# Lec 26: Symbolic Execution

**CSED415: Computer Security** 

Spring 2024

Seulbae Kim



## Administrivia



- All labs completed
  - Grace period for Lab 5 ends on May 26
  - Labs will be graded this weekend and be reviewed next week after project presentations
- Final exam will be on June 4
  - Note: June 6 is a national holiday

## Administrivia



- Project presentations next week
  - 15 min presentation + 5 min Q&A = 20 min per team
    - Three teams will present on Tue, May 28
    - The other three teams will present on Thu, May 30
  - Presentation order will be decided today
  - Presentation should include a demonstration (live or recorded)
  - All teams MUST submit their slides, code, and report by May 27

- May 28
  - ?
  - ?
  - ?
- May 30
  - ?
  - ?
  - ?

```
import random
import time
random.seed(time.time())
N = 10 \# to be selected in class
teams = \lceil
    "Agustina & Megan",
    "whysw",
    "구얏",
    "h@ckerz",
    "q1w2e3r4",
    "Poulpy"
for i in range(N):
    random.shuffle(teams)
print(teams)
```

# Program Analysis for Bug Finding – Part 2

## Motivation



- Fuzzing is sound if its bug oracle is precise
  - Bugs detected by a fuzzer are indeed bugs (no FP)
  - However, it is far from being complete (many FN)

Is there an approach that aims to be complete? (i.e., that does not miss any bug)

## Static vs Dynamic analysis



- Static analysis:
  - Analysis that is performed without executing a program
  - Examples:
    - Decompilation
    - Pointer analysis
    - Symbolic execution (Today's topic)

- Dynamic analysis:
  - Analysis that is performed during program execution
  - Examples:
    - Fuzzing (Last topic)
    - Concolic execution

# Symbolic Execution

## Concrete (dynamic) vs Symbolic execution

POSTECH

Consider the following simple program

```
if (input == 0xdeadbeef) {
  bug();
} else {
  no_bug();
}
```

 In our last in-class experiment, dumb fuzzing concretely executed the program with randomly generated inputs for over 4 million times but still failed to reach the bug

Can we do any better?

CSED415 – Spring 2024

## Concrete (dynamic) vs Symbolic execution

POSTECH

- We humans intuitively know the input that is required to trigger the bug by just looking at the code
  - How? We can easily solve the <u>path constraint</u> for the if branch that leads to the bug!

```
if (input == 0xdeadbeef) {
  bug();
} else {
  no_bug();
}
```

Can a computer do the same?

## Concrete (dynamic) vs Symbolic execution

POSTECH

- Concrete execution: Run a program with a concrete input
  - Concrete input is a fixed value
  - Program behavior (i.e., branches taken) is determined by the input
- Symbolic execution: Run a program with a symbolic input
  - Program inputs are represented by symbols
    - A symbol represents any possible value
    - We can reason about possible program behaviors using the symbols
  - Goals:
    - Explore all execution paths of a program
    - Obtain concrete test input leading to each the path

## Symbolic execution - How



- Symbolic executor maintains an internal state  $(st, \sigma, \pi)$ 
  - st: The next statement to evaluate
  - $\sigma$ : Symbolic store
  - $\pi$ : Path constraints
- Depending on st, symbolic execution proceeds as follows:
  - st is an assignment (e.g., var = e):
    - $\sigma$  is updated by associating LHS (var) with a new symbolic expression  $e_s$  obtained by evaluating RHS (e) symbolically
  - st is an **if statement** (e.g., if  $e_s$  then  $path_1$  else  $path_2$ ):
    - Program is forked by creating two states with path constraints  $\pi \wedge e_s$  and  $\pi \wedge \neg e_s$
  - st is an assertion (e.g., assert(e)):
    - ullet The validity of e is checked using path constraints

```
POSTECH
```

```
void buggy(int x, int y) {
  int i = 10;
  int z = y * 2;
  if (z == x) {
    if (x >= y + 10) {
       z = z / (i - 10); // divzero
    }
  }
}
```

