**TOPIC : RSA CRYPTOSYSTEM**

**SOFTWARE PROJECT – CS 685**

Navya Sruthi Sunkarapalli

991796918

**TABLE OF CONTENTS**

**1. Introduction**

**2. RSA Cryptography**

**3. Algorithm**

3.1 Key Generation

3.2 Key Distribution

3.3 Encryption

3.4 Decryption

**4. Mathematical Computations in RSA**

4.1 Basic Number Theory

4.2 Extended Euclidean Algorithm

4.3 Modular Exponentiation

4.4 Multiplicative Inverse

4.5 Primality Test Algorithm

**5. Operation of RSA**

**6. Advantages and Disadvantages of RSA**

**7. Applications**

**8. Attacks**

**9. References**

**10. Appendix**

**1. Introduction**

Cryptography is the study of techniques for secure communication of private information from a distance. It is derived from two greek words – *kryptos*, “hidden”; and *graphein*, “writing”. Cryptography enables you to store sensitive information or transmit it across an insecure network (like the Internet), such that only those for whom it is intended can read and process it. There are five basics functions of cryptography : Confidentiality, Data Integrity, Authentication, Non – Repudiation and Key Exchange. Applications of cryptography include computer passwords, digital signatures, electronic commerce, Secure Socket Layer, Military Communications.

A cryptographic algorithm is a mathematical function used in the encryption and decryption process. In cryptography, we start with the unencrypted data, referred to as plaintext. The plaintext message is encrypted into ciphertext, which will in turn be decrypted back into plaintext. This process is carried out using a variable value known as key. Key is secret piece of data and there are two types of keys : public key and private key. A public key is made freely available between the sender and the receiver. The second private key is kept secret. The security of encrypted data is entirely dependent on two things : the secrecy of the key and the strength of the cryptographic algorithm.

The modern field of cryptography can be mainly classified into two cryptographic procedures :

1) Symmetric Cryptography.

2) Asymmetric Cryptography.

Symmetric key cryptography :

It is also called as Secret key or private key cryptosystem. This method uses a single key for both encryption and decryption referred to figure 1. It uses a secret key that can either be a number, a word or a string of random letters. Both the sender and the receipient should know the secret key that is used to encrypt and decrypt all the messages. Blowfish, AES (Advanced Encryption Standard), DES (Data Encryption Standard) are examples of symmetric encryption.

Asymmetric key cryptography:

It is also called as public key cryptosystem, which uses two different keys for encrypting and decrypting the information. The key pair consists of public and private key referred in figure 2. A public key is made freely available to anyone who might want to send you a message. The second private key is kept secret so that only you can know. The most popular asymmetric key encryption algorithms include RSA, DSA (Digital Signal Algorithm), Elliptical Curve Techniques. The steps describing how to run the code is present at the Appendix.

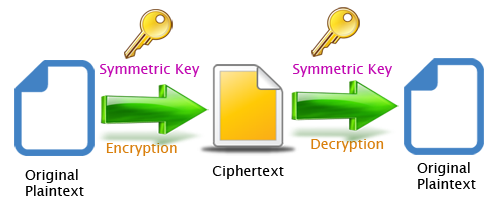


Figure 1 : Symmetric Cryptosystem

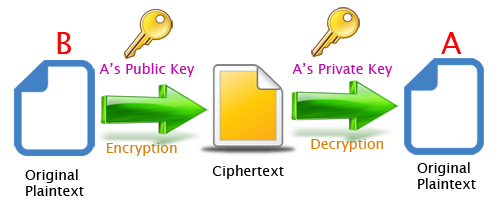


Figure 2 : Asymmetric Cryptosystem

**2. RSA Cryptography –**

RSA is the most widely used and tested algorithm used by modern computers to encrypt and decrypt messages. RSA stands for Rivest, Shamir and Adleman. In the year 1978, three professors from MIT named Ron Rivest, Adi Shamir and Leonard Adleman invented the RSA algorithm. It is based on a very simple number-theoretical idea and yet this algorithm has been able to resist all the cryptanalytic attacks. The security of RSA is based on the difficulty of factoring very large integers and length of the keys used. A user of RSA creates and then calculates the product of two large prime numbers, where the prime factors must be kept secret. RSA can be used for both public key encryption and digital signatures.

