

# Dynamic Symbolic Execution

CS 6340

{HEADSHOT}

This lesson introduces a powerful automated testing technique called dynamic symbolic execution. This technique is based on hybrid analysis: it combines static analysis and dynamic analysis in a manner that gains the benefits of both.

The goal of the technique is to maximize program path coverage and thereby help uncover potential bugs. To this end, this technique systematically generates inputs to a given program that drive its execution along different paths in the program.

The technique is highly versatile: it is not limited to any programming language constructs or idioms, and while it may result in false negatives -- that is, it may miss bugs -- it does not produce false positives, that is, every assertion violation it discovers is indeed real.

The remarkable success of this technique has led to open-source as well as commercial implementations of the technique for virtually every mainstream programming language.

This lesson will present the principles underlying this technique and prepare you to apply it to test small units of code as well as entire, large, complex programs.

## 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**

Writing and maintaining tests is tedious and error-prone. A compelling idea to overcome this problem is automated test generation. This idea has several benefits.

First, it can be used to generate a test suite that can then be run regularly to check for regressions in the program.

Second, it can be used to execute all reachable statements in the program, and thereby attain high code coverage.

Third, it can be used to catch any assertion violations. Assertions, as you may recall, are a general mechanism for specifying program correctness.

## 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

In this lesson, we will discuss a new technique for automated test generation called dynamic symbolic execution.

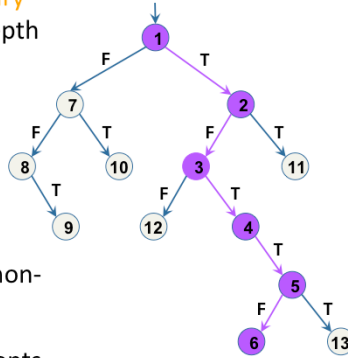
This technique keeps track of the program state both concretely, like a dynamic analysis, and symbolically, like a static analysis.

It solves constraints to guide the program's execution at branch points. In this manner, it systematically explores all execution paths of the unit being tested.

Dynamic symbolic execution is an example of a hybrid analysis: it 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



To understand how dynamic symbolic execution works, let's visualize a program as a binary tree with possibly infinite depth called the computation tree.

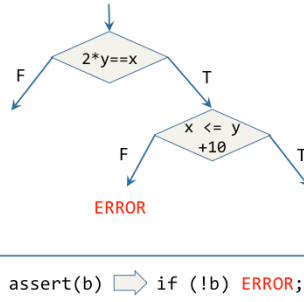
Each node in the tree represents the execution of a conditional statement, and each edge represents the execution of a sequence of non-conditional statements. The left child of a node N represents the branch point reached by taking the "false" branch at N, and the right child of a node N represents the branch point reached by taking the "true" branch at N.

Note that we've "unrolled" all loops in the program by representing each loop as a sequence of consecutive if-then-else statements. This means that our tree might have infinite depth, as some loops may be unbounded.

A path in the computation tree represents an equivalence class of inputs: if two inputs lead to the same set of branch points and statements executed, we consider those inputs to be equivalent. The goal of dynamic symbolic execution is to systematically generate non-equivalent inputs, that is, inputs that lead the program's execution along different paths in its computation tree. We have numbered the nodes in this example tree in a possible ordering, a depth-first ordering, in which dynamic symbolic execution will visit them. But let's not get too much into details of how dynamic symbolic execution chooses paths quite yet. In fact, for computational trees with infinite depth, this is a sophisticated problem!

## 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");  
}
```



Let's start with a comparatively simple computation tree corresponding to the following program `test_me`.

The program takes as input two integer variables  $x$  and  $y$ . It first tests whether  $2*y == x$ . If  $2*y != x$ , then the program exits normally. But if  $2*y == x$ , then the program proceeds to test whether  $x <= y + 10$ . If  $x <= y + 10$ , then the program exits normally. But if  $x > y + 10$ , then the program throws an error.

The computation tree that results from this program has just two nodes, corresponding to the two branch points. The root node is labeled " $2*y == x$ " which corresponds to the outer branch point. If this test fails, then the program exits normally, so the root has no left child. If the test succeeds, then we reach another branch point. So the root has a right child, labeled " $x <= y + 10$ " corresponding to the test we perform at this second branch point.

If this test succeeds, then the program exits normally. If the test fails, then the program throws an error, which we symbolize by marking the left edge of the corresponding node by "ERROR". In both these cases, there are no further child nodes, as there are no additional branch points in the program.

In general, we will represent an assertion in this manner: perform a test, and if the test fails, then the program reaches a distinguished ERROR label.

One last point of interest is that, because the program has no unbounded loops, the computational tree is finite.

## 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:  
 $1/2^{32} \approx 0.000000023\%$

To better motivate the dynamic symbolic execution approach, let's look at some existing approaches for automated test generation.

First, we'll consider random testing, in which we generate random inputs and execute the program on these generated inputs.

Let's look at the example program `test_me`, which takes an integer `x`, and, if `x` equals 94389, it raises an error. Otherwise, the program exits normally.

Assuming an int is 32 bits and each possible int has an equal chance of being generated, the probability that our random input will detect this error is astronomically small: one out of 2 to the 32nd power, which is about 23 billionths of a percent. So there is a high probability that random testing will generate a false negative in a limited amount of time: incorrectly stating that the ERROR label is unreachable.

## 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

Another approach that has existed since the 1970s is called “symbolic execution.” In this approach, input variables are represented symbolically instead of by concrete values. The program is executed symbolically, and symbolic path constraints are collected as the program runs. At each branch point, we invoke a theorem prover to determine whether a branch can be taken; if so, then we take the branch, otherwise, we ignore the branch as dead code.

For example, in this new version of the program `test_me`, instead of testing that the input variable `x` equals a particular integer value, we ask the theorem prover if there is any integer value for `x` that satisfies the condition `x * 3 != 15`. The theorem prover would respond “yes”, allowing us to deduce that the “false” branch is reachable. Because the false branch terminates the program, we now ask the theorem prover if the negation of `x * 3 != 15` has any satisfying assignment (that is, does `x * 3 == 15` have a satisfying assignment?). The theorem prover would respond “yes,” so we’d explore the “true” branch of the first condition while “collecting” the symbolic constraint that `x * 3 == 15`.

Next, we reach the second condition `x % 5 == 0`. We now ask the theorem prover if the expression `x * 3 == 15 AND x % 5 == 0` has a satisfying assignment. The theorem prover would respond “yes,” so we’d explore the “true” branch, leading to program termination. Finally, we negate the condition and ask if `x * 3 == 15 AND x % 5 != 0` has a satisfying assignment. The theorem prover would respond “no,” meaning that the false branch is unreachable, dead code. We therefore skip that branch. Since we have then explored all feasible paths and not reached an error, we can conclude that the program will not raise an error in any execution.

However, because of the possibility of exponential explosion in branch conditions, it becomes quickly obvious that this strategy 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

Another problem with purely symbolic approaches is that they may not be powerful enough to decide if a particular constraint has a satisfying assignment.

For example, in this version of the program `test_me`, we ask the theorem prover to decide whether there exists an integer  $x$  such that  $2^x$  modulo a product of two large prime numbers, denoted here by constant  $c$ , equals 17. If so, the program throws an error. Otherwise, the program terminates normally.

Note that this particular condition is an instance of the discrete logarithm problem, which is believed to be computationally intractable on classical computers. When dealing with such a difficult question to resolve symbolically, our theorem prover might throw up its hands and give up. In this situation, the theorem prover errs on the side of soundness by declaring the condition to be feasible, even if there really is no integer  $x$  such that  $2^x$  modulo  $c$  equals 17. In this case, the symbolic execution approach would yield a false positive, considering both branches of the condition to be reachable when really only the "false" branch is reachable.



## 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;
}
```

A common theme in this course is to try to combine two approaches in order to get the benefits of both without suffering from the limitations of either. In this case, we will combine the concrete execution approach of random testing with the symbolic execution approach we just discussed. This approach is called dynamic symbolic execution, or DSE for short.

Here's how it works: it initially sets the input values of the function to be tested randomly, and observes the branches of computation that are taken. It also keeps track of constraints on the program's state symbolically.

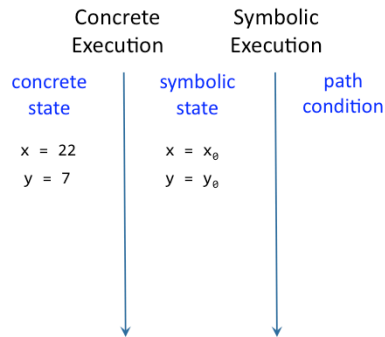
Upon reaching the end of some computational path, DSE will backtrack to some branch point and decide whether there is a satisfying assignment to the program's input variables that allows the other branch to be taken at that point. If so, the solver generates such an assignment, and DSE continues onward. If not, then DSE ignores that branch as dead code.

