Dynamic Symbolic Execution

CS 6340

Motivation

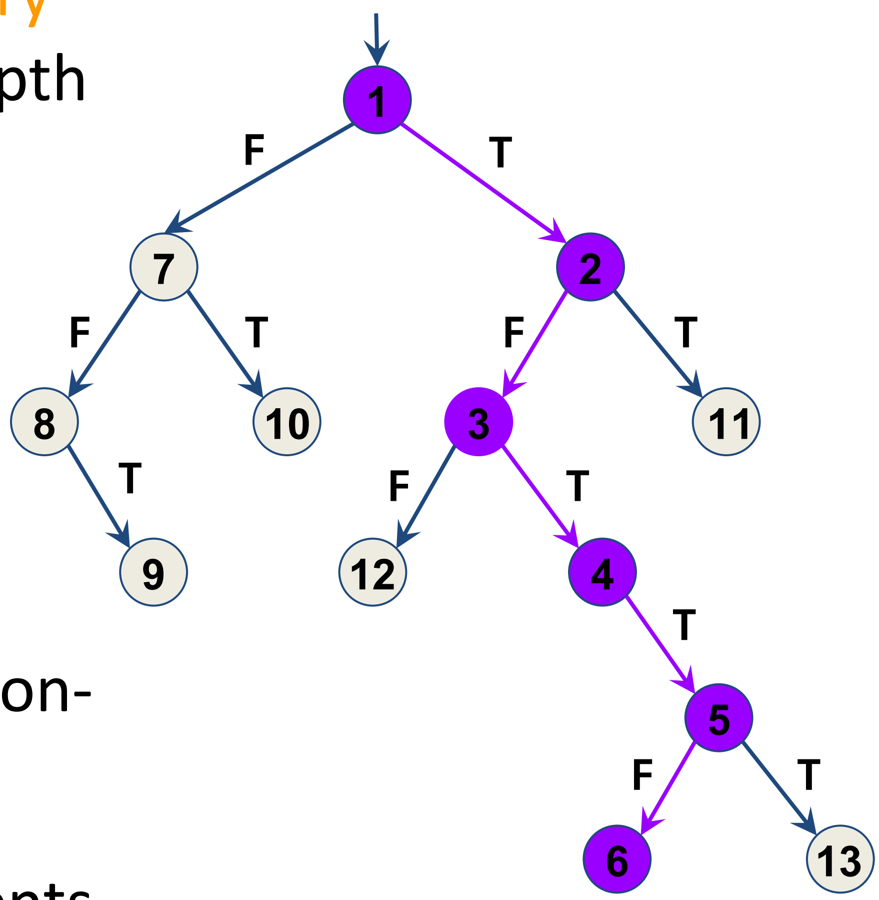
- Writing and maintaining tests is tedious and error-prone
- Idea: Automated Test Generation
  - Generate regression test suite
  - Execute all reachable statements
  - Catch any assertion violations

Approach

- Dynamic Symbolic Execution
  - Stores program state concretely and symbolically
  - Solves constraints to guide execution at branch points
  - Explores all execution paths of the unit tested
- Example of Hybrid Analysis
  - Collaboratively combines dynamic and static analysis