RSA algorithm uses two keys – public key and private key. The public key can be known by everyone and it is used for encrypting messages. The idea is based on the principle that messages that are encrypted with public key can only be decrypted in a reasonable amount of time using the private key.

The RSA algorithm involves four steps :

1) Key Generation

2) Key Distribution

3) Encryption

4) Decryption

**3. Algorithm**

**3.1 Key Generation :**

1. Generate two large random prime numbers, *p* and *q,* of approximately equal size where p and q are about 512 to 1024 bits.
2. Calculate the product of primes, .
3. Compute totient function ɸ(n), which is called as (phi). Totient function counts the positive integers upto a given integer *n*.

1. Choose an integer *e* , , such that gcd (e, phi) = 1.
2. Compute the secret exponent *d, ,* such that ed ≡ 1 ( mod phi).
3. The public key is (n, e) and the private key is (d, p, q)

**Terminology**

* *n* is known as the modulus.
* *e* is known as public or encryption exponent.
* *d* is known as private or decryption exponent.

**3.2 Key Distribution :**

RSA uses asymmetric key cryptosystem where different keys are used for encrypting and decrypting the messages. Public key is made freely available to anyone who want to send messages. Private key is unique and is only known to the intended receiver. The private key is used to decrypt the cipher text and therefore should not be shared with anyone.

**3.3 Encryption :**

The process of encoding a message or information in such a way that only intended and authorized parties can access it. In an encryption scheme, the intended message

referred to as plaintext, is encrypted using an encryption algorithm. The corresponding cipher text can be read only if it is decrypted. Encryption uses a public key pair (e,n) to encrypt a message. To encrypt messages using a particular key (e,n), follow the steps below:

1. Translate plain text message *M* into a sequence of integers.

* First translate each letter in the plain text into three-digit asci values i.e, if the ascii values are only 2 digits, add a zero at the beginning to make it three digits.

Eg:…

* Concatenate these three digits into a string of digits.

2. Divide the string into equal blocks such that value of each block does not exceed ‘*n*’ , where n is the product of primes.

**1 < *mi < n***where *i = 1,2,3,4,…..*

3. Now we have translated the plain text message *M* into a sequence of integer blocks *m1 , m2 , m3,……., mk* for some integer *k*.

4. Encryption proceeds by transforming each block *mi* to cipher text block *ci.*

5. Encryption of message ***m*** is :

**Sender A does the following:-**

1. Obtains the recipient B’s public key (n,e).
2. Represents the plaintext message as a positive integer *m,* 1< m <n
3. Computes the ciphertext c = me mod n
4. Sends the ciphertext c to B

**3.4 Decryption :**

The process of decoding the data which has been encrypted into a secret format is called as Decryption. In decryption, the system extracts the garbled data and transforms it to images or texts that are easily understandable by the reader and system. An authorized user can only decrypt the data because decryption requires password or a set of keys.

The plaintext message can be quickly recovered from cipher text message when the decryption key ***d***, an inverse of ***e modulo (p-1)(q-1***), is known using Modular Inverse Algorithm. To decrypt the message we use the private key pair (d,n). The decryption of cipher c will be :

**Decryption:**

Recipient B does the following:-

1. Uses his private key (d,n) to compute *m = cd mod n*
2. Extracts plaintext from the message representative *m.*

**4. Mathematical Computations in RSA –**

**4.1 Basic Number Theory :**

* Prime Number : A prime number is defined as a natural number that is greater than one, and that has no positive divisors other than one and itself.

Examples of first few prime numbers : 2, 3, 5, 7, 11,……

* Composite Number : A natural number greater than one, and that is not a prime number is called as composite number. A composite number is a positive integer that can be formed by multiplying two small positive integers.

