Symbolic Execution

Suman Jana

Acknowledgement: Baishakhi Ray (Uva), Omar Chowdhury (Purdue), Saswat Anand (GA Tech), Rupak Majumdar (UCLA), Koushik Sen (UCB)

What is the goal?

```
static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer signedParams,
                                 uint8_t *signature, UInt16 signatureLen)
{
    OSStatus        err;
    ...

    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
        goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;

    // code ommitted for brevity...

    err = sslRawVerify(ctx,
                       ctx->peerPubKey,
                       dataToSign,                /* plaintext */
                       dataToSignLen,             /* plaintext length */
                       signature,
                       signatureLen);
    if(err) {
        sslErrorLog("SSLDecodeSignedServerKeyExchange: sslRawVerify "
                    "returned %d\n", (int)err);
        goto fail;
    }

fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;

}
```

Oops...

Never gets called (but needed to be)...

Despite the name, always returns "it's OK!!!"

Testing

- Testing approaches are in general manual
- Time consuming process
- Error-prone
- Incomplete
- Depends on the quality of the test cases or inputs
- Provides little in terms of coverage

Yes, we can.

Can

rms of

an we some

e it automatic?

Background: SAT

Given a propositional formula in CNF, find if there exists an assignment to Boolean variables that makes the formula true:

literals

clauses

$$\omega_1 = (b \vee c)$$

$$\omega_2 = (\neg a \vee \neg d)$$

$$\omega_3 = (\neg b \vee d)$$

$$\varphi = \omega_1 \wedge \omega_2 \wedge \omega_3$$

$$A = \{a=0, b=1, c=0, d=1\}$$

SATisfying assignment!

Background: SMT (Satisfiability Modulo Theory)

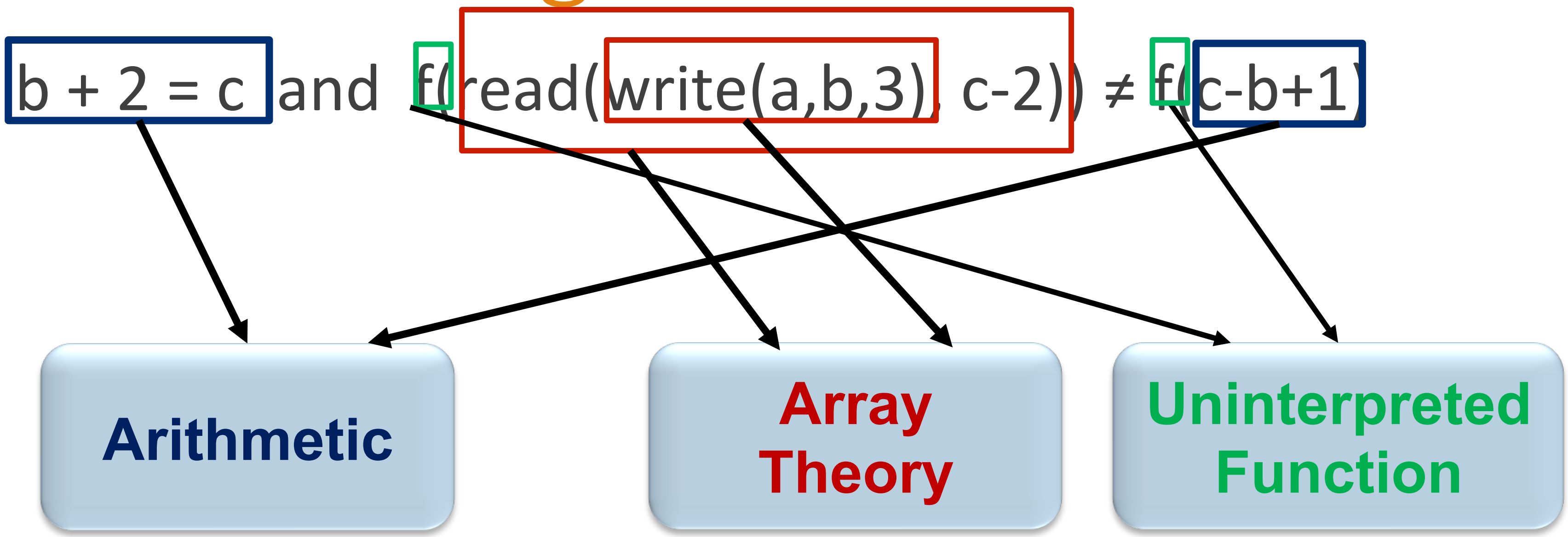
- An SMT instance is a generalization of a Boolean SAT instance
- Various sets of variables are replaced by predicates from a variety of underlying theories.

Input: a **first-order** formula φ over background theory (Arithmetic, Arrays, Bit-vectors, Algebraic Datatypes)

Output: is φ satisfiable?

- does φ have a model?
- Is there a refutation of φ = proof of $\neg\varphi$?

Background: SMT



Example SMT Solving

b + 2 = c and f(read(write(a,b,3), c-2)) ≠ f(c-b+1)

[Substituting c by b+2]

b + 2 = c and f(read(write(a,b,3), b+2-2)) ≠ f(b+2-b+1)

[Arithmetic simplification]

b + 2 = c and f(read(write(a,b,3), b)) ≠ f(3)

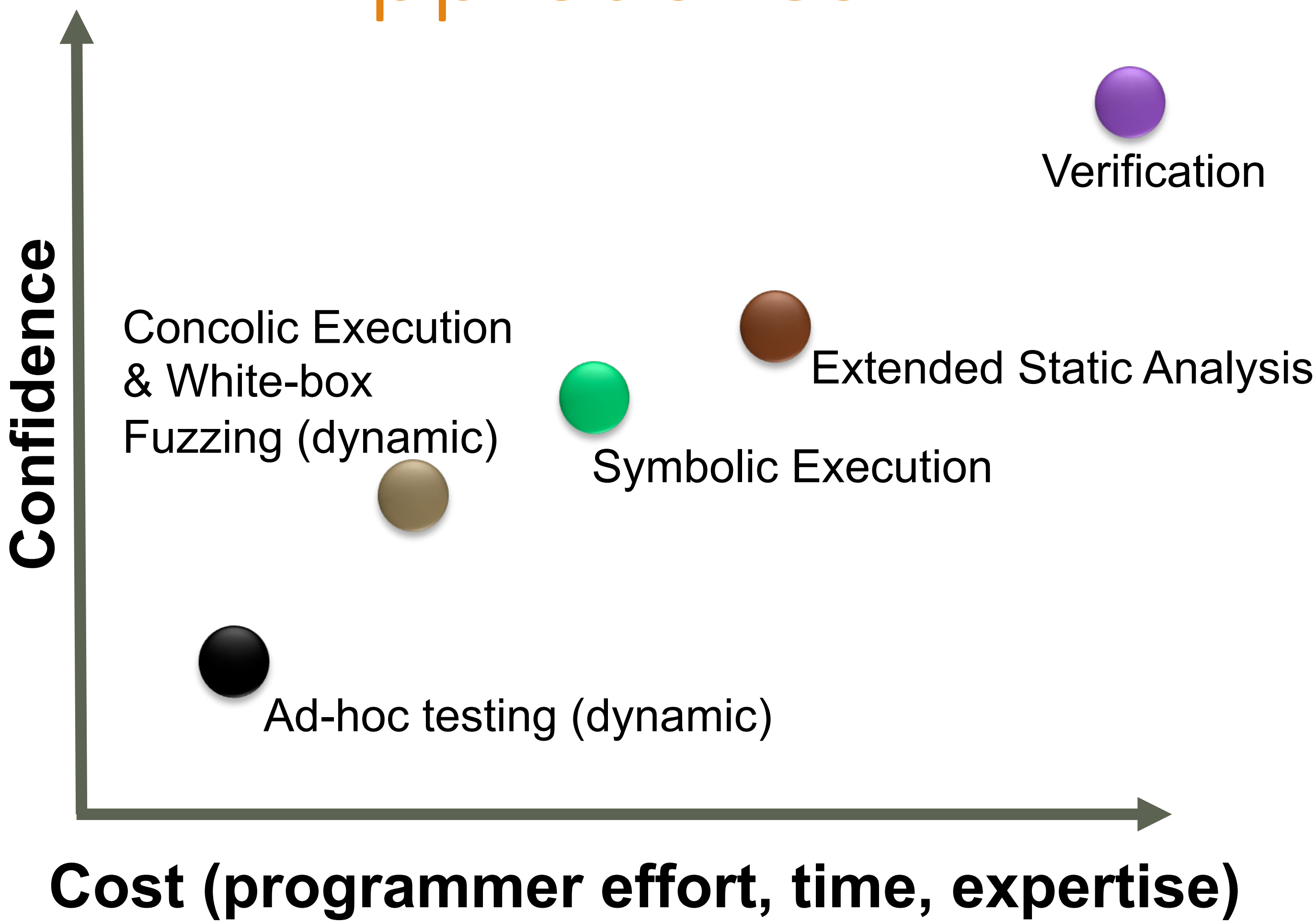
[Applying array theory axiom]

