Design and Implement Fast Algorithm of RSA Decryption using java

Ammar H. Jasim
University of Baghdad - College of Science for women
Department of computer Science

المستخلص :

تم تصميم وتطبيق خوارزمية سريعة لطريقة RSA باستخدام نظريةِ البقيّةِ الصينيةِ (CRT) وطريقة التربيع مع الضرب. كذلك تضمن تطبيق العدد الصحيح الكبيروتوليد أعداد اولية كبيرة وطريقة القاسم المشترك الاكبر الموسعة للاعداد الصحيحة الكبيرة.النظام صُمّمَ كمعالجات منفردة لتَضْمين العملياتِ الضروريةِ لحساب أس التشفير لخوارزمية الـRSA.الذي يحدد عدد العمليات المستعملة لضرب المعامل الضروري لإداء العمليةِ الأسّيةِ و يُحدد بدوره حجمَ النّتائج في CRT ونظرية Fermat. يركزهذا العمل على تسريع جزء حل التشفيرفي خوارزمية الـ RSA بالاستناد الى CRT. تم تصميم مجموعة لتوليد ارقام صحيحة اولية كبيرة خاصة لتوليد معاملات أسّية لـ CRT لتكوين مفاتيح التشفيرباستعمال سلسلة من التربيع والضرب لتقليص الوقت المطلوب لانجاز المعامل الأسي على كل عدد صحيح كبير أولي تم توليده بدلا من استعمال الدالة الأسية نفسها.

تم استعمال إختبار ميلير رابين الإحتمالي على أرقام صحيحة كبيرة. لاختبار الخوارزمية المستخدمة لتوليد ارقام صحيحة عشوائية مع احتمالية ان تكون أولية لطول محدد من الـثنائيات. وقد تم توليد ارقام عشوائية كبيرة ومن ثم تم اختبارها بأستعمال الخوارزمية المقترحة.

ABSTRACT :

Based on the principle of RSA, RSA cryptosystem using Chinese Remainder Theorem (CRT) and square-multiply method is designed and implemented, including large integer, generation of big primes and computing Extended Greatest Common Divisor (EGCD) of big Integer.

The system designed as threads to include the necessary operation to realize operation of computing decryption exponent of RSA algorithm which specifies the number of modular multiplications needed to perform the exponential process and the modulus to determine the size of the intermediate results, hence; make use of the properties stated by the CRT and Fermat's theorem. This paper focus on increasing RSA speed in the decryption part based on CRT. The design of a class for generating special prime big Integer to construct a special decryption keys and a class built as

a thread to generate special CRT modular exponentiations to construct the decryption keys. A sequence of squaring and multiplications are used to decrease the time to perform modular exponentiation on each generated prime Big Integer instead of using exponentiation.

A Miller-Rabin probabilistic test is used to run on the Big Integers. It is used to test an algorithm which generates a random integer with a primly probability at a specific bit-length. Large random numbers were generated and then a test for primarily using Miller-Rabin was tested.

1. Introduction

Public key cryptosystem was introduced in 1976 by Whitfield Diffe and Martin Hellman of Stanford University. It uses a pair of related keys one for encryption and other for decryption. One key, which is called the private key, is kept secret and other one known as public key is disclosed [1]. The message is encrypted with public key and can only be decrypted by using the private key. So, the encrypted message cannot be decrypted by anyone who knows the public key and thus secure communication is possible. RSA [2] (named after its authors Rivest, Shamir and Adleman) is the most popular public key algorithm. In relies on factorization the problem of mathematics that states, given a very large number it is quite impossible in today's aspect to find two prime numbers whose product is the given number. As the number become larger, the possibility for factorizing it decreases.

So, very large numbers were needed for a good public key cryptosystem. Java has an excellent library called BigInteger Package that can handle numbers of arbitrary precision [3]. This library is used to implement RSA algorithm. BigInteger instead of the standard integer must be used because an integer variable cannot exceed $2^{31}-1$ while a BigInteger can simulate arbitrary-precision integers [3].

2. System Design and Analysis

In RSA [2], the most extensive work being done is power and module operations. Power module operation cannot be computed directly therefore this reduces the speed of RSA and hence it is time-consuming. The multiplications modulo is being done on a large number which is the core of this algorithm. These operations are time consuming, making even a Pentium IV unable to perform more than few thousands cryptographic

operations per second [4]. Either division or series of subtractions must be used, but most algorithms for division can only calculate one (or two) bit(s) per cycle, hence a complete multiplication of a*b mod n, with n is 2048 bit modulo for examples, might at least take 32+2048 cycle, where the 32 cycles are used to calculate the product and the 2048 cycles are for the trail division [4].

