IMPLEMENTATION DESIGN FOR FASTER ENCRYPTION IN RSA

Suhrid Barthakur, Partha Pritam Paul, Jasbir Singh Birdi, Anirban Chatterjee
Department of Computer Science and Engineering, NIT Silchar

***Abstract*—This paper proposes an implementation design/procedure for faster encryption of the RSA[4]. It proceeds through optimizing the modular exponentiation for large key of sizes 1024 bits or more. The paper uses a recoded scheme and montgomery multiplication for reducing the total number of operations and faster modular multiplication respectively. The design incorporates parallel processing by two processor for achieving optimal performance. In addition, a Systolic array design provides potentially higher throughput for a large number of computations.The project is implemented using JAVA programming language and has been built in Eclipse Neon IDE.**

Keywords: Parallel processing, Systolic arrays, Montgomery reduction.

I. Introduction

Encryption guarantees us the security of sensitive information and much more. It provides subtle mechanisms for information confidentiality, functioned with digital signature, authentication, secret sub-keeping, system security and to prevent information from tampering, forgery and counterfeiting.

At present, the best known and most widely used public key system is the RSA[4]. It was first proposed in a paper named “A method for obtaining digital signatures and public-key cryptosystems” by RL Rivest et al. in 1978.

It is an asymmetric (public) key cryptosystem based on number theory, which is basically a block cipher system. It’s security is based on the difficulty of factorizing a large prime number, which is a well-known mathematical problem that has no effective solution. RSA is one of the most widely used public key cryptography in encryption and digital signature standards.

The RSA algorithm is summarized as below[4]:

1) Choose two large random primes p and q.
2) Generate: N=p$\times$q.
3) Calculate Euler’s value: $\phi(N) = (p-1) \times (q-1)$.
4) Determine the value of random large integer E such that, $GCD(E, \phi(N)) = 1$.
5) Determine ’D’ as the multiplicative inverse of $Emod(\phi(N))$.
6) Generate public key (E,N) and publish it, and generate private key (D,N).

$$Ciphertext : C = M^E(modN) \tag{1}$$

This paper proposes a design for faster encryption implementation, determined by the computation speed of performing the modular exponentiation $C = M^E(modN)$, where E and N are very large number of the public key pair (E,N). For security, it depends on large size keys which are hard to factorize, and the integrity of the RSA system depends solely on factorization of N, product of two large prime numbers.

One of the well known algorithm for computing M^E is exponentiation by repeated squaring and multiplication, called the binary algorithm. Computing $M^E(modN)$ by this algorithm requires $2(logE)$ modulo multiplications in the worst case, where N is the product of two large prime numbers. The proposed algorithm achieves better results over the binary algorithm and makes encryption faster and efficient, through reduction of total number of modular multiplications and making each of the modular multiplications faster than the usual approach by montgomery modular multiplication.

II. Related Work

Our method of the faster encryption in RSA is based upon some of the previous works and experiments done on RSA to improve the basic encryption implementation techniques through faster modular exponentiation. Some of them are enlisted here.

D.E Knuth in his book “Semi numerical Algorithm”[2] suggested the binary algorithm which exponentiation by repeated squaring and multiplication. This is the most simple and traditional algorithm and is inefcient for large key sizes.

Chang N. Zhang and Herold L. Martin[3] suggested a parallel algorithmic design based on the binary algorithm in their paper for fast modular exponentiation which made use of parallel processing designs to make the binary algorithm faster. The suggested algorithm here, makes use of these techniques to further optimize the modulo multiplications with Montgomery reduction algorithm.

The Montgomery multiplication is used for faster modular exponentiation[8]. The well known algorithm for computing M^E is exponentiation by repeated squaring and multiplication and called the binary algorithm.

Hung-Min Sun et al.[5] proposed dual RSA algorithm and also analyzed the security of the algorithm. They presented new variants of RSA whose key generation algorithms output two distinct RSA key pairs having the same public and private exponent’s, two applications for Dual RSA were blind signatures and authentication. The security of Dual RSA was raised in comparison to RSA when there were values of E and D is small. The main disadvantage of using dual RSA was that the computational complexity of the generation of key algorithms is increased.

H. C. Williams[6] modified RSA public-key encryption algorithm. His opinion is, if the encrypting message procedure was broken into a certain operations than remainder used as modulus could be factored after few more operations. This technique was in similar appearance to RSA. The main limitation of this scheme was that very large prime numbers were used and generated mathematical errors were observed.

Prabhat K. Panda et. al.[7] presented a new Hybrid security algorithm for RSA cryptosystems named as Hybrid RSA(HRSA). This scheme used four prime numbers which increases the key generation time and consequently increases the system breaking time. The main disadvantage with this is, it has been achieved at expense of increased time-complexity.

Gaurav Shrivastava proposes a new approach to enhance the security of cryptosystem. The Data Encryption Standard (DES) is the most common Secret Key Cryptography scheme. DES so far has been stronger than other cryptosystems in the security.DES may be attacked by parallel processing. If you want to protect DES encryption system strong you need to follow this approach .In approach they will use Triple DES Three Times with RSA Algorithm. This will provide 504 bit key length. This new algorithm enhance the security level but also responsible for increase in the file size[**?**].

