AAA615: Formal Methods

Lecture 9 — Symbolic Execution

Hakjoo Oh
2017 Fall
Symbolic Execution

- A program analysis technique that executes a program with symbolic – rather than concrete – input values.
- Popular for finding software bugs and vulnerabilities: e.g.,
  - In Microsoft, 30% of bugs are discovered by symbolic execution.
  - Symbolic execution is the key technique used in DARPA Cyber Grand Challenge.
- Symbolic execution tools:
  - Stanford: KLEE
  - NASA: PathFinder
  - Microsoft: SAGE
  - UC Berkeley: CUTE
  - EPFL: $S^2E$
- Slides are based on the paper:
Example

1. void foobar(int a, int b) {
2.    int x = 1, y = 0;
3.    if (a != 0) {
4.        y = 3+x;
5.        if (b == 0)
6.            x = 2*(a+b);
7.    }
8.    assert(x-y != 0);
9. }

The goal is to find the inputs that make the assertion fail.

- Random testing with concrete values unlikely generate the inputs.
- Symbolic execution overcomes the limitation of random testing by reasoning on classes of inputs, rather than single input values.
Symbolic Execution

- Program inputs are represented by symbols: $\alpha_a, \alpha_b$.
- Symbolic execution maintains a state $(stmt, \sigma, \pi)$:
  - $stmt$: the next statement to evaluate
  - $\sigma$: symbolic store
  - $\pi$: path constraints
- Depending on $stmt$, symbolic execution proceeds as follows:
  - $x = e$: It updates the symbolic store $\sigma$ by associating $x$ with a new symbolic expression $e_s$, where $e_s$ is a symbolic expression obtained by evaluating $e$ symbolically.
  - if $e$ then $s_1$ else $s_2$: It is forked by creating two states with path constraints $\pi \land e_s$ and $\pi \land \neg e_s$.
  - assert($e$): The validity of $e$ is checked.
    - If $\neg e \land \pi$ is unsatisfiable, the assertion is always true.
    - If $\neg e \land \pi$ is satisfiable, an assert-fail input is found.
Symbolic Execution Tree

A \[ \sigma = \{ a \mapsto \alpha_a, b \mapsto \alpha_b \} \quad \pi = \text{true} \]
2. int x = 1, y = 0

B \[ \sigma = \{ a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 0 \} \quad \pi = \text{true} \]
3. if (a != 0)

C \[ \sigma = \{ a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 0 \} \quad \pi = \alpha_a \neq 0 \]
4. y = 3 + x

D \[ \sigma = \{ a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 0 \} \quad \pi = \alpha_a = 0 \]
8. assert(x-y != 0)

E \[ \sigma = \{ a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 4 \} \quad \pi = \alpha_a \neq 0 \]
5. if (b == 0)

F \[ \sigma = \{ a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 4 \} \quad \pi = \alpha_a \neq 0 \land \alpha_b = 0 \]
6. x = 2*(a+b)

G \[ \sigma = \{ a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 4 \} \quad \pi = \alpha_a \neq 0 \land \alpha_b \neq 0 \]
8. assert(x-y != 0)

H \[ \sigma = \{ a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 2(\alpha_a + \alpha_b), y \mapsto 4 \} \quad \pi = \alpha_a \neq 0 \land \alpha_b = 0 \]
8. assert(x-y != 0)

\[ 2(\alpha_a + \alpha_b) - 4 = 0 \land \alpha_a \neq 0 \land \alpha_b = 0 \quad \text{if} \quad \alpha_a = 2 \land \alpha_b = 0 \quad \text{ERROR} \]
Challenges

Symbolic execution for real-world software is challenging:

- Pointers and arrays.
- Loops
- Constraint solving.
- Open programs (e.g. programs with external calls).
- Path explosion.
Handling of Pointers and Arrays

Classical approaches maintain fully symbolic memory addresses with state forking or if-then-else formulas. For example, consider the code:

```c
1. void foobar(unsigned i, unsigned j) {
2.     int a[2] = { 0 };
3.     if (i>1 || j>1) return;
4.     a[i] = 5;
5.     assert(a[j] != 5);
6. }
```
State Forking

If an operation reads from or writes to a symbolic address, the state is forked by considering all possible states that may result from the operation. The path constraints are updated accordingly for each forked state.
**If-then-else Formulas**

