

# 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/NetworkSecurityProject>.

## 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  $ed \equiv 1 \pmod{\phi(n)}$  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.

```

≡ plain_text.txt U X
≡ plain_text.txt
1 hello world!
2 This is our plain_text.
3 Thank you!
4 by liang jin

```

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
```

```

PS F:\courseware\homework\网安基础\PROJECT> python RSA.py -b 1024 -f plain_text.txt -m 0 -r rsa.pkl
PS F:\courseware\homework\网安基础\PROJECT>

```

Figure 2: Encryption

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

```

≡ ciphertext.txt X
≡ ciphertext.txt
1 82865506813563382338543872608836726084508648588295338435988487651533811363431557247984988611869721675755918871477662967149924699461188338273822
2 88845615401215189596479066435952940646697638832671628625484169591226761204679739834548055320197815852837448013295988156676198660146092984184892252
3 5894034210949548788794553616052684659881081132831939586338856808186354389808878996813084111168466185486685770759030274313257257207778095498816434
4 5894034210949548788794553616052684659881081132831939586338856808186354389808878996813084111168466185486685770759030274313257257207778095498816434
5 7851612431541488616802751809478658781886807287839587225846796825474888878551469124691796475137768023869484861385561344241974838518858881817433
6 5558466277534878640387490961812751820280262618564788158828661918752085004488988878836663223985327941389325164178813438368570681300768
7 682687828526587596920924569295290181912562458481421810175364144315913745543172025421733899862428564058093014080504049335368538739332246652951
8 7851612431541488616802751809478658781886807287839587225846796825474888878551469124691796475137768023869484861385561344241974838518858881817433
9 486278892849874633372726454738804895603845508863202458950197818101814349352586298411818456402347664880617713875564545773982925674872899474
10 5894034210949548788794553616052684659881081132831939586338856808186354389808878996813084111168466185486685770759030274313257257207778095498816434
11 988368399737845522715645978568947914958723367859641384486173985784432717998888858984486966664693918556424394953616492883859768188845478911619923
12 645258509156959231858318832716968556278120596725214767813887893283766896183184988979546374973661735798042135188712993824897274632392113365
13 6845768178436931767354720602385695808676263685732364262559853817746807379173589033784244408632206118860681176999673278817180721521872628253562
14 587662800583740238088042002252964244249424395542218214736477611436237485437784574849875548323282696311383044907552276911031081482559701648282878
15 82865506813563382338543872608836726084508648588295338435988487651533811363431557247984988611869721675755918871477662967149924699461188338273822
16 480237272586828986759131979581677649015024548255138177217884684084561634432322657488187362316784389563354928207031694774881148808146627888
17 119136646198367349753238146337818976927535838664166253277838172136814683745664389388155679428089579912702688138515232656769654284457538418896213817
18 5558466277534878640387490961812751820280262618564788158828661918752085004488988878836663223985327941389325164178813438368570681300768
19 480237272586828986759131979581677649015024548255138177217884684084561634432322657488187362316784389563354928207031694774881148808146627888
20 119136646198367349753238146337818976927535838664166253277838172136814683745664389388155679428089579912702688138515232656769654284457538418896213817
21 5558466277534878640387490961812751820280262618564788158828661918752085004488988878836663223985327941389325164178813438368570681300768
22 7851612431541488616802751809478658781886807287839587225846796825474888878551469124691796475137768023869484861385561344241974838518858881817433
23 4852585091569592318583188327169685562781205967252147678138878932837668961831849889795463749736661735798042135188712993824897274632392113365
24 486278892849874633372726454738804895693845508863202458950197818101814349352688288419181846560234799668886017713875564545773982925674875899874
25 5558466277534878640387490961812751820280262618564788158828661918752085004488988878836663223985327941389325164178813438368570681300768

```

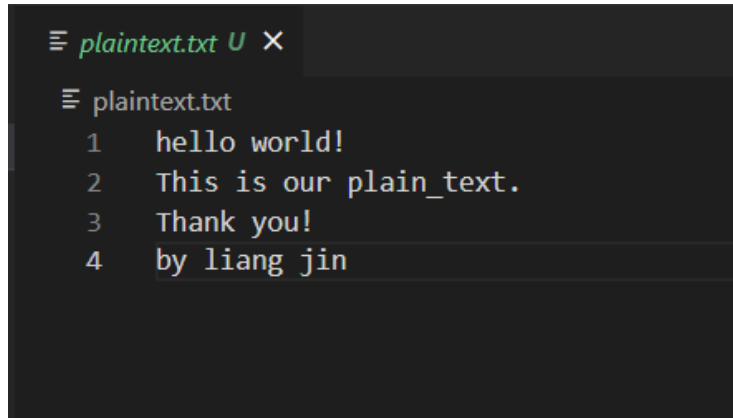
Figure 3: Ciphertext

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.



