EC4219: Software Engineering

Lecture 14 — Symbolic Execution

Sunbeom So
2024 Spring

Overview

- Automatic test generation approach based on computational logic
- Very widely used for finding software bugs and vulnerabilities.
  - In Microsoft, 30% of bugs have been discovered using symbolic execution.
- Key idea: execute programs "symbolically" using "symbolic" inputs
  - More systematic approach compared to random testing
- Conceptually similar to verification, but each has different goals and different design choices.
  - Verification: prove correctness – by overapproximating behaviors (inferring invariants)
  - Symbolic execution: find errors – by underapproximating behaviors (e.g., unrolling loops)

Symbolic Execution in a Nutshell

- Program inputs are represented by symbols (α, β, etc).
- Maintain a state $(\boldsymbol{stmt}, \boldsymbol{\sigma}, \boldsymbol{\pi})$ where
  - $\boldsymbol{stmt}$: the next statement to evaluate
  - $\boldsymbol{\sigma}$: symbolic store (a mapping from variables to expressions over concrete/symbolic values)
  - $\boldsymbol{\pi}$: path constraints (a formula that expresses a sequence of branches taken to reach $\boldsymbol{stmt}$). At the beginning of the analysis, $\boldsymbol{\pi} = \boldsymbol{true}$.

Symbolic Execution in a Nutshell (Cont'd)

Depending on $stmt$, symbolic execution proceeds as follows.[1]

- $(x := e)^l$: updates the symbolic store σ by associating x with a symbolic expression e_s (obtained by symbolically evaluating e).

$$((x := e)^l, \sigma, \pi) \rightsquigarrow (\textbf{succ}(l), \sigma[x \mapsto e_s], \pi)$$

  where $e_s = \textbf{eval}(e)(\sigma)$.

- s^l : `if` e `then` $s_1^{l_1}$ `else` $s_2^{l_2}$: creates (forks) two new states.

$$(s^l, \sigma, \pi) \rightsquigarrow (s_1^{l_1}, \sigma, \pi \wedge e_s) \text{ and } (s_2^{l_2}, \sigma, \pi \wedge \neg e_s)$$

- `assert`$^l(e)$: check the satisfiability of e and create a new state.
  - If $\pi \wedge \neg e_s$ is satisfiable, report the error.
  - If $\pi \wedge \neg e_s$ is unsatisfiable, e cannot be violated through the path (branch sequence) taken so far.

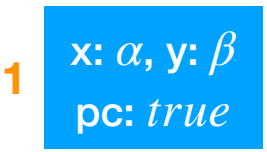
$$(\texttt{assert}^l(e), \sigma, \pi) \rightsquigarrow (\textbf{succ}(l), \sigma, \pi \wedge e)$$

[1] We assume each statement to evaluate has a unique label l.

Example: Symbolic Execution

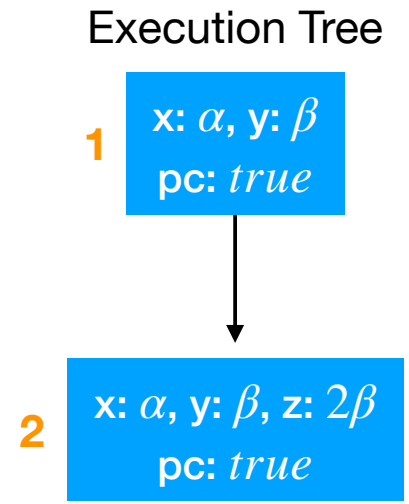
```
  int double (int v) {
    return 2*v;
  }

  void testme(int x, int y) {
1
    z := double (y);
2
    if (z==x) {
    3
      if (x>y+10) {
      4 Crash
      } else { 5 }
    }
6 }
```

Execution Tree



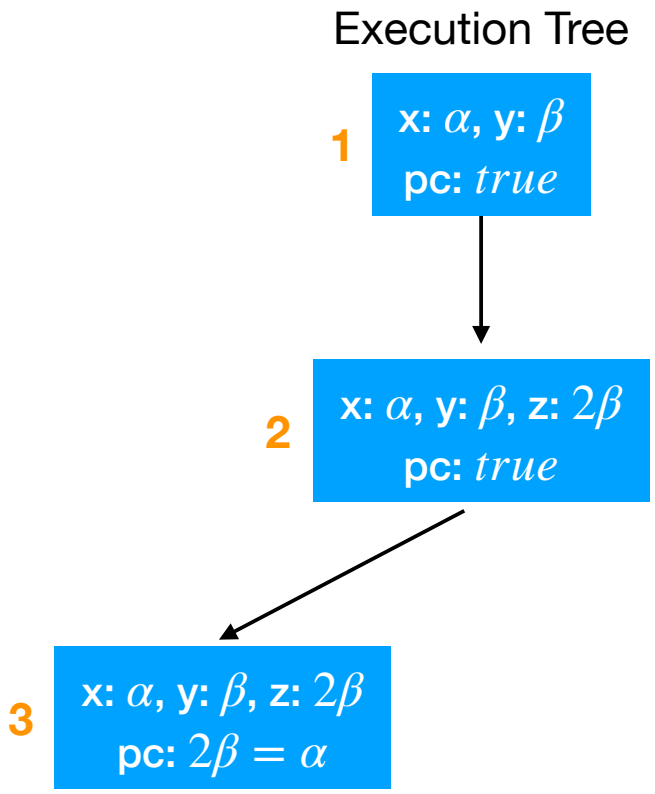
Example: Symbolic Execution (Cont'd)

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

  void testme(int x, int y) {
1
    z := double (y);
2
    if (z==x) {
    3
      if (x>y+10) {
      4 Crash
      } else { 5 }
    }
6 }
```



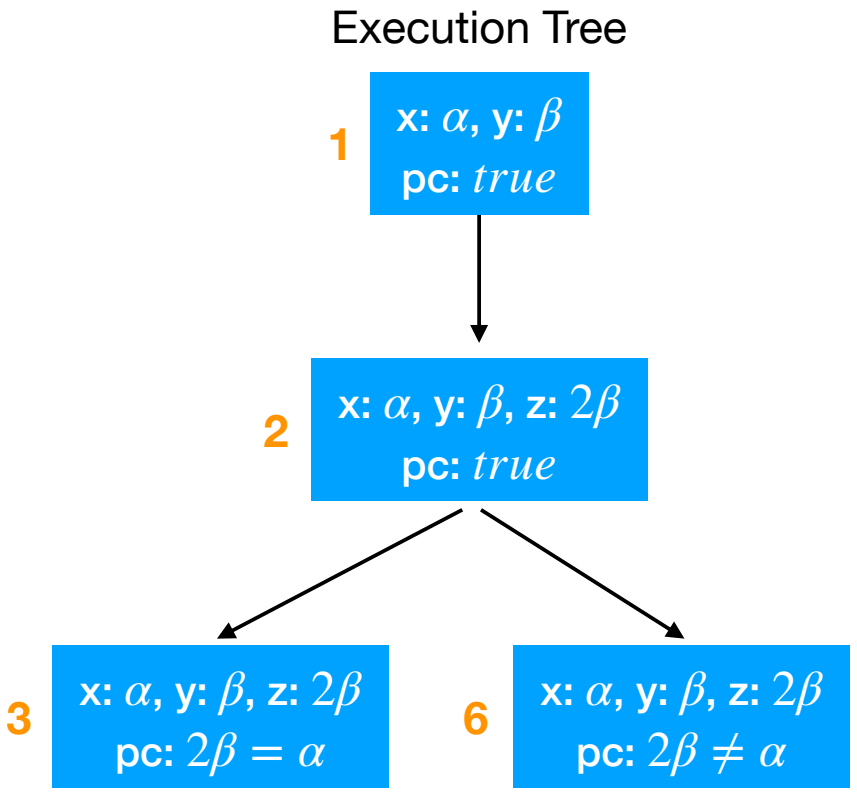
Example: Symbolic Execution (Cont'd)

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

  void testme(int x, int y) {
1
    z := double (y);
2
    if (z==x) {
    3
      if (x>y+10) {
      4 Crash
      } else { 5 }
    }
6 }
```



