Implementation of RSA algorithm Using Mersenne Prime

Shilpa M Pund, Chitra G Desai
Ph.D Student, Department of Computer Science,
University of Pune. Maharashtra, India.
Professor and Head, MCA Department,
MIT Aurangabad, India.

ABSTRACT
Cryptographic algorithms are oftenly based on large prime numbers. It is a difficult task to generate large prime numbers and test it for primality. This requires overheads on first generating a large number which should be probably prime and second, testing its primality. This is because the generated large prime number using different algorithms may not necessarily pass the primality test. In order to generate an assured large prime number we can make use of Mersenne primes. Small primes are the used to generate large mersenne primes. This paper implements RSA based cryptographic algorithm using mersenne prime.

General Terms
Network security, RSA algorithm, Mersenne primes, Integer factorization.

Indexing terms
File encryption and decryption, Euclidean algorithm, Bézout’s Identity.

I. INTRODUCTION

Encryption is one of the principal means to grantee the security of sensitive information. It not only provides the mechanisms in information confidentiality, but is also functioned with digital signature, authentication, secret sub-keeping, system security and etc. Therefore, the purpose of adopting encryption techniques is to ensure the information's confidentiality, integrity and certainty, prevent information from tampering, forgery and counterfeiting [1]. The well-known and most widely used public key system is RSA, which was first proposed in a paper " A method for obtaining digital signatures and public-key cryptosystems" R.L. Rivest, A. Shamir and L. Adleman in 1976. It is the most influential public-key encryption algorithm, and has been recommended for ISO public key data encryption standard. RSA algorithm is an asymmetric cryptographic algorithm, meaning that the algorithm requires a key pair, one for encryption, another for decryption. Its security is based on the difficulty of the large number prime factorization, which is a well-known mathematical problem that has no effective solution [1]. This paper designed a complete and practical RSA encoding and decoding solution on *.txt file and provided the analysis on keys generated and the file size. The implementation is done in MuPAD 5.4 i.e. a Symbolic Math Toolbox provided by Matlab.

RSA algorithm can be simply described as
- <Generate keys>
  - Take the prime numbers p, q, compute n = p * q
  - Select an integer e which is coprime with (p-1) * (q-1)
  - Compute a private key d such that the equation : d * e = 1 (mod (p-1) * (q-1)).
  - Pair (e, n) as a public key
  - Pair (d, n) as the private key
- < Encryption and decryption>
  - b = a^e mod n, where a is plain text
  - a = b^d mod n, where b is cipher text.[2]

In RSA algorithm, it is recommended that each user must (privately) choose two large random prime numbers p and q to create encryption and decryption keys. These numbers must be large so that it is not computationally feasible for anyone to factor n = p * q. Also it recommends use of 100-digit (decimal) prime numbers p and q, so that n has 200 digits.[3]

II Mathematical Foundation For Public-key Cryptography
Integer factorization

Integer factorization is one of the most challenging problems of number theory and hardness of this problem is behind the security of RSA. In a public key cryptosystem, if one knows how to encrypt a message does not mean that it can be easily decrypted. So if anybody knows how a message is encrypted - that does not create any risk to the security of the system. The public key of RSA consists of n = p*q, where p and q are large, distinct positive primes. Another positive integer is e which is inverse modulo of Φ(n), where Φ(n) is Euler totient function. For encryption we need n and a public key e. For Decryption we need n and a private key d. Here in RSA we are interested in factoring n which has exactly two prime divisors p and q of approximately equal size (where p and q are very large numbers.) Due to its relationship with RSA, factoring has become one of the most important research problems. Hence the size of modulus in RSA algorithm determines how secure the RSA cryptosystem is.[4]

Number theory

Various number theory algorithms form the base of the cryptographic algorithms. The important and widely used modular arithmetic concepts and algorithms are:

Prime number: A prime number is defined as an integer greater than 1 which has no positive divisor other than 1 and the number itself.

Fundamental Theorem of Arithmetic: Fundamental Theorem of Arithmetic states that any natural number n can be written uniquely as a product of prime numbers.

e.g. 4200 = 2^3 * 3 * 5^2 * 7.

Modular arithmetic

Congruences

Three integers a, b, and m, we say that “a is congruent to b modulo m”. This relationship is written as \(a \equiv b \pmod{m}\).

