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:
  - A Survey of Symbolic Execution Techniques. arXiv:1610.00502

Example

```
void foobar(int a, int b) {
   int x = 1, y = 0;
   if (a != 0) {
      y = 3+x;
      if (b == 0)
         x = 2*(a+b);
   }
   assert(x-y != 0);
}
```

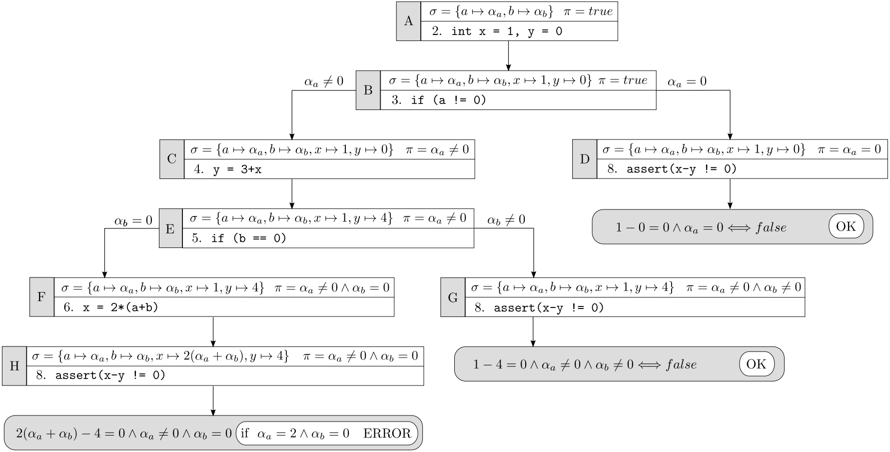
The goal is to find the inputs that make the assertion fail.

- 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: α_a, α_b.
- Symbolic execution maintains a state $(stmt, \sigma, \pi)$:
  - $stmt$: the next statement to evaluate
  - σ: symbolic store
  - π: path constraints
- Depending on $stmt$, symbolic execution proceeds as follows:
  - $x = e$: It updates the symbolic store σ 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 $\pi \wedge e_s$ and $\pi \wedge \neg e_s$.
  - `assert`(e): The validity of e is checked.
    - If $\neg e \wedge \pi$ is unsatisfiable, the assertion is always true.
    - If $\neg e \wedge \pi$ is satisfiable, an assert-fail input is found.

Symbolic Execution Tree



A: $\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b\}$ $\pi = true$

2. `int x = 1, y = 0`

B: $\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 0\}$ $\pi = true$

3. `if (a != 0)`

$\alpha_a \neq 0$

$\alpha_a = 0$

C: $\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 0\}$ $\pi = \alpha_a \neq 0$

4. `y = 3+x`

D: $\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 0\}$ $\pi = \alpha_a = 0$

8. `assert(x-y != 0)`

$1 - 0 = 0 \land \alpha_a = 0 \Longleftrightarrow false$ OK

E: $\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 4\}$ $\pi = \alpha_a \neq 0$

5. `if (b == 0)`

$\alpha_b = 0$

$\alpha_b \neq 0$

F: $\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 4\}$ $\pi = \alpha_a \neq 0 \land \alpha_b = 0$

6. `x = 2*(a+b)`

G: $\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 4\}$ $\pi = \alpha_a \neq 0 \land \alpha_b \neq 0$

8. `assert(x-y != 0)`

$1 - 4 = 0 \land \alpha_a \neq 0 \land \alpha_b \neq 0 \Longleftrightarrow false$ OK

H: $\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 2(\alpha_a + \alpha_b), y \mapsto 4\}$ $\pi = \alpha_a \neq 0 \land \alpha_b = 0$

8. `assert(x-y != 0)`

$2(\alpha_a + \alpha_b) - 4 = 0 \land \alpha_a \neq 0 \land \alpha_b = 0$ if $\alpha_a = 2 \land \alpha_b = 0$ 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.

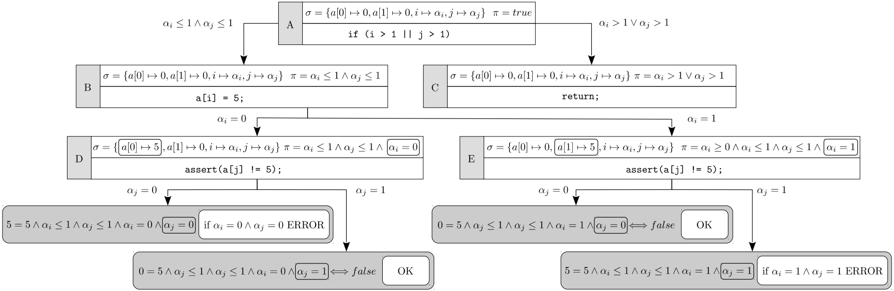
Handling of Pointers and Arrays

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

```
void foobar(unsigned i, unsigned j) {
   int a[2] = { 0 };
   if (i>1 || j>1) return;
   a[i] = 5;
   assert(a[j] != 5);
}
```

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.



