# L35-36: Concolic Execution

**Software Analysis**

IIT Guwahati · Oct 28 / 29

# Recap: Problems of Symbolic Execution (1/2)

- Loops and recursion: infinite execution trees
- Path explosion: number of paths is exponential in the number of conditionals
- Environment modeling: dealing with native, system and library calls
- Solver limitations: dealing with complex path conditions
- Heap modeling: symbolic representation of data structures and pointers

# Recap: Problems of Symbolic Execution (2/2)

- Loops and recursion: infinite execution trees
- Path explosion: number of paths is exponential in the number of conditionals
- **Environment modeling**: dealing with native, system and library calls
- **Solver limitations**: dealing with complex path conditions
- **Heap modeling**: symbolic representation of data structures and pointers

# Concolic Testing (1/4)

Mix **conc**rete and symb**olic** execution = **"concolic"**
also called **dynamic symbolic execution** (DSE)

```
1  int foo(int v) {
2    return 2*v;
3  }

5  void test_me(int x, int y) {
6    int z = foo(y);
7    if (z == x)
8      if (x > y+10)
9        ERROR;
10 }
```

| concrete run | symbolic run |
|---|---|
| x = 22, y = 7 | $x = x_0,\ y = y_0$ |

Pradel, Program Analysis, Stuttgart: Symbolic and Concolic Execution; Godefroid, Klarlund and Sen, PLDI 2005

# Concolic Testing (2/4)

Mix **conc**rete and symb**olic** execution = **"concolic"**
also called **dynamic symbolic execution** (DSE)

```
1  int foo(int v) {
2    return 2*v;
3  }

5  void test_me(int x, int y) {
6    int z = foo(y);
7    if (z == x)
8      if (x > y+10)
9        ERROR;
10 }
```

| concrete run | symbolic run |
|---|---|
| | |
| | |
| x = 22, y = 7 | $x = x_0,\ y = y_0$ |
| z = 14 | $z = 2y_0$ |

Pradel, Program Analysis, Stuttgart: Symbolic and Concolic Execution; Godefroid, Klarlund and Sen, PLDI 2005

# Concolic Testing (3/4)

Mix **conc**rete and symb**olic** execution = **"concolic"**
also called **dynamic symbolic execution** (DSE)

```
1  int foo(int v) {
2    return 2*v;
3  }

5  void test_me(int x, int y) {
6    int z = foo(y);
7    if (z == x)
8      if (x > y+10)
9        ERROR;
10 }
```

| concrete run | symbolic run |
|---|---|
| x = 22, y = 7 | $x = x_0,\ y = y_0$ |
| z = 14 | $z = 2y_0$ |
| 14 == 22: false ⟶ | record $2y_0 \neq x_0$ |

the concrete value picks the branch

Pradel, Program Analysis, Stuttgart: Symbolic and Concolic Execution; Godefroid, Klarlund and Sen, PLDI 2005

# Concolic Testing (4/4)

Mix **concrete** and symb**olic** execution = **"concolic"**
also called **dynamic symbolic execution** (DSE)

```
1  int foo(int v) {
2    return 2*v;
3  }

5  void test_me(int x, int y) {
6    int z = foo(y);
7    if (z == x)
8      if (x > y+10)
9        ERROR;
10 }
```

| concrete run | symbolic run |
|---|---|
| `x = 22, y = 7` | $x = x_0,\ y = y_0$ |
| `z = 14` | $z = 2y_0$ |
| `14 == 22: false` ⟶ | record $2y_0 \neq x_0$ |

the concrete value picks the branch

next run takes
the true side

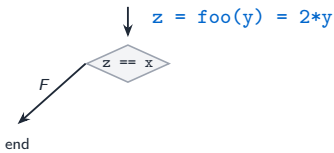
```
negate:  2y0 = x0
solver:  x = 2, y = 1
```

Pradel, Program Analysis, Stuttgart: Symbolic and Concolic Execution; Godefroid, Klarlund and Sen, PLDI 2005

# Execution Tree (1/3)

```
1  int foo(int v) {
2    return 2*v;
3  }

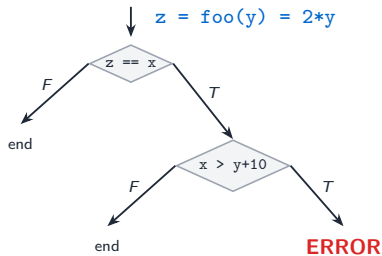
5  void test_me(int x, int y) {
6    int z = foo(y);
7    if (z == x)
8      if (x > y+10)
9        ERROR;
10 }
```



# Execution Tree (2/3)

```
1  int foo(int v) {
2    return 2*v;
3  }

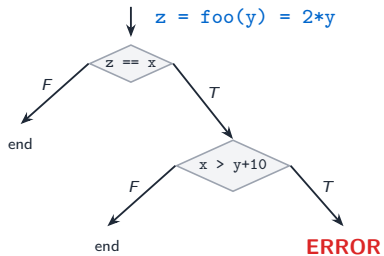
5  void test_me(int x, int y) {
6    int z = foo(y);
7    if (z == x)
8      if (x > y+10)
9        ERROR;
10 }
```



# Execution Tree (3/3)

```
1  int foo(int v) {
2    return 2*v;
3  }

5  void test_me(int x, int y) {
6    int z = foo(y);
7    if (z == x)
8      if (x > y+10)
9        ERROR;
10 }
```

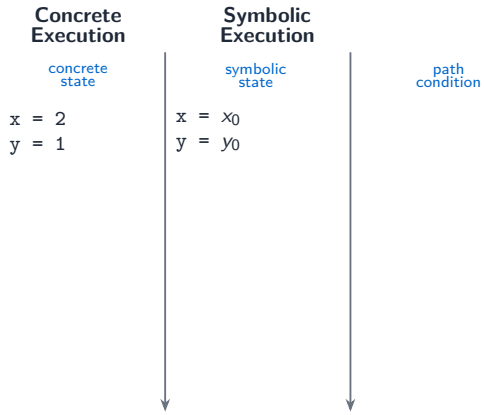



`assert(b)` $\Longrightarrow$ `if (!b) ERROR;`

# An Illustrative Example (1/9)

```
1  int foo(int v) {
2    return 2*v;
3  }

5  void test_me(int x, int y) {
6    int z = foo(y);
7    if (z == x)
8      if (x > y+10)
9        ERROR;
10 }
```

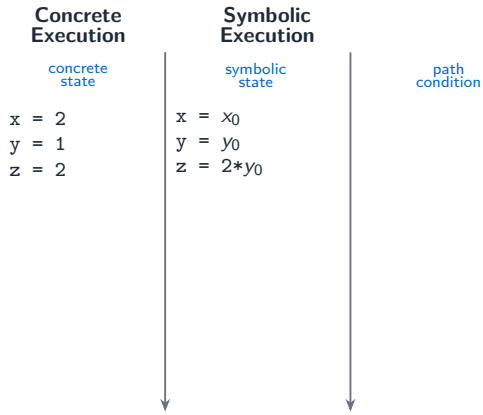



Run 2: input `x = 2, y = 1` from run 1

# An Illustrative Example (2/9)

```
1  int foo(int v) {
2    return 2*v;
3  }

5  void test_me(int x, int y) {
6    int z = foo(y);
7    if (z == x)
8      if (x > y+10)
9        ERROR;
10 }
```

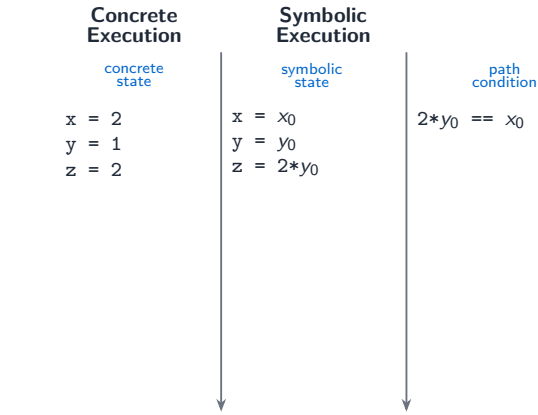



Run 2: input x = 2, y = 1 from run 1

# An Illustrative Example (3/9)

```
1  int foo(int v) {
2    return 2*v;
3  }

5  void test_me(int x, int y) {
6    int z = foo(y);
7    if (z == x)
8      if (x > y+10)
9        ERROR;
10 }
```