Two integers are congruent modulo \(m\) if and only if their difference is divisible by \(m\). \(m\) is called the modulus of congruence.

Example: 11 ≡ 19 (mod 8).

Modular Reduction

Modular arithmetic is based on a fixed integer \(m > 1\) called the modulus. The fundamental operation is reduction modulo \(m\). To reduce an integer \(a\) modulo \(m\), one divides \(a\) by \(m\) and takes the remainder \(r\).

This operation is written as \(r := a \mod m\).

The remainder must satisfy \(0 \leq r < m\).

Examples:

\[3 := 15 \mod 6\]

\[8 := -3 \mod 11\]

Congruence modulo \(m\).

For fixed \(m\), each equivalence class with respect to congruence modulo \(m\) has one and only one representative between 0 and \(m-1\). The set of equivalent classes (called residue classes) will be denoted \(\mathbb{Z}/m\mathbb{Z}\). Any set of representatives for the residue classes is called a complete set of residue modulo \(m\) represented as \(\mathbb{Z}_m = \{0, 1, \ldots, m-1\}\).

One performs addition, subtraction, and multiplication on the set \(\mathbb{Z}_m\) by performing the corresponding integer operation and reducing the result modulo \(m\). [5]

Multiplicative Inverse

The multiplicative inverse of \(a\) modulo \(m\) is the integer \(b\) modulo \(m\) such that \(ab \equiv 1 \pmod{m}\). The multiplicative inverse of \(a\) is commonly written as \(a^{-1} \pmod{m}\). It exists and is unique if \(a\) is relatively prime to \(m\) and not otherwise.

If \(a\) and \(b\) are integers modulo \(m\), and \(a\) is relatively prime to \(m\), then the modular quotient \(ab\) modulo \(m\) is the integer \(ab^{-1} \pmod{m}\).

If \(c\) is the modular quotient, then \(c\) satisfies \(a \equiv bc \pmod{m}\). The process of finding the modular quotient is called modular division.
Euclidean algorithm

The Euclidean algorithm is an efficient method to compute the greatest common divisor (gcd) of two integers. We write gcd(a, b) = d to mean that d is the largest number that will divide both a and b. If gcd(a, b) = 1 then we can say that a and b are coprime or relatively prime. The gcd is sometimes called the highest common factor (hcf). [6]

Program: Developed in MuPAD 5.4 Environment for computing the greatest common divisor of two integers using Euclidean algorithm.

Input: Two non-negative integers a and b with a \( \geq \) b.

euclid_algo:= proc(a, b)
local r;
begin
while b>0 do
  r:= a mod b;
  a:= b;
  b:= r;
end_while:
print(a)
end:

Program Execution:
euclid_algo(15618427, 24137569)
1419857

Bézout’s Identity

In number theory, Bézout’s identity for two integers a, b is an expression ax + by = d, where x and y are integers called Bézout coefficients for (a,b) such that d is a common divisor of a and b. If d is the greatest common divisor of a and b then Bézout’s identity ax + by = gcd(a,b) can be solved using Extended Euclidean Algorithm.

Finding the Modular Inverse

The modular inverse of an integer e modulo n is defined as the number d such that ed \equiv 1 \mod n. We write d = e^{-1} \mod n or d = e^{-1} \mod n. The inverse exists if and only if gcd(n,e)=1. To find this value for large numbers we use the extended Euclidean algorithm, but there are simpler methods for smaller numbers.

Extended Euclidean Algorithm

The Extended Euclidean Algorithm is an extension to the Euclidean algorithm. Besides finding the greatest common divisor of integers a and b, as the Euclidean algorithm does, it also finds integers x and y (one of which is typically negative) that satisfy Bézout’s identity ax + by = gcd(a,b).

The Extended Euclidean Algorithm is particularly useful when a and b are coprime, since x is the multiplicative inverse of a modulo b, and y is the multiplicative inverse of b modulo a.

Program: Developed in MuPAD 5.4 Environment for Extended Euclidean algorithm.
Input: Two non-negative integers a and b.

