Network Security - Project

liangjin

1 Introduction

This is a report for the project of course Network Security. The code can be find in $https://github.com/liangjindeamo-yuer/Network_Security_Project$.

2 Task 1:Textbook-RSA

2.1 RSA algorithm

The RSA algorithm is as follows:

- Choose two large primes p and q.
- Let $n = pq$ and $\phi(n) = (p-1)(q-1)$.
- Choose e such that $(e, \phi(n)) = 1$ and the public key is (e, n).
- Find d such that $ed \equiv 1 \mod \phi(n)$ and the private key is (d, n).

2.2 Some details about our code

Before we execute our code, we need to understand some specific parameters.

- $-b$ or $--bit\_length$: It determines the length of prime p and q.
- $-f$ or $--file$: You can use this parameter to define the path to the read file, which may be the file you need to encrypt or decrypt.
- $-m$ or $--mode$: You can choose to encrypt or decrypt using this parameter, 0 for encrypting and 1 for decrypting.
- $-r$ or $--rsa$: It is the path of class RSA. It will be written when encrypted and read when decrypted.
- $-o$ or $--oaep$: It determines whether to use OAEP mode for RSA encryption and decryption.

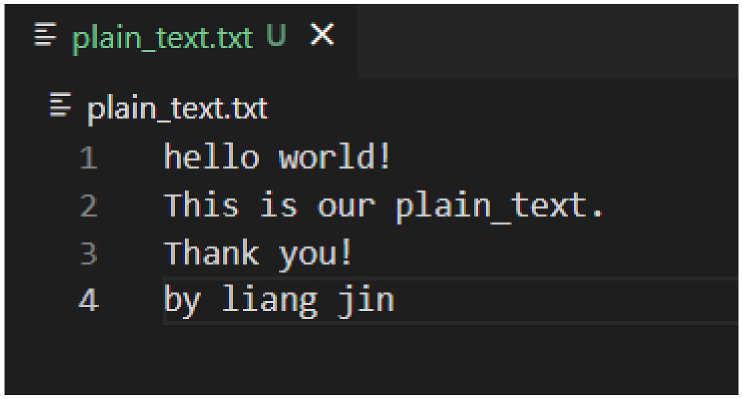


Figure 1: Original Plaintext

2.3 Experiment

First, we define the plaintext $\mathcal{P}$ as shown in Fig. 1:

Then we generate a random RSA key pair with a given key size (1024), and encrypt the plaintext. Execute the following command:

python RSA.py -b 1024 -f plain__text.txt -m 0 -r rsa.pkl

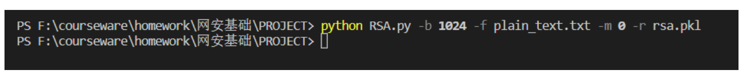


Figure 2: Encryption

Then we can obtain the *RSA* in file *rsa.pkl* and the ciphertext $\mathcal{C}$:

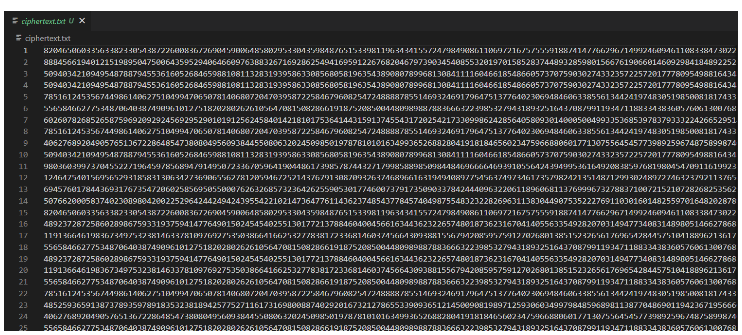


Figure 3: Ciphertext

We decrypt the ciphertext $\mathcal{C}$ using the following code:

python RSA.py -f ciphertext.txt -m 1 -r rsa.pkl

Then as shown in Fig. 4, we can get the decrypted plaintext $\mathcal{P}'$.

As we can see, the decrypted plaintext is the same as the original plaintext.

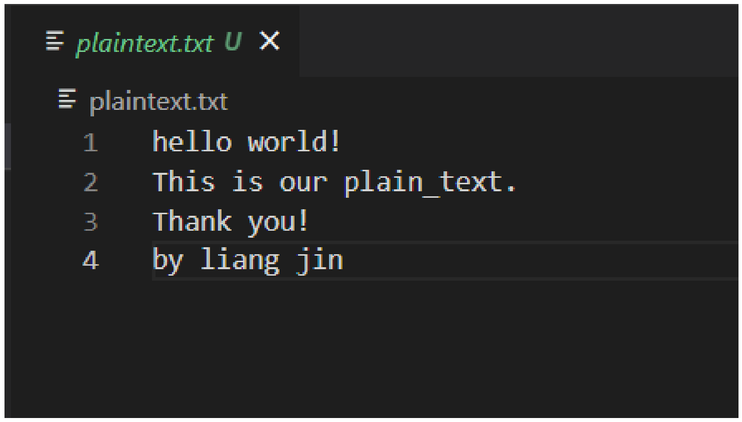


Figure 4: Decrypted Plaintext

3 Task 2:CCA2 Attack

3.1 Server-Client communication

A client can send WUP message to the server. It generate a 128-bit ASE session and encrypt this session key using a 1024-bit RSA public key which is generated by the server. Then it use the AES session key to encrypt the WUP request which includes WUP content, its mac and ID and send the RSA-encrypted AES session key and the encrypted WUP request to the server.

