

Introduction to Reversing with Z3

RPISec

Avi Weinstock (aweinstock)

December 3, 2019

```
static inline 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;
                }
            }
        }
    }
}
```

Overview

- ▶ What are SAT/SMT/Z3?
- ▶ How do the z3 python bindings work?
- ▶ Solving some small, isolated examples with Z3
- ▶ Solving a Cyberseed RE challenge with Z3
- ▶ Solving MBE lab1A with Z3

```
static inline 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) {
            /* 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 are SAT/SMT/Z3?

```
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;
                }
            }
        }
    }
}
```

What is Z3?

- ▶ SAT & SMT solver developed and maintained by Microsoft Research
- ▶ Libre and Open Source (MIT Licensed)
- ▶ C++, with python bindings (`pip install z3-solver`)
- ▶ Based on the CDCL algorithm

```
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;
        /* 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;
            }
        }
    }
}
```

What is SAT?

- ▶ SAT is the boolean satisfiability problem
- ▶ e.g. "Does the formula $(x \vee \neg y \vee z) \wedge (\neg x \vee y)$ have a satisfying assignment?"
 - ▶ (\neg, \wedge, \vee) mean (NOT, AND, OR)
- ▶ Brute forceable in $O(2^n)$ by trying all combinations of $\{0, 1\}$ for all variables
- ▶ NP-Complete
 - ▶ **Con:** Impossible¹ to solve quickly²
 - ▶ **Pro:** Many problems can be expressed as SAT instances, so heuristics for SAT can help solve many problems

¹Unless P=NP

²In polynomial time

```
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 small that it fits into a single word then we
       can do a quick check. */
    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) - get_word(rnd) - 1;
        }
        if (size_limit < maxdelta) {
            maxdelta = size_limit;
        }
    }
    delta = 0;

loop:
    if (is_single_word) {
        BN_ULONG rand_word = get_word(rnd);
        /* In this 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 rand-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] < rand_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 SMT?

- ▶ SMT is Satisfiability Modulo Theories
- ▶ "Does $(f(x, y) \vee z) \wedge (\neg g(x) = f(x, x))$ have a satisfying assignment?" (QF-EUF)
- ▶ "Does $(2 * x + y \leq z) \wedge (x + 3 * y \geq z)$ have a satisfying assignment?" (QF-LIA)
- ▶ Allows more compact translation of problems, e.g.
 - ▶ $x = 1 \vee x = 2 \vee x = 3 \vee \dots \vee x = 99 \vee x = 100$ (SAT)
 - ▶ $1 \leq x \wedge x \leq 100$ (SMT)
- ▶ Also NP-Complete

```
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) {
        /* All one size limit. */
        /* Valid under odd behavior. */
        size_limit = "((BN_ULONG)0) - get_word(rnd);
    } else {
        /* size limit is bits <= BN_BITS2 */
        size_limit = (BN_ULONG)-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) ||
                (primes[i] == 2 && (rnd_word & 1) == 0)) {
                delta += 2;
                if (delta > maxdelta) {
                    goto again;
                }
            }
        }
    }
    return 1;
}
```

Why are SAT/SMT useful if they're hard to solve quickly?

- ▶ Not all problems are as hard as the hardest ones
 - ▶ 2-SAT (each clause having at most 2 variables) is polytime solvable
 - ▶ Monotone circuits (only ANDs and ORs, no NOTs) are polytime solvable
- ▶ It's often possible to prune the search space
 - ▶ e.g. $x \vee \varphi(a, b, c, \dots)$ is solvable regardless of φ because $x = 1$ cancels out that subterm
- ▶ Algorithms like DPLL and CDCL make use of partial structure to solve some instances faster than others
- ▶ SMT can make use of the rules for the extra types of symbols to prune the search space at a higher level

```
static_prime(BIGNUM *rnd, int bits) {
    int i;
    uint64_t mod;
    BN_ULONG 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)) {
        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] = (uint32_t)mod;
        /* if it fits into a single word then we
         * additionally don't want to exceed that many bits. */
        if (bits <= BN_BITS2) {
            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;
            }
            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
     *    5) as being a prime number because it's a multiple of
     *    7) as being a prime number because it's a multiple of
     *    11) as being a prime number because it's a multiple of
     *    13) as being a prime number because it's a multiple of
     *    17) as being a prime number because it's a multiple of
     *    19) as being a prime number because it's a multiple of
     *    23) as being a prime number because it's a multiple of
     *    29) as being a prime number because it's a multiple of
     *    31) as being a prime number because it's a multiple of
     *    37) as being a prime number because it's a multiple of
     *    41) as being a prime number because it's a multiple of
     *    43) as being a prime number because it's a multiple of
     *    47) as being a prime number because it's a multiple of
     *    53) as being a prime number because it's a multiple of
     *    59) as being a prime number because it's a multiple of
     *    61) as being a prime number because it's a multiple of
     *    67) as being a prime number because it's a multiple of
     *    71) as being a prime number because it's a multiple of
     *    73) as being a prime number because it's a multiple of
     *    79) as being a prime number because it's a multiple of
     *    83) as being a prime number because it's a multiple of
     *    89) as being a prime number because it's a multiple of
     *    97) as being a prime number because it's a multiple of
     *    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;
        }
    }

    /* check that rnd is not a prime and also
     * that gcd(rnd-1, primes) = 1 (except for 2) */
    for (i = 1; i < NUMPRIMES; i++) {
        if (((mods[i] + delta) % primes[i]) != 0) {
            delta += 2;
            if (delta > maxdelta) {
                goto again;
            }
            goto loop;
        }
    }
}
```


Installing and importing z3

► pip install z3-solver

► import z3

► Some people do `from z3 import *`, but the remainder of this talk will use the qualified import version

```
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;
        }
    }

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;
            }
        }
    }
}
```

Variables and Values

- ▶ import z3

```
x = z3.Int('x')
```

```
y = z3.BitVec('y', 32)
```

```
z = z3.BitVec('z', 16)
```

```
w = z3.Real('w')
```

- ▶ Variables are symbolic

- ▶ Operator overloading allows creation of constraints

- ▶

```
a = z3.IntVal(42)
```

```
b = z3.BitVecVal(0xdeadbeef, 32)
```

```
c = z3.BitVecVal(0xbeef, 16)
```

```
d = z3.RealVal(__import__('math').pi)
```

