X, y = Y, z = 49 )</td>
</tr>
</tbody>
</table>

Path Condition

\( Z = X \)

\( Z \geq 10 \)

### Code Snippet

```c
int foo (int v):
1 return (v*v) % 50

void testme (int x, int y):
1 z = foo (y)
2 if (z == x):
3 if (z < 10):
4 Error
```
Concolic Testing

int foo (int v):
1  return (v*v) % 50

void testme (int x, int y):
1  z = foo (y)
2  if (z == x):
3       if (z < 10):
4       Error

Concrete Execution

Concrete State
x = 49, y = 7
x = 49, y = 7, z = 49

Symbolic Execution
Symbolic State
x = X, y = Y
x = X, y = Y, z = 49

Path Condition
Z == X
Z ≥ 10

Solve: 49 == X ,
49 < 10

No Solution!

Slide credit: Marsha Chechik
Concolic Testing

Concrete Execution

Symbolic Execution

Concrete State

Symbolic State

Path Condition

int foo (int v):
1  return (v*v) % 50

void testme (int x, int y):
1  z = foo (y)
2  if (z == x):
3    if (z < 10):
4      Error

The Path is feasible with X= 1, Y =1

Concretization may lose Completeness

Slide credit: Marsha Chechik
Concolic Testing Example

Concrete Input: X = 0, Y = 0

Explored path:

```c
Function foo (int x, int y):
1   Read x, y
2   if x > 0:
3       y = 2 * x
4   else:
5       y = x
6   endif
7   if y ≥ 0:
8       y = y + 1
9   endif
10  if x * y > 0:
11     return x
12   else:
13     return y
```
Concolic Testing Example

Concrete Input:  X = 0, Y = 0
Execution Path:  FTF

Unexplored Path’s PC:
1. X > 0
2. X ≤ 0 ∧ X < 0
3. X ≤ 0 ∧ X ≥ 0 ∧ X * (X + 1) > 0

Now, pick one path to explore!

Function foo (int x, int y):
1   Read x, y
2   if x > 0:
3     y = 2 * x
4   else:
5     y = x
6   endif
7   if y ≥ 0:
8     y = y + 1
9   endif
10  if x * y > 0:
11     return x
12   else:
13     return y
Concolic Testing Example

Concrete Input: X = 0, Y = 0
Execution Path: FTF

Unexplored Path’s PC:
1. X > 0
2. X ≤ 0 ∧ X < 0
3. X ≤ 0 ∧ X ≥ 0 ∧ X * (X + 1) > 0

Pick the first PC, solving yields:
X = 1, Y = 0

Function foo (int x, int y):
1       Read x, y
2       if x > 0:
3           y = 2 * x
4       else:
5           y = x
6       endif
7           if y ≥ 0:
8               y = y + 1
9           endif
10          if x * y > 0:
11              return x
12          else:
13              return y

Exploded path: FTF
Concolic Testing Example

Concrete Input: $X = 1$, $Y = 0$
Execution Path: TTT

Unexplored Path's PC:
1. $X \leq 0 \land X < 0$
2. $X \leq 0 \land X \geq 0 \land X \times (X + 1) > 0$
3. $X > 0 \land 2X < 0$
4. $X > 0 \land 2X \geq 0 \land (2X + 1) \leq 0$

Pick the first PC, solving yields:
$X = -1$, $Y = 0$
Concolic Testing Example

Concrete Input: $X = -1$, $Y = 0$
Execution Path: FFT

Unexplored Path’s PC:
1. $X \leq 0 \land X \geq 0 \land X \ast (X + 1) > 0$
2. $X > 0 \land 2X < 0$
3. $X > 0 \land 2X \geq 0 \land (2X + 1) \leq 0$
4. $X \leq 0 \land X < 0 \land X \ast X \leq 0$

Pick the first PC, but it has no solution, remove it

Function foo (int x, int y):
1       Read x, y
2       if x > 0:
3           y = 2 * x
4       else:
5           y = x
6       endif
7       if y \geq 0:
8           y = y + 1
9       endif
10      if x \ast y > 0:
11          return x
12      else:
13          return y
Concolic Testing Example