Example : Integer 12 is a composite number because it is the product of two small integers 4 and 3. 4, 6, 8, 9, 10, 12, 14, 15,……. are composite numbers.

* Greatest Common Divisor : The gcd of two or more non-zero integers, is the largest positive integer that divides each of the integers. It is denoted by gcd (a, b).

Example : gcd (8 , 12) = 4 and gcd (15 , 25) = 5.

* Two numbers are said to be coprime if and only if their gcd = 1. Let m, n be two integers, m and n are coprime if and only if ***gcd (m , n) =* 1.**
* Euler’s totient Function : It is defined as the number of positive numbers that are relatively prime. The Euler’s phi function or ***totient*** *ɸ* (n) defines number of positive integers not exceeding ‘*n’* which are relatively prime to ‘*n’*, (ɸ(1) = 1 ).

For any prime p , ɸ(p) = p – 1

* If m and n are two coprime numbers, then ɸ(m)ɸ(n) = ɸ(mn).
* Modular Exponentiation : It is a type of exponentiation performed over a modulus. Given base *b*, exponent *e*, and modulus *m*, the modular exponentiation *c* is :

*c = be ( mod m).*

* Modular Inversion : The modular multiplicative inverse or modular inverse of an integer ***a modulo n***is an integer ‘x’ such that *ax ≡ 1 (* mod *n ).* This can also be written as *x = a-1 (* mod *n ) or x = (1/a)* mod *n*. A modular inverse exists if and only if gcd (a , n) = 1.

**4.2 Extended Euclidean Algorithm :**

The Extended Euclidean algorithm is an extension to the Euclidean algorithm which computes the coefficients of ***Bezout’s******identity*** in addition to the greatest common divisor for the integers *a* and *b.* The cofficients of *Bezout’s identity* are integers *x* and *y* such that :

The extended Euclidean algorithm is particularly used when integers *a* and *b* are coprime, since ***x*** is the modular multiplicative inverse of , and ***y*** is the modular multiplicative inverse of . In addition to this, the computation of modular multiplicative inverse is an essential step in the RSA public-key encryption method.

The standard Euclidean algorithm proceeds in a series of steps whose quotients are not used and only the remainders are kept. The output obtained as remainder in each step is used as an input for the next step. For the Extended Euclidean algorithm, the successive quotients are used. The standard Euclidean algorithm with two integers *a* and *b* as input, consists of computing a sequence ***q1, q2, …., qk***of quotientsand a sequence ***r1, r2, …, rk+1***of remainders such that -

**r0 = a**

**r1 = b**

**.**

**.**

**ri+1 = ri-1 - qi ri  and 0 ≤ ri+1 < |ri|**

**.**

**.**

**.**

It is the main property of the Euclidean division that the inequalities on the right define uniquely ***qi*** and ***ri+1*** from ***ri-1*** and ***ri***. The computation stops when the remainder ***rk+1*** reaches zero. At this point the greatest common divisor is the last non-zero remainder **rk.** The Extended Euclidean algorithm proceeds in a similar way, but adds two other additional sequences as follows :

**r0 = a r1 = b**

**s0 = 1 s1 = 0**

**t0 = 0 t1 = 1**

**. .**

**. .**

**ri+1 = ri-1 - qi ri  and 0 ≤ ri+1 < |ri| (this defines qi)**

**si+1 = si-1 – qisi**

**ti+1 = ti-1 - qiti**

The computation stops when rk+1 = 0 and gives :

* The ***Bezout’s coefficients*** are ***sk*** and ***tk*** , that is ***gcd (a , b) = rk = ask + btk***
* ***rk*** is the greatest common divisor of the input set ***a = r0*** and ***b = r1.***
* The quotients of *a* and *b* by their greatest common divisor are given by -***sk+1 =* ±** and ***tk+1* =**

If *a* and *b* are both positive and they satisfy the condition **gcd (a ,b) ≠ min (a, b)** , then

**|si| < and | ti | <**

for 0 ≤ *i* ≤ k . This means that the pair of *Bezout’s* coefficients obtained by the extended Euclidean algorithm is one of the two minimal pairs of *Bezout’s* coefficients. In addition it means that when *a* and *b* are representable integers, then the algorithm can be done without integer overflow.