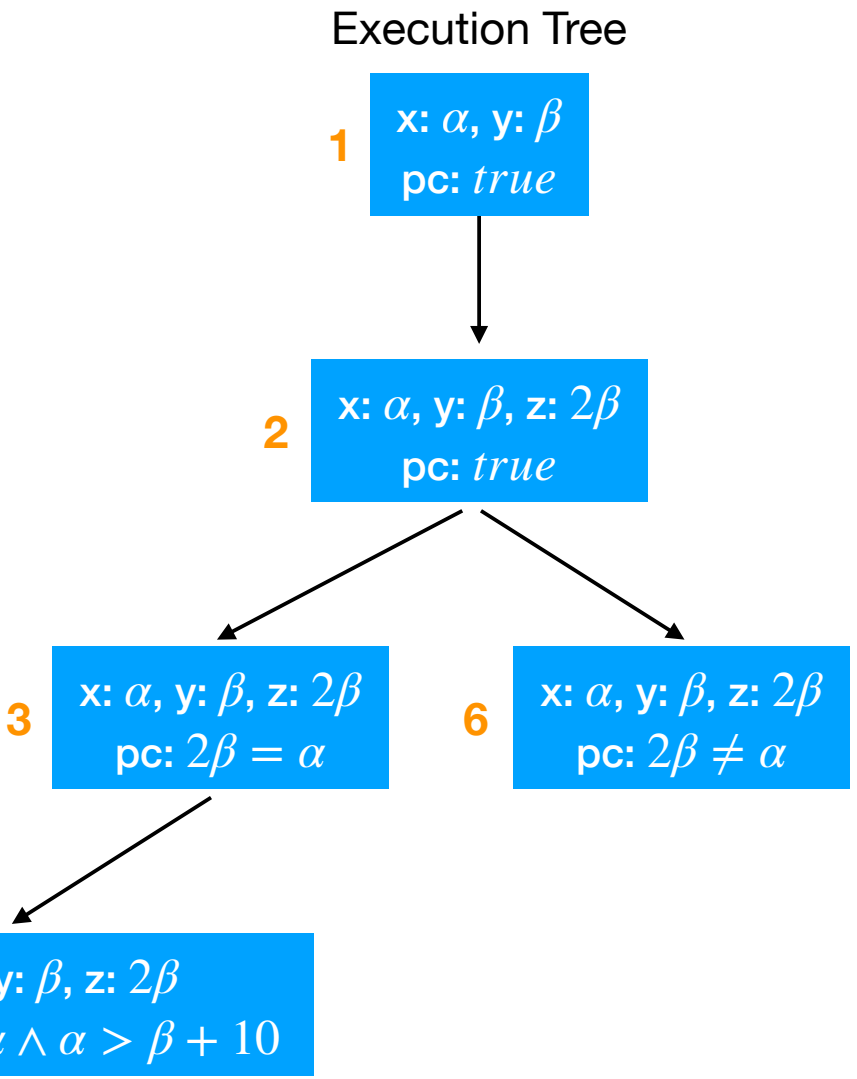
Example: Symbolic Execution (Cont'd)

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

  void testme(int x, int y) {
1
    z := double (y);
2
    if (z==x) {
    3
      if (x>y+10) {
       4 Crash
      } else { 5 }
    }
6 }
```



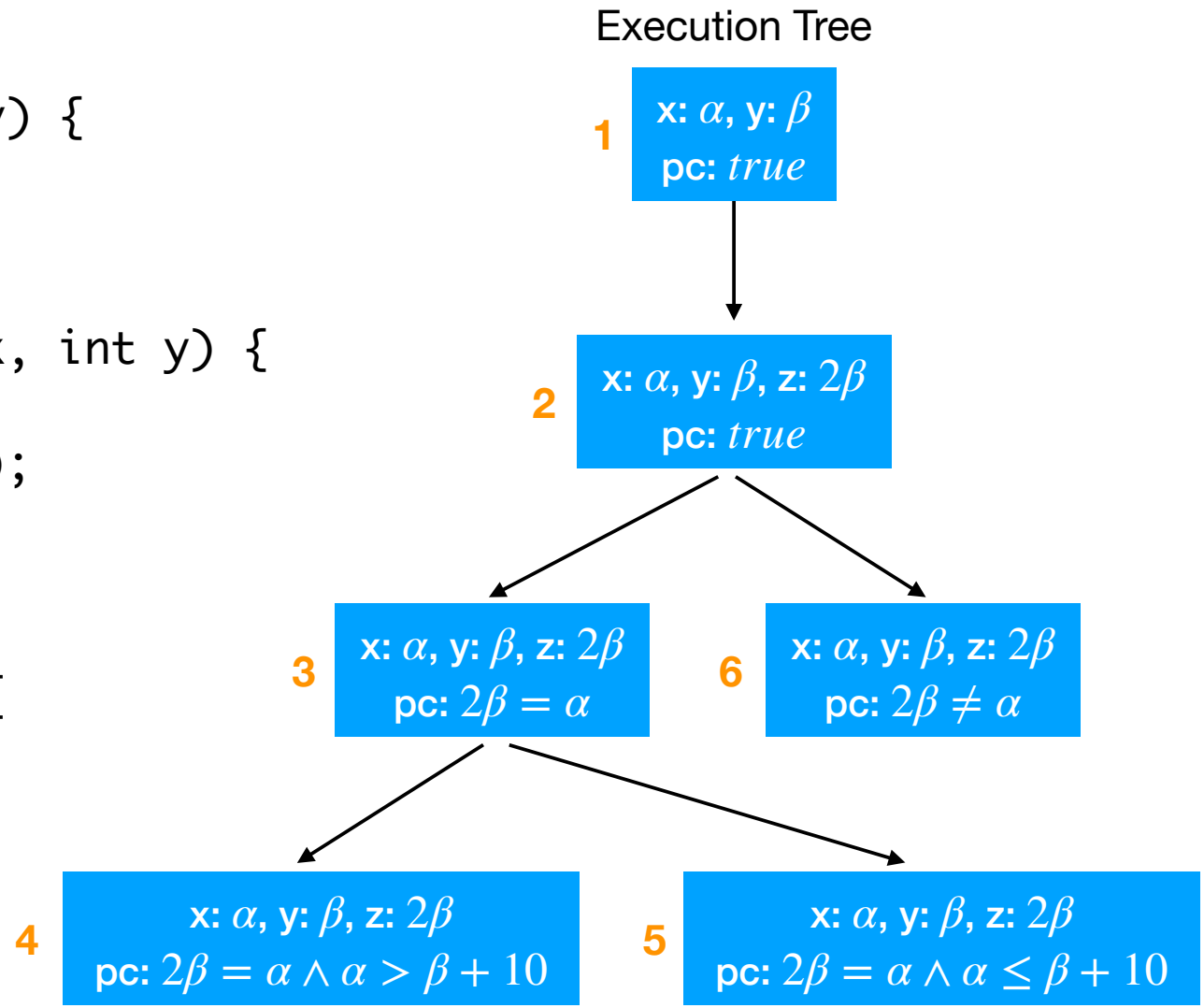
Example: Symbolic Execution (Cont'd)

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

void testme(int x, int y) {
1
  z := double (y);
2
  if (z==x) {
  3
    if (x>y+10) {
     4 Crash
    } else { 5 }
  }
6 }
```



Example: Symbolic Execution (Cont'd)

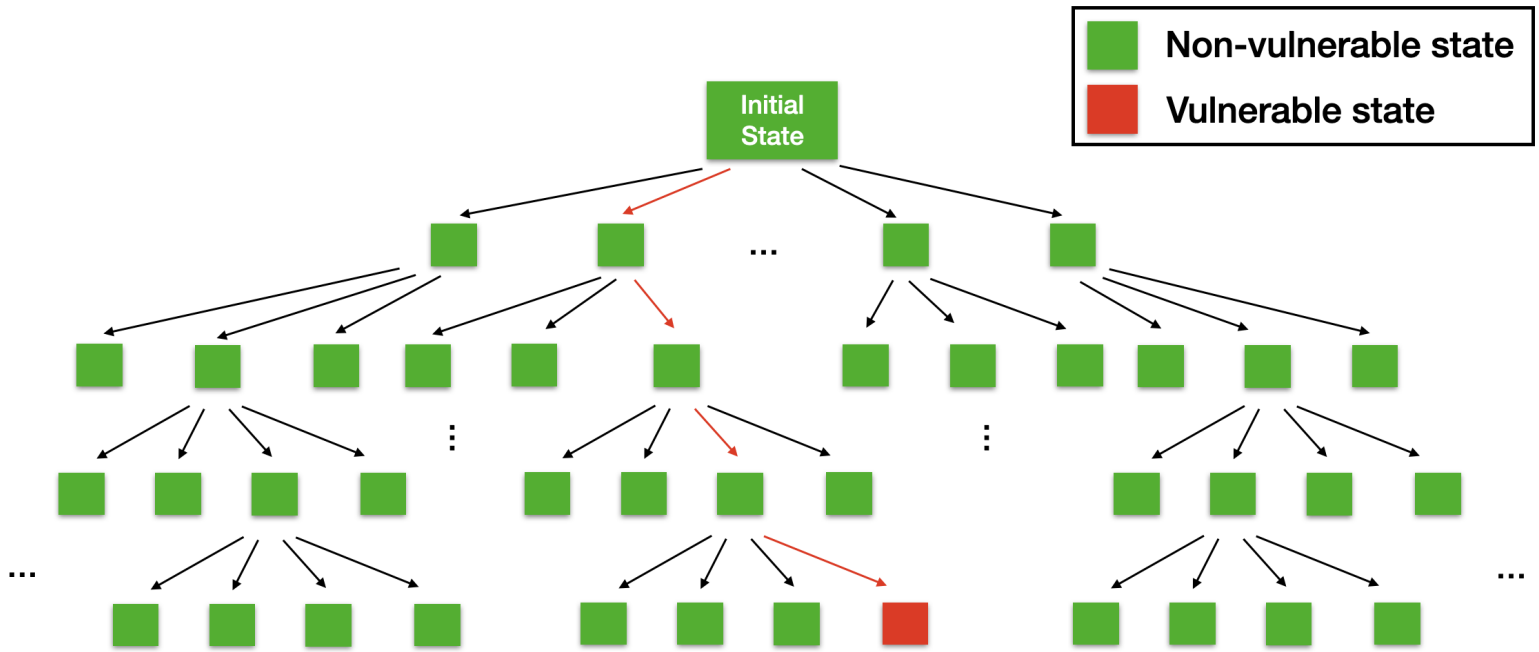
```
int double (int v) {
  return 2*v;
}

void testme(int x, int y) {
1
  z := double (y);
2
  if (z==x) {
    3
    if (x>y+10) {
      4 Crash
    } else { 5 }
  }
6 }
```