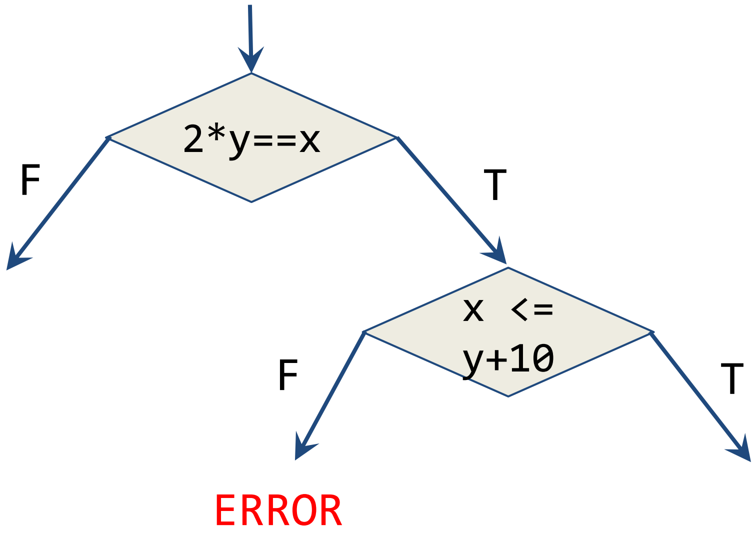
Execution Paths of a Program

- Program can be seen as binary tree with possibly infinite depth
  - Called Computation Tree
- Each node represents the execution of a conditional statement
- Each edge represents the execution of a sequence of non-conditional statements
- Each path in the tree represents an equivalence class of inputs



Example of Computation Tree

```
void test_me(int x, int y) {
   if (2*y == x) {
      if (x <= y+10)
         print("OK");
      else {
         print("something bad");
         ERROR;
      }
   } else
      print("OK");
}
```




```
assert(b) ⇒ if (!b) ERROR;
```

Existing Approach I

Random Testing

- Generate random inputs
- Execute the program on those (concrete) inputs

```
void test_me(int x) {
    if (x == 94389) {
        ERROR;
    }
}
```

Problem:

- Probability of reaching error could be astronomically small

Probability of ERROR:

$1/2^{32} \approx 0.00000023\%$

Existing Approach II

Symbolic Execution

- Use symbolic values for inputs
- Execute program symbolically on symbolic input values
- Collect symbolic path constraints
- Use theorem prover to check if a branch can be taken

```
void test_me(int x) {
    if (x*3 == 15) {
        if (x % 5 == 0)
            print("OK");
        else {
            print("something bad");
            ERROR;
        }
    } else
        print("OK");
}
```

Problem:

- Does not scale for large programs

Existing Approach II

Symbolic Execution

- Use symbolic values for inputs
- Execute program symbolically on symbolic input values
- Collect symbolic path constraints
- Use theorem prover to check if a branch can be taken

```
void test_me(int x) {
   // c = product of two
   // large primes
   if (pow(2,x) % c == 17) {
      print("something bad");
      ERROR;
   } else
      print("OK");
}
```

Symbolic execution will say both branches are reachable: False Positive

Problem:

- Does not scale for large programs

Combined Approach

Dynamic Symbolic Execution (DSE)

- Start with random input values
- Keep track of both concrete values and symbolic constraints
- Use concrete values to simplify symbolic constraints
- Incomplete theorem-prover

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

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

An Illustrative Example

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

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

Concrete Execution		Symbolic Execution	
concrete state		symbolic state	path condition
x = 22		$x = x_0$	
y = 7		$y = y_0$	

An Illustrative Example

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

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

Concrete Execution

Symbolic Execution

concrete state	symbolic state	path condition
x = 22	$x = x_0$	
y = 7	$y = y_0$	
z = 14	$z = 2*y_0$	

An Illustrative Example

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

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

Concrete Execution		Symbolic Execution	
concrete state	symbolic state	path condition
x = 22	$x = x_0$	$2*y_0 \; != \; x_0$
y = 7	$y = y_0$	
z = 14	$z = 2*y_0$	

An Illustrative Example

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

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

Concrete Execution		Symbolic Execution	
concrete state	symbolic state	path condition	
x = 22	$x = x_0$	$2*y_0 \mathrel{!=} x_0$	
y = 7	$y = y_0$		
z = 14	$z = 2*y_0$		

Solve: $2*y_0 == x_0$

Solution: $x_0 = 2$, $y_0 = 1$

An Illustrative Example

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

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

concrete state	symbolic state	path condition
x = 2	$x = x_0$	
y = 1	$y = y_0$	

Concrete Execution

Symbolic Execution

An Illustrative Example

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

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

Concrete Execution		Symbolic Execution	
concrete state	symbolic state		path condition
x = 2	$x = x_0$		
y = 1	$y = y_0$		
z = 2	$z = 2*y_0$		