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

`if (i > 1 || j > 1)`

$\alpha_i \leq 1 \land \alpha_j \leq 1$

$\alpha_i > 1 \lor \alpha_j > 1$

B: $\sigma = \{a[0] \mapsto 0, a[1] \mapsto 0, i \mapsto \alpha_i, j \mapsto \alpha_j\}$ $\pi = \alpha_i \leq 1 \land \alpha_j \leq 1$

`a[i] = 5;`

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

`return;`

$\alpha_i = 0$

$\alpha_i = 1$

D: $\sigma = \{\boxed{a[0] \mapsto 5}, a[1] \mapsto 0, i \mapsto \alpha_i, j \mapsto \alpha_j\}$ $\pi = \alpha_i \leq 1 \land \alpha_j \leq 1 \land \boxed{\alpha_i = 0}$

`assert(a[j] != 5);`

E: $\sigma = \{a[0] \mapsto 0, \boxed{a[1] \mapsto 5}, i \mapsto \alpha_i, j \mapsto \alpha_j\}$ $\pi = \alpha_i \geq 0 \land \alpha_i \leq 1 \land \alpha_j \leq 1 \land \boxed{\alpha_i = 1}$

`assert(a[j] != 5);`

$\alpha_j = 0$

$\alpha_j = 1$

$5 = 5 \land \alpha_i \leq 1 \land \alpha_j \leq 1 \land \alpha_i = 0 \land \boxed{\alpha_j = 0}$ if $\alpha_i = 0 \land \alpha_j = 0$ ERROR

$0 = 5 \land \alpha_j \leq 1 \land \alpha_j \leq 1 \land \alpha_i = 0 \land \boxed{\alpha_j = 1} \Longleftrightarrow false$ OK

$\alpha_j = 0$

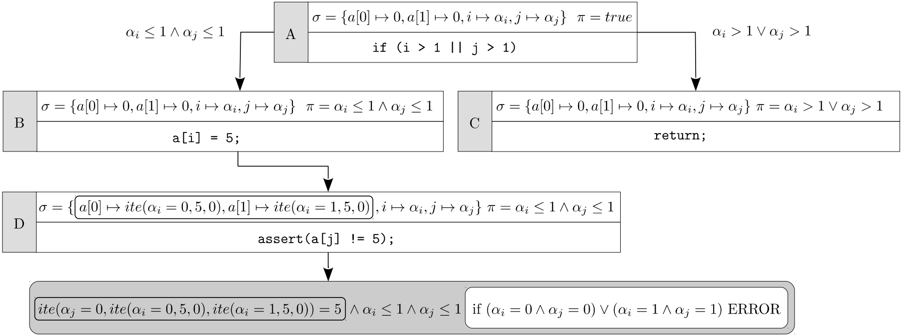
$\alpha_j = 1$

$0 = 5 \land \alpha_j \leq 1 \land \alpha_j \leq 1 \land \alpha_i = 1 \land \boxed{\alpha_j = 0} \Longleftrightarrow false$ OK

$5 = 5 \land \alpha_i \leq 1 \land \alpha_j \leq 1 \land \alpha_i = 1 \land \boxed{\alpha_j = 1}$ if $\alpha_i = 1 \land \alpha_j = 1$ ERROR

If-then-else Formulas

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



A

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

`if (i > 1 || j > 1)`

$\alpha_i \leq 1 \land \alpha_j \leq 1$

$\alpha_i > 1 \lor \alpha_j > 1$

B

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

`a[i] = 5;`

C

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

`return;`

D

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

`assert(a[j] != 5);`

$\boxed{\mathit{ite}(\alpha_j = 0, \mathit{ite}(\alpha_i = 0, 5, 0), \mathit{ite}(\alpha_i = 1, 5, 0)) = 5} \land \alpha_i \leq 1 \land \alpha_j \leq 1$

if $(\alpha_i = 0 \land \alpha_j = 0) \lor (\alpha_i = 1 \land \alpha_j = 1)$ 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:

```
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., $k = 2$:

```
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:

```
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.,

$$\boldsymbol{x + y < 10 \land x > 5 \mapsto \{x = 6, y = 3\}}$$

  - When we query a weaker formula, e.g., $\boldsymbol{x + y < 10}$, we can reuse the previously computed solution, without invoking an SMT solver.
  - When the formula is stronger, e.g., $\boldsymbol{x + y < 10 \land x > 5 \land y \geq 0}$, 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

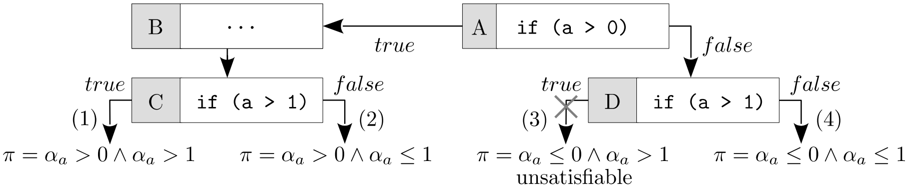
Because symbolic execution forks off a new state at every branch of the program, the total number of states easily becomes exponential in the number of branches. Techniques for addressing path explosion:

- 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,

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

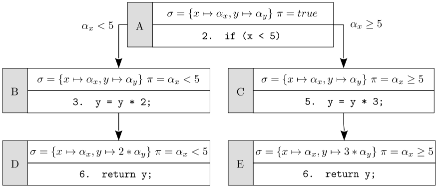


- Eager evaluation calls an SMT solver at each branch.
- Lazy evaluation does not to reduce the burden on the solver.

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. }
```



