Task 1 : Independent Implementation of AES

Sample 1

Key:

In ASCII: Thats my Kung Fu

In HEX: 5468617473206d79204b756e67204675

Plain Text:

In ASCII: Two One Nine Two

In HEX: 54776f204f6e65204e696e652054776f

Cipher Text:

In ASCII:)ÃP_Wö@"�³×:

In HEX: 29c3505f571420f6402299b31a2d73a

Decipher Text:

In ASCII: Two One Nine Two

In HEX: 54776f204f6e65204e696e652054776f

Execution Time:

Key Scheduling : 0.00011249998351559043 seconds

Encryption time : 0.00033060001442208886 seconds

Decryption time : 0.0004407000378705561 seconds

Sample 2

Key:

In ASCII: SUST CSE19 Batch

In HEX: 53555354204353453139204261746368

Plain Text:

In ASCII: IsTheirCarnivalSuccessful

In HEX: 4973546865697243617 26e6976616c5375636365737366756c

Cipher Text:

In ASCII: }⍰ÄÍ-´ÊBÈ⍰w3QeP"¦

Sã6⍰Ðo

In HEX:

7d58e0c4cd1a1eb4ca42c88d771c111f3351655022a6ea53e33682d026f

Decipher Text:

In ASCII: IsTheirCarnivalSuccessful

In HEX: 497354686569724361726e6976616c5375636365737366756c

Execution Time:

Key Scheduling : 0.00011259998427703977 seconds

Encryption time : 0.0006386999739333987 seconds

Decryption time : 0.0010950000141747296 seconds

Sample 3

Key:

In ASCII: SUST CSE19 Batch

In HEX: 53555354204353453139204261746368

Plain Text:

In ASCII: YesTheyHaveMadeItAtLast

In HEX: 5965735468657948617665 4d616465497441744c617374

Cipher Text:

In ASCII: TwX6ÁEq⍰@YÖñW⍰q⍰²ûî^½m:Ï

In HEX:

155415771458367c11457168f4059d618f1571f8e719bb2fbee5eb
d6d3acf

Decipher Text:

In ASCII: YesTheyHaveMadeItAtLast

In HEX: 59657354686579486176654d616465497441744c617374

Execution Time:

Key Scheduling : 0.00029090000316500664 seconds

Encryption time : 0.0018281000084243715 seconds

Decryption time : 0.003740699961781501 8 seconds

Sample 4

Key:

In ASCII: BUETCSEVSSUSTCSE

In HEX: 42554554435345565353555354435345

Plain Text:

In ASCII: BUETnightfallVsSUSTguessforce

In HEX:

425545546e6967687466616c6c5673535553546775657373666f72
6365

Cipher Text:

In ASCII: 64¡1⁄4é-ÔÌDCÍî¿1⁄22

In HEX:

368d9d9134a1bce994add4cc954443cd8beebfbd98df329dee10b8
17ecf8f

Decipher Text:

In ASCII: BUETnightfallVsSUSTguessforce

In HEX:

425545546e6967687466616c6c5673535553546775657373666f72
6365

Execution Time:

Key Scheduling : 0.00012909999350085855 seconds

Encryption time : 0.000990999978967011 seconds

Decryption time : 0.0016258999821729958 seconds

Issues That I Faced :

While converting the plaintexts to state matrix, it took me a while to understand the concepts of bytearray. Not only these but all of key expansion, procedure to perform mix columns etc. were very much tough for me to implemen. I had to go through numerous lessons on internet for these.

Again when I found that all samples other than the first one had plaintexts with length more than 16 bytes, I was confused as to how I should be able to encrypt a text that is not within 16 bytes length. Gradually, I turned to the slides of the cryptography and I got to understand that I should break the whole plaintext into chunks of 16 bytes, and then I should encrypt these chunks individually and combine the results into one single ciphertext.

In short, all of these tasks were very much cumbersome but I was able to implement them in the end.

