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?

```c
static int probable_prime(BIGNUM *rnd, int bits) {
  int i;
  uint16_t mods[NUMPRIMES];
  BN_ULONG delta;
  BN_ULONG maxdelta = BN_MASK2 - primes[NUMPRIMES - 1];
  char is_single_word = bits <= BN_BITS2;

again:
  if (!BN_rand(rnd, bits, BN_RAND_TOP_TWO, BN_RAND_BOTTOM_ODD)) {
    return 0;
  }

  /* we now have a random number 'rnd' to test. */
  for (i = 1; i < NUMPRIMES; i++) {
    BN_ULONG mod = BN_mod_word(rnd, (BN_ULONG)primes[i]);
    if (mod == (BN_ULONG)-1) {
      return 0;
    }
    mods[i] = (uint16_t)mod;
  }
  /* If bits is so small that it fits into a single word then we
   * additionally don't want to exceed that many bits. */
  if (is_single_word) {
    BN_ULONG size_limit;
    if (bits == BN_BITS2) {
      /* Avoid undefined behavior. */
      size_limit = ~((BN_ULONG)0) - get_word(rnd);
    } else {
      size_limit = (((BN_ULONG)1) << bits) - get_word(rnd) - 1;
    }
    if (size_limit < maxdelta) {
      maxdelta = size_limit;
    }
  }
  delta = 0;

loop:
  if (is_single_word) {
    BN_ULONG rnd_word = get_word(rnd);

    /* In the case that the candidate prime is a single word then
     * we check that:
     *   1) It's greater than primes[i] because we shouldn't rejec
     *      3 as being a prime number because it's a multiple of
     *      three.
     *   2) That it's not a multiple of a known prime. We don't
     *      check that rnd-1 is also coprime to all the known
     *      primes because there aren't many small primes where
     *      that's true. */
    for (i = 1; i < NUMPRIMES && primes[i] < rnd_word; i++) {
      if ((mods[i] + delta) % primes[i] == 0) {
        delta += 2;
        if (delta > maxdelta) {
          goto again;
        }
        goto loop;
      }
    }
  } else {
    for (i = 1; i < NUMPRIMES; i++) {
      /* check that rnd is not a prime and also
       * that gcd(rnd-1,primes) == 1 (except for 2) */
      if (((mods[i] + delta) % primes[i]) <= 1) {
        delta += 2;
```

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

```c
static int probable_prime(BIGNUM *rnd, int bits) {
  int i;
  uint16_t mods[NUMPRIMES];
  BN_ULONG delta;
  BN_ULONG maxdelta = BN_MASK2 - primes[NUMPRIMES - 1];
  char is_single_word = bits <= BN_BITS2;

again:
  if (!BN_rand(rnd, bits, BN_RAND_TOP_TWO, BN_RAND_BOTTOM_ODD)) {
    return 0;
  }

  /* we now have a random number 'rnd' to test. */
  for (i = 1; i < NUMPRIMES; i++) {
    BN_ULONG mod = BN_mod_word(rnd, (BN_ULONG)primes[i]);
    if (mod == (BN_ULONG)-1) {
      return 0;
    }
    mods[i] = (uint16_t)mod;
  }
  /* If bits is so small that it fits into a single word then we
   * additionally don't want to exceed that many bits. */
  if (is_single_word) {
    BN_ULONG size_limit;
    if (bits == BN_BITS2) {
      /* Avoid undefined behavior. */
      size_limit = ~((BN_ULONG)0) - get_word(rnd);
    } else {
      size_limit = (((BN_ULONG)1) << bits) - get_word(rnd) - 1;
    }
    if (size_limit < maxdelta) {
      maxdelta = size_limit;
    }
  }
  delta = 0;

loop:
  if (is_single_word) {
    BN_ULONG rnd_word = get_word(rnd);

    /* In the case that the candidate prime is a single word then
     * we check that:
     *   1) It's greater than primes[i] because we shouldn't rejec
     *      3 as being a prime number because it's a multiple of
     *      three.
     *   2) That it's not a multiple of a known prime. We don't
     *      check that rnd-1 is also coprime to all the known
     *      primes because there aren't many small primes where
     *      that's true. */
    for (i = 1; i < NUMPRIMES && primes[i] < rnd_word; i++) {
      if ((mods[i] + delta) % primes[i] == 0) {
        delta += 2;
        if (delta > maxdelta) {
          goto again;
        }
        goto loop;
      }
    }
  } else {
    for (i = 1; i < NUMPRIMES; i++) {
      /* check that rnd is not a prime and also
       * that gcd(rnd-1,primes) == 1 (except for 2) */
      if (((mods[i] + delta) % primes[i]) <= 1) {
        delta += 2;
```