$\alpha_x < 5$

A $\sigma = \{x \mapsto \alpha_x, y \mapsto \alpha_y\}$ $\pi = true$

`2. if (x < 5)`

$\alpha_x \geq 5$

B $\sigma = \{x \mapsto \alpha_x, y \mapsto \alpha_y\}$ $\pi = \alpha_x < 5$

`3. y = y * 2;`

C $\sigma = \{x \mapsto \alpha_x, y \mapsto \alpha_y\}$ $\pi = \alpha_x \geq 5$

`5. y = y * 3;`

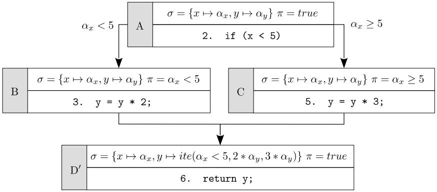
D $\sigma = \{x \mapsto \alpha_x, y \mapsto 2 * \alpha_y\}$ $\pi = \alpha_x < 5$

`6. return y;`

E $\sigma = \{x \mapsto \alpha_x, y \mapsto 3 * \alpha_y\}$ $\pi = \alpha_x \geq 5$

`6. return y;`

without state merging



$\alpha_x < 5$

A $\sigma = \{x \mapsto \alpha_x, y \mapsto \alpha_y\}$ $\pi = true$

`2. if (x < 5)`

$\alpha_x \geq 5$

B $\sigma = \{x \mapsto \alpha_x, y \mapsto \alpha_y\}$ $\pi = \alpha_x < 5$

`3. y = y * 2;`

C $\sigma = \{x \mapsto \alpha_x, y \mapsto \alpha_y\}$ $\pi = \alpha_x \geq 5$

`5. y = y * 3;`

D′ $\sigma = \{x \mapsto \alpha_x, y \mapsto ite(\alpha_x < 5, 2 * \alpha_y, 3 * \alpha_y)\}$ $\pi = true$

`6. return y;`

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 σ' merges σ_1 and σ_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.

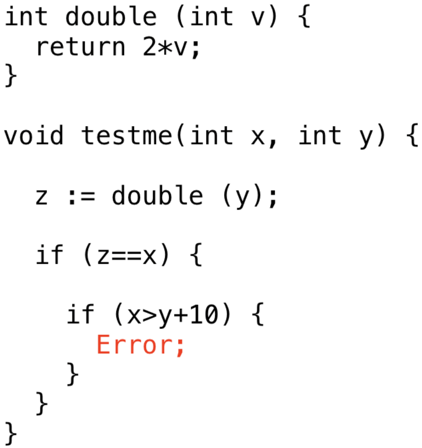
- external calls
- constraint solving
- pointers

Approaches to concolic execution:

- Dynamic symbolic execution (e.g. DART, SAGE, KLEE)
- Selective symbolic execution (e.g. S^2E)

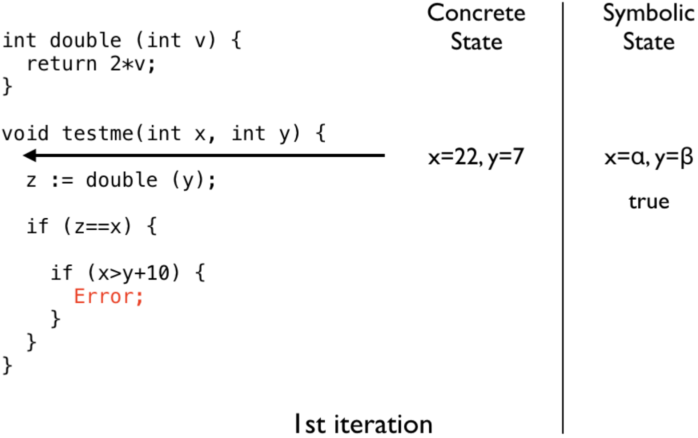
Dynamic Symbolic Execution

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



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

void testme(int x, int y) {

  z := double (y);

