Cairo University
Faculty of Engineering

Cryptography and Network Security
Spring 2022

CS Project Report

Ayman Mohamed Reda 120

Ahmed Mahmoud Hafez 1

Encrypt function: C = M^e (mod n)
If (M<n):

Encrypt(Modular exponentation) :

```
def modexp(x, y, N):
    if y==0:
        return 1
    z = modexp(x, y/2, N)
    if y%2 == 0:
        return (z*z) % N
    return (x*z*z) % N
```

Else:

Split message into blocks,
Encrypt each

To Decrypt: M = C^d (mod n)

If (M<n):

D = e ^ (-1)(mod((q-1)*(p-1)))

Modular inverse :

Extended euclidean:

```
if a == 0:
       return  (b, 0, 1)


   d, y, x = Ext_ecludean(b % a, a)
   return (d, x-(b //a) * y, y)
```

Else:

Decrypt each cipher,

Concatenate them

Brute force attack :

Iterate on prime numbers < n

Find n % number == 0 then decrypt (n, n // i)

---

conclusion :

On larger n, it is very difficult to attack. Takes a long time to attack, with time increasing exponentially.

CCA:

We choose a known ‘r’ which is coprime with n and less than n, after intercepting the ciphertext C.

Then get Alice to encrypt ‘r’, then multiply with C.

Then get Bob to decrypt the result,
Then calculate r inverse and multiply it by what is returned from Bob, then you have your message back.

```
def CCA (c,p,q, e):

    n= p*q

    #chosen r         (should be coprime with n )
    r = number.getPrime(10)

    #try to get Alice to encrypt r
    r_enc = RSA.Mod_exp(r, e, n)

    #multiply C with encrypted r
    c_dash = (c * r_enc) %n

    #send c_dash to bob and ask him to decrypt it
    m_dash = RSA.Decrypt(c_dash, p,q,e)

    #convert to number
    m_dash_n = RSA.MsgToNumber(m_dash)

    #now calculate r_inverse
    r_inv = RSA.Mod_inverse(r, n)

    #then multiply r_inverse with m_dash (decrypted c_dash) and convert back to string, now we have the Message
    M = RSA.NumberToMsg((m_dash_n * r_inv)%n)
    return (M)
```

Sender / Receiver :

- We used socket programming to make a client / server connection.
- Client acts as receiver (receiving from server)
- Server acts as sender
- In the beginning of communication, Receiver sends its public key (e,n) to the sender in order to encrypt messages
- Sender then sends stream of encrypted messages
- Receiver decrypts the messages one by one and outputs them
- When the sender sends “DISCONNECT” both terminals terminate the process

IO_test:

File containing IO style communication with the terminal

- We have 2 modes, first for full manual and the second for full autogenerated
- In the full auto mode, we generate all parameters and the users will only enter the message
- In full manual, the user HAS to enter p, then he can optionally enter the rest of the parameters. Missing parameters are auto generated.

Test_Cases.txt :

A file containing some sample test cases which were run on IO_test.py, they are different in text sizes and text types and parameters and modes.

efficiency.py :

Python code to plot encryption time versus n

Brut_Force.py :

Python code to plot Brute Force time to break RSA versus n

RSA.py :

Containing implementation of helper functions and RSA algorithm.

Efficiency graph :

Should be exponentially increasing, we didn't have much time nor memory to plot points approaching millions.

Brute Force :

Should be exponentially increasing, we didn't have much time nor memory to plot points approaching millions, as n increases, time increases exponentially.