Output d = gcd(a, b) and integers x and y satisfying ax + by = d.

```
exe2 := proc(a, b)
    local x, y, x1, y1, x2, y2, q, r, d, z;
    begin
        s := a;
        z := 0;
        print(s);
        if b = 0 then
            d := a;
            x := 1;
            y := 0;
            print(d, x, y);
        end_if;
        if b > a then
            temp := a;
            a := b;
            b := temp;
            print(a, b);
        end_if;
        x2 := 1;
        x1 := 0;
        y2 := 0;
        y1 := 1;
        while b > 0 do
            q := a div b;
            r := a - q*b;
            x := x2 - q*x1;
            y := y2 - q*y1;
            a := b;
            b := r;
            x2 := x1;
            x1 := x;
            y2 := y1;
            y1 := y;
        end_while;
        d := a;
        x := x2;
        y := y2;
        print("y =", y);
        if y < 0 then
            z := s + y;
        end_if;
    end
```
else
    \( z := y; \)
end_if;
print("gcd=", d);
print("Coefficient of a", x);
print("Coefficient of b", y);
print("The modular inverse is ", z);
end_proc:

Program Execution:
ex2(421, 111)
"y=" 110
"gcd=", 1
"Coefficient of a", -29
"Coefficient of b", 110
"The modular inverse is ", 110

ex2(5528047242541452596898475889286765965333928210300982447450712187201737523708865354024517010066
74489603060853772958959718200642118661364269352821349424656272877344066915948527395270316850869112336308
651610057445561707661001273512300337346977128787080931756865334031267832727830621451930401468985858018407655
5523047633228249323799801055586372593406587276191699794962363332730981113631273063009365832798279213992117718
42545804962254759253779831487584337626336222095380186157592038559294096490095916168391423623098650806390606
98305823011632172623869700472836, 429392673093

"y=", 25795948602209047438681166738097793317236303111761641341231100866295222073248022803685199813158947995430511453
902306035500248604202458556382107618069865938111301820856059913121443594108724151018149921669271017392882246
6295022066797373273625107082431244950644411121457552771135092355081265399752677470578236686505980192575722265
6175883535367664097621146491855273916932898013190336655373391921802415276762010430074884112883110958681249
13764284817441772550981026723876698167195896714732959075329282443982086946925798403387366333770291314068778407
3317438406667107645092265071

gcd = 1
"Coefficient of a", 20037044065
"Coefficient of b"

25795948602209047438681166738097793317236303111761641341231100866295222073248022803685199813158947995430511453
902306035500248604202458556382107618069865938111301820856059913121443594108724151018149921669271017392882246
6295022066797373273625107082431244950644411121457552771135092355081265399752677470578236686505980192575722265
6175883535367664097621146491855273916932898013190336655373391921802415276762010430074884112883110958681249
13764284817441772550981026723876698167195896714732959075329282443982086946925798403387366333770291314068778407
3317438406667107645092265071

"The modular inverse is ", 294845335502615358083826928588068625720507347967161567199773815228508623994553551888480312987174676015995
09372457307489085393459644260009424356460341471683312109567374612963097486125856425216763389420105970204269
471070224893734337838721664298690992424053301757250540404551441048045517873210394413614756500335259991500833257
4300439728816597021804094438706610489406329395569012327079959341176311215969360149050646339571425681012850305
44285337373015076524733834911949956521502505708761684505577105087313958990218435754999576094789237921423250
5573225496516224608207765.

III. Mersenne Prime Numbers
In mathematics, a Mersenne number, named after Marin Mersenne (a French monk who began the study of these numbers in early 17th century), is a positive integer that is one less than a power of two. Mersenne Primes are of the form \(2^p - 1\), where \(p\) is itself prime represented as \(M_p = 2^p - 1\). Since 1997, all newly-found Mersenne primes have been discovered by the "Great Internet Mersenne Prime Search" (GIMPS), a distributed computing project on the Internet. The search was on when it was noticed that most of the early primes worked, but \(2^{11} - 1\) was not prime. The great Mersenne Prime race has been in progress now for over 600 years and shows no sign of ending. Some of the more reasonably sized prime numbers which generate Mersenne primes are given in this list:

\[2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, 127, 521, 607, 1279, 2203, 2281, 3217, 4253, 4423, 9689, 9941, 11213, 19937, 21701, 23209, 44497, 86243, 110503, \ldots \]

As of: Date: 2012/06/12 05:06:11 the last Mersenne prime shown above was the largest known prime. \(2^{33112609} - 1\).