**Pseudo Code:** *Extended Algorithm*

**function** *extended \_gcd(a,b)*

s := 0; old\_s := 1

t := 1; old\_t := 0

r := b; old\_r := a

**while** r ≠ 0

quotient : = old\_ r **div** r

(old\_r, r) := ( r, old\_r – quotient \* r )

(old\_s, s) := ( s, old\_s – quotient \* s )

(old\_t, t) := ( t, old\_t – quotient \* t )

**Output** “Bezout coefficients : ” , (old\_s, old\_t)

**Output** ”greatest common divisor : ” , old\_r

**Output** ”quotients by the gcd : ”, (t, s)

If either one of *a* or *b* is zero and the other is negative, the greatest common divisor *i.e* the output is negative, and all the other signs of the output must be changed.

**4.3 Modular Exponentiation :**

The type of exponentiation performed over a modulus is called as Modular Exponentiation.

**Operation** : The operation of modular exponentiation calculates the remainder when an integer ***b*** (the base) raised to the ***e*th** power (the exponent) – ***be*** is divided by a positive integer ***m*** (the modulus). In symbolic representation, given base *b*, exponent *e*, and modulus *m*, the modular exponentiation *C* is given by : ***C ≡ be (mod m)****.*

**Straightforward method** : This method involves calculating ***be*** directly and then take this number modulo ***m***. The time required to perform the exponentiation depends on the processor and the operating environment. The straightforward method requires *O(e)* multiplications to complete.

**Memory- efficient method** : This is a second method to compute the modular exponentiation and it requires more number of operations than the first method. The memory required for this method is substantially less and operations take less time than before. The end result is that the algorithm is faster.

In cryptography it is important that we find **bn mod m** efficiently, where **b, n, and m** are very large integers. It is not practical to first compute **bn** and then find the remainder when divided by **m** because **bn**will be a huge number. Instead we can use an algorithm that employs the binary expansion of the exponent **n.**

This algorithm makes use of the fact that, for two given integers *a* and *b,* the following equations are equivalent :

*c* mod *m* = mod *m*

c mod *m* = [(*a* mod *m*) . (*b* mod *m*)] mod *m*

The algorithm is as follows :

1. Set *c* = 1, *e’ = 0.*

2. Increment *e’* by 1.

3. Set *c = (b . c)* mod *m.*

4. If *e’* < *e*, goto step 2. Else *c* contains the correct solution to *c* ≡ *be* mod *m.*

**function** modular\_pow ( base, exponent, modulus )

**if** modulus = 1 **then return** 0

c := 1

**for** e\_prime = 1 **to** exponent

c := (c \* base) **mod** modulus

**return** c

**Right-to-left- binary method :**

A third method that drastically reduces the number of operations to perform modular exponentiation, by using memory efficiently. It is a combination of the previous method and it uses a more general principle called exponentiation by squaring (also known as *binary exponentiation*).

This method first requires that the exponent *e* to be converted to a binary notation. i.e, *e* can be written as :

***e = ai 2i***

According to the definition a*n-1 =* 1. In this notion, the length of *e* is *n* bits. *ai*  can take the value of 0 or 1 for any *i* such that 0 ≤ *i* ≤ *n*.

Now the value of *be* can be written as :

***be =***

**Pseudo code:** Modular Exponentiation:

**Procedure : function**  modular\_pow (base, exponent, modulus)

**if** modulus = 1 **then return 0**

Assert :: (modulus – 1) \* (modulus – 1) *does not overflow base*

result := 1

base := base **mod** modulus

**while** exponent > 0

if (exponent **mod** 2 == 1) :

result := (result *\** base) **mod** modulus

exponent := exponent >> 1

base := (base \* base) **mod** modulus

**return** result

When you enter into the loop for the first time, the code variable *base* is equivalent to *b*. The repeated squaring in third line of code ensures that after the completion of each loop, the variable *base* is equivalent to ***b 2^i mod m*** , where *i* is the number of times the loop has been iterated.