An Illustrative Example

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

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

concrete state	symbolic state	path condition
x = 2	$x = x_0$	$2*y_0 == x_0$
y = 1	$y = y_0$	
z = 2	$z = 2*y_0$	

Concrete Execution — Symbolic Execution

An Illustrative Example

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

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

Concrete Execution

Symbolic Execution

concrete state	symbolic state	path condition
x = 2	x = x_0	$2*y_0 == x_0$
y = 1	y = y_0	$x_0 <= y_0+10$
z = 2	z = $2*y_0$	

An Illustrative Example

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

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

	Concrete Execution	Symbolic Execution	
	concrete state	symbolic state	path condition
	$x = 2$	$x = x_0$	$2*y_0 == x_0$
	$y = 1$	$y = y_0$	
	$z = 2$	$z = 2*y_0$	$x_0 <= y_0+10$

Solve: $(2*y_0 == x_0)$ and $(x_0 > y_0+10)$

Solution: $x_0 = 30$, $y_0 = 15$

An Illustrative Example

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

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

Concrete Execution		Symbolic Execution	
concrete state	symbolic state		path condition
x = 30 y = 15	$x = x_0$ $y = y_0$		

An Illustrative Example

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

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

Concrete Execution | Symbolic Execution

concrete state	symbolic state	path condition
x = 30	$x = x_0$	
y = 15	$y = y_0$	
z = 30	$z = 2*y_0$	

An Illustrative Example

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

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

Concrete Execution		Symbolic Execution	
concrete state	symbolic state		path condition
x = 30	$x = x_0$		$2*y_0 == x_0$
y = 15	$y = y_0$		
z = 30	$z = 2*y_0$		

An Illustrative Example

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

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

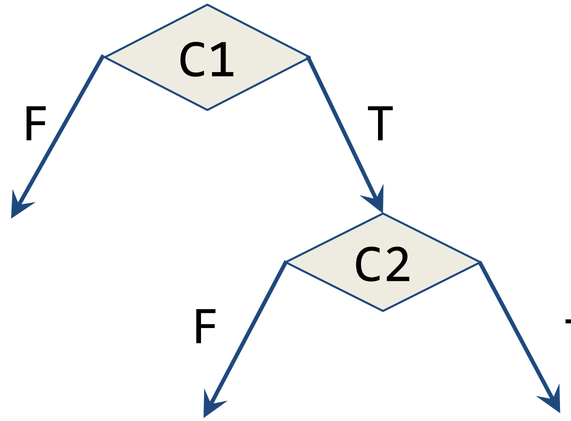
Program Error

	Concrete Execution	Symbolic Execution	
	concrete state	symbolic state	path condition
	x = 30	$x = x_0$	$2*y_0 == x_0$
	y = 15	$y = y_0$	
	z = 30	$z = 2*y_0$	$x_0 > y_0+10$

QUIZ: Computation Tree

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

- [ ] $C1$
- [ ] $C2$
- [ ] $\neg C1$
- [ ] $\neg C2$
- [ ] $C1 \land C2$
- [ ] $C1 \land \neg C2$
- [ ] $\neg C1 \land C2$
- [ ] $\neg C1 \land \neg C2$

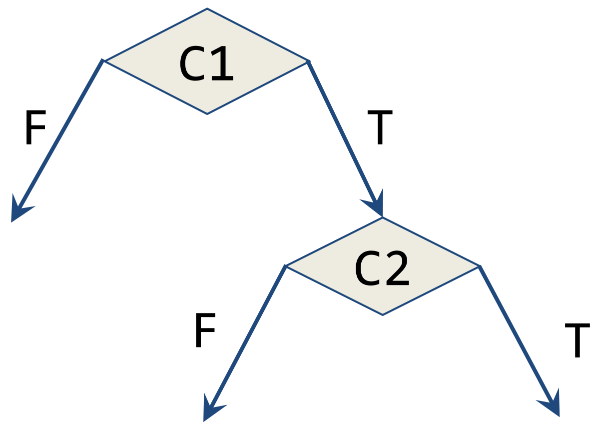


QUIZ: Computation Tree

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

- [x] $C1$
- [ ] $C2$
- [x] $\neg C1$
- [ ] $\neg C2$

- [x] $C1 \land C2$
- [x] $C1 \land \neg C2$
- [ ] $\neg C1 \land C2$
- [ ] $\neg C1 \land \neg C2$



