Vulnerability Detection - Symbolic Execution

Holistic Software Security

Aravind Machiry

Credits: Prof. Mayur Naik, UPenn

Why is fuzzing inadequate?

- Generating highly constrained inputs could take long time!

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

Probability of generating input that triggers ERROR:

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

Symbolic Execution (SymEx)

- A technique to explore a program systematically:
    - Symbolically execute all the paths in the program.
    - At each path check for possible error conditions.

Symbolic Execution (SymEx)

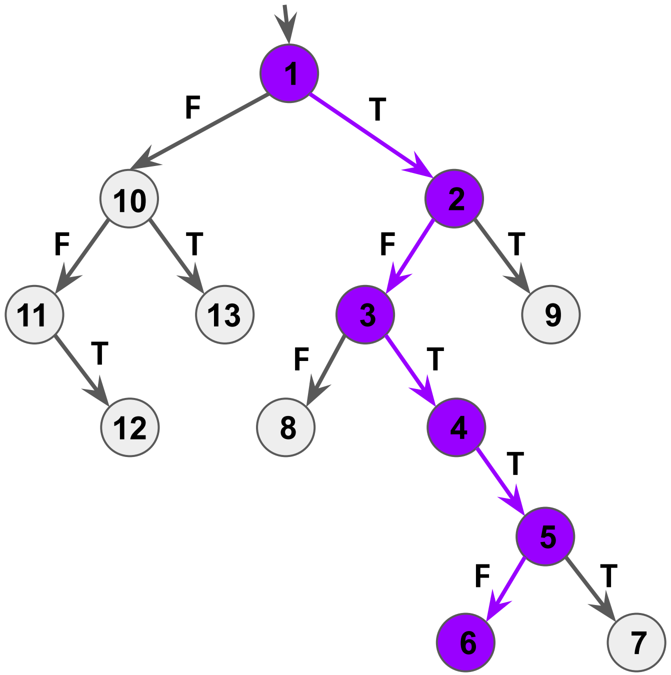
- Use symbolic values for inputs
- Execute program symbolically on symbolic input values:
    - Collect symbolic path constraints
- Use constraint solver to check if a path is feasible or to generate concrete inputs.
- **Unlike classic static analysis, symbolic execution:**
    - **Enables us to generate concrete inputs.**
    - **Can made to only explore feasible paths (states).**

SymEx Impact

We also used KLEE as a bug finding tool, applying it to 452 applications (over 430K total lines of code), where it found 56 serious bugs, including three in COREUTILS that had been missed for over 15 years. Finally, we used KLEE to cross-check purportedly identical BUSYBOX and COREUTILS utilities, finding functional correctness errors and a myriad of inconsistencies.

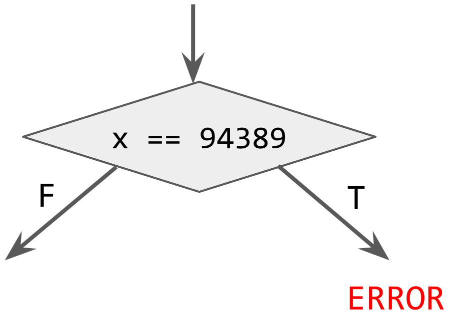
Execution Paths

- A path in the control flow graph (CFG) of a function.
- Edge represents a condition being taken.
- How many paths does a program with loops has?



Execution Paths: Example

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



Symbolic Execution: 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;
}
```

Symbolic Execution

	Path1
Symbolic State	$x = x_0$ $y = y_0$
Path Condition	

Symbolic Execution: 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;
}
```

Symbolic Execution

	Path1
Symbolic State	$x = x_0$ $y = y_0$ $z = 2*y_0$
Path Condition	

Symbolic Execution: 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)        Path2
         ERROR;
}        Path1
```

Symbolic Execution

	Path1	Path2
Symbolic State	$x = x_0$ $y = y_0$ $z = 2*y_0$	$x = x_0$ $y = y_0$ $z = 2*y_0$
Path Condition	$2*y_0 \;!= x_0$	$2*y_0 == x_0$

Symbolic Execution: 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)      <-- Path2
         ERROR;
}
```

