CSC2/455 Software Analysis and Improvement
Symbolic Execution

Sreepathi Pai
April 17, 2023

URCS
Outline

Basic Ideas

KLEE

Angr
Basic Ideas

KLEE

Angr
Concrete Execution

- When programs run, every value they encounter is *concrete*
- We don’t use concrete executions for analyses that must hold over all executions
  - A run is only one sample
- Static analysis results hold over all possible executions
  - Compromise: (Sound) approximations of values and paths
  - Will never claim a property is true when it isn’t
  - May claim a property is false when it is in fact true
- What if we “approximate” values by symbols?
  - Symbolic Execution
Symbolic Execution

- Originated in the 1970s
  - See King, “Symbolic Execution and Testing”, CACM 1976
- Resurgence in the last two decades
  - Underpinned by SMT solvers
- Closely related to Model Checking
Symbolic Execution Engines (or Symbolic Virtual Machines)

- Two main components of symbolic execution
  - A symbolic store (i.e. memory)
  - A symbolic path condition (a boolean formula that represents, in some ways, the symbolic program counter)
- Many symbolic execution engines available
  - Angr (for binary analysis)
  - KLEE (for analyzing C programs, but also LLVM bitcode)
  - Manticore (for analyzing Ethereum contracts, Linux binaries, Webassembly)
def min(a, b):
    if a < b:
        x = a
    else:
        x = a

    assert x == a or x == b
    assert x <= a and x <= b

In the program above, normally a and b have concrete values. Let’s make them symbolic:

- $a = \alpha$
- $b = \beta$
- Let $m(x)$ represent the memory store, retrieving the value for variable $x$
def min(a, b):

- At entry:
  - $m = \{a \mapsto \alpha, b \mapsto \beta\}$
  - $\pi = true$
if(a < b):
    x = a

- **Just before** $x = a$:
  - $m = \{a \mapsto \alpha, b \mapsto \beta\}$
  - $\pi = \alpha < \beta$

- **After** $x = a$:
  - $m = \{a \mapsto \alpha, b \mapsto \beta, x \mapsto \alpha\}$
  - $\pi = \alpha < \beta$
else:
    x = a

- Just before $x = a$:
  - $m = \{a \mapsto \alpha, b \mapsto \beta\}$
  - $\pi = \neg(\alpha < \beta)$

- After $x = a$:
  - $m = \{a \mapsto \alpha, b \mapsto \beta, x \mapsto \alpha\}$
  - $\pi = \neg(\alpha < \beta)$

- We have two path conditions now, one for the true part and one for the false part
  - The program has “forked”
Symbolically executing the true path

assert x == a or x == b
assert x <= a and x <= b

- Here:
  - \( \pi = \alpha < \beta \)
  - \( m = \{ a \mapsto \alpha, b \mapsto \beta, x \mapsto \alpha \} \)

- The symbolic conditions are (after substituting symbolic values from the store):
  - Prove \( (\alpha = \alpha \lor \alpha = \beta) \) is always true when \( (\alpha < \beta) \)
  - Prove \( (\alpha \leq \alpha \land \alpha \leq \beta) \) is always true when \( (\alpha < \beta) \)

- I.e. the following are UNSAT
  - \( \alpha \neq \alpha \land \alpha \neq \beta \land \alpha < \beta \)
  - \( (\alpha > \alpha \lor \alpha > \beta) \land \alpha < \beta \)
Symbolically executing the false path

assert x == a or x == b
assert x <= a and x <= b

- Here:
  - $\pi = \neg(\alpha < \beta)$
  - $m = \{a \mapsto \alpha, b \mapsto \beta, x \mapsto \alpha\}$

- The symbolic conditions are (after substituting symbolic values from the store):
  - Prove $(\alpha = \alpha \lor \alpha = \beta)$ is always true when $\neg(\alpha < \beta)$
  - Prove $(\alpha \leq \alpha \land \alpha \leq \beta)$ is always true when $\neg(\alpha < \beta)$

