

Rivest–Shamir–Adleman (RSA) (ICP)

Prof. Subhashis Banerjee
Niranjan Rajesh
Veda, Chavi, Diya, Satyakin, Aditi, Esha

By: Satwik Wats

Submission date: May 2, 2022

UG 24

Introduction

Ron Rivest, Adi Shamir and Leonard Adleman in the year 1977 presented an algorithm that would ensure secure data transmission. They named it as the RSA algorithm, it is a public key cryptosystem. It uses two large prime numbers to encrypt a message and with the help of the same two number it decrypts the message. This whole situation is possible because of one big practical difficulty that we face, the factorization of product of two large prime number. If someone figures out, or factors the product of the prime number then the RSA will break down. So, how this work is in the following way: the sender and reciever both have a public and a private key. The public key is given to everyone publicly, but the private key is just known to these two. And the encoded message will only get revealed by the private key, they both have to be inverse functions.

$$Cipher = P_A(M) \text{ } M \text{ is the message and } P_A \text{ is the public key}$$

$$M = S_A(Cipher) \text{ } S_A \text{ is the private key and } M \text{ is the message}$$

This idea of encryption and decryption involves many other basic concepts that we will cover in this report.

Developing the Algorithm

The basic idea behind developing RSA involves a certain sequence of steps. We need to have a public and a private key,

- The very first step is to find two large prime numbers, around 1024-2048 bits that is more than 300 digits.
- Next we compute the product of our two prime numbers. Let the prime numbers be p and q and their product as ϕ , so $\phi = p \times q$.

- Now our next step is to generate a public key so we select a small odd integer e that is relatively prime to $\Phi = (p - 1) \times (q - 1)$
- We have obtained our public key now that is $P = (e, \phi)$
- Now to generate private key we have to look for a value of d that satisfies the condition

$$ed \equiv 1 \mod \Phi$$

- As both should be inverse functions, so the transformation of a message M associated with the public key $P = (e, n)$ is,

$$P(M) = M^e \mod n$$

The transformation of the cipher C associated with the private key $S = (d, n)$ is,

$$S(C) = C^d \mod n$$

Understanding the reason behind the working of RSA

We know the RSA works, but is there rigorous mathematical proof behind its working? No, the RSA works on a very basic concept. We will build on it using few other simpler concepts first the primality testing. For testing that the number we choose are prime or not we use the Fermat's little theorem to show that the number are indeed prime.

The theorem says that if p is a prime then,

$$a^{p-1} \equiv 1 \mod p \quad (1)$$

The next theorem to look at is a variant of Chinese-remainder theorem, which states that if n_1, n_2, \dots, n_k are pair wise relatively prime and $n = n_1 n_2 \dots n_k$ then for all integers x and a ,

$$x \equiv a \mod n_i \quad (2)$$

if and only if,

$$x \equiv a \mod n \quad (3)$$

Now that we have two of the basic theorems, we will try to understand the working of RSA.

As we know that both of our public P , and private S key are multiplicative inverses modulo Φ then we should get

$$M^{ed} \equiv M \mod n$$

Proof of this statement has been referred directly from ICP course handout.

As we know that

$$ed = 1 + km = 1 + k(p - 1)(q - 1)$$

Thus if $M \neq 0 \mod p$ then we have,

$$M^{ed} \equiv M(M^{(p-1)k(q-1)}) \mod p \quad (4)$$

$$M^{ed} \equiv M(1)^{k(q-1)} \mod p \equiv M \mod p$$

We get $m^{p-1} \equiv 1 \mod p$ from equation (1)

Now if $M \equiv 0 \mod p$ we have,

$$M^{ed} \equiv M \mod p$$

Also,

$$M^{ed} \equiv M \mod q$$

So, we get

$$M^{ed} \equiv M \mod n$$

Steps for coding

We now know few properties of e and d , so we will try to utilize these and get them printed from a function in our computers. I will be using python to implement the code.

We use the concept of gcd to compute e , we can randomly pick e from a range $0 < e < \Phi$ then confirm if $\gcd(e, \Phi) = 1$, if not then choose again otherwise take that as the value of e . This is known as Euclid's gcd algorithm.