Naofumi Homma et. al applied Simple power analysis (SPA) to an RSA processor with a high-radix Montgomery multiplier on an FPGA platform and also applied an active attack where input data was set to a specific pattern to control the modular multiplication. The power dissipation for the multiplication was greatly reduced in comparison with modular squaring, resulting in success in revealing all of the secret key bits.

Majid Ahmadi et. al presents GPU implementation of the Montgomery multiplication algorithm that is heavily optimized for the GPU's SEVID architecture, as well as the field sizes and constraints required for elliptic curve cryptography.proposed algorithm's measured throughput in multiplication operations per second is 1.24 to 1.72 times greater than the next fastest GPU-based algorithm running on the same device, and is significantly greater than all other published CPU and GPU-based implementations.But the proposed work has some limitation as it leads to hardware complexity in implementation.

Arpan Mondal et. al proposed an efficient Montgomery modular multiplier design that exploits the efficiency of inbuilt multiplier and adder soft-cores of DSP blocks.It has some limitations as this multiplier computes Z in a single clock cycle only. and although the multiplier has the highest throughput also it consumes the highest amount of area resources.

S.S.Ghoreishi et.al proposed new hardware architecture for optimum implementation of rsa algorithm. According to the design, for encrypting an n-bit plaintext, About 3/4n (n + 11) clock cycles is needed. They used Xilinx VirtexII and XC4000 series FPGAs (Field Programmable Gate Array). As a result, it is shown that the processor can perform 1024-bit RSA operation in less than 15ms and 50ms at 54.6MHz and 16.1MHz on Xilinx VirtexII and XC4000 series FPGA, respectively.

III. Proposed Work

The proposed implementation design makes the RSA cryptosystem perform faster by fast encryption and decryption process. The encryption is a slow process when the key size is large like 1024 bits, 2048 bits or greater due to computation of modular exponentiation C = $M^E(modN)$, and also computations becomes expensive for large public key (E,N). Thereby the design makes the modular exponentiation fast by making use of Montgomery multiplication technique and parallel processing. Also a suitable systolic array design is proposed[2]. The Montgomery multiplication is a technique used for fast modular exponentiation with a little expense of pre and post computation it simplies the modular reduction steps, specically with two benets:

• It avoids possible mis-estimation of the quotient, which in classical algorithms leads to a special case; this is benecial from the standpoint of side channel leakage (e.g. timing attacks); and also avoiding the sophistication's of the classical quotient estimation and its special case that are hard to get right and test fully.

• The Montgomery equivalent of quotient estimation is performed based on the low-order bits of the value to be reduced (rather than high-order), and that eases implementation of multiplication and modular reduction interleaved in the same scan of a temporary result (that interleaving technique in turn limits the width of numbers manipulated to about the size of the modulus, and reduces the number of memory accesses, compared to naively computing a full product then reducing it).

IV. Modular Exponentiation Algorithm

A. Recoding

The proposed technique in order to compute C= M^E(mod N) involves representing E in binary form i.e

$$E = \sum_{i=0}^{n-1} e_i 2^i$$

,where n is number of bits in E and $e_i \in \{0,1\}$ similar to binary algorithm[9].

Now it known that the binary algorithm requires n + E1 total operations, where E1 is no of 1's representation in binary representation of E and n is the number of bits in E. Thus we need to reduce the value of E1 in order to make it faster.

To achieve this define a redundant expression of E say E'(E= E') which is given as:

$$E' = \sum_{i=0}^{n-1} e_i^* 2^i$$

where $e_i^* \in \{0,1,-1\}$ and is a redundant representation of E in base of 0 ,1 ,-1.

Let us represent -1 in the redundant representation as $\overline{1}$.

For example E=11110=1000$\bar{1}$0=100000-10 Thus as in the above example used reduce the number of non zero digit with the help of the redundant expression. For any given integer E, E¡2^n, the time complexity of computing M^E(mod M) is $(n + |\frac{n}{2}| + 1)$ Tn by applying the recording algorithm and redundant binary algorithm.

Suppose E is the integer randomly distributed in the range of 2^{n-1} and $2^n - 1$. It is easy to figure out that the average number of 1's for the integers between 2^{n-1} and $2^n - 1$ is $(\frac{n}{2} + 1)$.

The average number[14] of non-zero's after applying the recoding algorithm for the integers distributive in the range of 2^{n-1} and $2^n - 1$ can be calculated by the following equation:

$$n_2 = \frac{\sum_{i=0}^{2^{n-1}-1} f(i)}{2^{n-1}}$$

Where f(i) is the number of non-zero's after execution the recoding algorithm to the integer $2^{n-1} + i(i = 0...2^{n-1} - 1)$. Table 1 shows the comparison of the number of non-zero elements between the original integer and the recoded one.

