Symbolic Execution with Angr
RPISEC

Avi Weinstock (`aweinstock`), Luke Biery (`tiecoon`)

December 6, 2019

```
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;
```

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

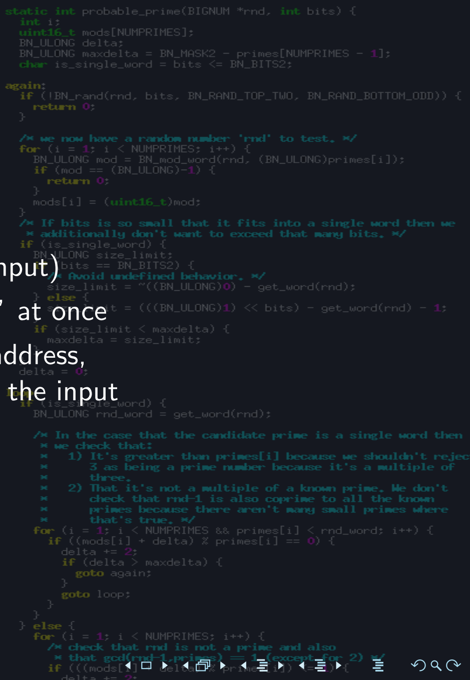
```
static int probable_prime(BIGNUM *rnd, int bits) {
  int i;
  uint16_t mods[NUMPRIMES];
  BN_ULONG delta;
  BN_ULONG maxdelta = BN_MASK2 - primes[NUMPRIMES - 1];
  char is_single_word = bits <= BN_BITS2;

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;
```

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?

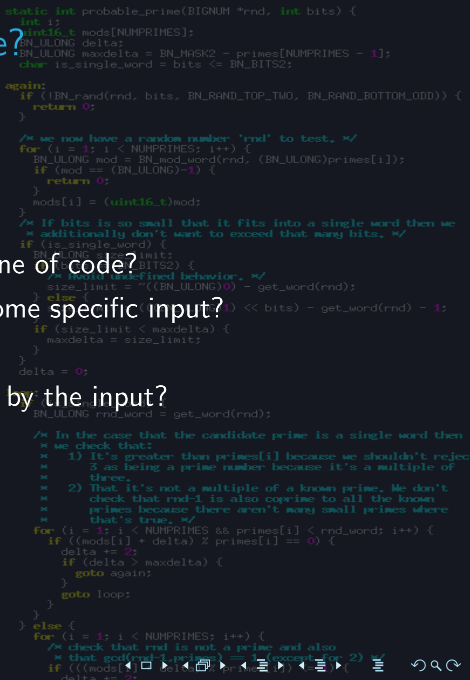
- Executes a program with symbolic data (usually input)
- Essentially runs a program on ”all possible inputs” at once
- Instead of having concrete data in each variable/address, variables/addresses store trees of what to do with the input



```
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 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?



```
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;
```

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

TODO: more comparisons/columns? emphasize that "all inputs" means that symexec can find constant-time comparisons against a giant constant, unlike coverage-guided?

How symbolic execution works in general

```
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

Symbolically executing branch-free code

- Translate arithmetic, indexing, etc into SMT constraints

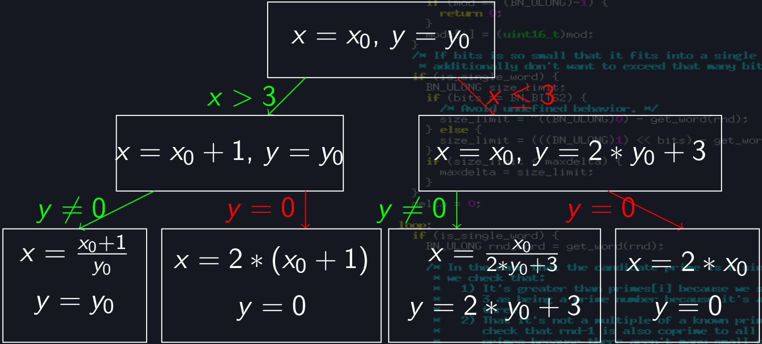
`mov eax, ebx`	`symstate[’eax’] = symstate[’ebx’]`
`add ecx, edx`	`symstate[’ecx’] += symstate[’edx’]`

`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)`

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;
}
```



$x = x_0, y = y_0$

$x > 3$

$x \leq 3$

$x = x_0 + 1, y = y_0$

$x = x_0, y = 2 * y_0 + 3$

$y \neq 0$

$y = 0$

$y \neq 0$

$y = 0$

$x = \frac{x_0+1}{y_0}$
$y = y_0$

$x = 2 * (x_0 + 1)$
$y = 0$

$x = \frac{x_0}{2*y_0+3}$
$y = 2 * y_0 + 3$

$x = 2 * x_0$
$y = 0$

Symbolically executing branches - 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)
```

TODO: Avi

- symbolic loops
- symbolizing tainted memory (e.g. `sym_memory[(EBP+username+i)]` vs `sym_username[i]`)
- state explosion when looping on symbolic data

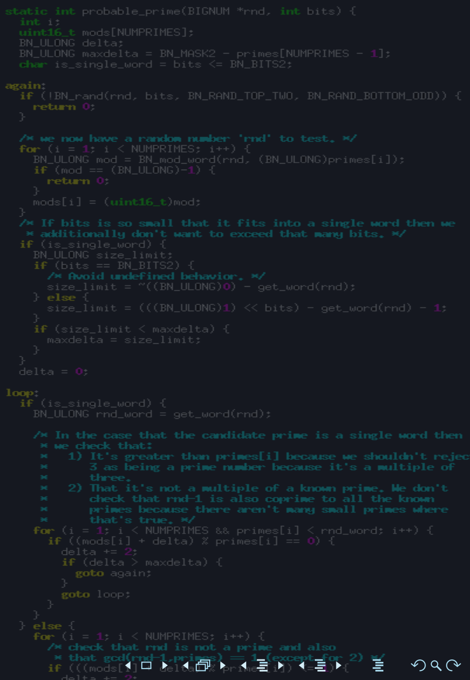
```
static int probable_prime(BIGNUM *rnd, int bits) {
  int i;
  uint16_t mods[NUMPRIMES];
  BN_ULONG delta;
  BN_ULONG maxdelta = BN_MASK2 - primes[NUMPRIMES - 1];
  char is_single_word = bits <= BN_BITS2;

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;
```

How to use Angr for symbolic execution

```
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;
```

TODO: Luke

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



```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;
```

Example: MBE lab1A with Angr

```
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;
```

Resources

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

```
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;
```