

# Asymmetric Cryptography Part 2

## RPISEC

Avi Weinstock (aweinstock)

Jacques Becker (off\_by\_1\_error), Ethan Wright (eth4)

November 1, 2019

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

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

    /* we now have a random number 'rnd' to test. */
    for (i = 1; i < NUMPRIMES; i++) {
        BN_ULONG mod = BN_mod_word(rnd, (BN_ULONG)primes[i]);
        if (mod == (BN_ULONG)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;
                if (delta > maxdelta) {
                    goto again;
                }
            }
        }
    }
}
```

# RSA Recap

```
from codecs import encode
from gmpy import invert, next_prime
import os
d = 0
while d == 0:
    p = next_prime(int(encode(os.urandom(1024/8), 'hex'), 16))
    q = next_prime(int(encode(os.urandom(1024/8), 'hex'), 16))
    n = p * q
    phi = (p - 1) * (q - 1)
    e = 65537
    d = invert(e, phi)
```

```
message = int(encode('hello', 'hex'), 16)
ciphertext = pow(message, e, n)
assert pow(ciphertext, d, n) == message
```

```
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;
            }
        }
    }
}
```

# Example RSA encryption/decryption

- ▶  $n = p * q$
- ▶  $\varphi(n) = (p - 1) * (q - 1)$
- ▶  $e * d \equiv 1 \pmod{\varphi(n)}$
- ▶  $\text{encrypt}(x) = x^{e \% n}$
- ▶  $\text{decrypt}(x) = x^{d \% n}$
- ▶  $\text{pow}(x, k, n) = x^{k \% n}$

- ▶ Public:  $(n, e) = (667, 3)$
- ▶ Message "hi", encoded as  $7 * 26 + 8 = 190$
- ▶  $\text{pow}(190, 3, 667) == 239$

```
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 =
                /* Randomly define a new size limit. */
                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;
                }
            }
        }
    }
    return 1;
}
```

# Example RSA encryption/decryption

- ▶  $n = p * q$
- ▶  $\varphi(n) = (p - 1) * (q - 1)$
- ▶  $e * d \equiv 1 \pmod{\varphi(n)}$
- ▶  $\text{encrypt}(x) = x^{e \% n}$
- ▶  $\text{decrypt}(x) = x^{d \% n}$
- ▶  $\text{pow}(x, k, n) = x^{k \% n}$

- ▶ Public:  $(n, e) = (667, 3)$
- ▶ Message "hi", encoded as  $7 * 26 + 8 = 190$
- ▶  $\text{pow}(190, 3, 667) == 239$
- ▶ Private:  $(p, q, d) = (23, 29, 411)$
- ▶  $(3 * 411) \% (22 * 28) == 1$
- ▶  $\text{pow}(239, 411, 23 * 29) == 190$

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

# Why to use $e = 65537$

- ▶  $2^{16+1} = 65537_{10} = 10001_{16} = 100000000000000001_2$
- ▶ It's prime, so  $\text{invert}(65537, \varphi(n))$  is more likely to exist
- ▶ It mitigates multiple attacks:
  - ▶ Cube root
  - ▶ Hastad's broadcast
  - ▶ Coppersmith's short pad
- ▶ Since it only has 2 bits set, it's efficient to compute via repeated squaring:  
$$m^{2^{16}+1} = m^{2^{16}} * m = (m^{2^8} * m^{2^8}) * m = ((m^{2^4} * m^{2^4}) * (m^{2^4} * m^{2^4})) * m = \dots$$

```
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) {
            size_limit = 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
             * 1) it's greater than primes[i] because we shouldn't reject
             *    2) it's being a multiple of primes[i] because it's a multiple of
             *    2) that it's not a multiple of a known prime. We don't
             *    check that rnd-1 is also coprime to all the known
             *    primes because there aren't many small primes where
             *    that's true. */
            for (i = 1; i < NUMPRIMES; i++) {
                if (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;
                    }
                }
            }
        }
    }
}
```

# Extended Euclidean Algorithm

```
► from gmpy import gcd
def eea(x, y):
    r, s, t = x, 1, 0
    R, S, T = y, 0, 1
    while R > 0:
        q = r/R
        new = r-q*R, s-q*S, t-q*T
        r, s, t = R, S, T
        R, S, T = new
    assert gcd(x, y) == r # gcd from euclidean algorithm
    assert r == x*s + y*t # s and t are the bezout coefficients
    xinvy = s + y*(s < 0) # modular inverse from bezout coefficients
    yinvx = t + x*(t < 0)
    if r == 1:
        assert (x * xinvy) % y == 1
        assert (y * yinvx) % x == 1
    return (r, s, t, xinvy, yinvx)
```

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

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

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

# Chinese Remainder Theorem - Statement

- ▶  $\forall \vec{a}, \vec{n} ((\forall i, j (i \neq j \rightarrow \gcd(n_i, n_j) = 1)) \rightarrow \exists x \forall i (x \equiv a_i \pmod{n_i}))$
- ▶ For a system of equations of the form  $x \equiv a_i \pmod{n_i}$
- ▶ if each  $(n_i, n_j)$  pair are relatively prime
- ▶ there is a (unique) solution  $x$  for the system of equations

