A Study of the High-Speed RSA Cryptosystem on CPU using Montgomery and Karatsuba

[Group 5]

[Frank 谢欣言 1930026136]

[Jorian 赵汝杨 1930026170]

[Connor 徐贺 1930026139]

A report submitted to

Dr. Donglong Chen

for

COMP4023 Computer and Network Security

Data Science (DS)

Division of Science and Technology (DST)

[2021.12.17]

Contents

1. Introduction:

The goal of this project is to implement a high-speed RSA cryptosystem on x86 computers. The system should support both RSA encryption and decryption, with 4096-bit key size and 1024-bit message.

To achieve that, we used the Montgomery modular multiplication Algorithm and the Karatsuba multiplication algorithm to reduce the time complexity of multiplication and modular multiplication, and finally optimize the running time of this algorithm.

2. Montgomery Modular Multiplication Algorithm:

1) Concept:

To an operation of (x · y) (mod n), normally, the modular operation in it is realized by division, and division is a particularly complex operation, which involves a lot of multiplication. Therefore, we should try to avoid division in large number operation. And in order to do that, we can use the Montgomery algorithm. But first we need to know some concepts:

- Montgomery representation:

  Given a number n, n has i bits in $b-ary$ (for example, in binary, b = 2), $GCD(n,b) = 1$, the following values are pre calculated:

  $$\rho = bk,$$

  specifies a minimum k such that bk > n;

  $$\omega = -N^{-1}(mod\ \rho);$$

  And the Montgomery representation will be:

  For $0 \leqslant x \leqslant n-1$,

  $$\hat{x} = x \cdot \rho\ (mod\ n).$$

- Montgomery reduction:

  Given integers t, the calculation result of Montgomery reduction is

  $$t \cdot \rho^{-1}(mod\ n)$$

The algorithm of Montgomery reduction can be expressed as:

Algorithm ($\mathbb{Z}_N$-MontRed)

Input: A base-b, unsigned integer, $0 \le t \le N \cdot \rho - 1$
Output: A base-b, unsigned integer $r = t \cdot \rho^{-1} \pmod N$

```
r ← t
for i = 0 upto l_N − 1 step +1 do
    u ← r_i · ω (mod b)
    r ← r + (u · N · b^i)
end
r ← r/b^{l_N}
if r ≥ N then
    r ← r − N
end
return r
```

Montgomery reduction can be used to calculate a modulo worthy operation. For example, if we want to calculate m(mod n), we only need to put m's Montgomery representation m · ρ as a parameter, bring in Montgomery reduction, and the calculation result is m (mod n).

- Montgomery modular multiplication:

  A Montgomery modular multiplication includes integer multiplication and Montgomery reduction. Now we have Montgomery representation:

  $$\hat{x} = x \cdot \rho \ (mod\ n)$$

  $$\hat{y} = y \cdot \rho \ (mod\ n)$$

  The result of their multiplication is:

  $$t = \hat{x} \cdot \hat{y}$$

  $$= (x \cdot \rho) \cdot (y \cdot \rho)$$

  $$= (x \cdot y) \cdot \rho^2$$

  Finally, the result is obtained by a Montgomery reduction:

  $$\hat{t} = (x \cdot y) \cdot \rho \ (mod\ n)$$

  As we can see from above, the given input parameters are $\hat{x}$ and $\hat{y}$ and the result is $(x \cdot y) \cdot \rho \ (mod\ n)$, so Montgomery multiplication can also be written in the following form: Given the input parameters x and y, Montgomery multiplication calculates $(x \cdot y) \cdot \rho^{-1} (mod\ N)$.

2) Example:

For $421 \cdot 422 \pmod{667}$

b = 10, that is, in base 10, n = 667

Let bk > n have a minimum k of 3, so ρ= bk=10^3=1000

$$\omega = -n^{-1}(mod\ \rho) = -667^{-1}(mod\ 1000) = 997$$

$$x = 421, \hat{x} = x \cdot \rho\ (mod\ n) = 421 \cdot 1000(mod\ 667) = 123$$

$$y = 422, \hat{y} = y \cdot \rho\ (mod\ n) = 422 \cdot 1000(mod\ 667) = 456$$

So the result of calculating the Montgomery multiplication of $\hat{x}$ and $\hat{y}$ is:

$$\hat{x} \cdot \hat{y} \cdot \rho\text{^}(-1) = (421 \cdot 1000 \cdot 422 \cdot 1000) \cdot 1000\text{^}(-1)(mod\ 667)$$