Task 2 : Independent Implementation of RSA

Sample 1

Bit Size = 16

n=60491

e=7

d=17143

Plain Text:

BUETCSEVSSUSTCSE

Encrypted Text (ASCII):

53296, 32514, 13868, 59107, 55418, 47636, 13868, 21544, 47636, 47636, 32514, 47636, 59107, 55418, 47636, 13868

Decrypted Text:

BUETCSEVSSUSTCSE

Execution Time:

Key Generation Time: 0.000354900024831295 seconds

Encryption Time: 1.4099990949034691e-05 seconds

Decryption Time: 2.859998494386673e-05 seconds

Sample 2

Bit Size = 32

n=4292870399

e=11

d=1560996131

Plain Text:

BUETCSEVSSUSTCSE

Encrypted Text (ASCII):

1268954923, 3419348944, 3190642909, 1478590601, 1579240218,2079935148, 3190642909, 1466721226, 2079935148, 2079935148,3419348944, 2079935148, 1478590601, 1579240218, 2079935148,3190642909

Decrypted Text:

BUETCSEVSSUSTCSE

Execution Time:

Key Generation Time: 0.0005658000009134412 seconds

Encryption Time: 2.5500019546598196e-05 seconds

Decryption Time: 0.00012799998512491584 seconds

Sample 3

Bit Size = 64

n=18446743979220271189

e=3

d=12297829313753557747

Plain Text:

BUETCSEVSSUSTCSE

Encrypted Text (ASCII):

287496, 614125, 328509, 592704, 300763, 571787, 328509, 636056, 571787,571787, 614125, 571787, 592704, 300763, 571787, 328509

Decrypted Text:

BUETCSEVSSUSTCSE

Execution Time:

Key Generation Time: 0.0016521000070497394 seconds

Encryption Time: 9.700015652924776e-06 seconds

Decryption Time: 0.00036940001882612705 seconds

Sample 4

Bit Size = 96

n=79228162514229434696431832827

e=3

d=528187750094859144976522744427

Plain Text:

BUETCSEVSSUSTCSE

Encrypted Text (ASCII):

287496, 614125, 328509, 592704, 300763, 571787, 328509, 636056, 571787,571787, 614125, 571787, 592704, 300763, 571787, 328509

Decrypted Text:

BUETCSEVSSUSTCSE

Execution Time:

Key Generation Time: 0.002066599961835891 seconds

Encryption Time: 1.1999974958598614e-05 seconds

Decryption Time: 0.000554499973077327 seconds

Issues That I Faced :

Initially it seemed to me that generating prime numbers with a faster method was a little challenge, but I was able to solve it very easily later.

Apart from that, other functionalities of RSA encryption decryption was not very tough to implement. Rather, I would say it is easier to implement compared to AES implementation. I went through the cryptography slide and from there I was able to understand the concept of RSA encryption and decryption.

What confused me a little was the provided public and private key in the assignment slide as sample input. RSA encryption requires generating two prime numbers and depending upon these two values, public and private keys are generated. So, if I already have with me the values of public and the private key, then generating prime numbers would not make sense.

However, I kept two versions of RSA encryption and decryption in my notebook file.

In one version (shown below), I have the predefined values of e , n, d (public and private key) so that the main RSA function does not necessarily have to compute the prime numbers.

RSA implementation for predefined value of e,n,d

```python
def rsa(rsa_key_size, plaintext, n, e, d):

    prime_number_bit_length = rsa_key_size // 2

    start_time = time.perf_counter()

    end_time = time.perf_counter()
    key_generation_time = end_time - start_time
    print("Key Generation Time:", key_generation_time, "seconds")


    start_time = time.perf_counter()

    cipher = rsa_encrypt(plaintext, e, n)

    end_time = time.perf_counter()
    encryption_time = end_time - start_time
    print("Encryption Time:", encryption_time, "seconds")

    print("Cipher Text:", cipher)

    start_time = time.perf_counter()

    decipher = rsa_decrypt(cipher, d, n)

    end_time = time.perf_counter()
```

Thus I could use this values directly to perform encryption and decryption and show the expected results as shown below –

```python
rsa_key_size=16
n=60491
e=7
d=17143

plaintext = "BUETCSEVSSUSTCSE"
rsa(rsa_key_size, plaintext, n, e, d)
```

0.0s Python

```
Key Generation Time: 5.00003807246685e-07 seconds
Encryption Time: 1.7200014553964138e-05 seconds
Cipher Text: [53296, 32514, 13868, 59107, 55418, 47636, 13868, 21544, 47636, 47636, 32514, 47636, 59107, 55418, 47636, 13868]
Decryption Time: 3.709999145939946e-05 seconds
Deciphered Text: BUETCSEVSSUSTCSE
```

In another version, I performed the prime number generation operation twice to obtain p and q, and depending upon these two computed the public and private key.