Challenges in Symbolic Execution (1): Path Explosion

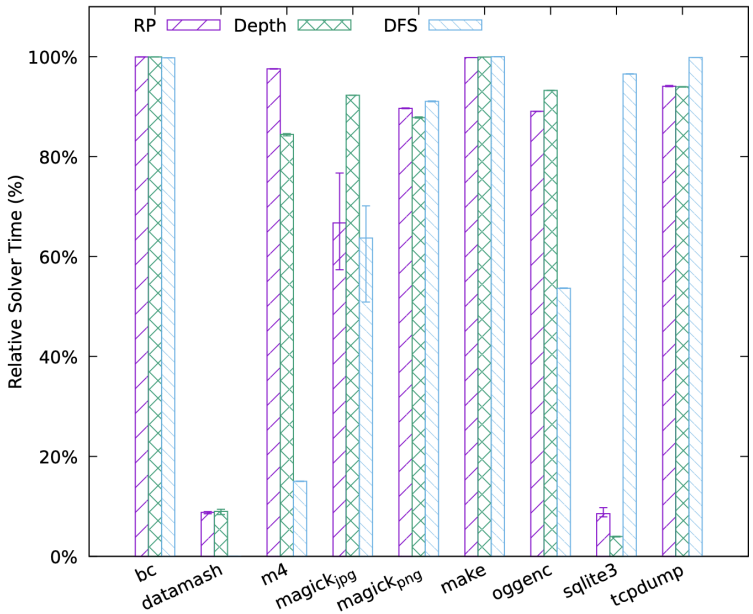
- Given n if/while statements, #path $= 2^n$ (worst case)
  - E.g., suppose there are multiple invocations of the function f containing 10 if-statements. We already have more than 1M ($= 2^{10} * 2^{10}$) execution paths!

$$\texttt{f(...);} \cdots \texttt{;f(...);} \cdots$$



Challenges in Symbolic Execution (2): Constraint Solving

- Despite the recent advances in SMT solvers, constraint solving is still a major bottleneck.
  - According to the recent report,[2] more than 80% of total testing time is spent on SMT solving (invoking a solver and waiting for its response).
- Constraint solving is sometimes practically impossible.
  - E.g., external function calls whose source code is unknown, hash function calls whose logic is extremely complicated



[2] Pending Constraints in Symbolic Execution for Better Exploration and Seeding. ASE 2020

Challenges in Symbolic Execution (2): Constraint Solving

Existing SMT solvers are highly unlikely to solve constraints that involve cryptographically secure hash functions.

```
int foo (int v) {
  return secure_hash(v);
}

void testme(int x, int y) {

  z := foo (y);

  if (z==x) {

    if (x>y+10) {
      Crash
    } else {   }
  }
}
```

Dynamic Symbolic Execution (Concolic Testing)

- An approach to mitigate the limitations of (pure) symbolic execution (in particular, the difficulty of constraint solving)
- Concolic: A compound word of concrete and symbolic
- Example of hybrid analysis: combines dynamic and static analysis
  1. Start with random input values
  2. Concretely execute the program using the inputs. During this step, keep track of both concrete states and symbolic states.
  3. Assuming the path constraint $\Phi : \phi_1 \wedge \cdots \wedge \phi_n$ upon termination is given, solve the constraint Φ' where one of the conjunct of Φ is negated. E.g., $\Phi' : \phi_1 \wedge \cdots \wedge \neg\phi_n$.
  4. Concretely execute the program using the inputs obtained by solving Φ'.
  5. Repeat Step 2–4.
- Symbolic execution (solving Φ') guides concrete execution to increase code coverage.
- Concrete execution guides symbolic execution to simplify constraints (optimize Φ' before solving it).

Example: Concolic Testing

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

void testme(int x, int y) {
  z := double (y);

  if (z==x) {

    if (x>y+10) {
      Crash
    } else {   }
  }
}
```

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

1st iteration

Example: Concolic Testing (Cont'd)

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

void testme(int x, int y) {

  z := double (y);
  if (z==x) {

    if (x>y+10) {
      Crash
    } else {   }
  }
}
```

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

1st iteration

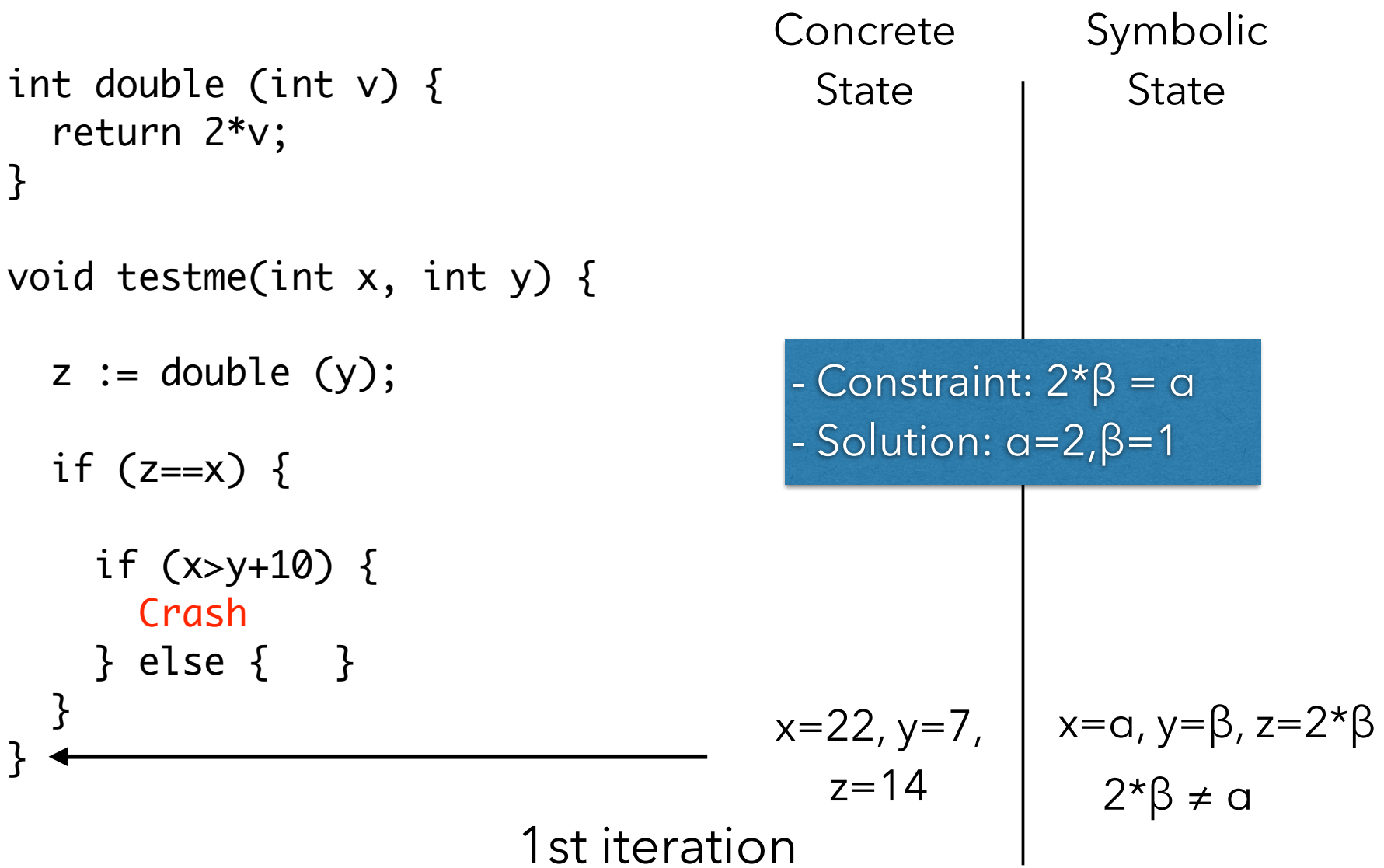
Example: Concolic Testing (Cont'd)

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

void testme(int x, int y) {

  z := double (y);

  if (z==x) {

    if (x>y+10) {
      Crash
    } else {   }
  }
}
```

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

1st iteration

Example: Concolic Testing (Cont'd)



Example: Concolic Testing (Cont'd)

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

void testme(int x, int y) {
  z := double (y);

  if (z==x) {

    if (x>y+10) {
      Crash
    } else {   }
  }
}
```