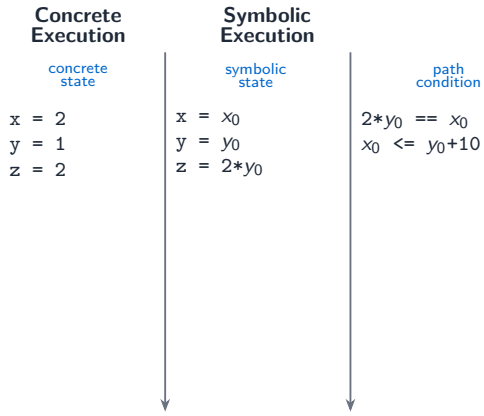



Run 2: input x = 2, y = 1 from run 1

# An Illustrative Example (4/9)

```
1  int foo(int v) {
2    return 2*v;
3  }

5  void test_me(int x, int y) {
6    int z = foo(y);
7    if (z == x)
8      if (x > y+10)
9        ERROR;
10 }
```

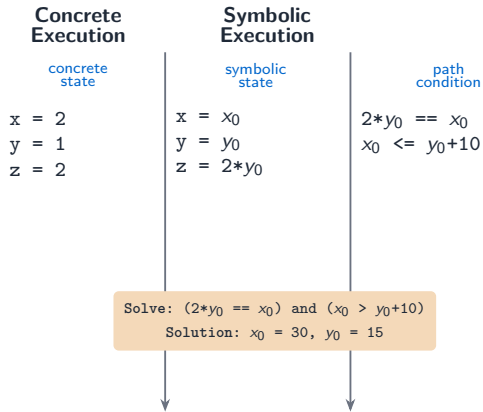



Run 2: input `x = 2, y = 1` from run 1

# An Illustrative Example (5/9)

```
1  int foo(int v) {
2    return 2*v;
3  }

5  void test_me(int x, int y) {
6    int z = foo(y);
7    if (z == x)
8      if (x > y+10)
9        ERROR;
10 }
```

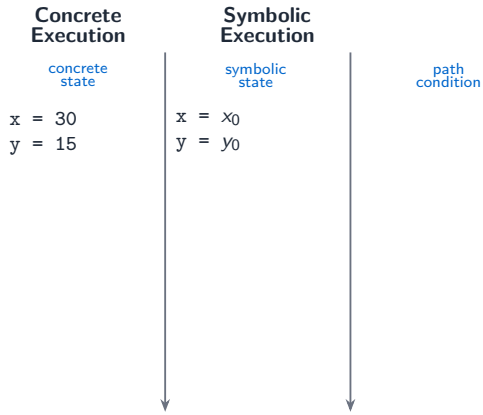



Run 2: input `x = 2, y = 1` from run 1

# An Illustrative Example (6/9)

```
1  int foo(int v) {
2    return 2*v;
3  }

5  void test_me(int x, int y) {
6    int z = foo(y);
7    if (z == x)
8      if (x > y+10)
9        ERROR;
10 }
```

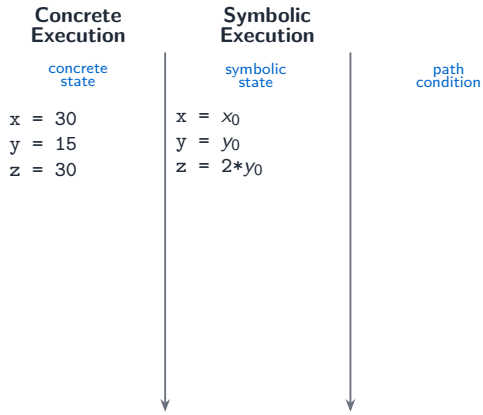



Run 3: input from run 2

# An Illustrative Example (7/9)

```
1  int foo(int v) {
2    return 2*v;
3  }

5  void test_me(int x, int y) {
6    int z = foo(y);
7    if (z == x)
8      if (x > y+10)
9        ERROR;
10 }
```

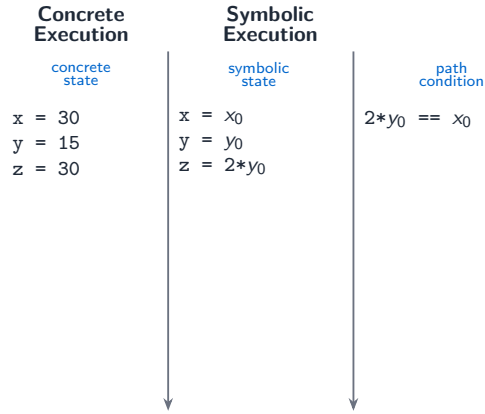



Run 3: input from run 2

# An Illustrative Example (8/9)

```
1  int foo(int v) {
2    return 2*v;
3  }

5  void test_me(int x, int y) {
6    int z = foo(y);
7    if (z == x)
8      if (x > y+10)
9        ERROR;
10 }
```



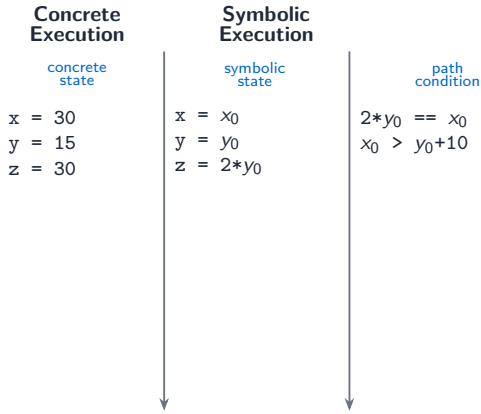

Run 3: input from run 2

# An Illustrative Example (9/9)

```
1  int foo(int v) {
2    return 2*v;
3  }

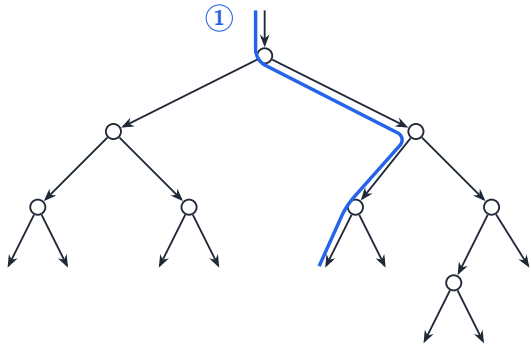
5  void test_me(int x, int y) {
6    int z = foo(y);
7    if (z == x)
8      if (x > y+10)
9        ERROR;
10 }
```

Program Error



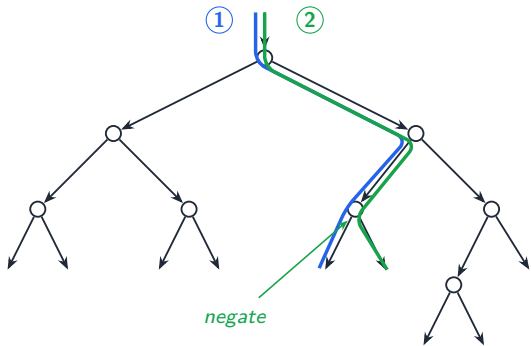

Run 3: input from run 2

# Exploring the Execution Tree (1/3)



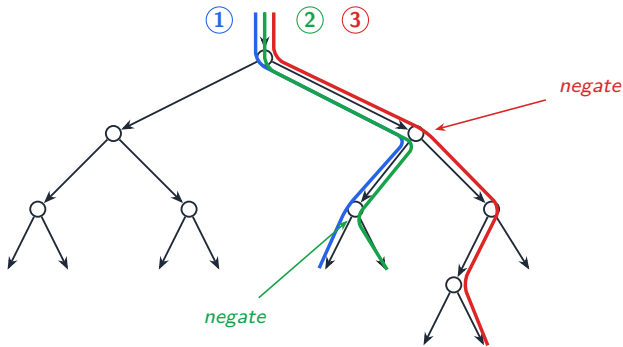

Each execution follows one path. Negating a decision at a node sends the next execution down the other side.