```
≡ plaintext.txt U X
≡ plaintext.txt
1 hello world!
2 This is our plain_text.
3 Thank you!
4 by liang jin
```

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:

```
1 # cswup
2 msg = client.send_wup("Hello_world!", server.
   public_key)
3 # s
4 rec_msg = server.receive_wup(msg)
5 # c
6 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

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

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:

```
Server:error message!
Server:wrong content!
Server:error message!
Server:error message!
Server receive message:attacking test
Server receive message:attacking test
Server:error message!
Server receive message:attacking test
Server:wrong content!
Server:error message!
Server:error message!
Server receive message:attacking test
Server:error message!
Server:error message!
Server receive message:attacking test
The wap content: Hello world!
Out[357]: 147293456690811218133769901006967505641
```

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:

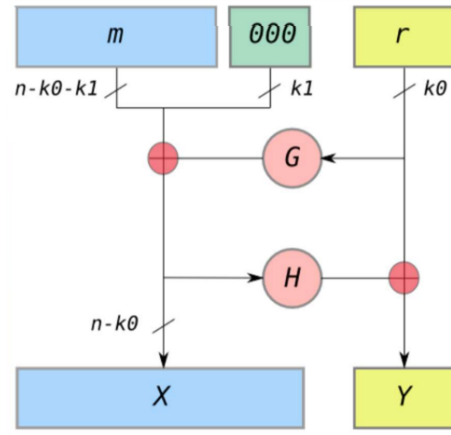


Figure 8: OAEP

```

1 def oaep_encode(plain_text):
2     oaep_msg = []
3     for num in plain_text:
4         m = num << oaep_k1
5         r = get_prime(oaep_k1)
6
7         X = m ^ hashFunction(r)
8         Y = r ^ hashFunction(X)
9
10        oaep_msg.append((X << oaep_k0) | Y)
11    return oaep_msg

```

Moreover the decryption code is as follows:

```

1 def oaep_decode(cipher_text):
2     oaep_msg = []
3     for num in cipher_text:
4         Y = num & bit_mask(oaep_k0)
5         X = num >> oaep_k0
6         r = Y ^ hashFunction(X)
7         m = (X ^ hashFunction(r)) >> oaep_k1
8         oaep_msg.append(m)
9     return oaep_msg

```

### 4.3 Experiment

First, we use the same plaintext *mathcal{P}* 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  $\mathcal{C}$ :

```

RSA.py M  ciphertext.txt U X  Server-Client.py M
ciphertext.txt
1 799451601921104216014471850151107845072814609217555927584899504725724595915076290617157422278518311800428429753612161628252575960427921676559091258
2 13113803114024010141191813030763975449994007707430517437915040614744253480461099348079556921509947446132132026692120066465879039171077723718
3 1347095657279186486452014492608150902701817255606632400108012215129582217608016769111497904972688809260612076047079980574011205011284248866397
4 667118497563573229740941682937313159948802724576718284287525772165268388189913376357717945889226061466138881378533636045434946613601522892
5 86961441710159160118664014700116670107690619742689380295581184252420869804472862822510992899457278918442104847953075039411286670862132671452932
6 1201118424646079877951744652116391141296983843730020621308424214357224322997254046330844838713133002793632925236322130353360896185131337
7 94658136933913648364180903121746978041425634007419679678695519128860606511364530241249559433857859597949886052152828179725408638141913738701664
8 52047183215286462276388340742716889813047745443828337914629101563593028494552780958461673983035597111978727653012092364534281352610151752279528
9 29787147915420013084980308916796818063144226381085994691472413083081133739621285991317399769782207345089740747432276202294140652013628
10 11468984841275179347698878994348251574286647811712614479576434813196298956782055380790874966165889813425647156229563186683525698167178435791871
11 11708073683699610921146314734895420247673376487916391514338487183679003389180178247475644866112688429136969976020724318282973428859384471755189
12 107967481770293649861129123382114152885364750766426881501614723542658026346294839136257886622877207837276958252307090344531812089389512717575712222
13 72441295121361122512102571922636421334053680623531730080693760110675971500080632972317627440392488081768514495501124457218139635286646838869
14 83719474302676165303460868781544277291159438628349870519517199610479488073297877986723556322520420327917044786621186860148918663926214
15 1156363180618073112495617349105907179494598216150813082225736113745602172167411608340638952005892822941985434303924613511349006667588021821372
16 118118215154518913792240074604997080808118158415188966939582378060410489961375295721504011808070470998080357559959791081114289941822538
17 120780384340626508257545295243399862377860402087964655772724074132400447726200001219996175467327010492122611903210365642414348501955670279466
18 262160016355120279200004019507675897918170912680095632866956875199282262110857258065924081725257913560972921084533612270962640160826122109226
19 22916095920994216818959373995381928977982085212541992787587105620543171459689132302816676213956711117946011810978302340114441291128692479387646
20 471482398670391630951370161699085010377774226615469256625278480009012095321738202491172545911272916929771115020733573768487087999
21 6495654463550557201167915529677160932542898367374143132695241268180908874207920326158938348510802400688274171590880243931792792198180800420405894
22 1080140558787496349201847794906149561548272631676759741841032184255243545678171199812425397017926989887631398138281973495565488301822568024404346
23 880621015660240049291184237841085291410740486259596968179451681672414248592379645884543171207223813062089340932067810956167109149042541241436
24 19808570158207122131312068743242444694347480979456768832803280374950274698263007866073620865795925144319848674404257327465720712940345
25 624216296432598065015122793578247484813686273512064296830960034095663396820569129963179446034232014876132275446578761475794990377967004252257787
26 38016413252895888281919111897227119632890828097827110718104611683868252683895091201102647300053029202279147549148780447559674539651920854858112101

```

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  $\mathcal{P}$ .

```

RSA.py M  plaintext.txt U X  Server-Client.py M
plaintext.txt
1 hello world!
2 This is our plain_text.
3 Thank you!
4 by liang jin

```

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.