So far, this sounds much like symbolic execution. However, there's one further subtlety. If a condition becomes complex enough that the solver cannot find a satisfying assignment, then the solver "plugs in" the concrete values that DSE is working with to one or more variables in the constraints to simplify them. This strategy makes the constraint solver into what is called an "incomplete" theorem prover: it will never declare an unsatisfiable constraint to be satisfiable, but it may fail to satisfy some satisfiable constraints because of the simplification being made. (This contrasts with pure symbolic execution, whose constraint solver was unsound -- it would declare some unsatisfiable constraints to be satisfiable.)

## 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;
}
```

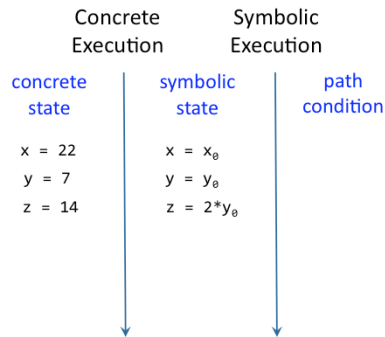


Let's walk through an example of how DSE would identify failure-generating inputs to a function. In this example, we will be looking at the following functions: `foo`, which takes an `int v` and returns the `int` 2 times `v`; and the function `test_me`, which takes two `ints`, `x` and `y`, and has no return type. `test_me` operates as follows: it sets the `int z` equal to `foo` of `y`. Then, if `z` equals `x`, it makes an additional check if `x` is greater than `y` plus 10. If this second check passes, then the program throws an error. If either of the checks fail, then the program terminates without error.

Let's look at how DSE would work on the `test_me` function. First, two random inputs would be generated for `x` and `y`: say, `x = 22` and `y = 7`. Additionally, DSE would keep track of the symbolic state of the program: `x` equals some number `x0` and `y` equals some number `y0`.

## 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;  
}
```

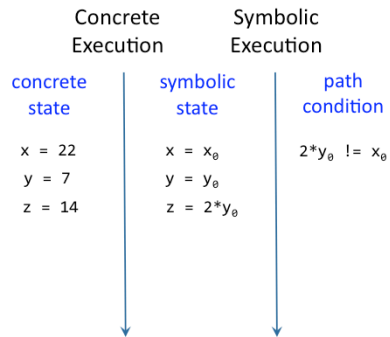


On the first line, integer  $z$  is assigned the output of the function `foo` applied to  $y$ . In the concrete state, this means that  $z$  now equals 14. And in the symbolic state, the variable  $z$  has the value 2 times  $y_0$ . Note that DSE has the ability to concretely and symbolically compute the output of a call to a function, such as `foo`.

## An Illustrative Example

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

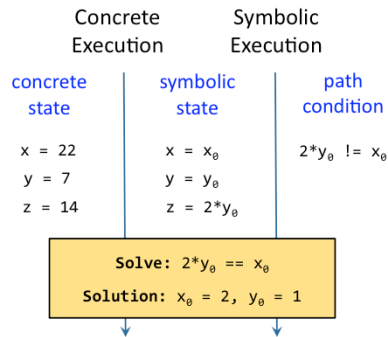
void test_me(int x, int y) {
    int z = foo(y);
    if (z == x)
        if (x > y+10)
            ERROR;
}
```



At the branch point “ $z == x$ ”, DSE observes that the current concrete value of  $x$  does not equal the current concrete value of  $z$ . Symbolically, DSE stores this constraint ( $z \neq x$ ) as a path condition over the symbolic values of  $z$  and  $x$  as:  $2*y_0 \neq x_0$ . DSE then follows the “false” branch from this point, leading to the end of the program.

## An Illustrative Example

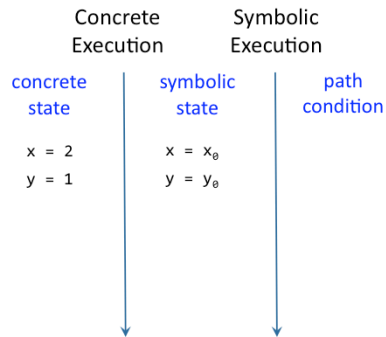
```
int foo(int v) {  
    return 2*v;  
}  
  
void test_me(int x, int y) {  
    int z = foo(y);  
    if (z == x)  
        if (x > y+10)  
            ERROR;  
}
```



Now, DSE will backtrack to this branch point and attempt to take the “true” branch. For this purpose, it negates the most recently added constraint in the path condition, which is  $2*y_0 \neq x_0$ , to  $2*y_0 == x_0$ . It asks a solver to find a satisfying assignment to the constraint  $2*y_0 == x_0$ . There are certainly two integers satisfying this constraint; let’s suppose the solver returns  $x_0 = 2$  and  $y_0 = 1$ .

## An Illustrative Example

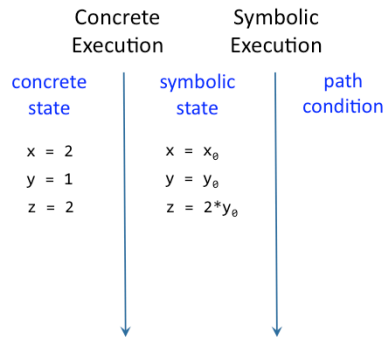
```
int foo(int v) {  
    return 2*v;  
}  
  
void test_me(int x, int y) {  
    int z = foo(y);  
    if (z == x)  
        if (x > y+10)  
            ERROR;  
}
```



DSE then restarts the `test_me` function, this time calling it with the concrete input values generated by the constraint solver:  $x = 2$  and  $y = 1$ . The symbolic state begins anew with  $x = x_0$  and  $y = y_0$ .

## 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;  
}
```

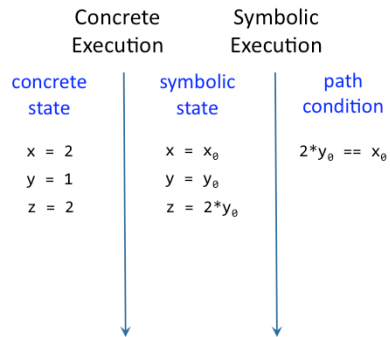


After executing the first line, z takes on the concrete value 2 (the output of foo(y)) and, as before, the symbolic value  $2 * y_0$ .

## 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;
}
```



At the next line, we inspect the branch condition  $z == x$ . In this case, the condition is true, so our path condition becomes  $2*y_0 == x_0$  (substituting the symbolic values for  $z$  and  $x$  into the branch condition). We then inspect the next line of the “true” branch of this condition.



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

At the next branch point, x has the concrete value 2 and y+10 has the concrete value 11, so we take the false branch, terminating the program. We also add the symbolic constraint  $x_0 \leq y_0 + 10$  to the path condition, which is the negation of the branch condition we found to be false (with appropriate symbolic substitutions for the variables x and y).

## 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;
}

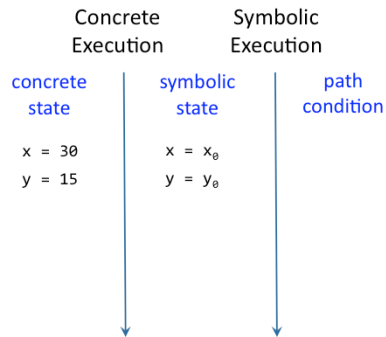
```

Concrete Execution	Symbolic Execution
concrete state	symbolic state
x = 2	x = $x_0$
y = 1	y = $y_0$
z = 2	z = $2*y_0$
	path condition
	$2*y_0 == x_0$
	$x_0 <= y_0+10$
<b>Solve: (<math>2*y_0 == x_0</math>) and (<math>x_0 &gt; y_0+10</math>)</b> <b>Solution: <math>x_0 = 30, y_0 = 15</math></b>	

Since DSE has reached the end of the program, it negates the most recently added constraint in the path condition to obtain  $x_0 > y_0 + 10$ , and then it passes the constraints  $2*y_0 == x$  AND  $x_0 > y_0 + 10$  to the solver. The solver finds that there is a solution to these constraints; in particular, it returns  $x_0 = 30$  and  $y_0 = 15$ .

