A Study and Implementation of RSA Cryptosystem

Implementing the well known Rivest Shamir Adleman Public Key Cryptosystem

Sinjan Chakraborty
Computer Science and Engineering Department
Jadavpur University
Kolkata, 700032 | WB IN
sinjanc@gmail.com

Vineet Kumar
Computer Science and Engineering Department
Jadavpur University
Kolkata, 700032 | WB IN
vntkumar8@gmail.com

ABSTRACT
This project involves an implementation of the Rivest Shamir Adleman (RSA) encryption algorithm in C. It consists of generation of two random prime numbers and a number coprime to \( \phi(n) \) also called Euler toitent function. These three are used to generate a public key and private key. The user has to enter a message which is encrypted by the public key. The algorithm also decrypts the generated cipher text with the help of the private key and returns the plain text message which was encrypted earlier.

General Terms
Plain Text: The general message needed to be encrypted is called plain text.
Cipher Text: The Garbage like looking string having no information about plain text is called cipher text.

Keywords
RSA, Public Key Cryptosystem, Asymmetric Key Cryptosystem, Cryptography, Security.

1. INTRODUCTION
This project has been done so that people interested in cryptography, especially students can get a better understanding of how generation of random primes, key generation and encryption and decryption process takes place in RSA. Being an undergrad students our search for an easy-to-understand C implementation of RSA that involved generation of random primes lead no where. On not finding a satisfactory implementation, we implemented it.

2. PUBLIC-KEY CRYPTOSYSTEMS
In a public key cryptosystem each user places in a public file an encryption procedure E. That is, the public file is a directory giving the encryption procedure of each user. The user keeps secret the details of his corresponding decryption procedure D. These procedures have the following four properties:

1. Deciphering the enciphered form of a message M yields M. Formally,
   \[ D(E(M)) = M \]
2. Both E and D are easy to compute
3. By publicly revealing E the user does not reveal an easy way to compute D. This means that in practice only he can decrypt messages encrypted with E, or compute D efficiently
4. If a message M is first deciphered and then enciphered, M is the result. Formally,
   \[ E(D(M)) = M \]

3. IMPLEMENTATION RSA ALGORITHM
For Implementation of RSA these four famous algorithms have been used:
- Sieve of Eratosthenes (for prime number generation)
- Fermat Primality Test
- Miller-Rabin Primality Test
- Encryption-Decryption Algorithms.

Proper discussion about each algorithm is mentioned in next subsections.
3.1 Sieve of Eratosthenes

The sieve of Eratosthenes is one of the most efficient ways to find all primes smaller than n when n is smaller than 10 million or so. Following is the algorithm to find all the prime numbers less than or equal to a given integer n by Eratosthenes method:

1. Create a list of consecutive integers from 2 to n : (2, 3, 4, ... , n).

2. Initially, let p equal 2, the first prime number.

3. Starting from p, count up in increments of p and mark each of these numbers greater than p itself in the list. These numbers will be 2p, 3p, 4p, etc.; note that some of them may have already been marked.

4. Find the first number greater than p in the list that is not marked. If there was no such number, stop. Otherwise, let p now equal this number (which is the next prime), and repeat from step 3. But When the algorithm terminates, all the numbers in the list that are not marked are prime.

3.2 Fermat’s Primality Test

If p is the number which we want to test for primality, then we could randomly choose a, such that $a < p$ and then calculate $a^{(p-1)} \mod p$. If the result is not 1, then by Fermat’s Little Theorem p cannot be prime. What if that is not the case? We can choose another a and then do the same test again. We could stop after some number of iterations and if the result is always 1 in each of them, then we can state with very high probability that our result is correct. You can notice that if the method returns composite, then the number is sure to be composite, otherwise it will be probably prime.

```c
int Fermat(long long p, int iteration){
if(p == 1){ /*1 isn’t prime; this conditional statement is not required for our given program. We write it for maintaining generality.*/
    return 0;
}
if(p<2){ //this conditional statement is not required for our given program.
    return 0;
}
if(p!=2 && p%2==0){
    return 0;
}
long long s=p-1;
while(s%2==0){
    s/=2;
}
int i=0;
for(i=0;i<iteration;i++){ // choose a random integer between 1 and p-1 (inclusive)
    long long a = rand()%(p-1)+1;
    long long mod=modulo(a,temp,p);
    while(temp!=p-1 && mod!=1 && mod!=p-1){
        mod=mulmod(mod,mod,p);
        temp *= 2;
    }
    if(mod!=p-1 && temp%2==0){
        return 0;
    }
}
return 1; // p is probably prime */
}
```