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:

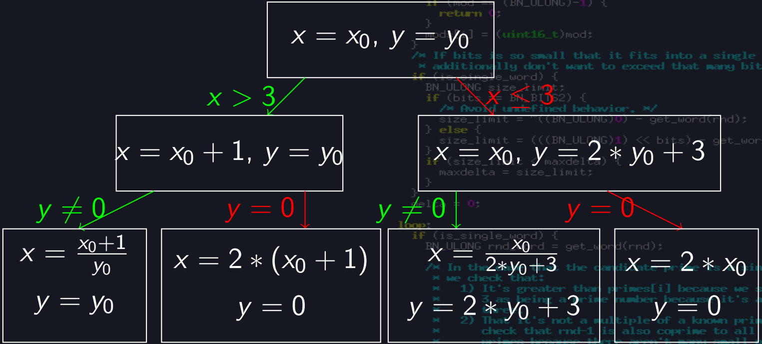
```
eax = z3.SignExt(24, sym_username[local_14h])
eax ^= local_10h
```

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 branches - Programmatically

```c
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;
}
```

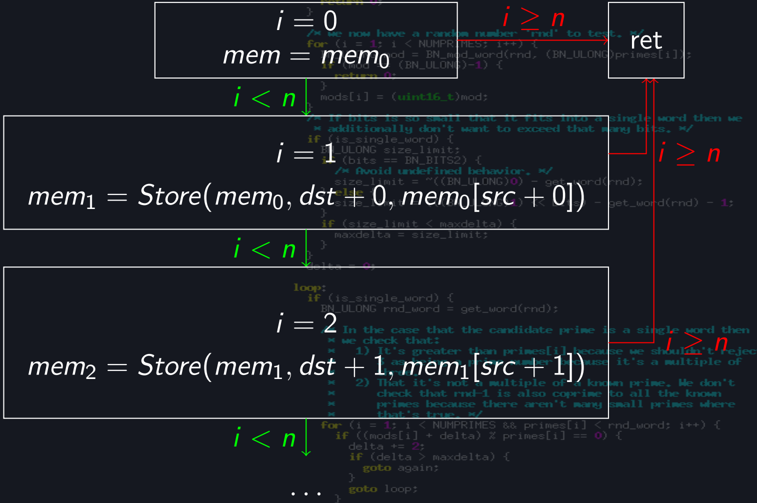
```python
import z3
x0, y0 = z3.Ints(’x0 y0’)
states, newstates = [(x0, y0, z3.Solver())], []
for (x, y, s) in states:
  t = s.__deepcopy__()
  s.add(x > 3); newstates.append((x+1, y, s))
  t.add(z3.Not(x > 3)); newstates.append((x, 2*y+3, t))

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

Angr

- A very useful tool for RE made by shellphish and now maintained by SEFCOM at Arizona State University as well
- Originally made for DARPAs cyber grand challenge.
- A very strong framework for emulation allowing symbolic values.

Angr - The basics

- It is written almost entirely in python and as such currently only has python bindings.
- Installation
  - Has a couple dependencies that may conflict with other packages so it is recommended to use python virtual environments.
  - `$ pip install --user angr`
  - Now you can just import angr in any python interpreter

Angr - Writing a script

- The basic block in angr is a project
- It is how you tell angr what file to load
- Can be used to obtain a bunch of metadata as well about the file
  - `project = angr.Project("./binary")`

Angr - Writing a script

- Accessing everything else will be done through factory
  - `project.factory`
- Next you will have to tell angr where it should start
- Most of the time you want this to be `entry_state()`
  - This will setup everything as it would be at the entry point if you ran the binary
  - This is where you would tell it if you wanted a special stdin or args
    - You can use claripy to make it symbolic

Angr - Simulation manager

- This will hold all your various states as they are created and abandoned
  - `sm = project.factory.simulation_manager(state)`
- Executing
  - To execute you will normally want to use `.run()` or `.explore()`
  - Explore
    - Allows you to guide execution better and limit computation neccesarry
    - You can specify find and avoid conditions via addresses or lambda functions
  - Run
    - Will just run every state until exhaustion
    - Theoretically tells you all possible outcomes of the program
    - Prone to path explosion
  - Many more that are more application specific

Example: Fairgame RE400 with Angr