forall a,i,v:read(write(a,i,v), i) = v]

b+2 = c and f(3) ≠ f(3) [**NOT SATISFIABLE**]

read : array × index → element
write : array × index × element → array

Program Validation Approaches



Automatic Test Generation
Symbolic & Concolic Execution

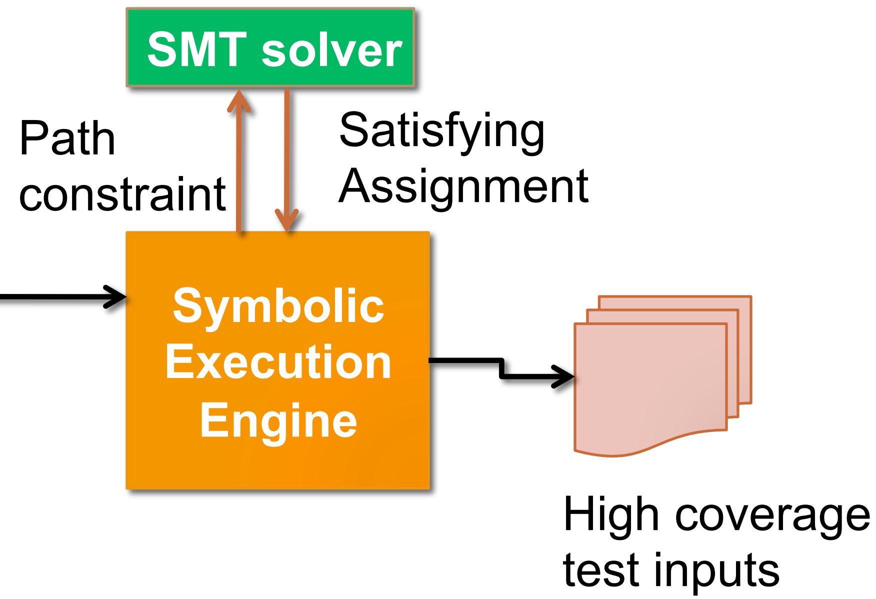
How do we automatically generate test inputs that induce the program to go in different paths?

Intuition:

- Divide the whole possible input space of the program into equivalent classes of input.
- For each equivalence class, all inputs in that equivalence class will induce the same program path.
- Test one input from each equivalence class.

Symbolic Execution

```
Void func(int x, int y){
 int z = 2 * y;
 if(z == x){
        if (x > y + 10)
            ERROR
 }
}
int main(){
 int x = sym_input();
 int y = sym_input();
 func(x, y);
 return 0;}
```



Symbolic Execution

Symbolic Execution

Execute the program with symbolic valued inputs (**Goal: good path coverage**)

Represents *equivalence class of inputs* with first order logic formulas (**path constraints**)

One path constraint abstractly represents all inputs that induces the program execution to go down a specific path

Solve the path constraint to obtain one representative input that exercises the program to go down that specific path

Symbolic execution implementations: KLEE, Java PathFinder, etc.

More details on Symbolic Execution

Instead of concrete state, the program maintains **symbolic states**, each of which maps variables to symbolic values

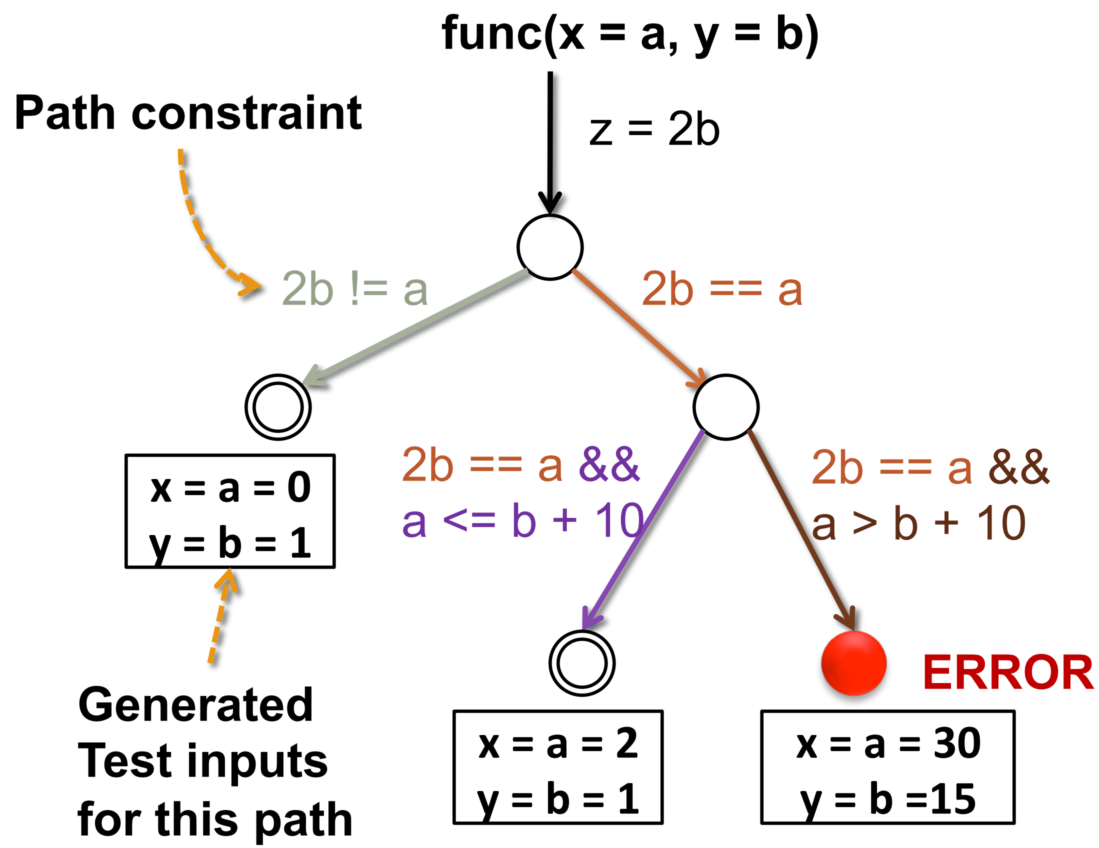
Path condition is a quantifier-free formula over the symbolic inputs that encodes all branch decisions taken so far

All paths in the program form its **execution tree**, in which some paths are feasible and some are infeasible