Table I: Comparison of Number of Non Zero's Between Binary Representation and Redundant Representation.

n number of bits	no. of 1's before recoding n_1	no. of non-zeroes after recoding n_2	$\frac{n_1 - n_2}{n_1}$
10	6	4.44	26.0
15	8.5	6.11	28.1
20	11	7.77	29.4
25	13.5	9.44	30.8

Formally it can be proved that the ratio of $\frac{(n_1-n_2)}{n_2}$ is greater than $\frac{1}{3}$ when $n \geq 25$. Notice that in practice, E is very large, and one may need to compute a series of computations of $M_i^E(modN)$, i=1, 2...m. Therefore, the total saving is $m * (n_1 - n_2)$ on average[12].

B. Montgomery Multiplication

Now in order to perform a modular exponentiation we use the montgomery multiplication .Montgomery modular multiplication[11], more commonly referred to as Montgomery multiplication, is a method for performing fast modular multiplication.For computations using montgomery multiplication , first perform modular exponentiation in montgomery form ,and then convert back the obtained result out of montgomery form to obtain the required result i.e

$$C = M^E modN$$

For montgomery multiplication it is defined as, given an integer $a < N$, where N is the k-bit modulus, A is said to be its N-residue with respect to R if ,

$$A = a * R(modN), whereR = 2^k$$

Likewise, given an integer $b < N$, B is said to be its N-residue with respect to r if ,

$$B = b * R(modN)$$

Ordinary Domain	$\Longleftrightarrow$	Montgomery Domain
X	$\leftrightarrow$	$X' = X \cdot 2^n \pmod M$
Y	$\leftrightarrow$	$Y' = Y \cdot 2^n \pmod M$
XY	$\leftrightarrow$	$(X \cdot Y)' = X \cdot Y \cdot 2^n \pmod M$

Figure 1: Mapping to montgomery form.

The Montgomery product of A and B can then be defined as:

$$Z = A * B * R'(modN),$$

where R' is the inverse of R modulo N. i.e $RR' \equiv 1(modN)$.

Since $GCD(N, R') = 1$,extended Euclidean algorithm can efficiently determine integers R' and N' that satisfy Bézout's identity:

$$0 < R' < N, 0 < N' < RandRR' - NN' = 1.$$

The montgomery reduction algorithm[13] is based on the fact that the computation of $A*B*R'modN$ can be done very efficiently by the montgomery reduction algorithm REDUCE as shown below .

```
1:  function REDUCE(X)
2:      begin
3:          $q \Leftarrow (XmodR)N'modR$;
4:          $a \Leftarrow (X + qN)/R$;
5:          if $a > N$ then
6:              $a \Leftarrow a - N$
7:          end if
8:          return a;
9:      end
10: end function
```

This reduction algorithm[8] is used for converting in and out of montgomery form and calculating the montgomery product.And hence can be used for modular exponentiation algorithm.

C. The Proposed Algorithm

For computing $C = M^E modN$,repeat the following steps of the algorithm.It is assumed that R', N' and R are predetermined as required for montgomery reduction[2].

1) Input M as integer , E as E' i.e redundant representation of binary expression for E and N as integer value
2) Take a variable result to store the temporary result for each iteration .Initialize result as the montgomery form of 1.Thus we initialize

   $$result = R(modN), where\, gcd(R, N) = 1 \, and \, R > N.$$

3) If k is the no of bits in E then starting from most significant bit in 'E' i.e k-1 th bit to the least significant bit,and for each bit i from k-1 to 0 do the following:
   a) result=REDUCE((result.R mod N)*(result.R mod N))
      result= (result.R') mod N
   b) **If the ith bit is 1 then set**

result=REDUCE((result.R mod N)*(M.R mod N))
result=result.R' mod N

OR

If the ith bit is $\overline{1}$ then set
result=REDUCE((result.R mod N)*(M^{-1}.R mod N))
result=result.R' mod N

OR

else if the ith bit is 0 then set result=result

4) After completion of all iterations return result, which is in montgomery form and we convert it out of montgomery form to obtain the final result of encryption

$$C = REDUCE(result.R\ modN) = M^E modN.$$

The algorithm computes modular exponentiation in O((T(n))(n+E1)) where T(n) is time complexity of Montgomery reduction algorithm,n is the no. of bits in redundant representation of E and E1 is no of non zero elements in redundant representation.

This is much faster in comparison to the binary algorithm which has complexity of O(M(n)(n +Z)),n being the number of bits in E, Z being the no of 1's in the binary form of E. M(n) is time complexity for performing modular multiplication. Since $M(n) > T(n)$ and $E1 < Z$.

V. Adding Parallelism and Systolic Array Design

A. Parallel Processing

The Step 3 in the mentioned algorithm can be made faster by using two parallel processor say A,B for faster computations[17].