## An Illustrative Example

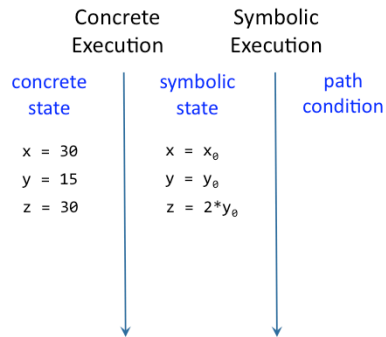
```
int foo(int v) {  
    return 2*v;  
}  
  
void test_me(int x, int y) {  
    int z = foo(y);  
    if (z == x)  
        if (x > y+10)  
            ERROR;  
}
```



Now, DSE runs the test\_me function again, this time with inputs x = 30 and y = 15. The symbolic state again starts as x = x<sub>0</sub> and y = y<sub>0</sub>.

## An Illustrative Example

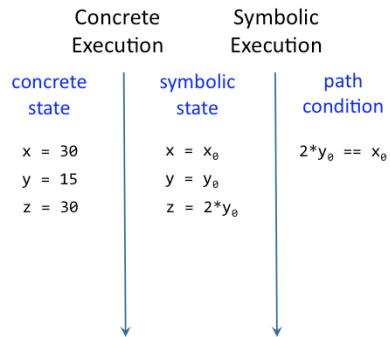
```
int foo(int v) {  
    return 2*v;  
}  
  
void test_me(int x, int y) {  
    int z = foo(y);  
    if (z == x) ←  
        if (x > y+10)  
            ERROR;  
}
```



z is assigned the concrete value 30, while its symbolic value is  $2*y_0$ , as before.

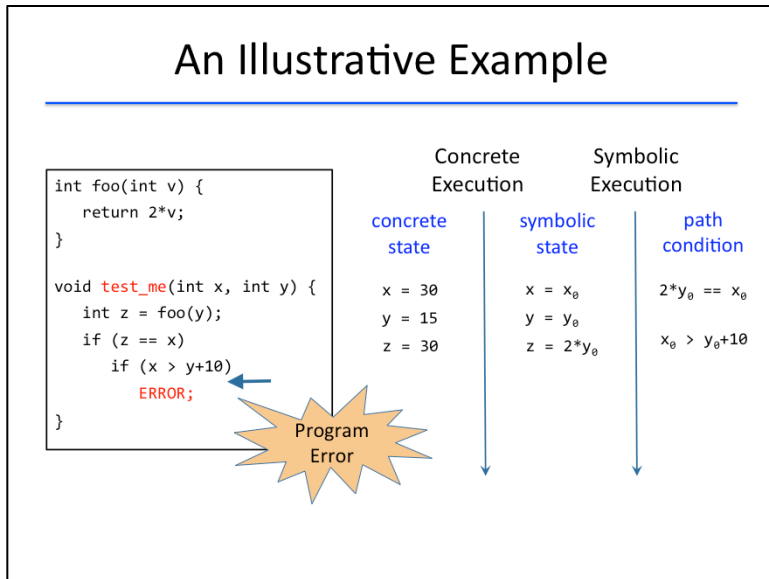
## An Illustrative Example

```
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;  
}
```



When we reach the branch condition  $z == x$ , we see that it is true, so we add the symbolic constraint  $2*y_0 == x_0$ .

## An Illustrative Example



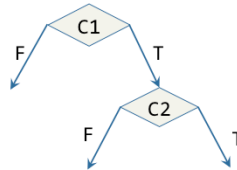
Then, at the next branch point, the concrete value of  $x$  is indeed greater than the concrete value of  $y$  plus 10, so we add the new symbolic constraint  $x_0 > y_0 + 10$ .

This branch leads us to the error, at which point we have identified a concrete input which causes the program to fail:  $x = 30$  and  $y = 15$ .

## QUIZ: Computation Tree

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

- |                                    |   |
|------------------------------------|---|
| <input type="checkbox"/> C1        | <input type="checkbox"/> C1 $\wedge$ C2               |
| <input type="checkbox"/> C2        | <input type="checkbox"/> C1 $\wedge$ $\neg$ C2        |
| <input type="checkbox"/> $\neg$ C1 | <input type="checkbox"/> $\neg$ C1 $\wedge$ C2        |
| <input type="checkbox"/> $\neg$ C2 | <input type="checkbox"/> $\neg$ C1 $\wedge$ $\neg$ C2 |



{QUIZ SLIDE}

Now that you've seen an example of how DSE works, take a moment to consider this quiz. We are given a computation tree as follows: the program starts by checking condition C1. If false, the program terminates. If true, the program checks condition C2. Regardless of how C2 evaluates, the program terminates.

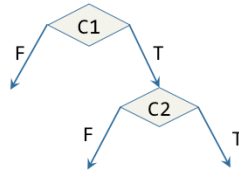
Which of these 8 constraints might DSE possibly solve in exploring this computation tree? That is, select each constraint that might be fed to the constraint solver.

- C1
- C2
- not C1
- not C2
- C1 and C2
- C1 and not C2
- not C1 and C2
- not C1 and not C2

## QUIZ: Computation Tree

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

- |   |   |
|---|---|
| <input checked="" type="checkbox"/> C1        | <input checked="" type="checkbox"/> $C1 \wedge C2$      |
| <input type="checkbox"/> C2                   | <input checked="" type="checkbox"/> $C1 \wedge \neg C2$ |
| <input checked="" type="checkbox"/> $\neg C1$ | <input type="checkbox"/> $\neg C1 \wedge C2$            |
| <input type="checkbox"/> $\neg C2$            | <input type="checkbox"/> $\neg C1 \wedge \neg C2$       |



{SOLUTION SLIDE}

If DSE evaluates C1 and initially finds it to be false, then DSE will subsequently attempt to solve C1. [Mark appears on C1.]

On the other hand, if C1 initially evaluates to true, then DSE will proceed to evaluate C2.

If C2 is found to be true, then DSE will subsequently attempt to solve (C1 and not C2) [mark appears on (C1 and not C2)]; however, if C2 is found to be false, then DSE will subsequently attempt to solve (C1 and C2) [mark appears on (C1 and C2)].

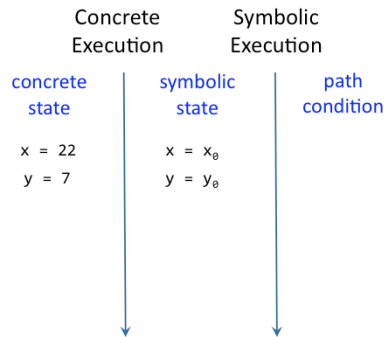
Finally, after DSE finishes exploring the “true” subtree of C1, it will attempt to solve (not C1) in order to try to explore the “false” subtree [mark appears on (not C1)].



## 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;
}
```



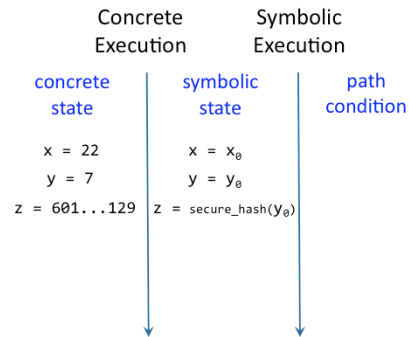
Let's look at how dynamic symbolic execution handles a more complicated example. Here, we've left the test\_me function the same, but we've altered the behavior of foo so that it now securely hashes its input and outputs the result of that hash.

DSE begins this example the same way as before: it takes the random inputs (we'll again use  $x = 22$  and  $y = 7$ ) and stores them in its concrete state; and it stores  $x = x_0$  and  $y = y_0$  in its symbolic state.

## A More Complex Example

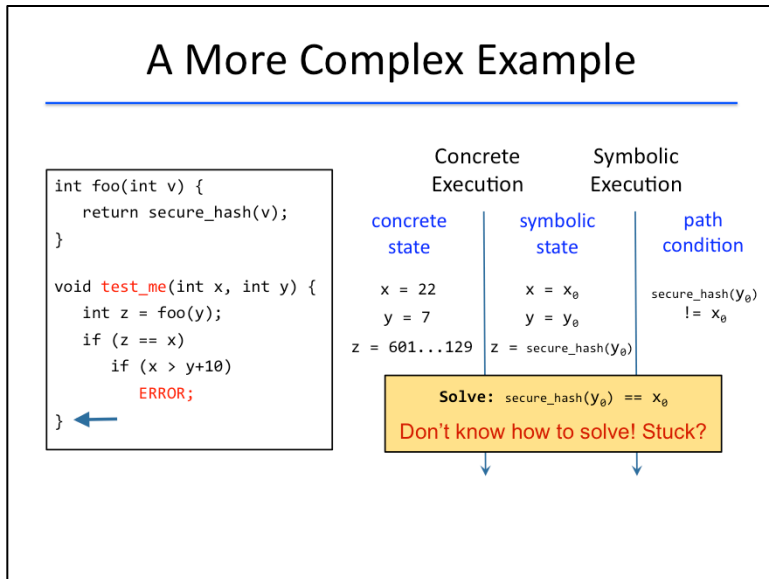
```
int foo(int v) {
    return secure_hash(v);
}

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



In this program, z is again assigned the output of foo(y). However, its concrete value this time is a large number with over 150 digits, starting with the digits 601 and ending in the digits 129. (Let's ignore for now the overflow that would occur in some languages in trying to store such a large number.) Symbolically, z takes on the value secure\_hash(y\_0).

