

Symbolic Execution with Angr

RPISEC

Avi Weinstock (aweinstock), Luke Biery (lbiery)

December 6, 2019

```
static int mod_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)) < 0)
        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;
        }
        (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) {
            if (BN_mod_word(rnd, size_limit) < 0)
                return 0;
            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 reject
         *    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]) == 0) {
                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) {
        /* avoid undefined behavior. */
        /* 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 reject
         *    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]) == 0) {
                delta += 2;
                if (delta > maxdelta) {
                    goto again;
                }
            }
        }
    }
}
```

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

```
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)0 - 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 reject
         *    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] == 0) {
                delta += 2;
                if (delta > maxdelta) {
                    goto again;
                }
            }
        }
    }
}
```

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;
        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;
    }
    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 reject
         *    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]) == 0) {
                delta += 2;
                if (delta > maxdelta) {
                    goto again;
                }
            }
        }
    }
}
```

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 inline prime(BIGNUM *rnd, int bits) {
    int i;
    uint16_t mods[NUMPRIMES];
    BN_ULONG delta;
    BN_ULONG maxdelta = BN_MASK2 - primes[NUMPRIMES - 1];
    clear_is_single_word = bits <= BN_BITS2;

again:
    if ((BN_rand(rnd, bits, BN_RAND_TOP_TWO, BN_RAND_BOTTOM_ODD)) < 0)
        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;
    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 reject
     *    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]) == 0) ||
            ((i > 1) && (rnd_word % primes[i]) == 0)) {
            delta += 2;
            if (delta > maxdelta) {
                goto again;
            }
            goto loop;
        }
    }
}
```

Symbolic Execution vs Fuzzing

Symbolic Execution

- + Explores all inputs
- + Very detailed output
- Uses more memory/time

Fuzzing

- Only explores random inputs
- Only learn crash vs non-crash
- + 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 time¹

¹Unless the compare is digit-by-digit

```
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 (bits == BN_BITS2) {
        /* Avoid undefined behavior. */
        BN_ULONG mod = BN_mod_word(rnd, -get_word(rnd));
    } else {
        size_limit = (((BN_ULONG)1) << bits) - get_word(rnd) - 1;
        if (size_limit > maxdelta) {
            maxdelta = size_limit;
        }
    }
    delta = 1;
loop:
    if (is_single_word) {
        BN_ULONG mod = 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 reject
         *    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`

```
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 (bits < BN_BITS2) {
        BN_ULONG limit;
        if (bits == BN_BITS2) {
            /* Avoid undefined behavior. */
            size_limit = ((BN_ULONG)0) * get_word(rnd);
        } else {
            size_limit = ((BN_ULONG)0) * bits - get_word(rnd) - 1;
        }
        if (size_limit < maxdelta) {
            maxdelta = size_limit;
        }
    }

loop:
    /* 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 reject
     *    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]) == 0) {
            delta += 2;
            if (delta > maxdelta) {
                goto again;
            }
            goto loop;
        }
    }
}
```

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

```
static prime(BIGNUM *rnd, int bits) {
    int i;
    BN_ULONG mod;
    BN_ULONG mask2 = BN_MASK2 - primes[NUMPRIMES - 1];
    int 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;
        }
        /* If bits is so small that it fits into a single word then we
         * can only don't want to exceed that many bits. */
        BN_ULONG size_limit =
            is_single_word ? (BN_ULONG)-1 :
            /* odd-numbered behavior */
            ((BN_ULONG)1) << bits;
        if (size_limit < maxdelta) {
            maxdelta = size_limit;
        }
    }