  if (z==x) {

    if (x>y+10) {
      Error;
    }
  }
}
```

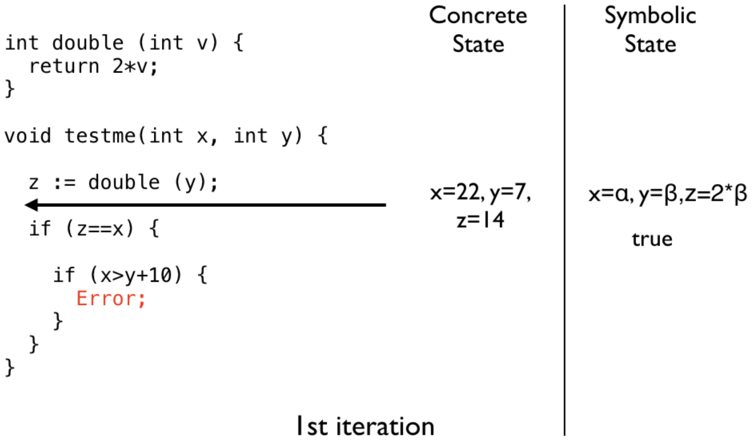
Dynamic Symbolic Execution



```
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

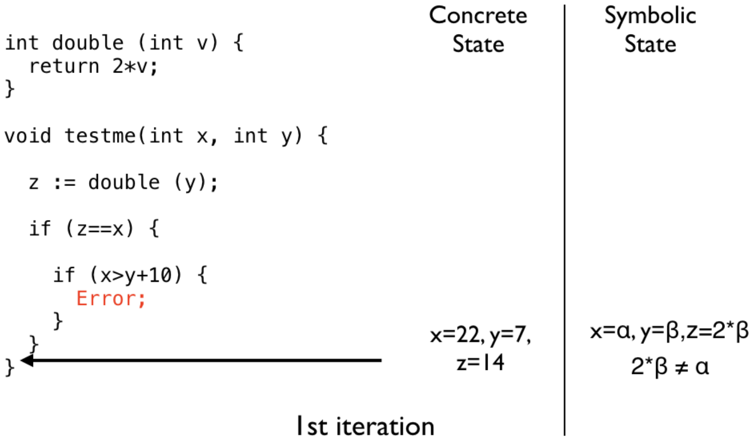
Dynamic Symbolic Execution



```
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

Dynamic Symbolic Execution

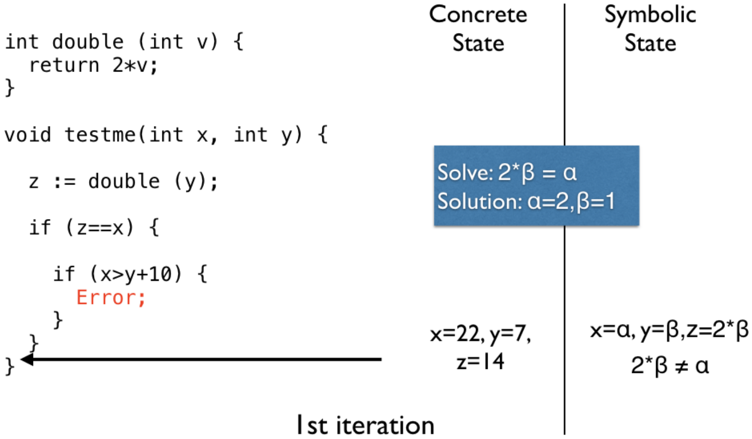


```
int double (int v) {
  return 2*v;
}

void testme(int x, int y) {

  z := double (y);

  if (z==x) {

    if (x>y+10) {
      Error;
    }
  }
}
```

Concrete State	Symbolic State
x=22, y=7, z=14	x=α, y=β,z=2*β 2*β ≠ α

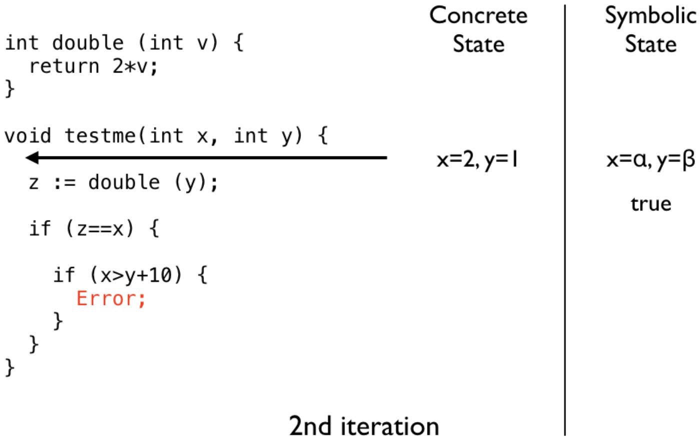
1st iteration

Dynamic Symbolic Execution