Square and multiply algorithm [5] was further modified to reduce the number of iterations, doubling the speed of modular multiplication, and then using CRT technique to reduce the RSA computation by a divide-and-conquer method.

By taking the advantage of the CRT [6],hence, designing the system that the computational effort of decryption can be reduced significantly if the two prime numbers P and Q of the modulus N are known, then it is possible to calculate the modular exponentiation M = $C^D \bmod N$ separately with mod P and mod Q being the shorter exponents, decomposition exponentiation. Since the length of the exponent is n/2, approximately, 3n/4 modular multiplications are needed for a single modular exponentiation [7].

The scenario of the proposed design is based to accelerate decryption part of the RSA algorithm using CRT. The modulus used in the RSA encryption schema is the product of two prime numbers. This allows utilizing CRT in order to speed up the private key operations. From a mathematical point of view, the usage of CRT for RSA decryption is well known. But for software implementations, a special algorithm is necessary to meet the requirements for efficient CRT-based decryption.

2.1. Modular Exponention Algorithm

To encrypt a message using the encryption key *(E, N)*, the message is first partitioned into a sequence of blocks and each block M is considered as an integer between *0* and *N-1*. Then, the message is encrypted by raising *M* to *Eth* power modulo *N*, i.e. $C = M^E \bmod N$. Similarly, to decrypt the ciphertext C using the decryption key (D, N), C is raised to the power of D modulo N, i.e., $M = C^D \bmod N$.

Clearly, modular exponention is the main operation of the RSA algorithm. For modular exponention, a sequence of modular multiplication can be performed instead [8].

2.2. Exponentiation by Squaring and Multiplication :

Both encryption and decryption involve raising a numerical representation of the message (original or encoded) to a power (e or d) and then finding the remainder when the result is divided by n. The usage of RSA algorithm, namely exponentiation, which involves squaring and multiplication engaging repeatedly squaring and multiplying a temporary variable by M (or C) and mod n at each step thus allowing the use of much larger values for n, e and d [5].

2.3. Square and Multiply Algorithm :

This algorithm starts at the least significant bit and works upward [8]. This algorithm requires a temporary variable to store the middle variable.

Algorithm (Square and multiply Method)

```
Input: M, E;
Output: M^E contained in C;
S=M;
C=1;
  For i=1 to K
    {
      If (i-th bit of E is 1) then
        C=C*S;   //Multiply
        S=S*S;   //Square
}
```

The multiplication and squaring of this algorithm are independent of one another and thus two operations at each loop can be parallelized using thread technique in java which is built by two threads, one for each.

3. BigInteger Implementation:

The BigInteger class used is intended to store large integers and execute any ordinary mathematical operation. The BigInteger class will represent numbers digitally, in a numeration basis of choice $2 \le base \le 2^{16}$. The traditional integer operators (+, –, *, /, %, <, >, =<, >=, = =, !=, =) have been overloaded, so that any subsystem that uses the BigInteger class is offered the typical and natural way of manipulating the arithmetical operations on a specific BigInteger. The following specific arithmetic,

abstract algebra and computational operations have also been loaded into the BigInteger class, in order to enhance its usefulness [3]:

- The power operation (through fast binary exponentiation)
- The greatest common divisor (GCD), using the standard Euclidean algorithm and the extended Euclidean algorithm.
- The modular multiplicative inverse algorithm.
- The modular exponentiation method, using the repeated squaring algorithm.
- The serializing and deserializing of a BigInteger.
- Adding and removing the salt digits of a BigInteger.
- Adding, checking and removing of the replication digits of a BigInteger.

4. RSA Decryption Complexity :

The complexity of the RSA decryption $M = C^D \bmod N$ depends on the size of *D* and *N*. The decryption exponent *D* specifies the numbers of modular multiplications necessary to perform the exponentiation and the modulus *N* determines the size of the intermediate results. A way of reducing the size of both *D* and *N* is to take advantage of properties stated by the CRT and Fermat's little theorem. Fermat's little theorem [8] is very useful for calculating the multiplicative inverse of an integer (a) because $a^{p-1} \equiv a^{-1} \bmod p$.