An alternative is to encode the possibilities in the symbolic store with if-then-else, without forking states.

\[ \alpha_i \leq 1 \land \alpha_j \leq 1 \]

\[ \sigma = \{ a[0] \mapsto 0, a[1] \mapsto 0, i \mapsto \alpha_i, j \mapsto \alpha_j \} \quad \pi = true \]

\[ \text{if} \ (i > 1 \ \text{||} \ j > 1) \]

\[ \alpha_i > 1 \lor \alpha_j > 1 \]

\[ \sigma = \{ a[0] \mapsto 0, a[1] \mapsto 0, i \mapsto \alpha_i, j \mapsto \alpha_j \} \quad \pi = \alpha_i > 1 \lor \alpha_j > 1 \]

\[ a[i] = 5; \]

\[ \sigma = \{ a[0] \mapsto ite(\alpha_i = 0, 5, 0), a[1] \mapsto ite(\alpha_i = 1, 5, 0), i \mapsto \alpha_i, j \mapsto \alpha_j \} \quad \pi = \alpha_i \leq 1 \land \alpha_j \leq 1 \]

\[ \text{assert}(a[j] \neq 5); \]

\[ ite(\alpha_j = 0, ite(\alpha_i = 0, 5, 0), ite(\alpha_i = 1, 5, 0)) = 5 \land \alpha_i \leq 1 \land \alpha_j \leq 1 \]

\[ \text{if} \ (\alpha_i = 0 \land \alpha_j = 0) \lor (\alpha_i = 1 \land \alpha_j = 1) \ \text{ERROR} \]
Other Approaches

Other approaches for scalability:

- Address concretization
- Partial memory modeling
- Lazy initialization
Handling Loops

Consider the program, where we do not know the loop bound:

```c
void f (unsigned int n) {
    i = 0;
    while (i < n) {
        i = i + 1;
    }
}
```

Symbolic execution would keep forking and running forever.
Handling Loops

A common solution in practice is to unroll the loop for a fixed bound, e.g., $k = 2$:

```c
void f (unsigned int n) {
    i = 0;
    if (i < n) {
        i = i + 1;
    }
    if (i < n) {
        i = i + 1;
    }
}
```

The resulting analysis compromises soundness.
Handling Loops

Another solution is to provide a loop invariant and let symbolic execution use it to skip the analysis of the loop:

```c
void f (unsigned int n) {
    i = 0;
    while (i < n) { // inv: i <= n
        i = i + 1;
    }
}
```

The resulting analysis is either semi-automatic or over-approximated.
Constraint Solving

A key component of symbolic execution is a constraint solver. Two problems:

- Invoking an SMT solver is expensive.
  - Symbolic execution maintains a mapping from formulas to satisfying assignments: e.g.,
    
    \[ x + y < 10 \land x > 5 \mapsto \{ x = 6, y = 3 \} \]
  
  - When we query a weaker formula, e.g., \( x + y < 10 \), we can reuse the previously computed solution, without invoking an SMT solver.
  
  - When the formula is stronger, e.g., \( x + y < 10 \land x > 5 \land y \geq 0 \), then we first try the solution in the cache. If it does not work, call the SMT solver.

- Constraints from real-world software are hard to solve.
  
  - E.g., non-linear constraints
Open Programs

How to handle unknown external calls?

- Environment modeling
- Execution with concrete values
Path Explosion

Because symbolic execution forks off a new state at every branch of the program, the total number of states easily becomes exponential in the number of branches. Techniques for addressing path explosion:

- Pruning unrealizable paths
- State merging
- Path selection
- Function and loop summarization
- Path subsumption and equivalence
Pruning Unrealizable Paths

We can reduce the state space by invoking an SMT solver to detect unrealizable paths. For example,

```plaintext
if (a > 0) { ... }
if (a > 1) { ... }
```

- Eager evaluation calls an SMT solver at each branch.
- Lazy evaluation does not to reduce the burden on the solver.
State Merging

State merging is a technique that merges different paths into a single state. For example,

1. void foo(int x, int y) {
2.     if (x < 5)
3.         y = y * 2;
4.     else
5.         y = y * 3;
6.     return y;
7. }

without state merging

with state merging
State Merging

- Given two states \((stmt, \sigma_1, \pi_1)\) and \((stmt, \sigma_2, \pi_2)\), the merged state is

\[
(stmt, \sigma', \pi_1 \vee \pi_2)
\]

where \(\sigma'\) merges \(\sigma_1\) and \(\sigma_2\) with \(ite\) expressions.