Symbolic Execution

	Path1	Path2
Symbolic State	x = x_0 y = y_0 z = $2*y_0$	x = x_0 y = y_0 z = $2*y_0$
Path Condition	$2*y_0$!= x_0	$2*y_0$ == x_0
	END	

Symbolic Execution: 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;          Path3
}          Path2
```

Symbolic Execution

	Path1	Path2	Path3
Symbolic State	$x = x_0$ $y = y_0$ $z = 2*y_0$	$x = x_0$ $y = y_0$ $z = 2*y_0$	$x = x_0$ $y = y_0$ $z = 2*y_0$
Path Condition	$2*y_0 \mathrel{!=} x_0$ **END**	$2*y_0 == x_0$ $x_0 <= y_0 + 10$	$2*y_0 == x_0$ $x_0 > y_0 + 10$

Symbolic Execution: 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;      <-- Path3
}
```

Symbolic Execution

	Path1	Path2	Path3
Symbolic State	$x = x_0$ $y = y_0$ $z = 2*y_0$	$x = x_0$ $y = y_0$ $z = 2*y_0$	$x = x_0$ $y = y_0$ $z = 2*y_0$
Path Condition	$2*y_0 \;!= x_0$	$2*y_0 == x_0$ $x_0 <= y_0 + 10$	$2*y_0 == x_0$ $x_0 > y_0 + 10$
	END	END	ERROR

Symbolic Execution: 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;      <- Path3
}
```

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

Input causing error: $x_0 = 30$, $y_0 = 15$

Symbolic Execution

	Path1	Path2	Path3
Symbolic State	$x = x_0$ $y = y_0$ $z = 2*y_0$	$x = x_0$ $y = y_0$ $z = 2*y_0$	$x = x_0$ $y = y_0$ $z = 2*y_0$
Path Condition	$2*y_0 \; != x_0$	$2*y_0 == x_0$ $x_0 <= y_0 + 10$	$2*y_0 == x_0$ $x_0 > y_0 + 10$
	END	END	ERROR

Symbolic Execution: Drawbacks

- Loops => Infinite paths.
- Can explore infeasible paths.
- Bugs should be converted into asserts.
- Does not scale for real-world programs.

```
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!

Symbolic Execution: Practical problems

- Memory model.

```
// Consider i and j as symbolic
arr[i]++;
arr[j] = arr[j] + 1;
```

Which memory cell to update?

- Symbolically sized memory.

```
// Consider i as symbolic
p = malloc(i*sizeof(int));
```

What should be the size of p?

- External or Library functions.

```
// Consider argv[1] as symbolic
j = atoi(argv[1]);
```

We do not have source code of library. How should we handle passing arguments?

Dynamic Symbolic Execution (DSE)

- Random testing or fuzzing cannot generate highly constraint inputs.
- Can we use symbolic execution to generate these constrained inputs?
- Also called Concolic execution.

DSE Approach

1. Start with random input values.

1. Keep track of both concrete values and symbolic constraints.

1. Use concrete values to simplify symbolic constraints.

DSE 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		$y = y_0$	

DSE 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	$z = 2*y_0$	

DSE 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		$x = x_0$	$2*y_0 \neq x_0$
	y = 7		$y = y_0$	
	z = 14		$z = 2*y_0$	