- I.e. the following are UNSAT
  - $\alpha \neq \alpha \land \alpha \neq \beta \land \alpha \geq \beta$
  - $(\alpha > \alpha \lor \alpha > \beta) \land \alpha \geq \beta$, but this not unsat!
The Symbolic Execution Graph

The Symbolic Execution Engine explores a graph:

- Nodes are program statements
- Edges are labeled with path conditions
def min3(a, b, c):
    if a < b:
        mn = c

        if c < a:
            mn = c
    else:
        mn = b

        if c < b:
            mn = c

    assert mn == a or mn == b or mn == c
    assert mn <= a and mn <= b and mn <= c
You can reach the assert statements under four different path conditions

Each conditional doubles the number of paths
Symbolic Execution: Expressions

\[ y = 1 \]
\[ x = x + y \]
\[ z = x \times 3 \]

- \( y \mapsto 1 \)
- \( x \mapsto \alpha_x + 1 \) (where \( m(x) = \alpha_x \))
- \( z \mapsto (\alpha_x + 1) \times 3 \)
Symbolic Execution: Loops

\[ i = 0 \]
\[ j = k \]
\[ \text{while } i < 3: \]
\[ \quad k = k + 1 \]
\[ \quad i = i + 1 \]
\[ \text{assert } k - j \geq 3 \]

- Each iteration adds to the path condition
  - \( \pi_0 = 0 < 3 \)
  - \( \pi_1 = 0 < 3 \land 1 < 3 \)
  - \( \pi_4 = 0 < 3 \land 1 < 3 \land \cdots \land 4 < 3 \)

- For the assert statement, the path conditions are negations:
  - \( \pi'_0 = \neg(0 < 3) \) (this is false, no further exploration)
  - \( \pi'_1 = \neg(0 < 3 \land 1 < 3) \) (this is false, no further exploration)
  - \( \pi'_4 = \neg(0 < 3 \land 0 < 3 \land \cdots \land 4 < 3) \) (this is true)

- For \( \pi'_4 \), the store is \( \{ i \mapsto 3, k \mapsto \kappa + 3, j \mapsto \kappa \} \)
- (assume \( k = \kappa \) initially)
Infinite/Symbolic loops

- When it cannot be proved that a path condition is false, the symbolic execution engine must continue exploring it.
- This leads to a “state space explosion”
Non-basic Ideas

- Change path exploration
  - Don’t use depth-first search
  - Random paths
  - etc.

- Concretize values
  - A mix of symbolic and concrete values ("concolic" execution)
  - May underapproximate paths executed

- Usually justified by noting we’re looking for an executable path containing a bug
  - I.e. not trying to prove absence of bugs

- Merging states

- Lots of other “tricks”
Implementing A Symbolic Execution Engine

- Ball and Daniel, Deconstructing Dynamic Symbolic Execution
  - https://www.github.com/thomasjball/PyExZ3/
  - for Python, in Python
Basic Ideas

KLEE

Angr
EXE and KLEE

- Cadar, Ganesh, Dill, and Engler, EXE: Automatically Generating Inputs of Death, CCS 2006
  - “This paper presents EXE, an effective bug-finding tool that automatically generates inputs that crash real code...”
- Cadar, Dunbar, and Engler, KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs, OSDI 2008
  - “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.”
  - “We used KLEE to cross-check purportedly identical BUSYBOX and COREUTILS utilities, finding functional correctness errors and a myriad of inconsistencies”
- See also the CACM paper referenced earlier by these authors
Using KLEE