$$= (421 \cdot 422) \cdot 1000\ (mod\ 667)$$

$$= 547$$

Example

In the challenge we had $b = 10$, $N = 667$, $l_N = 3$ and $k = 3$ meaning $\rho = 1000$ and $\omega = 997$. We also know that

$$\begin{aligned} x = 421 \rightsquigarrow \hat{x} &= x \cdot \rho \pmod N \\ &= 421 \cdot 1000 \pmod{667} = 123 \end{aligned}$$

$$\begin{aligned} y = 422 \rightsquigarrow \hat{y} &= y \cdot \rho \pmod N \\ &= 422 \cdot 1000 \pmod{667} = 456 \end{aligned}$$

so executing

$$\mathbb{Z}_N\text{-MontMul}(\hat{x}, \hat{y}) = \mathbb{Z}_N\text{-MontMul}(123, 456)$$

yields

i	r	$y_i\ x_0$	$u = (r_0 + y_i \cdot x_0) \cdot \omega \pmod b$	$t_0 = y_i \cdot x$	$t_1 = u \cdot N$	$t_2 = r + t_0 + t_1$	$r = t_2/b$
	⟨0,0,0,0⟩						
0	⟨0,0,0,0⟩	6 3	6	738	4002	⟨0,4,7,4⟩	⟨4,7,4,0⟩
1	⟨4,7,4,0⟩	5 3	3	615	2001	⟨0,9,0,3⟩	⟨9,0,3,0⟩
2	⟨9,0,3,0⟩	4 3	7	492	4669	⟨0,7,4,5⟩	⟨7,4,5,0⟩
	⟨7,4,5,0⟩						⟨7,4,5,0⟩

and hence

$$\hat{r} = x \cdot y \cdot \rho \pmod N = 547,$$

the Montgomery representation of $r = x \cdot y \pmod N$ expected.

3) Conclusion:

Montgomery algorithm is to simplify the complexity of modular n. When n is a large number, modular n needs multiple addition, subtraction and multiplication operations. From the above process, the complexity of Montgomery reduction is indeed reduced, because there is only the shift operation of modulo r. However, in the first step, the Montgomery representation calculates the module n twice, $\hat{x} = x \cdot \rho\ (mod\ n)$, $\hat{y} = y \cdot \rho\ (mod\ n)$, the complexity seems not decreased. But in fact, the first step can be regarded as Montgomery's

precomputation. In the hardware implementation, if the precomputation is well calculated first, it will run much faster later. Especially when a large number of modular multiplication operations occur, precomputation can be carried out through parallel operation, which will greatly save running time.

4) Our codes in the optimized algorithm:

```
void mont_mul(uint32_t *res, uint32_t *a, uint32_t *b, uint32_t *n, uint32_t *prime)
{
    uint32_t t[128], m[64], mn[128], v[128];

    int i;

    Quickmul(t, a, b);                    // t = a * b
    Quickmul(m, t, prime);                // m = t * prime
    Quickmul(mn, m, n);                   // mn = m*n

    uint32_t ov = big_add(128, v, t, mn);  // v = t+mn

    for(i=0; i<64; i++){
        res[i] = v[i+64];                 // operate division by shifting digits
    }

    if(ov>0 || leq(64, n, u)){
        big_sub(64, u, u, n);             //caluculate mod, u = u-m
    }
}
```

Here is our code for Montgomery Algorithm for $a \times b$ in $R(r, n)$.

Firstly, we use the Quick multiplication to for the pre calculation,

$$t = a \cdot b,\ m = t \cdot n'(mod\, r),\ u = \frac{t+m \cdot n}{r}$$

Then we use big number add function to sum up the multiple results.

Finally, we have a judgment that if the dividend u is larger than the divisor n, if so, we output $u - n$, otherwise, we just return n.