4.1. Decryption Part :

Let m be the plaintext and *C* the ciphertext. If *C* is not divisible by p and $d_p == d \bmod p-1$, then $C_p^{\,d_p} == c^d (\bmod p)$ for decryption.

4.1.1. RSA Decryption Part Method

RSADP(K,C)

Input: *K*: - where *K* has the following forms:

1- a pair (*N, d*)

2- quintuple ($p, q, d_p, d_q qInv$)

C: - ciphertext representative, an integer between *0* and *N-1*.

Output: *M*: message representative, an integer between *0* and *N*-1.

Errors: "ciphertext representation, an integer between *0* and *N-1* (out of range)".

Steps:-

1- If the ciphertext representing C is not between 0 and $N-1$, output "ciphertext representing out of range" and stop.

2- If the first form (n,d) of K is used:

2.1- let $M = C^d \bmod N$.

Else, the second form $(p,q,d_p,qInv)$ *and* (r_i,d_i,t_i) of K is used:

2.2- let $M_1 = C_p^{d_p} \bmod p$.

2.3- let $M_2 = C_q^{d_q} \bmod q$.

2.4- let $h = (M1 - M2).qInv \bmod p$.

2.5- let $M = M_2 + q.h$.

3- Output m.

5. System Algorithms

In this system a description of three constituent algorithms: key generation, encryption and decryption are given.

5.1. Key generation:

Input:

1. 1- *N=4096*; // standard security parameter.
2. Generate two *(N/2)*-bit primes *p,q*. differing in length by 10-20 bits. (If the primes are too close to sqrt(n), then factorizing is the solution). The primes are also chosen so that *p-1* and *q-1* do not have 3 as a factor, because this implementation uses 3 as the encryption exponent. A special generator is used to create a long bit seed.
3. Sets *N=p.q*.
4. Pick a small value of e which is relatively prime to $\phi(n) = (p-1)(q-1)$. Where encryption exponent usually chosen e=3.

Output:

1. RSA public key (N,e)
2. RSA private key (d)
3. $e.d = 1 \bmod \phi(N)$.

5.2. Encryption

First format the bit-string M to obtain an integer M in $Z_n = \{0,...., N-1\}$.

1. Use an exponent e of 3.

2. Use RSA public key (N,e)
3. Compute $C = M^e \bmod N$

5.3. Decryption

The RSA decryption operation can be speeded up by using the CRT [7], where the factors of the modulus N (i.e., P * Q) are assumed to be known. If c is ciphertext, then RSA decryption calculates:-

$$M = C^D \bmod N \text{ ....1}$$

By CRT, the computation of equation (1) can be partitioned into two parts:

$$v_1 = C_p^{D_p} \bmod P \quad and \quad v_2 = C_q^{D_q} \bmod q$$

where

$$C_p = C \bmod P,\ D_p = D \bmod (P-1), \text{.........2}$$

$$C_q = C \bmod Q,\ D_q = D \bmod (Q-1) \text{..........3}$$

Instead the CRT allows one to deduce *C mod N* from the knowledge of *C mod P* and *C mod Q*. In fact, their size is about half the original size. In the ideal case, a speedup of about 4 times may be achieved. Finally, compute *M* by CRT as follows:

Arithmetic *mod P* should be converted to *mod (P-1)* in an exponent because

$$a^{P-1} \bmod P = 1 \quad (fermat's\ theorem)$$

Thus a simpler calculation may be used which is:-

$$v_1 = c^{d \bmod (p-1)} \bmod p \quad and$$

$$v_2 = c^{d \bmod (q-1)} \bmod q$$

Finally, calculate:

$$C_2 = p^{-1} \bmod q \quad and$$

$$u = (v_2 - v_1) C_2 \bmod q$$

Giving:

$$M = C^d \bmod n = v_1 + up.$$

Or

$$M = (M_p (Q^{-1} \bmod P) Q + M_Q (P^{-1} \bmod Q) P) \bmod N.$$

When using CRT technique [7], a partitioning process of the modular multiplier is done which will give two smaller ones according to their lengths P and Q. Then, computation of $M_p = C_p^{D_p} \bmod P \ \ and \ \ M_q = C_q^{D_q} \bmod Q$ using square and multiply method, which is then, put into processes as independent threads for modular multipliers, in turn reducing the computation time. Note that the partition is done according to the lengths of P and Q, so each thread must be able to

read the data from the primary input directly, and be able to return the output data to the primary output variable to finish the final step of the modular multiplication. P and Q are fixed after key generation, therefore, the partition of N into P and Q needs to be done only once, before the RSA computation starts.