- ▶ Values are concrete

- ▶ Dynamic type checks on them are stricter than Python's defaults

```
static inline 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;  
                }  
            }  
        }  
    } 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;  
                }  
            }  
        }  
    }  
    return 1;  
}
```

Solver objects

- ```

import z3
solver = z3.Solver()

```

```

 } mod[i] = (uint32_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. */
 limit = "((BN_ULONG)0) - get_word(rnd);
 if (
 size_limit = "((BN_ULONG)1) << bits) - get_word(rnd) - 1;
 }
 if (size_limit < maxdelta) {
 maxdelta = size_limit;
 }
 }
 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) That it's not a multiple of a known prime. We don't reject
 * 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
 * three.
 * 3) Being a prime number because it's a multiple of

```

- ▶ The Solver class collects constraints/equations to solve
  - ▶ solver.add adds constraints to the current collection
  - ▶ solver.check() checks if the current constraints are solvable, returning one of {z3.sat, z3.unsat, z3.unknown}
  - ▶ After solver.check() returns z3.sat, z3.model() will give you the values that make the constraints true

```

static int prime_prob(BIGNUM *p, 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;
 }
 }
 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 not a multiple of 2 (we don't reject
 * 2) 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]) < 2) {
 delta += 2;
 }
 }
 }
}

```



## Solving some small examples with Z3

```
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;
 }
 }
 }
}
```

# Using Z3 on small examples: 1/3

►  $(x \vee \neg y \vee z) \wedge (\neg x \vee y)$

► 

```
import z3
solver = z3.Solver()
x, y, z = z3.Bools('x y z')
solver.add(z3.And(z3.Or(x, z3.Not(y), z), z3.Or(z3.Not(x), y)))
if solver.check().r == 1:
 print(solver.model())
```

► 

```
[z = False, y = False, x = False]
```

```
static inline 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;
 }
 }
 }
 }
}
```

## Using Z3 on small examples: 2/3

►  $(2 * x + y \leq z) \wedge (x + 3 * y \geq z) \wedge (z > 1)$

► 

```
import z3
solver = z3.Solver()
x, y, z = z3.Ints('x y z')
solver.add(2*x+y <= z)
solver.add(x+3*y >= z)
solver.add(z > 1)
if solver.check().r == 1:
 print(solver.model())
```

►  $[z = 5, y = 1, x = 2]$

```
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)) &
 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;
 }
 }
 }
 }
}
```



# Using Z3 on small examples: 3/3

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

```
import z3
x, y, z = z3.Reals('x y z')
solver = z3.Solver()
solver.add(1*x + 2*y + 3*z == 1)
solver.add(4*x + 5*y + 6*z == 2)
solver.add(7*x + 8*y + 9*z == 3)
status = solver.check()
assert solver.check().r == 1
print(solver.model())
solver.add(x != 0, y != 0, z != 0)
assert solver.check().r == 1
print(solver.model())
```

```
[z = 1/3, y = 0, x = 0]
[z = -1, y = 8/3, x = -4/3]
```

```
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) {
 /* 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 += primes[i];
 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 += primes[i];
 }
 }
 }
}
```



## Solving Cyberseed 2019's "Hasher" challenge

```
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]) != 1) {
 delta += 2;
 }
 }
 }
}
```

# Cyberseed Hasher - Problem description

- ▶ This year's Cyberseed, one of the challenges was a **Java crackme**
- ▶ We were only given a class file, no source
- ▶ DDG'ing "java decompiler online" found <https://devtoolzone.com/decompiler/java>
- ▶ Now we had source :)

```
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 == (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) {
 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;
 }
 }
 }
 }
}
```

# Cyberseed Hasher - Decompiled source

```
public class Hasher {
 private static boolean hash(final String s) {
 int n = 7;
 final int n2 = 593779930;
 for (int i = 0; i < s.length(); ++i) {
 n = n * 31 + s.charAt(i);
 }
 return n == n2;
 }

 public static void main(final String[] array) {
 if (array.length != 1) {
 System.out.println("Usage: java Hasher <password>");
 System.exit(1);
 }
 if (hash(array[0])) {
 System.out.println("Correct");
 }
 else {
 System.out.println("Incorrect");
 }
 }
}
```

```
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];
 if (is_single_word & bits <= BN_BITS2)
 return 1;

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;
 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;
 }
 }
 }
 }
 return 1;
}
```

# Cyberseed Hasher - Z3 script: 1/2

```
import z3
wanted_length = 6

names = ['x{i}'.format(i=i) for i in range(wanted_length)]
vars = [z3.Int(n) for n in names]

expr = 7
for i in range(wanted_length):
 expr *= 31
 expr += vars[i]

prob = z3.Solver()
prob.add((expr % (2**32)) == 593779930)
```

- Concrete input length: z3.Array exists, but is more expensive
- Only use z3.Array if you need symbolic indexing
- z3.Int(n) is faster than z3.BitVec(n, 32) here, needed trial and error for this

```
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 == (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
 * gcd(rnd-1, primes[i]) > 1. */
 for (i = 1; i < NUMPRIMES; i++) {
 if ((mods[i] + delta) % primes[i] == 0) {
 delta += 2;
 if (delta > maxdelta) {
 goto again;
 }
 }
 }
 } else {
 for (i = 1; i < NUMPRIMES; i++) {
 /* check that rnd is not a prime and also
 * that gcd(rnd-1, primes[i]) = 1 (except for 2) */
 if (((mods[i] + delta) % primes[i]) == 0) ||
 ((i != 1) && ((rnd_word - 1) % primes[i]) == 0)) {
 delta += 2;
 if (delta > maxdelta) {
 goto again;
 }
 }
 }
 }
}
```

# Cyberseed Hasher - Z3 script: 2/2

```
for v in vars:
 prob.add(ord('0') <= v)
 prob.add(v <= ord('z'))

res = prob.check()
print('z3 check: %r' % (res,))
if res.r == 1:
 soln = prob.model()
 print('z3 solution: %r' % (soln,))
 print(repr(''.join(chr(soln[v].as_long()) for v in vars)))
```

z3 check: sat

z3 solution: [x3 = 74, x2 = 98, x1 = 114, x5 = 48, x4 = 51, x0 = 100]

'drbJ30'