# Exploring the Execution Tree (2/3)




Each execution follows one path. Negating a decision at a node sends the next execution down the other side.

# Exploring the Execution Tree (3/3)

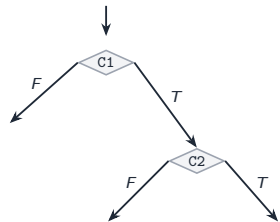



Each execution follows one path. Negating a decision at a node sends the next execution down the other side.

# Quiz: Computation Tree (1/2)

Check all constraints that DSE might possibly solve in exploring the computation tree shown below:

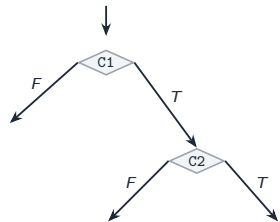
- ☐ $C_1$
- ☐ $C_2$
- ☐ $\neg C_1$
- ☐ $\neg C_2$
- ☐ $C_1 \wedge C_2$
- ☐ $C_1 \wedge \neg C_2$
- ☐ $\neg C_1 \wedge C_2$
- ☐ $\neg C_1 \wedge \neg C_2$



# Quiz: Computation Tree (2/2)

Check all constraints that DSE might possibly solve in exploring the computation tree shown below:

- ☒ $C_1$
- ☐ $C_2$
- ☒ $\neg C_1$
- ☐ $\neg C_2$
- ☒ $C_1 \wedge C_2$
- ☒ $C_1 \wedge \neg C_2$
- ☐ $\neg C_1 \wedge C_2$
- ☐ $\neg C_1 \wedge \neg C_2$



# Algorithm (1/5)

## Repeat until all paths are covered

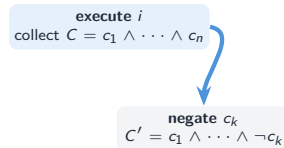
- **Execute** the program with concrete input $i$ and collect **symbolic constraints** at branch points: $C$

**execute** $i$
collect $C = c_1 \wedge \cdots \wedge c_n$

# Algorithm (2/5)

## Repeat until all paths are covered

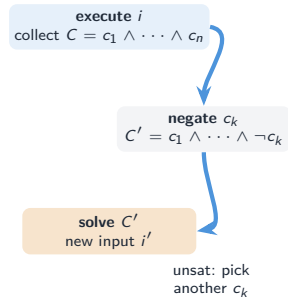
- **Execute** the program with concrete input $i$ and collect **symbolic constraints** at branch points: $C$
- **Negate one constraint** to force taking an alternative branch $b'$: constraints $C'$



# Algorithm (3/5)

## Repeat until all paths are covered

- **Execute** the program with concrete input $i$ and collect **symbolic constraints** at branch points: $C$
- **Negate one constraint** to force taking an alternative branch $b'$: constraints $C'$
- Call the constraint solver to **find a solution** for $C'$: new concrete input $i'$

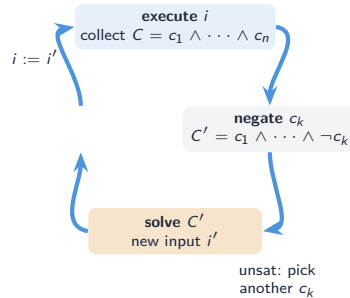



Pradel, Program Analysis, Stuttgart: Symbolic and Concolic Execution

# Algorithm (4/5)

## Repeat until all paths are covered

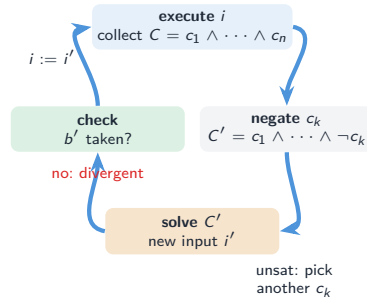
- **Execute** the program with concrete input $i$ and collect **symbolic constraints** at branch points: $C$
- **Negate one constraint** to force taking an alternative branch $b'$: constraints $C'$
- Call the constraint solver to **find a solution** for $C'$: new concrete input $i'$
- **Execute** with $i'$ to take branch $b'$



# Algorithm (5/5)

## Repeat until all paths are covered

- **Execute** the program with concrete input $i$ and collect **symbolic constraints** at branch points: $C$
- **Negate one constraint** to force taking an alternative branch $b'$: constraints $C'$
- Call the constraint solver to **find a solution** for $C'$: new concrete input $i'$
- **Execute** with $i'$ to take branch $b'$
- **Check** at runtime that $b'$ is indeed taken
  Otherwise: **"divergent execution"**

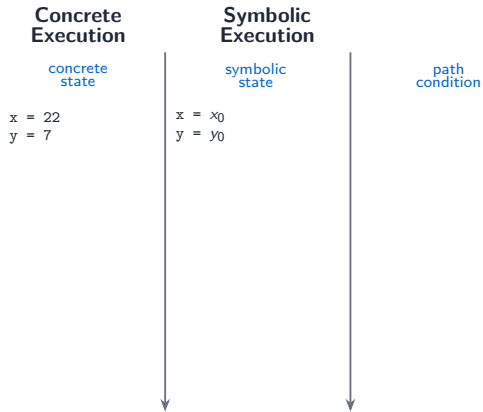



Pradel, Program Analysis, Stuttgart: Symbolic and Concolic Execution

# A More Complex Example (1/10)

```
1  int foo(int v) {
2    return secure_hash(v);
3  }

5  void test_me(int x, int y) {
6    int z = foo(y);
7    if (z == x)
8      if (x > y+10)
9        ERROR;
10 }
```

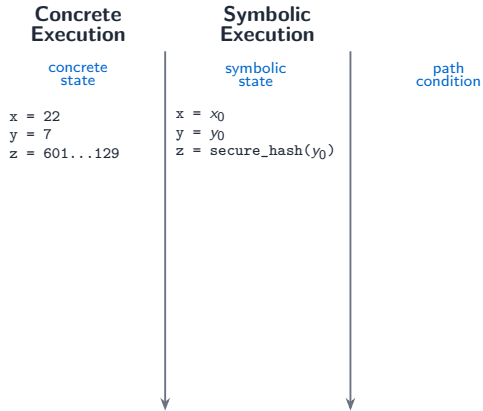



Run 1: random input

# A More Complex Example (2/10)

```
1  int foo(int v) {
2    return secure_hash(v);
3  }

5  void test_me(int x, int y) {
6    int z = foo(y);
7    if (z == x)
8      if (x > y+10)
9        ERROR;
10 }
```

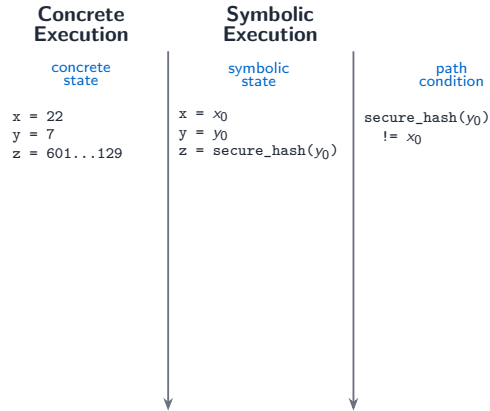



Run 1: random input

# A More Complex Example (3/10)

```
1  int foo(int v) {
2    return secure_hash(v);
3  }

5  void test_me(int x, int y) {
6    int z = foo(y);
7    if (z == x)
8      if (x > y+10)
9        ERROR;
10 }
```