3.3 Miller-Rabin Primality Test

Let p be the given number which we have to test for primality. First we rewrite p-1 as $2^r s$ . Now we pick some a in range [1, n-1] and then check whether $a^s = 1 \mod p$ or $a^s \cdot 2^r = -1 \mod p$. If both of them fail, then p is definitely composite. Otherwise p is probably prime. We can choose another a and repeat the same test. We can stop after some fixed number of iterations and claim that either p is definitely composite, or it is probably prime. It can be shown that for any composite number p, at least (3/4) of the numbers less than p will witness p to be composite when chosen as a in the above test. Which means that if we do 1 iteration, probability that a composite number is returned as prime is (1/4). With k iterations the probability of test failing is $(1/4)^k$ or $4^{-k}$. This test is comparatively slower compared to Fermat’s test but it doesn’t break down for any specific composite numbers and 18-20 iterations is a quite good choice for most applications.

```c
int Miller(long long p, int iteration){
if(p<2){ //this conditional statement is not required for our given program.
    return 0;
}
if(p!=2 && p%2==0){
    return 0;
}
long long s=p-1;
while(s%2==0){
    s/=2;
}
int i=0;
for(i=0;i<iteration;i++){ // choose a random integer between 1 and p-1 (inclusive)
    long long a = rand()%(p-1)+1,temp=s;
    long long mod=modulo(a,temp,p);
    while(temp!=p-1 && mod!=1 && mod!=p-1){
        mod=mulmod(mod,mod,p);
        temp *= 2;
    }
    if(mod!=p-1 && temp%2==0){
        return 0;
    }
}
return 1;
}
```

3.4 RSA Algorithm

The algorithm of RSA consists of the following steps:

1. Select two prime nos $p \& q$ such as $p=q$
2. Calculate n as product of $p \& q$, i.e. $n=p*q$
3. Calculate m as product of $(p-1) \& (q-1)$ i.e. $m=(p-1)(q-1)$
4. Select any integer $e<m$ such that it is co-prime to m, co-prime means gcd(e,m) = 1.
5. Calculate d such that $(d*e) \mod m = 1$ , i.e. $d = e^{-1} \mod m$
6. The public key is $e,n$ The private key is $d,n$
if you have a text $P$, its encrypted version (cipher text $C$ is) $C = P^e \mod n$
To decrypt it back to plain text use $P = C^d \mod n$

4. IMPLEMENTATION
Successful Implementation of above discussed algorithms can be achieved in following steps.

4.1 Generating Two Large Primes
In the implementation of RSA, the first thing we have to do is generate two large prime numbers, $p$ and $q$. The standard way to generate big prime numbers is to take a preselected random number of the desired length, apply a Fermat Test (best with the base 2 as it can be optimized for speed) and then to apply a certain number of Miller-Rabin tests (depending on the length and the allowed error rate like $2^{-100}$) to get a number which is very probably a prime number. The pre-selection is done either by test divisions by small prime numbers (up to a few hundred) or by sieving out primes up to $10,000 - 1,000,000$ considering many prime candidates of the form $b + 2i(b$ big, $i$ upto a few thousands).

Here pre selection of a random prime by first creating an array of primes of size 10000 by the Sieve of Eratosthenes. Then generated a random number ‘gen’ with the help of rand() function such that the number is greater than 1000 and less than 10000. I have extracted the number at ‘gen’-th index position of the array of primes.

Now as the extracted number might not be a prime number, applied the Fermat Primality Test. Though Fermat is highly accurate in practice there are certain composite numbers $p$ known as Carmichael numbers for which all values of $a < p$ for which $gcd(a,p)=1$ $(a(p-1))$ modulo $p = 1$. If we apply Fermat’s test on a Carmichael number the probability of choosing an $a$ such that $gcd(a,p) = 1$ is very low (based on the nature of Carmichael numbers), and in that case, the Fermat’s test will return a wrong result with very high probability.

For this reason, applied Miller Rabin Primality Test on the number in case it satisfies Fermat test. If the number satisfies all the tests, we get a large number with a very high probability of being prime.

Now put the entire set of above operations in a loop such that if the number generated does not satisfy the Miller Rabin Primality Test, the above operations will be repeated again and this will go on until we generate a prime number. As needed to generate two prime numbers, we provide another loop outside this which runs two times and stores the generated primes in an array of size 2.

