

Tool used :

<https://cse29-iiith.vlabs.ac.in/exp>

## RSA CRYPTOSYSTEM

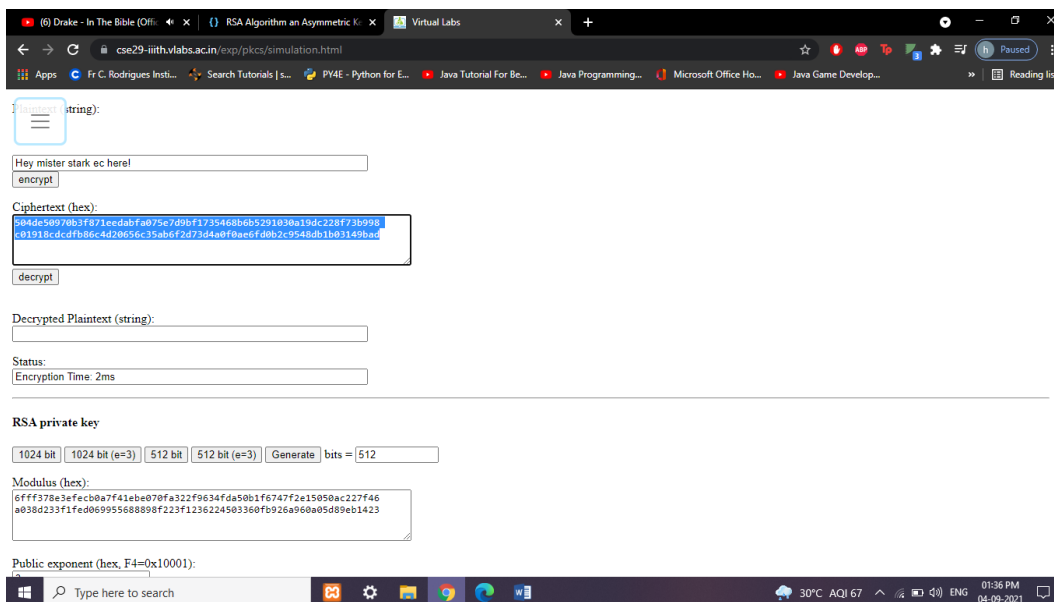
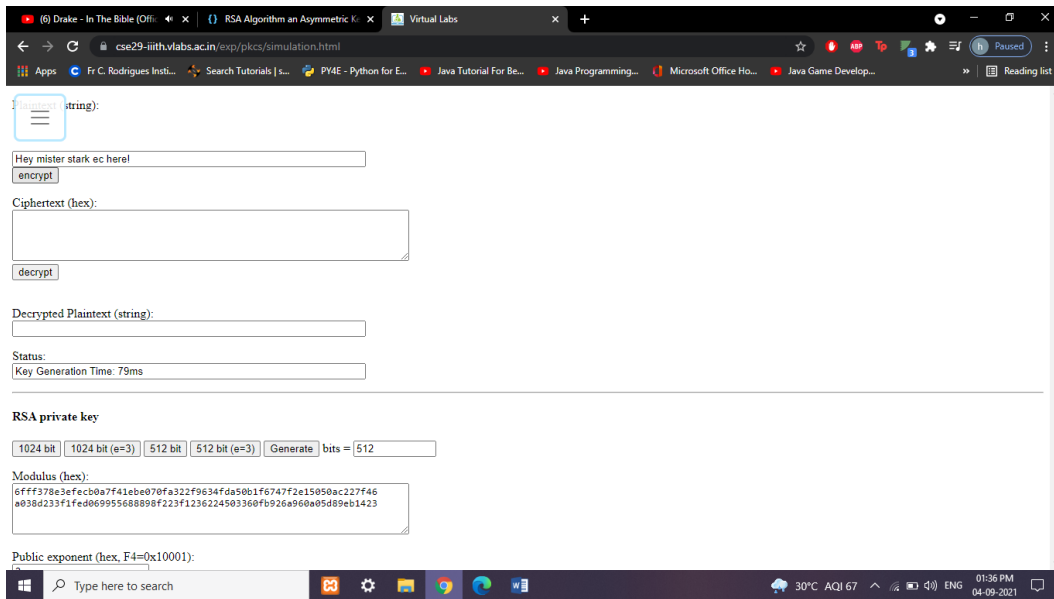
**Step 1 :** Enter the input text to be encrypted in the 'Plaintext' area