loop:
    if (BN_is_prime_ex(rnd, bits, 0, &is_prime)) {
        /* is the case that the candidate prime is a single word then
         * 1) it is a prime because we shouldn't reject
         * 2) as being a prime number because it's a multiple of
         * 3) 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 any small primes where
         * that's true. */
        for (i = 1; i < NUMPRIMES; i++) {
            if ((mod[i] + delta) % primes[i] == 0) {
                delta += 2;
                if (delta > maxdelta) {
                    goto again;
                }
                goto loop;
            }
        }
    } else {
        for (i = 1; i < NUMPRIMES; i++) {
            /* check that rnd-1 is not a prime and also
             * that god(rnd-1) == 1 (except for 2) */
            if ((mod[i] + delta) % primes[i] == 0) {
                delta += 2;
            }
        }
    }
}
```

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)
```

```
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  
     * should check that it does not exceed that many bits. */  
    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) {  
        delta = size_limit;  
    } else {  
        delta = 0;  
    }  
    loop:  
    if (is_single_word) {  
        BN_ULONG rnd_word = get_word(rnd);  
        /* If bits is small enough to fit into a single word then  
         * we can check that it does not exceed that many bits.  
         * If it's greater than primes[i] because we shouldn't reject  
         * 3 as being a prime number because it's a multiple of  
         * 2. If it's 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; i++) {  
            if ((mod[i] + delta) % primes[i] == 0) {  
                if (delta > maxdelta) {  
                    goto again;  
                }  
                goto loop;  
            }  
        }  
    } else {  
        for (i = 1; i < NUMPRIMES; i++) {  
            /* check that rnd-1 is not a prime and also  
             * that gcd(rnd-1, primes) = 1 (except for 2) */  
            if (((mod[i] + delta) % primes[i]) != 1) {  
                if (delta > maxdelta) {  
                    goto again;  
                }  
                goto loop;  
            }  
        }  
    }  
    return 1;  
}
```

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]
```

[illegible]

Symbolically executing branches - 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;
}
```

$x = x_0, y = y_0$

$x > 3$

$x = x_0 + 1, y = y_0$

$y \neq 0$

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

$y = 0$

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

$y \neq 0$

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

$y = 0$

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

```
static_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)) < 0)
        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 < primes[i]) {
            return 0;
        }
        /* 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 = ((BN_ULONG)1) << bits;
            if (size_limit < BN_MASK2) {
                size_limit = ((BN_ULONG)1) << bits;
            }
            if (size_limit < BN_MASK2) {
                size_limit = BN_MASK2;
            }
            if (mod > size_limit) {
                goto again;
            }
        }
        /* In the case of the candidate prime being a single word then
         * we check that:
         * 1) It's greater than primes[i] because we shouldn't reject
         *    3. It's not a prime number because it's a multiple of
         *    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 that's a much more complicated problem
         *    that's true. */
        for (i = 1; i < NUMPRIMES; i++) {
            if ((mod[i] + delta) % primes[i] == 0) {
                delta += 2;
                if (delta > maxdelta) {
                    goto loop;
                }
            }
        }
    }
    /* check that rnd is not a prime and also
     * that gcd(rnd-1, primes) = 1 (except for 2) */
    if (((mod[1] + delta) % primes[1]) < 2) {
        delta += 2;
        if (delta > maxdelta) {
            goto loop;
        }
    }
}
```

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)
```

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

again:
    if ((BN_rand(rnd, bits, BN_RAND_TOP_TWO, BN_RAND_BOTTOM_ODD)) < 0)
        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]);
        return 0;
    }
    mods[i] = (uint16_t)mod;

    /* If bits is so small that it fits into a single word then we
     * don't want to exceed that many bits. */
    if (is_single_word) {
        BN_ULONG size_limit = BN_ULONG_MAX;
        /* Need to avoid overflow. */
        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;

    if (BN_is_odd_word(rnd)) {
        BN_ULONG mod = 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 reject
         * 3 as being a prime number because it's a multiple of
         * three.
         * 2) that it's not a multiple of a small prime. We don't
         * check this for all the known primes because there aren't many small primes where
         * that's true. */
        for (i = 1; i < NUMPRIMES && primes[i] < mod; i++) {
            if ((mods[i] + delta) % primes[i] == 0) {
                delta = 2;
                if (delta > maxdelta) {
                    goto loop;
                }
            }
        }
    } else {
        for (i = 1; i < NUMPRIMES; i++) {
            /* check that rnd is not a prime and also
             * that get_word(rnd) = 1 (except for 2) */
            if (((mod + delta) % primes[i]) == 0) {
                delta = 2;
            }
        }
    }
}
```