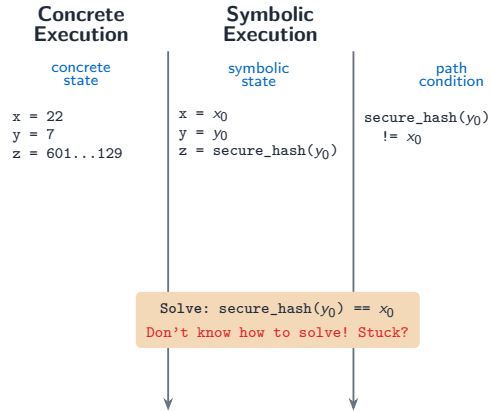



Run 1: random input

# A More Complex Example (4/10)

```
1  int foo(int v) {
2    return secure_hash(v);
3  }

5  void test_me(int x, int y) {
6    int z = foo(y);
7    if (z == x)
8      if (x > y+10)
9        ERROR;
10 }
```



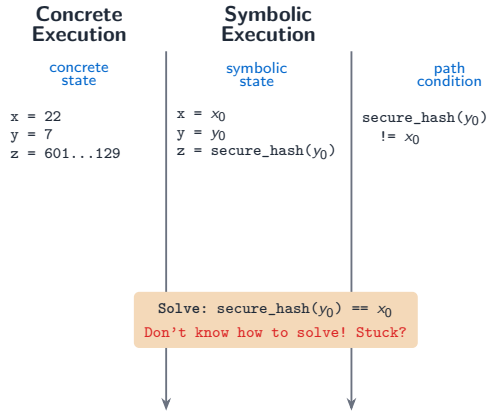

Run 1: random input

# A More Complex Example (5/10)

```
1  int foo(int v) {
2    return secure_hash(v);
3  }

5  void test_me(int x, int y) {
6    int z = foo(y);
7    if (z == x)
8      if (x > y+10)
9        ERROR;
10 }
```

Not stuck! Use concrete state: replace $y_0$ by 7

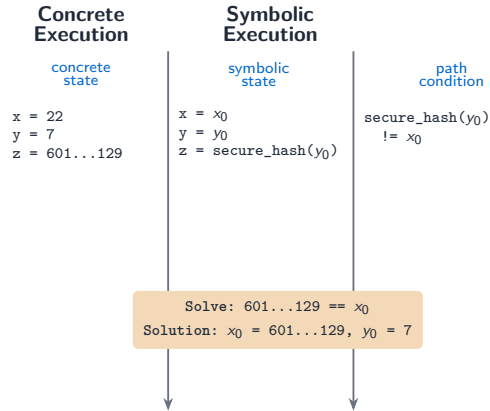



Run 1: random input

# A More Complex Example (6/10)

```
1  int foo(int v) {
2    return secure_hash(v);
3  }

5  void test_me(int x, int y) {
6    int z = foo(y);
7    if (z == x)
8      if (x > y+10)
9        ERROR;
10 }
```

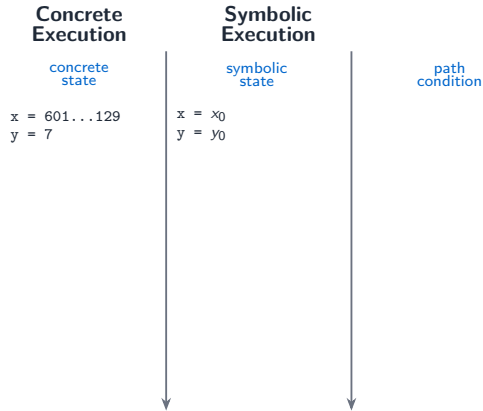



Run 1: random input

# A More Complex Example (7/10)

```
1  int foo(int v) {
2    return secure_hash(v);
3  }

5  void test_me(int x, int y) {
6    int z = foo(y);
7    if (z == x)
8      if (x > y+10)
9        ERROR;
10 }
```

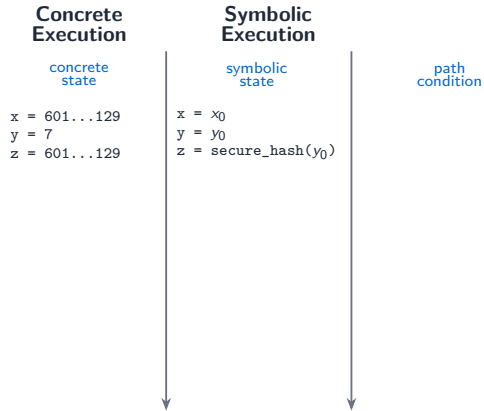



Run 2: input from run 1

# A More Complex Example (8/10)

```
1  int foo(int v) {
2    return secure_hash(v);
3  }

5  void test_me(int x, int y) {
6    int z = foo(y);
7    if (z == x)
8      if (x > y+10)
9        ERROR;
10 }
```

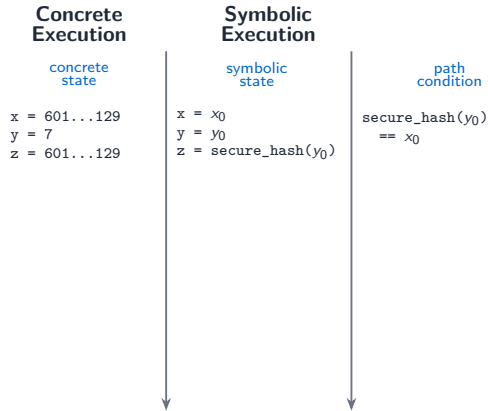



Run 2: input from run 1

# A More Complex Example (9/10)

```
1  int foo(int v) {
2    return secure_hash(v);
3  }

5  void test_me(int x, int y) {
6    int z = foo(y);
7    if (z == x)
8      if (x > y+10)   <-
9        ERROR;
10 }
```



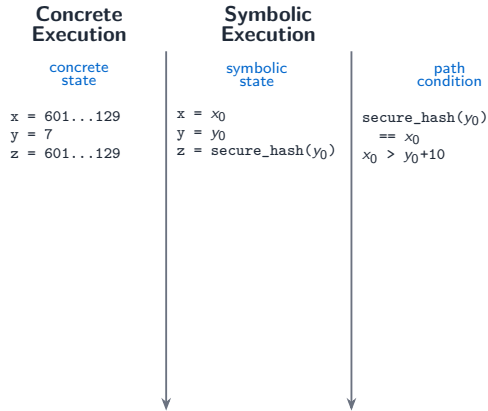

Run 2: input from run 1

# A More Complex Example (10/10)

```
1  int foo(int v) {
2    return secure_hash(v);
3  }

5  void test_me(int x, int y) {
6    int z = foo(y);
7    if (z == x)
8      if (x > y+10)
9        ERROR;
10 }
```



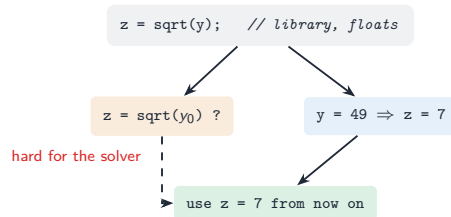




Run 2: input from run 1

# Benefits of the Concolic Approach

When symbolic reasoning is impossible or impractical,
**fall back to concrete values**

- Native, system and API functions
- Operations hard for the solver
  (e.g. floating point)




⇒ the solver becomes an **incomplete** theorem prover: never "sat" for an unsat constraint, but it may miss solutions

Pradel, Program Analysis, Stuttgart: Symbolic and Concolic Execution; Naik, CIS 5470, Penn: Dynamic Symbolic Execution

# Quiz: Example Application (1/6)

DSE tests the program, starting with `x = 1`, depth-first. For each run: which path is recorded, and which constraint is solved next?

| Run | x | path recorded | solved next |
|---|---|---|---|
| 1 | 1 | $5 \neq x_0 \wedge 7 \neq x_0 \wedge 9 \neq x_0$ | |
| 2 | | | |
| 3 | | | |
| 4 | | | |

```
1 int test_me(int x) {
2   int[] A = { 5, 7, 9 };
3   int i = 0;
4   while (i < 3) {
5     if (A[i] == x) break;
6     i++;
7   }
8   return i;
9 }
```