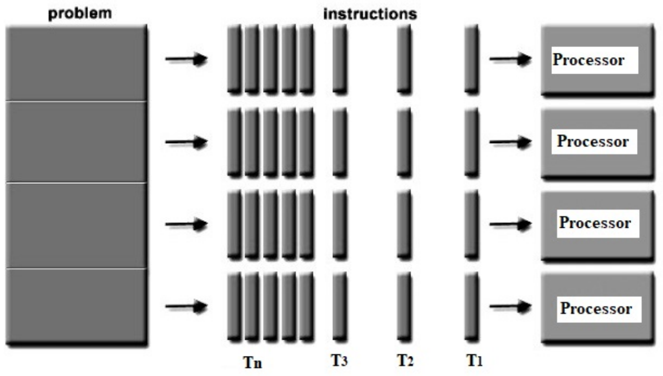


Figure 2: Parallel processing model to solve a problem.

By using two parallel processor the computations can be made twice as fast than usual.

The following is the modification to the step 3(a) and 3(b)

Step 3(a):
[Processor A]
result=REDUCE((result.R mod N)*(result.R mod N))
[Processor A]
result= (result.R') mod N

AND

At time same time the processor B will simultaneously compute...

Step 3(b):
If the ith bit is 1 then set
[Processor B]
result=REDUCE((result.R mod N)*(M.R mod N))
[Processor B]
result=result.R' mod N

OR

If the ith bit is $\overline{1}$ then set
[Processor B]
result=REDUCE((result.R mod N)*(M^{-1}.R mod N))
[Processor B]
result=result.R' mod N

OR

else if the ith bit is 0 then set
[Processor B]
result=result

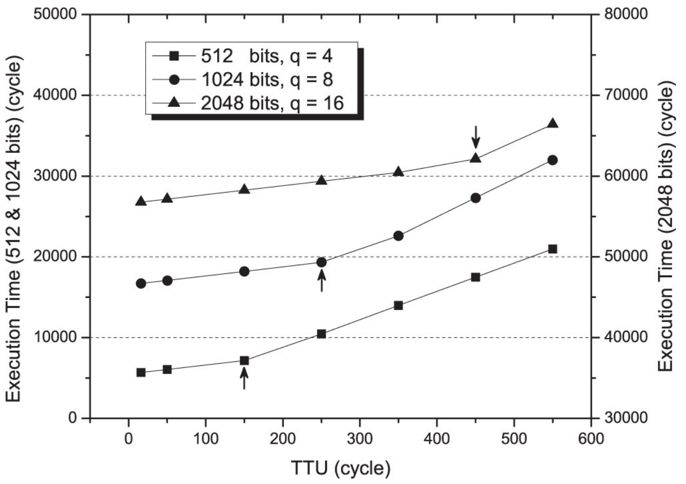


Figure 3: Execution time of Montgomery multiplication on multi core system. q: number of cores.

Here instead of computing step 3(a) and 3(b) sequentially one after the other we parallelly compute their value using two different processors thereby saving time and overload involved when using a single processor[11].

Thus it makes the time of computation get reduced by a factor of 2.

B. Systolic Array Design

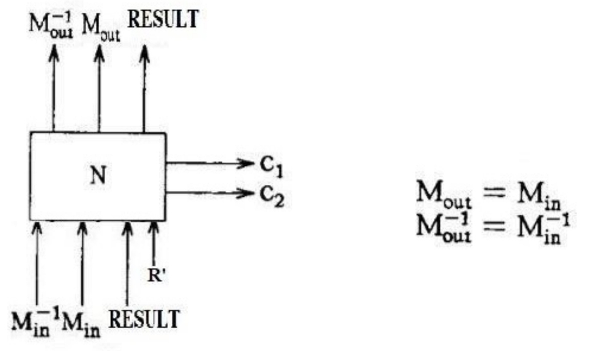
It is often in RSA cryptosystems that the values of private key and public key are kept same for encryption and decryption of multiple messages say[3]

$$M_j^E modN for j = 1, 2, ...m.$$

In this section[12] a systolic array design is proposed for the corresponding algorithm. Proposed design is a double linear systolic architecture for implementing the algorithm .This structure consists of 2n processing elements,n is the no. of bits in E and takes advantage of using the pipeline structure to reduce average computation time.The proposed double linear systolic array design can compute $M^E mod N$ in asymptotically constant time on average.

Each of the processing elements has two control inputs to perform three different operation depending on whether the current bit in E is 1,0 or $\overline{1}$. Here $M^{-1}.RmodN$ and $M.RmodN$ is precomputed .The processing unit is as shown in figure below .C_1 and C_2 are the control inputs which are set to either 10,01 or 00 depending on the ith bit in E that is currently being processed[17].

$$C_1C_2 = 01\, if\ e_i^* = 1$$
$$C_1C_2 = 10\, if\ e_i^* = \overline{1}$$
$$C_1C_2 = 00\, if\ e_i^* = 0$$



M_{out}^{-1} M_{out} RESULT
N
C_1
C_2
R'
M_{in}^{-1} M_{in} RESULT
$M_{out} = M_{in}$
$M_{out}^{-1} = M_{in}^{-1}$

Figure 4: Basic Function of Processing Element.

The processing elements perform the montgomery reduction and multiplication depending on the control inputs[16]:

.