A More Complex Example

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

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

Concrete Execution		Symbolic Execution	
concrete state		symbolic state	path condition
x = 22		$x = x_0$	
y = 7		$y = y_0$	

A More Complex Example

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

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

Concrete Execution		Symbolic Execution	
concrete state	symbolic state	path condition
x = 22	$x = x_0$	
y = 7	$y = y_0$	
z = 601...129	$z = \text{secure\_hash}(y_0)$	

A More Complex Example

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

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

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 22 y = 7 z = 601...129	x = x_0 y = y_0 z = secure_hash(y_0)	secure_hash(y_0) != x_0

Solve: secure_hash(y_0) == x_0

Don’t know how to solve! Stuck?

A More Complex Example

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

void test_me(int x, int y) {
   int z = foo(y);
   if (z == x)
      if (x > y+10)
```

	Concrete Execution		Symbolic Execution	
	concrete state	symbolic state	path condition
	x = 22	x = x_0	secure_hash(y_0) != x_0
	y = 7	y = y_0	
	z = 601...129	z = secure_hash(y_0)	

Solve: secure_hash(y_0) == x_0

Don't know how to solve! Stuck?

Not stuck! Use concrete state: replace y_0 by 7

A More Complex Example

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

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

	Concrete Execution		Symbolic Execution
concrete state		symbolic state	path condition
x = 22		$x = x_0$	$secure\_hash(y_0) \mathrel{!=} x_0$
y = 7		$y = y_0$	
z = 601...129		$z = secure\_hash(y_0)$	

Solve: $601...129 == x_0$

Solution: $x_0 = 601...129,\ y_0 = 7$

A More Complex Example

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

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

Concrete Execution		Symbolic Execution	
concrete state	symbolic state		path condition
x = 601...129	$x = x_0$		
y = 7	$y = y_0$		

A More Complex Example

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

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

Concrete Execution		Symbolic Execution	
concrete state		symbolic state	path condition
x = 601...129		$x = x_0$	
y = 7		$y = y_0$	
z = 601...129		z = secure_hash(y_0)	

A More Complex Example

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

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

Concrete Execution		Symbolic Execution
concrete state	symbolic state	path condition
x = 601...129	$x = x_0$	secure_hash(y_0) == x_0
y = 7	$y = y_0$	
z = 601...129	z = secure_hash(y_0)	

A More Complex Example

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

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

Program Error

Concrete Execution

Symbolic Execution

concrete state	symbolic state	path condition
x = 601...129	x = x_0	secure_hash(y_0) == x_0
y = 7	y = y_0	
z = 601...129	z = secure_hash(y_0)	$x_0 > y_0+10$

QUIZ: Example Application

DSE tests the below program starting with input x = 1. What is the input and constraint (C1 ∧ C2 ∧ C3) solved in each run of DSE? Use depth-first search and leave trailing constraints blank if unused.

Run	x	C1	C2	C3
1	1	5 != x0	7 != x0	9 == x0
2				
3				
4				

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

QUIZ: Example Application

DSE tests the below program starting with input x = 1. What is the input and constraint (C1 ∧ C2 ∧ C3) solved in each run of DSE? Use depth-first search and leave trailing constraints blank if unused.

Run	x	C1	C2	C3
1	1	5 != x0	7 != x0	9 == x0
2	9	5 != x0	7 == x0	
3	7	5 == x0		
4	5			

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

A Third Example

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

void test_me(int x, int y) {
   if (x != y)
      if (foo(x) == foo(y))
         ERROR;
}
```