 $\sigma$ : Symbolic store

 $\pi$ : Path constraints

CSED415 – Spring 2024

```
POSTECH
```

```
void buggy(int x, int y) {
   int i = 10;
   int z = y * 2;
   if (z == x) {
      if (x >= y + 10) {
        z = z / (i - 10); // divzero
      }
   }
}
```

x and y are symbolic values

 $\sigma$ : Symbolic store

$$\chi \to \chi_S$$

$$y \rightarrow y_s$$

(Notation:  $var \rightarrow sym$ )

 $\pi$ : Path constraints

true

(branch always taken)

```
POSTECH
```

```
void buggy(int x, int y) {
   int i = 10;
   int z = y * 2;
   if (z == x) {
      if (x >= y + 10) {
        z = z / (i - 10); // divzero
      }
   }
}
```

i is a concrete value

 $\sigma$ : Symbolic store  $\chi \to \chi_{s}$  $y \rightarrow y_s$ 

 $\pi$ : Path constraints true



```
void buggy(int x, int y) {
    int i = 10;
    int z = y * 2;
    if (z == x) {
        if (x >= y + 10) {
            z = z / (i - 10); // divzero
        }
     }
}
```

#### st is an assignment

 $\sigma$  is updated by associating LHS (z) with a new symbolic expression  $e_s$  obtained by evaluating RHS (y\*2) symbolically

# $\sigma$ : Symbolic store

$$x \to x_S$$

$$y \to y_S$$

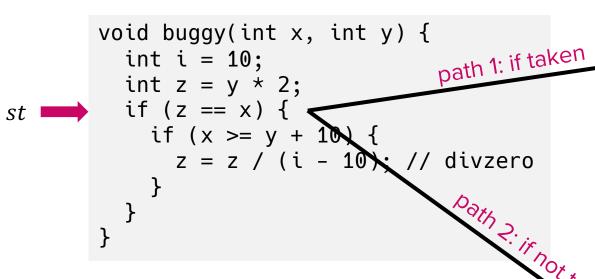
$$z \to 2 * y_S$$

 $\pi$ : Path constraints

true

CSED415 – Spring 2024

POSTECH



 $\sigma$ : Symbolic store

$$x \to x_s$$
$$y \to y_s$$

$$z \rightarrow 2 * y_s$$

 $\pi$ : Path constraints

$$x_s = 2 * y_s$$

st is an if statement

Program is forked by creating two states with path constraints  $\pi \land e_s$  and  $\pi \land \neg e_s$ 