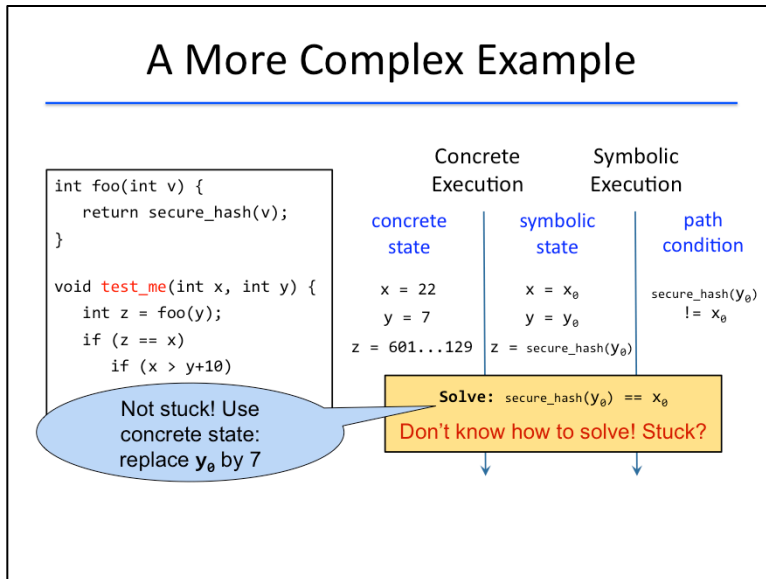
## A More Complex Example



Comparing the concrete values of x and z shows that they are different, so the symbolic constraint `secure_hash(y_0) != x_0` is added to the path condition, and we reach the end of the program.

In order to take the other branch, DSE needs to determine a pair of inputs  $x_0$  and  $y_0$  such that the most recently added constraint in the path condition evaluates to false. That is, such that `secure_hash(y_0) == x_0`. However, the nature of a secure hash function is that it is extremely difficult to solve an equation like this.

## A More Complex Example



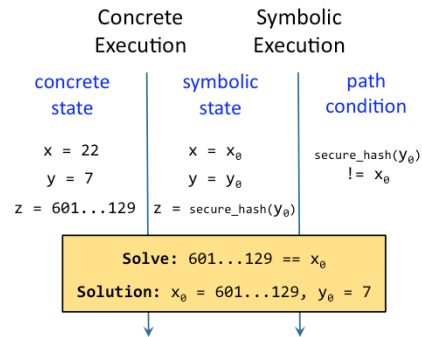
This example showcases the difference between symbolic execution, as previously described, and dynamic symbolic execution. Recall that symbolic execution would have thrown up its hands at this point and, by default, declared the constraint  $\text{secure\_hash}(y_0) == x_0$  satisfiable, thereby continuing down the “true” branch of execution.

Dynamic symbolic execution, by contrast, uses its concrete state to simplify the symbolic constraint. In this case, it would replace  $y_0$  in the symbolic constraint by 7, the concrete value of  $y$  in the program at that point.

## A More Complex Example

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

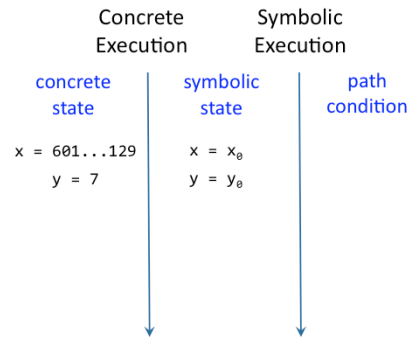
void test_me(int x, int y) {
    int z = foo(y);
    if (z == x)
        if (x > y+10)
            ERROR;
}
```



The constraint to be solved is then  $601\dots129 == x_0$ , which is easy for our constraint solver to solve: just take  $x$  equal to that number. (Note that it wouldn't work to plug in 22 for  $x_0$  and then solve for  $y_0$ , as secure hashes are deliberately difficult to invert.)

## A More Complex Example

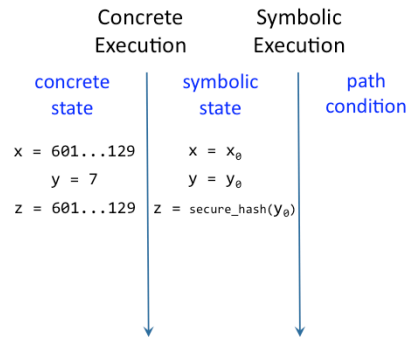
```
int foo(int v) {  
    return secure_hash(v);  
}  
  
void test_me(int x, int y) {  
    int z = foo(y);  
    if (z == x)  
        if (x > y+10)  
            ERROR;  
}
```



Dynamic symbolic execution re-evaluates the test\_me function using these new concrete inputs: x = 601...129, y = 7. The symbolic state as usual starts as x = x<sub>0</sub> and y = y<sub>0</sub>. Then,

## 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;  
}
```

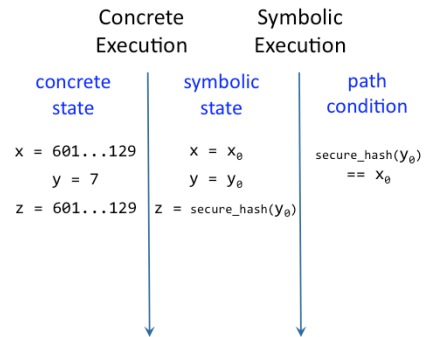


The variable  $z$  is assigned  $\text{foo}(7)$ , which is the output of the secure hash of 7. The symbolic value of  $z$  is again  $\text{secure\_hash}(y_0)$ .

## A More Complex Example

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

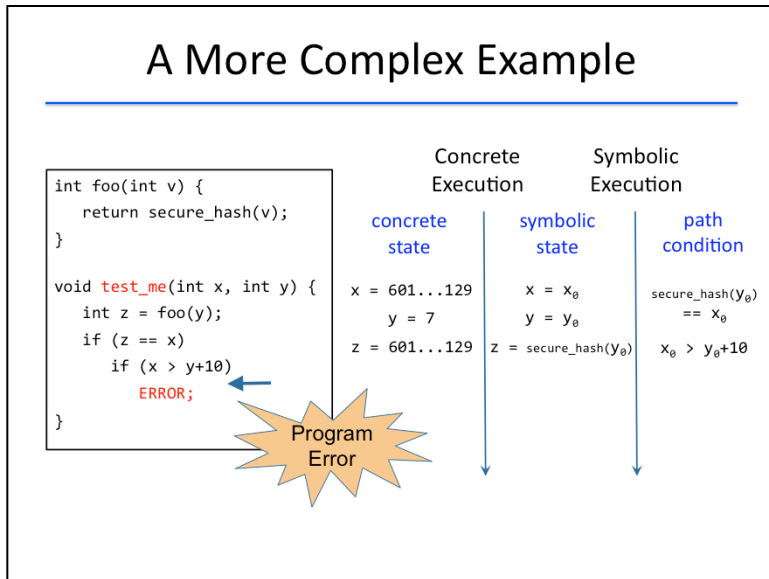
void test_me(int x, int y) {
    int z = foo(y);
    if (z == x)
        if (x > y+10) ←
            ERROR;
}
```



Now, at the branch point, the concrete values of  $x$  and  $z$  are indeed equal, so the “true” branch is taken, as expected.



## A More Complex Example



At the next branch point, we check whether  $x$  is greater than  $y + 10$ . The concrete values of  $x$  and  $y$  satisfy this constraint (integer overflow notwithstanding), so we take the true branch again, which leads to the error in the program.

## QUIZ: Example Application

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

Run	x	C1	C2	C3
1	1	$5 \neq x0$	$7 \neq x0$	$9 == x0$
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 SLIDE}

Now consider the following function, `test_me`, which takes the `int x` as an argument and returns an `int`. The program reads as follows:

“int brackets A is assigned open-curly-brace five, seven, nine, close-curly-brace”

“int i is assigned zero”

“while i is less than 3 open-curly-brace”

“If A sub i is equal to x, break”

“Increment i”

“close-curly-brace”

“return i”

Suppose DSE tests this function starting with the input  $x = 1$ . Write the input used and constraints solved in each iteration of DSE. Assume a depth-first search of the program’s computation tree, and leave a trailing constraint blank if it is unused (for example, if only two constraints are solved for some iteration, leave C3 blank for that iteration). Also use the name `x0` to represent the symbolic variable corresponding to the input variable `x`.

I’ve filled in the first row for you: on iteration 1, the input `x` is 1, C1 is the constraint 5 does not equal `x0`, C2 is the constraint 7 does not equal `x0`, and C3 is the constraint 9 equals `x0`.

## QUIZ: Example Application

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

Run	x	C1	C2	C3
1	1	$5 \neq x0$	$7 \neq x0$	$9 == x0$
2	9	$5 \neq x0$	$7 == x0$	
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;
}
```