	Concrete Execution		Symbolic Execution	
concrete state		symbolic state		path condition
x = 22		$x = x_0$		
y = 7		$y = y_0$		

A Third Example

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

void test_me(int x, int y) {
   if (x != y)
      if (foo(x) == foo(y))
         ERROR;
}
```

	Concrete Execution		Symbolic Execution
concrete state		symbolic state	path condition
x = 22		$x = x_0$	$x_0 \neq y_0$
y = 7		$y = y_0$	

A Third Example

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

void test_me(int x, int y) {
   if (x != y)
      if (foo(x) == foo(y))
         ERROR;
}
```

Concrete Execution		Symbolic Execution	
concrete state	symbolic state	path condition	
x = 22	x = x_0	x_0 != y_0	
y = 7	y = y_0	secure_hash(x_0) != secure_hash(y_0)	

Solve: x_0 != y_0 and secure_hash(x_0) == secure_hash(y_0)

Use concrete state: replace y_0 by 7.

A Third Example

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

void test_me(int x, int y) {
   if (x != y)
      if (foo(x) == foo(y))
         ERROR;
}
```

	Concrete Execution		Symbolic Execution
	concrete state	symbolic state	path condition
	x = 22	x = x_0	x_0 != y_0
	y = 7	y = y_0	secure_hash(x_0) != secure_hash(y_0)

Solve: x_0 != 7 and secure_hash(x_0) == 601...129

Use concrete state: replace x_0 by 22.

A Third Example

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

void test_me(int x, int y) {
   if (x != y)
      if (foo(x) == foo(y))
         ERROR;
}
```

False negative!

Concrete Execution		Symbolic Execution	
concrete state	symbolic state		path condition
x = 22	x = x_0		x_0 != y_0
y = 7	y = y_0		secure_hash(x_0) != secure_hash(y_0)

Solve: 22 != 7 and
438...861 == 601...129

Unsatisfiable!

QUIZ: Properties of DSE

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

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

- [ ] DSE is guaranteed to terminate.
- [x] 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

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

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;

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

int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

Data-Structure Example

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;

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

int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

Concrete Execution		Symbolic Execution	
concrete state	symbolic state		path condition
x = 236 p = NULL	x = x_0 p = p_0		

Data-Structure Example

```
typedef struct cell {
   int data;
   struct cell *next;
} cell;

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

int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

Concrete Execution		Symbolic Execution	
concrete state		symbolic state	path condition
$x = 236$ $p = NULL$		$x = x_0$ $p = p_0$	$x_0 > 0$

Data-Structure Example

```
typedef struct cell {
   int data;
   struct cell *next;
} cell;

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

int test_me(int x, cell *p) {
  if (x > 0)
    if (p != NULL)
      if (foo(x) == p->data)
        if (p->next == p)
          ERROR;
  return 0;
}
```

Concrete Execution

Symbolic Execution

concrete state

x = 236

p = NULL

symbolic state

$x = x_0$

$p = p_0$

path condition

$x_0 > 0$

$p_0 == NULL$

Data-Structure Example

```
typedef struct cell {
   int data;
   struct cell *next;
} cell;

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

int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

Concrete Execution		Symbolic Execution	
concrete state	symbolic state		path condition
x = 236 p = NULL	x = x_0 p = p_0		$x_0 > 0$ p_0 == NULL

Solve: $x_0 > 0$ and p_0 != NULL

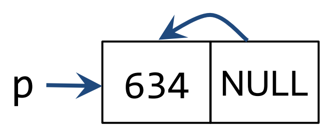
Solution: $x_0 = 236$, $p_0 \rightarrow$ [634 | NULL]