# Quiz: Example Application (2/6)

DSE tests the program, starting with `x = 1`, depth-first. For each run: which path is recorded, and which constraint is solved next?

| Run | x | path recorded | solved next |
|---|---|---|---|
| 1 | 1 | $5 \neq x_0 \wedge 7 \neq x_0 \wedge 9 \neq x_0$ | $5 \neq x_0 \wedge 7 \neq x_0 \wedge 9 = x_0$ ← negated |
| 2 | | | |
| 3 | | | |
| 4 | | | |

```
1 int test_me(int x) {
2   int[] A = { 5, 7, 9 };
3   int i = 0;
4   while (i < 3) {
5     if (A[i] == x) break;
6     i++;
7   }
8   return i;
9 }
```

# Quiz: Example Application (3/6)

DSE tests the program, starting with `x = 1`, depth-first. For each run: which path is recorded, and which constraint is solved next?

| Run | x | path recorded | solved next |
|---|---|---|---|
| 1 | 1 | $5 \neq x_0 \wedge 7 \neq x_0 \wedge 9 \neq x_0$ | $5 \neq x_0 \wedge 7 \neq x_0 \wedge 9 = x_0$ ← negated |
| 2 | 9 | $5 \neq x_0 \wedge 7 \neq x_0 \wedge 9 = x_0$ | |
| 3 | | | |
| 4 | | | |

```
1 int test_me(int x) {
2   int[] A = { 5, 7, 9 };
3   int i = 0;
4   while (i < 3) {
5     if (A[i] == x) break;
6     i++;
7   }
8   return i;
9 }
```

# Quiz: Example Application (4/6)

DSE tests the program, starting with `x = 1`, depth-first. For each run: which path is recorded, and which constraint is solved next?

| Run | x | path recorded | solved next |
|---|---|---|---|
| 1 | 1 | $5 \neq x_0 \wedge 7 \neq x_0 \wedge 9 \neq x_0$ | $5 \neq x_0 \wedge 7 \neq x_0 \wedge 9 = x_0$ ← negated |
| 2 | 9 | $5 \neq x_0 \wedge 7 \neq x_0 \wedge 9 = x_0$ | $5 \neq x_0 \wedge 7 = x_0$ ← |
| 3 | | | |
| 4 | | | |

```
1 int test_me(int x) {
2   int[] A = { 5, 7, 9 };
3   int i = 0;
4   while (i < 3) {
5     if (A[i] == x) break;
6     i++;
7   }
8   return i;
9 }
```

# Quiz: Example Application (5/6)

DSE tests the program, starting with `x = 1`, depth-first. For each run: which path is recorded, and which constraint is solved next?

| Run | x | path recorded | solved next |
|---|---|---|---|
| 1 | 1 | $5 \neq x_0 \wedge 7 \neq x_0 \wedge 9 \neq x_0$ | $5 \neq x_0 \wedge 7 \neq x_0 \wedge 9 = x_0$ ← negated |
| 2 | 9 | $5 \neq x_0 \wedge 7 \neq x_0 \wedge 9 = x_0$ | $5 \neq x_0 \wedge 7 = x_0$ ← |
| 3 | 7 | $5 \neq x_0 \wedge 7 = x_0$ | $5 = x_0$ ← |
| 4 | | | |

```
1 int test_me(int x) {
2   int[] A = { 5, 7, 9 };
3   int i = 0;
4   while (i < 3) {
5     if (A[i] == x) break;
6     i++;
7   }
8   return i;
9 }
```

# Quiz: Example Application (6/6)

DSE tests the program, starting with `x = 1`, depth-first. For each run: which path is recorded, and which constraint is solved next?

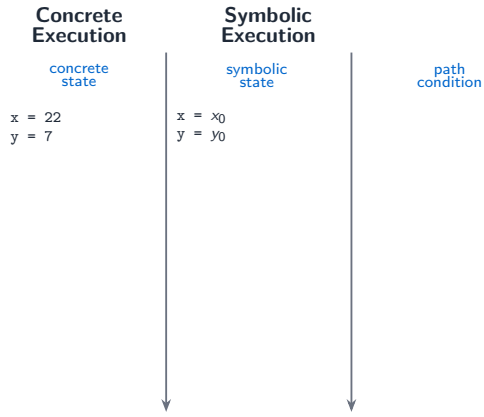
| Run | x | path recorded | solved next |
|---|---|---|---|
| 1 | 1 | $5 \neq x_0 \wedge 7 \neq x_0 \wedge 9 \neq x_0$ | $5 \neq x_0 \wedge 7 \neq x_0 \wedge 9 = x_0$ ← negated |
| 2 | 9 | $5 \neq x_0 \wedge 7 \neq x_0 \wedge 9 = x_0$ | $5 \neq x_0 \wedge 7 = x_0$ ← |
| 3 | 7 | $5 \neq x_0 \wedge 7 = x_0$ | $5 = x_0$ ← |
| 4 | 5 | $5 = x_0$ | nothing left: stop |

```
1 int test_me(int x) {
2   int[] A = { 5, 7, 9 };
3   int i = 0;
4   while (i < 3) {
5     if (A[i] == x) break;
6     i++;
7   }
8   return i;
9 }
```

# A Third Example (1/5)

```
1 int foo(int v) {
2   return secure_hash(v);
3 }

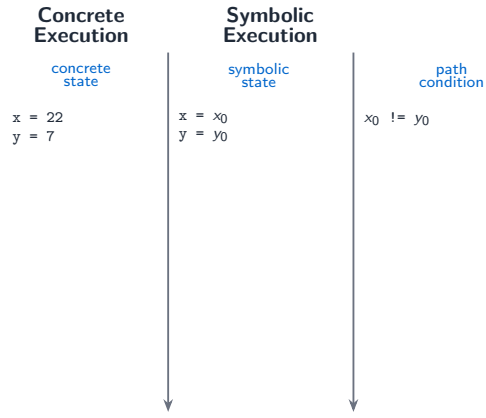
5 void test_me(int x, int y) {
6   if (x != y)
7     if (foo(x) == foo(y))
8       ERROR;
9 }
```



# A Third Example (2/5)

```
1 int foo(int v) {
2   return secure_hash(v);
3 }

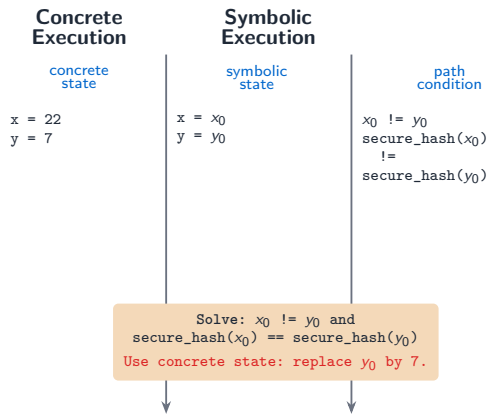
5 void test_me(int x, int y) {
6   if (x != y)
7     if (foo(x) == foo(y))
8       ERROR;
9 }
```



# A Third Example (3/5)

```
1 int foo(int v) {
2   return secure_hash(v);
3 }

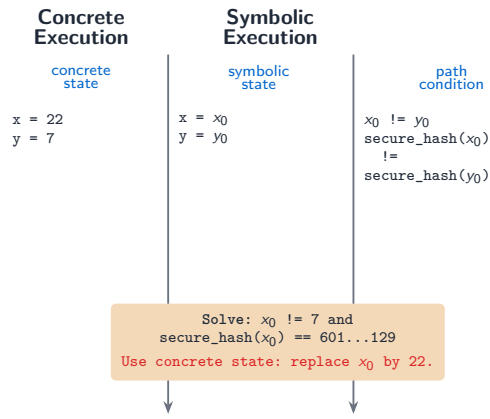
5 void test_me(int x, int y) {
6   if (x != y)
7     if (foo(x) == foo(y))
8       ERROR;
9 }
```



# A Third Example (4/5)

```
1 int foo(int v) {
2   return secure_hash(v);
3 }

5 void test_me(int x, int y) {
6   if (x != y)
7     if (foo(x) == foo(y))
8       ERROR;
9 }
```

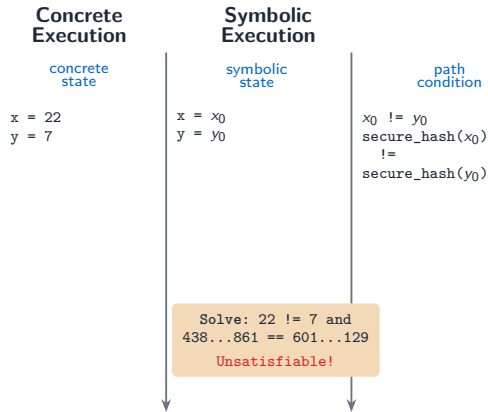


# A Third Example (5/5)

```
1 int foo(int v) {
2   return secure_hash(v);
3 }

5 void test_me(int x, int y) {
6   if (x != y)
7     if (foo(x) == foo(y))
8       ERROR;
9 }
```

False negative!



