Symbolic and Concolic Execution of Programs

Information Security, CS 526

Omar Chowdhury

Reading for this lecture

- 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
- A Few Billion Lines of Code Later Using Static Analysis to Find Bugs in the Real World - Engler et. al.
- DART: Directed Automated Random Testing - Godefroid et. al.
- CUTE: A Concolic Unit Testing Engine for C - Sen et. al.

What is the goal?

```c
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

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

Obvious Questions?

Can y … of test
q … ome how make
… tomatic?

Yes, we can.

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

SMT: **Satisfiability Modulo Theories**

Input: a **first-order** formula φ over background theory

Output: is φ satisfiable?

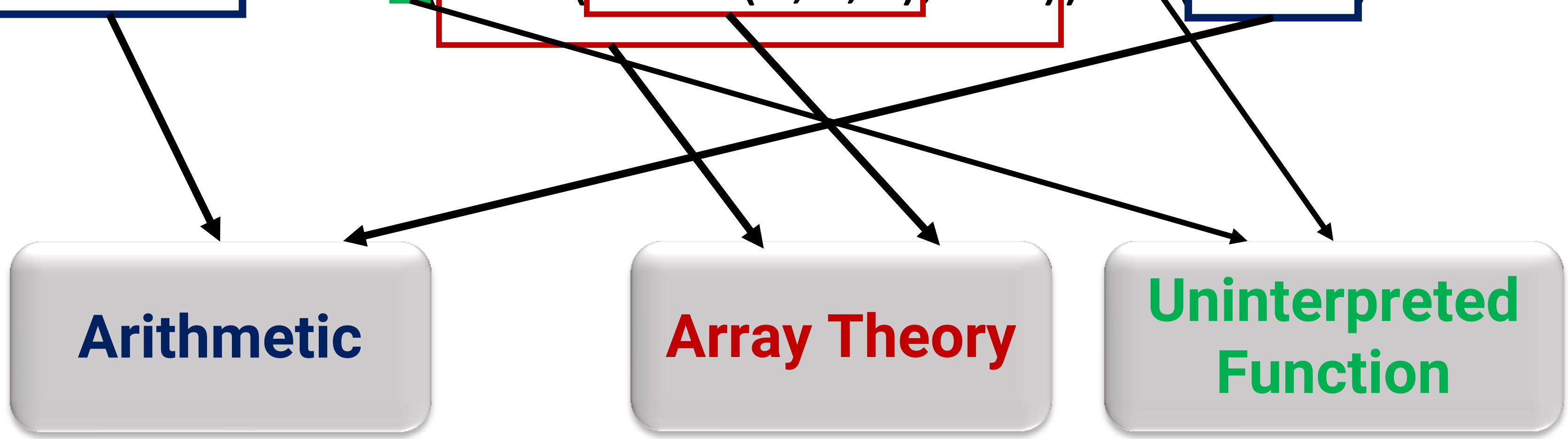
- does φ have a model?
- Is there a refutation of φ = proof of $\neg\varphi$?

For most SMT solvers: φ is a ground formula

- Background **theories**: Arithmetic, Arrays, Bit-vectors, Algebraic Datatypes
- Most SMT solvers support **simple first-order sorts**

Background: SMT

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