- State merging has trade-offs: merging decreases the number of paths to explore but also put a burden on constraints solvers.

- State merging heuristics:
  - See Query cost estimation, Veritesting, etc
  - See also (Efficient State Merging in Symbolic Execution. PLDI 2012)
Path Selection Heuristics

Since enumerating all paths of a program can be prohibitively expensive, symbolic execution prioritizes the most promising paths. Several strategies for selecting the next path to be explored have been proposed: e.g.,

- Depth-first search
- Breadth-first search
- Random path selection
- Coverage optimize search
- Subpath-guided search
- Buggy-path first search
- . . .
Concolic Execution

An approach that combines concrete and symbolic execution to address the limitations of symbolic execution.

- external calls
- constraint solving
- pointers

Approaches to concolic execution:

- Dynamic symbolic execution (e.g. DART, SAGE, KLEE)
- Selective symbolic execution (e.g. S²E)
Dynamic Symbolic Execution

One popular concolic execution approach, where concrete execution drives symbolic execution. Consider the code:

```c
int double (int v) {
    return 2*v;
}

void testme(int x, int y) {
    z := double (y);
    if (z==x) {
        if (x>y+10) {
            Error;
        }
    }
}
```
Dynamic Symbolic Execution

```c
int double (int v) {
    return 2*v;
}

void testme(int x, int y) {
    z := double (y);
    if (z==x) {
        if (x>y+10) {
            Error;
        }
    }
}
```

<table>
<thead>
<tr>
<th>Concrete State</th>
<th>Symbolic State</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=22, y=7</td>
<td>x=α, y=β true</td>
</tr>
</tbody>
</table>
Dynamic Symbolic Execution

```c
int double (int v) {
    return 2*v;
}

void testme(int x, int y) {
    z := double (y);
    if (z==x) {
        if (x>y+10) {
            Error;
        }
    }
}
```

1st iteration

Concrete State: $x=22, y=7, z=14$

Symbolic State: $x=a, y=\beta, z=2^\beta$, $\text{true}$
Dynamic Symbolic Execution

```c
int double (int v) {
    return 2*v;
}

void testme(int x, int y) {
    z := double (y);
    if (z==x) {
        if (x>y+10) {
            Error;
        }
    }
}
```

1st iteration

**Concrete State**
- `x=22, y=7, z=14`

**Symbolic State**
- `x=a, y=β, z=2*β`
- `2*β ≠ a`
Dynamic Symbolic Execution

```c
int double (int v) {
    return 2*v;
}

void testme(int x, int y) {
    z := double (y);
    if (z==x) {
        if (x>y+10) {
            Error;
        }
    }
}
```

Concrete State

Symbolic State

Solve: $2\beta = \alpha$
Solution: $\alpha=2, \beta=1$

1st iteration

- $x=22, y=7, z=14$
- $x=\alpha, y=\beta, z=2\beta$
- $2\beta \neq \alpha$
Dynamic Symbolic Execution

```c
int double (int v) {
    return 2*v;
}

void testme(int x, int y) {
    z := double (y);
    if (z==x) {
        if (x>y+10) {
            Error;
        }
    }
}
```

Concrete State:
- `x=2, y=1`

Symbolic State:
- `x=α, y=β`
  - true

2nd iteration
Dynamic Symbolic Execution

```c
int double (int v) {
    return 2*v;
}

void testme(int x, int y) {
    z := double (y);
    if (z==x) {
        if (x>y+10) {
            Error;
        }
    }
}
```

Concrete State

Symbolic State

2nd iteration
Dynamic Symbolic Execution

```c
int double (int v) {
    return 2*v;
}

void testme(int x, int y) {
    z := double (y);
    if (z==x) {
        if (x>y+10) {
            Error;
        }
    }
}
```

<table>
<thead>
<tr>
<th>Concrete State</th>
<th>Symbolic State</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=2, y=1, z=2</td>
<td>x=α, y=β, z=2*β</td>
</tr>
<tr>
<td></td>
<td>2*β = α</td>
</tr>
</tbody>
</table>