So we can generate very large primes by using \(M_p = 2^p - 1\) where \(M_p\) itself is a Mersenne Prime.

IV. RSA Algorithm Using Mersenne Primes

Declarations:

a) Ptext.txt : Plain text (source) file to be encrypted.

b) Ctext.txt : Cipher text file to be decrypted.

c) Array L : Used to store the contents of Ptext.txt for encryption.

d) Array S : Used to store the contents of Ctext.txt for decryption.

Algorithm:

1. Choose two large Prime numbers \(p\), \(q\).

2. Generate two very large Mersenne prime numbers as: \(m = 2^p - 1\) and \(n = 2^q - 1\).

3. Calculate \(c = m \times n\).

4. Calculate the value of \(\Phi\) using the formula: \(\Phi(c) = (m-1) \times (n-1)\).

5. Generate the public key '\(e\)' such that it is coprime with \(\Phi(c)\).

6. Find the value of private key '\(d\)' such that \((d \times e) \equiv 1 \mod \Phi(c)\).

7. Read plaintext in the form of binary data from the file Ptext.txt, store it in an array (L).

8. Perform \(L^e \mod c\) (on each element of array L) to get cipher text and store it in the file Ctext.txt.

9. For decryption, read the file Ctext.txt, store it in an array (S).

10. Perform \(S^d \mod c\) (on each element of array S) to get a plain text.

V. Output Details

1. Input: \(p=127,q=2203\)

"The value of \(m\) is", 170141183460469231736187303715884105727 (39-digits)

"The value of \(n\) is" 147597991521785209934866084717257940822166780976702240119928021704749484747247247420110823560848507250

2420519452587542875349976758587267022963962575212637477897785501552646522609988669914013540483809865866125041

9479686679771007. (658 digits)

"The value of \(c\) is" 25112496953842366118263729475441408070278227765779090306220370493691927159579420559912685511835941439481523

894432472725127248132399702332365759117303701682734203804753649533992054202779926589484425196730875232743

563759287213541918068577672095312529225040435511767431727662932474456642327811074164749365717415999751993075

8366681282133873258428509964283584075175337435809018404885672421744374995263624528630813074529935285423

38191057567544692158111898135787578979647338149471629231811440438483613303149751102810234559977956137910156
VI. Implementation Details

Initially, we the algorithms were implemented using MatLab. But as it does not support large integer for computations, MuPAD has used. MuPAD can compute big numbers efficiently. The length of a number that can be computed is limited only by the available main storage [8] . So, MuPAD has provided preferable environment to develop this program. The algorithmic steps are as follows.

1. Choose two large prime numbers p and q.

2. Generate two very large mersenne prime numbers using the formula \( m = 2^p - 1 \) and \( n = 2^q - 1 \). As, a prime can generate a large mersenne using the above formula.

3. Calculate the value of \( c = m \times n \).

4. Calculate the value of \( \Phi(c) \), Euler Totient function using the formula: \( \Phi(c) = (m-1)(n-1) \).

5. Generate the public key \( 'e' \) randomly using random function such that it is coprime with \( \Phi(c) \). Using gcd function based on Euclid's Algorithm. i.e. \( \text{gcd}(e, \Phi(c)) = 1 \).

6. Find the value of private key \( 'd' \) by implementing Extended Euclidian Algorithm.

7. For Encryption, use readbytes function which reads plaintext and converts it into binary data ,store it in an array. \( L \). Permissible values for binary data are Bytes, SignedBytes, Short, SignedShort, Float and Double.

8. Perform \( L \mod c \) (on each element of an array \( L \)) by using powermod\((L,e,c)\) function provided by MuPAD. Map each element of array \( L \) with the powermod value using map function.

9. Open a new file say ctext.txt in write mode using fopen function.

10. Now \( L \) contains cipher text, write array \( L \) in the file ctext.txt using write function.

11. For decryption, read the file ctext.txt by using read function , store it in an array \( S \).

12. Perform \( S^d \mod c \) (on each element of an array \( S \)) by using powermod\((L,e,c)\) function. Map each element of array \( S \) with the powermod value using map function.

13. Open a new file say ptext.txt in write mode using fopen function.

14. Use writebytes function to convert binary data into plain text and write it in the file ptext.txt. The above algorithm works for the mersenne primes greater than 17.