A server can receive the WUP request from client and reply to those valid requests. It decrypt the RSA-encrypted AES key it received from the client and decrypt the WUP request using the AES session key. Then it choose the least significant 128 bits of the plaintext to be the AES session key and send an AES-encrypted response if the WUP request is valid.

Experiment. We test the Server-Client communication using the following code:

```
# cswup
msg = client.send_wup("Hello world!",server.
    public_key)
# s
rec_msg = server.receive_wup(msg)
# c
fianl_msg = client.receive_msg(rec_msg)
```

The client send a WUP request with message "Hello world!" to the server. The server receives the WUP request and verifies its legitimacy and validity and then sends the feedback to the client. The result in Fig. 5 shows that when client sends "Hello world" to the server, it receives the WUP successfully and then the client receives the feedback from the server successfully too.

```
Sever receive message:Hello world!
Client receive:Responce:Hello world!
To:52:54:00:7f:da:e7and830013-64-616960-7
```

Figure 5: Client-Server Communication Test

3.2 CCA2 Attack

3.2.1 Description of CCA2 attack

According to [1], the core of CCA2 attack is shown as follows:

Let C be the RSA encryption of 128-bit AES key k with RSA public key (n, e). Thus, we have

$$C \equiv k^e \pmod{n}$$

Now let C_b be the RSA encryption of the AES key

$$k_b = 2^b k$$

i.e., k bitshifted to the left by b bits. Thus, we have

$$C_b \equiv {k_b}^e \pmod{n}$$

We can compute C_b from only C and the public key, as

$$\begin{aligned} C_b &\equiv C(2^{be} \bmod n) \pmod{n} \\ &\equiv (k^e \bmod n)(2^{be} \bmod n) \pmod{n} \\ &\equiv k^e 2^{be} \pmod{n} \\ &\equiv (2^b k)^e \pmod{n} \\ &\equiv {k_b}^e \pmod{n} \end{aligned}$$

Figure 6: CCA2 Attack

The attacker use k_b as the *AES* key to encrypt some message, use C_b as the encrypted *AES* key and send them to the server. If the server reply to the attack, it means that the test bit in k_b is right. Otherwise, the test bit should be the result of operation NOT of the bit in k_b.

3.2.2 Experiment

We execute the following code and select to attack:

python Server-Client.py

And then the attacking process is as follows:

Figure 7: CCA2 Attack Process

And as we can see, the cca2 attack successfully decrypted the message "Hello world" sent by the client to the server. Moreover, the attack may fail because the rabin miller algorithm may not be capable of producing true prime.

4 Task 3:OAEP

4.1 Description of OAEP

Since textbook RSA is vulnerable to attacks, using OAEP key padding algorithm can defend the attack. In cryptography, Optimal Asymmetric Encryption Padding (OAEP) is a padding scheme often used together with RSA encryption. OAEP satisfies the following two goals:

1. Add an element of randomness which can be used to convert a deterministic encryption scheme (e.g., traditional RSA) into a probabilistic scheme.

2. Prevent partial decryption of ciphertexts (or other information leakage) by ensuring that an adversary cannot recover any portion of the plaintext without being able to invert the trapdoor one-way permutation.

The frame of OAEP is shown in Fig. 8. In the Fig. 8, n is the number of bits in the RSA modulus. k_0 and k_1 are integers fixed by the protocol. m is the plaintext message, an $(n - k_0 - k_1)$ bit string. G and H are typically some cryptographic hash functions fixed by the protocol. And $+$ means xor operation.

4.2 Code

The OAEP can be realized by adding some code in RSA. The encryption code is as follows:

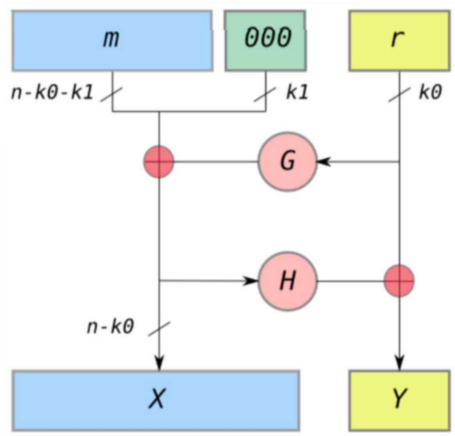


Figure 8: OAEP

```
def oaep_encode(plain_text):
    oaep_msg = []
    for num in plain_text:
        m = num << oaep_k1
        r = get_prime(oaep_k1)

        X = m ^ hashFunction(r)
        Y = r ^ hashFunction(X)

        oaep_msg.append((X << oaep_k0) | Y)
    return oaep_msg
```