### {SOLUTION SLIDE}

After the end of the first iteration of DSE, the constraint C3 forces the new input to be 9.

In the second iteration, the first path constraint added is  $5 \neq x0$ , when the concrete input 9 fails to equal  $A[0]$ . The second path constraint added is  $7 \neq x0$ , and the third path constraint added is  $9 == x0$ , leading to the termination of the program. Normally we'd negate the most recently added constraint and solve the resulting expression, but this would lead us to take a path we've already explored. So the next step is to discard the third path constraint and negate the second path constraint, resulting in C1 being  $5 \neq x0$ , C2 being  $7 == x0$ , and C3 being left blank.

These constraints require the third concrete input to be 7. For the resulting run of `test_me`, the first path constraint added is again  $5 \neq x0$ , and then  $7 == x0$  would be added before the program terminates. DSE has already fully explored the subtree where  $7 == x0$  has been negated, so it needs to backtrack and negate  $5 \neq x0$  in order to continue exploring the computation tree. So the only constraint to be solved is  $5 == x0$ .

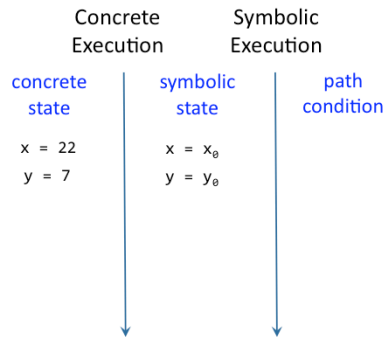
This forces the next concrete value of  $x$  to be 5, and this leads the program to terminate after the first branch point, the only path constraint added being  $5 == x0$ . Negating  $5 == x0$  leads to a previously explored subtree, so there's nothing left to do: the entire computation tree has been explored, and so DSE terminates. There are no constraints to solve.

Note that this program has a bounded loop; in general, loops can result in infinite computation trees, but in this case the tree remains finite. This example also illustrates that not all conditions in the program (for example, the condition " $i < 3$ ") result in nodes in the computation tree. The reason is that only conditions that are data-dependent upon the program's input result in branch points in the computation tree. Note also that even expressions such as  $A[i]$  are constants (being represented as 5, 7, or 9 in the constraints) for the same reason: neither  $A$  nor  $i$  are data-dependent upon the program's input  $x$ .

## 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;
}
```

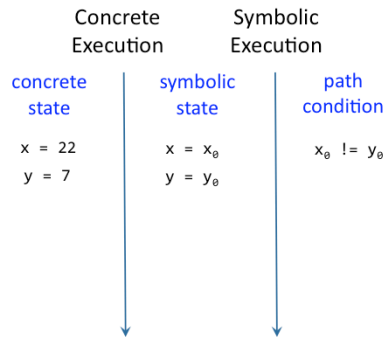


Let's take a look at one more example to showcase how dynamic symbolic execution differs from its static counterpart. Here, the `foo` function still returns a secure hash of its input, but the `test_me` function operates as follows: if its inputs `x` and `y` are different, then if `foo(x)` equals `foo(y)`, the program throws an error. If either of these conditions is false, then the program terminates without error.

Suppose DSE starts again with the concrete random inputs `x = 22` and `y = 7`. The symbolic state again is set to `x = x0` and `y = y0`.

## 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;  
}
```



At the first condition, since the concrete values of  $x$  and  $y$  are different, the “true” branch is taken, and we add the symbolic constraint  $x_0 \neq y_0$  to the path condition.

## 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 y = 7	x = $x_0$ y = $y_0$	$x_0 \neq y_0$ secure_hash( $x_0$ ) != secure_hash( $y_0$ )
<p><b>Solve:</b> <math>x_0 \neq y_0</math> and secure_hash(<math>x_0</math>) == secure_hash(<math>y_0</math>)</p> <p>Use concrete state: replace <math>y_0</math> by 7.</p>		

At the second condition, the output of foo(22) and foo(7) is different, so we take the “false” branch and add the symbolic constraint secure\_hash( $x_0$ ) != secure\_hash( $y_0$ ) to the path condition.

In order to take the “true” branch of the second condition, we need to find a satisfying assignment to the path condition with the most recently added constraint negated: that is, we need to find  $x_0$  and  $y_0$  with the same secure\_hash but so that  $x_0 \neq y_0$ . Finding such a pair of inputs -- called a collision -- is a hard problem for cryptographically secure hashes, so our solver is likely not going to be able to find them.

It will first start by trying to simplify the constraint by inserting a concrete value for one of the inputs: in this case, 7 for  $y_0$ .

## 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;
}

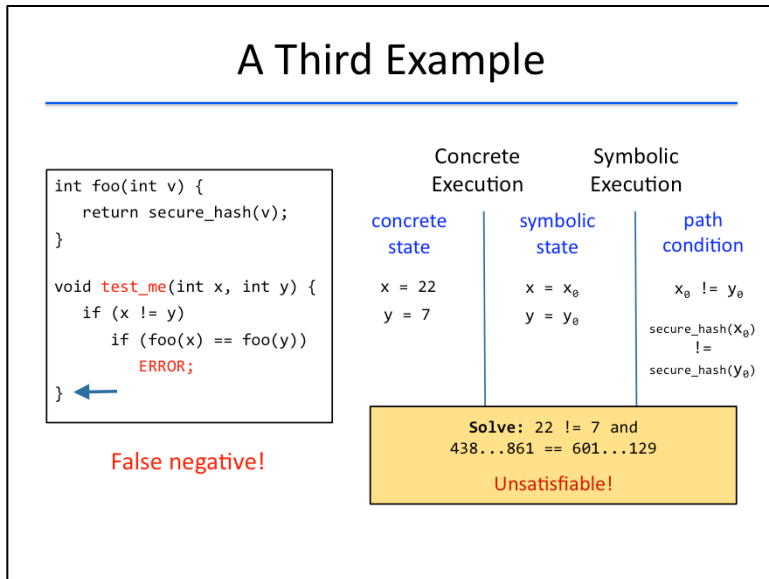
```

Concrete Execution	Symbolic Execution
<b>concrete state</b> x = 22 y = 7	<b>symbolic state</b> x = $x_0$ y = $y_0$
	<b>path condition</b> $x_0 \neq y_0$ secure_hash( $x_0$ ) != secure_hash( $y_0$ )

**Solve:**  $x_0 \neq 7$  and  
 secure\_hash( $x_0$ ) == 601...129  
 Use concrete state: replace  $x_0$  by 22.

The constraint has been partially simplified, but we are left with a similarly hard problem for a secure hash function: finding an input with a specified output. We know taking  $x_0 = 7$  would work, but we can't choose 7 because of the second constraint that  $x_0 \neq 7$ . So DSE will use the other concrete value in its repertoire in an attempt to simplify the condition: plugging in 22 for  $x_0$ .

## A Third Example



Now the constraint is entirely concrete, with no symbolic quantities left. However, as it stands, it is unsatisfiable, because the two large numbers in the equality condition are different. In this case, the solver would declare the constraint unsatisfiable and ignore the branch that satisfying the constraint would have led to.

This means that DSE would not find the error in the code, as the branch it lies on is considered to be unreachable. In this example, DSE has returned a false negative: it has failed to find the error in the code.

The difference between dynamic symbolic execution and “pure” symbolic execution is therefore similar to the difference between dynamic and static analysis. Dynamic analysis will never model a run of the code that could not actually occur, so it will never return false positives: in other words, dynamic analysis is complete. But it can miss actual runs of the code that lead to errors, so it is not sound.

In contrast, symbolic execution on its own will always take a branch that it isn’t sure cannot be reached. So it may model runs of the program that could never happen, sometimes returning spurious errors (hence it is incomplete), but it will take all reachable branches as well, so it will never incorrectly declare a program to be error-free (hence it is sound).



## 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 SLIDE}

So far we've focused on example programs with finite computation trees. However, what properties does DSE exhibit in general when considering programs with possibly infinite computation trees?

- DSE is guaranteed to terminate.
- DSE is complete: if it ever reaches an error, the program can indeed 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.

Select all the statements that are true of DSE applied to such programs.

## 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.

### {SOLUTION SLIDE}

Remember that the undecidability of the halting problem implies that no program analysis can have all three of these properties, so at least one of these statements will be incorrect.

In an general program with unbounded loops, DSE (as we've described it in this lesson) is not guaranteed to terminate. As an exercise, try to construct a short program that DSE would be unable to finish its analysis on. Note however that we could make DSE always terminate by specifying an arbitrary stopping condition (e.g., go no deeper than 50 branch points).

DSE, despite being based on an incomplete theorem-proving strategy, turns out to be complete. Any error it reaches corresponds to the execution of the program on a concrete input, so the error can be reproduced by running the program on that same input.

On the other hand, DSE is not sound. As we saw in the third example, even on finite computation trees, the solver may fail to identify a solution to a given set of constraints, leading DSE not to take a potential program path that would lead to an error.

## 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;
}
```