```c
static int probable_prime(BIGNUM *rnd, int bits) {
  int i;
  uint16_t mods[NUMPRIMES];
  BN_ULONG delta;
  BN_ULONG maxdelta = BN_MASK2 - primes[NUMPRIMES - 1];
  char is_single_word = bits <= BN_BITS2;

again:
  if (!BN_rand(rnd, bits, BN_RAND_TOP_TWO, BN_RAND_BOTTOM_ODD)) {
    return 0;
  }

  /* we now have a random number 'rnd' to test. */
  for (i = 1; i < NUMPRIMES; i++) {
    BN_ULONG mod = BN_mod_word(rnd, (BN_ULONG)primes[i]);
    if (mod == (BN_ULONG)-1) {
      return 0;
    }
    mods[i] = (uint16_t)mod;
  }
  /* If bits is so small that it fits into a single word then we
   * additionally don't want to exceed that many bits. */
  if (is_single_word) {
    BN_ULONG size_limit;
    if (bits == BN_BITS2) {
      /* Avoid undefined behavior. */
      size_limit = ~((BN_ULONG)0) - get_word(rnd);
    } else {
      size_limit = (((BN_ULONG)1) << bits) - get_word(rnd) - 1;
    }
    if (size_limit < maxdelta) {
      maxdelta = size_limit;
    }
  }
  delta = 0;

loop:
  if (is_single_word) {
    BN_ULONG rnd_word = get_word(rnd);

    /* In the case that the candidate prime is a single word then
     * we check that:
     *   1) It's greater than primes[i] because we shouldn't rejec
     *      3 as being a prime number because it's a multiple of
     *      three.
     *   2) That it's not a multiple of a known prime. We don't
     *      check that rnd-1 is also coprime to all the known
     *      primes because there aren't many small primes where
     *      that's true. */
    for (i = 1; i < NUMPRIMES && primes[i] < rnd_word; i++) {
      if ((mods[i] + delta) % primes[i] == 0) {
        delta += 2;
        if (delta > maxdelta) {
          goto again;
        }
        goto loop;
      }
    }
  } else {
    for (i = 1; i < NUMPRIMES; i++) {
      /* check that rnd is not a prime and also
       * that gcd(rnd-1,primes) == 1 (except for 2) */
      if (((mods[i] + delta) % primes[i]) <= 1) {
        delta += 2;
```

Fairgame RE400 script and output

Script:

```
import angr
import claripy

p = angr.Project("./re400")

for j in range(0xFF):
    # make a symbolic list of j bytes
    flag_chars = [claripy.BVS("serial_%d" % i, 8) for i in range(j)]
    # combine them all
    flag = claripy.Concat(*flag_chars)

    # tell angr to start at entry
    init = p.factory.entry_state(stdin=flag)

    sm = p.factory.simulation_manager(init)

    # find a state with "unlocked" in stdout
    sm.explore(find=lambda s: b"unlocked" in s.posix.dumps(1))

    # try next length if not found
    if len(sm.found) == 0:
        continue
    print('Good length: %d' % (j,))

    print(sm)
    for i in sm.found:
        print(i.posix.dumps(1))
        print(i.posix.dumps(0))
    break
```

Redacted output:

```
Good length: 25
<SimulationManager with 1 active, 1 found>
b'[+] Welcome to the REvision 400 lock firmware.\n
[!] Please enter the serial:\n
[+] REvision 400 lock firmware unlocked.\n'
b'flag{__________________}\xff'
python script.py  154.01s user 2.08s system
102% cpu 2:31.85 total
```

Resources

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