And here are some related codes of Montgomery function:

```
uint32_t big_add(int count, uint32_t* res, uint32_t* x, uint32_t* y) //reduce root nummbers
{
    int i;
    uint32_t carry = 0;
    for (i = 0; i < count; i++) //
    {
        res[i] = x[i] + y[i];
        uint32_t carry1 = res[i] < x[i];        // check if res[i] < x[i]
        res[i] += carry;
        uint32_t carry2 = res[i] < carry;       // check if res[i] < carry
        carry = carry1 | carry2;                // if either carry1 or carry 2 = 1, carry = 1
    }
    return carry;
}
```

```
uint32_t big_sub(int count, uint32_t* res, uint32_t* x, uint32_t* y) //reduce root nummbers
{
    int i;
    uint32_t carry = 0;
    for (i = 0; i < count; i++)
    {
        uint32_t temp = x[i] - y[i];
        uint32_t carry1 = temp > x[i];          // check if temp > x[i]
        res[i] = temp - carry;
        uint32_t carry2 = res[i] > temp;        // check if res[i] > temp
        carry = carry1 | carry2;                // if either carry1 or carry 2 = 1, carry = 1
    }
    return carry;
}
```

```
void quick_mul(bn_t* result, bn_t* x, bn_t* y)   //
{
    int N;
    int xlength = length(x);
    int ylength = length(y);
    uint32_t digits, temp, b, d;
    uint32_t ydigits = bn_digits(x, digits);
    if (xlength > ylength)
        N = xlength;
    else
        N = ylength;
    if (N < 10)
        bn_mul(result, x, y, ydigits);

    N = (N / 2) + (N % 2);


    uint32_t multi = square(10, N);
    bn_div(b, x, multi, ydigits, temp, ydigits);
    uint32_t a = x - (b * multi);
    bn_div(d, y, multi, ydigits, temp, ydigits);
    uint32_t c = y - (d * N);
    uint32_t h = square(10, (N * 2));
    uint32_t g = square(10, N);
    result = (h * (b * d)) + (g * ((b * c) + (a * d))) + (c * a);

    // Clear potentially sensitive information
    memset((uint8_t*)multi, 0, sizeof(multi));
    memset((uint8_t*)a, 0, sizeof(a));
    memset((uint8_t*)b, 0, sizeof(b));
    memset((uint8_t*)c, 0, sizeof(c));
    memset((uint8_t*)d, 0, sizeof(d));
    memset((uint8_t*)g, 0, sizeof(g));
    memset((uint8_t*)h, 0, sizeof(h));
}
```

```
uint32_t square(uint32_t m, uint32_t n)
{
    int a[50] = { 1 };
    int i;

    int k = 0;
    for (i = 0; i < n; i++)
        k = 10 * k + a[i];
    return k;
}

int length(bn_t value)  //to get the length
{
    int counter = 0;
    while (value != 0)
    {
        counter++;
        value /= 10;
    }
    return counter;
}
```

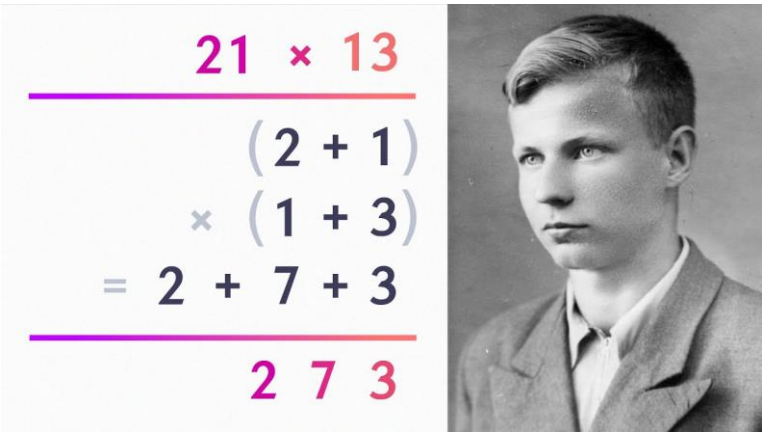
```
void mulhilo(uint32_t x, uint32_t y, uint32_t* hi, uint32_t* lo)
{
    uint64_t res = (uint64_t)x * (uint64_t)y;
    *lo = res;
    *hi = res >> 32;
}
```

```
uint32_t leq(int count, uint32_t* x, uint32_t* y)
{
    int i;
    for (i = count - 1; i >= 0; i--)
    {
        if (x[i] > y[i]) return 0;
        if (x[i] < y[i]) return 1;                    // check if x[i] > y[i]
    }
    return 1;
}
```

```
void modexp(uint32_t* res, uint32_t* base, uint32_t* exponent, uint32_t* m, uint32_t* m_prime, uint32_t* r_modp, uint32_t* r2_modp)
{
    int i, j;
    uint32_t base2[64];
    mmul(base2, base, r2_modp, m, m_prime);
    for (i = 0; i < 64; i++)
        res[i] = r_modp[i];

    for (i = 0; i < 64; i++)
    {
        uint32_t exp = exponent[i];
        for (j = 0; j < 32; j++)
        {
            if (exp & 0x1)
                mmul(res, res, base2, m, m_prime);
            mmul(base2, base2, base2, m, m_prime);
            exp >>= 1;
        }
    }

    uint32_t one[64];
    one[0] = 1;
    for (i = 1; i < 64; i++)
        one[i] = 0;

    mmul(res, res, one, m, m_prime);
}
```

But since it's hard to deal with several indexing problems occurred in the Montgomery function, we eventually didn't call it in our final version of codes.