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];
    }
}
```

$i = 0$
 $mem = mem_0$

$i < n$

$i = 1$
 $mem_1 = Store(mem_0, dst + 0, mem_0[src + 0])$

$i < n$

$i = 2$
 $mem_2 = Store(mem_1, dst + 1, mem_1[src + 1])$

$i < n$

```
static_prime(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)) <
        BN_BITS2)
        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 == 0)
            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)0) - 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 reject
         *    a prime because it's a multiple of
         * 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; 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]) == 0) {
                delta += 2;
                if (delta > maxdelta) {
                    goto again;
                }
            }
        }
        goto loop;
    }
}
```

$i \geq n$

ret

$i \geq n$

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

```
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 =
            /* limit based on BN_BITS2 */
            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;
    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 reject
         *    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]) == 0) {
                delta += 2;
            }
        }
    }
}
```

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

```
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
     * can avoid the modulo operation for the small primes. */
    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 can check that:
         * 1) It's greater than primes[i] because we shouldn't reject
         *    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]) == 0) {
                delta += 2;
            }
        }
    }
}
```

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")`

```
static int 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 reject
         *    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]) != 0) {
                delta += 2;
            }
        }
    }
}
```

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

```
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;
    }
    /* If (is_single_word) {
       BN_ULONG mod = BN_mod_word(rnd, (BN_ULONG)primes[i]);
       if (mod == (BN_ULONG)-1) {
           return 0;
       }
       mods[i] = (uint16_t)mod;
    }

    /* 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 reject
       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]) == 0) {
            delta += 2;
            if (delta > maxdelta) {
                goto again;
            }
            goto loop;
        }
    }
}
```

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 necessary
 - ▶ 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

```
static int 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)) < 0)
        return 0;

    /* we now have a random number 'rnd' to test. */
    for (i = 1; i < NUMPRIMES; i++) {
        /* we check that rnd is not a multiple of primes[i] */
        if ((BN_ULONG)primes[i] < (BN_ULONG)rnd)
            return 0;
    }

    /* 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) {
            /* special case behavior. */
            size_limit = ((BN_ULONG)0) - get_word(rnd);
        } else {
            size_limit = (((BN_ULONG)1) << bits) - get_word(rnd) - 1;
        }
        if (size_limit < maxdelta) {
            /* size limit is too small, so we need to
             * increase the number of bits. */
            goto again;
        }
    }

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 reject
         *    2) being a prime number because it's a multiple of
         *    3) 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]) == 0) ||
                ((BN_ULONG)primes[i] < (BN_ULONG)rnd)) {
                delta += 2;
                if (delta > maxdelta) {
                    goto again;
                }
            }
        }
    }
}
```

Angr - States

- ▶ States will allow you to introspect, touch or check memory values;
- ▶ For now we will only be touching state.posix
 - ▶ This allows access to the various posix style file descriptors on supported systems
 - ▶ A lot you can do here by using hooks and moving states between stashes
- ▶ Stashes
 - ▶ Active
 - ▶ Deadended
 - ▶ Pruned
 - ▶ Unconstrained
 - ▶ Unsat

```
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;
        }
    }
    /* 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) {
            size_limit = ((BN_ULONG)0) - get_word(rnd);
        } else {
            size_limit = ((BN_ULONG)0) - 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 reject
         *    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]) == 0) {
                delta += 2;
                if (delta > maxdelta) {
                    goto again;
                }
            }
        }
    }
}
```


Angr - Symbolic Values

- ▶ Angr uses a library called claripy which is their wrapper for constraint solvers
- ▶ Used in place of direct Z3 calls so it is portable between different backends
- ▶ Shouldn't need to touch anything besides
 - ▶ BV - bitvector
 - ▶ FP - floating point
 - ▶ Bool - boolean
- ▶ Then just like in Z3 you can perform math with these, however to combine them one after another you need to use Concat()
- ▶ NOTE: The default python $>$, $<$, \geq , and \leq are unsigned in Claripy. This is different than their behavior in Z3, because it seems more natural in binary analysis.