If the ith bit is $\overline{1}$ i.e C_1C_2=10 then the processing element perform:

result=REDUCE((result.R mod N)*(M^{-1}.R mod N))
result=result.R' mod N

Or if the ith bit is 1 i.e C_1C_2=01 then,

result=REDUCE((result.R mod N)*(M.R mod N))
result=result.R' mod N

Otherwise, if the ith bit is 0 i.e C_1C_2=00 then,

DO NOTHING

The figure below shows the double systolic array design having a pipeline network of processing elements to compute $M_j^E modN$ in montgomery form.

The network consists of 2n processing elements ,the left side computes:

result=REDUCE((result.R mod N)*(result.R mod N))

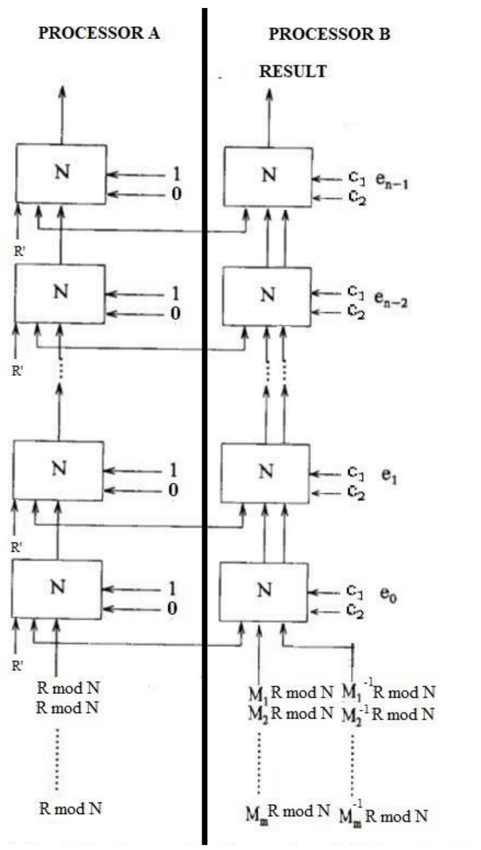


Figure 5: Systolic Array Implementation for computing $M_j^E modN$ in montgomery form.

result= (result.R') mod N

which corresponds to processor A in the proposed algorithm and is synchronized to the right side using a common clock. The control inputs are always kept 10 for left as they need to perform a fixed operation ,while for the right side corresponding to Processor B , the control varies depend on e_i bit in E under consideration[19].

The starting input to the left side is always R mod N and R' while the right side is fetched a sequence of M's (multiple messages) in their montgomery form and also M^{-1} in its corresponding montgomery form. The final RESULT is then return by the top right processing element (Processor B) which is in montgomery form and from this result we obtain the final encrypted text by reverting out of the montgomery form.

$$C = M^E modN = REDUCE(RESULT.RmodN)$$

It takes n time units delay to get the first result, from the time of the first message sent in and then the following results will be obtained in subsequent each time unit delay.The average time delay per message,$T(n) = O(n(m+n-1)/m)$

Now if m=1 then $T(n) = O(n^2)$ which is the case of implementing by two processing elements[3].

If m=n-1 then T(n)=O(2n) and if $m >> n$ then naturally T(n)=O(1), and therefore this network achieves constant time delay to compute each $M_j^E modN$ (j=1,2,....m) on average if m (no. of messages) is big enough.

VI. Results and Discussion

1: DECLARATION OF MONTGOMERY REDUCER CLASS

```
public class MontgomeryReducerDemo
{
private BigInteger p;
private BigInteger q;
private BigInteger mod;
private BigInteger phi;
private BigInteger y;
private BigInteger d;
private int;
bitlength = 1024;
private Random r;

public MontgomeryReducerDemo()
{
r = new Random();
p = BigInteger
    .probablePrime(bitlength, r);
q = BigInteger
    .probablePrime(bitlength, r);
mod= p.multiply(q);
phi = p.subtract(BigInteger.ONE)
.multiply(q.subtract(BigInteger.ONE));
y = BigInteger
.probablePrime(bitlength / 2, r);
while (phi.gcd(y)
.compareTo(BigInteger.ONE) > 0
&& y.compareTo(phi) < 0)
{
y.add(BigInteger.ONE);
}
d = y.modInverse(phi);
}

public MontgomeryReducerDemo
(BigInteger y,BigInteger d,
BigInteger mod)
{
this.y = y;
this.d = d;
this.mod = mod;
}
```