5.4. Pseudo Code of RSA Encryption

1. Read *E*, *N*, message.
2. Convert message to BigInteger representation (m).
3. BigInteger $C = (M^E)\%N$.
4. Printout *C*.

5.5. Pseudo Code for RSA Key Generation

1. Read two prime BigInteger *p*, *q*.
2. Compute BigInteger *N*, *v* where $N=p*q$, $v=(p\text{-}1)*(q\text{-}1)$.
3. Read small odd integer E=3 such that gcd (E, v) =1.
4. Compute integer d such that $(d*E)\ \%v=1$.
5. Write (*N*, *E*, *d*).

5.6. Pseudo Code for RSA Decryption

1. Read *d*, *N* ,*C*. (*C* is integer representation of ciphertext message)
2. Compute $D = (C^d)\%N \equiv C_p{}^d\%P, C_q{}^d\%Q$. (*D* integer representation of decrypted ciphertext)
3. Write *D*.

5.7. Pseudo Code for Probable Prime

Using Rabin-Miller test for determine if a given number is prime. (Create a random number generator)

Random rng=new Random ();. (Declare p and q as type BigInteger)

BigInteger p, q; (Assign values to p and q as required)

P=BigInteger.ProbablePrime (2048, rng);

Q=BigInteger.ProbablePrime(2048, rng);

5.8. Pseudo Code for Power Module

Compute M mod N using BigInteger.

```
BigInteger PowMod(BigInteger M, int[] b, BigInteger N, int K)
{
BigInteger Result=1;
```

```
 For (int i=k; i>=0; i--)
  {
 Result=Sqr(Result,Result);    //Run Thread to compute
          Result=Result*Result;  (squaring)
   If(b[i]==1)
   {
 Result=Mul(Result,M); // Run Thread to compute Result=Result*M;
        (Multiplication)
   }
   If(Result>=N)
     {
       Result=Result%N;
     }
  }
Return Result;
}
```

6. System Development and Implementation

This work focuses primarily on the implementation of Fast RSA. For efficient design and implementation, MATH and BigInteger library in JAVA have been used. Exploration of the behavior and feasibility of the algorithm when changing various input parameters, have been discussed. Because any practical implementation of RSA cryptosystem should involve working with large integers (in this case, of 2048 bits). One way of dealing with this requirement would be to write a library that handles all the required functions. While this would indeed make such application independent of any other third-party library, however, this was not done due to mainly two considerations. First, the speed of this implementation would not match the speed of the libraries available for such purposes. Second, it would probably be not as secure as some available open-source libraries. A choice of two libraries; the BigInteger library and MATH library [3]. The BigInteger seemed to suit such needs. The BigInteger library aims to provide the fastest possible arithmetic and arbitrary precision integer for applications that need a higher precision than the ones that directly supported under JAVA by using highly optimized method. The BigInteger data types which provide immutable arbitrary-precision integers have been used to ensure accuracy in calculations involving large values.

The message, key generation, encryption and decryption routines all use the integer handling functions offered by this library. All the usual mathematical operations to BigInteger as well as others like modular arithmetic, gcd, primality testing etc. may be applied.
In this design software was developed to encrypt and decrypt alphanumeric information using Fast RSA algorithm. This software allows generating different public keys from two prime numbers provided by the user, the user must select a public key to generate the corresponding private key. To encrypt the information, the user must provide the public key of the recipient as well as the message to be encrypted. The generated ciphertext can then be sent through an insecure channel, at the end of the transmission, the recipient can decrypt the original message if the public key and the corresponding private key, were provided.

6.1. Establishing the parameters and operations for Fast RSA scheme

1. System parameters
   1- p and q are two big primes whose bit sizes 2048, and they are kept secret. For the safety case, the difference of two sizes must be big enough.
   2- $N = P * Q, \phi = (P-1)(Q-1)$, and ϕ is kept secret. Our software uses a 4096 bit modulus Fast RSA implementation.
   3- E can only plus integer but the great common division of e and ϕ must be 1. D is an integer and $d * e \equiv 1 \bmod \phi$.
   4- $K_p = (E,N)$ Is the public key and $K_s(D,N)$ is the private key.
   5- Let M is a plaintext , $(M^E)^D \equiv M \bmod N \equiv M \bmod P \;, M \bmod Q$.