Data-Structure Example

```
typedef struct cell {
   int data;
   struct cell *next;
} cell;

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

int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

Concrete Execution		Symbolic Execution	
concrete state		symbolic state	path condition

x = 236

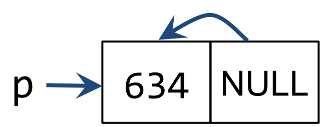


$x = x_0$
$p = p_0$
$p\text{->}data = v_0$
$p\text{->}next = n_0$

Data-Structure Example

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;

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

int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 236	$x = x_0$ $p = p_0$ $p\text{->}data = v_0$ $p\text{->}next = n_0$	$x_0 > 0$



Data-Structure Example

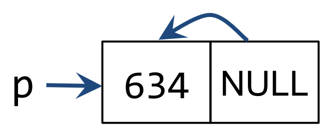
```
typedef struct cell {
    int data;
    struct cell *next;
} cell;

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

int test_me(int x, cell *p) {
    if (x > 0)
        if (p != NULL)
            if (foo(x) == p->data)
                if (p->next == p)
                    ERROR;
    return 0;
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 236	$x = x_0$	$x_0 > 0$
p → [634 | NULL]	$p = p_0$	p_0 != NULL
	$p\text{->}data = v_0$	
	$p\text{->}next = n_0$	

Data-Structure Example

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;

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

int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 236	$x = x_0$	$x_0 > 0$
	$p = p_0$	$p_0 \neq$ NULL
	$p\text{->}data = v_0$	$2*x_0+1 \neq v_0$
	$p\text{->}next = n_0$	



Data-Structure Example

```
typedef struct cell {
   int data;
   struct cell *next;
} cell;

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

int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

Concrete Execution | **Symbolic Execution**

concrete state

$x = 236$

p → | 634 | NULL |

symbolic state

$x = x_0$

$p = p_0$

$p\text{->}data = v_0$

$p\text{->}next = n_0$

path condition

$x_0 > 0$

$p_0 \ != NULL$

$2{*}x_0{+}1 \ != v_0$

Solve: $x_0 > 0$ and $p_0 \ != NULL$ and $2{*}x_0{+}1 == v_0$

Solution: $x_0 = 1$, p_0 → | 3 | NULL |

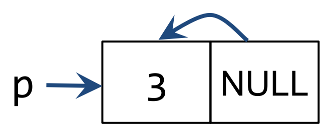
Data-Structure Example

```
typedef struct cell {
   int data;
   struct cell *next;
} cell;

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

int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

Concrete Execution

concrete state

x = 1



Symbolic Execution

symbolic state

$x = x_0$

$p = p_0$

$p\text{->}data = v_0$

$p\text{->}next = n_0$

path condition

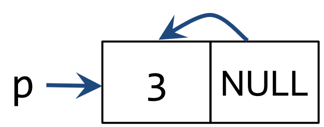
Data-Structure Example

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;

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

int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

Concrete Execution

Symbolic Execution

concrete state	symbolic state	path condition
x = 1	$x = x_0$	$x_0 > 0$
	$p = p_0$	
	$p\text{->}data = v_0$	
	$p\text{->}next = n_0$	



Data-Structure Example

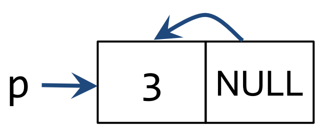
```
typedef struct cell {
   int data;
   struct cell *next;
} cell;

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

int test_me(int x, cell *p) {
  if (x > 0)
    if (p != NULL)
      if (foo(x) == p->data)
        if (p->next == p)
          ERROR;
  return 0;
}
```