2: ORIGINAL RSA CODE SNIPPET

```
@SuppressWarnings("deprecation")
public static void main(String[] args)
throws IOException
{
MontgomeryReducerDemo rsa
= new MontgomeryReducerDemo();
DataInputStream in
= new DataInputStream(System.in);
String teststring;
System.out
.println("Enter␣the␣plain␣text:");
teststring = in.readLine();
System.out.println("Encrypting␣String:"
+teststring);
byte[] sample=teststring.getBytes();
System.out.println("String␣in␣Bytes:"+
bytesToString(teststring.getBytes()));
/////RSA CRYPTOGRAPHY/////
System.out
.println("RSA␣IMPLEMENTATION");

/////ENCRYPT/////
long estartTime_rsa =System.nanoTime();
byte[] encrypted_rsa
= rsa.encrypt(sample);
long estopTime_rsa =System.nanoTime();
System.out
.println("value␣of␣e␣:␣"+ rsa.y);
System.out
.println("value␣of␣N␣:␣"+ rsa.mod);
System.out
.println("Time␣in␣encryption:"+
(estopTime_rsa-estartTime_rsa)+"ns\n");
System.out
.println("Encrypted␣text:␣" +
bytesToString(encrypted_rsa));

/////DECRYPT/////
long dstartTime = System.nanoTime();
byte[] decrypted_rsa
= rsa.decrypt_rsa(encrypted_rsa);
long dstopTime = System.nanoTime();
System.out
.println("\nTime␣in␣decryption:"+
(dstopTime-dstartTime)+"ns");
System.out
.println("Decrypting␣Bytes:␣" +
bytesToString(decrypted_rsa));
System.out
.println("Decrypted␣String:␣" +
new String(decrypted_rsa));
```

3: ENCRYPTION AND DECRYPTION FUNCTIONS

```
//Encrypt message//
public byte[] encrypt(byte[] message)
{
return (new BigInteger(message))
.modPow(y,mod).toByteArray();
}

private static String
bytesToString(byte[] encrypted)
{
String test = "";
for (byte b : encrypted)
{ test += Byte.toString(b); }
return test;
}

//Decrypt message//
public byte[]
decrypt_rsa(byte[] message)
{
return (new BigInteger(message))
.modPow(d, mod).toByteArray();
}
```

4: MONTGOMERY CODE

```
System.out.println
("MONTGOMERY␣REDUCER␣IMPLEMENTATION");
System.out.print
("Operation(\"times\"or\"pow\"):");
String oper = in.readLine();
BigInteger x = new BigInteger(sample);

//encrypt
byte[] encrypted = rsa.encrypt(sample);
//System.out.println("Encrypted text:"+
bytesToString(encrypted)+"\n");
long estartTime = System.nanoTime();
rsa.function(x,oper);
long estopTime = System.nanoTime();
System.out
.println("Time␣in␣encryption:"+
(estopTime-estartTime)+"ns\n");
}
```

5: MONTGOMERY CODE FUNCTION

```
public void
function(BigInteger x,String oper)
{
MontgomeryReducer red
= new MontgomeryReducer(mod);
BigInteger xm = red.convertIn(x);
BigInteger zm;
BigInteger z;
if(oper.equals("times")) {
zm = red.multiply(xm,red.convertIn(y));
z = x.multiply(y).mod(mod);
}
else if (oper.equals("pow"))
{
zm = red.pow(xm, y);
z = x.modPow(y, mod);
}
else
throw new IllegalArgumentException
("Invalid␣operation:␣" + oper);
if (!red.convertOut(zm).equals(z))
throw new
AssertionError("Self-check␣failed");
System.out
.println("\nEncrypted␣Text␣=␣" + z);
}
```

6: MONTGOMERY MULTIPLICATION

```
public final class MontgomeryReducer{
private BigInteger modulus;
private BigInteger reducer;
private int reducerBits;
private BigInteger reciprocal;
private BigInteger mask;
private BigInteger factor;
private BigInteger convertedOne;
public MontgomeryReducer
(BigInteger modulus)
{
//Modulus
if (modulus == null)
throw new NullPointerException();
if (!modulus.testBit(0) ||
modulus.compareTo(BigInteger.ONE) <= 0)
throw new IllegalArgumentException
("Modulus␣must␣be␣an␣odd␣number␣>=␣3");
this.modulus = modulus;

// Reducer
reducerBits =
(modulus.bitLength() / 8 + 1) * 8;
reducer = BigInteger.ONE
.shiftLeft(reducerBits);
mask=reducer.subtract(BigInteger.ONE);
assert reducer.compareTo(modulus) > 0
&& reducer.gcd(modulus)
.equals(BigInteger.ONE);

//Other Computed Numbers
reciprocal=reducer.modInverse(modulus);
factor = reducer.multiply(reciprocal)
.subtract(BigInteger.ONE)
.divide(modulus);
convertedOne = reducer.mod(modulus);
}
```