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)

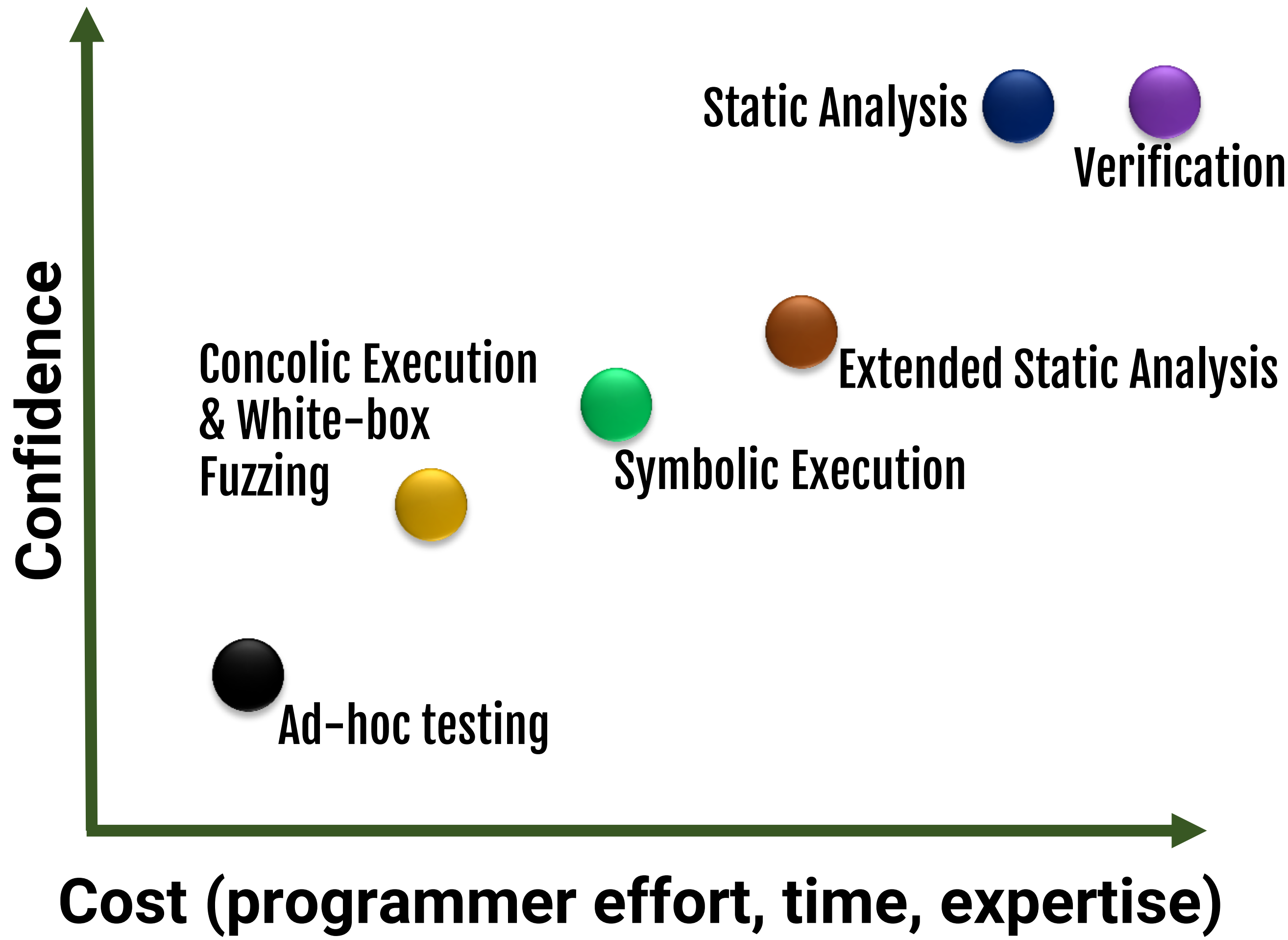
[Arithmetic simplification]

- 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**]

Program Validation Approaches



Automatic Test Generation
Symbolic & Concolic Execution

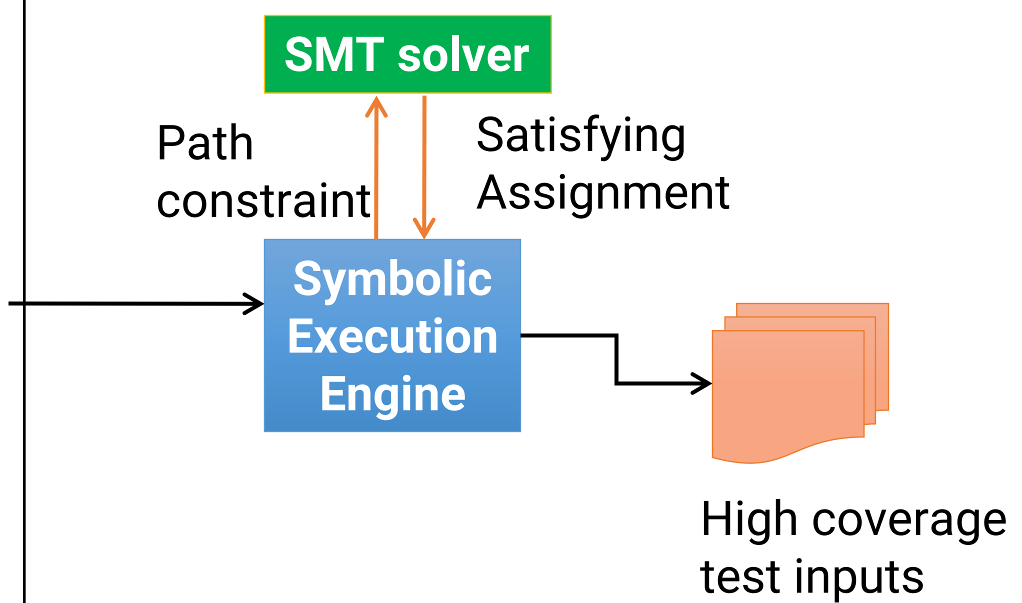
- How do we automatically generate test inputs that induce the program to go in different paths?
- **Intuition**:
  - Divide the whole possible input space of the program into equivalent classes of input.
  - For each equivalence class, all inputs in that equivalence class will induce the same program path.
  - Test one input from each equivalence class.

Symbolic Execution – History

- **1976**: A system to generate test data and symbolically execute programs (Lori Clarke)
- **1976**: Symbolic execution and program testing (James King)
- **2005-present**: practical symbolic execution
  - Using SMT solvers
  - Heuristics to control exponential explosion
  - Heap modeling and reasoning about pointers
  - Environment modeling
  - Dealing with solver limitations

Symbolic Execution (contd.)