Moreover the decryption code is as follows:

```
def oaep_decode(cipher_text):
    oaep_msg = []
    for num in cipher_text:
        Y = num & bit_mask(oaep_k0)
        X = num >> oaep_k0
        r = Y ^ hashFunction(X)
        m = (X ^ hashFunction(r)) >> oaep_k1
        oaep_msg.append(m)
    return oaep_msg
```

4.3 Experiment

First, we use the same plaintext *mathcalP* and decrypt it:
python RSA.py -b 1024 -m 0 -f plain_text.txt -o 1

Then we can obtain the RSA in file *rsa.pkl* and the ciphertext $\mathcal{C}$:

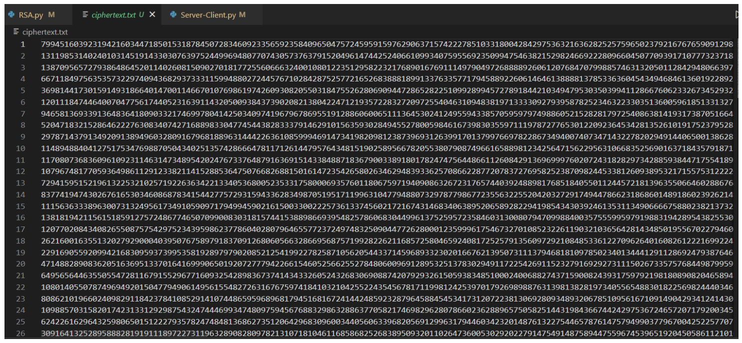


Figure 9: Ciphertext with OAEP

We decrypt the ciphertext $\mathcal{C}$ using the following code:
python RSA.py -m 1 -f ciphertext.txt -o 1 -r rsa.pkl
Then as shown in Fig.10, we can get the decrypted plaintext $\mathcal{P}'$ which is the same as $\mathcal{P}$.

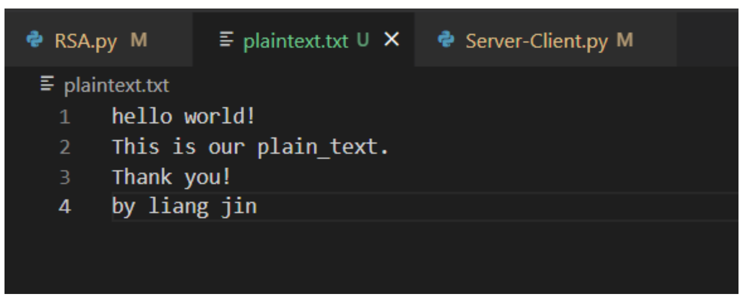
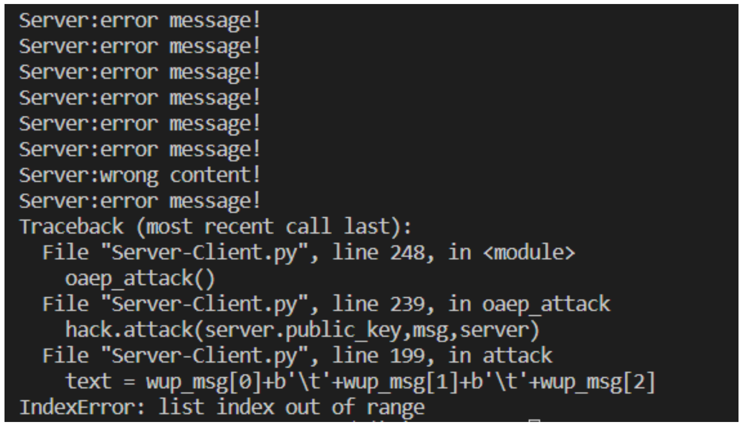


Figure 10: Decrypted Plaintext with OAEP

Ultimately, we execute the CCA2 attack when the RSA is equipped with $OAEP$. To avoid errors caused by the prime number generation algorithm, we tried the attacks for 100 times (as shown in Fig. 11) and all the attacks failed.

5 Conclusion

Language *python* is used to do the project. And in code $RSA.py$ and $Server-Client.py$, all the tasks in the project PPT are realized and the results are shown above.



```
Server:error message!
Server:error message!
Server:error message!
Server:error message!
Server:error message!
Server:error message!
Server:wrong content!
Server:error message!
Traceback (most recent call last):
  File "Server-Client.py", line 248, in <module>
    oaep_attack()
  File "Server-Client.py", line 239, in oaep_attack
    hack.attack(server.public_key,msg,server)
  File "Server-Client.py", line 199, in attack
    text = wup_msg[0]+b'\t'+wup_msg[1]+b'\t'+wup_msg[2]
IndexError: list index out of range
```

Figure 11: Attack Result under OAEP

References

[1] Jeffrey Knockel, Thomas Ristenpart, and Jedidiah Crandall. When textbook rsa is used to protect the privacy of hundreds of millions of users. *arXiv preprint arXiv:1802.03367*, 2018.