2. The procedures
   1- Encryption: $C = Enc(E,M) = M^E \bmod N$.
   2- Decryption: $M = Dec(D,C) = C^D \bmod N = C^d \bmod P, C^d \bmod Q.$.

3. The operations
   1- Select two big primes, and compute N and ϕ.
   2- To speedup encryption select E=3. Based on E and N, D can be computed.
   3- Compute $M_{key} \bmod P, M_{key} \bmod Q$, where key can be E as well as D.

E can be tried from 3 for $\phi = (P-1)(Q-1)$ is even, therefore E can be increased with 2 once. The algorithm of computing E can be:

```
BigInteger ComputeE(BigInteger φ)
{
BigInteger e=3;
BigInteger division=0;
  While (GCD(φ,e)!=1) // Here GCD is the function to compute the
greatest common
                         division of two integers, and it can be implemented
with
                         Euclid Algorithm.
{
   E=e+2;
  }
 Return e;
}
```

Figure (1) illustrates all steps of proposed schema. While figure (2) show the specific operation of Fast RSA schema.

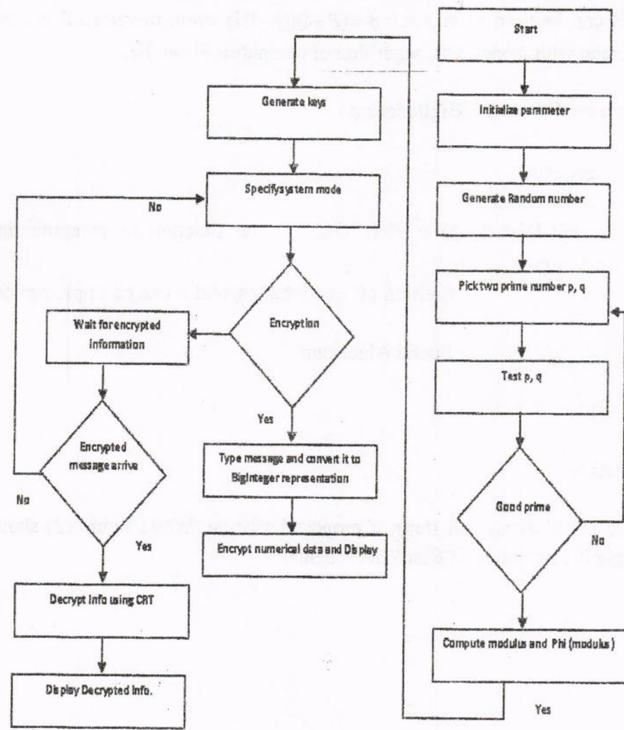

Figure 1: Proposed System

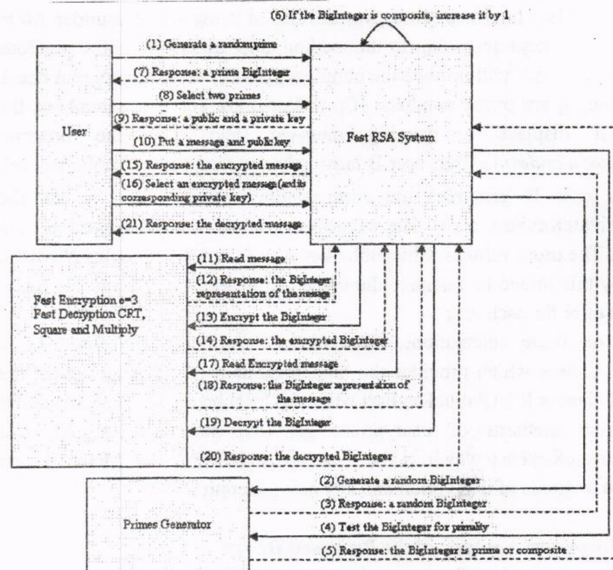

Figure 2: Fast RSA System Operation Diagram

6.2. System Model

Figure (3) depicts the suggested layer for the implementation of the FastRSA system.