- Requiring that `ord('0') <= v <= ord('z')` mostly-guarantees alphanumeric input (there's a few exceptions in the middle)

```
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++) {
 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 can check it more efficiently. */
 /* 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; i++) {
 if ((mods[i] - delta) % primes[i] == 0) {
 delta += primes[i];
 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 != 2) && (rnd_word & primes[i]) == 0)) {
 delta += primes[i];
 goto again;
 }
 }
 }
}
```





# Solving Modern Binary Exploitation's lab1A

```
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;
 }
 }
 }
 }
}
```

# MBE Lab1A - Acquiring it to follow along

- ▶ `git clone https://github.com/RPISEC/MBE`
- ▶ `cd MBE/src/lab01`
- ▶ Load the lab1A binary into your favorite disassembler

```
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;
 }
 }

 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;
 }
 }
 }
}
```

# MBE Lab1A - Just Running It

```
avi@aweinstock-debian-ii:~/Documents/cloned-repos/MBE/src/lab01$./lab1A
```

```
+-----+
|----- RPISEC -----|
+ SECURE LOGIN SYS v. 3.0 +
~- Enter your Username: ~-

```

username

```
+-----+
!! NEW ACCOUNT DETECTED !!
~- Input your serial: ~-

```

password

```
avi@aweinstock-debian-ii:~/Documents/cloned-repos/MBE/src/lab01$ echo $?
```

1

```
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)) {
```

```
 } 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 (((mod[i] = delta % primes[i]) <= 1) &&
 delta % 2) {
```

# MBE Lab1A - Username Entry

```
0x08048b69 c70424738d04, mov dword [esp], str..-----, ; [0x8048d73:4]=0x2d2d2d2e ; ",-----",
0x08048b70 e89bfcffff call sym.imp.puts ; int puts(const char *s)
0x08048b75 c70424918d04, mov dword [esp], str,RPISEC ; [0x8048d91:4]=0x2d2d2d7c ; "|----- RPISEC -----|"
0x08048b7c e88ffcffff call sym.imp.puts ; int puts(const char *s)
0x08048b81 c70424af8d04, mov dword [esp], str.SECURE_LOGIN_SYS_v._3.0 ; [0x8048daf:4]=0x53202b7c ; "|+ SECURE LOGIN SYS v. 3.0 |+|"
0x08048b88 e883fcffff call sym.imp.puts ; int puts(const char *s)
0x08048b8d c70424cd8d04, mov dword [esp], str. ; [0x8048dcd:4]=0x2d2d2d7c ; "|-----|"
0x08048b94 e877fcffff call sym.imp.puts ; int puts(const char *s)
0x08048b99 c70424eb8d04, mov dword [esp], str.Enter_your_Username ; [0x8048deb:4]=0x202d7e7c ; "|~- Enter your Username: ~-|"
0x08048ba0 e86bfcffff call sym.imp.puts ; int puts(const char *s)
0x08048ba5 c70424098e04, mov dword [esp], str. ; [0x8048e09:4]=0x2d2d2d27 ; "'-----'"
0x08048bac e85ffcffff call sym.imp.puts ; int puts(const char *s)
0x08048bb1 a160b00408 mov eax, dword [obj.stdin] ; [0x804b060:4]=0
0x08048bb6 89442408 mov dword [local_8h], eax
0x08048bba c74424042000, mov dword [local_4h], 0x20 ; [0x20:4]=-1 ; 32
0x08048bc2 8d44241c lea eax, [local_1ch] ; 0x1c ; 28
0x08048bc6 890424 mov dword [esp], eax
0x08048bc9 e802fcffff call sym.imp.fgets ; char *fgets(char *s, int size, FILE *stream)
```

```
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;
 }

 /* 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 ((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 is not a prime and also
 * that gcd(rnd-1, primes) = 1 (except for 2) */
 if (((mod[i] + delta) % primes[i]) != 0) {
 delta += 2;
 }
 }
 }
}
```

# MBE Lab1A - Serial Entry

```
0x08048bce c70424738d04, mov dword [esp], str..-----+ ; [0x8048d73:4]=0x2d2d2d2e ; ".-----".
0x08048bd5 e836fcffff call sym.imp.puts ; int puts(const char *s)
0x08048bda c70424278e04, mov dword [esp], str.NEW_ACCOUNT_DETECTED ; [0x8048e27:4]=0x2121207c ; "I !! NEW ACCOUNT DETECTED !!!"
0x08048be1 e82afcffff call sym.imp.puts ; int puts(const char *s)
0x08048be6 c70424cd8d04, mov dword [esp], str. ; [0x8048dcd:4]=0x2d2d2d7c ; "|-----|"
0x08048bed e81efcffff call sym.imp.puts ; int puts(const char *s)
0x08048bf2 c70424458e04, mov dword [esp], str.Input_your_serial; ; [0x8048e45:4]=0x202d7e7c ; "|~- Input your serial: ~-|"
0x08048bf9 e812fcffff call sym.imp.puts ; int puts(const char *s)
0x08048bfe c70424098e04, mov dword [esp], str. ; [0x8048e09:4]=0x2d2d2d27 ; "'-----'"
0x08048c05 e806fcffff call sym.imp.puts ; int puts(const char *s)
0x08048c0a 8d442418 lea eax, [local_18h] ; 0x18 ; 24
0x08048c0e 89442404 mov dword [local_4h], eax
0x08048c12 c70424008d04, mov dword [esp], 0x8048d00 ; [0x8048d00:4]=0xa007525
0x08048c19 e842fcffff call sym.imp.__isoc99_scanf ; int scanf(const char *format)
```

| - offset - | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | A  | B  | C  | D  | E  | F  | 0123456789ABCDEF |
|------------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|------------------|
| 0x08048d00 | 25 | 75 | 00 | 0a | 00 | 00 | 00 | 00 | 1b | 5b | 33 | 32 | 6d | 2e | 2d | 2d | %u+++++[32m,--   |

```
* 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 ((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 is not a prime and also
 * that god(rnd-1) primes = 1 (except for 2) */
 if (((mod[i] + delta) % primes[i]) == 0) {
 delta += 2;
 }
 }
}
```