```
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 – Description

- 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 represent 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

More details on Symbolic Execution

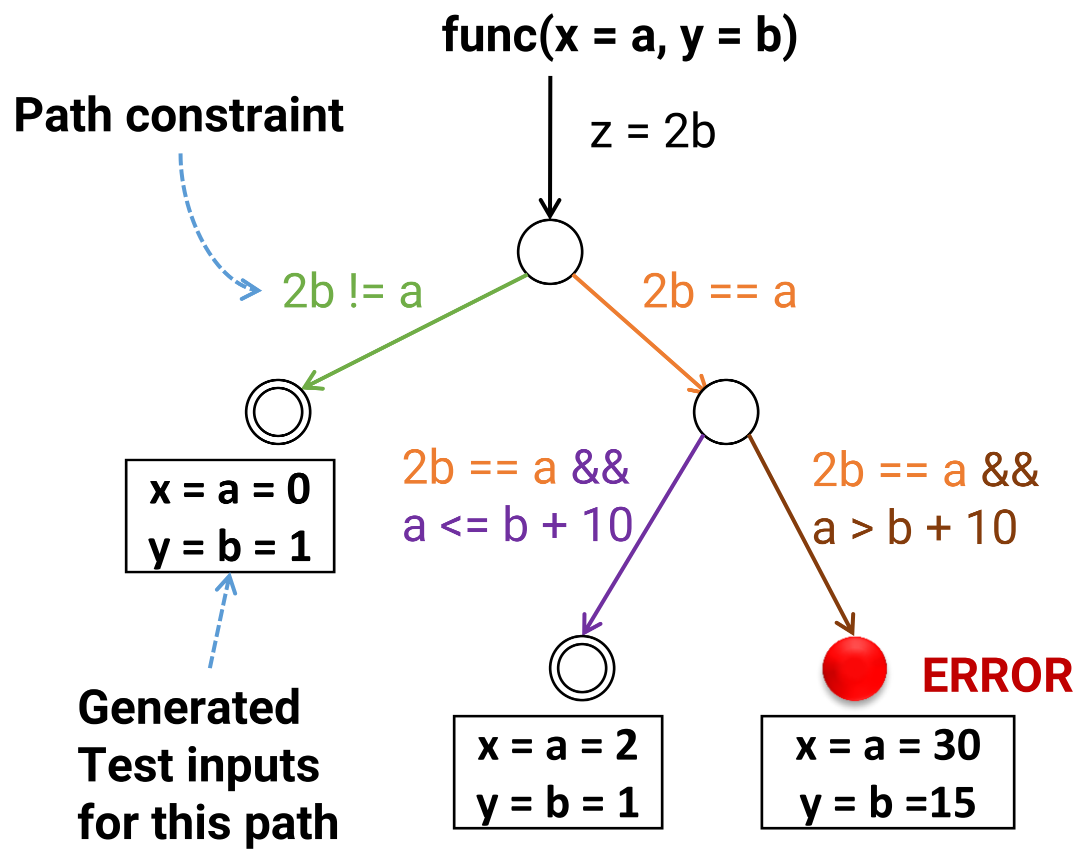
- Instead of concrete state, the program maintains **symbolic states**, each of which maps variables to symbolic values
- **Path condition** is a quantifier-free formula over the symbolic inputs that encodes all branch decisions taken so far
- All paths in the program form its **execution tree**, in which some paths are feasible and some are infeasible