7: MONTGOMERY EXPONENTIATION ALGORITHM CODE

```
//The range of x is unlimited
public BigInteger
convertIn(BigInteger x) {
return x.shiftLeft(reducerBits)
.mod(modulus);
}

//The range of x is unlimited
public BigInteger
convertOut(BigInteger x) {
return x.multiply(reciprocal)
.mod(modulus);
}

/*Inputs and output are in Montgomery
form and in the range [0, modulus)*/

public BigInteger
multiply(BigInteger x, BigInteger y) {
assert x.signum() >= 0 &&
x.compareTo(modulus) < 0;
assert y.signum() >= 0 &&
y.compareTo(modulus) < 0;
BigInteger product = x.multiply(y);
BigInteger temp = product.and
.multiply(factor).and(mask);
BigInteger reduced =
product.add(temp.multiply(modulus))
.shiftRight(reducerBits);
BigInteger result = reduced
.compareTo(modulus) < 0 ?
reduced : reduced.subtract(modulus);
assert result.signum() >= 0 &&
result.compareTo(modulus) < 0;
return result;
}

/*Input x(base) and output(power)are in
Montgomery form and in the range
[0, modulus);input y (exponent)
is in standard form*/

public BigInteger
pow(BigInteger x, BigInteger y) {
assert x.signum() >= 0 &&
x.compareTo(modulus) < 0;
if (y.signum() == -1)
throw new IllegalArgumentException
("Negative␣exponent");
BigInteger z = convertedOne;
for(int i = 0, len = y.bitLength();
i < len; i++) {
if (y.testBit(i))
z = multiply(z, x);
x = multiply(x, x);
}
return z;}}
```

Table II: Comparison between Encryption time of original RSA with Proposed Algorithm.

Input String	Size of 'N' (bits)	Encryption time in RSA(a) (ns)	Encryption time in Montgomery(b) (ns)	$\frac{a-b}{a} * 100$
Nit Silchar	565	28889714	1777774	93.8
Institute of Electrical and Electronics Engineers	630	28738602	2296884	92
Computer Science and Engineering	443	28763492	2484439	91.3
Assam	230	28244470	1787113	93.6
ZigBee and IEEE 802.15.4 are standads based protocols that provide the network infrastructure required for wireless sensor network applications	800	29064471	2676002	90.7

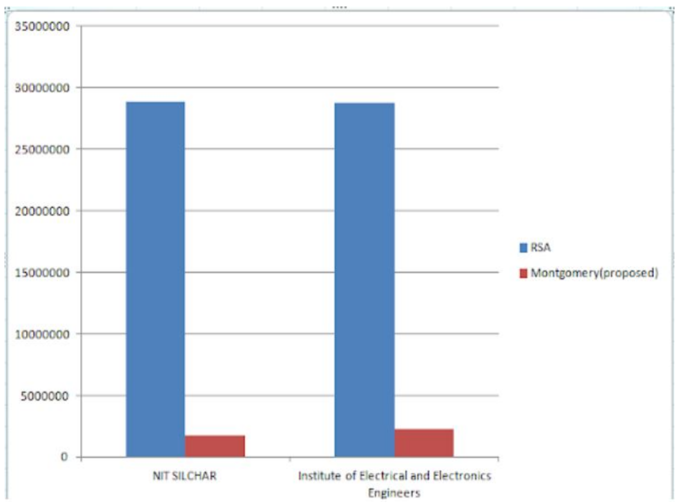


Figure 6: Encyption time comparision between RSA Algorithm and Proposed Algorithm for two given input strings.

Depending on the various input plaintexts and (N,E) values the encryption time for the proposed algorithm has been compared with the original RSA implementation and based on the results obtained an exhaustive comparative table has been laid out as shown in the table II in the next page.

It can been observed from the table II and figure 4.1 that with different inputted words and keeping the public key same for both original RSA and proposed algorithm for each input word, that:

- efficiency/speed up is achieved over classical RSA encryption.
- efficiency decreases minimally with the increasing size of public key.Thus even for large keys the speed up is achieved.

VII. Conclusion and Future Work

Thus the proposed algorithm makes the encryption faster through by reducing the number of iterations with recoding of E into a redundant representation.The encryption process achieves further speed up by optimizing the modular multiplications through the use of Montgomery reduction algorithm . On top of it the parallel processor and systolic array design further increases computation speed by factor of 2 and enables fast encryption of a sequence of messages for fixed public and private key.

The proposed algorithm reduces the total operations and optimizes each of the operation by the underlying algorithm and in case of very large keys this optimization makes a great effort in reducing the time for encryption. Experimental analysis of the algorithm in parallel environments is left as a future scope of study .Enhancements and side effects on the other supporting processes in the cryptosystems are kept as future scope of study.

References

[1] Ravi Shankar Dhakar,Amit Kumar Gupta, Prashant Sharma," Modified RSA Encryption Algorithm (MREA)",*pp.426 – 429, 7-8 Jan. 2012, Rohtak, Haryana, India*

[2] Martin Kochanski, "Montgomery Multiplication" a colloquial explanation.

[3] Chang N.Zhang , Herold L.Martin, "Parallel algorithms and Systolic Array Design For RSA Cryptosystems" Systolic Arrays, 1988., Proceedings of the *International Conference on 25-27 May 1988,San Diego, CA, USA, USA.*

[4] R. L. Rivest, A. Shamir, and L. Adleman. "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM, vol. 21, no. 2, pp. 120–126, Feb. 19*

[5] Hung-Min Sun, Mu-En Wu, Wei-Chi Ting, and M. Jason Hinek." Dual RSA and Its Security Analysis",pp. 2922 – 2933.vol:53,2007.