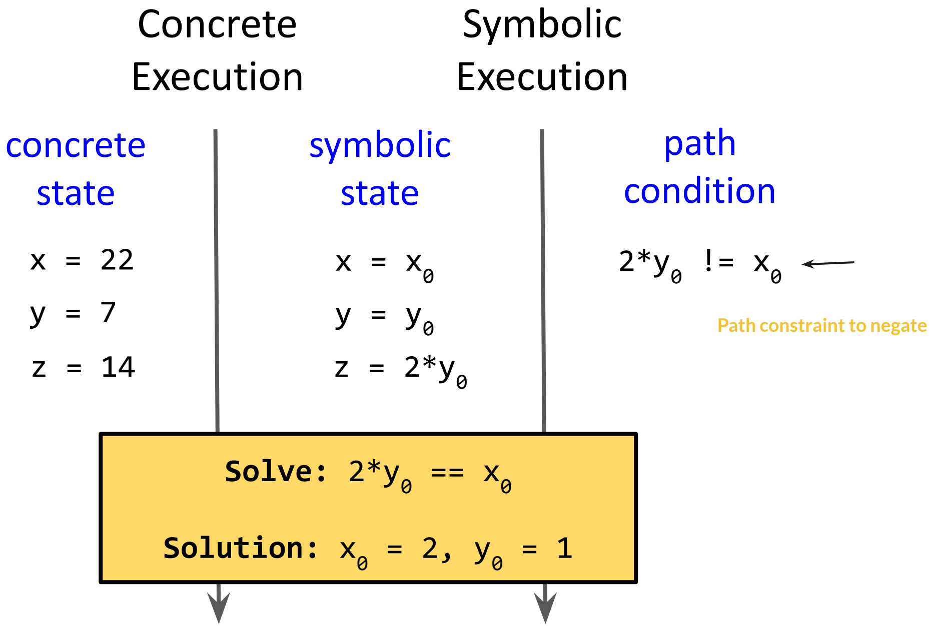
DSE Example

Now to generate new input => Negate the path condition and solve to get inputs that will make the program execute another path.

DSE 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 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	Concrete Execution	symbolic state	Symbolic Execution	path condition
x = 2		$x = x_0$		
y = 1		$y = y_0$		

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

DSE 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	$z = 2*y_0$		

DSE 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		$z = 2*y_0$		

DSE 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	$z = 2*y_0$	

Path constraint to negate

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

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

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

DSE 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

x = 30

y = 15

z = 30

symbolic state

$x = x_0$

$y = y_0$

$z = 2*y_0$

path condition

DSE 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	$z = 2*y_0$	

DSE 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	$x_0 > y_0+10$
z = 30	z = $2*y_0$	

DSE 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;
}
```

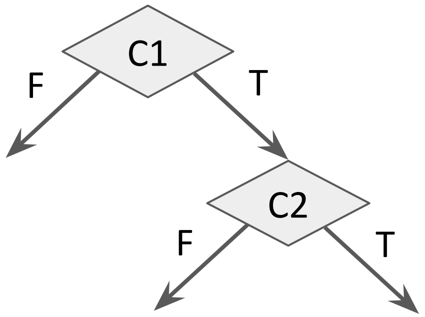
Program Error

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

DSE Constraints

Which of the following constraints DSE might possibly solve in exploring the computation tree shown below:

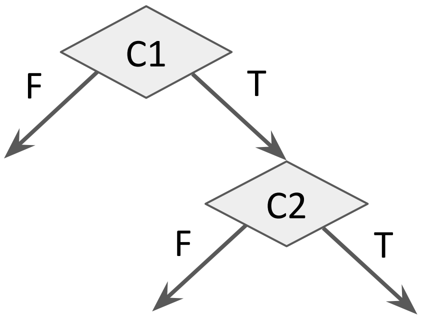
- [ ] C1
- [ ] C2
- [ ] ¬C1
- [ ] ¬C2
- [ ] C1 ∧ C2
- [ ] C1 ∧ ¬C2
- [ ] ¬C1 ∧ C2
- [ ] ¬C1 ∧ ¬C2



DSE Constraints

Which of the following constraints DSE might possibly solve in exploring the computation tree shown below:

- [x] C1
- [ ] C2
- [x] ¬C1
- [ ] ¬C2
- [x] C1 ∧ C2
- [x] C1 ∧ ¬C2
- [ ] ¬C1 ∧ C2
- [ ] ¬C1 ∧ ¬C2



DSE Example 2

```
int foo(int v) {
   return 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	$y = y_0$		

