Symbolic Execution with Angr
RPISEC

Avi Weinstock (aweinstock), Luke Biery (tiecoon)

December 6, 2019

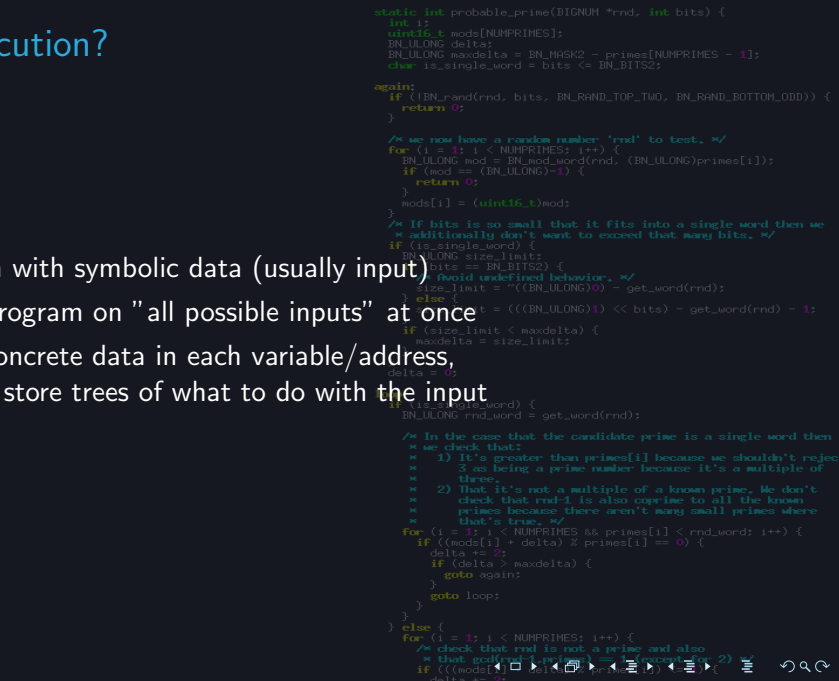
Overview

- What is Symbolic Execution? What techniques does it compete with?
- How symbolic execution works (theory)
- How symbolic execution works (Angr commands)
- Solving MBE lab1A with Angr

Background - What it is and what is the problem space?

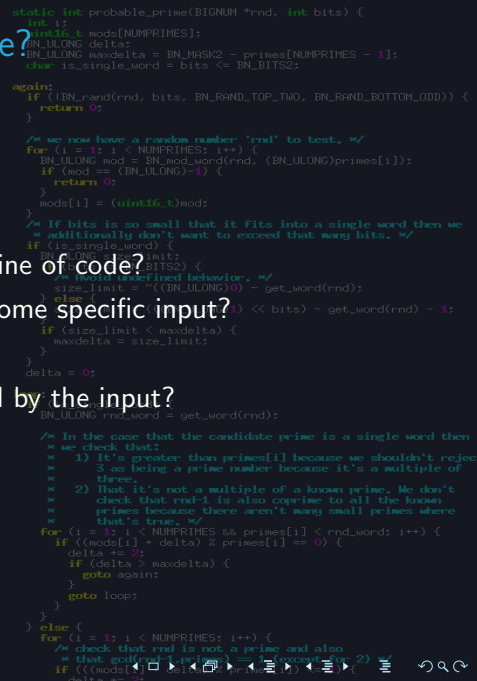
What is Symbolic Execution?

- Executes a program with symbolic data (usually input)
- Essentially runs a program on "all possible inputs" at once
- Instead of having concrete data in each variable/address, variables/addresses store trees of what to do with the input



What problems does Symbolic Execution solve?

- What input to provide to reach/avoid a specific line of code?
- How is a value deep in the program affected by some specific input?
- Do any inputs lead to any crash?
- On a crashing input, what registers are controlled by the input?