# MBE Lab1A - Calling the authentication routine

```
static int probable_prime(BIGNUM *rnd, int bits) {
 int i;
 int t;
 unsigned long delta;
 unsigned long 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)) {
 return 0;
 }
}
```

```
0x08048c1e 8b442418 mov eax, dword [local_18h] ; [0x18:4]=-1 ; 24
0x08048c22 89442404 mov dword [local_4h], eax
0x08048c26 8d44241c lea eax, [local_1ch] ; 0x1c ; 28
0x08048c2a 890424 mov dword [esp], eax
0x08048c2d e8dddfffff call sym.auth
0x08048c32 85c0 test eax, eax
,=< 0x08048c34 751f jne 0x8048c55
| 0x08048c36 c70424638e04. mov dword [esp], str.Authenticated ; [0x8048e63:4]=0x68747541 ; "Authenticated!"
| 0x08048c3d e8cefbffff call sym.imp.puts ; int puts(const char *s)
| 0x08048c42 c70424728e04. mov dword [esp], str.bin_sh ; [0x8048e72:4]=0x6e69622f ; "/bin/sh"
| 0x08048c49 e8d2fbffff call sym.imp.system ; int system(const char *string)
| 0x08048c4e b800000000 mov eax, 0
,==< 0x08048c53 eb05 jmp 0x8048c5a
|`-> 0x08048c55 b801000000 mov eax, 1
| ; CODE XREF from main (0x8048c53)
`-> 0x08048c5a 8b54243c mov edx, dword [local_3ch] ; [0x3c:4]=-1 ; '<' ; 60
0x08048c5e 653315140000. xor edx, dword gs:[0x14]
,=< 0x08048c65 7405 je 0x8048c6c
| 0x08048c67 e894fbffff call sym.imp.__stack_chk_fail ; void __stack_chk_fail(void)
`-> 0x08048c6c c9 leave
0x08048c6d c3 ret
```

```
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 (((mod[0] * delta % prime[i]) <= 1) &&
 delta % 2)
```

```
static int probable_prime(BIGNUM *rnd, int bits) {
 int i;
 unsigned long mod = NUMPRIMES;
 BIGNUM p; BN_GENCB cb; primes[0] = primes[NUMPRIMES - 1];
 while (BN_is_probable_prime(rnd, bits, &p, &cb))
 if (++i == mod) break;
 return i;
}

again:
if (!BN_rand(rnd, bits, BN_RAND_TOP_TWO, BN_RAND_BOTTOM_ODD))
```

```

 }
} else {
 for (i = 1; i < NUMPRIMES; i++) {
 /* check that rnd is not a prime and also
 * that gcd(rnd-1,prime) = 1 (except for 2) */
 if (((mod(rnd-1,prime) != 0) || gcd(rnd-1,prime) != 1) && i < NUMPRIMES) {

```

# MBE Lab1A - auth() 2/6: Antidebugging with ptrace

```

0x08048a5f c744240c0000. mov dword [local_ch_2], 0
0x08048a67 c74424080100. mov dword [local_8h], 1
0x08048a6f c74424040000. mov dword [local_4h], 0
0x08048a77 c70424000000. mov dword [esp], 0
0x08048a7e e8edfdffff call sym.imp.ptrace
0x08048a83 83f8ff cmp eax, 0xffffffffffffffff
0x08048a86 752e jne 0x8048ab6
0x08048a88 c70424088d04. mov dword [esp], str.e_32m.-----+
0x08048a8f e87cfdffff call sym.imp.puts ; int puts(const char *s)
0x08048a94 c704242c8d04. mov dword [esp], str.e_31m____TAMPERING_DETECTED ; [0x
0x08048a9b e870fdffff call sym.imp.puts ; int puts(const char *s)
0x08048aa0 c70424508d04. mov dword [esp], str.e_32m ; [0x8048d50:4]=0x32335b1b
0x08048aa7 e864fdffff call sym.imp.puts ; int puts(const char *s)
0x08048aac b801000000 mov eax, 1
0x08048ab1 e98c000000 jmp 0x8048b42

```

```

static int probable_prime(BIGNUM *rnd, int bits) {
 int i;
 BN_CTX *ctx = BN_CTX_new();
 BN_GENCB cb = {0};
 BN_GENCB_set(&cb, NULL, NULL, NULL, NULL);
 BN_rand(rnd, bits, BN_RAND_TOP_TWO, BN_RAND_BOTTOM_ODD);
 /* we now have a random number 'rnd' to test. */
 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 (((mod[0] - 1) % primes[i]) == 0) {
 delta *= 2;
 if (delta > maxdelta) {
 goto again;
 }
 goto loop;
 }
 }
}
return 0;
}

```



# MBE Lab1A - auth() 3/6: Pre-loop math

0x08048ab6  
0x08048ab9  
0x08048abc  
0x08048abf  
0x08048ac2  
0x08048ac7  
0x08048acc

8b4508  
83c003  
0fb600  
0fbec0  
3537130000  
05eded5e00  
8945f0

mov eax, dword [arg\_8h]  
add eax, 3  
movzx eax, byte [eax]  
movsx eax, al  
xor eax, 0x1337  
add eax, 0x5eeded  
mov dword [local\_10h], eax

```
static int probable_prime(BIGNUM *rnd, int bits) {
 int i;
 uint64_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;
 }

 /* 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;
 }
 }
 }
}
```