Concrete Execution

Symbolic Execution

concrete state

x = 1



symbolic state

$x = x_0$

$p = p_0$

$p\text{->}data = v_0$

$p\text{->}next = n_0$

path condition

$x_0 > 0$

$p_0 \text{ != NULL}$

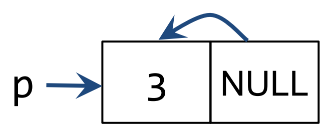
Data-Structure Example

```
typedef struct cell {
   int data;
   struct cell *next;
} cell;

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

int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

Concrete Execution

concrete state

$x = 1$



Symbolic Execution

symbolic state

$x = x_0$

$p = p_0$

$p\text{->}data = v_0$

$p\text{->}next = n_0$

path condition

$x_0 > 0$

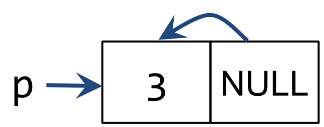
$p_0 \; != \; NULL$

$2*x_0+1 == v_0$

Data-Structure Example

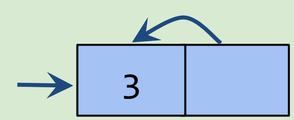
```
typedef struct cell {
   int data;
   struct cell *next;
} cell;

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

int test_me(int x, cell *p) {
  if (x > 0)
    if (p != NULL)
      if (foo(x) == p->data)
        if (p->next == p)
          ERROR;
  return 0;
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 1	$x = x_0$ $p = p_0$ $p\text{->}data = v_0$ $p\text{->}next = n_0$	$x_0 > 0$ $p_0 \mathrel{!=} NULL$ $2*x_0+1 == v_0$ $n_0 \mathrel{!=} p_0$



Solve: $x_0 > 0$ and $p_0 \mathrel{!=} NULL$ and $2*x_0+1==v_0$ and $n_0 == p_0$

Solution: $x_0 = 1$, p_0



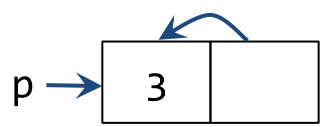
Data-Structure Example

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;

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

int test_me(int x, cell *p) {
    if (x > 0)
        if (p != NULL)
            if (foo(x) == p->data)
                if (p->next == p)
                    ERROR;
    return 0;
}
```

Concrete Execution

concrete state

x = 1



Symbolic Execution

symbolic state

$x = x_0$

$p = p_0$

$p\text{->}data = v_0$

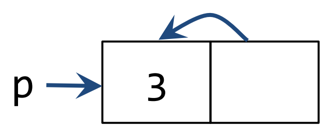
$p\text{->}next = n_0$

path condition

Data-Structure Example

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;

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

int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

Concrete Execution		Symbolic Execution	
concrete state		symbolic state	path condition
x = 1		$x = x_0$ $p = p_0$ $p\text{->}data = v_0$ $p\text{->}next = n_0$	$x_0 > 0$



Data-Structure Example

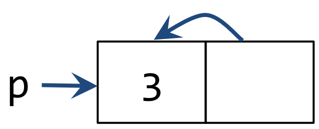
```
typedef struct cell {
    int data;
    struct cell *next;
} cell;

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

int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

Concrete Execution

Symbolic Execution

concrete state

$x = 1$



symbolic state

$x = x_0$

$p = p_0$

$p\text{->}data = v_0$

$p\text{->}next = n_0$

path condition

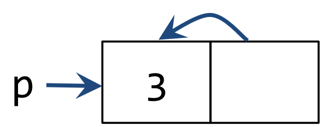
$x_0 > 0$

$p_0 \text{ != NULL}$

Data-Structure Example

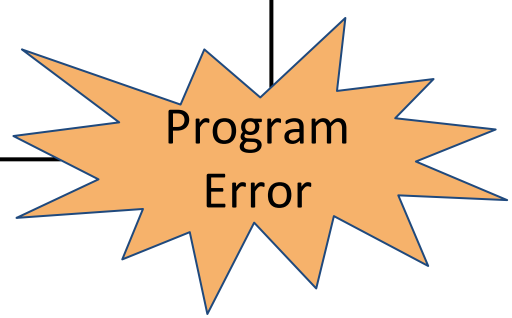
```
typedef struct cell {
    int data;
    struct cell *next;
} cell;

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

int test_me(int x, cell *p) {
    if (x > 0)
        if (p != NULL)
            if (foo(x) == p->data)
                if (p->next == p)
                    ERROR;
    return 0;
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 1	$x = x_0$ $p = p_0$ $p\text{->}data = v_0$ $p\text{->}next = n_0$	$x_0 > 0$ $p_0 \mathrel{!=} NULL$ $2*x_0+1 == v_0$



Data-Structure Example