The time complexity or the running time for this algorithm is **O(log *exponent*)**.

This algorithm offers a substantial speed benefit when working with large values of *exponent*, when compared to other two algorithms whose time complexity is O(exponent).

**4.4 Multiplicative Inverse :**

This algorithm is used during the decryption process in RSA crypto-system. A **multiplicative inverse** or **reciprocal** for a number *x,* is denoted as 1 / *x* or *x-1,* is a number when multiplied by *x* yields product which is multiplicative identity, 1.

Example : A modular multiplicative inverse of an integer *a* is an integer *x* such that the product *ax* is congruent to 1, with respect to *the modulus m*.

***a . x ≡* 1 (mod m)**

In cryptography, the use of modular inversion permits some operations to be carried out in a more efficient way using less memory. In RSA algorithm, encryption and decryption of a message is done using a pair of keys in number format, that are multiplicative inverses over a selected *modulus.* One key is made public and can be used in the process of encryption, while the second key used in the decryption procedure, is kept hidden. Determining the hidden number from public number is considered to be computationally infeasible and this is what makes the RSA ensure privacy.

We use the Extended Euclidean algorithm for finding the modular multiplicative inverse.

* Given two integers *‘a’* and *‘m’*, the modular multiplicative inverse is an integer *‘x’* such that **a x ≡ 1 (mod m).**
* The multiplicative inverse of “*a modulo m*” exists if and only if *a* and *m* are relatively prime. i.e., if gcd (a , m) =1.
* The idea is to use **Extended Euclidean Algorithm** that takes two integers *‘a’* and *‘b’*, to find gcd and also compute ‘*x*’ and *‘y’* such that :

*a x + b y = gcd(a , b)*

* To find multiplicative inverse of *‘a’* under *‘m’*, we substitute *b=m* in the above formula. Since we know that *a* and *m* are relatively prime, we can replace value of gcd as 1.

*a x + m y =* 1

* If we take *modulo* *m* on both sides, we get

*a x* (mod m) *+ m y* (mod m) *=* 1 (mod m)

* We can remove the second term on left side as ‘*m y* (mod m)’ would be always 0 for an integer y.

*a x ≡* 1 (mod m)

Modular *m* division can be done only for numbers which are relatively prime to *m* and the division is actually carried out by multiplying by the inverse.

**4.5 Primality Test Algorithm :**

### In setting up the RSA Cryptosystem, it is necessary to generate large “random primes”. The way this is done is to first generate large random numbers, and then test them for primality. Some of the primality test algorithms used are Solovay-Strassen Algorithm, Fermat’s primality test, Frobenius primality test and Rabbin Miller’s test.

### Fermat’s Little Theorem : It states that if *p* is a prime number, then for any integer *a*, the number *ap – a* is an integer multiple of *p*. In modular arithematic notation, it is expressed as :

### *a p ≡ a* (mod *p*)

### If *a* is not divisible by *p*, Fermat’s little theorem is equivalent to the statement that *a p-1* - 1 is an integer multiple of *p.* In symbolic notation it is represented as -

### *a p-1* ≡ 1 (mod *p*)

### Fermat’s theorem : If we want to test whether *p* is prime or not, then pick random *a’s* not divisible by *p* to check whether the equality holds or not. If the equality does not hold for a value of *a*, then *p* is composite. This congruence is unlikely to hold for a random *a* if *p* is composite. Therefore if the equality holds for one or more values of *a*, then we say that *p* is probably prime.