3. Karatsuba Multiplication Algorithm:



1) Basic step:

The basic step of Karatsuba's algorithm is a formula that allows one to compute the product of two large numbers x and y using three multiplications of smaller numbers, each with about half as many digits as x or y, plus some additions and digit shifts. This basic step is, in fact, a generalization of a similar complex multiplication algorithm, where the imaginary unit i is replaced by a power of the base.

Let x and y be represented as n-digit strings in some base B. For any positive integer m less than n, one can write the two given numbers as:

$$x = x_1 B^m + x_0$$
$$y = y_1 B^m + y_0$$

where x_0 and y_0 are less than B^m. The product is then:

$$\begin{aligned} xy &= (x_1 B^m + x_0)(y_1 B^m + y_0) \\ &= z_2 B^{2m} + z_1 B^m + z_0 \end{aligned}$$

where

$$z_2 = x_1 y_1,$$
$$z_1 = x_1 y_0 + x_0 y_1,$$
$$z_0 = x_0 y_0.$$

These formulae require four multiplications and were known to Charles

Babbage. Karatsuba observed that xy can be computed in only three multiplications, at the cost of a few extra additions. With z_0 and z_2 as before one can observe that

$$z_1 = (x_1 + x_0)(y_1 + y_0) - z_2 - z_0.$$

An issue that occurs, however, when computing z_1 is that the above computation of $(x_1 + x_0)$ and $(y_1 + y_0)$ may result in overflow (will produce a result in the range $B^m \leq$ result $< 2B^m$), which require a multiplier having one extra bit. This can be avoided by noting that

$$z_1 = (x_0 - x_1)(y_1 - y_0) + z_2 + z_0.$$

This computation of $(x_0 - x_1)$ and $(y_1 - y_0)$ will produce a result in the range of $-B^m <$ result $< B^m$. This method may produce negative numbers, which require one extra bit to encode signedness, and would still require one extra bit for the multiplier. However, one way to avoid this is to record the sign and then use the absolute value of $(x_0 - x_1)$ and $(y_1 - y_0)$ to perform an unsigned multiplication, after which the result may be negated when both signs originally differed. Another advantage is that even though $(x_0 - x_1)(y_1 - y_0)$ may be negative, the final computation of z_1 only involves additions.

2) Example:

To compute the product of 12345 and 6789 , where $B = 10$, choose $m = 3$. We use m right shifts for decomposing the input operands using the resulting base $(B^m = 1000)$, as:

$$12345 = 12 \cdot 1000 + 345$$

$$6789 = 6 \cdot 1000 + 789$$

Only three multiplications, which operate on smaller integers, are used to compute three partial results:

$$z_2 = 12 \times 6 = 72$$

$$z_0 = 345 \times 789 = 272205$$

$$z_1 = (12 + 345) \times (6 + 789) - z_2 - z_0 = 357 \times 795 - 72 - 272205$$

$$= 283815 - 72 - 272205 = 11538$$

We get the result by just adding these three partial results, shifted accordingly (and then taking carries into account by decomposing these three inputs in base 1000 like for the input operands):

result $= z_2 \cdot (B^m)^2 + z_1 \cdot (B^m)^1 + z_0 \cdot (B^m)^0$, i.e.

result $= 72 \cdot 1000^2 + 11538 \cdot 1000 + 272205 = 83810205.$

Note that the intermediate third multiplication operates on an input domain which is less than two times larger than for the two first multiplications, its output domain is less than four times larger, and base- 1000 carries computed from the first two multiplications must be taken into account when computing these two subtractions.

3) Recursive application:

If n is four or more, the three multiplications in Karatsuba's basic step involve operands with fewer than n digits. Therefore, those products can be computed by recursive calls of the Karatsuba algorithm. The recursion can be applied until the numbers are so small that they can (or must) be computed directly.