Concrete Input: $X = -1$, $Y = 0$
Execution Path: FFT

Unexplored Path's PC:
1. $X > 0 \land 2X < 0$
2. $X > 0 \land 2X \geq 0 \land (2X + 1) \leq 0$
3. $X \leq 0 \land X < 0 \land X \times X \leq 0$

Pick the first PC, but it has no solution, remove it

Function foo (int $x$, int $y$):
1. Read $x$, $y$
2. \textbf{if} $x > 0$:
3. \hspace{1em} $y = 2 \times x$
4. \textbf{else}:
5. \hspace{1em} $y = x$
6. \textbf{endif}
7. \textbf{if} $y \geq 0$:
8. \hspace{1em} $y = y + 1$
9. \textbf{endif}
10. \textbf{if} $x \times y > 0$:
11. \hspace{1em} return $x$
12. \textbf{else}:
13. \hspace{1em} return $y$
Function foo (int x, int y):

1. Read x, y
2. if x > 0:
   3. y = 2 * x
3. else:
   4. y = x
5. endif
6. if y \geq 0:
   7. y = y + 1
6. endif
7. if x * y > 0:
   10. return x
8. return y

Concrete Input: X = -1, Y = 0
Execution Path: FFT

Unexplored Path's PC:
1. $X > 0 \land 2X \geq 0 \land (2X + 1) \leq 0$
2. $X \leq 0 \land X < 0 \land X \times X \leq 0$

Pick the first PC, but it has no solution, remove it
Concolic Testing Example

Concrete Input: X = -1, Y = 0
Execution Path: FFT

Unexplored Path's PC:
1. $X \leq 0 \land X < 0 \land X \times X \leq 0$

Pick the first PC, but it has no solution, remove it

Function foo (int x, int y):
1. Read x, y
2. if x > 0:
3.     y = 2 * x
4.   else:
5.     y = x
6. endif
7. if y ≥ 0:
8.     y = y + 1
9. endif
10. if x * y > 0:
11.     return x
12.   else:
13.     return y
Concolic Testing Example

Concrete Input: X = -1, Y = 0
Execution Path: FFT

Unexplored Path’s PC:

Function foo (int x, int y):
1       Read x, y
2         if x > 0:
3            y = 2 * x
4         else:
5            y = x
6         endif
7         if y ≥ 0:
8            y = y + 1
9         endif
10        if x * y > 0:
11            return x
12        else:
13            return y

No more unexplored path, we are done!
The Path Explosion Problem

- The program under test may have too many paths
  - Number of paths is exponential in branching structure
  - Infinitely many paths for programs with unbound loop
- We want to find as many bugs as possible with limited resources

```c
int main() {
    int p, n, b;
    p = 42;
    if (b == 1 && n < 500 ){
        Possible Error
    }
    while ( n>0 ) {
        assert p != 0;
        if (n == 0) {
            p = 0;
        }
        n--;
    }
    return 0;
}
```
Search Strategy

• Basic search: BFS, DFS
• Random search
• Coverage guided heuristic
• Generation search
• Fuzzing + Symbolic Execution
Basic Search: BFS, DFS

- DFS: Depth-First Search:
  - Pick an unexplored direction from the last encountered branch point
  - The search order we followed for concolic testing example! (Slides 27-34)

- BFS: Breath-First Search:
  - Pick an unexplored direction from the first encountered branch point
  - The search order we followed for exploring the path Tree (Slides 10)

- Neither is guided by program knowledge
- DFS can get stuck on a part of the program

```c
int main() {
    int p, n, b;
    p = 42;
    if (b == 1 & n == 500) {
        Possible Error
    }
    while (n>=0) {
        assert p != 0;
        if (n == 0) {
            p = 0;
        }
        n--;
    }
    return 0;
}
```