We can extend this algorithm to compute d , this is called Euclid's extended algorithm. Here calculate x and y such that

$$\gcd(e, \Phi) = 1 = ex + \Phi y$$

then we take d as x This is how we define values for our public and private key. From our steps we have the prime numbers and the keys now we just need to change the string values we enter as message into ASCII characters so we can encrypt it. For that we use the simple python function of *ord* to encrypt from strings to numbers and *chr* to decrypt from numbers to string.

Using my own packages for RSA

Implementing python code for RSA using my own multiplication, division, exponent, power function.

Multiplication

To make a function that can multiply is very simple, we just have to do repetitive additions. But, this algorithm will take a long time to encrypt even a 3 digit number. It is an order n^2 algorithm in terms of time complexity. To get this out we use the Karatsuba's multiplication algorithm, which reduces the number of multiplication and therefore reducing the time complexity.

We know how high school multiplication works, it is just repetitive addition with shift in places. So, to make this more efficient we reduce the multiplication. Take two numbers x and y and both of them are in some base B . So what the algorithm says is that if there exists a positive integer m less than n , then we can write the numbers as,

$$x = x_1 B^m + x_0$$

and

$$y = y_1 B^m + y_0$$

So, now the product of the numbers are

$$xy = (x_1B^m + x_0)(y_1B^m + y_0)$$

We assume

$$x_1y_1 = z_2$$

$$x_1y_0 + x_0y_1 = z_1$$

and

$$z_0 = x_0y_0$$

So, this gives us the function as

$$z_2B^{2m} + z_1B^m + z_0$$

These functions require four separate multiplication, but Karatsuba reduced them to 3 by applying just few extra addition.

$$z_1 = (x_1 + x_0)(y_1 + y_0) - x_1y_1 - x_0y_0$$

So, now instead of initial value of z_1 where we had to multiply four times we only do it three times.

This doesn't reduce the efficiency from $O(n^2)$ to $O(n)$, but surely reduces it to $O(n^{1.58})$ which is much better than our previous high school method. Now we implement the algorithm using python.

#multi is my defined function that will perform karatsuba multiplication

```
def multi(x,y):
```

```
    n1=len(str(x))  ## determining length of x and y
```

```
    n2=len(str(y))
```

```
    if n1==1 or n2==1: #Assert: If n1 or n2 are equal to
        zero then perform simple multiplication
        return x*y
```

```
    else:
```

```
        n=max(n1,n2)//2 #We split them in half
```

```
        a=x//10**n # 10**n becomes my new base
```

```
        b=x%10**n # mod of this plus above value will return my entire function.
        ## Example: 12345= 12*10**3+345
```

```

c=y//10**n ## repeat the same steps
d=y%10**n

ac=multi(a,c) #Now we again recursively compute ac and bd then adbc

bd=multi(b,d)

adbc=multi(a+b,c+d)-ac-bd #reducing 4 multiplication to 3 multiplication

return ac*10**(2*n)+adbc*10**n+bd ## return function in its normal form

```

Modular Exponentiation function

The next important algorithm in RSA structure is the fast exponentiation algorithm. We need to calculate power mod values in order to have fast encryption and decryption. For this algorithm we use the idea of odd and even separation, here we see if the power value is even we perform simple one step multiplication and take mod of that function and half the power value.

If the power value continues to be even we repeat the steps, but if it is odd we multiply it with a carry value of $i = 0$ and take new i value as

$$i = (x \times i) \bmod n$$

and then reduce the value of n by 1. The algorithm takes time complexity of $O(\log_2 y)$ as the loop runs through $y/2$ every time.

```

def expo(x, y, n):

    if (x == 0):
        return 0
    if (y == 0):
        return 1

    i = 0
    if (y % 2 == 0):
        i = expo(x, y // 2, n)
        i = (i*i) % n

    else:
        i = x % n
        i = (i * expo(x, y - 1, n) % n) % n
    return ((i + n) % n)

```

Power function

To compute exponent function we will divide the algorithm in different steps, first we can divide the power in half for even numbers and for odd we can still divide it in half but we will have to do an extra multiplication.

$$2^{10} = (2^5)^2$$

and as,

$$2^5 = ((2^2)^2 * 2)$$

So we use the same logic in our algorithm. The algorithm takes time complexity of $O(\log_2 n)$ as the loop runs through $n/2$ every time and goes through the number .

```
def powe(x, n):
##Computing the power of a number with base x and power n

    if n == 0: ## if n=0 we return 1, as 2^0=1
        return 1

    p = powe(x, n // 2) # WE use the dividing technique

    if n & 1:      #if the number is odd we do one extra step
        return x * p * p

    #if the number is odd we continue with our even mulitplication steps
    return p * p
```