The algorithm can be written as follows:

### Inputs : *n* : a value to test for primality, *n* > 3;

### *k* : a parameter that determines the number of times to test for primality.

### Output : composite if *n* is composite, otherwise *probably prime*

### Repeat *k* times :

### Pick a random number in the range [ 2, n-2 ]

### If *a n-1* !≡ 1 (mod n), then return *composite.*

### If composite is not returned : return *probably prime.*

### 5.1 Rabin Miller’s Algorithm :

### The Miller-Rabin test is the most widely used probabilistic primality test. This can be implemented using modular exponentiation by repeated squaring, the running time of this algorithm is O*(k log3 n)*, where *k* is the number of different values that we test. Similar to the Fermat and Solovay – Strassen tests, the Miller-Rabin relies on equality or a set of equalities that hold true for prime values, then checks whether they hold or not for a number that needs to be tested for primality.

### Algorithm :

Assume *p* is prime number and *p* > 2. Certainly 1 and −1 always yield 1 when squared modulo *p*. Call these [trivial](https://en.wikipedia.org/wiki/Trivial_(mathematics)) [square roots](https://en.wikipedia.org/wiki/Square_root) of 1.

Now we have to show that : *There are no nontrivial square roots of 1 modulo p.*

* Let us suppose that *x* is a square root of 1 modulo *p*. Then:

*x2* ≡ 1 (mod *p*)

(*x –* 1) (*x* + 1) ≡ 0 (mod *p*)

* Prime *p* divides the product (*x* − 1)(*x* + 1).
* According to [Euclid's lemma](https://en.wikipedia.org/wiki/Euclid%27s_lemma) if it divides one of the factors (*x* – 1) or (*x* + 1), it implies that *x* is congruent to either 1 or −1 modulo *p*.
* {\displaystyle x^{2}\equiv 1{\pmod {p}}}Let *‘n’* be a prime and odd, where *n* > 2. It follows that *n –* 1 is even and it can be written as 2s.*d* , where *s* and *d* are positive integers and *d* is odd.

***a d ≡* 1 (*mod n*) (or) *a 2 ^ r. d* ≡ -1 (*mod n*)** for some 0 ≤ r ≤ s – 1

* {\displaystyle (x-1)(x+1)\equiv 0{\pmod {p}}.}Using Fermat’s little theorem, for prime number n : ***a n-1* ≡ 1 (mod n)**
* TTTThe Rabin-Miller’s primality test is based on the contrapositive of the above claim. i.e., if we find an integer *‘a’* such that –

***a d* !≡ 1 (*mod n*) and *a 2^r.d* !≡ -1 (*mod n*)** for all 0 *≤ r* ≤ s -1

then *n* is not prime.

### Pseudocode :

### Input #1 : n > 3, an odd integer to be tested for primality;

### Input #2 : *k*, a parameter that determines the accuracy of the test

**Output :** *composite* if *n* is composite, otherwise *probably prime*

### 

write *n –* 1 as 2r.*d* with *d* odd by factoring powers of 2 from *n –* 1

Loop : **repeat** *k*  times :

pick a random integer *a* in the range [ 2, n-2 ]

x ← *a d* mod *n*

**if** x = 1 or x = *n* – 1 **then**

**continue** Loop

**repeat** *r* *–* 1 times :

x ← x2 mod *n*

**if** x = 1 **then**

**return** *composite*

**if** x = n – 1 **then**

**continue** Loop

**return** *composite*

**return** *probably prime*

**5. Operation of RSA :**

1. Generate two large random primes *p* and *q*.
2. Calculate *n* such that *n = pq*
3. Calculate *phi*, such that ɸ(n) = ( *p - 1*)( *q - 1*)
4. Choose random encryption key e, such that

0 < e < ɸ(n) and gcd (e, ɸ(n)) =1

1. Calculate decryption key d using d = e-1mod ɸ(n)
2. Generate cipher text c using c = me mod n, 1<m<n
3. Decrypt the cipher text c using m = cd mod n, to get original message m

**6. Advantages and Disadvantages of RSA**

**6.1 Advantages :**

* RSA algorithm is safe and secure for its users because it uses complex mathematics computations.
* This algorithm uses public key to encrypt the data and the key is known to everyone, therefore it is easy to share the public key.
* RSA ensures secure transmission of data since it is hard to crack as it involves factorization of large prime numbers which are difficult to factorize.