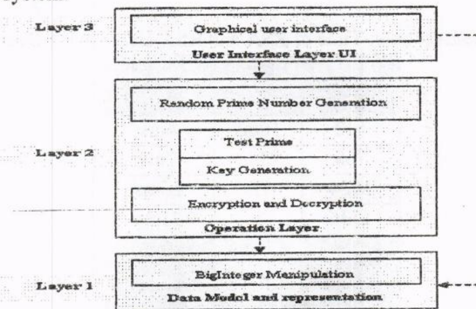

Figure 3: System Layers

Graphical User Interface (GUI) was developed using JAVA Builder 7.0 to generate the keys according to the methodology described in the previous section. The user will provide the numbers p and q, then the program check that p and q are prime numbers. If p and/or q are not prime numbers, the program displays a warning message, and then the program calculates $n, and \phi(n)$ .After, user B has to choose randomly a public key (e). Finally user B generates the corresponding private key, by run the RSAPrivateKeyFast class. The suggested method was implemented as a thread. The input values of the public key e and the RSA modulus N will be used by this thread to calculate the value of the secret key d and the prime factors of N for each user.

Based on these calculations, three classes are built. The first one is FastRSA class which provides the implementation of core algorithms; the second class is RSAPublicKeyFast which is the subclass of FastRSA, and it has three methods of encryption, get *(N)*, set modulus *(N)*, and RSAPrivateKeyFast which is the subclass of RSAPublicKeyFast. It also has two methods of decryption and key pair generation.

7. Performance Analysis of the Proposed Design

The Experimental results are listed in table (1). Note, each time measurement correspond to the average of 5 executions. The outputs are calculated using e equals to 3 as encryption exponent with the variation of the constant which representing the key size. The following execution times were recorded using the message ("Welcome To Baghdad University"):

Table (1): The Execution Time of RSA and RSA-Based-CRT Decryption

Decryption		
Key Size	RSA	RSA-CRT
512	16ms	4ms
768	23ms	7ms
1024	47ms	15ms
1536	125ms	31ms
1792	172ms	47ms
2048	281ms	78ms
2560	531ms	140ms
4096	2000ms	547ms

Table (2): The Execution Time of RSA Encryption

Encryption e=3	
Key Size	RSA
512	0.19ms
768	0.22ms
1024	0.47ms
1536	0.66ms
1792	0.92ms
2048	1.53ms
2560	1.78ms
4096	2.1ms

The algorithms were implemented on an Intel(R) Core(TM) 2 Due CPU T7250 @ 2.00 GHz with 1024MB RAM, running the Microsoft Windows XP Professional edition version 2002 Service Pack 3.

While the 512-bit RSA-CRT is definitely the fastest among the ones shown, it is not the most secure, providing marginal security from an intensive attack. The 1792-bit and 2048-bit RSA-CRT reduces balance better than RSA method between speed and security and supposes to be used in critical situations since it offers maximum resistance to attacks. The result shows that the level of security between 2048 and 4096 bit reduces a suitable balance between speed and security in RSA-based-CRT according to the gained results.

8. CONCLUSIONS

Looking at the results shown in table (1), RSA-CRT is faster than the standard RSA in most cases. Feasibility analysis is done by comparing the time taken for RSA and RSA-based-CRT. It shows that when increasing key length then the decryption total time increases steadily. However, when comparing table (2) results with that of [9], which uses e=65536 as encryption exponent, it can be concluded that RSA encryption with e=3 is faster as compared with e=65536.

Practical comparisons have been made to RSA-CRT and that of the standard RSA for different key sizes. It can thus be said that software implementation of the RSA-CRT is the fastest.

REFERENCES

[1] Tom St Denis "Cryptography for Developers ", Syngress Publishing, Inc.2007

[2] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems", *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978.

[3] Jonathan B. Knudsen "Java Cryptography", First Edition May 1998 ISBN: 1-56592-402-9, 362 pages

[4] Carsten Siggaard "Using MatLab to aid the implementation of a fast RSA processor on a Xilinx FPGA", 2008.

[5] Amrit Tuladhar and George Kachergis "RSA Encryption with JAVA", March 8, 2006.

[6] Sarad A.V aka Data "Applications to Chinese Remainder Theorem", 2005.

[7] Johann.Groszschaedl "The Chinese Remainder Theorem and its Application in a High-Speed RSA Crypto Chip", 2001.

[8] A. Menezes, P. van Oorschot and S. Vanstone "Handbook of Applied Cryptography ", Y 1997 by CRC Press, Inc.

[9] G. Joseph and W.T. Penzhorn "Design and Implementation of Fast Multiplication Algorithms in Public Key Cryptosystems for Smart Cards", 2002.