Here,  $e_s$  is the symbolic evaluation of z == x

 $\sigma$ : Symbolic store

$$\chi \to \chi_{\rm S}$$

$$y \rightarrow y_s$$

$$z \rightarrow 2 * y_s$$

 $\pi$ : Path constraints

$$x_s \neq 2 * y_s$$

**POSTECH** 

#### Path 1

 $\sigma$ : Symbolic store

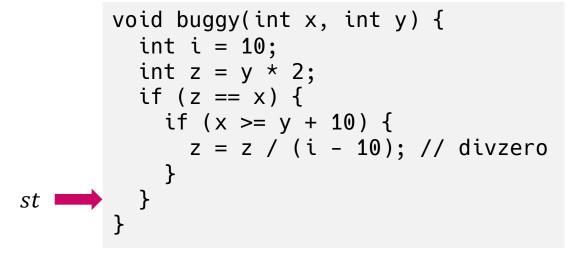
$$\chi \to \chi_{S}$$

$$y \rightarrow y_s$$

$$z \rightarrow 2 * y_s$$

 $\pi$ : Path constraints

$$x_s = 2 * y_s$$



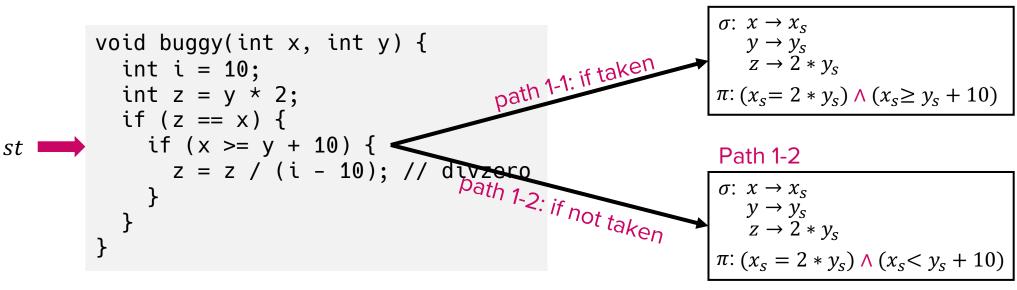
st hits a dead end if path 2 is followed

Nothing left to do for path 2. Go back and further explore path 1. Final states

#### Path 2

$$\sigma: x \to x_s y \to y_s z \to 2 * y_s \pi: x_s \neq 2 * y_s$$

POSTECH



#### st is an if statement

Program is forked by creating two states with path constraints  $\pi \land e_s$  and  $\pi \land \neg e_s$ 

Here,  $e_s$  is the symbolic evaluation of x>=y+10

#### Final states

Path 1-1

#### Path 2

$$\sigma: x \to x_S y \to y_S z \to 2 * y_S \pi: x_S \neq 2 * y_S$$

```
void buggy(int x, int y) {
  int i = 10;
  int z = y * 2;
  if (z == x) {
    if (x >= y + 10) {
      z = z / (i - 10); // divzero
```

st hits a dead end if path 1-2 is followed

Nothing left to do for path 1-2. Go back and further explore path 1-1.

#### **Path 1-1**

$$\sigma: x \to x_{S} y \to y_{S} z \to 2 * y_{S} \pi: (x_{S} = 2 * y_{S}) \land (x_{S} \ge y_{S} + 10)$$

#### Final states

#### Path 2

$$\sigma: x \to x_s \\ y \to y_s \\ z \to 2 * y_s$$
 
$$\sigma: x \to x_s \\ y \to y_s \\ z \to 2 *$$

#### Path 1-2

$$\begin{array}{c|cccc}
\sigma: & x \to x_s & & \sigma: & x \to x_s \\
& y \to y_s & & y \to y_s \\
& z \to 2 * y_s & & z \to 2 * y_s
\end{array}$$

$$\pi: & x_s \neq 2 * y_s & \pi: (x_s = 2 * y_s) \land (x_s < y_s + 10)$$

```
void buggy(int x, int y) {
  int i = 10;
  int z = y * 2;
  if (z == x) {
    if (x >= y + 10) {
     z = z / (i - 10); // divzero
```

#### st is an assignment

 $\sigma$  is updated by associating LHS (z) with a new symbolic expression  $e_s$ obtained by evaluating RHS (z/(i-10))symbolically

Note: Here, i is concrete

#### **Path 1-1**

$$\sigma: x \to x_{S} y \to y_{S} z \to 2 * y_{S}/0 \pi: (x_{S} = 2 * y_{S}) \land (x_{S} \ge y_{S} + 10)$$

#### Final states

#### Path 2

$$\sigma: x \to x_S$$

$$y \to y_S$$

$$z \to 2 * y_S$$

$$\pi: x_S \neq 2 * y_S$$

#### Path 1-2

$$\begin{array}{c|cccc}
\sigma: & x \to x_S & & \sigma: & x \to x_S \\
y \to y_S & & y \to y_S & & y \to y_S \\
z \to 2 * y_S & & z \to 2 * y_S & & \pi: & (x_S = 2 * y_S) \land (x_S < y_S + 10)
\end{array}$$

POSTECH

All program paths have been explored

#### Final states

### Path 1-1

$$\sigma: x \to x_{S} y \to y_{S} z \to 2 * y_{S} / 0 \pi: (x_{S} = 2 * y_{S}) \land (x_{S} \ge y_{S} + 10)$$