Symbolic Execution vs Fuzzing

Symbolic Execution	Fuzzing
+ Explores all inputs	- Only explores random inputs
+ Very detailed output	- Only learn crash vs non-crash
- Uses more memory/time	+ Uses around as much memory/time as target program

- Symbolic execution can the path `if(input == 0xdeadbeefdeadbeef) { ... }`
- Even coverage-guided fuzzing will only find it $\frac{1}{2^{64}}$ of the time[1]

[1] Unless the compare is digit-by-digit

How symbolic execution works in general

Setting up a state for symbolic execution

```
import z3
registers = ['eax', 'ebx', 'ecx', 'edx', 'ebp', 'esp'] # and so on
symstate = {reg: z3.BitVec(reg, 32) for reg in registers}
symstate['memory'] = z3.Array('memory', z3.BitVecSort(32), z3.BitVecSort(8))
```

- Note that the z3 variable eax in the model will be the starting value of eax
- `symstate['eax']` will be mutated throughout the computation, and will contain an expression corresponding to the ending value of eax

z3.Array vs dict of z3.BitVec for representing memory

- `memory = z3.Array('memory', z3.BitVecSort(32), z3.BitVecSort(8))` symbolically represents an array of 2^{32} bytes (around 4GB)
- `z3.Store(memory, index, value)` represents a modified memory (with `value` written to `index`), even with *symbolic* `index` and `value`
- `memory[index]` represents a read from memory, even if `index` is symbolic
- `memory = {i: z3.BitVec('mem[{i}]'.format(i=i), 8) for i in idxs}` only allows concrete indices, while still allowing symbolic values, and is more efficient when we know we won't have symbolic-indexed reads/writes

Symbolically executing branch-free code

- Translate arithmetic, indexing, etc into SMT constraints
- Angr internally uses VEX for this instead of translating x86 directly

`mov eax, ebx`	`symstate['eax'] = symstate['ebx']`
`add ecx, edx`	`symstate['ecx'] += symstate['edx']`

`mov byte [esp+0x10], al`	`esp_10 = symstate['esp']+0x10`
	`al = z3.Extract(7, 0, symstate['eax'])`
	`symstate['memory'] = z3.Store(symstate['memory'], esp_10, al)`

`movsx eax, byte [eax]`	`star_eax = z3.Select(symstate['memory'], eax)` `symstate['eax'] = z3.SignExt(24, star_eax)`

Handling symbolic reads with z3.Array vs z3.BitVec

C:

```
tmp = username[i];
tmp ^= serial;
```

Assembly:

```
0x08048aee      mov edx, dword [local_14h]
0x08048af1      mov eax, dword [arg_8h]
0x08048af4      add eax, edx
0x08048af6      movzx eax, byte [eax]
0x08048af9      movsx eax, al
0x08048afc      xor eax, dword [local_10h]
```

List of z3.BitVec:

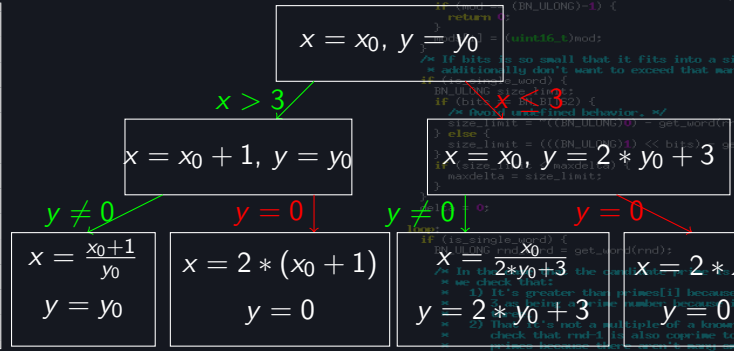
```
eax = z3.SignExt(24, sym_username[local_14h])
eax ^= local_10h
```

z3.Array:

```
local_14 = symstate['esp']+0x14 # &i
symstate['edx'] = symstate['memory'][local_14]
arg_8 = symstate['ebp']+0x8 # &username
symstate['eax'] = symstate['memory'][arg_8]
symstate['eax'] += symstate['edx']
symstate['eax'] = z3.ZeroExt(24,symstate['eax'])
al = z3.Extract(7, 0, symstate['eax'])
symstate['eax'] = z3.SignExt(24,al)
local_10 = symstate['esp']+0x10 # &serial
symstate['eax'] ^= symstate['memory'][local_10]
```