Division Algorithm

We will use a simple algorithm to determine the remainder and quotient of x/y . The function will take $T(x) = T(x \text{ div } 2) + O(1); T(0) = 0$, which gives us $T(x) = O(\log_2 x)$.

```
def divide(x,y):
    if x == 0: ##if x=0 return 0
        return (0,0)

    elif y==0: ##if y=0 return 'Error'
        print('Error')

    elif x==0 and y==0: ##if both x and y are 0 return 0
        print('Error')

    #For 0 ≤ n < x divide (n,y) to return (q,r) such that n=qy+r
    else:
        (q,r) = divide(x // 2, y)
        q1 = 2*q
        r1 = 2*r
        ## If x is odd then divide(x//2, y) and return r=2*q+1
        if (x%2 == 1):
            r2 = r1+1
            if (r2 < y): ##if r2 is less than y then return (2*q,2*q+1)
                return (q1,r2)
            else: #if r2 ≥ y then return (q1+1,r2 - y = 2r + 1 - y)
                return (q1+1,r2-y)
        ##if the x is odd
```

```

        #then algorithm returns  $x = (q1 + 1)y + (r1 - y)$ 
    else:
        if (r1%y):
            return (q1,r1)
        else:
            return(q1+1,r1-y)
def rem(d) : #This function finds the mod value of (x,y)
    return d[1]
rem(divide(5,2))

```

Getting started with the code

The first step is to confirm the numbers we use are prime or not.

```

def isPrime(a):
    if a==2:
        return True
    i=0
    while i<5:
        p=np.random.randint(2,20)
        if expo(p,a-1,a)!=1:
            return False
        else:
            return True
    i+=1

```

After doing the primality testing, we will develop the functions for gcd, extended gcd and multiplicative inverse.

GCD

```

def gcd(e, phi):

    while phi != 0:
        e, phi = phi, e%phi
    return e

```

Time complexity: $O(\log N)$

Space complexity: $O(1)$

Extended GCD

```

def euclidextalgo(a,b): #x = y1 - b/a * x1 ##y = x1
    de,re=divide(a,b) #de=div and re=mod
    if(a%b==0):
        return(b,0,1)
    else:
        g,x,t = euclidextalgo(b,re) #recursive call of gcd(b,a%b)
        f=multi((de),t)
        x =x-f

```

```

    return(g,t,x)
Time complexity: O(log N)
Space complexity: O(1)

```

Multiplicative Inverse

```

def mulinv(e,phi):
    g,x,=euclidextalgo(e,phi)
    if(g!=1):
        return None
    else:
        if(x<0):
            (x,x,rem(divide(x,phi)))
        elif(x<0):
            print(x)
        return rem(divide(x,phi))

```

Developing the private and public key

```

def priv“pub(p,q):
    n=multi(p,q)
    phi=multi((p-1),(q-1))
    #We randomly generate e value and check its gcd if it is true we
    ##take it and continue further calculation ow repeat the step
    e = random.randrange(1, 1000000)

    g = gcd(e, phi)
    while g != 1:
        e = random.randrange(1, 1000000)
        g = gcd(e, phi)
    d=mulinv(e,phi)
    return (d,n),(e,n)

```

Generating encryption and decryption algorithm

```

def encrypt(pub“key, message):
    d, n = pub“key
    secret = [expo(ord(char), d,n) for char in message]
    return secret

def decrypt(pri“key, secret):
    e, n = pri“key
    message = [chr(expo(char,e,n)) for char in secret]

    return ''. join(message)

```

This method of RSA using my own package reduces the efficiency of the algorithm. The major issue is using a better division algorithm to calculate *mod* fast. The above function can produce *p* and *q* value of 13 digits, but If we use

the inbuilt *mod* function we can use p and q value of 200 digits each. And, we can make it much better if we use multiplication, division, and power from python and implement them then we get p and q values of more than 400 digits. This gives RSA an overall time complexity for encryption as $O(b^2)$ and for decryption $O(nb^2)$, where the public and private modulus n has b bits

Implementing RSA using python big-num package

GCD

```
def gcd(e, phi):
    while phi != 0:
        e, phi = phi, e%phi
    return e
```

