Implement and Analyze Exponential Multiplication Algorithms Inside RSA system

Ke Ni

Abstract

In this report, We study different algorithms for modular exponentiation, in cryptographic algorithms and protocols. Since RSA system is one of the most common cryptographic systems, we select as well as implement RSA system and apply different algorithms for the two basic operations to it. We record real run time and analyze results of basic operation algorithms, including those designed for efficiency and others for security, Since the topic is chosen before homework 3 and homework 4, some parts of this report is included in the latter part of the class. We also include reports for further speeding up the multiplication by parallelization.

1 Modular Exponentiation

In the latter part of the course after choosing our topics, we happen to study a lot of modular exponentiation algorithms. Here I provide naive implementations in python as pseudocode in Section 5. In this section, I briefly introduce these algorithms in plain text.

brute-force It is a simple algorithm that we simply use a loop and do mod operation inside loop. The time complexity is $\mathcal{O}(n)$.

binary method In the implementation section 5, we provide left to right binary method implementation. This algorithm scan from the leftest or rightest end of binary representation of exponent, do square and multiplication based on bit values one at a time. Thus time complexity is $\mathcal{O}(\log n)$

the square-and-multiply-always algorithm One defect of binary method is that binary method is susceptible to power analysis attack. For power attack, the attacker analyzes power consumption of machines, so the attacker can then obtain exponent part because a ”0” bit lacks one multiplication in the loop of binary method compared with a ”1” bit case. The square-and-multiply-always algorithm avoids this by doing a dummy multiplication for ”0” bit. Other parts are the same as binary method. Time complexity is $\mathcal{O}(\log n)$.

the Montgomery powering ladder algorithm (Joye and Yen, 2003) provides sufficient backgrounds for understanding the Montgomery powering ladder algorithm. It is an algorithm without evaluating a relational expression and jumping to branch. It always square and multiply at each iteration, but the operations are not dummy. Overall, it is less susceptible to power analysis attack and fault attack. Time complexity is $\mathcal{O}(\log n)$.

the atomic square-and-multiply algorithm It is a variant of square-and-multiply. This method updates equal to or more than the number of bits of the exponent. Every iteration does a same set of operations. Time complexity is $\mathcal{O}$.

2 RSA System

(Rivest et al., 1978) invented RSA algorithm. RSA system consists of three parts: key generation, encryption and decryption. The procedure is not very complicated: find two large prime numbers, p and q, obtain their product, $n = p * q$, and then choose a random exponent $e < n$. Finally we obtain d, where $d^e = 1(\text{mod } \phi(n))$, where ϕ is Euler's totient function. Assume M is message and C is cipher, our encryption and decryption process is

$$E(M) = M^e(mod\, n) \tag{1}$$

$$D(C) = C^d(mod\, n) \tag{2}$$

In our experiments, we implement a simple RSA system to encode and decode messages. (Template implementation comes from a github gist post without any license issue.)

Method	time (s)	#call
original	4.246	50
binary	0.173	50
square-and-multiply-always	0.219	50
monPowerLadder	0.260	50
atomic-square-and-multiply	0.174	50
brute-force	>100	50

Table 1: Results of our experiments. ”Time” column is for time spent by the encryption and decryption function to process a message with 1000 characters.

3 Compare Modular Exponentiation Algorithm

Table 3 shows results on our implemented naive RSA system. The original exponentiation is a binary method which we thought was not efficient, so we rewrite it in our way. There are many other improvement we can make for our python implementation, but here it is enough for us to explore how modular exponentiation algorithms perform in the system. As we can see, the results are proportional to time complexity we analyzed. Binary method (left to right in our implementation) is the fastest one as we expected because it does fewer operations than others. Brute-force never stops during our experiments.

4 Parallelized Algorithms for RSA system

After comparing differences of different modular exponentiation algorithms, we would like to further improve the results. As we can see in profiling results of our experiments, modular exponent algorithm occupies most of computation resources (cpu time). So it is meaningful for us to further investigate algorithms to parallelize modular exponentiation part. (Fadhil and Younis, 2014) describes several possibilities for the whole RSA systems.(Emmart et al., 2016) introduces several algorithms for optimizing modular exponentiation part. Papers we studied can be categorized into two branches.

- Split data and do encryption on each data segment on different cores
- Directly parallelize modular exponentiation

Split Data We studied CUDA language to use NVIDIA GPU resources online and download as well as modify code provided by authors of (Fadhil and Younis, 2014). We got similar results (speedup factor=10) on our NVIDIA 1080TI graphics card. It is also intuitive that using multiple cores to encrypt messages and combine them is an easy but effective way to linearly reduce the total time.

Optimize Modular Exponentiation Unfortunately, we failed to obtain a good speed up factor for this branch. Our method can be summarized as following:

- Split work and then assign to GPU threads (assume n thread)
- Each thread computes $d^{e/n}$ (mod n) with a modular exponentiation method
- Return results to a main thread and then combine them together.

Our speed up factor is low for this method. We analyzed our method and find the reason is that binary methods already reduces the number of iterations to $\mathcal{O}(n)$, so divide work based on exponentiation is not effective. Meanwhile, overheads introduced by communication between threads covers the advantages of splitting work. However, the second branch can be promising. (Emmart et al., 2016) optimize modular exponentiation on NVIDIA graphics card. We did not further investigate their approach related to utilize special GPU structures and generate better assembly code for a great speedup for modular exponentiation. We put their paper here for readers who are interested.

References

Niall Emmart, Justin Luitjens, Charles Weems, and Cliff Woolley. 2016. Optimizing modular multiplication for nvidia’s maxwell gpus.

Heba Fadhil and Mohammed Younis. 2014. Parallelizing rsa algorithm on multicore cpu and gpu. Volume 87:15–22.

Marc Joye and Sung-Ming Yen. 2003. The montgomery powering ladder. In *Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, CHES ’02, pages 291–302, London, UK, UK. Springer-Verlag.

R. L. Rivest, A. Shamir, and L. Adleman. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126.

5 Appendix

5.1 Modular Exponential Python Simple Implementation

```
def bruteForce_modExp(m, d, n):
    temp = 1
    for _ in range(d):
        temp = (temp*m)%n
    return temp

def left2right_binary(m, d, n):
    r0 = 1
    d = [int(x) for x in bin(d)[2:]]
    k = len(d)
    for i in range(k):
        r0 = r0**2 % n
        if d[i] == 1:
            r0 = r0 * m % n
    return r0

def always_multiply(m, d, n):
    rs = [1, 1]
    d = [int(x) for x in bin(d)[2:]]
    k = len(d)
    for i in range(k):
        rs[0] = rs[0]**2 % n
        b = 1 - d[i]
        rs[b] = rs[b] * m % n;
    return rs[0]

def monPowerLadder(m, d, n):
    r = [1, m]
    d = [int(x) for x in bin(d)[2:]]
    k = len(d)
    for i in range(k):
        b = 1 - d[i]
        r[b] = r[0]*r[1] % n
        r[d[i]] = r[d[i]]**2 % n
    return r[0]

def atomic_square_and_multiply(m, d, n):
    r = [1, m]
    d = [int(x) for x in bin(d)[2:]]
    d.reverse()
    k = len(d)
    i = k-1
    b = 0
    while i >= 0:
        r[0] = r[0]*r[b] % n
        b = b ^ d[i]
        i = i - (not b)
    return r[0]
```