Symbolic Execution (contd.)

```
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;
}
```

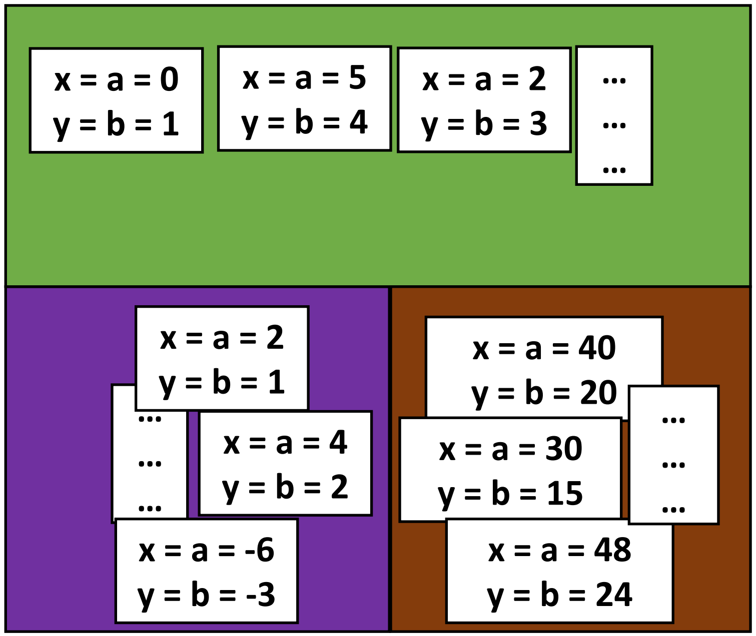
How does symbolic execution work?