```
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



```
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

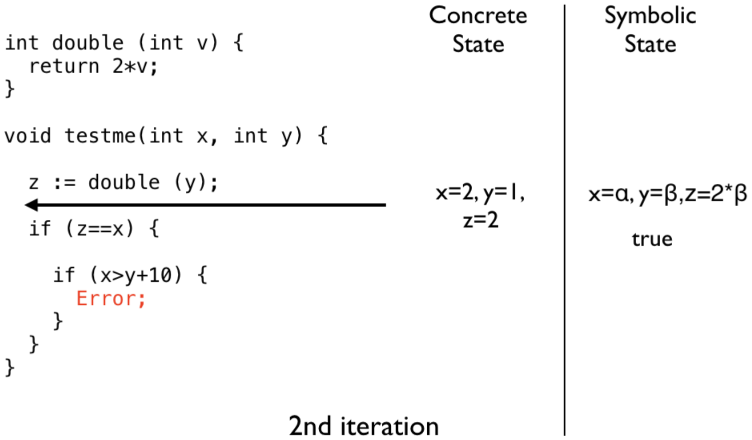
Symbolic State

x=α, y=β

true

2nd iteration

Dynamic Symbolic Execution

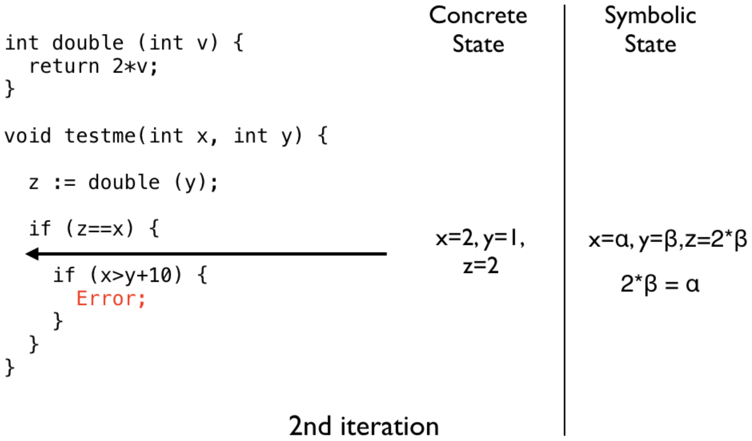


```
int double (int v) {
  return 2*v;
}

void testme(int x, int y) {

  z := double (y);

  if (z==x) {

    if (x>y+10) {
      Error;
    }
  }
}
```

Concrete State	Symbolic State
$x=2, y=1,$ $z=2$	$x=\alpha, y=\beta, z=2*\beta$ true

2nd iteration

Dynamic Symbolic Execution



```
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

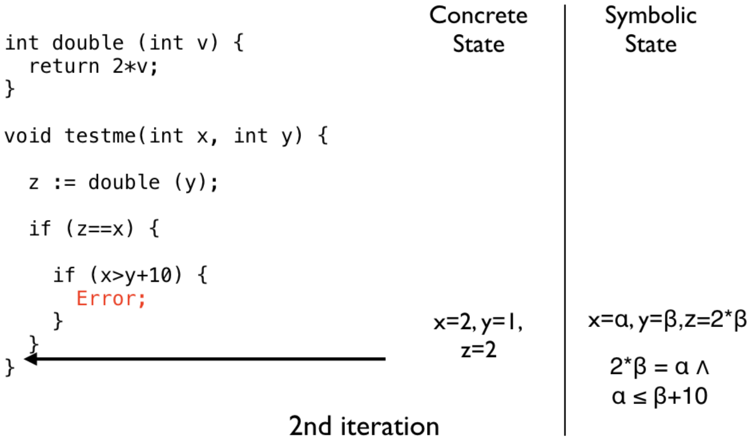
Symbolic State

x=α, y=β, z=2*β

2*β = α

2nd iteration

Dynamic Symbolic Execution

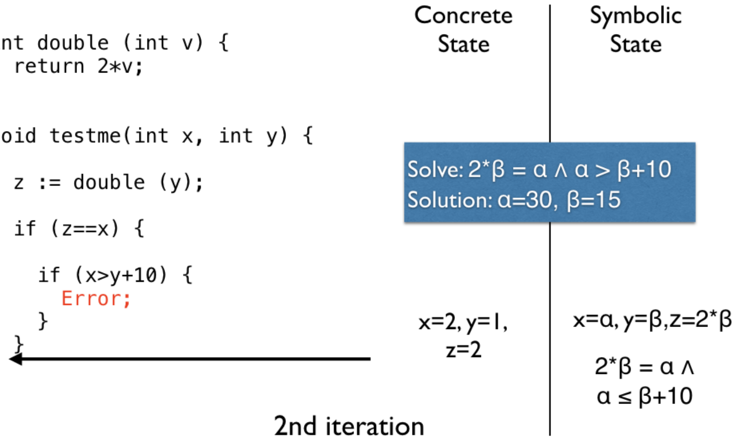


```
int double (int v) {
  return 2*v;
}

void testme(int x, int y) {

  z := double (y);

  if (z==x) {

    if (x>y+10) {
      Error;
    }
  }
}
```