In a computer with a full 32 -bit by 32 -bit multiplier, for example, one could choose $B = 2^{31} = 2147483648$, and store each digit as a separate 32 -bit binary word. Then the sums $x_1 + x_0$ and $y_1 + y_0$ will not need an extra binary word for storing the carry-over digit (as in carry-save adder), and the Karatsuba recursion can be applied until the numbers to multiply are only one-digit long.

4) Pseudocode and time complexity analysis:

Karatsuba's Pseudocode:

Procedure Karatsuba(num1, num2)

if (num1 < 10) or (num2 < 10)

return num1*num2 // Back to traditional Multiplication

m = max(size_base10(num1), size_base10(num2)) // Calculates the size of the numbers

m2 = m/2

high1, low1 = split_at(num1, m2) // Split the digit sequences about the middle

high2, low2 = split_at(num2, m2)

// split_at("12345", 3) will extract the 3 final digits, giving: high="12", low="345".

z0 = Karatsuba(low1,low2)

z1 = Karatsuba((low1+high1),(low2+high2))

z2 = Karatsuba(high1,high2)

return (z2*10^(2*m2))+((z1−z2−z0)*10^(m2))+(z0)

Karatsuba's basic step works for any base B and any m, but the recursive algorithm is most efficient when m is equal to $n/2$, rounded up. In particular, if n is 2^k, for some integer k, and the recursion stops only when n is 1, then the number of single-digit multiplications is 3^k, which is n^c where $c = \log_2 3$.

Since one can extend any inputs with zero digits until their length is a power of two, it follows that the number of elementary multiplications, for any n, is at most $3^{\lceil \log_2 n \rceil} \leq 3n^{\log_2 3}$.

Since the additions, subtractions, and digit shifts (multiplications by powers of B) in Karatsuba's basic step take time proportional to n, their cost becomes negligible as n increases. More precisely, if $t(n)$ denotes the total number of

elementary operations that the algorithm performs when multiplying two n-digit numbers, then

$$T(n) = 3T(\lceil n/2 \rceil) + cn + d$$

for some constants c and d. For this recurrence relation, the master theorem for divide-and-conquer recurrences gives the asymptotic bound $T(n) = \Theta\left(n^{\log_2 3}\right)$. It follows that, for sufficiently large n, Karatsuba's algorithm will perform fewer shifts and single-digit additions than longhand multiplication, even though its basic step uses more additions and shifts than the straightforward formula. For small values of n, however, the extra shift and add operations may make it run slower than the longhand method. The point of positive return depends on the computer platform and context. As a rule of thumb, Karatsuba's method is usually faster when the multiplicands are longer than $320 - 640$ bits.

5) Our codes in the optimized algorithm:

Firstly, we change the original multiplication function to call our Karatsuba fuction:

bn_mul(): to call Karatsuba main function:

```
void bn_mul(bn_t *a, bn_t *b, bn_t *c, uint32_t digits) //to use the karatsuba multiplication
{
    bn_t t[2 * BN_MAX_DIGITS] = {0};
    uint32_t bdigits, cdigits, i;

    bn_assign_zero(t, 2 * digits);
    bdigits = bn_digits(b, digits);
    cdigits = bn_digits(c, digits);

    bn_t tt[2 * BN_MAX_DIGITS] = {0};
    Karatsuba(tt, b, bdigits, c, cdigits);   //Karatsuba

    bn_assign(a, tt, 2 * digits);
    // Clear potentially sensitive information
    memset((uint8_t *)t, 0, sizeof(t));
}
```

Then, this is the main function of Karatsuba multiplication:

Karatsuba(): the main function to do the multiplication:

```
void Karatsuba(bn_t *re, bn_t *num1, uint32_t len1, bn_t *num2, uint32_t len2)   //karatsuba main function
{
    int begin1 = 0, begin2 = 0, n;
    int i;
    uint32_t digits = MAX(len1, len2);  //let digits be the larger length of num1 and num2
    if (digits == 0)    //if the length is 0, return
        return 0;
    else if (digits <= 32)  //if the length of the numbers we calculate is not big enough (bits), use normal multiplication
    {
        bn_mul1(re, num1, num2, digits);
        return;
    }

    //initialize and copy the nums into A, B, C, D:
    bn_t A[BN_MAX_DIGITS] = {0};
    bn_cut(A, num1, 0, digits / 2); //copy the front half part of num1 to A

    bn_t B[BN_MAX_DIGITS] = {0};
    bn_cut(B, num1, digits / 2, digits); //copy the last half part of num1 to B

    bn_t C[BN_MAX_DIGITS * 2 + 1] = {0};
    bn_cut(C, num2, 0, digits / 2); //copy the front half part of num2 to C

    bn_t D[BN_MAX_DIGITS / 2] = {0};
    bn_cut(D, num2, digits / 2, digits); //copy the front half part of num2 to D

    bn_t BD[BN_MAX_DIGITS * 2] = {0};
    Karatsuba(BD, B, digits - digits / 2, D, digits - digits / 2);   //use Karatsuba to calculate B*D

    bn_t AC[BN_MAX_DIGITS + 1] = {0};
    Karatsuba(AC, A, digits / 2, C, digits / 2);   //use Karatsuba to calculate A*C

    //get the length of A*C and B*D:
    uint32_t len_AC = bn_digits(AC, BN_MAX_DIGITS + 1);
    uint32_t len_BD = bn_digits(BD, BN_MAX_DIGITS + 1);

    bn_t AB[BN_MAX_DIGITS / 2 + 1] = {0};
    uint32_t lenAB = add(AB, A, digits / 2, B, digits - digits / 2);    //calculate A+B

    bn_t CD[BN_MAX_DIGITS / 2 + 1] = {0};
    uint32_t lenCD = add(CD, C, digits / 2, D, digits - digits / 2);    //calculate C+D

    // bn_t M0[BN_MAX_DIGITS] = {0};
    Karatsuba(A, AB, lenAB, CD, lenCD); // A : M0 = (A+B)*(C+D)
```

```
    // bn_t M1[BN_MAX_DIGITS] = {0};

    sub(B, A, bn_digits(A, BN_MAX_DIGITS), AC, len_AC); // B :M1 = (A+B)*(C+D) - AC

    // bn_t M2[BN_MAX_DIGITS] = {0};
    bn_assign_zero(C, BN_MAX_DIGITS);
    sub(C, B, bn_digits(B, BN_MAX_DIGITS), BD, len_BD); //C: (A+B)*(C+D) - AC - BD (= AD + BC)


    uint32_t len_BD_Patch = patch_zero(BD, bn_digits(BD, BN_MAX_DIGITS), digits / 2 * 2);  //patch zero to the bits and get the new length
    uint32_t len_M2_Patch;
    uint32_t lenC = bn_digits(C, BN_MAX_DIGITS);
    len_M2_Patch = patch_zero(C, lenC, digits / 2);    //patch zero to the bits and get the new length

    bn_t W0[2 * BN_MAX_DIGITS] = {0};
    uint32_t len_W0 = add(W0, BD, len_BD_Patch, C, len_M2_Patch);   //= (AD + BC) + BD

    uint32_t len_W1 = add(re, W0, len_W0, AC, len_AC);  //=AC + (AD + BC) + BD
}
```

In the main Karatsuba function, we use the following new functions to realize

it:

bn_mul1(): to do the normal (slower) multiplication, which is just the same as the original multiplication function:

```
void bn_mul1(bn_t *a, bn_t *b, bn_t *c, uint32_t digits)    //this is the original normal multiplication
{
    bn_t t[2 * BN_MAX_DIGITS];
    uint32_t bdigits, cdigits, i;

    bn_assign_zero(t, 2 * digits);
    bdigits = bn_digits(b, digits);
    cdigits = bn_digits(c, digits);

    for (i = 0; i < bdigits; i++)
    {
        t[i + cdigits] += bn_add_digit_mul(&t[i], &t[i], b[i], c, cdigits);
    }

    bn_assign(a, t, 2 * digits);
    // Clear potentially sensitive information
    memset((uint8_t *)t, 0, sizeof(t));
}
```

bn_cut(): to copy the big number from one to another:

```
void bn_cut(bn_t* re, bn_t* a, int s, int e)  // Mainly used in Karartsuba function
{
    memcpy(re, a + s, sizeof(bn_t) * (e - s)); // Define a pointer to store the copy content in bignum 'a'
}
```

add(): to do the new addition with carry (the original add function just return the carry, but this one calculate it and return the length of the result):

```
uint32_t add(bn_t *re, bn_t *num1, uint32_t len1, bn_t *num2, uint32_t len2)    //for calculate the adding with carry (the original one just return the carry,
                                                                                //this one return the length of the result)
{
    bn_t carry = 0;
    bn_t ai = 0;
    int i = 0;
    for (; i < MAX(len1, len2); i++)
    {

        if ((ai = num1[i] + carry) < carry)
        {
            ai = num2[i];
        }
        else if ((ai += num2[i]) < num2[i])
        {
            carry = 1;
        }
        else
        {
            carry = 0;
        }
        re[i] = ai;
    }
    if (carry)
    {
        re[i++] = carry;
    }
    return i;
}
```