# MBE Lab1A - auth() 4/6: Loop header, restricting chars

|         |            |              |                            |
|---------|------------|--------------|----------------------------|
|         | 0x08048acf | c745ec000000 | mov dword [local_14h], 0   |
| ,=<     | 0x08048ad6 | eb4e         | jmp 0x8048b26              |
| .,----> | 0x08048ad8 | 8b55ec       | mov edx, dword [local_14h] |
| :       | 0x08048adb | 8b4508       | mov eax, dword [arg_8h]    |
| :       | 0x08048ade | 01d0         | add eax, edx               |
| :       | 0x08048ae0 | 0fb600       | movzx eax, byte [eax]      |
| :       | 0x08048ae3 | 3c1f         | cmp al, 0x1f               |
| ,=====< | 0x08048ae5 | 7f07         | jg 0x8048aee               |
| l:      | 0x08048ae7 | b801000000   | mov eax, 1                 |
| ,=====< | 0x08048aec | eb54         | jmp 0x8048b42              |

```
static int probable_prime(BIGNUM *rnd, int bits) {
 int i;
 if (bits < 16) {
 mode = NUMPRIMES;
 maxdelta = BN_MASK2 - primes[NUMPRIMES - 1];
 if (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_rand_word(rnd);
 if ((BN_ULONG)primes[i] < mod) {
 goto again;
 }
 if ((mod % primes[i]) == 0) {
 return 0;
 }
 if ((mod >= primes[i] + maxdelta)) {
 goto again;
 }
 }
 /* that's true. */
 for (i = 1; i < NUMPRIMES; i++) {
 if ((mod % primes[i]) == 0) {
 return 0;
 }
 }
 /* check that rnd is not a prime and also
 * that god(rnd-1) == 1 (except for 2) */
 if ((mod % 2) == 0) {
 return 0;
 }
 if ((mod % 3) == 0) {
 return 0;
 }
 if ((mod % 5) == 0) {
 return 0;
 }
 if ((mod % 7) == 0) {
 return 0;
 }
 if ((mod % 11) == 0) {
 return 0;
 }
 if ((mod % 13) == 0) {
 return 0;
 }
 if ((mod % 17) == 0) {
 return 0;
 }
 if ((mod % 19) == 0) {
 return 0;
 }
 if ((mod % 23) == 0) {
 return 0;
 }
 if ((mod % 29) == 0) {
 return 0;
 }
 if ((mod % 31) == 0) {
 return 0;
 }
 if ((mod % 37) == 0) {
 return 0;
 }
 if ((mod % 41) == 0) {
 return 0;
 }
 if ((mod % 43) == 0) {
 return 0;
 }
 if ((mod % 47) == 0) {
 return 0;
 }
 if ((mod % 53) == 0) {
 return 0;
 }
 if ((mod % 59) == 0) {
 return 0;
 }
 if ((mod % 61) == 0) {
 return 0;
 }
 if ((mod % 67) == 0) {
 return 0;
 }
 if ((mod % 71) == 0) {
 return 0;
 }
 if ((mod % 73) == 0) {
 return 0;
 }
 if ((mod % 79) == 0) {
 return 0;
 }
 if ((mod % 83) == 0) {
 return 0;
 }
 if ((mod % 89) == 0) {
 return 0;
 }
 if ((mod % 97) == 0) {
 return 0;
 }
 if ((mod % 101) == 0) {
 return 0;
 }
 if ((mod % 103) == 0) {
 return 0;
 }
 if ((mod % 107) == 0) {
 return 0;
 }
 if ((mod % 109) == 0) {
 return 0;
 }
 if ((mod % 113) == 0) {
 return 0;
 }
 if ((mod % 127) == 0) {
 return 0;
 }
 if ((mod % 131) == 0) {
 return 0;
 }
 if ((mod % 137) == 0) {
 return 0;
 }
 if ((mod % 139) == 0) {
 return 0;
 }
 if ((mod % 149) == 0) {
 return 0;
 }
 if ((mod % 151) == 0) {
 return 0;
 }
 if ((mod % 157) == 0) {
 return 0;
 }
 if ((mod % 163) == 0) {
 return 0;
 }
 if ((mod % 167) == 0) {
 return 0;
 }
 if ((mod % 173) == 0) {
 return 0;
 }
 if ((mod % 179) == 0) {
 return 0;
 }
 if ((mod % 181) == 0) {
 return 0;
 }
 if ((mod % 187) == 0) {
 return 0;
 }
 if ((mod % 191) == 0) {
 return 0;
 }
 if ((mod % 193) == 0) {
 return 0;
 }
 if ((mod % 197) == 0) {
 return 0;
 }
 if ((mod % 199) == 0) {
 return 0;
 }
 if ((mod % 211) == 0) {
 return 0;
 }
 if ((mod % 223) == 0) {
 return 0;
 }
 if ((mod % 227) == 0) {
 return 0;
 }
 if ((mod % 229) == 0) {
 return 0;
 }
 if ((mod % 233) == 0) {
 return 0;
 }
 if ((mod % 239) == 0) {
 return 0;
 }
 if ((mod % 241) == 0) {
 return 0;
 }
 if ((mod % 251) == 0) {
 return 0;
 }
 if ((mod % 257) == 0) {
 return 0;
 }
 if ((mod % 263) == 0) {
 return 0;
 }
 if ((mod % 269) == 0) {
 return 0;
 }
 if ((mod % 271) == 0) {
 return 0;
 }
 if ((mod % 277) == 0) {
 return 0;
 }
 if ((mod % 281) == 0) {
 return 0;
 }
 if ((mod % 283) == 0) {
 return 0;
 }
 if ((mod % 293) == 0) {
 return 0;
 }
 if ((mod % 307) == 0) {
 return 0;
 }
 if ((mod % 311) == 0) {
 return 0;
 }
 if ((mod % 313) == 0) {
 return 0;
 }
 if ((mod % 317) == 0) {
 return 0;
 }
 if ((mod % 331) == 0) {
 return 0;
 }
 if ((mod % 337) == 0) {
 return 0;
 }
 if ((mod % 347) == 0) {