Symbolic Execution

```
Void func(int x, int y){

 int z = 2 * y;

 if(z == x){
          if (x > y + 10)
              ERROR

 }

}

int main(){
 int x = sym_input();
 int y = sym_input();
 func(x, y);
 return 0;

}
```

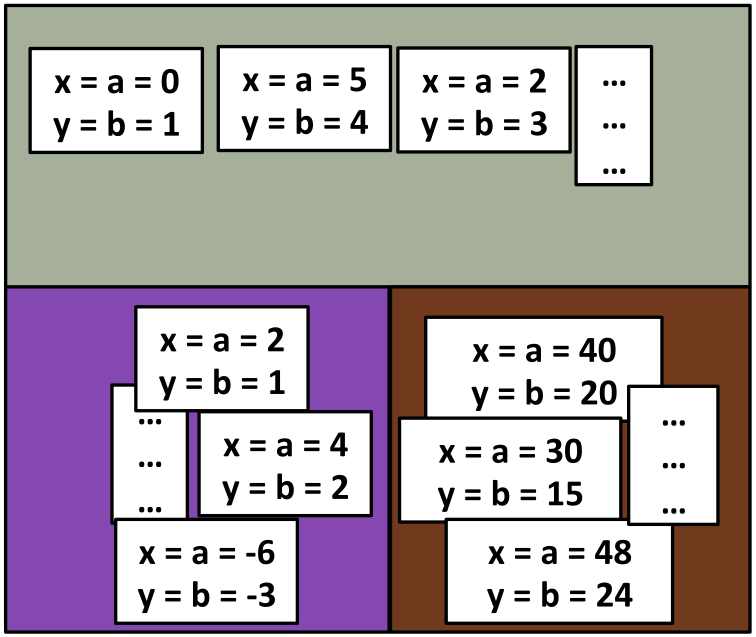
How does symbolic execution work?



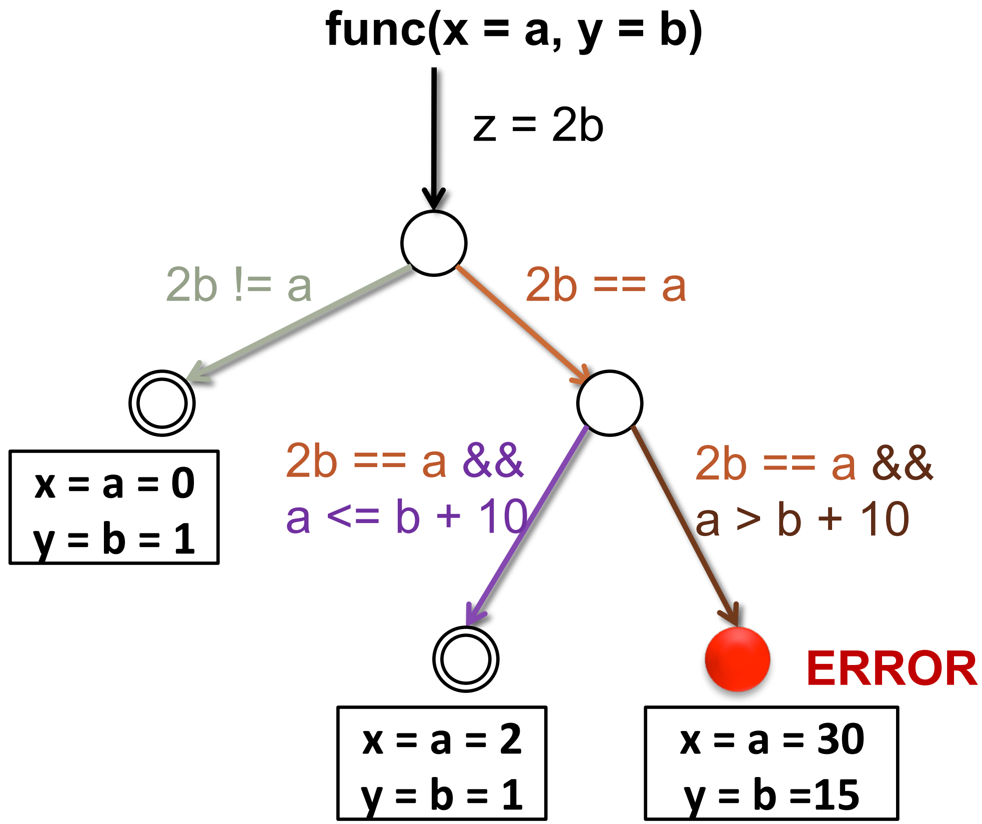
Note: Require inputs to be marked as symbol

Symbolic Execution

How does symbolic execution work?



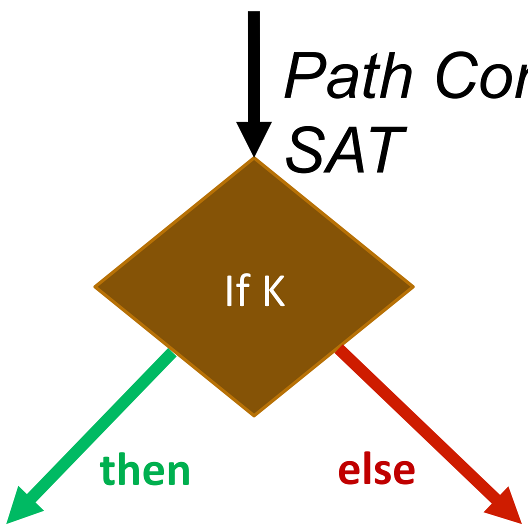
Path constraints represent equivalence classes of inputs



SMT Queries

Counterexample queries (generate a test case)

Branch queries (whether a branch is valid)



Path Constraints = $\{C_1, C_2, \ldots, C_n\}$;
SAT

<u>Use queries to determine validity of a branch</u>

else path is impossible: $C_1 \wedge C_2 \wedge \ldots \wedge C_n \wedge \neg K$ is UNSAT

then path is impossible: $C_1 \wedge C_2 \wedge \ldots \wedge C_n \wedge K$ is UNSAT

Optimizing SMT Queries

Expression rewriting

- Simple arithmetic simplifications (x * 0 = 0)
- Strength reduction ($x * 2^n = x << n$)
- Linear simplification (2 * x - x = x)

Constraint set simplification

- x < 10 && x = 5 --> x = 5

Implied Value Concretization

- x + 1 = 10 --> x = 9

Constraint Independence

- i<j && j < 20 && k > 0 && i = 20 --> i<j && i<20 && i=20

Optimizing SMT Queries (contd.)

Counter-example Cache

- i < 10 && i = 10 (no solution)
- i < 10 && j = 8 (satisfiable, with variable assignments i → 5, j → 8)

Superset of unsatisfiable constraints

- {i < 10, i = 10, j = 12} (unsatisfiable)

Subset of satisfiable constraints

- i → 5, j → 8, satisfies i < 10

Superset of satisfiable constraints

- Same variable assignments might work

How does Symbolic Execution Find bugs?

It is possible to extend symbolic execution to help

How: Dedicated checkers

- **Divide by zero example** --- y = x / z wh ... bles and assume current PC is ***f***
- Even though we only fork in ... the division operator
- One branch in wh ... !=0
- We will get ... constraints:

z ...

- ... & ***f*** will give us concrete input values that will ... error.

Write a dedicated checker for each kind of bug (e.g., buffer overflow, integer overflow, integer underflow)

Classic Symbolic Execution --- Practical Issues

Loops and recursions --- infinite execution tree

Path explosion --- exponentially many paths

Heap modeling --- symbolic data structures and pointers

SMT solver limitations --- dealing with complex path constraints

Environment modeling --- dealing with native/system/library calls/file operations/network events

Coverage Problem --- may not reach deep into the execution tree, specially when encountering loops.