Potential div-by-zero error is detected! If  $\pi$  is satisfiable, this is an actual bug

#### Path 1-2

$$\sigma: x \to x_S$$

$$y \to y_S$$

$$z \to 2 * y_S$$

$$\pi: (x_S = 2 * y_S) \land (x_S < y_S + 10)$$

#### Path 2

$$\sigma: x \to x_S y \to y_S z \to 2 * y_S \pi: x_S \neq 2 * y_S$$

Next step: Solving  $\pi$  to obtain concrete test inputs for each path

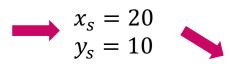
#### **Path 1-1**

$$\sigma: x \to x_{s} y \to y_{s} z \to 2 * y_{s} / 0 \pi: (x_{s} = 2 * y_{s}) \land (x_{s} \ge y_{s} + 10)$$

#### Solving $\pi$

Find  $x_s$  and  $y_s$  that satisfy •  $x_s = 2 * y_s$  and •  $x_s \ge y_s + 10$ 

#### Concrete input



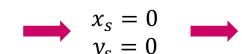
#### Verification?

#### Path 1-2

$$σ: x \to x_S$$
  
 $y \to y_S$   
 $z \to 2 * y_S$   
 $π: (x_S = 2 * y_S) ∧ (x_S < y_S + 10)$ 

Find  $x_s$  and  $y_s$  that satisfy

- $\bullet$   $x_s = 2 * y_s$  and
  - $x_s < y_s + 10$



<pre>void buggy(int x, int y)</pre>	{
int i = 10;	
int $z = y * 2;$	
if (z == x) {	
if $(x >= y + 10)$ {	
z = z / (i - 10);	
}	
}	
}	

#### Path 2

$$\sigma: x \to x_s y \to y_s z \to 2 * y_s \pi: x_s \neq 2 * y_s$$

Find  $x_s$  and  $y_s$  that satisfy  $x_s = 1$   $y_s = 0$ 

• 
$$x_s \neq 2 * y_s$$

$$\begin{array}{c} x_s = 1 \\ y_s = 0 \end{array}$$



Program is completely tested; all paths and corresponding inputs are discovered

## **SMT Solver**

## Constraint solving

POSTECH

- We manually solved the path constraints
- To automate symbolic execution, the constraints should be solved by a machine (computer)
- There exist "solvers" for this task

## Satisfiability

- Satisfiability (SAT) is the problem of determining if there exists an assignment of values to variables that makes a given Boolean formula true
  - Example formula:  $(A \lor \neg B) \land (B \lor C)$
  - A, B, and C are Boolean variables
    - Can be assigned to either true or false
  - Satisfiability assignment:
    - A = true, B = false, C = true (one of the viable solutions)

## Satisfiability Modulo Theories (SMT)

**POSTECH** 

- SMT extends the SAT problem to more complex domains
  - Including theorems for arithmetic, bit-vectors, and arrays
- SMT solvers determine the satisfiability of logical formulas
  - Example formula:  $(x = 2 * y) \land (x \ge y + 10)$
  - Satisfiable assignment:
    - x = 20, y = 10 (one of the viable solutions)

For symbolic execution, we can utilize existing SMT solvers

## Example: Z3 solver

- POSTECH
- A widely-used SMT solver developed by Microsoft Research
- Using Z3 (with its Python binding)
  - Installation

```
$ pip3 install z3-solver
```

• Usage

```
# sat.py
from z3 import *
x = Int("x")
y = Int("y")
solve(x == 2 * y, x >= y + 10)
```

```
$ python3 sat.py
[y = 10, x = 20]
```

```
# unsat.py
from z3 import *
x = Int("x")
y = Int("y")
solve(x == 2 * y, x != 2 * y)
```