The screenshot shows the RSA Algorithm simulation interface. The 'Plaintext (string):' field contains 'Hey mister stark ec here!'. The 'encrypt' button is highlighted. The 'Ciphertext (hex):' field is empty. The 'Decrypted Plaintext (string):' field is empty. The 'Status:' field is empty. The 'RSA private key' section shows '1024 bit', '1024 bit (e=3)', '512 bit', and '512 bit (e=3)' buttons, with 'bits = 512' displayed. The 'Modulus (hex):' field is empty. The 'Public exponent (hex, F4=0x10001):' field is empty. The browser address bar shows 'cse29-iiith.vlabs.ac.in/exp/pkcs/simulation.html'.

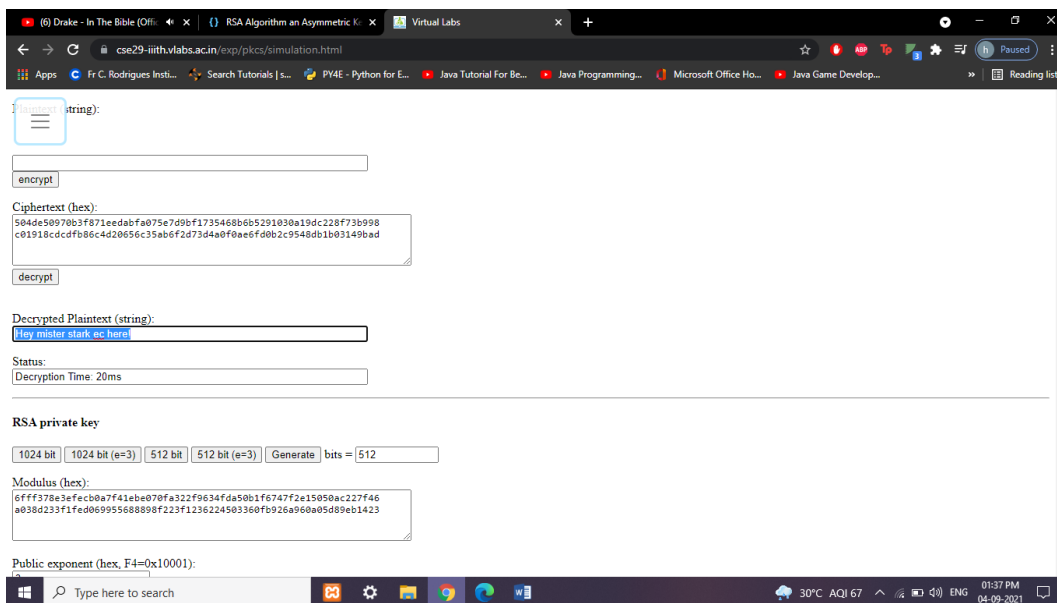
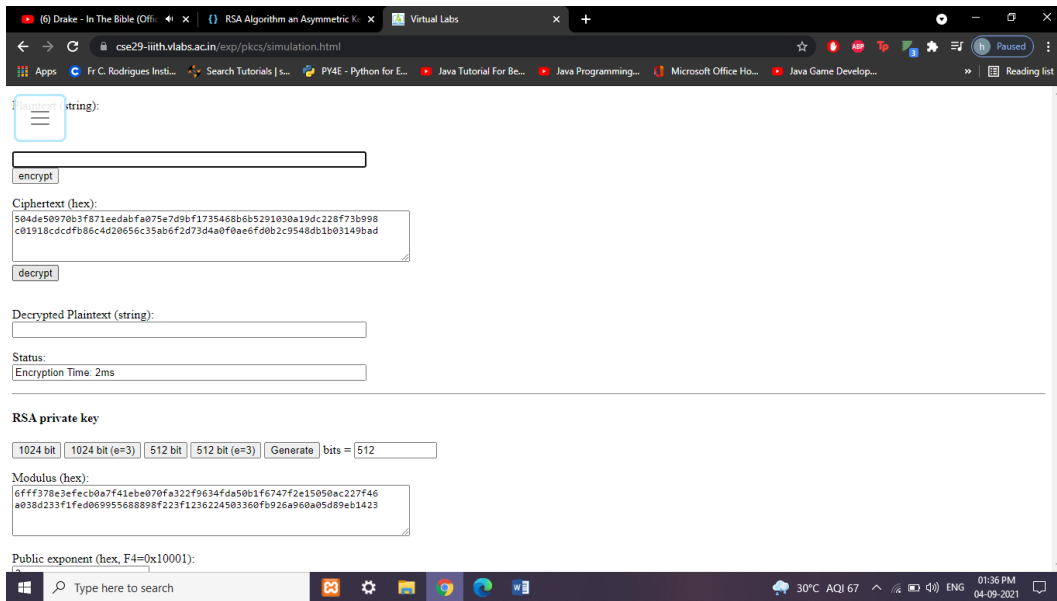
**Step 2 :** Select key size of public key from **RSA Private key** section by clicking on one of the key button.