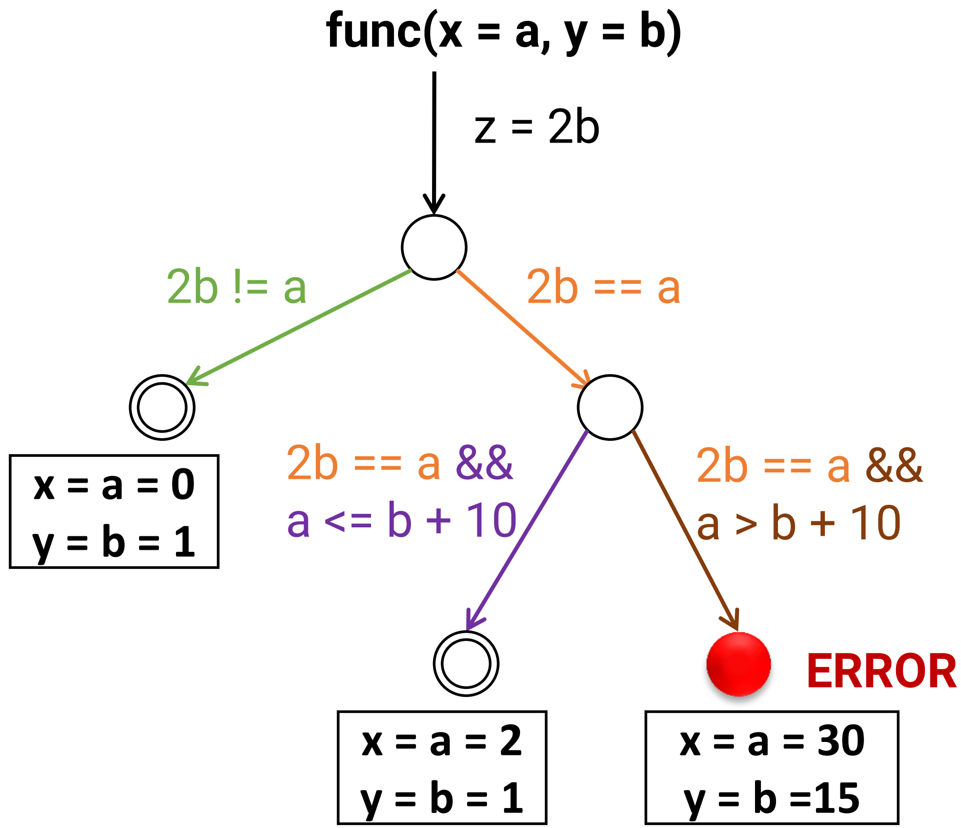
Note: Require inputs to be marked as symbolic

Symbolic Execution (contd.)

How does symbolic execution work?



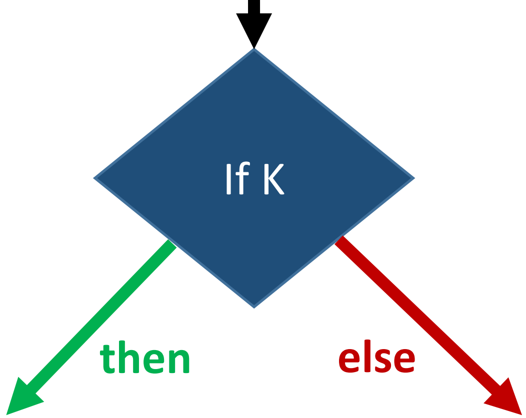
Path constraints represent equivalence classes of inputs



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 && i < 20 && k > 0; new constraint 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 works

How does Symbolic Execution Find bugs?

- It is possible to extend symbol… …p us catch bugs
- **How**: Dedicated chec…
  - **Divide by zero c…** …ere x and z are symbolic va… …rrent PC is ***f***
  - Even th… …anches we will now fork in the…
  - … z = 0 and another where z !=0
  - …paths with the following constraints:
  - z != 0 && ***f***
  - …g the constraint z = 0 && ***f*** will give us concrete …t values that will trigger the divide by zero 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

Classic Symbolic Execution – Practical Issues (possible solutions)

Path Constraints

- **Infinite execution tree**
  - Finitize paths by limiting the PC size (bounded verification)
  - Use loop invariants (verification)
- **Path explosion**
  - Select next branch at random
  - Select next branch based on coverage
  - Interleave symbolic execution with random testing
- **Heap modeling**
  - Segmented address space via the theory of arrays (Klee)
  - Lazy concretization (JPF)
  - Concolic lazy concretization (CUTE)

Classic Symbolic Execution – Practical Issues (possible solutions)

- **SMT solver limitations**
  - On-the-fly expression simplification
  - Incremental solving
  - Solution caching
  - Counterexample caching
  - Substituting concrete values for symbolic in complex PCs (CUTE)
- **Environment modeling**
  - Partial state concretization
  - Manual models of the environment (Klee)

Symbolic Execution Coverage Problem

Symbolic execution may not reach deep into the execution tree. Specially when encountering loops.