 return 0;
 }
 if ((mod % 353) == 0) {
 return 0;
 }
 if ((mod % 359) == 0) {
 return 0;
 }
 if ((mod % 367) == 0) {
 return 0;
 }
 if ((mod % 373) == 0) {
 return 0;
 }
 if ((mod % 379) == 0) {
 return 0;
 }
 if ((mod % 383) == 0) {
 return 0;
 }
 if ((mod % 389) == 0) {
 return 0;
 }
 if ((mod % 397) == 0) {
 return 0;
 }
 if ((mod % 401) == 0) {
 return 0;
 }
 if ((mod % 409) == 0) {
 return 0;
 }
 if ((mod % 419) == 0) {
 return 0;
 }
 if ((mod % 421) == 0) {
 return 0;
 }
 if ((mod % 431) == 0) {
 return 0;
 }
 if ((mod % 433) == 0) {
 return 0;
 }
 if ((mod % 439) == 0) {
 return 0;
 }
 if ((mod % 443) == 0) {
 return 0;
 }
 if ((mod % 449) == 0) {
 return 0;
 }
 if ((mod % 457) == 0) {
 return 0;
 }
 if ((mod % 461) == 0) {
 return 0;
 }
 if ((mod % 463) == 0) {
 return 0;
 }
 if ((mod % 467) == 0) {
 return 0;
 }
 if ((mod % 479) == 0) {
 return 0;
 }
 if ((mod % 487) == 0) {
 return 0;
 }
 if ((mod % 491) == 0) {
 return 0;
 }
 if ((mod % 499) == 0) {
 return 0;
 }
 if ((mod % 503) == 0) {
 return 0;
 }
 if ((mod % 509) == 0) {
 return 0;
 }
 if ((mod % 521) == 0) {
 return 0;
 }
 if ((mod % 523) == 0) {
 return 0;
 }
 if ((mod % 541) == 0) {
 return 0;
 }
 if ((mod % 547) == 0) {
 return 0;
 }
 if ((mod % 557) == 0) {
 return 0;
 }
 if ((mod % 563) == 0) {
 return 0;
 }
 if ((mod % 569) == 0) {
 return 0;
 }
 if ((mod % 571) == 0) {
 return 0;
 }
 if ((mod % 577) == 0) {
 return 0;
 }
 if ((mod % 587) == 0) {
 return 0;
 }
 if ((mod % 593) == 0) {
 return 0;
 }
 if ((mod % 599) == 0) {
 return 0;
 }
 if ((mod % 601) == 0) {
 return 0;
 }
 if ((mod % 607) == 0) {
 return 0;
 }
 if ((mod % 613) == 0) {
 return 0;
 }
 if ((mod % 617) == 0) {
 return 0;
 }
 if ((mod % 619) == 0) {
 return 0;
 }
 if ((mod % 623) == 0) {
 return 0;
 }
 if ((mod % 629) == 0) {
 return 0;
 }
 if ((mod % 631) == 0) {
 return 0;
 }
 if ((mod % 637) == 0) {
 return 0;
 }
 if ((mod % 641) == 0) {
 return 0;
 }
 if ((mod % 643) == 0) {
 return 0;
 }
 if ((mod % 647) == 0) {
 return 0;
 }
 if ((mod % 653) == 0) {
 return 0;
 }
 if ((mod % 659) == 0) {
 return 0;
 }
 if ((mod % 661) == 0) {
 return 0;
 }
 if ((mod % 673) == 0) {
 return 0;
 }
 if ((mod % 677) == 0) {
 return 0;
 }
 if ((mod % 683) == 0) {
 return 0;
 }
 if ((mod % 687) == 0) {
 return 0;
 }
 if ((mod % 691) == 0) {
 return 0;
 }
 if ((mod % 697) == 0) {
 return 0;
 }
 if ((mod % 701) == 0) {
 return 0;
 }
 if ((mod % 709) == 0) {
 return 0;
 }
 if ((mod % 713) == 0) {
 return 0;
 }
 if ((mod % 719) == 0) {
 return 0;
 }
 if ((mod % 727) == 0) {
 return 0;
 }
 if ((mod % 733) == 0) {
 return 0;
 }
 if ((mod % 739) == 0) {
 return 0;
 }
 if ((mod % 743) == 0) {
 return 0;
 }
 if ((mod % 751) == 0) {
 return 0;
 }
 if ((mod % 757) == 0) {
 return 0;
 }
 if ((mod % 761) == 0) {
 return 0;
 }
 if ((mod % 769) == 0) {
 return 0;
 }
 if ((mod % 773) == 0) {
 return 0;
 }
 if ((mod % 787) == 0) {
 return 0;
 }
 if ((mod % 797) == 0) {
 return 0;
 }
 if ((mod % 809) == 0) {
 return 0;
 }
 if ((mod % 811) == 0) {
 return 0;
 }
 if ((mod % 823) == 0) {
 return 0;
 }
 if ((mod % 827) == 0) {
 return 0;
 }
 if ((mod % 833) == 0) {
 return 0;
 }
 if ((mod % 839) == 0) {
 return 0;
 }
 if ((mod % 853) == 0) {
 return 0;
 }
 if ((mod % 857) == 0) {
 return 0;
 }
 if ((mod % 859) == 0) {
 return 0;
 }
 if ((mod % 863) == 0) {
 return 0;
 }
 if ((mod % 877) == 0) {
 return 0;
 }
 if ((mod % 881) == 0) {
 return 0;
 }
 if ((mod % 883) == 0) {
 return 0;
 }
 if ((mod % 887) == 0) {
 return 0;
 }
 if ((mod % 893) == 0) {
 return 0;
 }
 if ((mod % 899) == 0) {
 return 0;
 }
 if ((mod % 907) == 0) {
 return 0;
 }
 if ((mod % 911) == 0) {
 return 0;
 }
 if ((mod % 919) == 0) {
 return 0;
 }
 if ((mod % 929) == 0) {
 return 0;
 }
 if ((mod % 937) == 0) {
 return 0;
 }
 if ((mod % 941) == 0) {
 return 0;
 }
 if ((mod % 947) == 0) {
 return 0;
 }
 if ((mod % 953) == 0) {
 return 0;
 }
 if ((mod % 967) == 0) {
 return 0;
 }
 if ((mod % 971) == 0) {
 return 0;
 }
 if ((mod % 977) == 0) {
 return 0;
 }
 if ((mod % 983) == 0) {
 return 0;
 }
 if ((mod % 991) == 0) {
 return 0;
 }
 if ((mod % 997) == 0) {
 return 0;
 }
 return 1;
 }
 }
}
```