Solution: Concolic Execution

Concolic = **Con**crete + Symb**olic**

Combining Classical Testing with Automatic Program Analysis

Also called **dynamic symbolic execution**

The intention is to visit deep into the program execution tree

Program is simultaneously executed with concrete and symbolic inputs

Start off the execution with a random input

Specially useful in cases of remote procedure call

Concolic execution implementations: SAGE (Microsoft), CREST

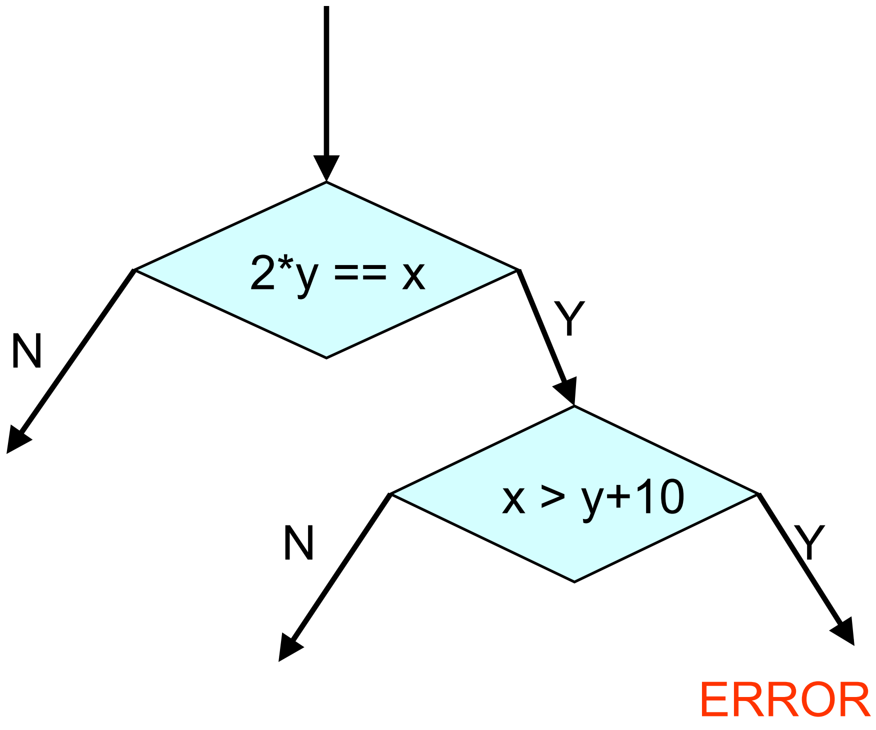
Concolic Execution Steps

- Generate a random seed input to start execution
- Concretely execute the program with the random seed input and collect the path constraint
- Example: **a && b && c**
- In the next iteration, negate the last conjunct to obtain the constraint **a && b && !c**
- Solve it to get input to the path which matches all the branch decisions except the last one

Why not from the first?

Example

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



Concolic execution example

```
void testme (int x, int y) {

z = 2* y;
if (z == x) {
        if (x > y+10) {
            ERROR;
        }
 }
}
```

Concrete Execution | Symbolic Execution

concrete state	symbolic state	path condition
x = 22, y = 7	x = a, y = b	

Concolic execution example

```
void testme (int x, int y) {
z = 2* y;

if (z == x) {
        if (x > y+10) {
            ERROR;
        }
 }
}
```

Concrete Execution	Symbolic Execution	
concrete state	symbolic state	path condition
x = 22, y = 7, z = 14	x = a, y = b, z = 2*b	

Concolic execution example

```
void testme (int x, int y) {
z = 2* y;

if (z == x) {
        if (x > y+10) {
            ERROR;
        }
 }
}
```

	Concrete Execution	Symbolic Execution	
	concrete state	symbolic state	path condition
←	x = 22, y = 7, z = 14	x = a, y = b, z = 2*b	

Concolic execution example

	Concrete Execution		Symbolic Execution	
	concrete state		symbolic state	path condition

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

2*b != a

x = 22, y = 7, z = 14

x = a, y = b, z = 2*b

Concolic execution example

Concrete Execution | Symbolic Execution

concrete state | symbolic state | path condition

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

Solve: 2*b == a
Solution: a = 2, b = 1

concrete state	symbolic state	path condition
		2*b != a
x = 22, y = 7, z = 14	x = a, y = b, z = 2*b	

Concolic execution example

	Concrete Execution		Symbolic Execution	
	concrete state		symbolic state	path condition
	x = 2, y = 1		x = a, y = b	

```
void testme (int x, int y) {

 z = 2* y;
 if (z == x) {
         if (x > y+10) {
              ERROR;
         }
 }
}
```

Concolic execution example

```
void testme (int x, int y) {
 z = 2* y;

 if (z == x) {
        if (x > y+10) {
            ERROR;
        }
 }
}
```

	Concrete Execution		Symbolic Execution	
	concrete state		symbolic state	path condition
	x = 2, y = 1, z = 2		x = a, y = b, z = 2*b	

Concolic execution example

```
void testme (int x, int y) {
 z = 2* y;
 if (z == x) {

        if (x > y+10) {
              ERROR;
        }
 }
}
```

	Concrete Execution		Symbolic Execution	
	concrete state	symbolic state	path condition	
if (z == x)			2*b == a	
←	x = 2, y = 1, z = 2	x = a, y = b, z = 2*b		

Concolic execution example

```
void testme (int x, int y) {
 z = 2* y;
 if (z == x) {

     if (x > y+10) {
         ERROR;
     }
 }
}
```

Concrete Execution		Symbolic Execution	
concrete state	symbolic state	path condition	
x = 2, y = 1, z = 2	x = a, y = b, z = 2*b	2*b == a a < b + 10	

Concolic execution example

```
void testme (int x, int y) {
 z = 2* y;
 if (z == x) {

      if (x > y+10) {
          ERROR;
      }
 }
}
```