Concrete State	Symbolic State
x=2, y=1	x=α, y=β
	true

2nd iteration

Example: Concolic Testing (Cont'd)

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

void testme(int x, int y) {

  z := double (y);
  if (z==x) {

    if (x>y+10) {
      Crash
    } else {   }
  }
}
```

Concrete State	Symbolic State
x=2, y=1, z=2	x=α, y=β, z=2*β
	true

2nd iteration

Example: Concolic Testing (Cont'd)

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

void testme(int x, int y) {

  z := double (y);

  if (z==x) {
    if (x>y+10) {
      Crash
    } else {   }
  }
}
```

Concrete State	Symbolic State
x=2, y=1, z=2	x=α, y=β, z=2*β 2*β = α

2nd iteration

Example: Concolic Testing (Cont'd)

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

void testme(int x, int y) {

  z := double (y);

  if (z==x) {

    if (x>y+10) {
      Crash
    } else {   }
  }
}
```

Concrete State	Symbolic State
x=2, y=1, z=2	x=α, y=β, z=2*β 2*β = α ∧ α ≤ β+10

2nd iteration

Example: Concolic Testing (Cont'd)

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

void testme(int x, int y) {

  z := double (y);

  if (z==x) {

    if (x>y+10) {
      Crash
    } else {   }
  }
}
```

Concrete State	Symbolic State
	- Constraint: $2*\beta = \alpha \wedge \alpha > \beta+10$ - Solution: $\alpha=30, \beta=15$
x=2, y=1, z=2	x=α, y=β, z=2*β $2*\beta = \alpha \wedge$ $\alpha \leq \beta+10$

2nd iteration

Example: Concolic Testing (Cont'd)

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

void testme(int x, int y) {
  z := double (y);

  if (z==x) {

    if (x>y+10) {
      Crash
    } else {   }
  }
}
```

	Concrete State	Symbolic State
←	x=30, y=15	x=α, y=β
		true

3rd iteration

Example: Concolic Testing (Cont'd)

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

void testme(int x, int y) {

  z := double (y);
  if (z==x) {

    if (x>y+10) {
      Crash
    } else {   }
  }
}
```

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

3rd iteration

Example: Concolic Testing (Cont'd)

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

void testme(int x, int y) {

  z := double (y);

  if (z==x) {
    if (x>y+10) {
      Crash
    } else {   }
  }
}
```

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

3rd iteration

Example: Concolic Testing (Cont'd)

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

void testme(int x, int y) {

  z := double (y);

  if (z==x) {

    if (x>y+10) {
      Crash
    } else {   }
  }
}
```

Concrete State	Symbolic State
x=30, y=15, z=30	x=α, y=β, z=2*β 2*β = α ∧ α > β+10

crashing input

3rd iteration

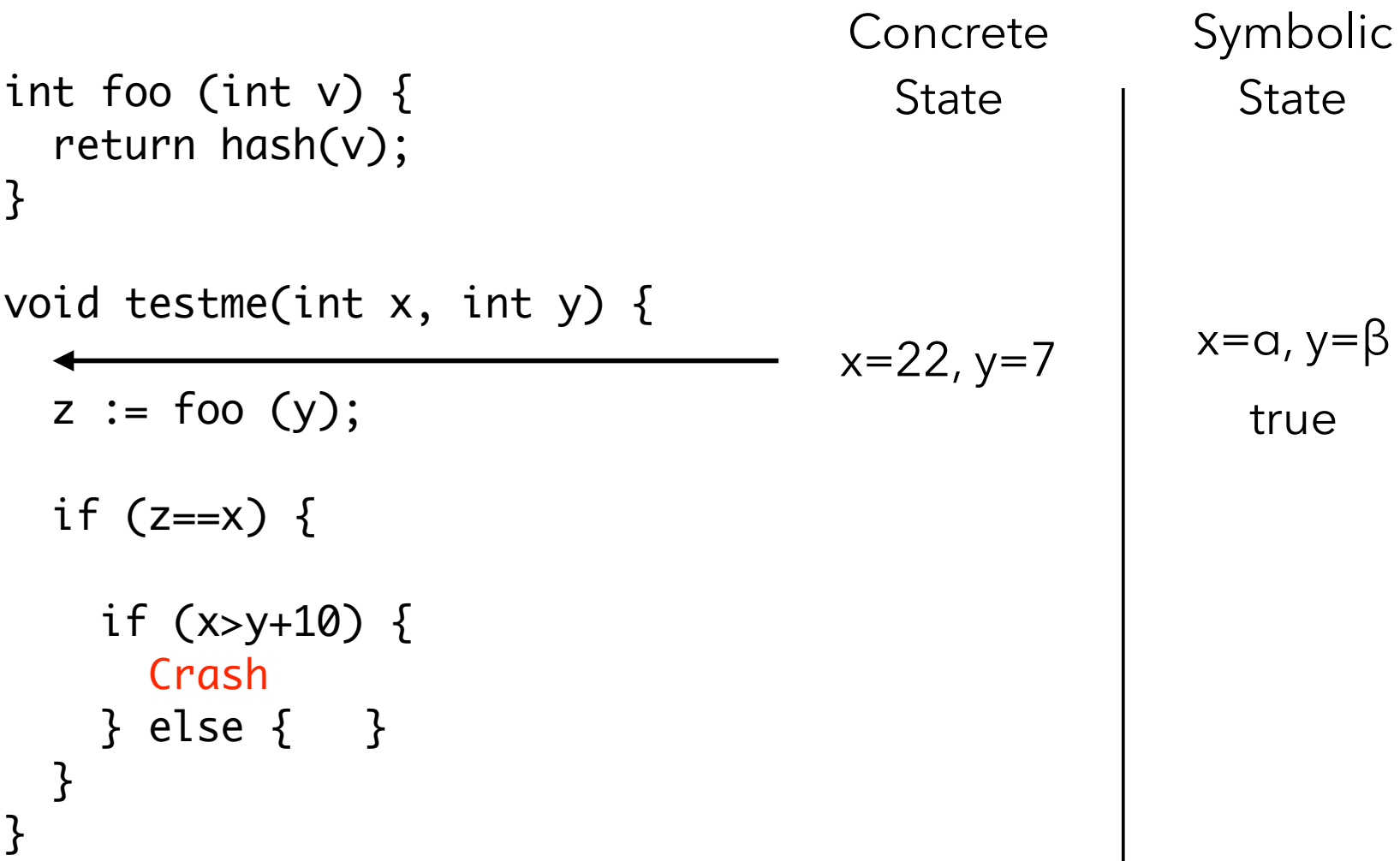
Concolic Testing Algorithm

Input : Program P, initial input vector v_0, budget N
Output: The number of branches covered

1: $T \leftarrow \langle\rangle$
2: $v \leftarrow v_0$
3: **for** $m = 1$ to N **do**
4: $\quad \Phi_m \leftarrow \text{RunProgram}(P, v)$
5: $\quad T \leftarrow T \cdot \Phi_m$
6: $\quad$ **repeat**
7: $\quad\quad (\Phi, \phi_i) \leftarrow \text{Choose}(T) \qquad (\Phi = \phi_1 \wedge \cdots \wedge \phi_n)$
8: $\quad$ **until** $\text{SAT}(\bigwedge_{j<i} \phi_j \wedge \neg\phi_i)$
9: $\quad v \leftarrow \text{model}(\bigwedge_{j<i} \phi_j \wedge \neg\phi_i)$
10: **end for**
11: **return** $|\text{Branches}(T)|$

DSE's Advantage: Unknown/Complex Calls

Recall: "Concrete execution guides symbolic execution to simplify constraints (optimize Φ' before solving it)."



DSE's Advantage: Unknown/Complex Calls (Cont'd)

Recall: "Concrete execution guides symbolic execution to simplify constraints (optimize Φ' before solving it)."

```
int foo (int v) {
  return hash(v);
}

void testme(int x, int y) {

  z := foo (y);
  if (z==x) {

    if (x>y+10) {
      Crash
    } else {   }
  }
}
```

Concrete State	Symbolic State
x=22, y=7, z=601...129	x=α, y=β, z=hash(β) true

DSE's Advantage: Unknown/Complex Calls (Cont'd)