The screenshot shows the RSA Algorithm simulation interface. The 'Plaintext (string):' field is empty. The 'Status:' field shows 'Key Generation Time: 79ms'. The 'RSA private key' section shows '1024 bit', '1024 bit (e=3)', '512 bit', and '512 bit (e=3)' buttons, with 'Generate' and 'bits = 512' displayed. The 'Modulus (hex):' field contains a long hexadecimal string. The 'Public exponent (hex, F4=0x10001):' field contains '3'. The 'Private exponent (hex):' field contains a long hexadecimal string. The 'P (hex):' field contains a long hexadecimal string. The 'Q (hex):' field contains a long hexadecimal string. The browser address bar shows 'cse29-iiith.vlabs.ac.in/exp/pkcs/simulation.html'.

**Step 3 :** Click on **encrypt** button to generate a ciphertext.



Conversely we can decrypt with the same steps , i.e. put the same ciphertext in the cipher box then the generated key for the encryption is used to decrypt the cipher.



## CODE:

```
import math

print("RSA ENCRYPTOR/DECRYPTOR")

print(" ")

#Input Prime Numbers

print("PLEASE ENTER THE 'p' AND 'q' VALUES BELOW:")

p = int(input("Enter a prime number for p: "))

q = int(input("Enter a prime number for q: "))

print(" ")
```

```

#Check if Input's are Prime
def prime_check(a):
    if(a==2):
        return True
    elif((a<2) or ((a%2)==0)):
        return False
    elif(a>2):
        for i in range(2,a):
            if not(a%i):
                return False
        return True
check_p = prime_check(p)
check_q = prime_check(q)
while(((check_p==False) or (check_q==False))):
    p = int(input("Enter a prime number for p: "))
    q = int(input("Enter a prime number for q: "))
    check_p = prime_check(p)
    check_q = prime_check(q)
#RSA Modulus
n = p * q
print("RSA Modulus(n) is:",n)
#Eulers Toitent
r= (p-1)*(q-1)
print("Eulers Toitent(r) is:",r)
print(" ")
#GCD for 'e' cal
def egcd(e,r):
    while(r!=0):
        e,r=r,e%r
    return e
#Euclid's Algorithm

```

```

def eugcd(e,r):
    for i in range(1,r):
        while(e!=0):
            a,b=r//e,r%e
            if(b!=0):
                print("%d = %d*(%d) + %d"%(r,a,e,b))
            r=e
            e=b

#Extended Euclidean Algorithm
def eea(a,b):
    if(a%b==0):
        return(b,0,1)
    else:
        gcd,s,t = eea(b,a%b)
        s = s-((a//b) * t)
        print("%d = %d*(%d) + (%d)*(%d)"%(gcd,a,t,s,b))
        return(gcd,t,s)

#Multiplicative Inverse
def mult_inv(e,r):
    gcd,s,_=eea(e,r)
    if(gcd!=1):
        return None
    else:
        if(s<0):
            print("s=%d. Since %d is less than 0, s = s(modr), i.e.,
s=%d."%(s,s,s%r))
            elif(s>0):
                print("s=%d."%(s))
            return s%r

#e Value Calculation
'''FINDS THE HIGHEST POSSIBLE VALUE OF 'e' BETWEEN 1 and 1000 THAT
MAKES (e,r) COPRIME.'''