VII. Comparison
The size of the encrypted file depends on the size of keys generated, which further depends on the values of p and q. As we increase the values of p and q, the size of the encrypted file also increases. As the size of private key increases the execution time and memory requirement are also increased.

Table 1. Comparison of the original, encrypted and decrypted file sizes with respect to the value of p and q.

<table>
<thead>
<tr>
<th>Value of p</th>
<th>Value of q</th>
<th>Private Key Size (In digits)</th>
<th>File name</th>
<th>Original file size</th>
<th>Encrypted File Size</th>
<th>Decrypted File size</th>
<th>Execution Time</th>
<th>Memory Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>521</td>
<td>607</td>
<td>340</td>
<td>conversion.txt</td>
<td>3.14KB</td>
<td>1.08 MB</td>
<td>3.14KB</td>
<td>13 sec</td>
<td>7MB</td>
</tr>
<tr>
<td>127</td>
<td>521</td>
<td>195</td>
<td>conversion.txt</td>
<td>3.14KB</td>
<td>644KB</td>
<td>3.14KB</td>
<td>3 sec</td>
<td>8MB</td>
</tr>
<tr>
<td>127</td>
<td>2203</td>
<td>689</td>
<td>modrsa.txt</td>
<td>2.27KB</td>
<td>1.62 MB</td>
<td>2.27KB</td>
<td>69 sec</td>
<td>8 MB</td>
</tr>
<tr>
<td>607</td>
<td>1279</td>
<td>568</td>
<td>rsa.txt</td>
<td>1.01KB</td>
<td>599KB</td>
<td>2KB</td>
<td>17 sec</td>
<td>7 MB</td>
</tr>
<tr>
<td>3217</td>
<td>4253</td>
<td>2249</td>
<td>rsa.txt</td>
<td>1.01KB</td>
<td>230MB</td>
<td>1.01KB</td>
<td>823 sec</td>
<td>7 MB</td>
</tr>
<tr>
<td>4253</td>
<td>4423</td>
<td>2562</td>
<td>rsa.txt</td>
<td>1.01KB</td>
<td>2.68 MB</td>
<td>1.01KB</td>
<td>1220 sec</td>
<td>8 MB</td>
</tr>
<tr>
<td>9689</td>
<td>9941</td>
<td>5910</td>
<td>T1.txt</td>
<td>27 Bytes</td>
<td>161 KB</td>
<td>27 Bytes</td>
<td>262 sec</td>
<td>7 MB</td>
</tr>
<tr>
<td>9689</td>
<td>9941</td>
<td>5909</td>
<td>x25.txt</td>
<td>172 Bytes</td>
<td>1 MB</td>
<td>172 Bytes</td>
<td>1656 sec</td>
<td>7 MB</td>
</tr>
<tr>
<td>11213</td>
<td>19937</td>
<td>9378</td>
<td>T1.txt</td>
<td>27 Bytes</td>
<td>256 KB</td>
<td>27 Bytes</td>
<td>879 sec</td>
<td>6 MB</td>
</tr>
</tbody>
</table>

VIII. Conclusion
In this paper we have implemented RSA algorithm using mersenne primes. The real challenge in case of RSA is the selection of public key and generation of private key. Here public key is generated randomly. The use of mersenne prime has been made to generate large primes to enhance the security. This algorithm can generate a private key in the range of 15 to 5000 digits. We have executed this program for the primes greater than 9000 on Intel(R) Core i5-2410 M CPU with 6144 MB RAMS, it generates the private and public key. The system is able to do encryption and decryption for the file size up to 180 bytes. If the file size is too large and the primes are beyond 9000 then MuPAD does not respond because of the main storage limitation. The above algorithm will not give proper result for the primes less than 17.

The future work intends to reduce the size of encrypted file which is the limitation of the present system.

Acknowledgment
First author would like to thank Dr. Nivedita H Mahajan for her valuable guidance in Mathematical concepts.

References
[1] Xin Zhou and Xiaofei Tang, ”Research and Implementation of RSA Algorithm for Encryption and Decryption” 2011 The 6th International Forum on Strategic Technology. IEEE.
[2] Suli Wang, Ganlai Liu, ”File encryption and decryption system based on RSA algorithm” 2011 International Conference on Computational and Information Sciences. IEEE.

