Lec 26: Symbolic Execution

CSED415: Computer Security
Spring 2024

Seulbae Kim

POSTECH
POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY

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**

Presentation order

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

```
import random
import time

random.seed(time.time())
N = 10 # to be selected in class

teams = [
    "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

- 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?

Concrete (dynamic) vs Symbolic execution

- 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 path constraint 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

- 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, σ, π)
  - st: The next statement to evaluate
  - σ: Symbolic store
  - π: Path constraints
- Depending on st, symbolic execution proceeds as follows:
  - st is an **assignment** (e.g., $var = e$):
    - σ 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)):
    - The validity of e is checked using path constraints

Example of symbolic execution

```
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
    }
  }
}
```

σ: Symbolic store

π: Path constraints

Example of symbolic execution

st →

```
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

σ: Symbolic store

$x \rightarrow x_s$

$y \rightarrow y_s$

(Notation: $var \rightarrow sym$)

π: Path constraints

$true$

(branch always taken)

Example of symbolic execution

st →

```
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

σ: Symbolic store

$x \rightarrow x_s$

$y \rightarrow y_s$

π: Path constraints

$true$

Example of symbolic execution

st →

```
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

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

σ: Symbolic store

$x \rightarrow x_s$

$y \rightarrow y_s$

$z \rightarrow 2 * y_s$

π: Path constraints

$true$

Example of symbolic execution

st →

```
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
    }
  }
}
```

path 1: if taken

path 2: if not taken

st is an if statement

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

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

Path 1 (if taken)

σ: Symbolic store

$x \to x_s$

$y \to y_s$

$z \to 2 * y_s$

π: Path constraints

$x_s = 2 * y_s$

Path 2 (if not taken)

σ: Symbolic store

$x \to x_s$

$y \to y_s$

$z \to 2 * y_s$

π: Path constraints

$x_s \neq 2 * y_s$

Example of symbolic execution

Path 1

σ: Symbolic store	π: Path constraints
$x \rightarrow x_s$ $y \rightarrow y_s$ $z \rightarrow 2 * y_s$	$x_s = 2 * y_s$

st →

```
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 2 is followed

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

Final states

Path 2

σ: $x \rightarrow x_s$, $y \rightarrow y_s$, $z \rightarrow 2 * y_s$

π: $x_s \neq 2 * y_s$

Example of symbolic execution

st →

```
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
    }
  }
}
```

path 1-1: if taken

path 1-2: if not taken

st is an if statement

Path 1-1

σ: $x \rightarrow x_s$
$y \rightarrow y_s$
$z \rightarrow 2 * y_s$

π: $(x_s = 2 * y_s) \wedge (x_s \geq y_s + 10)$

Path 1-2

σ: $x \rightarrow x_s$
$y \rightarrow y_s$
$z \rightarrow 2 * y_s$

π: $(x_s = 2 * y_s) \wedge (x_s < y_s + 10)$

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

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

Final states

Path 2

σ: $x \rightarrow x_s$
$y \rightarrow y_s$
$z \rightarrow 2 * y_s$

π: $x_s \neq 2 * y_s$

Example of symbolic execution

```
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 → (pointing at the closing brace of the inner `if`)

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

σ: $x \rightarrow x_s$
$y \rightarrow y_s$
$z \rightarrow 2 * y_s$

π: $(x_s = 2 * y_s) \wedge (x_s \geq y_s + 10)$

Final states

Path 2

σ: $x \rightarrow x_s$
$y \rightarrow y_s$
$z \rightarrow 2 * y_s$

π: $x_s \neq 2 * y_s$

Path 1-2

σ: $x \rightarrow x_s$
$y \rightarrow y_s$
$z \rightarrow 2 * y_s$

π: $(x_s = 2 * y_s) \wedge (x_s < y_s + 10)$

Example of symbolic execution

st →

```
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

σ 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

σ: $x \rightarrow x_s$
$y \rightarrow y_s$
$z \rightarrow 2 * y_s/0$

π: $(x_s = 2 * y_s) \wedge (x_s \geq y_s + 10)$

Final states

Path 2

σ: $x \rightarrow x_s$
$y \rightarrow y_s$
$z \rightarrow 2 * y_s$

π: $x_s \neq 2 * y_s$

Path 1-2

σ: $x \rightarrow x_s$
$y \rightarrow y_s$
$z \rightarrow 2 * y_s$

π: $(x_s = 2 * y_s) \wedge (x_s < y_s + 10)$

Example of symbolic execution

```
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 → (final line `}`)