DSE Example 2

```
int foo(int v) {
   return 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

x = 22

y = 7

z = 601...129

symbolic state

$x = x_0$

$y = y_0$

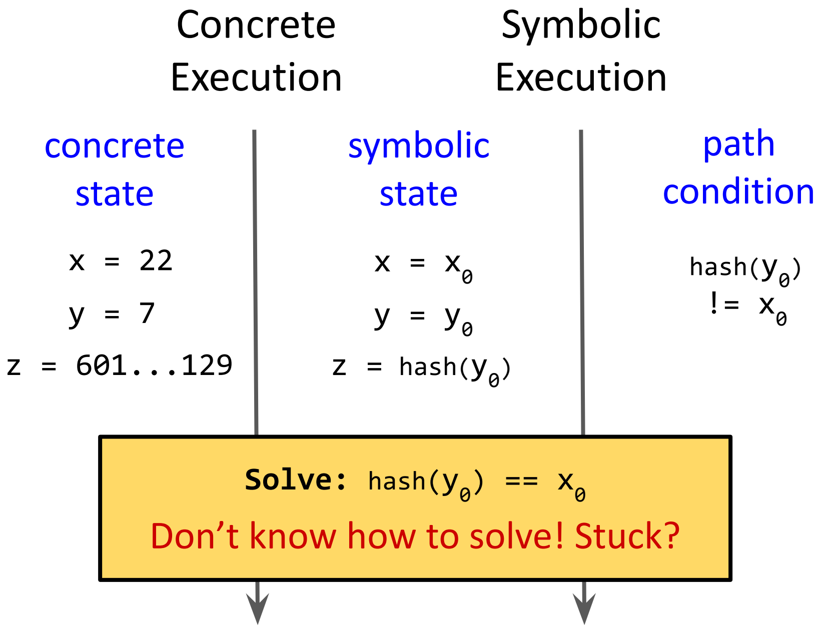
$z = hash(y_0)$

path condition

DSE Example 2

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

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



DSE Example 2

```
int foo(int v) {
   return 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		hash(y_0) != x_0
y = 7		y = y_0		
z = 601...129		z = hash(y_0)		

Solve: hash(y_0) == x_0

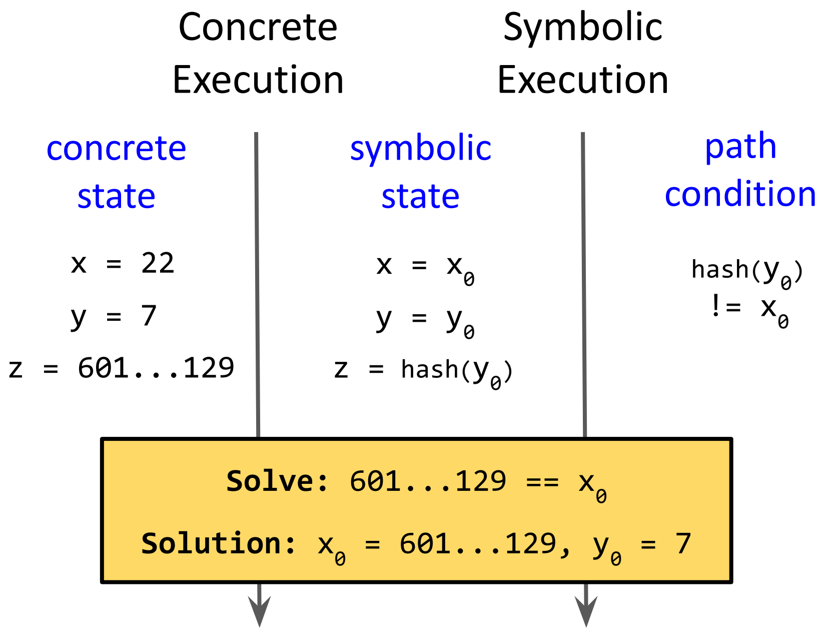
Don't know how to solve! Stuck?

Not stuck! Use concrete state: replace y_0 by 7

DSE Example 2

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

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



DSE Example 2

```
int foo(int v) {
   return 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	$y = y_0$	

DSE Example 2

```
int foo(int v) {
   return 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	$z = hash(y_0)$	

DSE Example 2

```
int foo(int v) {
   return 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		$x = x_0$	$hash(y_0) == x_0$
y = 7		$y = y_0$	
z = 601...129		$z = hash(y_0)$	

DSE Example 2

```
int foo(int v) {
    return 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	$x = x_0$	$hash(y_0) == x_0$
y = 7	$y = y_0$	$x_0 > y_0+10$
z = 601...129	$z = hash(y_0)$	

DSE Example 3

We may not be able to generate inputs for all the paths.