So far we have seen DSE's usefulness in the context of testing functions as units. Now let's take a look at how DSE could be used when the unit of test is a data structure.

Previously, the tools we have seen to produce tests in this context are Korat and Randoop. We could also use random testing for data structures, but the same problem inherent to random testing still occurs: an error could be difficult to reach via randomness alone.

In this example, we have a data structure which models a linked list in C++-like syntax. We define the type "cell" to consist of an integer field named `data` and a pointer to another cell, named `next`. We next define the function `foo` which takes an integer  $v$  as its argument and returns the integer  $2*v + 1$ . Finally, we define the function `test_me`, which takes an integer  $x$  and a pointer to a cell called  $p$ , and does four nested if-checks:

```
if  $x > 0$ 
if  $p \neq \text{NULL}$ 
if  $\text{foo}(x) == p \rightarrow \text{data}$ , and
if  $p \rightarrow \text{next} == p$ 
```

If all four of these conditions are true, then the function throws an error. Otherwise, the function returns 0.

Here is what a typical random test driver would do. It would generate a random value of  $x$  and a random memory graph (filling in random values for the data field in each node) reachable from an initial pointer  $p$  to give to `test_me`. The probability that even the third condition,  $\text{foo}(x) == p \rightarrow \text{data}$ , is true is extremely small (in fact, 0 if  $p \rightarrow \text{data}$  is even). So it's highly unlikely that the error in this function would be caught by a random tester.

## 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 = 236  
p = NULL

Symbolic  
Execution

symbolic  
state

x =  $x_0$   
p =  $p_0$

path  
condition

Dynamic symbolic execution, on the other hand, would find this error after at most five runs of the test\_me function.

For example, suppose the randomly generated inputs first given to test\_me are  $x = 236$  and  $p = \text{NULL}$ . As before, DSE stores both the concrete values of these variables and their symbolic values, which we'll call  $x_0$  and  $p_0$ .

## 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 = 236  
p = NULL

Symbolic  
Execution

symbolic  
state

x =  $x_0$   
p =  $p_0$

path  
condition

$x_0 > 0$

These concrete values lead DSE to take the “true” branch for the first condition,  $x > 0$ , so it adds the symbolic constraint  $x_0 > 0$  to the 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

concrete  
state

x = 236  
p = NULL

Symbolic  
Execution

symbolic  
state

x =  $x_0$   
p =  $p_0$

path  
condition

$x_0 > 0$   
 $p_0 == \text{NULL}$

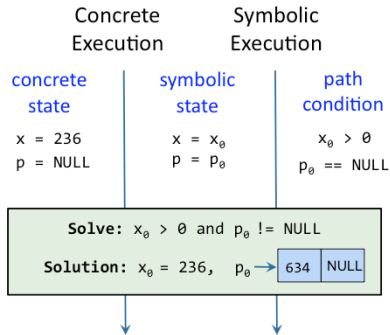
But, because p is NULL, the condition  $p \neq \text{NULL}$  evaluates to false, and the function returns 0. DSE stores the negation of this condition as  $p_0 == \text{NULL}$  in its path condition.

## 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;
}
```



As always, since the function has terminated, DSE will negate the most recently-stored constraint in the path condition and then pass the conjunction of all the constraints in the resulting path condition to the solver.

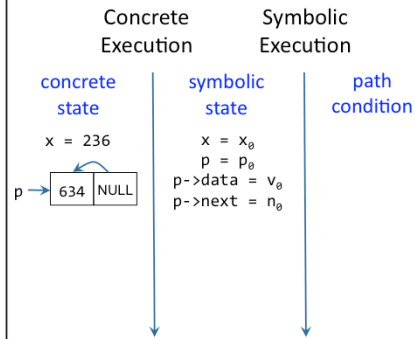
The solver will attempt to find values for  $x_0$  and  $p_0$  satisfying  $x_0 > 0$  and  $p_0 \neq \text{NULL}$ . In this case, the solver will need to allocate memory for a cell data structure and then generate values for the members of that cell. A satisfying assignment in this case might be:  $x_0 = 236$ ,  $p_0 \rightarrow \text{data} = 634$ , and  $p_0 \rightarrow \text{next} = \text{NULL}$ .

## 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;
}
```



This forms the new concrete state for the next run of the `test_me` function. The symbolic state is expanded as well with the symbolic values  $v_0$  (assigned to `p->data`) and  $n_0$  (assigned to `p->next`).

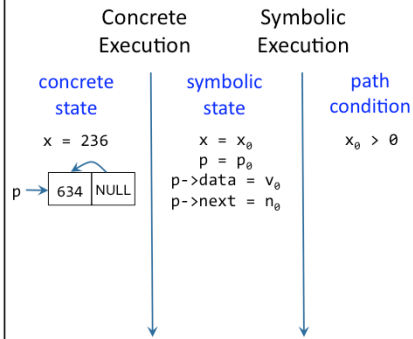


## Data-Structure Example

```
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;
}
```



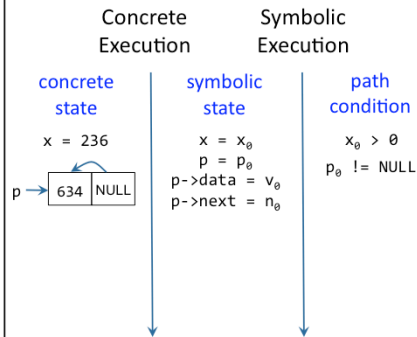
The first condition again evaluates to true, so the symbolic constraint `x0 > 0` is added to the path condition.

## 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;
}
```



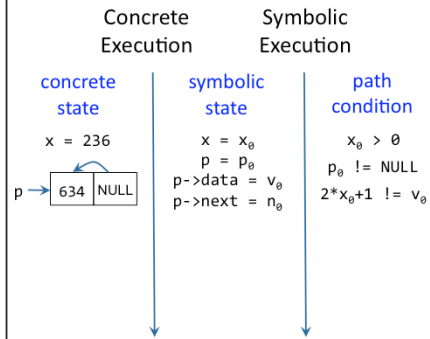
This time, the second condition, `p != NULL`, also evaluates to true, so the symbolic constraint `p_0 != NULL` is added to the path condition.

## 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;
}
```



But the third condition,  $\text{foo}(x) == p \rightarrow \text{data}$ , evaluates to false, so the symbolic constraint  $2*x_0 + 1 \neq v_0$  is added to the path condition.

## Data-Structure Example

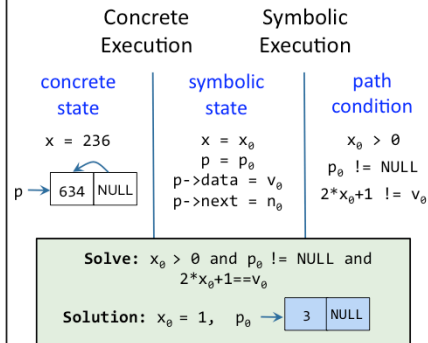
```

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;
}

```



DSE then passes the path condition (with  $2x_0 + 1 \neq v_0$  negated) to the solver to attempt to find inputs that will satisfy the third branch condition.