If we start with b=0, DFS could stuck in exploring the loop

We can do better if we know where the possible error locations are

Slide credit: Michael Hicks
Random Search

- If we don’t know a priori which paths to take, then adding randomness seems like a good idea:
  - Idea 1: pick the next path to explore uniformly random
  - Idea 2: randomly restart search if haven’t anything interesting
  - Idea 3: when have equal priority paths to explore, choose the next one randomly

- Drawback: *reproducibility*
  - Use pseudo-random with fixed seed instead
Coverage guided heuristic

• Idea: Try to visit statement we haven’t seen before

• Approach:
  • Score of statement: statement visited count and frequency
  • Pick next statement with lowest score (more likely to discover new behavior)

• Why might this work?
  • Error are often in hard-to-reach parts of the program
  • This strategy tries to reach everywhere

• Why might this not work?
  • How do we get to a statement if proper precondition is not set up

Slide credit: Michael Hicks
Coverage guided heuristic

```c
int main() {
    int p, n, b;
    p = 42;
    if (b == 1 & n == 500) {
        Possible Error
    }
    while (n >= 0) {
        assert p != 0;
        if (n == 0) {
            p = 0;
        }
        n--;
    }
    return 0;
}
```

If we start with b=0, and stuck on exploring the loop

High score for statements inside the loop

Low score for Possible error

Prioritize the path with PC: b ==1 & n==500
Generational search

• Hybrid of BFS and coverage-guided
• Generation 0: a complete run of the program
• Generation 1: paths produced by negating one of the branch conditions from the PC of generation 0.
• Generation n: similar, but branching off from gen n-1
• Use a coverage heuristic (maximize block coverage) to pick priority

• The choice of Generation 0 is important! This could be an effective strategy with some prior knowledge about the program


Slide credit: Michael Hicks
Generational search

Gen 0:
TTT

\[ x > 0, \quad 2x \geq 0 \]
Generational search

Gen 0:
TTT
Gen 1:
TF
TTF
FTF

\[ X > 0 \]
\[ 2X \leq 0 \]

\[ X \leq 0 \]
\[ 2X \geq 0 \]

\[ X * (2X+1) < 0 \]
Generational search

Gen 0:
TTT
Gen 1:
TF
TTF
FTF

Gen 2
FTT
FFT

\[ X > 0, \quad 2X \geq 0 \]
\[ X > 0, \quad 2X < 0 \]
\[ X \leq 0, \quad X \geq 0 \]
\[ X \leq 0, \quad X < 0 \]
\[ X \leq 0, \quad X \geq 0 \]
\[ X \leq 0, \quad X < 0 \]
\[ X \leq 0, \quad X \geq 0 \]
\[ X \leq 0, \quad X < 0 \]
\[ X \leq 0, \quad X \geq 0 \]
\[ X \leq 0, \quad X < 0 \]
\[ X \leq 0, \quad X \geq 0 \]
Generational search

Gen 0:
TTT

Gen 1:
TF
TTF
FTF

Gen 2
FTT
FFT

Gen 3
FFF

\[
X > 0, \
2X \geq 0 \\
X > 0, \
2X < 0
\]

\[
X \leq 0, \
X \geq 0 \\
X \leq 0, \
X < 0
\]

\[
X \leq 0, \
X \geq 0 \\
X \leq 0, \
X < 0
\]

\[
X \leq 0, \
X \geq 0 \\
X \leq 0, \
X < 0
\]

\[
X \leq 0, \
X \geq 0 \\
X \leq 0, \
X < 0
\]

\[
X \leq 0, \
X \geq 0 \\
X \leq 0, \
X < 0
\]
Random Testing (Fuzzing) + Symbolic Execution

• Fuzzing is simple, cheap and effective in achieving code coverage and finding shallow bugs.
• But cannot effectively explore paths and detect path-specific deep bugs

• Symbolic execution is effective at explore path specific deep bugs
• But expensive

• Combine both:
  • Use symbolic execution to discover preconditioned inputs as new seed for fuzzing
  • alternate between fuzzing and symbolic execution. Switch to symbolic execution to explore unexplored paths when fuzzing becomes less effective.

Pak, Brian S. "Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution." School of Computer Science Carnegie Mellon University (2012).