2nd iteration

Dynamic Symbolic Execution

```
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: $2^*\beta = \alpha \wedge \alpha > \beta+10$

Solution: $\alpha=30,\ \beta=15$

$x=2, y=1,$
$z=2$

$x=\alpha, y=\beta, z=2^*\beta$

$2^*\beta = \alpha \wedge$
$\alpha \leq \beta+10$

2nd iteration

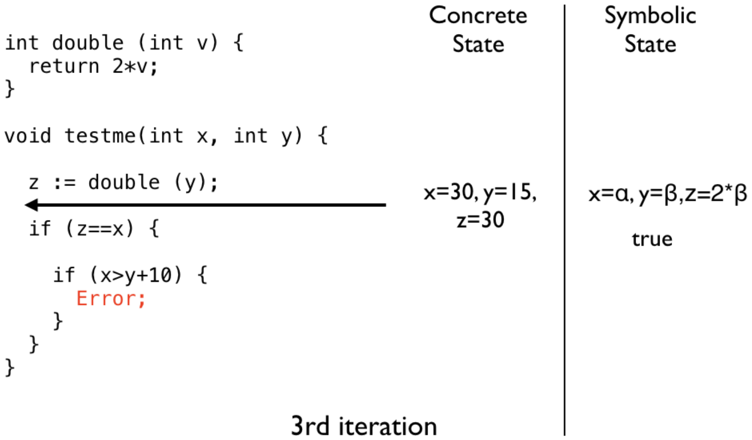
Dynamic Symbolic Execution

```
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
x=30, y=15	x=α, y=β true

3rd iteration

Dynamic Symbolic Execution



```
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

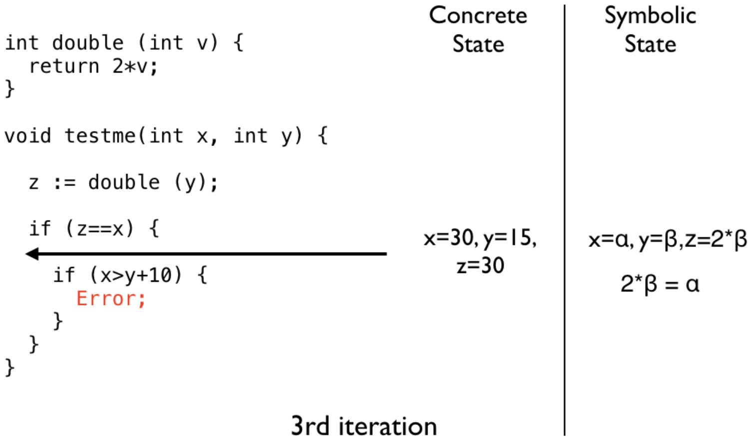
Symbolic State

x=α, y=β,z=2*β

true

3rd iteration

Dynamic Symbolic Execution

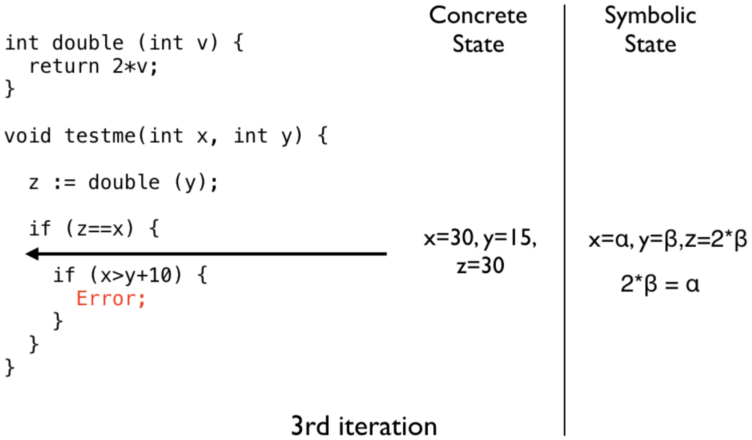


```
int double (int v) {
  return 2*v;
}

void testme(int x, int y) {

  z := double (y);

  if (z==x) {

    if (x>y+10) {
      Error;
    }
  }
}
```

Concrete State	Symbolic State
x=30, y=15, z=30	x=α, y=β,z=2*β 2*β = α

3rd iteration

Dynamic Symbolic Execution



```
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

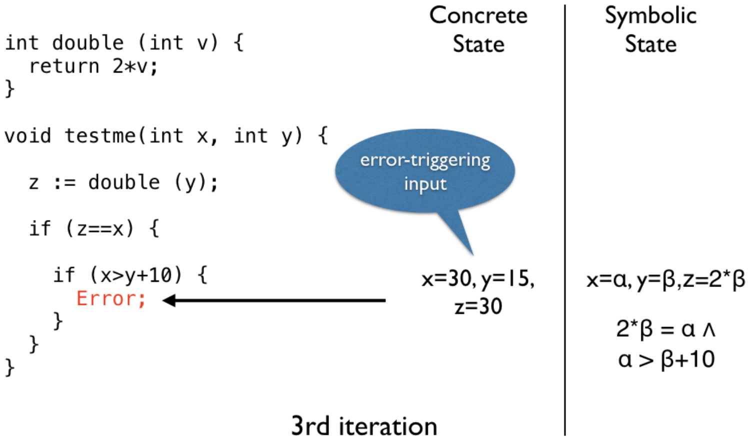
Symbolic State

x=α, y=β, z=2*β

2*β = α

3rd iteration

Dynamic Symbolic Execution



Concrete State

Symbolic State

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

void testme(int x, int y) {

  z := double (y);