sub(): to do the new subtraction with borrow (the original add function just return the borrow, but this one calculate it and return the length of the result):

```
uint32_t sub(bn_t *re, bn_t *num1, uint32_t len1, bn_t *num2, uint32_t len2)    //for calculate the subtracting with carry (the original one just return the borrow,
                                                                                //this one return the length of the result)
{
    bn_t ai, borrow;
    uint32_t i;

    borrow = 0;
    for (i = 0; i < MIN(len1, len2); i++)
    {
        if ((ai = num1[i] - borrow) > (BN_MAX_DIGIT - borrow))
        {
            ai = BN_MAX_DIGIT - num2[i];
        }
        else if ((ai -= num2[i]) > (BN_MAX_DIGIT - num2[i]))
        {
            borrow = 1;
        }
        else
        {
            borrow = 0;
        }
        re[i] = ai;
    }
    if (borrow > 0)
    {
        num1[i]--;
    }
    while (i < len1)
    {
        re[i] = num1[i];
        i++;
    }
    return i;
}
```

patch_zero(): to add 0s to the numbers because these numbers we are calculating are not at the same position:

```
uint32_t patch_zero(bn_t *num1, uint32_t len1, uint32_t zero)   //to invert and patch 0s before the num
{
    for (int i = len1 + zero - 1, j = 0; j < len1; j++, i--)
    {
        num1[i] = num1[len1 - j - 1];
    }
    memset(num1, 0, sizeof(bn_t) * zero);
    return len1 + zero;
}
```

4. Result Analysis:

Firstly, we wrote a **check()** function to check whether the calculated result (result of encryption and decryption) is correct:

```
void check(bn_t *a, bn_t *b, uint32_t digits)   //check whether it's correct
{
    for (int i = 0; i < digits; i++)
    {
        if (a[i] != b[i])
        {
            printf("error %d  %u %u\n", i, a[i], b[i]);
            // exit(0);
        }
    }
}
```

Then, we compare the running time before and after the optimization on an x86 computer:

Result before optimization:

```
root@p930026136:~/Downloads/RSA4096/RSA4096# ./release/main
RSA encryption decryption test is beginning!

key_e1:[255]: 00C23237AE137D1169E2B4A0128A857C93EE4ABF13B9434DD010A1728C4D94AF1E23916280394642494B9F6B504FF6C26ABD6F050B697C548A9380
D4DEAE50389B29F36F255E993CDEB25B1EB07BE11414AC7D6F9C025FAA6B416DF656D1AAFDBE4342AD7D0C7D79F45E161332E486D70F7656AE4563759C9BC996FEF8
C5FA5147A3C59C7AD51970AAE600DA8A26CFC841941C066720646B0D00EBD6837A62473B4563D00B4051BA81C10AD125631A4ACCA984FF31753CD3944707CC884892
D380D0660F81D42DA171D1D2A7C73A2FC1051C42550DCC38EE63982D8ED69EB0AFAACACB60D5487E58B8009117608236 8CF3E38359F99E224553B1C235D13323
key_e2:[256]: A25B2B1F2FD81A3789D18C5C32F74040B2858F605E9D6E24EACE08BBD9B44069BB067826CB86208240FD091FB0FAEC844F727A46007C50FD254E58
B0A80F5A53D67621713C74B29341AEF79E5BFBB6D4CB8BC455A703D5FBF90544FF15BD9A2BE8AC7D40929158A25A1DE5E6C91A561E23C8D377D527632493F3023AEA
B1003F97BF0C540A878F69B726AA41D091FF7FE370EEC1CBC5896FDAAA53616F68C416B7EC8005CED3823563F144927F2D44DEA109AC0A6FE1C28EF0F71A171EEF9A
F65C3AF07860ED011A63A0CE9FEA56DD241305B2A7AF99BB5B4E8BFC334499EB0F2753B5F313C9849833BBEDDBF652F37EEF7B3313CB4DF8E0F7F4D436B4ADEA97
rsa_public_encrypt_any_len Average time(s): 0.038972; rsa_private_decrypt_any_len Average time(s): 2.304360
Public Encrypt and private decrypt success!
```