# Divergent Execution: Example (1/6)

```
1 void flaky(int a) {
2   if (std::time(nullptr) % 2 == 0) {
3     if (a > 1) {
4       // ...
5     }
6   }
7 }
```

**Execution 1**

input `a = 0`

# Divergent Execution: Example (2/6)

```
1 void flaky(int a) {
2   if (std::time(nullptr) % 2 == 0) {
3     if (a > 1) {
4       // ...
5     }
6   }
7 }
```

**Execution 1**

| | |
|---|---|
| input | `a = 0` |
| line 2 | even second: **true** |

# Divergent Execution: Example (3/6)

```
1 void flaky(int a) {
2   if (std::time(nullptr) % 2 == 0) {
3     if (a > 1) {
4       // ...
5     }
6   }
7 }
```

**Execution 1**

| | |
|---|---|
| input | `a = 0` |
| line 2 | even second: **true** |
| line 3 | 0 > 1: **false** |

# Divergent Execution: Example (4/6)

```
1 void flaky(int a) {
2   if (std::time(nullptr) % 2 == 0) {
3     if (a > 1) {
4       // ...
5     }
6   }
7 }
```

**Execution 1**

| | |
|---|---|
| input | a = 0 |
| line 2 | even second: **true** |
| line 3 | 0 > 1: **false** |
| path constraint | $a_0 \leq 1$ |

# Divergent Execution: Example (5/6)

```
1 void flaky(int a) {
2   if (std::time(nullptr) % 2 == 0) {
3     if (a > 1) {
4       // ...
5     }
6   }
7 }
```

**Execution 1**

| | |
|---|---|
| input | a = 0 |
| line 2 | even second: **true** |
| line 3 | 0 > 1: **false** |
| path constraint | $a_0 \leq 1$ |
| negate and solve | $a_0 > 1$: $a_0 = 2$ |

# Divergent Execution: Example (6/6)

```
1 void flaky(int a) {
2   if (std::time(nullptr) % 2 == 0) {
3     if (a > 1) {
4       // ...
5     }
6   }
7 }
```

**Execution 1**

| | |
|---|---|
| input | a = 0 |
| line 2 | even second: **true** |
| line 3 | 0 > 1: **false** |
| path constraint | $a_0 \leq 1$ |
| negate and solve | $a_0 > 1$: $a_0 = 2$ |

**Execution 2**

input a = 2
line 2 odd second: **false**
line 3 is never reached:
**divergent execution**

Pradel, Program Analysis, Stuttgart: Symbolic and Concolic Execution

# Quiz: Properties of DSE (1/2)

Assume that programs can have infinite computation trees.
Which statements are true of DSE applied to such programs?

- ☐ DSE is guaranteed to terminate.
- ☐ DSE is **complete**: if it ever reaches an error, the program can reach that error in some execution.
- ☐ DSE is **sound**: if it terminates and did not reach an error, the program cannot reach an error in any execution.

# Quiz: Properties of DSE (2/2)

Assume that programs can have infinite computation trees.
Which statements are true of DSE applied to such programs?

- ☐ DSE is guaranteed to terminate.
- ☒ DSE is **complete**: if it ever reaches an error, the program can reach that error in some execution.
- ☐ DSE is **sound**: if it terminates and did not reach an error, the program cannot reach an error in any execution.

# Another Example: Testing Data Structures (1/2)

- Random test driver:
  - random value for x
  - random memory graph reachable from p

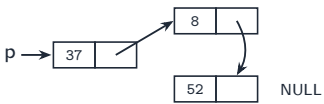
```
1  typedef struct cell {
2    int data;
3    struct cell *next;
4  } cell;

6  int foo(int v) { return 2*v + 1; }

8  int test_me(int x, cell *p) {
9    if (x > 0)
10     if (p != NULL)
11       if (foo(x) == p->data)
12         if (p->next == p)
13           ERROR;
14   return 0;
15 }
```

# Another Example: Testing Data Structures (2/2)

- Random test driver:
  - random value for x
  - random memory graph reachable from p
- Probability of reaching ERROR is extremely low




a random graph: 2*x+1 == p->data
and p->next == p almost never hold

```
1  typedef struct cell {
2    int data;
3    struct cell *next;
4  } cell;

6  int foo(int v) { return 2*v + 1; }

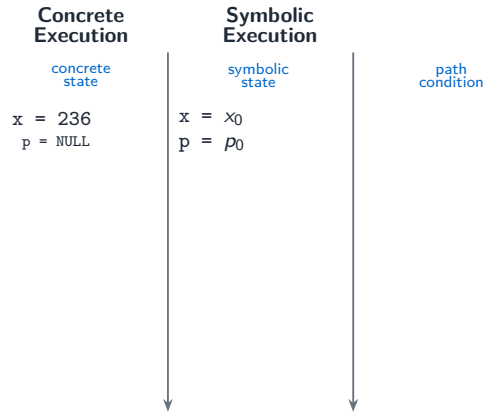
8  int test_me(int x, cell *p) {
9    if (x > 0)
10     if (p != NULL)
11       if (foo(x) == p->data)
12         if (p->next == p)
13           ERROR;
14   return 0;
15 }
```

# Data-Structure Example (1/12)

```
1  typedef struct cell {
2    int data;
3    struct cell *next;
4  } cell;

6  int foo(int v) { return 2*v + 1; }

8  int test_me(int x, cell *p) {
9    if (x > 0)
10     if (p != NULL)
11       if (foo(x) == p->data)
12         if (p->next == p)
13           ERROR;
14   return 0;
15 }
```



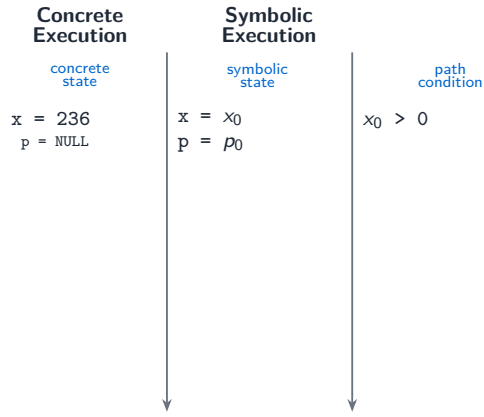

Run 1: random input

# Data-Structure Example (2/12)

```
1  typedef struct cell {
2    int data;
3    struct cell *next;
4  } cell;

6  int foo(int v) { return 2*v + 1; }

8  int test_me(int x, cell *p) {
9    if (x > 0)
10     if (p != NULL)
11       if (foo(x) == p->data)
12         if (p->next == p)
13           ERROR;
14   return 0;
15 }
```