```
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];
    clear_is_single_word = bits <= BN_BITS2;

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

/* we now have a random number 'rnd' to test. */
for (i = 1; i < NUMPRIMES; i++) {
    /* mod = rnd % primes[i] */
    return 0;
}

/* 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;

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 reject
     *    it if it's a multiple of a known prime.
     * 2) That it's not a multiple of a known prime. We don't
     *    check that rnd+1 is a prime to all the known
     *    primes because the small primes where
     *    that's true are very rare.
     */
    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]) == 0) {
            delta += 2;
            if (delta > maxdelta) {
                goto again;
            }
            goto loop;
        }
    }
}
```

Example: Fairgame RE400 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 reject
         *    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]) == 0) {
                delta += 2;
                if (delta > maxdelta) {
                    goto again;
                }
            }
        }
    }
}
```

Fairgame RE400 Ghidra decompilation

```
int main () {
    puts("[+] Welcome to the REvision 400 lock firmware.");
    uStack288 = 0x1008b1;
    puts("[!] Please enter the serial:");
    uStack288 = 0x1008cc;
    fgets(&local_118,0xfe,stdin);
    uStack288 = 0x1008db;
    sVar2 = strlen(&local_118);
    *(&uStack288 + sVar2 + 7) = 0;
    uStack288 = 0x1008f6;
    mangle(&local_118);
    uStack288 = 0x100911;
    iVar1 = memcmp(&local_118,expect,0xfe);
    if (iVar1 == 0) {
        uStack288 = 0x100921;
        puts("[+] REvision 400 lock firmware unlocked.");
    }
    else {
        uStack288 = 0x10092f;
        puts("[-] Invalid serial.");
    }
    if (local_10 != *(in_FS_OFFSET + 0x28)) {
        /* WARNING: Subroutine does not return */
        uStack288 = 0x100948;
        __stack_chk_fail();
    }
    return 0;
}
```

```
void mangle(char *input) {
    int i;
    i = 0;
    while (i < 0xfe) {
        swap(input + i,
            input + *(map + i * 4),
            input + *(map + i * 4),
            i);
        i = i + 1;
    }
    return;
}
```

```
oid swap(char *param_1,char *param_2){
    char cVar1;

    cVar1 = *param_1;
    *param_1 = *param_2;
    *param_2 = cVar1;
    return;
}
```

```
static 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)) < 0)
        return 0;

    /* we now have a random number 'rnd' to test, */
    for (i = 1; i < NUMPRIMES; i++) {
        if (BN_mod_word(rnd, (BN_ULONG)primes[i]) < 0)
            return 0;
        mods[i] = (uint16_t)mod;
    }
    while (i < 0xfe) {
        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 limit;
            BN_ULONG 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 (BN_MOD_WORD(rnd) < 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 reject
        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 god(rnd-1 primes) = 1 (except for 2) */
            if (((mods[i] + delta) % primes[i]) < 0) {
                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
```

```
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)) < 0)
        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) {
            /* defined behavior. */
            size_limit = (((BN_ULONG)0) - get_word(rnd));
        } else {
            size_limit = (((BN_ULONG)1) << bits) - get_word(rnd) - 1;
        }
        if (BN_cmp(rnd, size_limit) > 0)
            return 0;
    }
    /* 1) It's greater than primes[i] because we shouldn't reject
     *    2 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; 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 god(rnd-1, primes) = 1 (except for 2) */
        if (((mods[i] + delta) % primes[i]) == 0) {
            delta += 2;
            if (delta > maxdelta) {
                goto again;
            }
            goto loop;
        }
    }
}
```


Resources

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

```
static int 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 reject
         *    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]) != 0) {
                delta += 2;
            }
        }
    }
}
```