Extended GCD

```
def euclidextalgo(a,b): #x = y1 - b/a * x1 ##y = x1
    if(a%b==0):
        return(b,0,1)
    else:
        gcd,x,t = euclidextalgo(b,a%b) #recursive call of gcd(b,a%b)

        x =x-((a//b)*t)
        return(gcd,t,x)
```

Multiplicative Inverse

```
def mulinv(e,phi):
    gcd,x=euclidextalgo(e,phi)
    if(gcd!=1):
        return None
    else:
        if(x<0):
            (x,x,x%phi)
        elif(x<0):
            print(x)
        return x%phi
```

Developing private and public key

```
def privpub(p,q):
    n=(p*q)
    phi=((p-1)*(q-1))
    e = random.randrange(1, phi)

    g = gcd(e, phi)
    while g != 1:
        e = random.randrange(1, phi)
```

```

g = gcd(e, phi)
d=mulinv(e,phi)
return (d,n),(e,n)

```

Generating encryption and decryption algorithm

```

def encrypt(pubkey, message):
d, n = pubkey
secret = [pow(ord(char), d,n) for char in message]
return secret

```

```

def decrypt(prikey, secret):
e, n = prikey
message = [chr(pow(char,e,n)) for char in secret]

return ''. join(message)

```

Snippet of the code

```

print(p)

3081182531755546671351585109024083090894147180347198701414827785710399687623942014571042070151668512107765569277179926432893455
17568096984095753459283671417697212321993649727490211297520848139563191406145071110839598369684120344426632061255184913
766554630497856944946641386235640402222805249927191398576301483456361766683502377884609246327106123587480877879625844630484304
558076090676004060985221767048764896715823905408211184884054254075039
8181718953101240639077027211530992172272081709424335648522766706378881257773112125023373490965109382998846420212550557998817124
45851801355821457104411540809862291675100101919667181128016927644037321657435084780614297139103514344814093510944868704450
2466974044870219386151847268355448230721312149761550048136847114865990484771963324074886620995428783599105249815120245465189
3066051964648138933952210576596475388076923441135624876745613458232829886424171721892717509715679403401089906670457651369425
40274029955961225157703829996340942950039116309726641287174618042071362165850923841192180234765589801668454912059220
8251082373529936564088159930423780379958856872538555115813353824708101262577736271243631363638664563496269468847178758788
6469420593937155421067291990437227958397255310833028148168693390962089898956069871633006622172467276840741049108928232687
46767080074348

]: generateLargePrime(1500)
#q=13
print(q)
n=(p*q)
phi=(p-1)*(q-1)
print(phi)

2720290299817340927907535836843924708774079186022188401508630813248005609535984629729001619109132653511706183300617373955958946
17568096984095753459283671417697212321993649727490211297520848139563191406145071110839598369684120344426632061255184913
766554630497856944946641386235640402222805249927191398576301483456361766683502377884609246327106123587480877879625844630484304
558076090676004060985221767048764896715823905408211184884054254075039
8181718953101240639077027211530992172272081709424335648522766706378881257773112125023373490965109382998846420212550557998817124
45851801355821457104411540809862291675100101919667181128016927644037321657435084780614297139103514344814093510944868704450
2466974044870219386151847268355448230721312149761550048136847114865990484771963324074886620995428783599105249815120245465189
3066051964648138933952210576596475388076923441135624876745613458232829886424171721892717509715679403401089906670457651369425
40274029955961225157703829996340942950039116309726641287174618042071362165850923841192180234765589801668454912059220
8251082373529936564088159930423780379958856872538555115813353824708101262577736271243631363638664563496269468847178758788
6469420593937155421067291990437227958397255310833028148168693390962089898956069871633006622172467276840741049108928232687
46767080074348

]: print(len(str(p)))

452