Run 1: random input

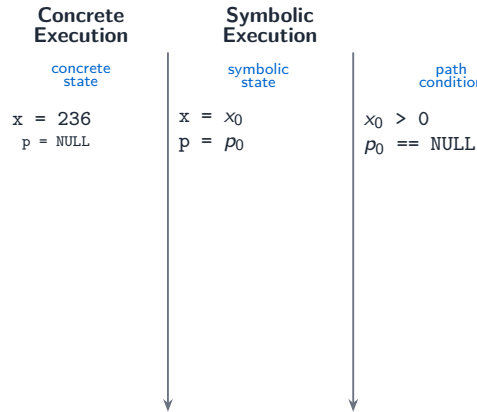
# Data-Structure Example (3/12)

```
1  typedef struct cell {
2    int data;
3    struct cell *next;
4  } cell;

6  int foo(int v) { return 2*v + 1; }

8  int test_me(int x, cell *p) {
9    if (x > 0)
10     if (p != NULL)
11       if (foo(x) == p->data)
12         if (p->next == p)
13           ERROR;
14   return 0;
15 }
```

Run 1: random input



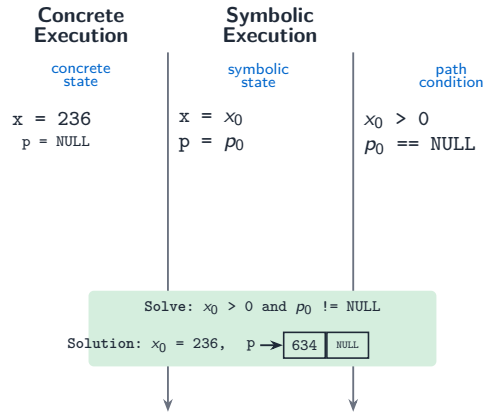
# Data-Structure Example (4/12)

```
1  typedef struct cell {
2    int data;
3    struct cell *next;
4  } cell;

6  int foo(int v) { return 2*v + 1; }

8  int test_me(int x, cell *p) {
9    if (x > 0)
10     if (p != NULL)
11       if (foo(x) == p->data)
12         if (p->next == p)
13           ERROR;
14   return 0;
15 }
```

Run 1: random input

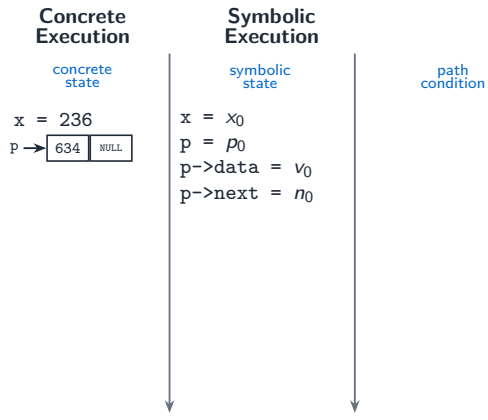


# Data-Structure Example (5/12)

```
1  typedef struct cell {
2    int data;
3    struct cell *next;
4  } cell;

6  int foo(int v) { return 2*v + 1; }

8  int test_me(int x, cell *p) {
9    if (x > 0)
10     if (p != NULL)
11       if (foo(x) == p->data)
12         if (p->next == p)
13           ERROR;
14   return 0;
15 }
```



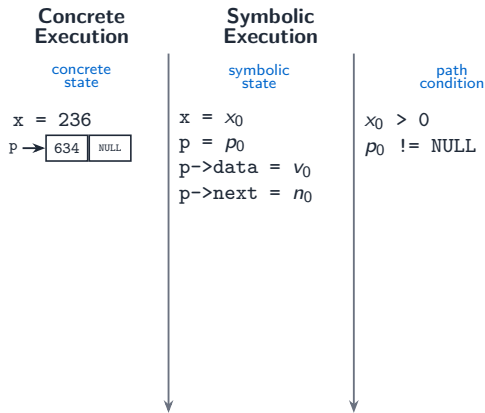

Run 2

# Data-Structure Example (6/12)

```
1  typedef struct cell {
2    int data;
3    struct cell *next;
4  } cell;

6  int foo(int v) { return 2*v + 1; }

8  int test_me(int x, cell *p) {
9    if (x > 0)
10     if (p != NULL)
11       if (foo(x) == p->data)
12         if (p->next == p)
13           ERROR;
14   return 0;
15 }
```




Run 2

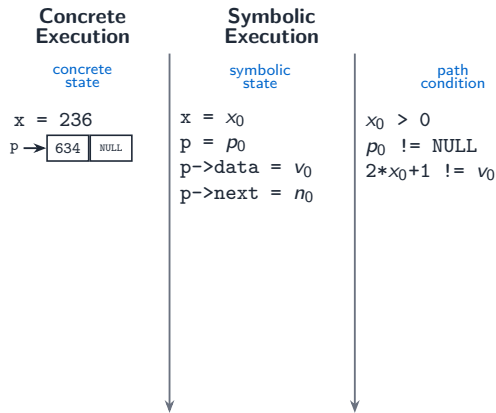
# Data-Structure Example (7/12)

```
1  typedef struct cell {
2    int data;
3    struct cell *next;
4  } cell;

6  int foo(int v) { return 2*v + 1; }

8  int test_me(int x, cell *p) {
9    if (x > 0)
10     if (p != NULL)
11       if (foo(x) == p->data)
12         if (p->next == p)
13           ERROR;
14   return 0;
15 }
```

Run 2

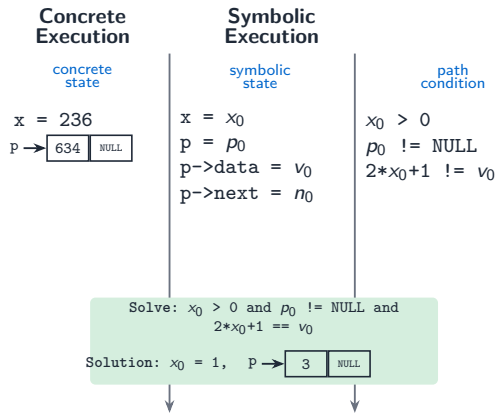


# Data-Structure Example (8/12)

```
1  typedef struct cell {
2    int data;
3    struct cell *next;
4  } cell;

6  int foo(int v) { return 2*v + 1; }

8  int test_me(int x, cell *p) {
9    if (x > 0)
10     if (p != NULL)
11       if (foo(x) == p->data)
12         if (p->next == p)
13           ERROR;
14   return 0;
15 }
```



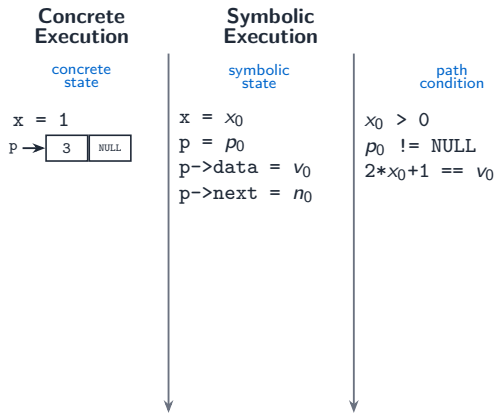

Run 2

# Data-Structure Example (9/12)

```
1  typedef struct cell {
2    int data;
3    struct cell *next;
4  } cell;

6  int foo(int v) { return 2*v + 1; }

8  int test_me(int x, cell *p) {
9    if (x > 0)
10     if (p != NULL)
11       if (foo(x) == p->data)
12         if (p->next == p)
13           ERROR;
14   return 0;
15 }
```




Run 3

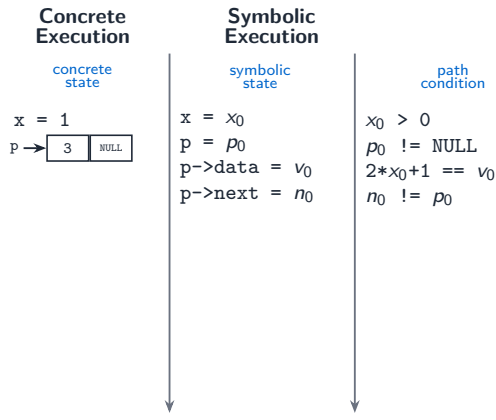
# Data-Structure Example (10/12)

```
1  typedef struct cell {
2    int data;
3    struct cell *next;
4  } cell;

6  int foo(int v) { return 2*v + 1; }

8  int test_me(int x, cell *p) {
9    if (x > 0)
10     if (p != NULL)
11       if (foo(x) == p->data)
12         if (p->next == p)
13           ERROR;
14   return 0;
15 }
```