2nd iteration
```c
int double (int v) {
    return 2*v;
}

void testme(int x, int y) {
    z := double (y);
    if (z==x) {
        if (x>y+10) {
            Error;
        }
    }
}
```

**Concrete State**
- `x=2, y=1, z=2`
- `x=\alpha, y=\beta, z=2*\beta`
- `2*\beta = \alpha \land \alpha \leq \beta + 10`

**Symbolic State**

2nd iteration
Dynamic Symbolic Execution

```c
int double (int v) {
    return 2*v;
}

void testme(int x, int y) {
    z := double (y);
    if (z==x) {
        if (x>y+10) {
            Error;
        }
    }
}
```

<table>
<thead>
<tr>
<th>Concrete State</th>
<th>Symbolic State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solve: $2\beta = \alpha \land \alpha &gt; \beta+10$</td>
<td>$x=\alpha, y=\beta, z=2^\beta$</td>
</tr>
<tr>
<td>Solution: $\alpha=30, \beta=15$</td>
<td>$2^\beta = \alpha \land \alpha \leq \beta+10$</td>
</tr>
</tbody>
</table>

2nd iteration
int double (int v) {
    return 2*v;
}

void testme(int x, int y) {
    z := double (y);
    if (z==x) {
        if (x>y+10) {
            Error;
        }
    }
}

Concrete State
x=30, y=15

Symbolic State
x=α, y=β
true

3rd iteration
Dynamic Symbolic Execution

```c
int double (int v) {
    return 2*v;
}

void testme(int x, int y) {
    z := double (y);
    if (z==x) {
        if (x>y+10) {
            Error;
        }
    }
}
```

Concrete State
- x=30, y=15,
- z=30

Symbolic State
- x=a, y=β, z=2*β
- true

3rd iteration
Dynamic Symbolic Execution

```c
int double (int v) {
    return 2*v;
}

void testme(int x, int y) {
    z := double (y);
    if (z==x) {
        if (x>y+10) {
            Error;
        }
    }
}
```

3rd iteration

<table>
<thead>
<tr>
<th>Concrete State</th>
<th>Symbolic State</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=30, y=15, z=30</td>
<td>x=α, y=β, z=2*β</td>
</tr>
<tr>
<td></td>
<td>2*β = α</td>
</tr>
</tbody>
</table>
int double (int v) {
    return 2*v;
}

void testme(int x, int y) {
    z := double (y);
    if (z==x) {
        if (x>y+10) {
            Error;
        }
    }
}

Concrete State
x=30, y=15, z=30

Symbolic State
x=α, y=β, z=2*β
2*β = α

3rd iteration
int double (int v) {
    return 2*v;
}

void testme(int x, int y) {
    z := double (y);
    if (z==x) {
        if (x>y+10) {
            Error;
        }
    }
}

Concrete State
x=30, y=15, z=30

Symbolic State
x=α, y=β, z=2*β
2*β = α ∧
α > β+10

3rd iteration
Dynamic Symbolic Execution

Consider the program with non-linear expression:

```cpp
int double (int v) {
    return v*v;
}

void testme(int x, int y) {
    z := double (y);
    if (z==x) {
        if (x>y+10) {
            Error;
        }
    }
}
```
Dynamic Symbolic Execution

```c
int double (int v) {
    return v*v;
}

void testme(int x, int y) {
    z := double (y);
    if (z==x) {
        if (x>y+10) {
            Error;
        }
    }
}
```

<table>
<thead>
<tr>
<th>Concrete State</th>
<th>Symbolic State</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=22, y=7</td>
<td>x=α, y=β</td>
</tr>
<tr>
<td></td>
<td>true</td>
</tr>
</tbody>
</table>

1st iteration
```c
int double (int v) {
    return v*v;
}

void testme(int x, int y) {
    z := double (y);
    if (z==x) {
        if (x>y+10) {
            Error;
        }
    }
}
```

**Concrete State**
- $x=22$, $y=7$, $z=49$

**Symbolic State**
- $x=\alpha$, $y=\beta$, $z=\beta^*\beta$
- True
Dynamic Symbolic Execution

```c
int double (int v) {
    return v*v;
}

void testme(int x, int y) {
    z := double (y);
    if (z==x) {
        if (x>y+10) {
            Error;
        }
    }
}
```