```

Figure 1: Generating very large prime number around 400 digits.

```

In [27]: public,private=priv_pub(p,q)
print(public,private)

(62717899418512026141431790551751605711803164902980741345430154953987438872363605500727153926349731433933504117643116801226966
1963973042475360110261110974404123819979600027715867314012851045254090649080785110829002717863953085268204905628734324119670
6846591496001771315175124093811849494529673648771318915163018056873189446622512125130980966109039624172817708062156440211
10760792765828115500569313845380166827909992131104063935604600808745964124490284384539708284629280426707699040518557150987525
606791729146258365627155841380149624799573597796272730152200599077621445344202638195523770055424626824758401617371107052022
0515006171413549984642067594814264862891856214518253806649115396177462117308720353021124890552123246481918851219437554
689040945991887140525771840044149620204175272935646826751086040617032527310896180754133298443529051364899312649560720418035
331028597255847, 83817189531012406390770272115309921722720817094243356485227667063788812577731121250233734909651093829988464292
125505579988172445818013558214571044115408098622916751001019196671811280169276440373216574350847806142971391035143448140935
1094486870444502669740448702193861518472683554482307213121497615500481368471148659904847719633240748866209954287835991052
4081512024546518930660519646481389339522105765964753880769234411356248767456134582344100337072901085264321810262041810567
2930704590797173836091954001809411804348152953577023065968588133504061612089598203655678187048720034079438335991728771
0723045032912950685126159310074069617328517700257649866871867397112497101495994087804552465870695272405543965722739262261779552
6206208162755225861304925807706257021003245684719481281555622533483498745961425313528610985978197688891813834722690902453980
14221988111004813684146309565809) (73474100805201218615377927456532592148828740882721198109404049762395985081548142459235
109304604918285866301421162504165456292838214707000580967642511090412110996315285362876104458955610014317067002658887728203
439877019870176739290131915620736623449984639744836012038758341146623909108759364705629581055706844499776717615796309106173
6057683777205613957762156848176190808045047481973442779640909554301326260611704531390216970670804359525087461049622811366393
277250456567851542310543340925527248502192263376314903997717248207173373859251624520960879591759166446781675048589613452475
98007968139971073660468475364939295749003055711343513304648717176912321817616444725846449193134888961471801105217919640558924
7084710442164078781540926692081168071476237249529157816724220959650014146623706155031857473222329843256430627939670474338
430290539528374138855138451106828456429860779563, 83817189531012406390770272115309921722720817094243356485227667063788812577731
12125023373490965109382998846429212550557998817244585180135582145710441154080986229167510010191966718112801692764403732165743
5084708614297139103514344814093510944868704445026697404487021938615184726835544823072131214976155004813684711486599048477
196332407488662099542878359910524981512024546518930660519646481389339522105765964753880769234411356248767456134582344100337
07290106264342181026204181056729387094590079717838609195400190411884348152953527702306659685881335046161326095820365567
918570848720034074503359917287710723045032912950685126159310074069617328517700257649866871867397112497101495994087804552465870695272405543965722739262261779552
6065724055439657273926226177955262062086725525861304925807706257021003245684719481281555622533483498745961425313528610985
9781976889183384722690824539801422198811100481368414630956809)

```

Figure 2: Generating very large public and private keys for encrypting and decrypting.

```

In [13]: msg="Is this okay?" ##Enter your message!!
         value=encrypt(public,msg)
         print(value)

[116874290779447766230147, 60086293322882751421929, 144681021746757688711077, 29907140276024712988260, 2426740204038441297205,
97357695785429658837675, 60086293322882751421929, 144681021746757688711077, 132054475200072536111682, 96787653506413354936773,
148539308807553617047051, 70578380251355897928934, 11983186138701764900973]

In [14]: f=input('Do you wanna decrypt your message? If yes type "yes" otherwise "no": ')
         if f=="yes":
             print(decrypt(private,value))
         elif f=="no":
             print("Your loss!!")
         else:
             print('Wrong option bud!!')

Do you wanna decrypt your message? If yes type "yes" otherwise "no": yes
Is this okay?

```

Figure 3: Encrypting and decrypting a message using 13 digit prime number RSA code.

References

- https://en.wikipedia.org/wiki/Fermat_primality_test: :text=adds
- <https://www.codespeedy.com/fast-exponentiation-python/>
- <https://pythonandr.com/2015/10/13/karatsuba-multiplication-algorithm-python-code/>
- <https://www.codespeedy.com/modular-exponentiation-in-cpp-power-in-modular/>
- [https://en.wikipedia.org/wiki/RSA_\(cryptosystem\)](https://en.wikipedia.org/wiki/RSA_(cryptosystem))
- <https://www.rsa.com/>
- FinalProject_CS:1101 handout
- <https://crypto.stackexchange.com/questions/6164/how-do-i-derive-the-time-complexity-of-encryption-and-decryption-based-on-modula/61946194>
- https://en.wikipedia.org/wiki/Modular_multiplicative_inverse
- <https://cp-algorithms.com/algebra/module-inverse.html>