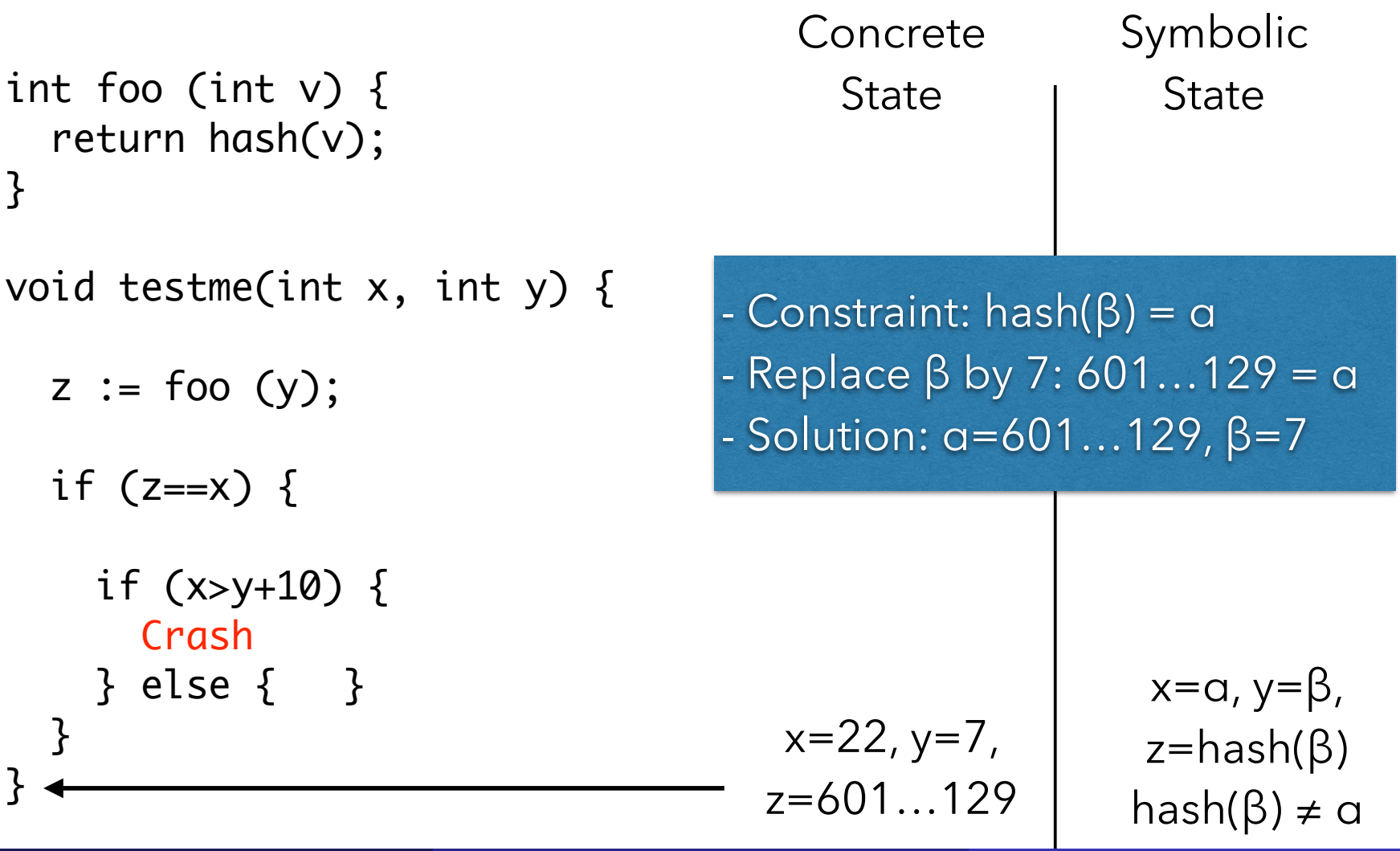
Recall: "Concrete execution guides symbolic execution to simplify constraints (optimize Φ' before solving it)."

```
int foo (int v) {
  return hash(v);
}

void testme(int x, int y) {

  z := foo (y);

  if (z==x) {

    if (x>y+10) {
      Crash
    } else {   }
  }
}
```

Concrete State	Symbolic State
x=22, y=7, z=601…129	x=α, y=β, z=hash(β) hash(β) ≠ α

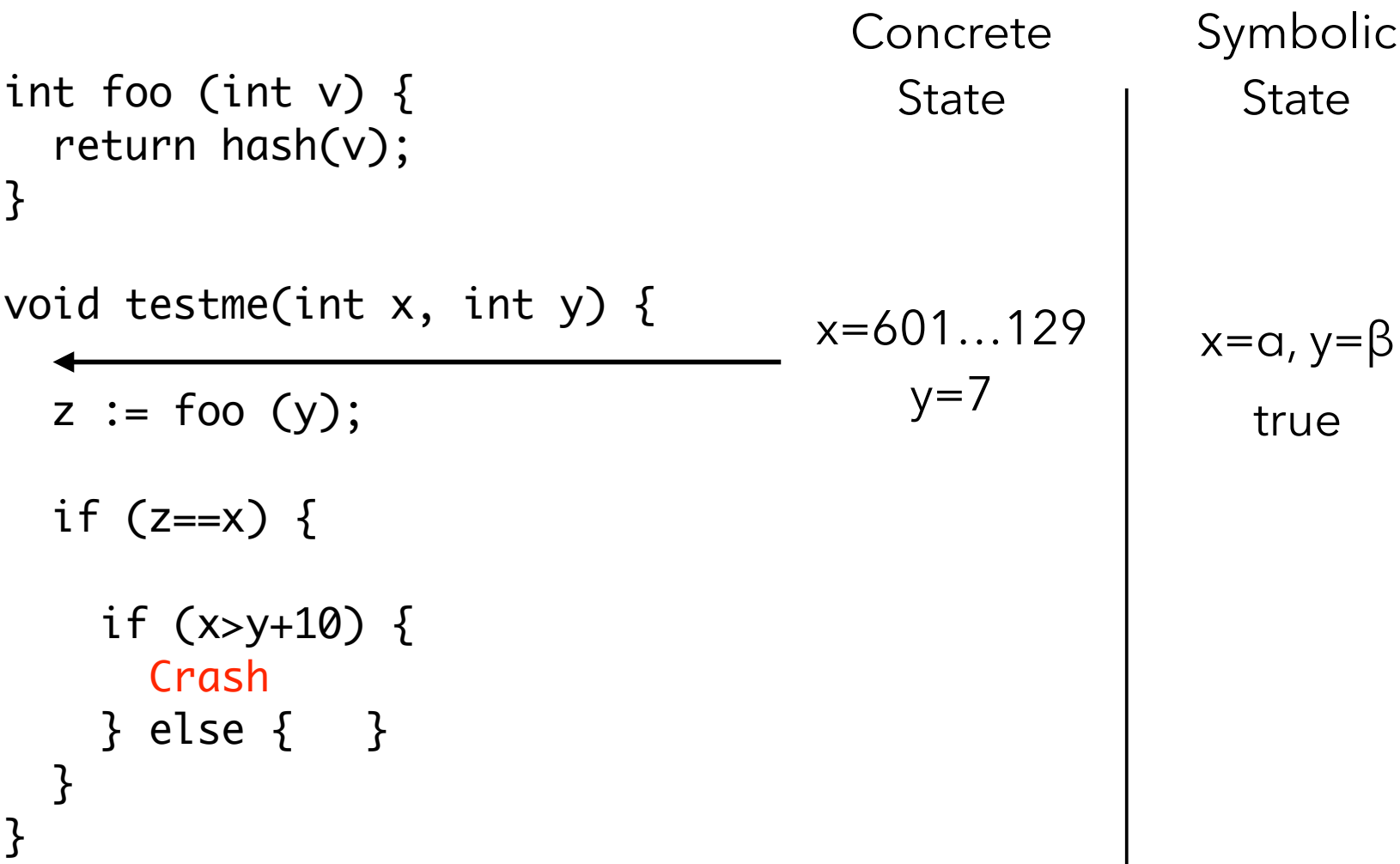
DSE's Advantage: Unknown/Complex Calls (Cont'd)

Recall: "Concrete execution guides symbolic execution to simplify constraints (optimize Φ' before solving it)."



DSE's Advantage: Unknown/Complex Calls (Cont'd)

Recall: "Concrete execution guides symbolic execution to simplify constraints (optimize Φ' before solving it)."



DSE's Advantage: Unknown/Complex Calls (Cont'd)