Concrete Execution | Symbolic Execution

concrete state | symbolic state | path condition

Solve: (2*b == a) ^ (a – b> 10)
Solution: a = 30, b = 15

2*b == a

a - b < 10

x = 2, y = 1, z = 2

x = a, y = b, z = 2*b

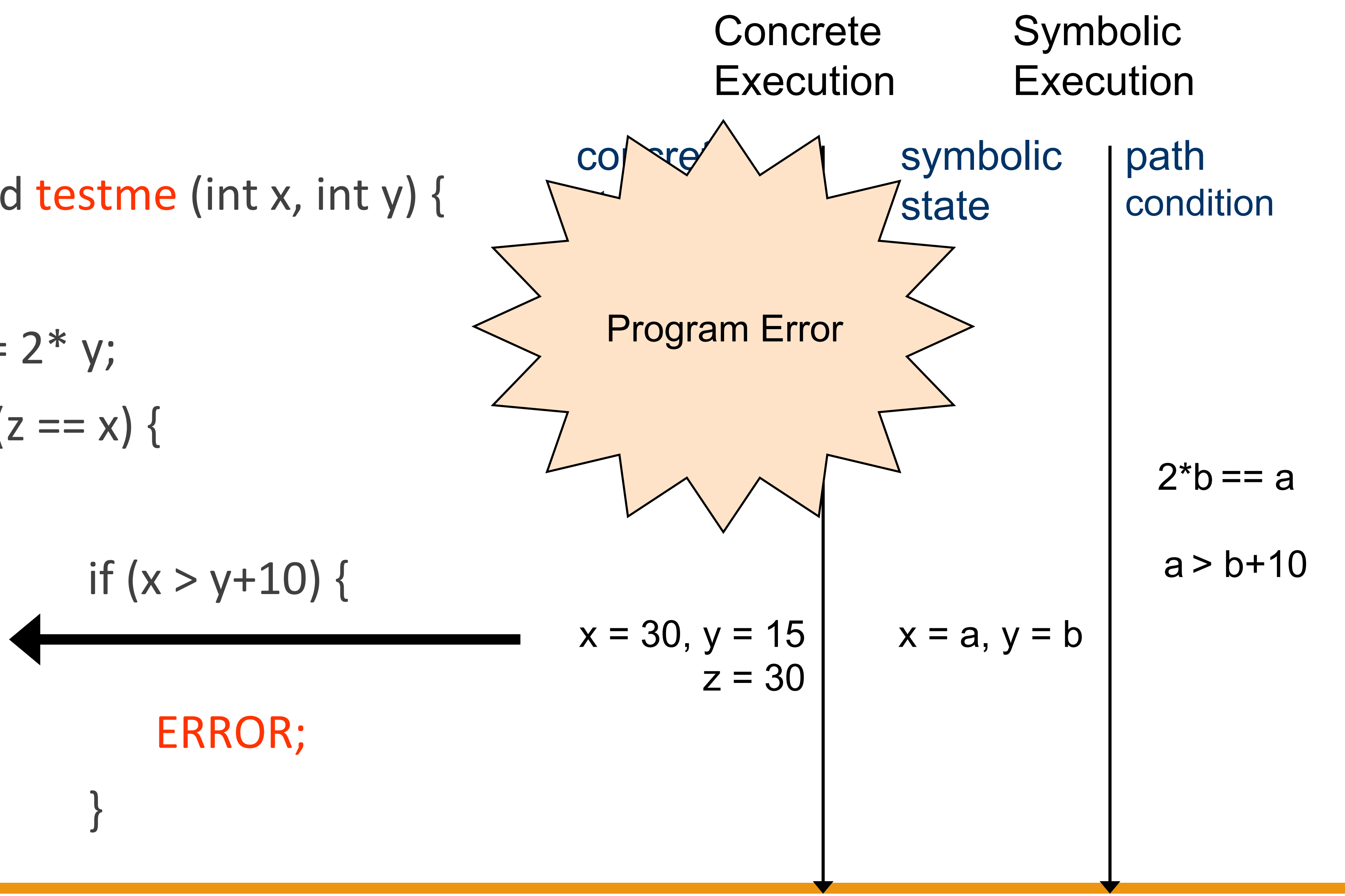
Concolic execution example

	Concrete Execution		Symbolic Execution	
	concrete state		symbolic state	path condition
	x = 30, y = 15		x = a, y = b	

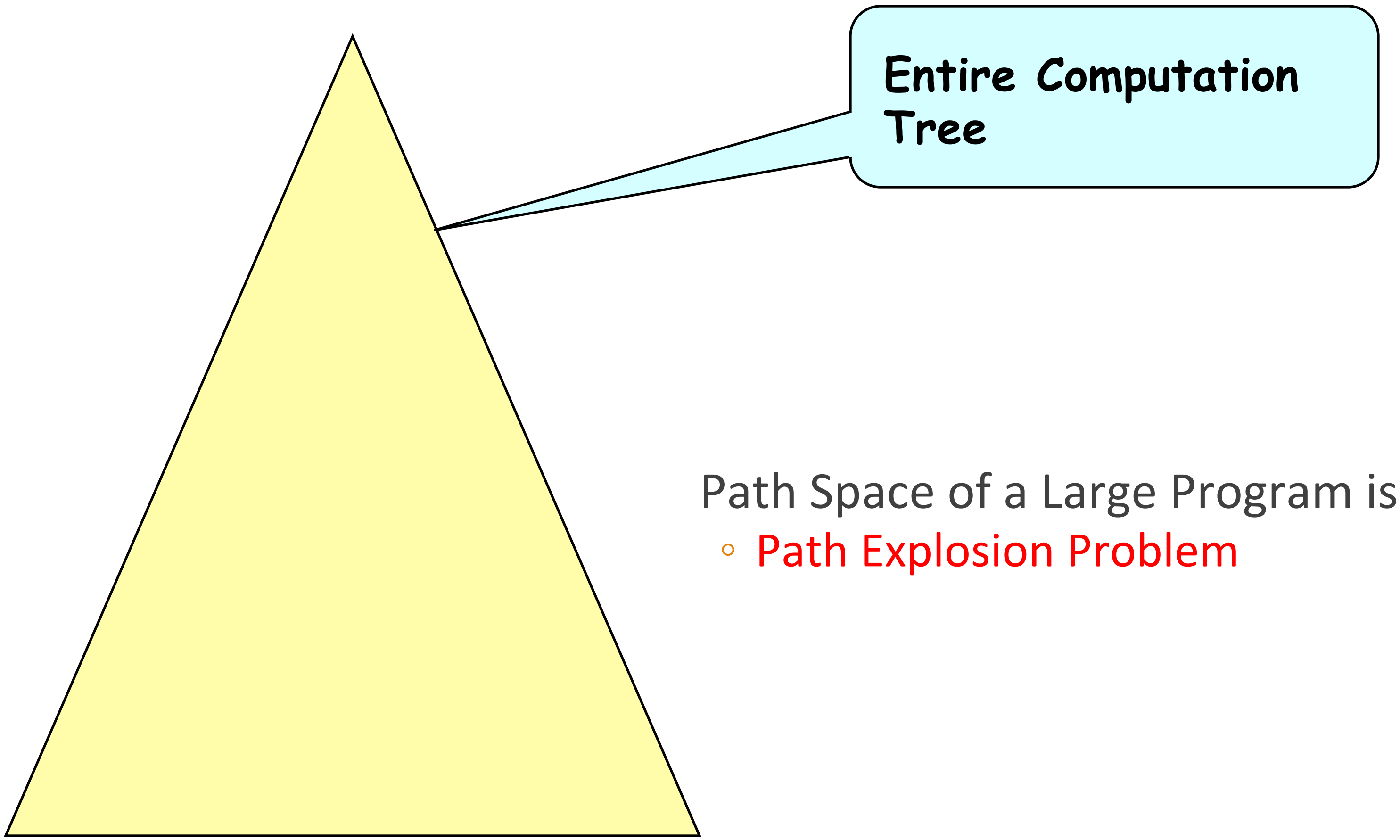
```
void testme (int x, int y) {
 z = 2* y;

 if (z == x) {
          if (x > y+10) {
               ERROR;
          }
 }
}
```

Concolic execution example

```
void testme (int x, int y) {

 z = 2* y;
 if (z == x) {

      if (x > y+10) {

           ERROR;

      }
 }
```