  if (z==x) {

    if (x>y+10) {
      Error;
    }
  }
}
```

$x=30, y=15, z=30$

$x=\alpha, y=\beta, z=2^*\beta$

$2^*\beta = \alpha \land \alpha > \beta+10$

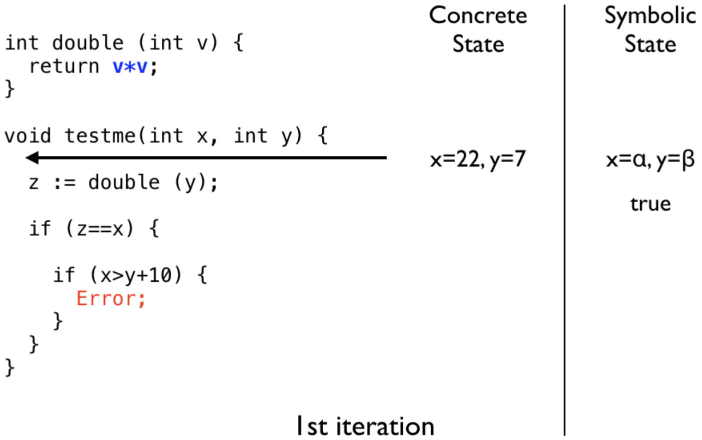
3rd iteration

Dynamic Symbolic Execution

Consider the program with non-linear expression:

```
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

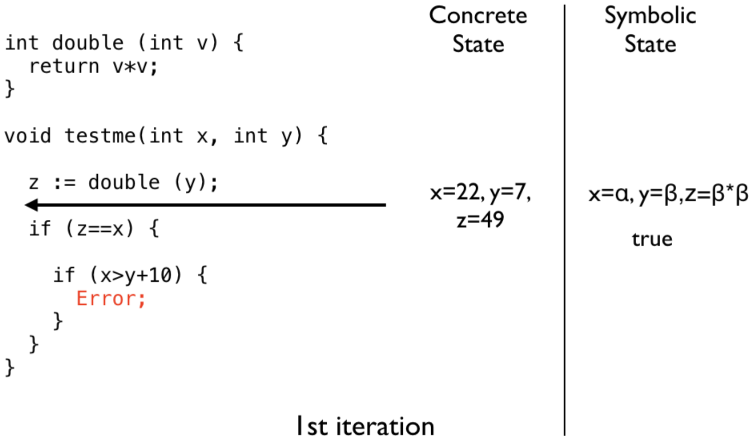


```
int double (int v) {
  return v*v;
}

void testme(int x, int y) {

  z := double (y);

  if (z==x) {

    if (x>y+10) {
      Error;
    }
  }
}
```

Concrete State	Symbolic State
x=22, y=7	x=α, y=β
	true

1st iteration

Dynamic Symbolic Execution

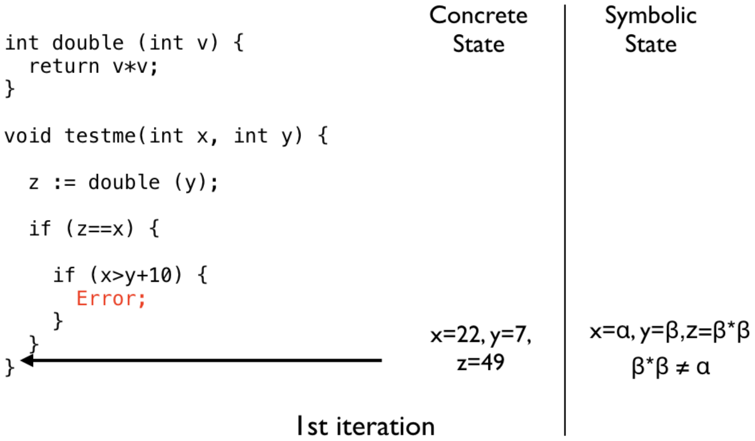


```
int double (int v) {
  return v*v;
}

void testme(int x, int y) {

  z := double (y);

  if (z==x) {

    if (x>y+10) {
      Error;
    }
  }
}
```

Concrete State	Symbolic State
x=22, y=7, z=49	$x=\alpha, y=\beta, z=\beta*\beta$ true