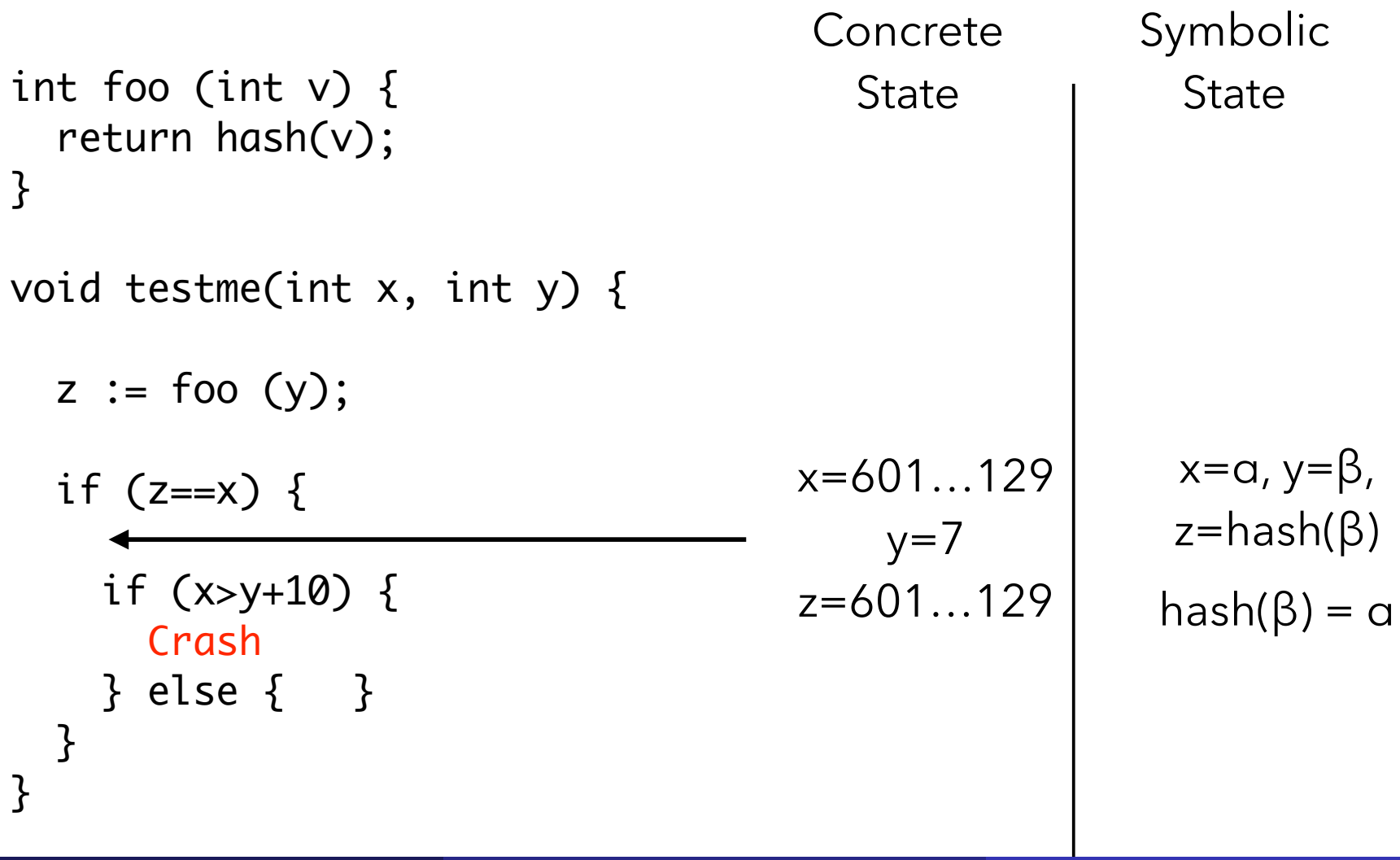
Recall: "Concrete execution guides symbolic execution to simplify constraints (optimize Φ' before solving it)."

```
int foo (int v) {
  return hash(v);
}

void testme(int x, int y) {

  z := foo (y);
  if (z==x) {

    if (x>y+10) {
      Crash
    } else {   }
  }
}
```

Concrete State	Symbolic State
x=601...129 y=7 z=601...129	x=α, y=β, z=hash(β) true

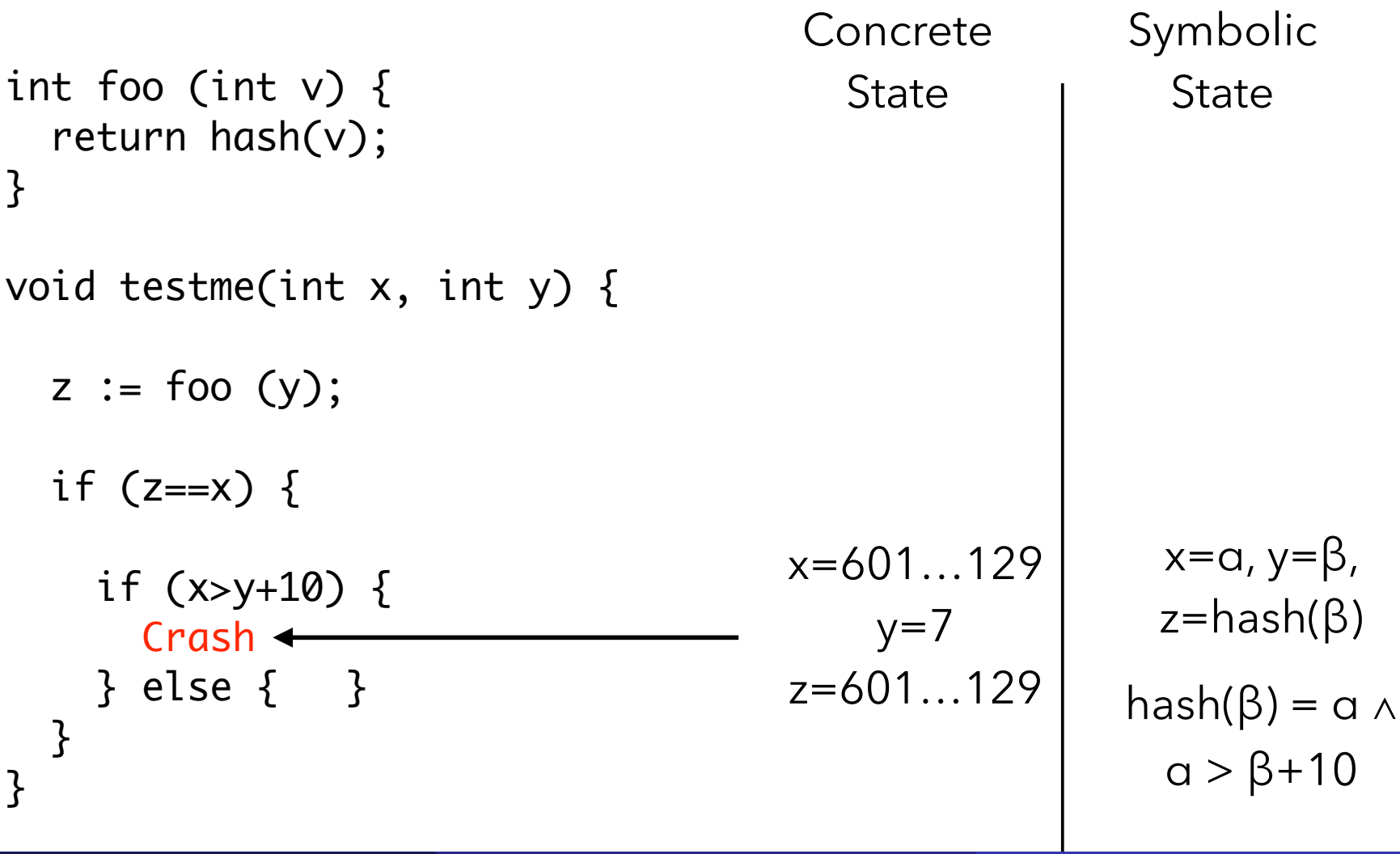
DSE's Advantage: Unknown/Complex Calls (Cont'd)

Recall: "Concrete execution guides symbolic execution to simplify constraints (optimize Φ' before solving it)."



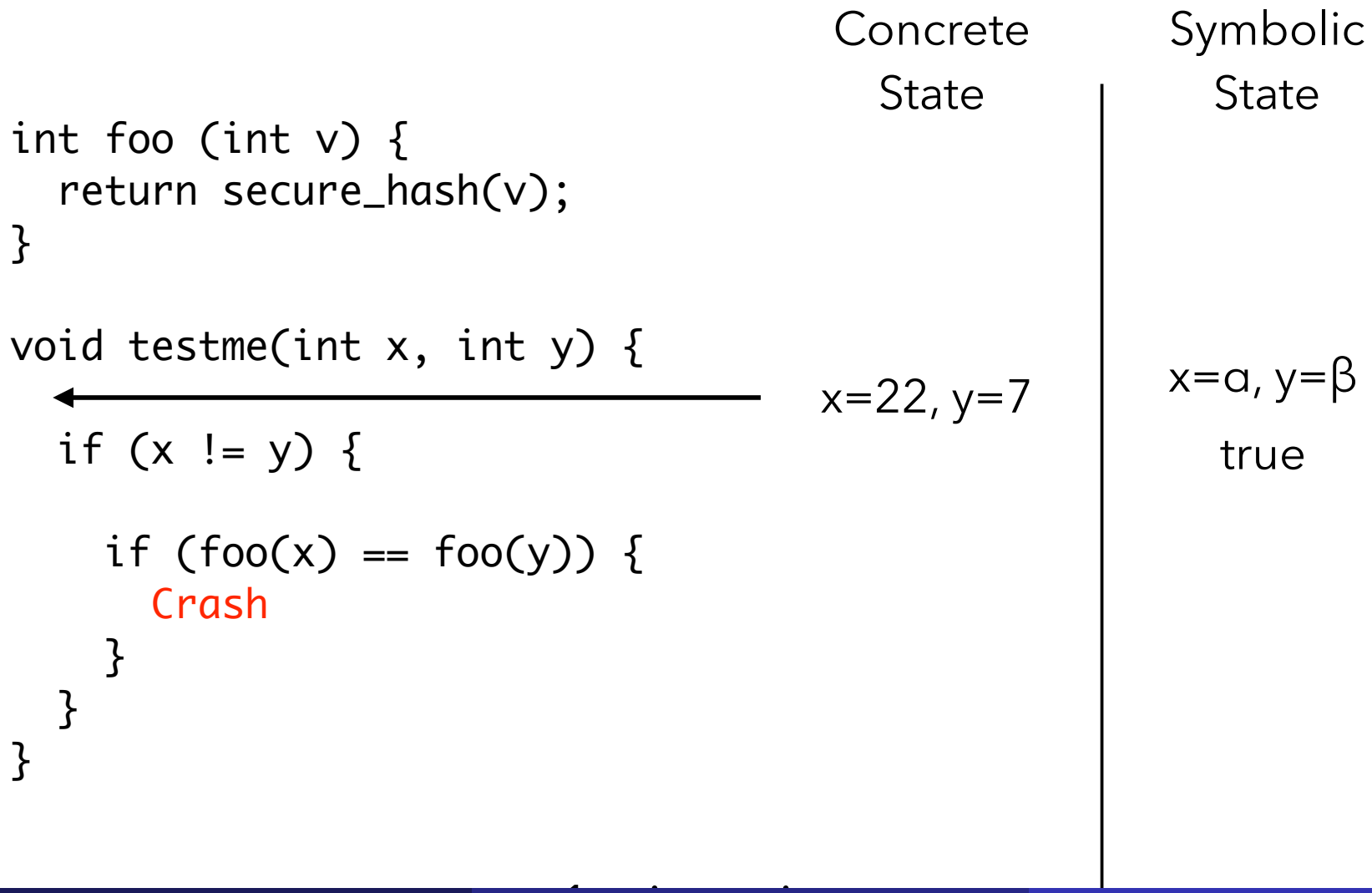
DSE's Advantage: Unknown/Complex Calls (Cont'd)