```

```

for i in range(1,1000):
    if(egcd(i,r)==1):
        e=i
print("The value of e is:",e)
print(" ")
#d, Private and Public Keys
'''CALCULATION OF 'd', PRIVATE KEY, AND PUBLIC KEY.'''
print("EUCLID'S ALGORITHM:")
eugcd(e,r)
print("END OF THE STEPS USED TO ACHIEVE EUCLID'S ALGORITHM.")
print(" ")
print("EUCLID'S EXTENDED ALGORITHM:")
d = mult_inv(e,r)
print("END OF THE STEPS USED TO ACHIEVE THE VALUE OF 'd'.")
print("The value of d is:",d)
print(" ")
public = (e,n)
private = (d,n)
print("Private Key is:",private)
print("Public Key is:",public)
print(" ")
#Encryption
'''ENCRYPTION ALGORITHM.'''
def encrypt(pub_key,n_text):
    e,n=pub_key
    x=[]
    m=0
    for i in n_text:
        if(i.isupper()):
            m = ord(i)-65
            c=(m**e)%n
            x.append(c)

```

```

        elif(i.islower()):

            m= ord(i)-97

            c=(m**e)%n

            x.append(c)

        elif(i.isspace()):

            spc=400

            x.append(400)

    return x


#Decryption

'''DECRYPTION ALGORITHM'''

def decrypt(priv_key,c_text):

    d,n=priv_key

    txt=c_text.split(',')

    x=''

    m=0

    for i in txt:

        if(i=='400'):

            x+=' '

        else:

            m=(int(i)**d)%n

            m+=65

            c=chr(m)

            x+=c

    return x

# #Message

message = input("What would you like encrypted or decrypted?(Separate numbers with ',' for decryption):")

print("Your message is:",message)

#Choose Encrypt or Decrypt and Print

choose = input("Type '1' for encryption and '2' for decryption.")

```

```

if(choose=='1'):

    enc_msg=encrypt(public,message)

    print("Your encrypted message is:",enc_msg)
elif(choose=='2'):

    print("Your decrypted message is:",decrypt(private,message))
else:

    print("You entered the wrong option.")