Concrete State

- x=22, y=7, z=49
- x=α, y=β, z=β*β
- β*β ≠ α

Symbolic State

1st iteration
Dynamic Symbolic Execution

```c
int double (int v) {
    return v*v;
}

void testme(int x, int y) {
    z := double (y);
    if (z==x) {
        if (x>y+10) {
            Error;
        }
    }
}
```

Concrete State

- Solve: $\beta^2 = \alpha \land \beta = 7$
- Solution: $\alpha=49, \beta=7$

Symbolic State

- $x=22, y=7, z=49$
- $x=\alpha, y=\beta, z=\beta^2$
- $\beta^2 \neq \alpha$

1st iteration
Dynamic Symbolic Execution

```c
int double (int v) {
    return v*v;
}

void testme(int x, int y) {
    z := double (y);
    if (z==x) {
        if (x>y+10) {
            Error;
        }
    }
}

2nd iteration
```

Concrete State

- x=49, y=7

Symbolic State

- x=α, y=β
- true
Dynamic Symbolic Execution

```c
int double (int v) {
    return v*v;
}

void testme(int x, int y) {
    z := double (y);
    if (z==x) {
        if (x>y+10) {
            Error;
        }
    }
}
```

Concrete State:
- $x=49$, $y=7$, $z=49$

Symbolic State:
- $x=a$, $y=b$, $z=b^2 + b$
  - true

2nd iteration
Dynamic Symbolic Execution

```c
int double (int v) {
    return v*v;
}

void testme(int x, int y) {
    z := double (y);
    if (z==x) {
        if (x>y+10) {
            Error;
        }
    }
}
```

<table>
<thead>
<tr>
<th>Concrete State</th>
<th>Symbolic State</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=49, y=7, z=49</td>
<td>$x=\alpha, y=\beta, z=\beta \cdot \beta$</td>
</tr>
<tr>
<td>$\beta \cdot \beta = \alpha$</td>
<td></td>
</tr>
</tbody>
</table>

2nd iteration
Dynamic Symbolic Execution

```c
int double (int v) {
    return v*v;
}

void testme(int x, int y) {
    z := double (y);
    if (z==x) {
        if (x>y+10) {
            Error;
        }
    }
}
```

Concrete State

Symbolic State

- $x=49, y=7, z=49$
- $x=\alpha, y=\beta, z=\beta\beta$
- $\beta\beta = \alpha \land \alpha > \beta+10$

2nd iteration
Trade-off

By replacing symbolic values by concrete values, the analysis cannot generate the inputs that exercise the false branch of $x > y + 10$. 
Handling of External Calls

External calls are executed with concrete values:

```c
void foo(int x, int y) {
    int a = bar(x);
    if (y < 0) ERROR;
}
```

- Assume that \( x = 1 \) and \( y = 2 \) are initial input parameters.
- The concolic engine executes \( \text{bar} \) (which returns \( a = 0 \)) and skips the branch that would trigger the error statement.
- At the same time, the symbolic execution tracks the path constraint \( \alpha_y \geq 0 \) inside function \( \text{foo} \).
- Notice that branch conditions in function \( \text{bar} \) are not known to the engine.
- To explore the alternative path, the engine negates the path constraint of the branch in \( \text{foo} \), generating inputs, such as \( x = 1 \) and \( y = -4 \), that actually drive the concrete execution to the alternative path.
- With this approach, the engine can explore both paths in \( \text{foo} \) even if \( \text{bar} \) is not symbolically tracked.
Downside: Path Divergence

```c
void baz(int x) {
    abs(&x);
    if (x < 0) ERROR;
}
```

- Function `baz` invokes the external function `abs`, which simply computes the absolute value of a number.
- Choosing $x = 1$ as the initial concrete value, the concrete execution does not trigger the error statement, but the concolic engine tracks the path constraint $\alpha_x \geq 0$ due to the branch in `baz`, trying to generate a new input by negating it.
- However the new input, e.g., $x = -1$, does not trigger the error statement due to the (untracked) side effects of `abs`.
- In this case, after generating a new input the engine detects a path divergence: a concrete execution that does not follow the predicted path.
- Interestingly, in this example no input could actually trigger the error, but the engine is not able to detect this property.
Summary

- Symbolic execution is a popular technique for finding software bugs and vulnerabilities.
- The key idea is to execute a program symbolically, rather than concretely.
- Remaining challenges:
  - path explosion, external environment, constraint solving, etc.