```
typedef struct cell {
    int data;
    struct cell *next;
} cell;

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

int test_me(int x, cell *p) {
   if (x > 0)
      if (p != NULL)
         if (foo(x) == p->data)
            if (p->next == p)
               ERROR;
   return 0;
}
```

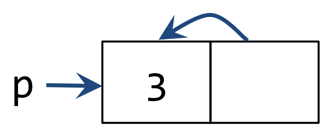


Concrete Execution

Symbolic Execution

concrete state

x = 1



symbolic state

$x = x_0$
$p = p_0$
$p\text{->}data = v_0$
$p\text{->}next = n_0$

path condition

$x_0 > 0$
$p_0 \;!= NULL$
$2*x_0+1 == v_0$
$n_0 \;!= p_0$

Approach in a Nutshell

- Generate concrete inputs, each taking different program path
- On each input, execute program both concretely and symbolically
- Both cooperate with each other:
  - Concrete execution guides symbolic execution
    - Enables it to overcome incompleteness of theorem prover
  - Symbolic execution guides generation of concrete inputs
    - Increases program code coverage

QUIZ: Characteristics of DSE

- The testing approach of DSE is:

  ☐ Automated, black-box ☐ Manual, black-box
  ☐ Automated, white-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

QUIZ: Characteristics of DSE

- The testing approach of DSE is:

  ☐ Automated, black-box ☐ Manual, black-box
  ☑ Automated, white-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

Case Study: SGLIB C Library

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

Case Study: SGLIB C Library

Name	Run time (sec.)	# iterations	# branches explored	% branch coverage	# functions tested	# bugs found
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

Case Study: Needham-Schroeder Protocol

- Tested a C implementation of a security protocol (Needham-Schroeder) with a known (man-in-the-middle) attack
  - 600 lines of code
  - Took fewer than 13 seconds on a machine with 1.8 GHz processor and 2 GB RAM to discover the attack

- In contrast, a software model-checker (VeriSoft) took 8 hours

Realistic Implementations

- KLEE: LLVM (C family of languages)
- PEX: .NET Framework
- jCUTE: Java
- Jalangi: Javascript
- SAGE and S2E: binaries (x86, ARM, ...)

Case Study: SAGE Tool at Microsoft

- SAGE = Scalable Automated Guided Execution
- Found many expensive security bugs in many Microsoft applications (Windows, Office, etc.)
- Used daily in various Microsoft groups, runs 24/7 on 100's of machines
- What makes it so useful?
  - Works on large applications => finds bugs across components
  - Focus on input file fuzzing => fully automated
  - Works on x86 binaries => easy to deploy (not dependent on language or build process)

Example: SAGE Crashing a Media Parser

```
00000000h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000060h: 00 00 00 00                                     ; ....
```

```
00000000h: 52 49 46 46 00 00 00 00 00 00 00 00 00 00 00 00 ; RIFF............
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000060h: 00 00 00 00                                     ; ....
```

```
00000000h: 52 49 46 46 00 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF....*** ....
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000040h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000060h: 00 00 00 00                                     ; ....
```

… after a few more iterations:

```
00000000h: 52 49 46 46 3D 00 00 00 ** ** ** 20 00 00 00 00 ; RIFF=...*** ....
00000010h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 73 74 72 68 00 00 00 00 76 69 64 73 ; ....strh....vids
00000040h: 00 00 00 00 73 74 72 66 B2 75 76 3A 28 00 00 00 ; ....strf²uv:(...
00000050h: 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 ; ................
00000060h: 00 00 00 00                                     ; ....
```

What Have We Learned?

- What is (dynamic) symbolic execution?
- Systematically generate (numeric and pointer) inputs
- Computation tree and error reachability
- Tracking concrete state, symbolic state, path condition
- Combined dynamic and static analysis => Hybrid analysis
- Complete, but no soundness or termination guarantees