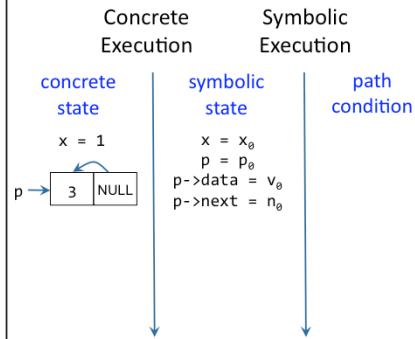
The solver might come up with the following: changing  $x_0$  to 1 and  $v_0$  to 3, and otherwise leaving the inputs the same.

## 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;
}
```



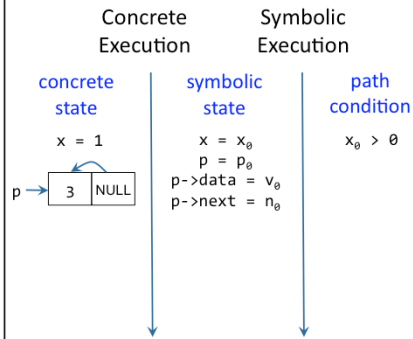
DSE will then run `test_me` again with the new input values,

## 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;
}
```



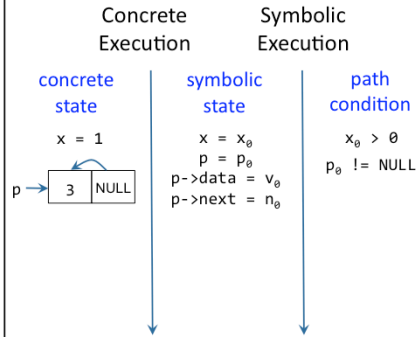
adding the appropriate symbolic constraints to the path condition as each branch condition is evaluated.

# 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;
}
```



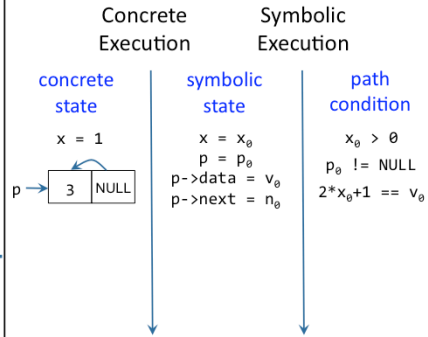
[no text]

# 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;
}
```



[no text]



## Data-Structure Example

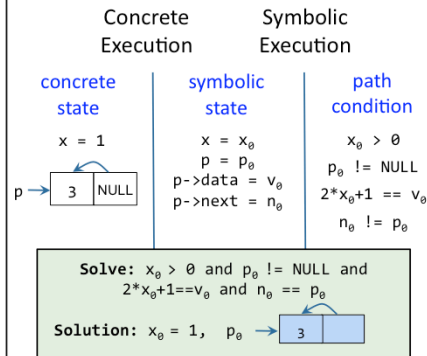
```

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;
}

```



Eventually, the fourth condition,  $p \rightarrow \text{next} == p$ , evaluates to false, so the symbolic constraint  $n_0 \neq p_0$  is added to the path condition.

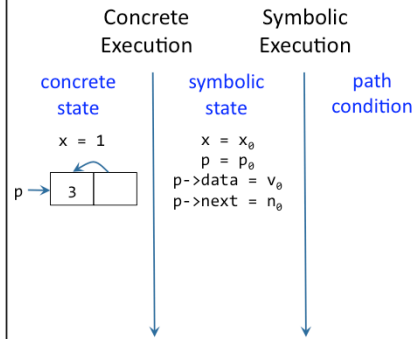
Negating this most recently added constraint, the solver then attempts to construct inputs satisfying the constraints that  $x_0 > 0$ ,  $p_0 \neq \text{NULL}$ ,  $2 * x_0 + 1 == v_0$ , and  $n_0 == p_0$ . In this case, it would just set  $p_0 \rightarrow \text{next}$  to point to the same place as  $p_0$ .

## 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;
}
```



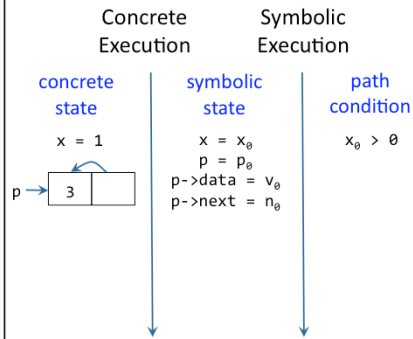
Now DSE takes one more stroll through the `test_me` function.

## 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;
}
```



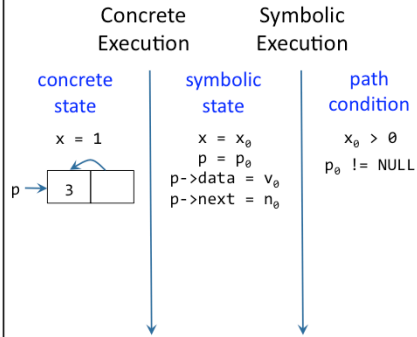
Each of the branch conditions for these inputs evaluates to true.

# 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;
}
```



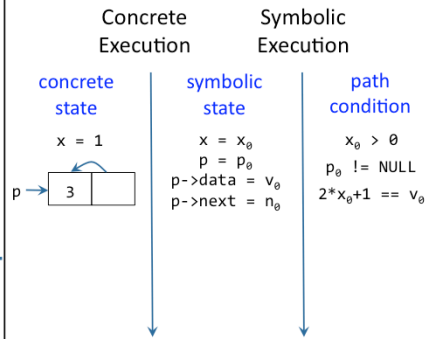
[no text]

# Data-Structure Example

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

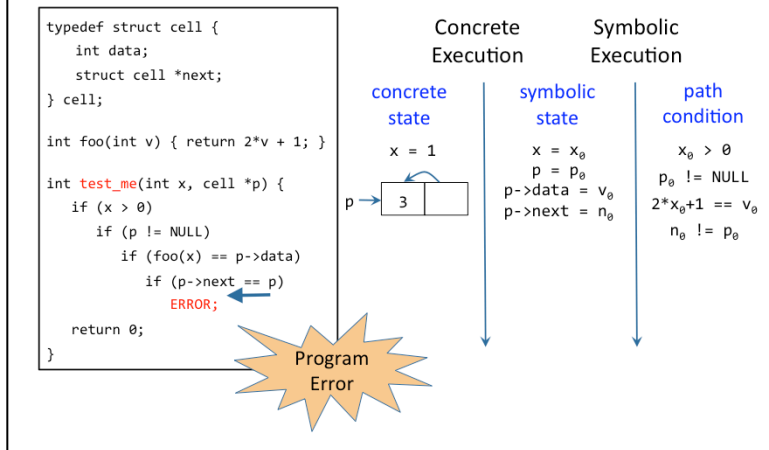
int foo(int v) { return 2*v + 1; }

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;
}
```



[no text]

## Data-Structure Example



And, finally, the program's error is triggered, so DSE has identified a particular concrete input that triggers the error.

## 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

Dynamic symbolic execution is a hybrid approach to software testing that attempts to strike a balance between the costs and benefits of dynamic and static analysis. As you saw, it generates concrete inputs one-by-one such that each input takes a different path through the program's computation tree. And it executes the program both concretely and symbolically.

These two types of execution cooperate with each other. On the one hand, the concrete execution guides the symbolic execution. By replacing symbolic expressions with concrete values if the symbolic expressions become too complex, the concrete execution enables DSE to overcome the incompleteness of the theorem prover.

On the other hand, the symbolic execution allows DSE to generate new concrete inputs for the next execution of the program. This increases the coverage potential of DSE over other dynamic analyses such as pure random testing.

## QUIZ: Characteristics of DSE

---

- 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

{QUIZ SLIDE}

Now that you've seen how DSE works, take some time to synthesize what you have learned by answering these questions about the characteristics of DSE. Choose the best answer for each question. As you answer, compare and contrast your answers with the characteristics of previous types of analyses.

The testing of DSE is best described as which of the following?

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

The input search strategy DSE uses is:

- Randomized
- Systematic

What is the sensitivity of DSE to program structure?

- It is flow-insensitive
- It is flow-sensitive but not path-sensitive
- It is path-sensitive

Which of these best describes the instrumentation performed in DSE?