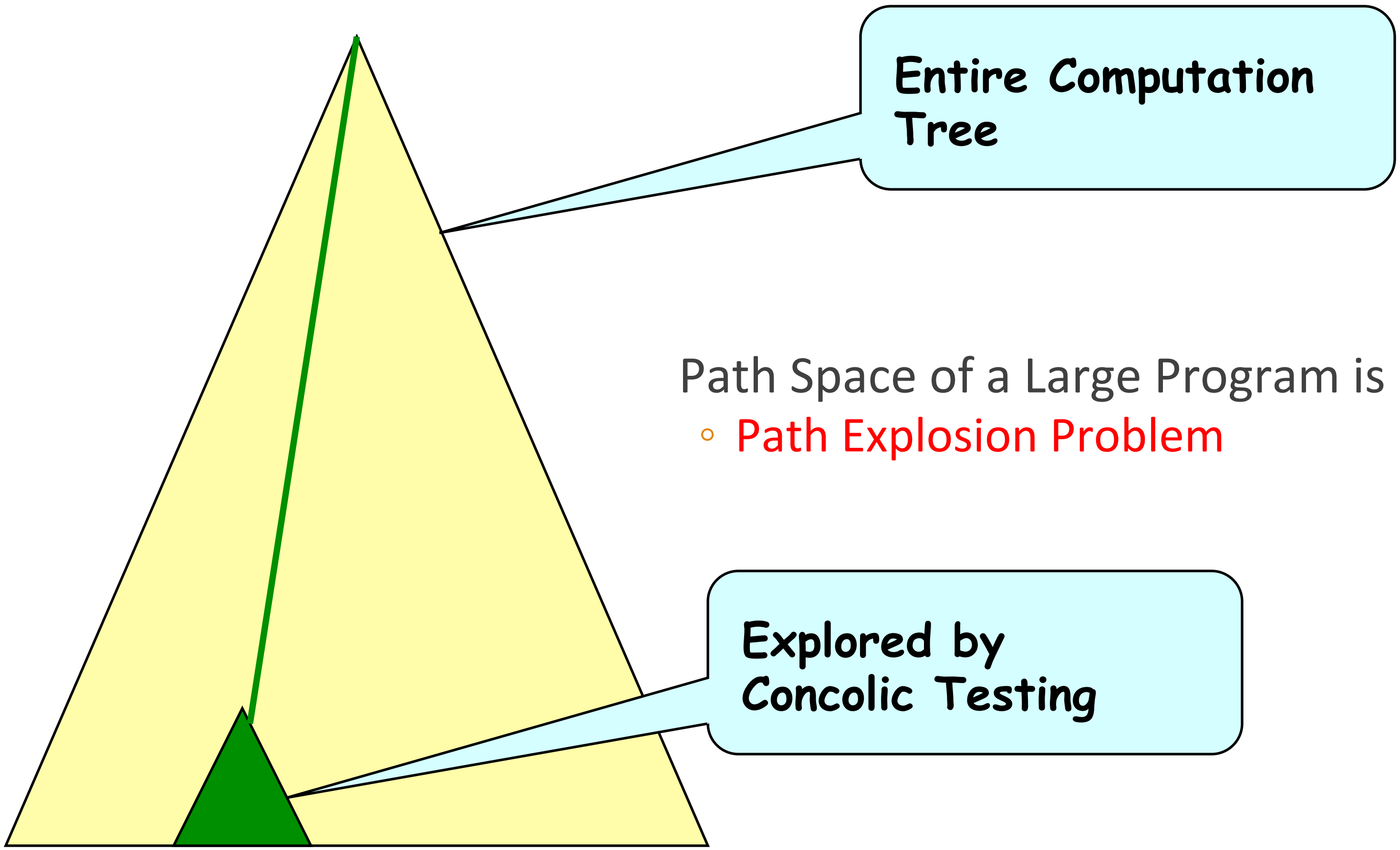
Limitations



Path Space of a Large Program is Huge

- Path Explosion Problem

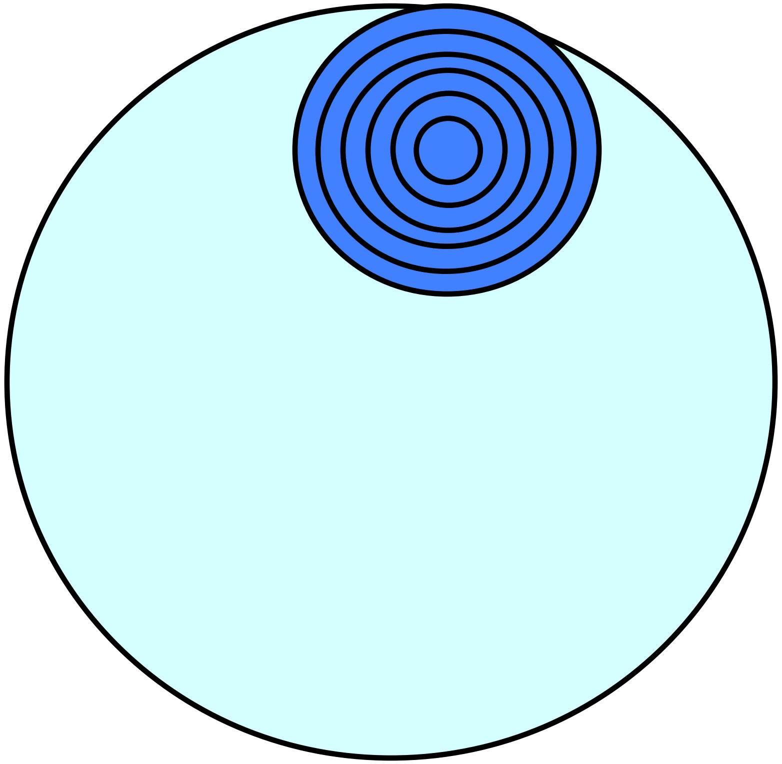
Limitations



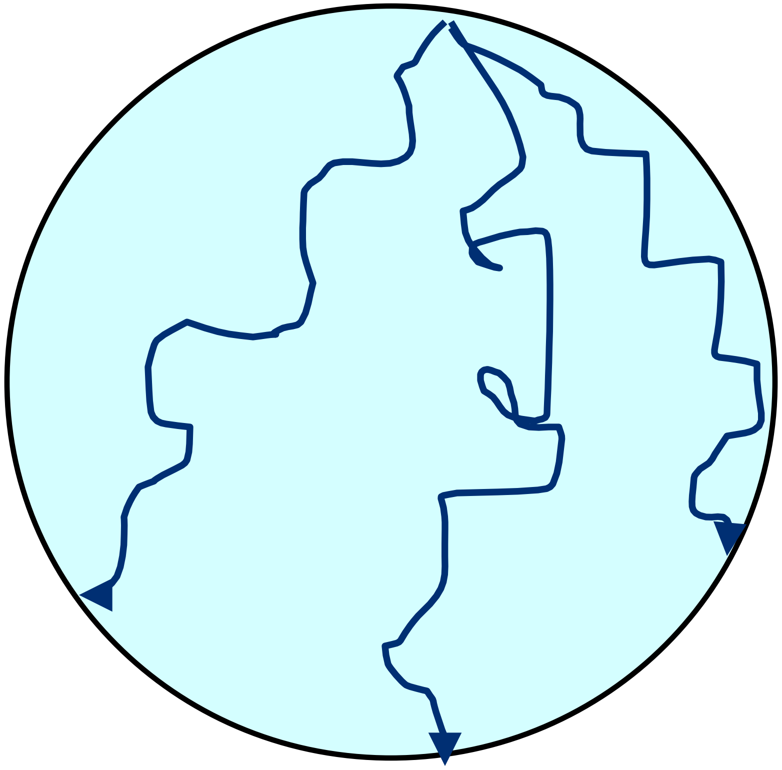
Path Space of a Large Program is Huge

- Path Explosion Problem

Limitations: a comparative view



Concolic: Broad, shallow



Random: Narrow, deep

Limitations: Example

```
Example ( ) {
1: state = 0;
2: while(1) {
3:   s = input();
4:   c = input();
5:   if(c==':' && state==0)
       state=1;
6:   else if(c=='\n' && state==1)
       state=2;
7:   else if (s[0]=='I' &&
       s[1]=='C' &&
       s[2]=='S' &&
       s[3]=='E' &&
       state==2) {
                  COVER_ME:;
     }
   }
}
```

- **Want to hit COVER_ME**
- **input() denotes external input**
- **Can be hit on an input sequence**
  - **s = "ICSE"**
  - **c : ':' '\n'**

Similar code in

- **Text editors (vi)**
- **Parsers (lexer)**
- **Event-driven programs (GUI)**

Limitations: Example

```
Example ( ) {
1: state = 0;
2: while(1) {
3:   s = input();
4:   c = input();
5:   if(c==':' && state==0)
       state=1;
6:   else if(c=='\n' && state==1)
       state=2;
7:   else if (s[0]=='I' &&
       s[1]=='C' &&
       s[2]=='S' &&
       s[3]=='E' &&
       state==2) {
                  COVER_ME:;
     }
   }
}
```