DSE Example 3

```
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	$y = y_0$		

DSE Example 3

```
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		$y = y_0$		

DSE Example 3

```
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)

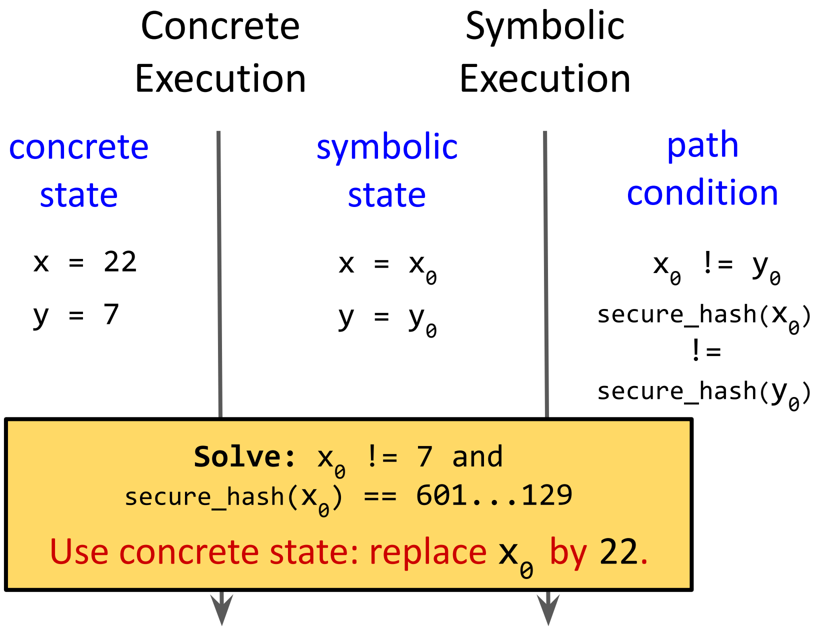
Solve: x_0 != y_0 and secure_hash(x_0) == secure_hash(y_0)

Use concrete state: replace y_0 by 7.

DSE Example 3

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

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



DSE Example 3

```
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!

DSE v/s SymEx

- Similar to SymEx, DSE may not terminate.

- Unlike SymEx:
    - DSE has false negatives, i.e., might not be able to execute all paths.
    - DSE will always generate concrete inputs.

SymEx/DSE tools

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

SymEx based Vulnerability Detection

- **ucklee:** Under constrained symbolic execution.
  - Symbolically execute arbitrary functions.

user input. We evaluate the checkers on over 20,000 functions from BIND, OpenSSL, and the Linux kernel, find 67 bugs, and verify that hundreds of functions are leak free and that thousands of functions do not access uninitialized data.

SymEx based Vulnerability Detection

- SAGE = Scalable Automated Guided Execution
- Found many expensive security bugs in many Microsoft applications (Windows, Office, etc.)
- Used daily in various Microsoft groups, runs continuously in the cloud
- 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 upon programming language or build process)
  - Target-Driven Compositional Concolic Testing

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                                     ; ....
```

Using SymEx/DSE

- *Blindly using symbolic execution is inefficient and does not scale well for real programs.*
- Lot of massaging happens before using symex on real-programs:
  - https://adalogics.com/blog/symbolic-execution-with-klee
- Using symex smartly is an art, Examples:
  - Force symex to execute certain interesting paths:
    - BootStomp: On the Security of Bootloaders in Mobile Devices (USENIX 2017).
  - Try to concolically execution individual functions:
    - Target-Driven Compositional Concolic Testing (FSE 2019).
  - Execute only interesting parts of a program:
    - Chopped Symbolic Execution (ICSE 2018).

SymEx/DSE Trends

- May works use symex/DSE as an auxiliary technique:
  - Driller/Vuzzer: Uses concolic execution to assist fuzzer.
  - Sys: Filter out false positives.
- Applying SymEx to different domains:
  - Symbolic execution of smart contracts.
  - Symbolic execution of RTL code.
- Making SymEx/DSE fast:
  - Qsym: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing
  - SymCC: Compiling concolic execution engine into the program.
  - Neuro Symbolic Execution: Augmenting symbolic execution with neural nets.

SymEx/DSE Final Remarks!

- Symbolic execution is a powerful technique, which is impractical and cannot work for large real-world programs.

- However, we can be smart and use it opportunistically.