RSA implementation with randomly generated value of p & q

```python
def rsa(rsa_key_size, plaintext):

    prime_number_bit_length = rsa_key_size // 2

    start_time = time.perf_counter()

    # Generate prime numbers p and q
    p = generate_prime_number(prime_number_bit_length)
    q = generate_prime_number(prime_number_bit_length)

    end_time = time.perf_counter()
    key_generation_time = end_time - start_time
    print("Key Generation Time:", key_generation_time, "seconds")

    n = p * q
    e = 65537

    d = calculate_private_key(e,p,q)

    start_time = time.perf_counter()

    cipher = rsa_encrypt(plaintext, e, n)
```

Using this public and private key, encryption and decryption were performed, which upon appropriate plaintext and key size being provided, resulted in expected output.

```python
plaintext = input('Please enter the plaintext : ')
#BUETCSEVSSUSTCSE
rsa_key_size = int(input("Please enter key size(16/32/64/96) : "))

rsa(rsa_key_size, plaintext)
```

66] ✓ 10.9s Python

```
Key Generation Time: 0.0006315999780781567 seconds
Encryption Time: 7.100001676008105e-05 seconds
Cipher Text: [1670895383, 921725888, 1108904878, 1391361755, 1335792962, 115770154, 1108904878, 2066953858, 115770154, 11577015
Decryption Time: 0.00020589999621734023 seconds
Deciphered Text: BUETCSEVSSUSTCSE
```

Task 3 : Hybrid Cryptosystem using AES & RSA

Sample

Plaintext :

Two One Nine Two

Key :

Thats my Kung Fu

Encrypted Text :

In Hex : 29c3505f571420f6402299b31a02d73a

Encrypted Key (Not Fixed) :

44985,18118,26142,24844,13105,1845,27677,1434,1845,39654,23889,43075,596,1845,15323,23889

Decrypted Key :

Thats my Kung Fu

Decrypted Text :

Two One Nine Two

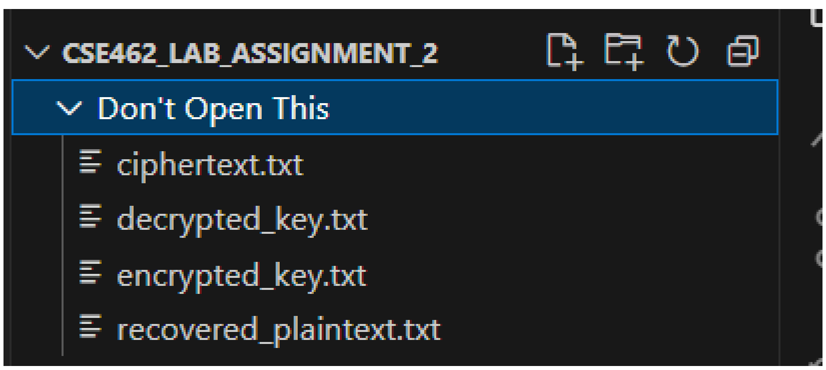
Issues That I Faced :

At first, I was unable to understand how I should be able to combine these two encryption method. Gradually after implementing AES and RSA independently I found a way out.

Keeping all the functionalities of the RSA and AES cryptosystem, I encrypted the above mentioned text using AES encryption and the key using RSA encryption.

I stored the encypted key and the encrypted text (ciphertext) in a folder named “Don’t Open This” as instructed in the assignment. Essentially this is what Alice(sender) should do.

Bob, who is the receiver, reads the encrypted key and cipher text from the folder and then using the private key decrypts the key using RSA decryption. Later Bob recovers the plaintext using the ciphertext using AES decryption and stores both the plaintext and the decrypted key in the same folder.



Finally Bob, matches the recovered plaintext with the original plaintext that was sent by Alice.

```
#Bob checks if the reovered plaintext and the original plaintext matches or not
if recovered_plaintext == plaintext:
    print("Decryption successful! Original plaintext recovered:", recovered_plaintext_str)
else:
    print("Decryption failed! Original plaintext not recovered.")
```

[6] ✓ 24.6s

```
Decryption successful! Original plaintext recovered: Two One Nine Two
```

And with that the hybrid cryptosystem is completed.

END