# MBE Lab1A - auth() 5/6: Loop body, much math

|         |            |              |       |                    |                   |                                                    |
|---------|------------|--------------|-------|--------------------|-------------------|----------------------------------------------------|
| `-----> | 0x08048aee | 8b55ec       | mov   | edx,               | dword [local_14h] | static int probable_prime(BIGNUM *rnd, int bits) { |
| :       | 0x08048af1 | 8b4508       | mov   | eax,               | dword [arg_8h]    | int i;                                             |
| :       | 0x08048af4 | 01d0         | add   | eax,               | edx               | while (i < mods[NUMPRIMES];                        |
| :       | 0x08048af6 | 0fb600       | movzx | eax,               | byte [eax]        | delta;                                             |
| :       | 0x08048af9 | 0fbec0       | movsx | eax,               | al                | maxdelta = BN_MASK2 - primes[NUMPRIMES - 1];       |
| :       | 0x08048afc | 3345f0       | xor   | eax,               | dword [local_10h] | return 0;                                          |
| :       | 0x08048aff | 89c1         | mov   | ecx,               | eax               | if (i == 0) {                                      |
| :       | 0x08048b01 | ba2b3b2388   | mov   | edx,               | 0x88233b2b        | BN_RAND_BOTTOM_ODD)) {                             |
| :       | 0x08048b06 | 89c8         | mov   | eax,               | ecx               | test, */                                           |
| :       | 0x08048b08 | f7e2         | mul   | edx                |                   | (i < NUMPRIMES)primes[i]);                         |
| :       | 0x08048b0a | 89c8         | mov   | eax,               | ecx               | if (i < NUMPRIMES) {                               |
| :       | 0x08048b0c | 29d0         | sub   | eax,               | edx               | if (i < NUMPRIMES) {                               |
| :       | 0x08048b0e | d1e8         | shr   | eax,               | 1                 | if (i < NUMPRIMES) {                               |
| :       | 0x08048b10 | 01d0         | add   | eax,               | edx               | if (i < NUMPRIMES) {                               |
| :       | 0x08048b12 | c1e80a       | shr   | eax,               | 0xa               | if (i < NUMPRIMES) {                               |
| :       | 0x08048b15 | 69c039050000 | imul  | eax,               | eax, 0x539        | if (i < NUMPRIMES) {                               |
| :       | 0x08048b1b | 29c1         | sub   | ecx,               | eax               | if (i < NUMPRIMES) {                               |
| :       | 0x08048b1d | 89c8         | mov   | eax,               | ecx               | if (i < NUMPRIMES) {                               |
| :       | 0x08048b1f | 0145f0       | add   | dword [local_10h], | eax               | if (i < NUMPRIMES) {                               |

## MBE Lab1A - auth() 6/6: Loop footer, return targets

```
static int probable_prime(BIGNUM *rnd, int bits) {
 int i;
 int32_t mod[NUMPRIMES];
 BN_CTX *ctx = BN_CTX_new();
 BN_mask2b(rnd, 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)) {
```

```

| :||| 0x08048b22 8345ec01 add dword [local_14h], 1
| :||| ; CODE XREF from sym.auth (0x8048ad6)
| :||`-> 0x08048b26 8b45ec mov eax, dword [local_14h]
| :|| 0x08048b29 3b45f4 cmp eax, dword [local_ch]
| `====< 0x08048b2c 7caa jl 0x8048ad8
| || 0x08048b2e 8b450c mov eax, dword [arg_ch]
| || 0x08048b31 3b45f0 cmp eax, dword [local_10h]
| ||,=< 0x08048b34 7407 je 0x8048b3d
| ||| 0x08048b36 b801000000 mov eax, 1
| ,====< 0x08048b3b eb05 jmp 0x8048b42
| |||`-> 0x08048b3d b800000000 mov eax, 0
| ||| ; CODE XREFS from sym.auth (0x8048a5a, 0x8048ab1, 0x8048a
| -````--> 0x08048b42 c9 leave
| 0x08048b43 c3 ret

```

```

 }
} else {
 for (i = 1; i < NUMPRIMES; i++) {
 /* check that rml is not a prime and also
 * that gcd(rml-1, primes[i]) = 1 (except for 2) */
 if (((mod(rml-1, primes[i]) != 1) ||
 (primes[i] == 2 && mod(rml-1, 2) == 0)) {

```

Any questions on the assembly, before we get to z3ing it?

```
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)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;
 }
 }
 }
 }
}
```

# MBE Lab1A - Z3ing auth() 1/7: Setting up variables

```
import z3
solver = z3.Solver()
wanted_length = 8
assert wanted_length > 5 # checked at 0x08048a4f
sym_username = [z3.BitVec('x{i}'.format(i=i), 8) for i in range(wanted_length)]
sym_serial = z3.BitVec('serial', 32)
```

- ▶ 8-bit entries for each character
- ▶ 32-bit serial number
- ▶ Concrete input length: z3.Array is more expensive

```
static int probable_prime(BIGNUM *rnd, int bits) {
 int i;
 unsigned t;
 mods[NUMPRIMES];
 BN_ULONG mod;
 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)primes[i]) {
 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)1) - 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;
 }
 }
 }
 }
 return 1;
}
```

```
static int probable_prime(BIGNUM *rnd, int bits) {
 int i;
 uint32_t mods[NUMPRIMES];
 for (i = 0; i < NUMPRIMES; i++)
 mods[i] = BIGNUMS[i] % primes[i];
 char is_single_word = bits <= BN_BITS2;
 if (is_single_word) {
```

```
again;
if ((BN_rand(rnd, bits - BN_RAND_TOP_TWO, BN_RAND_ODD)) < 0)
 return 0;
}

/* We now have a random number 'rnd' to test. */
for (i = 1; i < NUMPRIMES; i++) {
 BN_ULONG mod = BN_get_word(rnd, (BN_ULONG)primes[i]);
 r = *(uint8_t*)(arg_8h+3)
 return 0;
 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;
 }
 arg_8h+3) *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
 * these three primes. (We don't
 * know if there are any larger than 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 % primes[i] == 0)
 goto loop;
 }
 }
} else {
 for (i = 1; i < NUMPRIMES; i++) {
 /* check that rnd is not a prime and also
 * that gcd(rnd, primes[i]) == 1 (except for 2) */
 if ((mods[i] + delta) % primes[i] == 0) {
 delta += 2;
 if (delta % primes[i] == 0)
 goto loop;
 }
 }
}

```

- ```
ad(rnd-1,prime)=1 (except for 2) ✓
```

MBE Lab1A - Z3ing auth() 3/7: Translating the loop header/footer

The following x86:

```

0x08048acf      c745ec000000.    mov dword [local_14h], 0
0x08048ad6      eb4e             jmp 0x8048b26
...
0x08048b22      8345ec01         add dword [local_14h], 1
0x08048b26      8b45ec           mov eax, dword [local_14h]
0x08048b29      3b45f4           cmp eax, dword [local_ch]
0x08048b2c      7caa            j! 0x8048ad8

```

```
static int probe_prime(BIGNUM *int_bits) {
    int i;
    uint16_t mods[NUMPRIMES];
    BN_ULONG mod;
    BN_ULONG size_limit = BN_ULONG_MAX - 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;
}
```