Solution: Concolic Execution

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

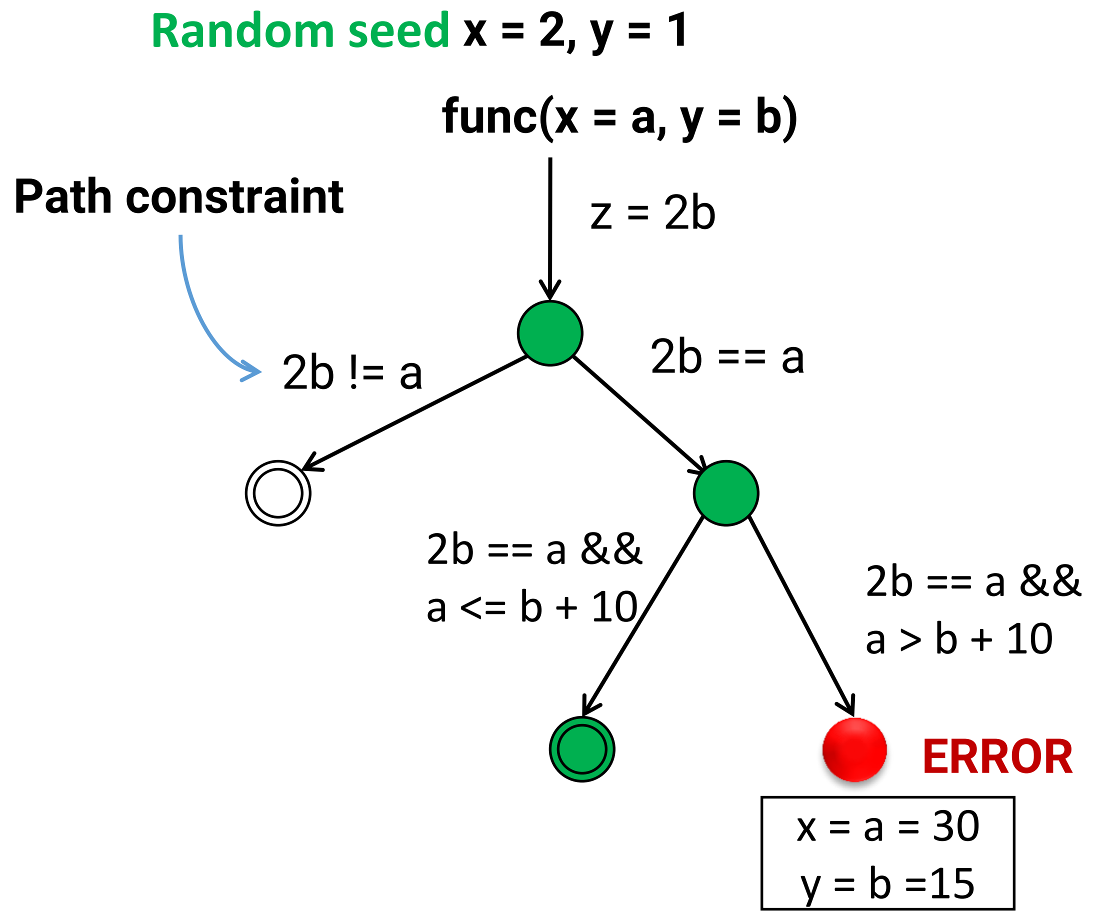
Concolic Execution Steps

- 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?

Concolic Execution

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



Acknowledgement

Some of the content are derived from the slides of Endadul Haque, Emina Torlak, Nikolaj Bjørner, Bruno Dutertre, and Leonardo de Moura