[6] H. C. Williams, "A Modification of the RSA Public-Key Encryption Procedure", IEEE Transactions on Information Theory, Vol. 26, No. 6, pp. 726-729, 1980

[7] Prabhat K. Panda and Sudipta Chattopadhyay, " A Hybrid Security Algorithm for RSA Cryptosystem", Jan. 06 – 07, 2017, Coimbatore, India.

[8] Nitha Thampia, Meenu Elizabath Jose, "Montgomery Multiplier for Faster Cryptosystems" *Procedia Technology 25 (2016) 392 – 398 Available online at www.sciencedirect.com*

[9] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. "Handbook of Applied Cryptography". *CRC Press, 1996. ISBN 0-8493-8523-7, chapter 14.*

[10] H. T. Kung, C. E. Leiserson: Algorithms for VLSI processor arrays; in: C. Mead, L. Conway (eds.): Introduction to VLSI Systems; Addison-Wesley, 1979

[11] A Parallel Implementation of Montgomery Multiplication on Multicore Systems: *Algorithm, Analysis, and Prototype,Issue No. 12 - December (2011 vol. 60)* Zhimin Chen , Virginia Polytechnic Institute and State University, Blacksburg,Patrick Schaumont , Virginia Polytechnic Institute and State University, Blacksburg.

[12] S. Y. Kung: VLSI Array Processors; Prentice-Hall, Inc., 1988

[13] Chih-Yuang Su, Shih-Arn Hwang, Po-Song Chen and Cheng-Wen Wu,"An Improved Montgomery's Algorithm for High-Speed RSA Public-Key Cryptosystem", *IEEE Transactions on VLSI Systems, vol. 7, no. 2, pp. 280 - 284 June 1999*

[14] Hubbard, Steven S, "Radix – 4 Implementation of a Montgomery Multiplier for a RSA Cryptosystem." (2006).

[15] C. McIvor, M. McLoone and J.V. McCanny, "Modified Montgomery modular multiplication and RSA exponentiation techniques", *IEE Proceedings - Computers and Digital Techniques, Volume: 151, Issue: 6, pp. 402 - 408, 18 Nov. 2004*

[16] S. E. Eldridge, “A faster modular multiplication algorithm,” *Int. J. Comput. Math., vol. 40, pp. 63–68, 1991.*

[17] N. Petkov: ”Systolic Parallel Processing”; North Holland Publishing Co, 1992

[18] Sushanta Kumar Sahu and Manoranjan Pradhan,”Implementation of Modular multiplication for RSA Algorithm”, *2011 International Conference on Communication Systems and Network Technologies.*

[19] Min-Sup Kang, Dong-Wook Kim, ”Systolic array based on fast modular multiplication algorithm for RSA cryptosystem”,*TENCON 99. Proceedings of the IEEE Region 10 Conference,pp. 305 - 308 vol.1, 15-17 Sept. 1999,South Korea*

[20] Diffie, W., and Hellman, M. ”New directions in cryptography”. IEEE Trans. Inform. Theory IT-22, 6 (Nov. 1976), 644-654.

[21] Diffie, W., and Hellman, M. ”Exhaustive cryptanalysis of the NBS data encryption standard”. Computer 10 (June 1977), 74-84.

[22] Knuth, D. E. ”The Art of Computer Programming”, Vol 2: Seminumerical Algorithms. Addison-Wesley, Reading, Mass., 1969.

[23] Levine, J., and Brawley, J.V. ”Some cryptographic applications of permutation polynomials”. Cryptologia 1 (Jan. 1977), 76-92.

[24] M O. Rabin, “Digitized signatures and pubic-key functions as intractable as factorization” M.Z.T. Lab. for Computer Science, *Tech. Rep. LCS/TR-212, 1979.*

[25] G. J. Simmons and M. J. Norris, “F’ Preliminary comments on the M.I.T. public-key cryptosystem,” *Cryptologia, vol. 1, pp. 406-414, 1977.*

[26] B.R. Ambedkar, A. Gupta,P. Gautam and S.S.Bedi, ”An Efficient Method to Factorize the RSA Public Key Encryption.” *Communication Systems and Network Technologies (CSNT), 2011 International Conference on. IEEE, pp.108-111, 2011.*

[27] M.Thangavel, P. Varalakshmi, M. Murrali and K.Nithya, “An enhanced and secured RSA key generation scheme” *Journal of Information Security and applications, Elsevier, vol 20, pp.3-10, 2015.*

[28] R. Minni, K. Sultania and S.Mishra, “An algorithm to enhance security in RSA” ,*4th ICCCNT, IEEE , pp.1-4, 2013.*

[29] A.H. Al-Hamami and Aldariseh IA, “Enhanced method for RSA cryptosystem algorithm”, *international conference on Advanced Computer Science Applications and Technologies, Kuala Lumpur, IEEE, pp. 402-408, 2012.*

[30] W. Rui, C. Ju and D. Guangwen, “A k-RSA algorithm”,*3rd ICCSN, Xi’an, China,IEEE, pp.21-24, 2011.*