Run 3

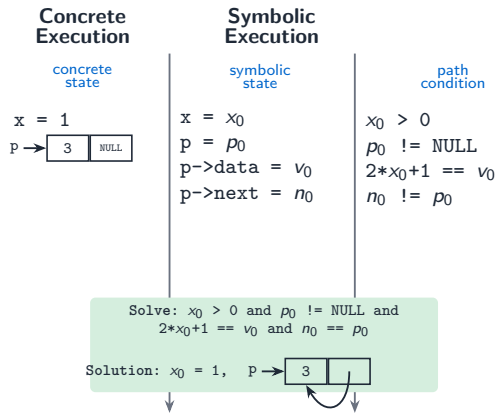


# Data-Structure Example (11/12)

```
1  typedef struct cell {
2    int data;
3    struct cell *next;
4  } cell;

6  int foo(int v) { return 2*v + 1; }

8  int test_me(int x, cell *p) {
9    if (x > 0)
10     if (p != NULL)
11       if (foo(x) == p->data)
12         if (p->next == p)
13           ERROR;
14   return 0;
15 }
```




Run 3

# Data-Structure Example (12/12)

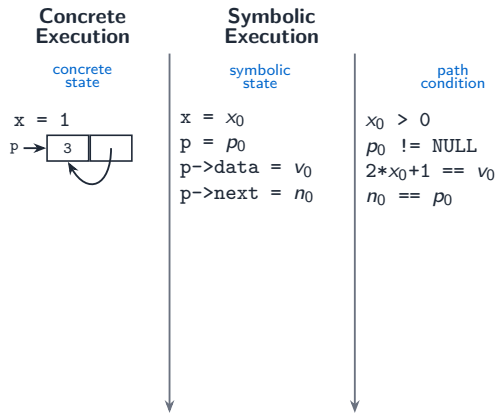
```
1  typedef struct cell {
2    int data;
3    struct cell *next;
4  } cell;

6  int foo(int v) { return 2*v + 1; }

8  int test_me(int x, cell *p) {
9    if (x > 0)
10     if (p != NULL)
11       if (foo(x) == p->data)
12         if (p->next == p)
13           ERROR;
14   return 0;
15 }
```

Program Error

Run 4



# Quiz: Characteristics of DSE (1/2)

- The testing approach of DSE is:
  ☐ automated, black-box ☐ automated, white-box ☐ manual, black-box ☐ manual, white-box
- The input search of DSE is:
  ☐ randomized ☐ systematic
- The static analysis of DSE is:
  ☐ flow-insensitive ☐ flow-sensitive ☐ path-sensitive
- The instrumentation in DSE is:
  ☐ sampled ☐ non-sampled

**White-box**: uses the code. **Sampled instrumentation**: records only some of the events, to save time.

# Quiz: Characteristics of DSE (2/2)

- The testing approach of DSE is:
  ☐ automated, black-box ☒ automated, white-box ☐ manual, black-box ☐ manual, white-box
- The input search of DSE is:
  ☐ randomized ☒ systematic
- The static analysis of DSE is:
  ☐ flow-insensitive ☐ flow-sensitive ☒ path-sensitive
- The instrumentation in DSE is:
  ☐ sampled ☒ non-sampled

**White-box**: uses the code. **Sampled instrumentation**: records only some of the events, to save time.

# Case Study: SGLIB C Library (1/2)

- Found **two bugs** in sglib 1.0.1
  reported to authors, fixed in 1.0.2
- **Bug 1: doubly-linked list**
  segmentation fault when a non-zero length list is concatenated with a zero-length list; found in 140 iterations ($<$ 1 second)
- **Bug 2: hash table**
  an infinite loop in `is_member`; found in 193 iterations (1 second)

Sen, Marinov and Agha, FSE 2005 (CUTE); Naik, CIS 5470, Penn: Dynamic Symbolic Execution

# Case Study: SGLIB C Library (2/2)

- Found **two bugs** in sglib 1.0.1
  reported to authors, fixed in 1.0.2
- **Bug 1: doubly-linked list**
  segmentation fault when a non-zero length list is concatenated with a zero-length list; found in 140 iterations ($<$ 1 second)
- **Bug 2: hash table**
  an infinite loop in `is_member`; found in 193 iterations (1 second)

| Name | time (s) | iterations | branches | % branch cov. | functions | bugs |
|---|---|---|---|---|---|---|
| Array Quick Sort | 2 | 732 | 43 | 97.73 | 2 | 0 |
| Array Heap Sort | 4 | 1764 | 36 | 100.00 | 2 | 0 |
| Linked List | 2 | 570 | 100 | 96.15 | 12 | 0 |
| Sorted List | 2 | 1020 | 110 | 96.49 | 11 | 0 |
| Doubly Linked List | 3 | 1317 | 224 | 99.12 | 17 | 1 |
| Hash Table | 1 | 193 | 46 | 85.19 | 8 | 1 |
| Red Black Tree | 2629 | 1,000,000 | 242 | 71.18 | 17 | 0 |

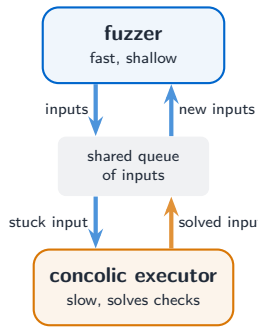
Sen, Marinov and Agha, FSE 2005 (CUTE); Naik, CIS 5470, Penn: Dynamic Symbolic Execution

# Hybrid Fuzzing

**Hybrid fuzzing**: a fast *fuzzer* explores; when it is stuck, concolic execution solves the blocking branch

fuzzer: **fast**, but **stuck** at narrow checks such as `x == 94389`

concolic: **slow**, but **solves** such checks




Stephens et al., NDSS 2016; Yun et al., USENIX Security 2018

# Summary: Concolic Execution (1/5)

- Run a **concrete** and a **symbolic** execution side by side

Pradel, Program Analysis, Stuttgart: Symbolic and Concolic Execution; Naik, CIS 5470, Penn: Dynamic Symbolic Execution

# Summary: Concolic Execution (2/5)

- Run a **concrete** and a **symbolic** execution side by side
- After each run, **negate one decision** and solve: the next input takes a new path

Pradel, Program Analysis, Stuttgart: Symbolic and Concolic Execution; Naik, CIS 5470, Penn: Dynamic Symbolic Execution

# Summary: Concolic Execution (3/5)

- Run a **concrete** and a **symbolic** execution side by side
- After each run, **negate one decision** and solve: the next input takes a new path
- When a constraint is too hard, **use the concrete value**: environment, hashes, floating point

Pradel, Program Analysis, Stuttgart: Symbolic and Concolic Execution; Naik, CIS 5470, Penn: Dynamic Symbolic Execution

# Summary: Concolic Execution (4/5)

- Run a **concrete** and a **symbolic** execution side by side
- After each run, **negate one decision** and solve: the next input takes a new path
- When a constraint is too hard, **use the concrete value**: environment, hashes, floating point
- Every reported error is real, but errors can be **missed**; runs can **diverge**

Pradel, Program Analysis, Stuttgart: Symbolic and Concolic Execution; Naik, CIS 5470, Penn: Dynamic Symbolic Execution

# Summary: Concolic Execution (5/5)

- Run a **concrete** and a **symbolic** execution side by side
- After each run, **negate one decision** and solve: the next input takes a new path
- When a constraint is too hard, **use the concrete value**: environment, hashes, floating point
- Every reported error is real, but errors can be **missed**; runs can **diverge**
- **Hybrid fuzzing**: a fast fuzzer explores, concolic execution solves the hard branches

Pradel, Program Analysis, Stuttgart: Symbolic and Concolic Execution; Naik, CIS 5470, Penn: Dynamic Symbolic Execution