\$ python3 unsat.py
no solution

# KLEE: A Symbolic Execution Engine

## KLEE (OSDI '08)

POSTECH

- Cristian Cadar, et al.,
   "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs",
   OSDI, 2008
  - One of the most popular open-source symbolic execution engines

- Installation
  - Recommended: Docker with KLEE pre-installed

```
$ docker pull klee/klee:3.0
$ docker run --rm -ti --ulimit='stack=-1:-1' klee/klee:3.0
klee@[container_id]:~$
```

Target program: Example from the previous lecture

#### target.c

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void bug(void) {
   printf("bug!\n");
   raise(SIGSEGV);
}

int main(void) {
   setvbuf(stdout, NULL, _IONBF, 0);
   setvbuf(stdin, NULL, _IONBF, 0);
```

```
char in[16];
FILE *fp = fopen("/dev/stdin", "rb");
fread(&in, 4, 1, fp);

if (in[0] == '\xde') {
   if (in[1] == '\xad') {
     if (in[2] == '\xbe') {
      if (in[3] == '\xef') {
       bug();
     }
   }
  }
}

fclose(fp);
return 0;
}
```

- Specify symbolic inputs
  - We want to find a 4-byte string that triggers the bug

#### target.c

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <assert.h>

void bug(void) {
   printf("bug!\n");
   // raise(SIGSEGV);
   assert(0);
}

int main(void) {
   setvbuf(stdout, NULL, _IONBF, 0);
   setvbuf(stdin, NULL, _IONBF, 0);
```

```
char in[4];
// FILE *fp = fopen("/dev/stdin", "rb");
// fread(&in, 4, 1, fp);
klee make symbolic(in, 4, "in");
if (in[0] == '\xde') {
  if (in[1] == '\xad') {
    if (in[2] == '\xbe') {
      if (in[3] == '\xef') {
        bug();
// fclose(fp);
return 0;
```

### Compile target and run KLEE

```
klee@[container id]:~$ clang -I klee src/include -emit-llvm -g -c target.c
klee@[container id]:~$ klee target.bc
KLEE: output directory is "/home/klee/klee-out-0"
KLEE: Using STP solver backend
KLEE: SAT solver: MiniSat
KLEE: WARNING: undefined reference to function: printf
KLEE: WARNING ONCE: calling external: printf(94191341347320, 94191341347184) at target.c:19 7
r:
bug!
KLEE: ERROR: target.c:9: ASSERTION FAIL: 0
KLEE: NOTE: now ignoring this error at this location
KLFF: done: total instructions = 41
KLEE: done: completed paths = 4
KLEE: done: partially completed paths = 1
KLEE: done: generated tests = 5
```

### Check KLEE-generated test cases

```
klee@[container id]:~$ cd klee-last
klee@9048d3ab7cf9:~/klee-last$ ls | grep ktest
test0000001.ktest test0000002.ktest test0000003.ktest test0000004.ktest test0000005.ktest
klee@[container id]:~/klee-last$ ktest-tool test000005.ktest
ktest file : 'test000005.ktest'
           : ['target.bc']
args
num objects: 1
object 0: name: 'in' The input we marked symbolic
object 0: size: 4
object 0: data: b'\xde\xad\xbe\xef' Exact value of the symbolic input for the path
klee@[container id]:~/klee-last$ cat test000005.assert.err
Error: ASSERTION FAIL: 0
File: target.c
line: 9
                          Detected error and the stack trace
assembly.ll line: 17
State: 1
Stack:
        #000000017 in bug() at target.c:9
        #100000065 in main() at target.c:23
```

# Limitations of Symbolic Execution

POSTECH

- Loops and recursions
  - Leads to infinite execution tree
- Path explosion
  - Number of paths exponentially increase
- SMT solver limitations
  - Complex path constraints cannot be solved
- Environment modeling
  - System calls, library calls, file operations, ...