- Sampled
- Non-sampled



## QUIZ: Characteristics of DSE

---

- 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

### {SOLUTION SLIDE}

In the landscape of testing techniques, there are two separate spectra: automated versus manual and black-box versus white-box. The quadrant of the landscape in which DSE falls is automated & white-box. It is an algorithmic technique for deriving inputs leading to programming errors, so it is certainly an automated technique. Moreover, it requires access to the program's code, so it is unequivocally a white-box technique.

Let's look at the second question. Different automated tools have different strategies for searching the space of inputs for error-producing inputs. Examples of randomized searches include Randoop, Monkey, and Cuzz, whereas Korat is an example of systematic or enumerative search. DSE is also an example of a systematic search: even though its first input is randomly generated, all remaining inputs are derived by systematically solving constraints relevant to the program's computation tree.

Third, by its very nature, DSE is a path-sensitive static analysis. The basis of its operation requires distinguishing between different paths in a program's computation tree.

Finally, the instrumentation in DSE is non-sampled, as it is in the case of Korat preconditions. An example of sampled instrumentation is statistical debugging wherein the runtime overhead of tracking all instrumented predicates without sampling is prohibitive.

## 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)

Now that you've learnt how DSE works, let's look at a few real-world examples where DSE has been applied.

In a case study, DSE found two bugs in version 1.0.1 of SGLIB, a data structure library for C that was inspired by the Standard Template Library from C++. Both the bugs were reported to the authors of the library who fixed them in version 1.0.2

The first bug, in the doubly-linked list library, is a segmentation fault that occurs when a non-zero length list is concatenated with a zero-length list. This bug was discovered in 140 iterations in under 1 second. This bug is easy to fix by putting a check on the length of the second list in the concatenation function.

The second bug, which is a more serious one, was discovered in the hash-table library in 193 iterations in 1 second. Specifically, DSE constructed a valid sequence of function calls which gets the hash-table library's `is_member` function into an infinite loop.

## 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

This table shows, for each data structure that SGLIB implements, the time that DSE took to test the data structure in seconds, the number of iterations that DSE made, the number of branches it executed, the branch coverage it obtained, the number of functions it executed, and the number of bugs that it found.

Notice that the branch coverage in most cases is very high, approaching 100%. The authors of the case study investigated the few branches that weren't covered, and found that most of them were in fact unreachable.

You can read more about this case study in a technical paper linked from the instructor notes.

<http://mir.cs.illinois.edu/marinov/publications/SenETAL05CUTE.pdf>

## 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

In another case study, DSE was applied to test a C implementation of a security protocol: the Needham Schroeder public key protocol. This protocol is known to be vulnerable to a man-in-the-middle attack. The implementation comprised 600 lines of code. It took DSE fewer than 13 seconds on a machine with a 1.8 GHz CPU and 2 GB of RAM to discover this attack. In contrast, a software model checker Verisoft that is suited for testing such protocols by using a state-space exploration technique took 8 hours.

## Realistic Implementations

---

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

The remarkable success of dynamic symbolic execution has led to open-source as well as commercial implementations of the technique for virtually all mainstream high-level and low-level languages. Here is a listing of a few of these implementations:

- KLEE based on the LLVM compiler which supports the C family of languages including C, C++, Objective C, and Objective C++.
- PEX for applications written using Microsoft's .NET framework
- jCUTE for Java programs
- Jalangi for Javascript programs, and
- SAGE and S2E for binaries on common architectures such as X86 and 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)

As we illustrated using a series of examples in this lesson, DSE is useful for testing small units of code. But it has also been applied to test entire, large, complex programs. Let's look at one of the most successful case studies in this category: the SAGE tool developed by Microsoft.

SAGE is an acronym for scalable automated guided execution. To date, it has discovered many expensive security bugs in many Microsoft applications such as Windows and Office. It is used daily in various Microsoft groups and runs continuously on 100s of machines.

What makes SAGE so useful? There are several reasons.

First, it works on large applications, not just small units. So it can detect bugs due to problems across components.

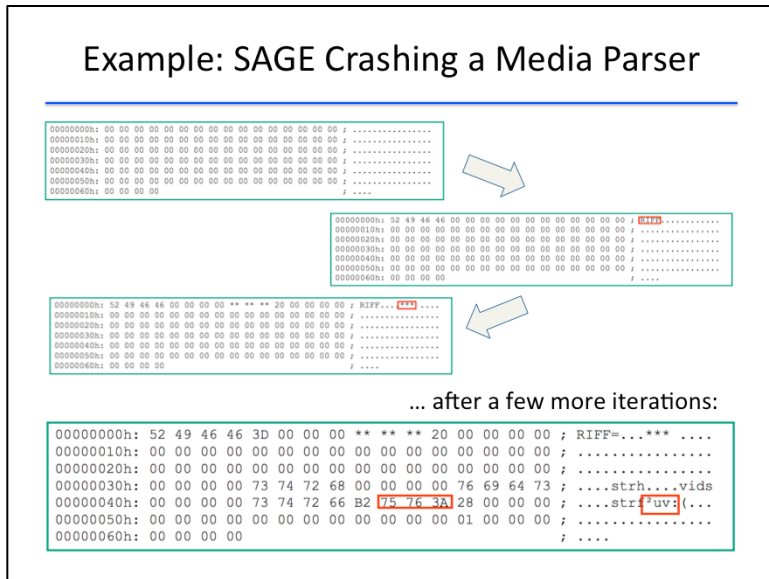
Second, it focuses on fuzzing input files, which are a typical kind of input to many applications. For instance, a typical input to a web browser application is an HTML file. This in turn enables SAGE to be fully automated: for instance, a user need not specify the input format of the application.

Third, SAGE works on x86 binaries, making it easy to deploy: in particular, it is not dependent on the programming language or build process used by the application.

You can read more about SAGE by following the links in instructor notes.

[http://research.microsoft.com/en-us/um/people/pg/public\\_psfiles/ndss2008.pdf](http://research.microsoft.com/en-us/um/people/pg/public_psfiles/ndss2008.pdf)  
[http://research.microsoft.com/en-us/um/people/pg/public\\_psfiles/talk-spin2009.pdf](http://research.microsoft.com/en-us/um/people/pg/public_psfiles/talk-spin2009.pdf)

## Example: SAGE Crashing a Media Parser



Let's look at an example of how SAGE is able to crash a real media parser application.

SAGE begins with an input media file that has 100 zero bytes. The contents of each byte are indicated by two adjoining zeros [point to the 00 00 00 portion of the file]. A human-readable form of each byte is also shown here on the right [point to the ..... portion of the file].

In each successive iteration, SAGE replaces a subset of these bytes with characters that it obtains by solving the path constraint of an execution of the media parser.

For instance, in the second iteration, it replaces the first four bytes by the characters RIFF respectively, and in the third iteration, it replaces the 9th, 10th, 11th, and 12th bytes by the characters \*, \*, \*, and black respectively. After a few more iterations, it generates the input file that crashes the application.

In 60 machine hours, SAGE is able to automatically find 357 such crashes corresponding to 12 unique bugs in this media parser application.

## 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

Now that the lesson is coming to a close, let's review what we have learned about dynamic symbolic execution.

Symbolic execution is a technique for simulating the execution of a program on symbolic inputs. It tracks symbolic constraints over such inputs to decide whether certain paths of computation are possible. Dynamic symbolic execution, or DSE, is a hybrid between symbolic execution and concrete execution that overcomes limitations of using either of those approaches alone.

DSE systematically generates numeric and pointer inputs in order to explore a program's computation tree with as much coverage as possible while eliminating redundant executions. (Recall that the computation tree is a model of all possible paths that a program's execution can take.) The goal of DSE is to determine if an error is reachable under some input to the program.

DSE simultaneously tracks three pieces of information: the program's current concrete state, the program's current symbolic state, and the symbolic constraints for the execution so far, called the path condition. It uses the dynamic, concrete state to simplify the static analysis part of constraint-solving, and it uses the static, symbolic state to guide the dynamic analysis part of selecting non-redundant concrete inputs to exercise next. In this way, it is a hybrid between dynamic and static analysis.

Finally, DSE is complete: if it reports an error, it is certain that the error can be reached on some run of the program (in fact, DSE can report the exact inputs to generate the error). However, unlike pure symbolic execution, DSE has no guarantee of soundness: it might fail to report an error in a program. Additionally, as we've discussed in this lesson, DSE is not guaranteed to terminate in the presence of input-dependent loops, as these loops may unroll into infinitely many paths in the computation tree. However, we can modify DSE to terminate after exploring a finite number of paths in the computation tree, giving up soundness in the process.