Result after optimization:

```
root@p930026136:~/Downloads/rsa_final/rsa# ./release/main
RSA encryption decryption test is beginning!

key_e1:[255]: 00C23237AE137D1169E2B4A0128A857C93EE4ABF13B9434DD010A1728C4D94AF1E23916280394642494B9F6B504FF6C26ABD6F050B697C548A9380
D4DEAE50389B29F36F255E993CDEB25B1EB07BE11414AC7D6F9C025FAA6B416DF656D1AAFDBE4342AD7D0C7D79F45E161332E486D70F7656AE4563759C9BC996FEF8
C5FA5147A3C59C7AD51970AAE600DA8A26CFC841941C066720646B0D00EBD6837A62473B4563D00B4051BA81C10AD125631A4ACCA984FF31753CD3944707CC884892
D380D0660F81D42DA171D1D2A7C73A2FC1051C42550DCC38EE63982D8ED69EB0AFAACACB60D5487E58B8009117608236 8CF3E38359F99E224553B1C235D13323
key_e2:[256]: A25B2B1F2FD81A3789D18C5C32F74040B2858F605E9D6E24EACE08BBD9B44069BB067826CB86208240FD091FB0FAEC844F727A46007C50FD254E58
B0A80F5A53D67621713C74B29341AEF79E5BFBB6D4CB8BC455A703D5FBF90544FF15BD9A2BE8AC7D40929158A25A1DE5E6C91A561E23C8D377D527632493F3023AEA
B1003F97BF0C540A878F69B726AA41D091FF7FE370EEC1CBC5896FDAAA53616F68C416B7EC8005CED3823563F144927F2D44DEA109AC0A6FE1C28EF0F71A171EEF9A
F65C3AF07860ED011A63A0CE9FEA56DD241305B2A7AF99BB5B4E8BFC334499EB0F2753B5F313C9849833BBEDDBF652F37EEF7B3313CB4DF8E0F7F4D436B4ADEA97
rsa_public_encrypt_any_len Average time(s): 0.028294; rsa_private_decrypt_any_len Average time(s): 2.126049
Public Encrypt and private decrypt success!
```

We ran the algorithm for 20 times and calculated the average running time, and it can be found that the average **encryption time** is **0.028294 (within 30ms)** and the average **decryption time** is **2.126049**, which means that the encryption was improved **10ms**, and the decryption was improved **0.1~0.2s**.

5. Conclusion and Reflection:

To optimize the high-speed RSA cryptosystem on CPU, we worked on reducing the time complexity of the multiplication and modular multiplication, and used the Montgomery and Karatsuba algorithms to finally decrease the running time to

0.028294/2.126049s.

However, in our optimization process, we have also found some difficulties hindering us:

1) The given data is not big enough and does not take full advantage of Karatsuba big number multiplication, because Karatsuba multiplication only fits those big numbers;

   Besides, it's also difficult to decide the specific size of a "big" number.

2) Although the time complexity of Karatsuba is low but the recursion is used many times, so it is difficult to optimize.
3) The data structure is complicated to calculate, and multiple functions are needed to complete various calculations.
4) The indexing occurred in the Montgomery function is hard to deal with.

In the future researches, we will work hard on those problems to improve our optimization.

6. Reference:

Fang, X., & Li, L. (2007, November). *On Karatsuba multiplication algorithm*. In The First International Symposium on Data, Privacy, and E-Commerce (ISDPE 2007) (pp. 274-276). IEEE.

Chow, G. C., Eguro, K., Luk, W., & Leong, P. (2010, August). *A Karatsuba-based Montgomery multiplier*. In 2010 International Conference on Field Programmable Logic and Applications (pp. 434-437). IEEE.

Eyupoglu, C. (2015). *Performance analysis of Karatsuba multiplication algorithm for different bit lengths*. Procedia-Social and Behavioral Sciences, 195, 1860-1864.

Kwon, T. W., You, C. S., Heo, W. S., Kang, Y. K., & Choi, J. R. (2001, May). *Two*

implementation methods of a 1024-bit RSA crypto processor based on modified Montgomery algorithm. In ISCAS 2001. The 2001 IEEE International Symposium on Circuits and Systems (Cat. No. 01CH37196) (Vol. 4, pp. 650-653). IEEE.

Koc, C. K., Acar, T., & Kaliski, B. S. (1996). *Analyzing and comparing Montgomery multiplication algorithms*. IEEE micro, 16(3), 26-33.

Ha, J. C., & Moon, S. J. (1998). *A common-multiplicand method to the Montgomery algorithm for speeding up exponentiation*. Information processing letters, 66(2), 105-107.