Recall: "Concrete execution guides symbolic execution to simplify constraints (optimize Φ' before solving it)."



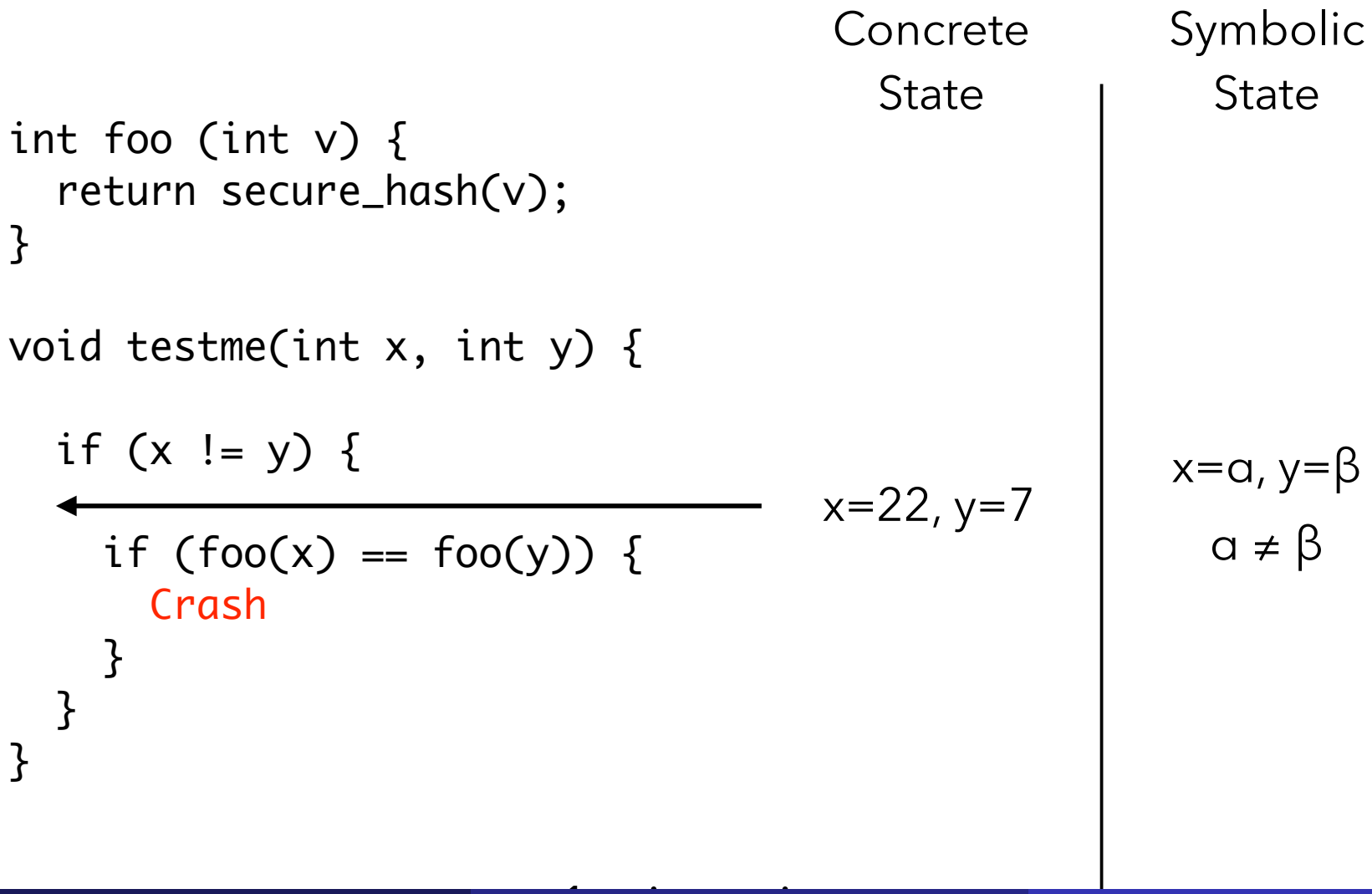
DSE's Downside: False Negative

Due to constraints simplified using runtime information, satisfiable constraints might be judged as unsatisfiable during DSE.



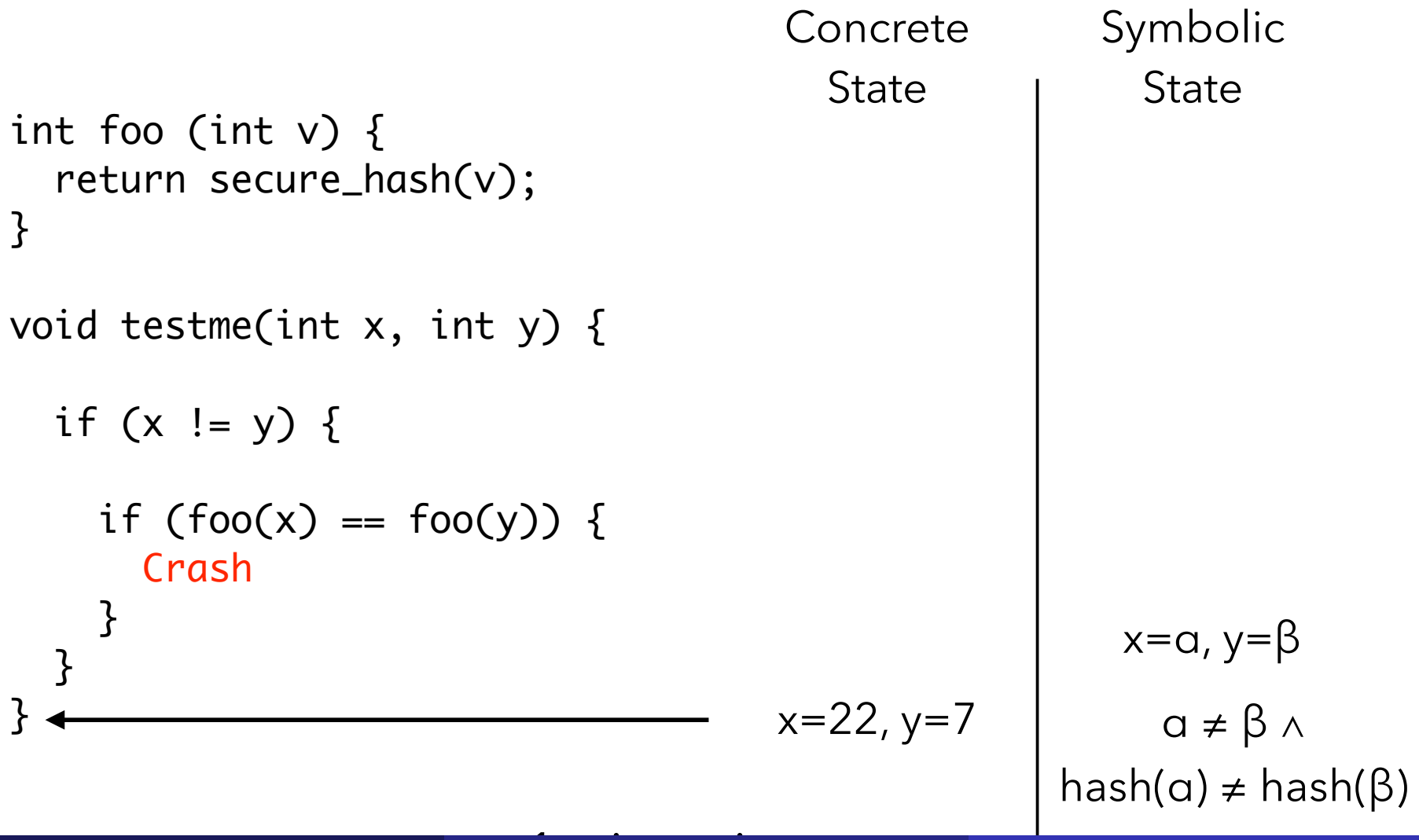
DSE's Downside: False Negative (Cont'd)

Due to constraints simplified using runtime information, satisfiable constraints might be judged as unsatisfiable during DSE.