**6.2 Disadvantages :**

* RSA algorithm is very slow when large data needs to be encrypted by the same computer.
* This algorithm requires a third party to verify the reliability of public keys.

If the factorization of *n* ( *p \* q*) is possible then the whole RSA algorithm is compromised.

**7. Applications of RSA**

**1) Strong Authentication :** The process by which identity of users are verified Authentication. Authentication of users plays an important role in the identity management process. The different methods used for authentication are user-name, password, biometric pattern recognition.

**2) Tamperproof hardware :** This is one of the major problem in any authentication system that uses cryptography to store the key safely. This problem arises when user’s desktop computer is shared among several users, which cannot be trusted or in an attacker model which allows an intruder to install any malware on the user’s desktop computer.

**3) One time password generating tokens :** Password generators use cryptography to generate session passwords, also called as one-time-password or OTP. The password generator tokens have a clock whose value is hashed and encrypted using a key shared with the verifying party. The verifying party has a clock that is synchronized with the token’s clock.

**4) Internet :** RSA cryptography is used during transmission of sensitive data through the internet which includes credit card information during online transactions.

**5) Digital Signatures :** A digital signature is a mathematical scheme which describes the authenticity of digital messages or documents. A valid digital signature gives a reason for the recipient to believe that the message was created by a known sender and the sender cannot deny having sent the message.

**8. Attacks :**

**1. Factoring Large Integers :** The first attack on RSA public key pair (N, e) is factoring the modulus N. The factoring of modulus is considered as a *brute-force* attack on RSA. If given the factorization of *N,* an attacker can easily construct ɸ(*N*), from which the decryption exponent *d = e -1 mod ɸ*(*N*) can be found. The factoring algorithms are steadily improving, but the current state is still far away from posting a threat to the security of RSA when it is used properly.

**2. Low Private Exponent :** To reduce the running time involve in decryption, one may wish to use a small value of *d* rather than selecting a random value. Since modular exponentiation takes time, a small value of *d* can improve the performance. But a smaller value of *d* results in a total break of the cryptosystem.

**3.** **Common Modulus :** To avoid generating a different modulus each time (*N = pq*) for each user, one may wish to fix *N*. Then the same value of *N* is used by all the users. A trusted central authority could provide user *i* with a unique pair (*ei*, *di)* from which a user *i* forms a secret key (*N,di*) and a public key (*N,ei*).

**4.** **Low public Exponent :** In order to reduce signature-verification time or encryption time, a small public exponent *‘e’* is used by the user. The smallest possible value available for *e* is 3, but the value of e = 216 + 1 is recommended to defeat certain attacks. In general when a random *e < ɸ*(N) is used, approximately 1000 multiplications are required for signature verification. When the value of 216 + 1 is used, only 17 multiplications are required.

**References**

[1] Douglas Stinson . *Cryptography : Theory and Practise, Second Edition.*

CRC/ C&H, 2nd edition, 2002

[2] Kenneth H. Rosen : *Discrete Mathematics and its* *Applications.*

[3]RSA Algorithm : *DI Management*

<https://www.di-mgt.com.au/rsa_alg.html#note2>

[4] Wikipedia - <https://simple.wikipedia.org/wiki/RSA_(algorithm)>

[5] <https://www.cs.unibo.it/babaoglu/courses/security/resources/documents/int> ro-to-crypto.pdf

[6]Attacks on RSA Crypto-system :

https://crypto.stanford.edu/~dabo/papers/RSA-survey.pdf

**APPENDIX**

**Steps for running Code :**

1) Download or install any IDE for java like – Eclipse, NetBeans, IntelliJ Idea.

2) Start Eclipse.

3) Create a new Java Project : Algorithm

4) Create a new Java Class : RSA.java

5) A Java editor for RSA.java will open.

6) Save the code using Ctrl + S.

7) Click the “Run” button in the toolbar.

8) You will be prompted to enter the input in the console.

9) Enter a text message and you get the required output.