All program paths have been explored

Final states

Path 1-1

σ: $x \rightarrow x_s$
$y \rightarrow y_s$
$z \rightarrow 2 * y_s / 0$

π: $(x_s = 2 * y_s) \wedge (x_s \geq y_s + 10)$

→ Potential div-by-zero error is detected! If π is satisfiable, this is an actual bug

Path 1-2

σ: $x \rightarrow x_s$
$y \rightarrow y_s$
$z \rightarrow 2 * y_s$

π: $(x_s = 2 * y_s) \wedge (x_s < y_s + 10)$

Path 2

σ: $x \rightarrow x_s$
$y \rightarrow y_s$
$z \rightarrow 2 * y_s$

π: $x_s \neq 2 * y_s$

Next step: Solving π to obtain concrete test inputs for each path

Example of symbolic execution

	Solving π	Concrete input
Path 1-1 σ: $x \rightarrow x_s$ $y \rightarrow y_s$ $z \rightarrow 2 * y_s\ /0$ π: $(x_s = 2 * y_s) \wedge (x_s \geq y_s + 10)$	Find x_s and y_s that satisfy • $x_s = 2 * y_s$ and • $x_s \geq y_s + 10$	$x_s = 20$ $y_s = 10$
Path 1-2 σ: $x \rightarrow x_s$ $y \rightarrow y_s$ $z \rightarrow 2 * y_s$ π: $(x_s = 2 * y_s) \wedge (x_s < y_s + 10)$	Find x_s and y_s that satisfy • $x_s = 2 * y_s$ and • $x_s < y_s + 10$	$x_s = 0$ $y_s = 0$
Path 2 σ: $x \rightarrow x_s$ $y \rightarrow y_s$ $z \rightarrow 2 * y_s$ π: $x_s \neq 2 * y_s$	Find x_s and y_s that satisfy • $x_s \neq 2 * y_s$	$x_s = 1$ $y_s = 0$

Verification?

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

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

SMT Solver

Constraint solving

- 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: $(A \vee \neg B) \wedge (B \vee C)$
  - 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)

- 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: $(x = 2 * y) \land (x \geq y + 10)$
  - Satisfiable assignment:
    - $x = 20,\ y = 10$ (one of the viable solutions)

For symbolic execution, we can utilize existing SMT solvers

Example: Z3 solver

- 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)

- 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

Using KLEE

- 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]:~$
```

Using KLEE

- Target program: Example from the previous lecture

target.c

```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;
}
```

Using KLEE

- 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;
}
```

Using KLEE

- 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

KLEE: done: total instructions = 41
KLEE: done: completed paths = 4
KLEE: done: partially completed paths = 1
KLEE: done: generated tests = 5
```

Using KLEE

- Check KLEE-generated test cases

```
klee@[container_id]:~$ cd klee-last
klee@9048d3ab7cf9:~/klee-last$ ls | grep ktest
test000001.ktest test000002.ktest test000003.ktest test000004.ktest test000005.ktest

klee@[container_id]:~/klee-last$ ktest-tool test000005.ktest
ktest file : 'test000005.ktest'
args       : ['target.bc']
num objects: 1
object 0: name: 'in'
object 0: size: 4
object 0: data: b'\xde\xad\xbe\xef'

klee@[container_id]:~/klee-last$ cat test000005.assert.err
Error: ASSERTION FAIL: 0
File: target.c
Line: 9
assembly.ll line: 17
State: 1
Stack:
        #000000017 in bug() at target.c:9
        #100000065 in main() at target.c:23
```

The input we marked symbolic

Exact value of the symbolic input for the path

Detected error and the stack trace

Limitations of Symbolic Execution

Practical issues 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, …

Practical issues of symbolic execution

- 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.,

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

Practical issues of symbolic execution

- 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

Practical issues of symbolic execution

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

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

→ assume the parameters are symbolic

→ open and read are external functions (syscalls)

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

Practical issues of symbolic execution

- 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", O_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

Fuzzing coverage saturates here → `if (x == 0xdeadbeef)`

Symbolic executor engages and finds `x = 0xdeadbeef`

Fuzzer mutates buf and easily enters the branch → `if (count >= 8)`

```
int x; // user input
char buf[32]; // user input
if (x == 0xdeadbeef) { // hard for fuzzing
  int count = 0;
  for (int i = 0; i < 32; i++) {
    if (buf[i] >= ‘a’) {
      count++;
    }
  }
  if (count >= 8) { // hard for symbolic execution
    /* ... */
  }
}
```

$\frac{1}{2^{32}}$ chance to randomly generate correct x

No. of feasible paths = 2^{32} (two for each element of buf[32])
→ Path explosion!

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?