```
static int prime_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 = 1;
        /* avoid underflow behavior. */
        size_limit = (((BN_ULONG)0) - get_word(rnd));
    } else {
        size_limit = (((BN_ULONG)1) << bits) - get_word(rnd) - 1;
    }
    if (size_limit < maxdelta) {
        maxdelta = size_limit;
    }
    delta = 0;

    BN_ULONG rnd_word = get_word(rnd);

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

# Chinese Remainder Theorem - Code

- ▶  $\forall \vec{a}, \vec{n} ((\forall i, j (i \neq j \rightarrow \gcd(n_i, n_j) = 1)) \rightarrow \exists x \forall i (x \equiv a_i \pmod{n_i}))$
- ▶ from eea import eea  
from gmpy import gcd  
from itertools import combinations  
def crt(eqns):  
 assert len(eqns) >= 2  
 assert [gcd(eqns[i][1], eqns[j][1]) == 1 for (i, j) in combinations(range(len(eqns)), 2)]  
 a0, n0 = eqns[0]  
 a1, n1 = eqns[1]  
 \_, m0, m1, \_, \_ = eea(n0, n1)  
 assert m0\*n0 + m1\*n1 == 1  
 x = (a0\*m1\*n1 + a1\*m0\*n0) % (n0 \* n1)  
 if len(eqns) > 2:  
 x = crt([(x, n0\*n1)]+eqns[2:])  
 for (a, n) in eqns:  
 assert x % n == a % n  
 return x

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

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

/* we now have a random number 'rnd' to test. */
for (i = 1; i < NUMPRIMES; i++) {
    BN_ULONG mod = get_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);
    }
    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;
            }
        }
    }
}
```



# Chinese Remainder Theorem - Example

- ▶  $\forall \vec{a}, \vec{n} ((\forall i, j (i \neq j \rightarrow \gcd(n_i, n_j) = 1)) \rightarrow \exists x \forall i (x \equiv a_i \pmod{n_i}))$
- ▶  $x \equiv 3 \pmod{5} \wedge x \equiv 4 \pmod{7}$
- ▶  $\text{eea}(5, 7)$  gives us  $(3, -2)$  as the Bezout coefficients
- ▶ This tells us that  $3 * 5 + (-2) * 7 = 1$
- ▶ CRT gives us that  $x = 3 * (-2) * 7 + 5 * 3 * 5$  solves the equation

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

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

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

# CRT Application - Breaking same-message RSA Theory

- ▶ Suppose we have  $c_1 \equiv m^3 \pmod{n_1}$ ,  $c_2 \equiv m^3 \pmod{n_2}$ , and  $c_3 \equiv m^3 \pmod{n_3}$ .
- ▶  $\text{crt}([(c_1, n_1), (c_2, n_2), (c_3, n_3)]) \equiv m^3 \pmod{n_1 * n_2 * n_3}$
- ▶ Since  $n_1 * n_2 * n_3 > \max(n_1, n_2, n_3)$ , even if  $m^3$  wrapped on each of the moduli, it is likely cube-rootable mod the product of the moduli

```
static int probable_prime(BIGNUM *rnd, int bits) {
    int i;
    uint16_t mods[NUMPRIMES];
    BN_ULONG mod = 0;
    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 =
            ((BN_ULONG)1) << bits;
        BN_ULONG rnd_word = get_word(rnd);
        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;
            }
        }
    }
}
```

# CRT Application - Breaking same-message RSA - Example

- ▶  $\text{crt}([(c_1, n_1), (c_2, n_2), (c_3, n_3)]) \equiv m^3 \pmod{n_1 * n_2 * n_3}$
- ▶  $(c_1, n_1) = (239, 667), (c_2, n_2) = (95, 589), (c_3, n_3) = (643, 1517)$
- ▶  $\text{crt}([(239, 667), (95, 589), (643, 1517)]) = 6859000$
- ▶  $\sqrt[3]{6859000} = 190$

```
static int probable_prime(BIGNUM *rnd, int bits) {
    int i;
    uint16_t mods[NUMPRIMES];
    BN_ULONG mod;
    BN_ULONG maxdelta = (BN_ULONG)2 - primes[NUMPRIMES - 1];
    char is_single_word = bits <= BN_BITS2;