```

Output :

```

RSA ENCRYPTOR/DECRYPTOR
PLEASE ENTER THE 'p' AND 'q' VALUES BELOW:
Enter a prime number for p: 7
Enter a prime number for q: 13

RSA Modulus(n) is: 91
Eulers Totient(r) is: 72

The value of e is: 997

EUCLID'S ALGORITHM:
72 = 0*(997) + 72
997 = 13*(72) + 61
72 = 1*(61) + 11
61 = 5*(11) + 6
11 = 1*(6) + 5
6 = 1*(5) + 1
END OF THE STEPS USED TO ACHIEVE EUCLID'S ALGORITHM.

EUCLID'S EXTENDED ALGORITHM:
1 = 6*(1) + (-1)*(5)
1 = 11*(-1) + (2)*(6)
1 = 61*(2) + (-11)*(11)
1 = 72*(-11) + (13)*(61)
1 = 997*(13) + (-180)*(72)
s=13.
END OF THE STEPS USED TO ACHIEVE THE VALUE OF 'd'.
The value of d is: 13

Private Key is: (13, 91)
Public Key is: (997, 91)

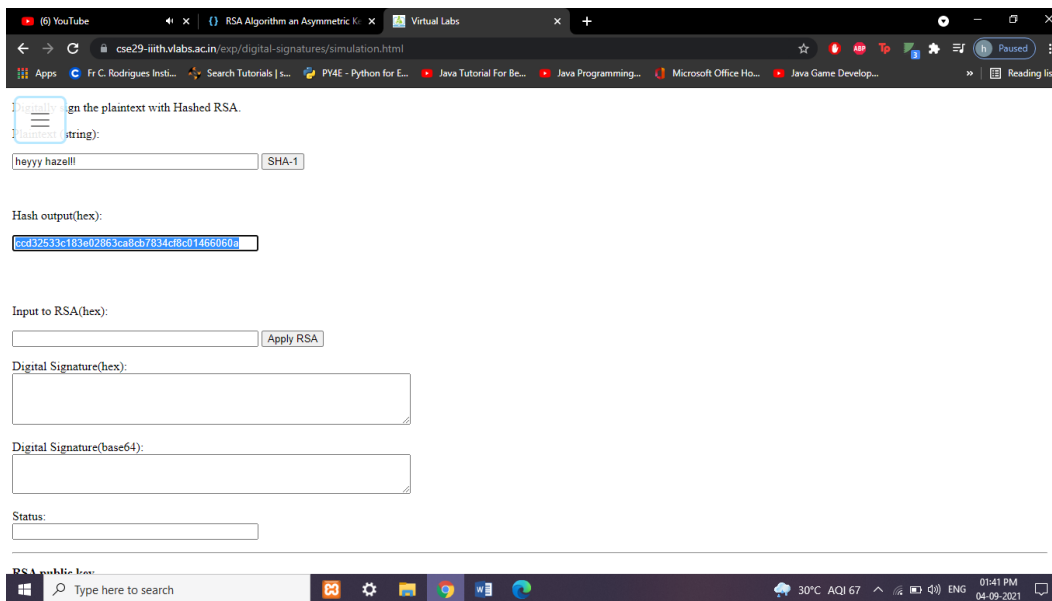
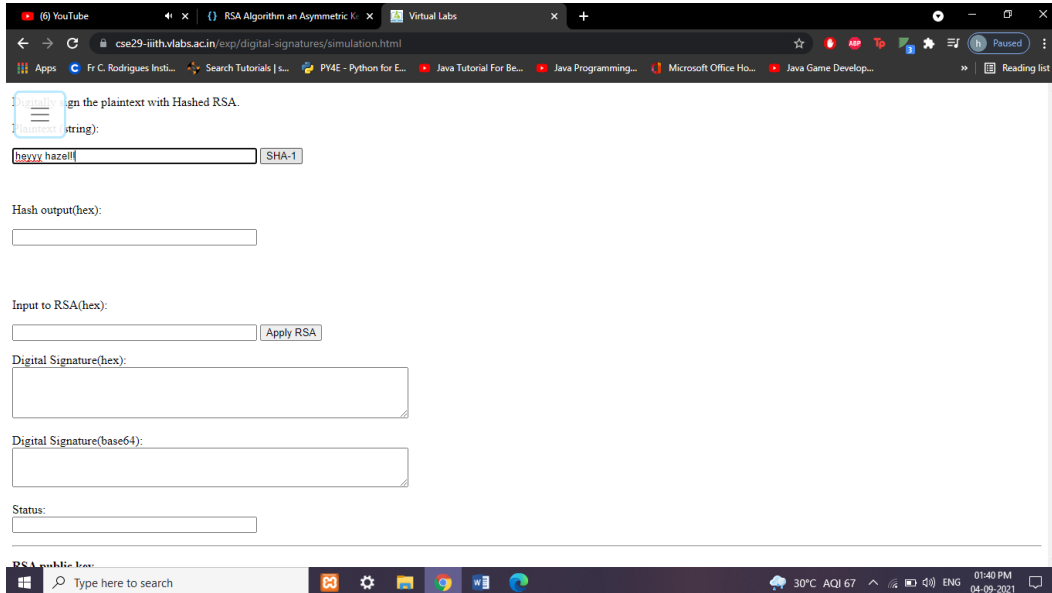
What would you like encrypted or decrypted?(Separate numbers with ',' for decryption):yes
Your message is: yes
Type '1' for encryption and '2' for decryption.1
Your encrypted message is: [24, 4, 18]