1st iteration

Dynamic Symbolic Execution



```
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

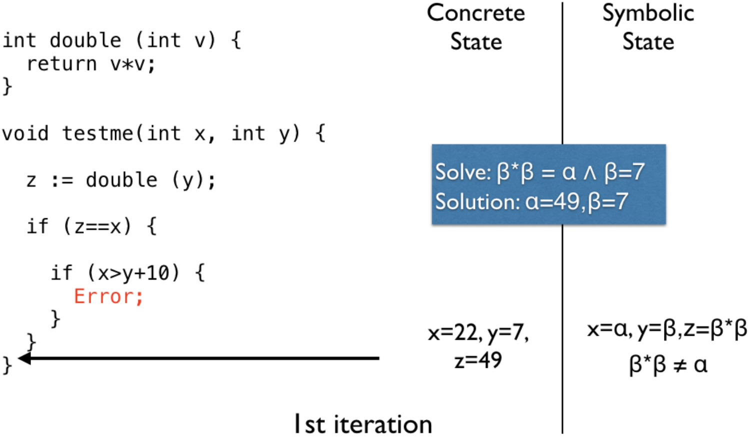
Symbolic State

x=α, y=β, z=β*β

$\beta * \beta \neq \alpha$

1st iteration

Dynamic Symbolic Execution



```
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

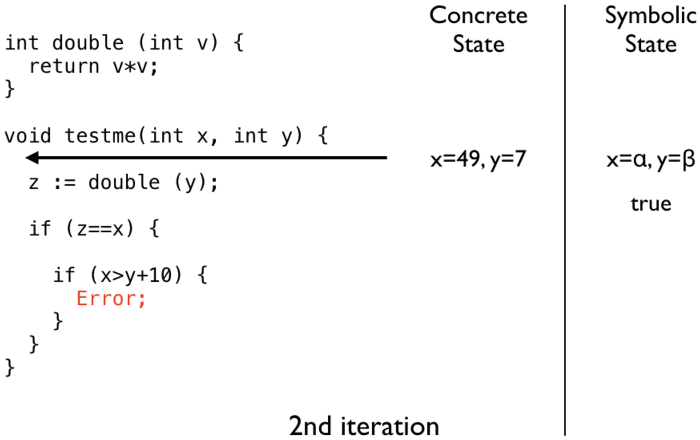
Solve: β*β = α ∧ β=7
Solution: α=49,β=7

x=22, y=7,
z=49

x=α, y=β,z=β*β
β*β ≠ α

1st iteration

Dynamic Symbolic Execution



```
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

Dynamic Symbolic Execution

```
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

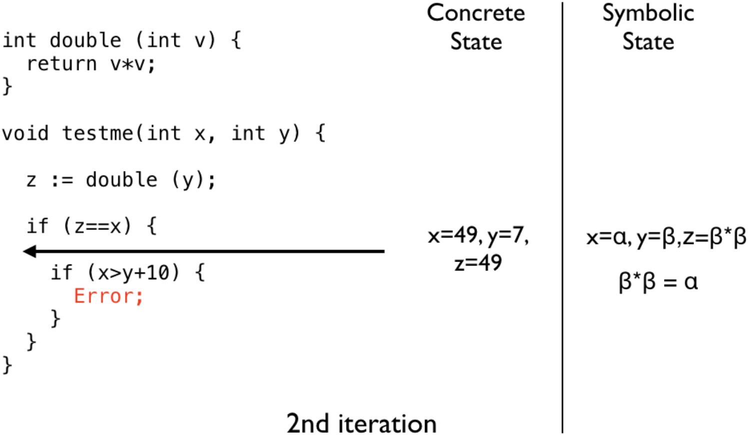
Symbolic State

x=α, y=β,z=β*β

true

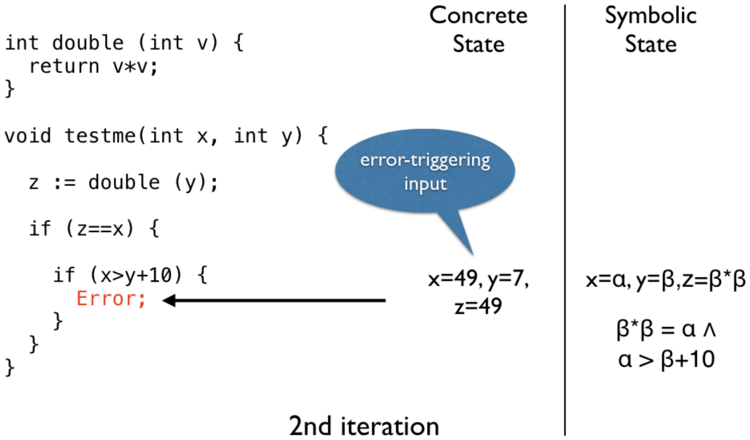
2nd iteration

Dynamic Symbolic Execution



```
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



```
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

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:

```
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 `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 `foo`.
- Notice that branch conditions in function `bar` are not known to the engine.
- To explore the alternative path, the engine negates the path constraint of the branch in `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 `foo` even if `bar` is not symbolically tracked.

Downside: Path Divergence

```
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 $\boldsymbol{x = 1}$ as the initial concrete value, the concrete execution does not trigger the error statement, but the concolic engine tracks the path constraint $\boldsymbol{\alpha_x \geq 0}$ due to the branch in `baz`, trying to generate a new input by negating it.
- However the new input, e.g., $\boldsymbol{x = -1}$, 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