Symbolically executing branchs - Graphically

```
int f(int x, int y) {
    if (x > 3) {
        x += 1;
    } else {
        y = 2*y+3;
    }
    if(y != 0) {
        x /= y;
    } else {
        x *= 2;
    }
    return x + y;
}
```



Symbolically executing branchs - Programmatically

```
int f(int x, int y) {
    if (x > 3) {
        x += 1;
    } else {
        y = 2*y+3;
    }
    if(y != 0) {
        x /= y;
    } else {
        x *= 2;
    }
    return x + y;
}
```

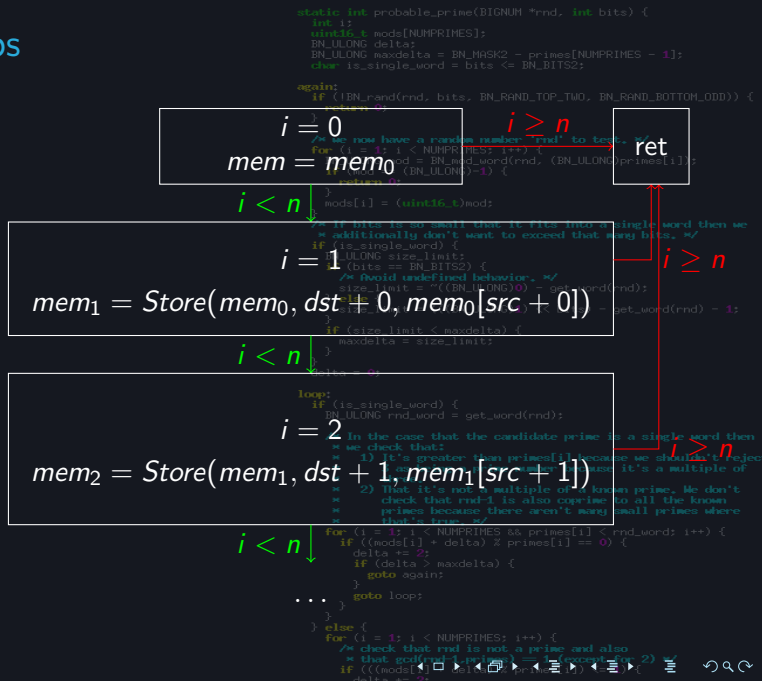
```
import z3
x0, y0 = z3.Ints('x0 y0')
states, newstates = [(x0, y0, z3.Solver())], []
for (x, y, s) in states:
  t = s.__deepcopy__()
  s.add(x > 3); newstates.append((x+1, y, s))
  t.add(z3.Not(x > 3)); newstates.append((x, 2*y+3, t))

states, newstates = newstates, []
for (x, y, s) in states:
  t = s.__deepcopy__()
  s.add(y != 0); newstates.append((x/y, y, s))
  t.add(z3.Not(y != 0)); newstates.append((2*x, y, t))

for (x, y, s) in newstates:
  print('x: %r; y: %r; s: %r; check: %r' % (x, y, s, s.check()))
  if s.check() == z3.sat:
    m = s.model()
    print('m: %r; x: %r; y: %r' % (m, m.evaluate(x), m.evaluate(y)))
    print('-'*5)
```

Symbolically executing loops

```
void memcpy(
    char *dest,
    const char *src,
    size_t n) {
    for(size_t i=0; i<n; i++) {
        dest[i] = src[i];
    }
}
```



How to use Angr for symbolic execution

TODO: Luke

- loading binariess
- marking input as symbolic
- initiating the search/pruning the search space
- simprocedures for shortcutting syscalls?

Example: MBE lab1A with Angr

Resources

- https://github.com/angr/
- https://github.com/Z3Prover/z3/
- https://github.com/RPISEC/MBE