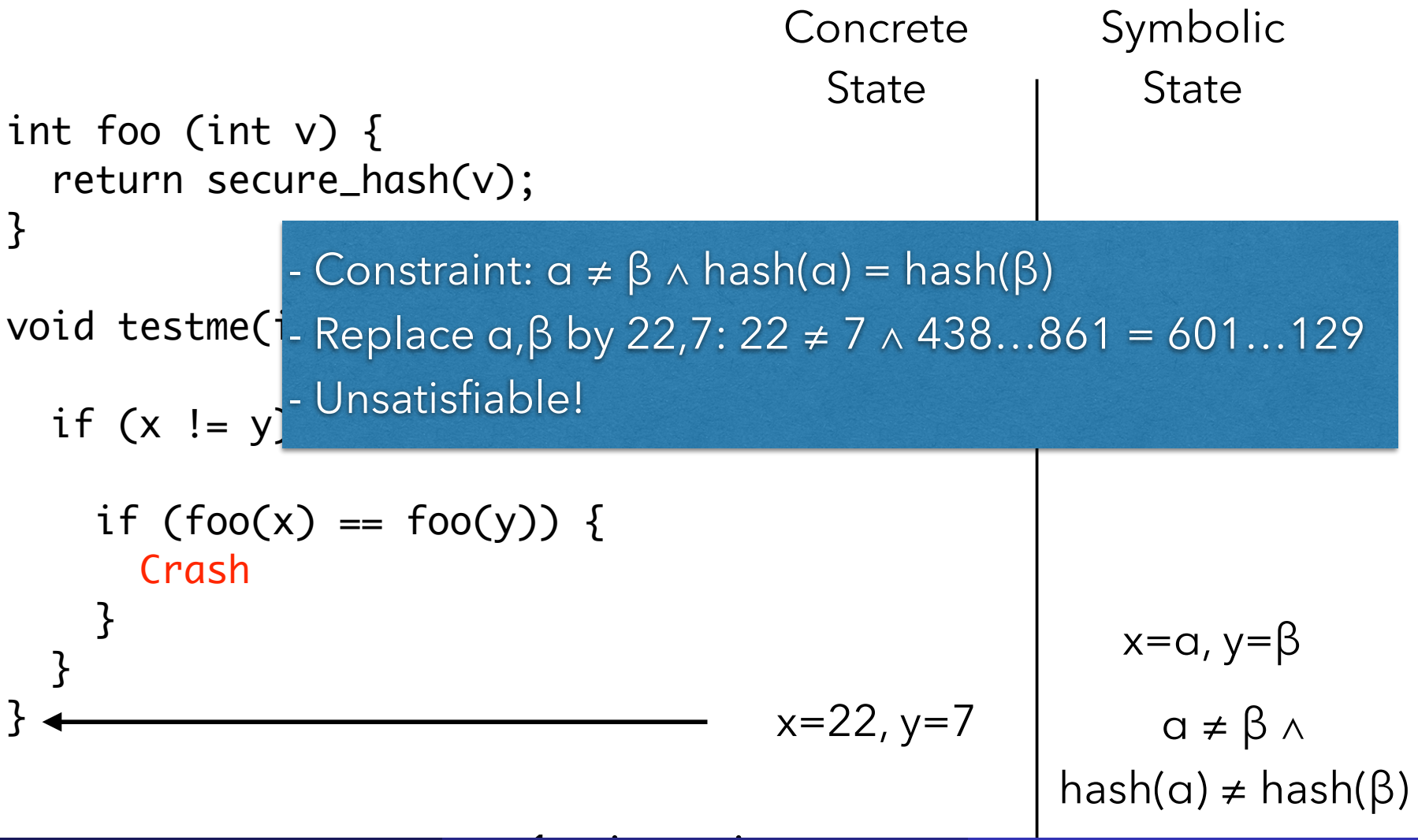
DSE's Downside: False Negative (Cont'd)

Due to constraints simplified using runtime information, satisfiable constraints might be judged as unsatisfiable during DSE.



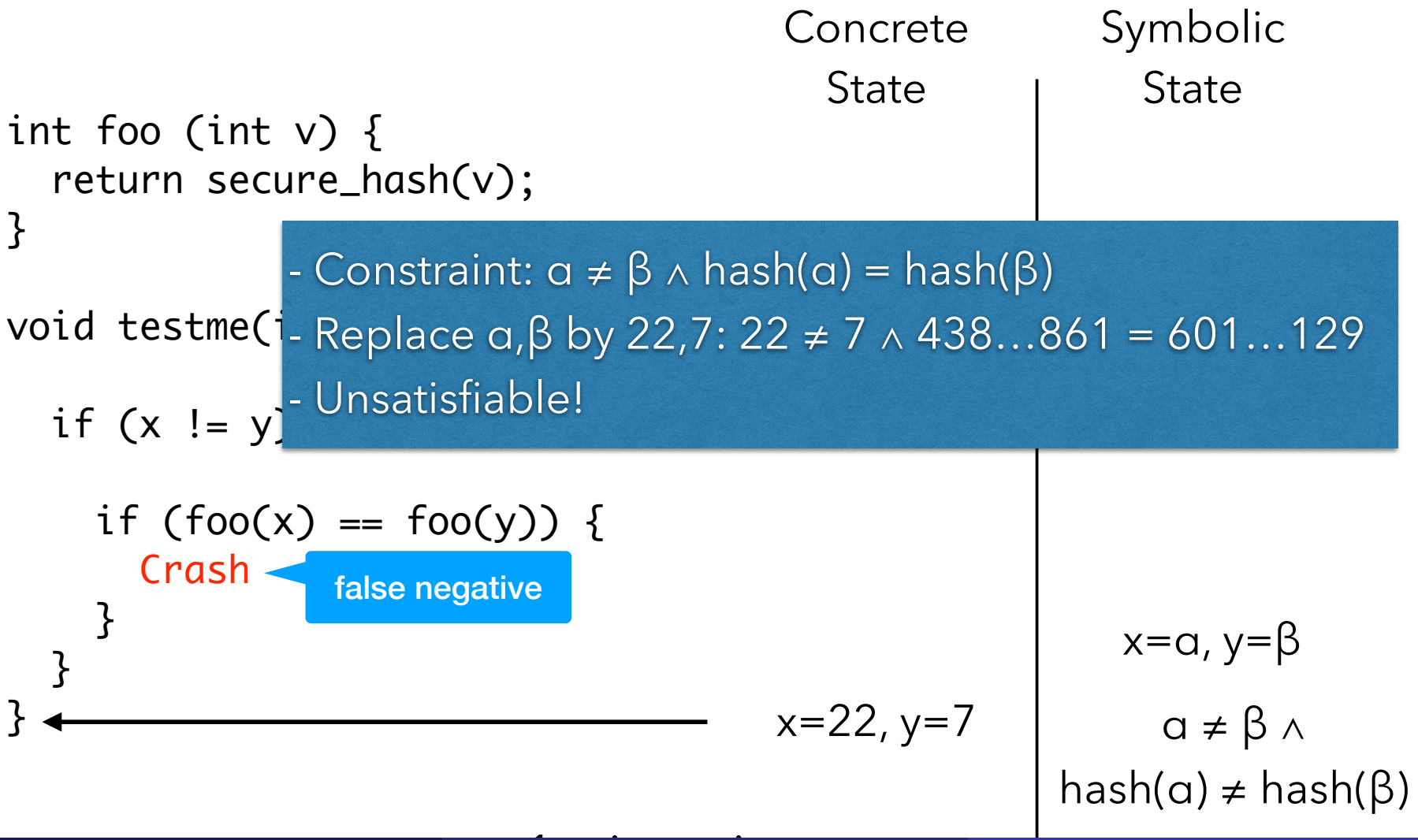
DSE's Downside: False Negative (Cont'd)

Due to constraints simplified using runtime information, satisfiable constraints might be judged as unsatisfiable during DSE.



DSE's Downside: False Negative (Cont'd)

Due to constraints simplified using runtime information, satisfiable constraints might be judged as unsatisfiable during DSE.



Summary

- Symbolic execution is a testing approach based on computational logic.
  - Key idea: execute programs "symbolically" using "symbolic" inputs
- Main challenges: path explosion, constraint solving
- DSE is an effective hybrid approach that (partly) addresses constraint solving.
  - Symbolic execution (solving a path constraint) guides concrete execution to increase code coverage.
  - Concrete execution guides symbolic execution to simplify constraints (optimize a path constraint before solving it).