- Download and compile from the KLEE website
  - Build Instructions
- Then, use LLVM to generate bitcode
```c
#include <klee/klee.h>
#include <assert.h>

int min3(int a, int b, int c) {
    int mn;
    if(a < b) {
        mn = a;
        if(c < a)
            mn = c;
    }
    else {
        mn = b;
        if(c < b)
            mn = c;
    }
    return mn;
}

int main() {
    int a, b, c;
    klee_make_symbolic(&a, sizeof(a), "a");
    klee_make_symbolic(&b, sizeof(b), "b");
    klee_make_symbolic(&c, sizeof(c), "c");
    return min3(a, b, c);
}
```
$ klee min3.bc
KLEE: output directory is "/src/klee-out-2"
KLEE: Using Z3 solver backend

KLEE: done: total instructions = 63
KLEE: done: completed paths = 4
KLEE: done: generated tests = 4
$ ktest-tool klee-last/test00000?.ktest

object 0: name: 'a'
object 0: int : 0

object 1: name: 'b'
object 1: uint: 0

object 2: name: 'c'
object 2: int : 0

• a = 0, b = 1, c = 0
• a = 1073741824, b = 1, c = 0
• a = 1, b = 2, c = 0
Adding asserts (and a bug!)

```c
#include <klee/klee.h>
#include <assert.h>

int min3(int a, int b, int c) {
    int mn;
    if(a < b) {
        mn = a;
        if(c < a)
            mn = c;
    }
    else {
        mn = b;
        if(c < b)
            mn = b;
    }
    assert(mn == a || mn == b || mn == c);
    assert(mn <= a && mn <= b && mn <= c);
    return mn;
}

int main() {
    int a, b, c;
    klee_make_symbolic(&a, sizeof(a), "a");
    klee_make_symbolic(&b, sizeof(b), "b");
    klee_make_symbolic(&c, sizeof(c), "c");
    return min3(a, b, c);
}
```
Running KLEE

KLEE: output directory is "src/klee-out-10"
KLEE: Using Z3 solver backend
KLEE: ERROR: min3asserts.c:18: ASSERTION FAIL: mn <= a && mn <= b && mn <= c
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 169
KLEE: done: completed paths = 6
KLEE: done: generated tests = 5

- a = 1, b = 1, c = 0
KLEE on Coreutils

- See tutorial: https://klee.github.io/tutorials/testing-coreutils/
Outline

Basic Ideas

KLEE

Angr
• Angr is a symbolic execution engine for x86-64 code
• It is used from within Python3
  • Very easy to install
• Angr is a very programmable symbolic execution engine
In a Mission (Not So) Impossible scenario, we have a binary crackme, which can only be unlocked using a password. The password is in the binary, but it is encrypted. Can we figure it out?
Control Flow Graph

```
check_password:
  ret_addr 0x8
  004005b0  mov  r11, [passwd]
  004005b7  xor  r8d, r8d
  004005ba  xor  ecx, ecx
  004005bc  xor  r9d, r9d
  004005bf  mov  eax, 0x1
  004005c4  jmp  0x4005f9

loc_0x4005f9:
  004005f9  movsx edx, [r11+r8]
  004005fe  test dl, dl
  00400600  jne  0x4005d0

loc_0x4005d0:
  004005d0  movsx esi, [rdi+r8]
  004005d5  test sil, sil
  004005d8  je  0x400602

loc_0x400602:
  00400602  ret

loc_0x4005da:
  004005da  movsx  r10, ecx
  004005dd  xor  edx, [r10+0x4+otp]
  004005e5  cmp  edx, esi
  004005e7  cmovne eax, r9d
  004005eb  add  ecx, 0x1
  004005ee  cmp  ecx, 0xc
  004005f1  cmovne ecx, r9d
  004005f5  add  r8, 0x1
```
proj = angr.Project(args.binary, auto_load_libs=False)
cfg = proj.analyses.CFG()

cpfn = find_function("check_password", cfg)
s = proj.factory.blank_state(addr=cpfn.addr)

- This creates an Angr project, by loading the binary, and finding the address of check_password in it
- Then we create a blank symbolic state, with program counter at the entry to check_password
Symbolically Executing crackme

```
simgr = proj.factory.simulation_manager(s)
simgr.explore(find=0x400602, num_find = 20)
```