Translates to the following C:

```
for(int local_14h=0; local_14h < local_ch; local_14h++) {
    /* loop body is between 0x08048ad6 and 0x08048b22, exclusive on both ends */
}
```

```

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

        // We should not end a 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
        *    3.
        * 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 ((mod[(i + delta) % primes[i]] == 0) {
                if (i == 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[i]) = 1 (except for 2) */
            if (((mod[i] * delta % primes[i]) > 1) &&

```

So we'll gloss that as the following in Python

```
local_ch = len(sym_username) # this is set by the strlen at 0x08048a3e
for local_14h in range(local_ch):
    pass # we'll translate the loop body here
```


MBE Lab1A - Z3ing auth() 4/7: Translating the loop body: 1/2

x86:

```
0x08048ad8    mov edx, dword [local_14h]
0x08048adb    mov eax, dword [arg_8h]
0x08048ade    add eax, edx
0x08048ae0    movzx eax, byte [eax]
0x08048ae3    cmp al, 0x1f
0x08048ae5    jg 0x8048aee
```

```
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]
```

```
0x08048aff    mov ecx, eax
0x08048b01    mov edx, 0x88233b2b
0x08048b06    mov eax, ecx
```

Python:

```
solver.add(sym_username[local_14h] > 0x1f)
```

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

```
ecx = eax
edx = z3.BitVecVal(0x88233b2b, 32)
eax = ecx
```

```
static int probable_prime(BIGNUM *rnd, int bits) {
    int i;
    BN_ULONG mod;
    BN_ULONG mod_bits = BN_BITS2 - 1;
    BN_ULONG mod2 = mod * UNPRIMES - 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 < UNPRIMES; 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) {
                /* 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;
            }
        }
        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
            *    0 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 we don't want to reject too many small primes where
            *    that's not the case.
            *    One should be able to check that rnd-1 is coprime to all the known
            *    primes by checking that (rnd-1) % primes[i] != 0 for all i.
            */
            for (i = 1; i < UNPRIMES; i++) {
                if ((mod[i] + delta) % primes[i] == 0) {
                    delta += 2;
                    if (delta > maxdelta) {
                        goto again;
                    }
                    goto loop;
                }
            }
        } else {
            for (i = 1; i < UNPRIMES; i++) {
                /* check that rnd is not a prime and also
                * that gcd(rnd-1, primes) = 1 (except for 2) */
                if (((mod[i] + delta) % primes[i]) == 0) {
                    delta += 2;
                    if (delta > maxdelta) {
                        goto again;
                    }
                    goto loop;
                }
            }
        }
    }
}
```

```

static int probable_prime(BIGNUM *rnd, int bits) {
    int i;
    /* odd */
    mod = BN_MOD_2;
    BN_ULONG rnd_bits = BN_BITS2 - primes[NUMPRIMES - 1];
    /* is single word */
    if (bits <= BN_BITS2) {
        again:
        if (!BN_rand(rnd, bits, BN_RAND_TOP_TWO, BN_RAND_BOTTOM_ODD)) {
            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) {
                delta = size_limit;
            }
        }

        /*
         * 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 ((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 is not a prime and also
             * that gcd(rnd-1, primes) = 1 (except for 2) w/
             * delta = 2;
             */
            if ((mod[i] + delta) % primes[i] == 0) {
                delta += 2;
            }
        }
    }
}

```

Python:

```
mul    edx
```

```
mul_result = z3.ZeroExt(32, eax) * z3.ZeroExt(32, edx)
edx = z3.Extract(63, 32, mul_result)
eax = z3.Extract(31, 0, mul_result)
```

```
mov eax, ecx
```

```
sub    eax, edx
```

```
shr eax, 1
```

```
add eax, edx
```

```
shr eax, 0xa
```

```
eax -= edx
```

```
eax = eax >> 1
```

```
eax += edx
```

```
eax = eax >> 0xa
```

```
imul eax, eax, 0x539
```

```
eax = z3.Extract(31, 0, z3.SignExt(32, eax) * 0x539)
```

```
sub ecx, eax
```

```
mov eax, ecx
```

```
add dword [local 10h], eax
```

```
ecx -= eax
```

```
eax = ecx
```

```
local 10h += eax
```

MBE Lab1A - Z3ing auth() 6/7: Solving for a valid serial

```
solver.add(sym_serial == local_10h) # outside the loop
solver.push() # backtracking point for next demo
username = 'username'
for (x, y) in zip(username, sym_username):
    solver.add(ord(x) == y)
assert solver.check().r == 1
model = solver.model()
serial = model.evaluate(sym_serial)
print('serial for name %r is %r' % (username, serial))
```

```
serial for name 'username' is 6234463
```

```
static inline int prime(BIGNUM *rnd, int bits) {
    int i;
    uint32_t mods[NUMPRIMES];
    BN_ULONG maxsize = 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] = (uint32_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) ||
                ((i > 1) && (rnd_word % primes[i]) == 0)) {
                delta += 2;
                if (delta > maxdelta) {
                    goto again;
                }
            }
        }
    }
}
```

MBE Lab1A - Z3ing auth() 7/7: Solving for a valid username

```
solver.pop() # remove the constraints on the provided username
solver.add(sym_serial == serial+1)
assert solver.check().r == 1
model = solver.model()
username2 = ''.join(chr(model.evaluate(x).as_long()) for x in sym_username)
serial2 = model.evaluate(sym_serial)
print('serial for name %r is %r' % (username2, serial2))
```

```
serial for name 'sEa2):2-' is 6234464
```

```
static inline prime(BIGNUM *rnd, int bits) {
    int i;
    uint32_t mods[NUMPRIMES];
    BN_ULONG maxdelta = 0, delta;
    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] = (uint32_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 (mods[i] > size_limit) {
                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;
                    }
                }
            }
        }
    }
}
```


Resources

- ▶ <https://github.com/Z3Prover/z3/>
- ▶ <https://pypi.org/project/z3-solver/>
- ▶ <https://rise4fun.com/Z3/tutorialcontent/guide>
- ▶ https://en.wikipedia.org/wiki/Satisfiability_modulo_theories
- ▶ <https://github.com/RPISEC/MBE>

```
static int 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)) &
        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) {
            delta = size_limit;
        }
    } else {
        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;
                }
            }
        }
    }
}
```