Table 1: Data Types Used in Implementation

| Variable Name | Data Type | Use                           |
|---------------|-----------|-------------------------------|
| p,q           | long long | Stores the two generated primes |
| n             | long long | Stores product of $p$ and $q$. |
| M             | long long | Stores the message entered    |
| phi           | long long | Stores value of $\phi(n)=(p-1)(q-1)$ |
| e             | long long | Stores the public exponent $e$ |
| d             | long long | Stores the private exponent $d$ |
| C             | long long | Stores the Cipher             |

4.2 Key Generation
Calculate $n$ as the product of $p \not\equiv q$, the two generated primes. The next thing to do is to generate a number $e$ such that $e < \phi(n)$ and $gcd(e, \phi(n)) = 1$. Here generated a random number less than $\phi(n)$ and checked their gcd. Continued the process until their gcd is 1. After generating $e$, calculated the value of $d$ such that $d = e^{-1} \mod \phi(n)$. The public key is {$(e,n)$}. The private key is {$(d,n)$}.

4.3 Encryption
Created a function encrypt() to encrypt a message with the help of public key {$(e,n)$}. The encrypted keyword $C$ is calculated by doing $C = M^e \mod n$.

void encrypt(){
    long long i;
    C = 1;
    for(i=0;i< e;i++)
        C=C*C%n;
    printf("\n\tEncrypted keyword : %lld",C);
}

4.4 Decryption
Created a function decrypt() to decrypt a cipher with the help of private key {$(d,n)$}. The decrypted message $M$ is calculated by doing $M = C^d \mod n$.

void decrypt(){
    long long i;
    M = 1;
    for(i=0;i< d;i++)
        M=M*C%n;
    printf("\n\tDecrypted keyword : %lld",M);
}

5. SIMPLE EXAMPLES
Example 1:

| . 62011 |
| 12269 |

F(n) phi value = 760738680
Public Key : {11723299,760812959}
Private Key : {288096259,760812959}

Enter The Plain Text : 5321
Encrypted keyword: 573183424

Enter the Cipher text: 573183424

Decrypted keyword: 5321

Example 2:

39703
66883

F(n) phi value = 2655349164
Public Key: {8068769, 2655455749}
Private Key: {149069429, 2655455749}
Enter The Plain Text: 5321
Encrypted keyword: 2521426694
Enter the Cipher text: 2521426694
Decrypted keyword: 5321

Comment: These two examples show how on entering the same message on two different runs, the value of the cipher text is different because of the generation of different prime numbers p and q and the different public and private exponents e and d, each time the program is run.

6. PROBLEMS IN IMPLEMENTATION

The main problem faced is that for a given message M, the cipher text C generated is large, nearly a 10 digit number. Also, the value of d, i.e., the private exponent is of 10 digits. To decrypt M, we have to calculate $M = C^d \mod n$. This calculation is taking a long time in some cases.

7. ANALYSIS

The time taken on computing is not fixed it changes in each & every run. After performing a statistical test of encrypting a plain text (which is 25000) & decrypting it time taken is noted

| Value of F(n) | Time (in sec.) |
|--------------|----------------|
| 1614044880   | 16.5           |
| 946335840    | 21.5           |
| 2186888896   | 60.0           |
| 1927086672   | 17.6           |
| 1595209116   | 18.8           |
| 1424163096   | 9.4            |
| 2917216608   | 28.2           |

Table 2: Computational Complexity (Data)

Plotting the data it is extremely clear that computational time of phi function, encryption & decryption of plain text & cipher text respectively is not predictable. Some runs of test with takes more time, some takes less time while complexity of computations remaining approximately fixed.

8. REFERENCES

[1] R.L. Rivest, A. Shamir, and L. Adleman, A Method for Obtaining Digital Signatures and Public-Key Cryptosystems, Comm. of ACM, 1983.
[2] W. Stallings, Cryptography and Network Security Principles and Practice, 5th ed. USA: Pearson Education, Inc, 2nd edition, 2011.
[3] Knuth, D. E., The Art of Computer Programming, Vol 2: Seminumerical Algorithms., Addison-Wesley, Reading, Mass., 1969.
[4] Diffie, W., and Hellman, M. New directions in cryptography. IEEE Trans. Inform. Theory IT-22, (Nov. 1976), 644-654. (Nov. 1976), 644-654.
[5] Diffie, W., and Hellman, M. New directions in cryptography. IEEE Trans. Inform. Theory IT-22, (Nov. 1976), 644-654. (Nov. 1976), 644-654.
[6] Diffie, W., and Hellman, M. Exhaustive cryptanalysis of the NBS data encryption standard, Computer 10, June 1977), 74-84.
[7] Miller, G.L. Riemann’s hypothesis and tests for primality. Proc. Seventh Annual ACM Symp. on the Theory of Computing, Albuquerque, New Mex., May 1975, pp. 234-239
[8] Goldwasser, S. and Bellare, M., Lecture Notes on Cryptography, MIT, July 2008.
[9] Rabin, M.O., Probabilistic algorithms. In Algorithms and Complexity, J. F. Traub, Ed., Academic Press, New York, 1976, pp. 21-40.