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

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

# CRT Application - Speeding up RSA decryption - Theory

- ▶  $\text{encrypt}(m) = \text{pow}(m, e, n)$ ,  $\text{decrypt}(c) = \text{pow}(c, d, n)$
- ▶ If  $e = 65537$ , it's small and fast to compute with (around 16 bits), but  $d$  is around the size of  $n$  (2048 bits if  $p$  and  $q$  are each 1024 bits)
- ▶  $\text{fastdecrypt}(c) = \text{crt}([\text{pow}(c, d_p, p), p], [\text{pow}(c, d_q, q), q])$ , where  $e * d_p \equiv 1 \pmod{p-1}$  and  $e * d_q \equiv 1 \pmod{q-1}$
- ▶ Works because  $\text{pow}(c, d_p, p) \equiv m \pmod{p}$  (and likewise for  $q$ ), so CRT reconstructs  $x \equiv m \pmod{p * q}$
- ▶ Is faster because  $p$  and  $q$  are only 1024 bit, so computations mod those are cheaper

```
static int prime(BIGNUM *rnd, int bits) {
    int i;
    uint16_t *p;
    BN_ULONG mod;
    BN_ULONG mask1 = 1_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;
        }
        mod |= (uint16_t)mod;
    }
    /* If bits is small that it fits into a single word then we
     * can check that it's not a multiple of any of the primes. */
    if (is_single_word) {
        BN_ULONG size_limit;
        BN_ULONG mask2 = 1_MASK2;
        /* Fixed 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 < 0)) {
        maxdelta = size_limit;
    }
    delta = 0;
    for (i = 1; i < NUMPRIMES; i++) {
        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
         *    primes that are a multiple of primes[i] because it's a multiple of
         *    2) that it's not a multiple of a known prime. We don't
         *    check that rnd-1 is also coprime to all the known
         *    primes because there aren't many small primes where
         *    that's true. */
        for (j = 1; j < NUMPRIMES && primes[j] < rnd_word; j++) {
            if (((mod[j] + delta) % primes[j]) == 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 (((mod[i] + delta) % primes[i]) == 0) {
            delta += 2;
            if (delta > maxdelta) {
                goto again;
            }
            goto loop;
        }
    }
    return 1;
}
```

# CRT Application - Speeding up RSA decryption - Code

```
► from codecs import encode
from gmpy import invert, next_prime
from crt import crt
from time import time
import os
```

```
p = next_prime(int(encode(os.urandom(4096/8), 'hex'), 16))
q = next_prime(int(encode(os.urandom(4096/8), 'hex'), 16))
n = p * q
e = 65537
d = invert(e, (p-1)*(q-1))
dp = invert(e, p-1)
dq = invert(e, q-1)

msg = int(encode('hello', 'hex'), 16)
s0 = time(); ctxt = pow(msg, e, n); t0 = time()-s0
s1 = time(); m1 = pow(ctxt, d, n); t1 = time()-s1
s2 = time(); m2 = crt([(pow(ctxt, dp, p), p), (pow(ctxt, dq, q), q)]); t2 = time()-s2
assert m1 == m2 == msg
```

```
static int_prime(BIGNUM *rnd, int bits) {
    int i;
    uint16_t mods[NUMPRIMES];
    BN_ULONG delta;
    BN_ULONG size_limit = 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 (i == NUMPRIMES - 1)
                    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;
            goto again;
        }
    }
}
```

# Saltstack 2013 e=1 bug

```
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)) <
```

## Change key generation seq

[Browse files](#)

0.15

**thatch45** authored and **basepi** committed on May 8, 20131 parent [43d8c16](#)commit [5dd304276ba5745ec21fc1e6686a0b28da29e6fc](#)

Showing 1 changed file with 1 addition and 1 deletion.

Unified

Split

2 salt/crypt.py

...

File	Line	Code
@@ -47,7 +47,7 @@		def gen_keys(keydir, keyname, keysize, user=None):

47	47	priv = '{0}.pem'.format(base)
----	----	-------------------------------

48	48	pub = '{0}.pub'.format(base)
----	----	------------------------------

49	49	
----	----	--

50	-	gen = RSA.gen_key(keysize, 1, callback=lambda x, y, z: None)
----	---	--

50	+	gen = RSA.gen_key(keysize, 65537, callback=lambda x, y, z: None)
----	---	--

51	51	cumask = os.umask(191)
----	----	------------------------

52	52	gen.save_key(priv, None)
----	----	--------------------------

53	53	os.umask(cumask)
----	----	------------------

--	--	--

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

# Resources

- ▶ [https://en.wikipedia.org/wiki/RSA\\_\(cryptosystem\)](https://en.wikipedia.org/wiki/RSA_(cryptosystem))
- ▶ [https://en.wikipedia.org/wiki/Chinese\\_remainder\\_theorem](https://en.wikipedia.org/wiki/Chinese_remainder_theorem)
- ▶ [https://en.wikipedia.org/wiki/Modular\\_arithmetic](https://en.wikipedia.org/wiki/Modular_arithmetic)
- ▶ <https://crypto.stanford.edu/~dabo/papers/RSA-survey.pdf>
- ▶ <https://cryptopals.com/>, Sets 5 and 6
- ▶ <https://docs.saltstack.com/en/latest/topics/releases/0.15.1.html#rsa-key-generation-fault>

```
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
     * don't want to exceed that many bits. */
    if (is_single_word) {
        BN_ULONG size_limit;
        if (bits == BN_BITS2) {
            size_limit = (((BN_ULONG)0) - get_word(rnd));
        } else {
            size_limit = (((BN_ULONG)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;
            }
        }
    }
}
```