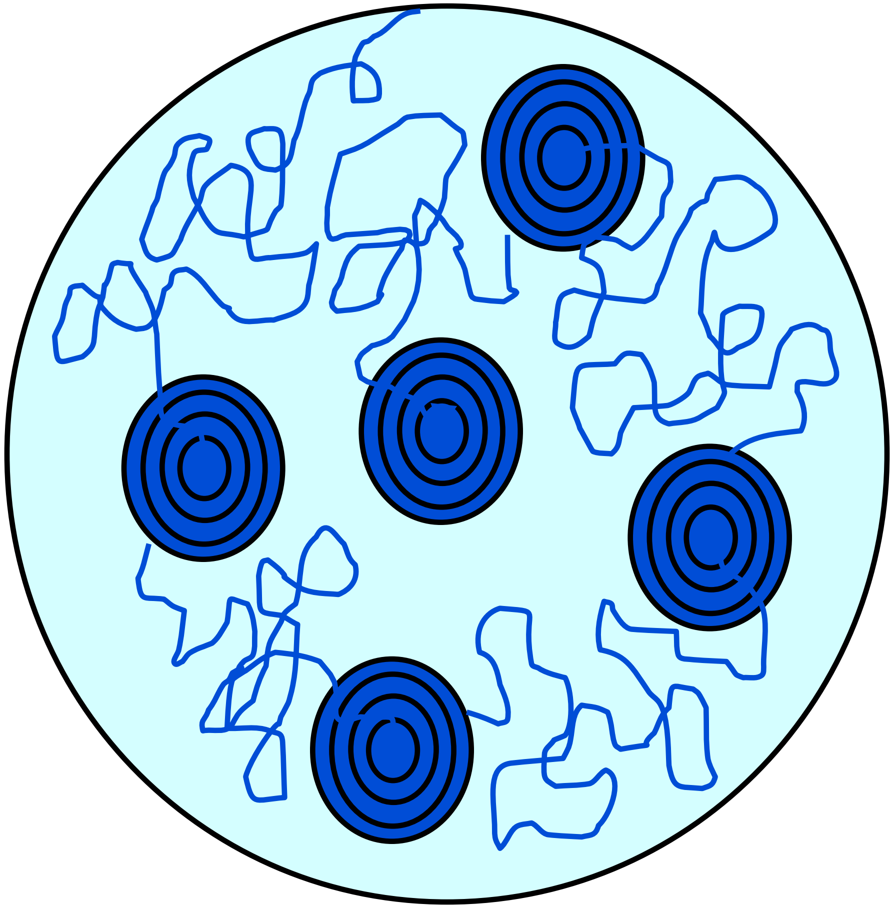
- **Pure random testing can get to state = 2**

But difficult to get 'ICSE' as a Sequence

Probability $1/(2^8)^6$ » $3X10^{-15}$

- **Conversely, concolic testing can generate 'ICSE' but explores many paths to get to state = 2**

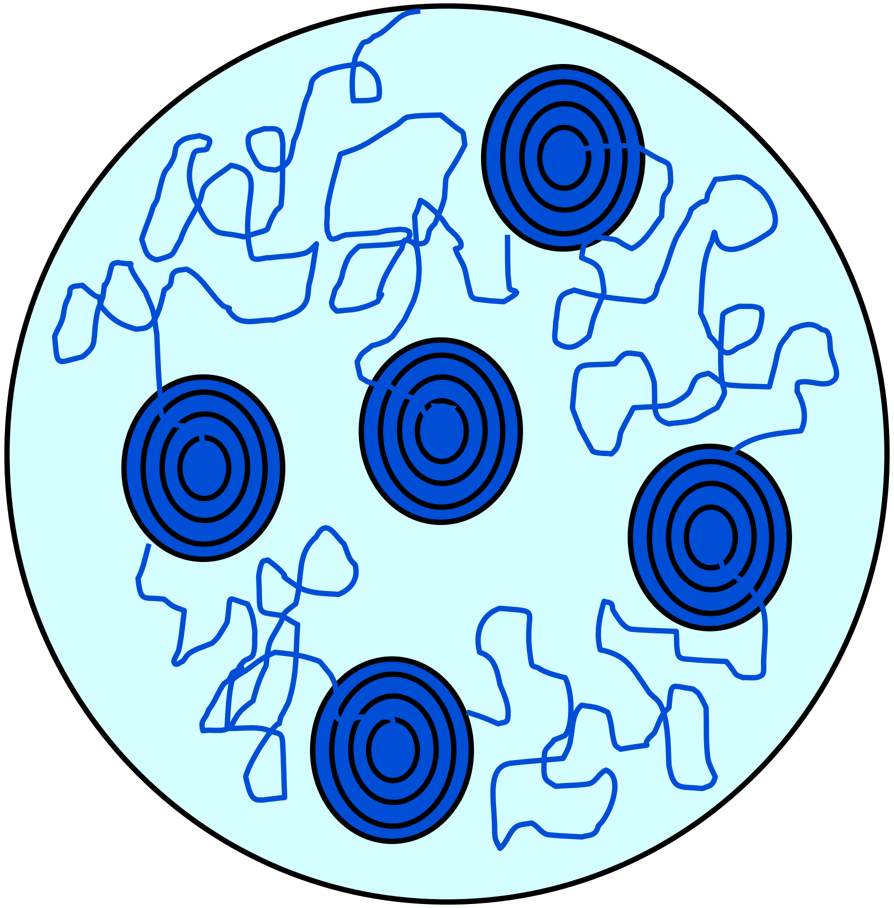
Hybrid concolic testing



```
while (not required coverage) {

    while (not saturation)
        perform random testing;
    Checkpoint;
    while (not increase in coverage)
        perform concolic testing;
    Restore;
}
```

Interleave Random Testing and Concolic Testing to increase coverage

Hybrid Concolic Testing



```
while (not required coverage) {
    while (not saturation)
        perform random testing;
    Checkpoint;
    while (not increase in coverage)
        perform concolic testing;
    Restore;
}
```

Interleave Random Testing and Concolic Testing to increase coverage

Deep, broad search
Hybrid Search

Hybrid Concolic Testing

```
Example ( ) {
1: state = 0;
2: while(1) {
3:   s = input();
4:   c = input();
5:   if(c==‘:’ && state==0)
       state=1;
6:   else if(c==‘\n’ && state==1)
       state=2;
7:   else if (s[0]==‘I’ &&
       s[1]==‘C’ &&
       s[2]==‘S’ &&
       s[3]==‘E’ &&
       state==2) {
                 COVER_ME:;
     }
   }
}
```

Random Phase

- ‘$’, ‘&’, ‘-’, ‘6’, ‘:’, ‘%’, ‘^’, ‘\n’, ‘x’, ‘~’ …
  - Saturates after many (~10000) iterations
  - In less than 1 second
  - **COVER_ME** is not reached

Hybrid Concolic Testing

```
Example ( ) {
1: state = 0;
2: while(1) {
3:   s = input();
4:   c = input();
5:   if(c==':' && state==0)
        state=1;
6:   else if(c=='\n' && state==1)
        state=2;
7:   else if (s[0]=='I' &&
        s[1]=='C' &&
        s[2]=='S' &&
        s[3]=='E' &&
        state==2) {
                   COVER_ME:;
     }
   }
}
```