- We start symbolic execution and explore all states until we reach a state where the PC is 0x400602
  - The is the return address
- Angr returns the first state it finds, but we ask for more (up to 20)
Results

(Pdb) simgr.found
[<SimState @ 0x400602>,
 <SimState @ 0x400602>,
 <SimState @ 0x400602>,
 <SimState @ 0x400602>,
 <SimState @ 0x400602>,
 <SimState @ 0x400602>,
 <SimState @ 0x400602>,
 <SimState @ 0x400602>,
 <SimState @ 0x400602>,
 <SimState @ 0x400602>,
 <SimState @ 0x400602>,
 <SimState @ 0x400602>,
 <SimState @ 0x400602>,
 <SimState @ 0x400602>,
 <SimState @ 0x400602>,
 <SimState @ 0x400602>,
 <SimState @ 0x400602>,
 <SimState @ 0x400602>,
 <SimState @ 0x400602>,
 <SimState @ 0x400602>,
 <SimState @ 0x400602>]

- There are thirteen states that exit the function
Which basic blocks were executed?

(Pdb) simgr.found[0].history.bbl_addrs.hardcopy
[0x4005b0, 0x4005f9, 0x4005d0]
What was the (symbolic) return value?

(Pdb) simgr.found[1].history.bbl_addrs.hardcopy
[0x4005b0, 0x4005f9, 0x4005d0, 0x4005da, 0x4005d0]

(Pdb) simgr.found[1].regs.eax
<BV32 if memffffffffffffff000_13_8{UNINITIALIZED} == 72 &&
  memffffffffffffff000_13_8{UNINITIALIZED}[7:7] == 0 then
  0x1 else 0x0>

- This is a symbolic if-then-else expression
  - eax contains 1 or 0
  - if the expression involving memory is true
  - (the memory address contains 72)
(Pdb) simgr.found[2].history.bbl_addrs.hardcopy
[0x4005b0, 0x4005f9, 0x4005d0, 0x4005da, 0x4005d0, 0x4005da, 0x4005d0]

(Pdb) simgr.found[2].regs.eax
<BV32 if mem_fffffffffffff001_14_8{UNINITIALIZED} == 101 &&
mem_fffffffffffff001_14_8{UNINITIALIZED}[7:7] == 0 then
(if mem_fffffffffffff000_13_8{UNINITIALIZED} == 72 &&
mem_fffffffffffff000_13_8{UNINITIALIZED}[7:7] == 0
then 0x1 else 0x0)
else 0x0>

• This is still a symbolic if-then-else expression
  • eax contains 1 or 0 ultimately
  • if the expression involving memory is true
  • (the memory address contains 72 and the address adjacent
    contains 101)
  • 'He'
(Pdb) simgr.found[12].solver.add(simgr.found[12].regs.eax == 1)

- We're adding a constraint that on exit eax is 1
  - Presumably, indicating successful match
- We then ask the solver to solve the symbol in rsi
  - rsi held the characters of the password during comparison
  - This returns 0x21, i.e. 33, i.e. ’!’
- But it also concretizes the memory
Looking at the concretized memory

(Pdb) simgr.found[12].solver.eval(simgr.found[12].regs.rdi)
0xffffffffffff000
(Pdb) simgr.found[12].solver.eval(simgr.found[12].memory.load(simgr.found[12].regs.rdi, 12), cast_to=bytes)
b’Hello, World!’
int check_password(const char *u) {
    int i = 0;
    int j = 0;
    int succ = 1;

    while(passwd[i] != '\0' && *u != '\0') {
        if((passwd[i] ^ otp[j]) != *u) {
            succ = 0;
        }
        i++;
        u++;
        j++;
        if(j == n_otp) j = 0;
    }

    return succ;
}
Symbolic Execution can be a powerful analysis tool
  - Used alone or in conjunction with other tools
- Try out the tools we talked about today!