POSTECH

- Loops and recursions
  - Leads to infinite execution tree

```
void loopy(int x, int y) {
  int i = 0;
  while (i < 500) {
    if (x + i > 10 * y) {
      bug();
    }
    i++;
  }
}
```

As the loop repeats, path constraint becomes massive, e.g.,

 $\pi: (x_0 > 10 * y_0) \land (x_0 + 1 > 10 * y_0) \land (x_0 + 2 > 10 * y_0) \land \cdots$ 

POSTECH

- Path explosion
  - Symbolic executor forks the program under test at every branch
    - Results in two copies of the execution states per branch
  - Number of paths exponentially increase due to nested branches

POSTECH

- Environment modeling
  - How to deal with external calls?
     e.g., system calls, library calls, file operations, ...

```
void read_pixels(int width, int height) { → assume the parameters are symbolic
    char pixel_buf[1024];
    int fd = open("/tmp/image.png", 0_RDWR);
    ssize_t num_bytes = read(fd, pixel_buf, width + height);
    if (num_bytes == -1) {
        assert(0);
    }
}
```

We cannot symbolically represent num\_bytes in terms of width and height as it depends on the actual size of the image file

No path constraint can be derived for the if branch



- SMT solver limitations
  - Solvers are not omni-potent
  - Some path constraints require long time to be solved
  - Complex path constraints cannot be solved at all

Combined with the path explosion problem, a complete analysis of a large and complex program is often infeasible

# **Practical Solutions**

#### Concolic execution



- Concolic = Concrete + Symbolic
  - Also called dynamic symbolic execution
  - Program is executed simultaneously with both concrete and symbolic inputs
  - Concrete inputs help dealing with external calls (e.g., read file)
  - Symbolic inputs help exploring branches

### Concolic execution

Concolic = Concrete + Symbolic

```
void read_pixels(int width, int height) {
  char pixel_buf[1024];
  int fd = open("/tmp/image.png", 0_RDWR);
  ssize_t num_bytes = read(fd, pixel_buf, width + height);
  if (num_bytes == -1) {
    assert(0);
  }
}
```

- Concrete execution reveals the file size of /tmp/image.png, i.e., actual\_sz
- It also reveals the semantics of read syscall:
  - if (width + height) >= actual\_sz, then num\_bytes = actual\_sz
     → num\_bytes is concrete, therefore the if branch is not taken
  - else, num\_bytes = width + height
     num\_bytes is symbolic, therefore symbolic execution can solve the path constraint of the if branch

# Hybrid fuzzing

POSTECH

- Idea: Use symbolic execution for difficult branches and fuzzing to resolve path explosion
  - Run a fuzzer until code coverage saturates at one point
  - Run symbolic execution to find the input to get past the branch
  - Use that concrete input as seed and continue fuzzing

# Hybrid fuzzing

 Idea: Use symbolic execution for difficult branches and fuzzing to resolve path explosion

```
int x; // user input
                                 char buf[32]; // user input
     Fuzzing coverage
                                 if (x == 0 \times deadbeef) \{ // hard for fuzzing <math>\frac{1}{2^{32}} chance to randomly generate correct x
       saturates here
                                   int count = 0:
                                   for (int i = 0; i < 32; i++) {
Symbolic executor engages
                                     if (buf[i] >= 'a') {
and finds x = 0x deadbeef
                                        count++;
    Fuzzer mutates buf
                                   if (count >= 8) { // hard for symbolic execution
                                     /* · · · */
      and easily enters
                                                           No. of feasible paths = 2^{32} (two for each element of buf[32])
         the branch
                                                           → Path explosion!
```

#### . POSTPEL

# Summary

- Bug finding is crucial for securing computer systems
  - Manual analysis can be daunting as modern systems have become too large and complex
- Greybox fuzzing aims to provide soundness
  - It finds real bugs, but misses existing bugs
- Symbolic execution aims to provide completeness
  - In theory, it finds all bugs by exploring all program paths
  - However, complete analysis is impossible due to practical limitations
- Both techniques are widely used in practice
  - Various combinations of the two are being proposed to achieve soundness and completeness at the same time

# Questions?