```



## Digital Signatures Scheme

**Step 1 :** Enter the input text to be encrypted in the 'Plaintext' area and generate hash value for message by clicking on the **SHA-1** button



**Step 2 :** Copy content of Hash **Output(hex)** field and paste it in **Input to RSA(hex)** field.

The screenshot shows the 'RSA Algorithm Asymmetric Key Simulation' web application. The 'Hash output(hex):' field contains the value 'ccd32533c183e02863ca8cb7834cf8c01466060a'. This value is copied and pasted into the 'Input to RSA(hex):' field. The 'Apply RSA' button is visible. Below the input field, there are empty fields for 'Digital Signature(hex):', 'Digital Signature(base64):', and 'Status:'. The browser's address bar shows the URL 'cse29-iith.vlabs.ac.in/exp/digital-signatures/simulation.html'.

**Step 3 :** Select key size of public key from **RSA Public key** section by clicking on any key button.

The screenshot shows the 'RSA Algorithm Asymmetric Key Simulation' web application. The 'Input to RSA(hex):' field contains the value 'ccd32533c183e02863ca8cb7834cf8c01466060a'. The 'Apply RSA' button is visible. Below the input field, there are empty fields for 'Digital Signature(hex):', 'Digital Signature(base64):', and 'Status:'. The 'RSA public key' section is expanded, showing the 'Public exponent (hex, F4=0x10001):' field with the value '10001'. The 'Modulus (hex):' field contains a long hexadecimal string. Below the modulus field, there are buttons for selecting key sizes: '1024 bit', '1024 bit (e=3)', '512 bit', and '512 bit (e=3)'. The browser's address bar shows the URL 'cse29-iith.vlabs.ac.in/exp/digital-signatures/simulation.html'.

**Step 4 :** Click on **Apply RSA** button to generate a digital signature.

The screenshot shows a web browser window with the URL `cse29-iith.vlabs.ac.in/exp/digital-signatures/simulation.html`. The page contains a form for generating a digital signature. The input field for the message (hex) contains the value `ccd32533c183e02863ca8cb7834cf8c01466060a`. The **Apply RSA** button is highlighted. Below the input field, the **Digital Signature(hex):** field displays the signature `66880208645cda79ffeda5e801c2a4ec77fd8edc042c40aef627842e57dcf34b4741b5c5c30b4823777f599aae7188ec40e9f7076ec5febec8d954e9b3b51ef`. The **Digital Signature(base64):** field displays the signature `ZogCCGRc2nn/7aXoAck7HF9jtwELECu91eEL1fc80tHQbXFwwtII3d/WZqucyjsQOn3827F/r7Hj2V0mztR7u==`. The **Status:** field shows `Time: 15ms`. Below the signature fields, the **RSA public key** section is visible, showing the **Public exponent (hex, F4=0x10001):** `10001` and the **Modulus (hex):** field. The browser's taskbar at the bottom shows the Windows logo, a search bar, and various application icons. The system tray on the right shows the date and time as 04-09-2021, 01:42 PM, and the weather as 30°C.

Input to RSA(hex):  
`ccd32533c183e02863ca8cb7834cf8c01466060a`

Input to RSA(hex):  
`ccd32533c183e02863ca8cb7834cf8c01466060a` **Apply RSA**

Digital Signature(hex):  
`66880208645cda79ffeda5e801c2a4ec77fd8edc042c40aef627842e57dcf34b4741b5c5c30b4823777f599aae7188ec40e9f7076ec5febec8d954e9b3b51ef`

Digital Signature(base64):  
`ZogCCGRc2nn/7aXoAck7HF9jtwELECu91eEL1fc80tHQbXFwwtII3d/WZqucyjsQOn3827F/r7Hj2V0mztR7u==`

Status:  
Time: 15ms

**RSA public key**

Public exponent (hex, F4=0x10001):  
`10001`

Modulus (hex):