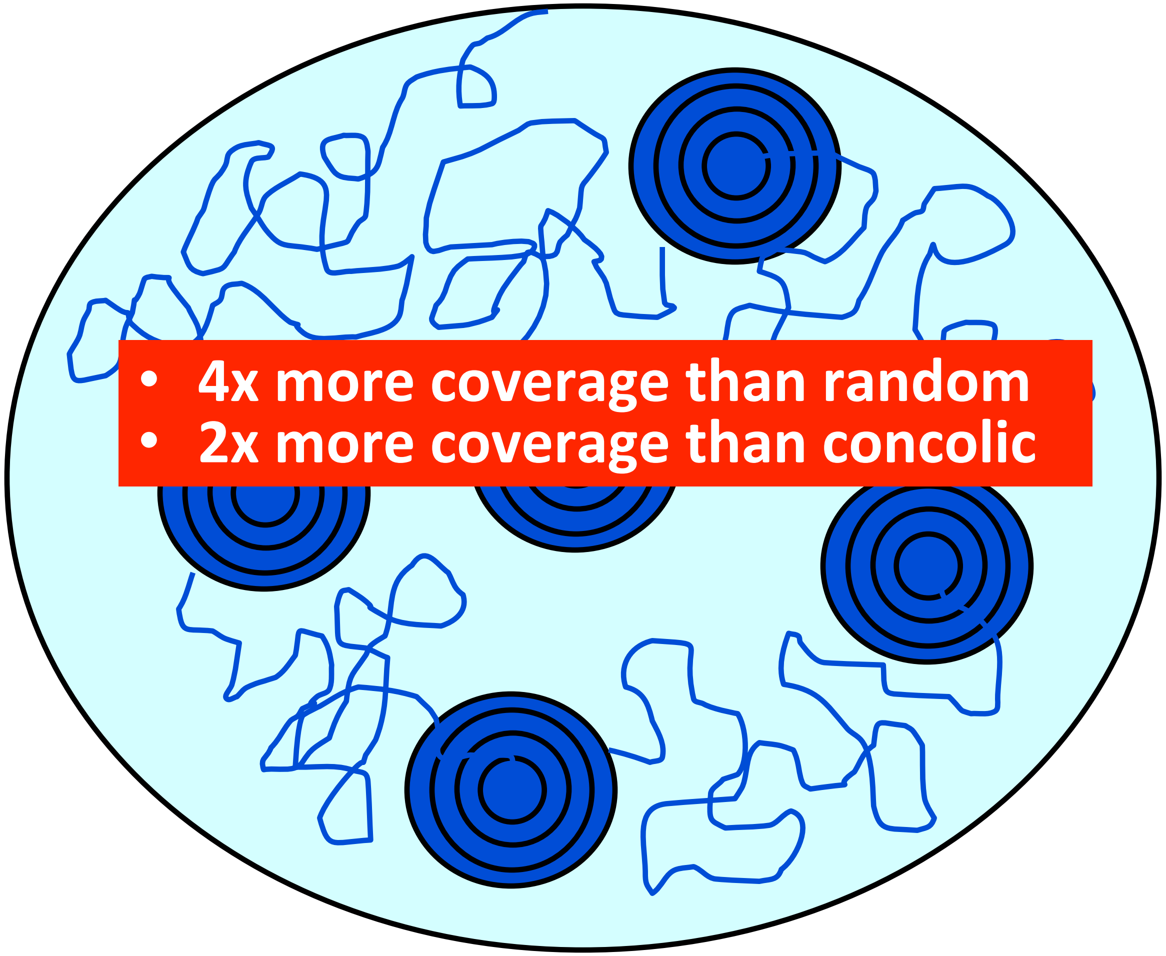
Random Phase

- '$', '&', '-', '6', ':', '%', '^', '\n', 'x', '~' …
  - Saturates after many (~10000) iterations
  - In less than 1 second
  - **COVER_ME** is not reached

Concolic Phase

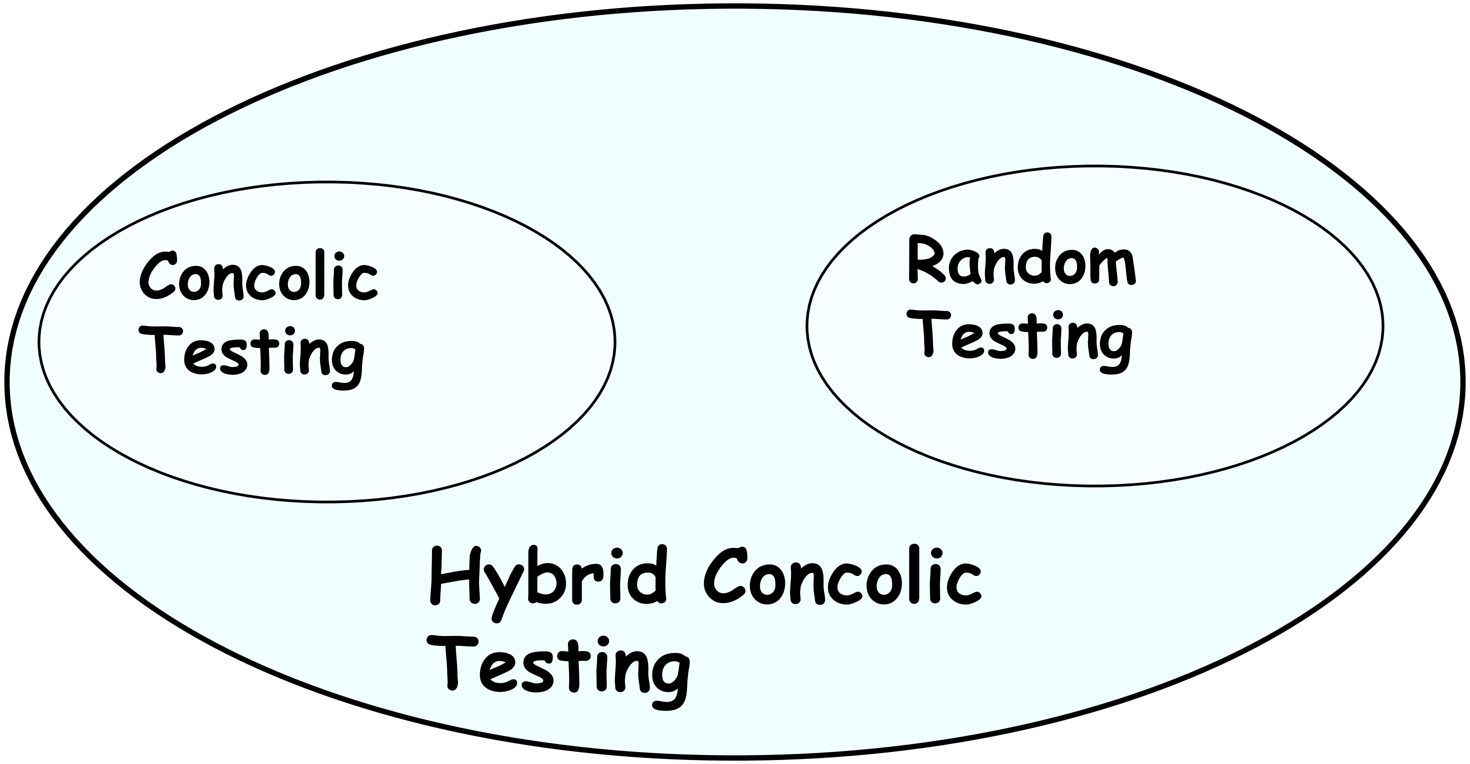
- s[0]='I', s[1]='C', s[2]='S', s[3]='E'
  - Reaches **COVER_ME**

Hybrid Concolic Testing



- **4x more coverage than random**
- **2x more coverage than concolic**

Summary



Further reading

Symbolic execution and program testing - James King

KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs - Cadar et. al.

Symbolic Execution for Software Testing: Three Decades Later - Cadar and Sen

DART: Directed Automated Random Testing - Godefroid et. al.

CUTE: A Concolic Unit Testing Engine for C - Sen et. al.