RSA Project: Report

BY 冯璇 520021911147

Table of contents

1 Textbook RSA Implementation . . . 1

1.1 Code Logic Explanation . . . 1
1.2 Input & Output Explanation . . . 1
1.3 Result Demonstration . . . 2

2 CCA2 Attack Simulation . . . 2

2.1 WUP Format . . . 2
2.2 Server-Client Communication & The Attack Process . . . 2
2.3 Input & Output Descriptions of the program . . . 3
2.4 Result Demonstration . . . 3

3 OAEP . . . 3

3.1 Code Logic Explanation . . . 3
3.2 Input & Output Description . . . 4
3.3 Result Demonstration . . . 4

4 CCA2 Attack on OAEP . . . 5

1 Textbook RSA Implementation

1.1 Code Logic Explanation

The key algorithms I used to complish this part are explained as following:

- Key Generation:

  Given key size(in bits), first generate prime number p and q, using **Rabin-Miller Algorithm** to practice prime test for large numbers and a prime number set for small numbers; then check if $n = p \times q$ is of the required size, if not then adjust size of p or q and generate again;

  To generate the public key e and the private key d, fix $e = 65537$ (prime) and then calculate its modulu inverse d using **Extended Euclid Algorithm**. Note that if $\gcd(\phi(n), e) \neq 1$, we'll generate p, q, n again.

- Encrypt & Decrypt:

  All we need for textbook RSA encryption and decryption is $(a^b \bmod c)$. We'll use **power decomposition** to make it faster.

1.2 Input & Output Explanation

The program first asks the size of n, then generates the RSA key accordingly, and writes p, q, n, e, and d to RSA_p.txt, RSA_q.txt, RSA_Modular.txt, RSA_Public_Key.txt, RSA_Private_Key respectively.

Then the program reads the content to be encrypted from Raw_Message.txt. After encryption, the ciphertext will be written to Encrypted_Message.txt.

1.3 Result Demonstration

The result is demonstrated as following:

```
(base) PS D:\1PersonalFiles\2022-23seme2\RSA-CCA2\Task 1> python ./RSA.py
key size: 1024

raw_message =  It has significant impacts on various domains of modern society.

cipher =   12028351541134957635762118709004918586766294635631515579289472833992504472819461036767165142533021400309331665681683167654916881530050189829985108139880720321913433469133
171929290607059111406337939953424408112632149064099932979455004535354321299910171771346001927458107721907736031152254673925955807003997 1476

decipher =  It has significant impacts on various domains of modern society.
```

which shows that the encryption and decryption processes are correspondent with expectation.

To check more about the result, please input the raw message into Raw_Message.txt then run RSA.py and check the files described above.

2 CCA2 Attack Simulation

2.1 WUP Format

Each WUP message consists of an AES-encrypted request and an AES key encrypted by RSA.In consideration of convenience, we do not include respond in the WUP here, and we'll practice the AES key generation process in Server's initiation, which is to select a random number.

The WUP request format is a **hexadecimal byte stream**, and the AES key is a decimal integer.

All AES encryption and decryption are enbled by AES library in `Crypto.Cipher`.

2.2 Server-Client Communication & The Attack Process

The generate_history() function of class Server generates a history message based on the RSA public and private keys and AES keys that have been generated, and both the client and the attacker can access this message.

The attacker guesses the AES key each time a bit based on the history message, and combines C_i with the request encrypted by guessed AES key, then sends this WUP message to the server. The detailed attcking method is shown as following, suppose we are now trying to guess the bth least significant bit of AES key:

$$\begin{aligned} \text{Define} \quad C &\equiv k^e (\bmod n) \\ k_b &= 2^b k \end{aligned}$$

Our target is the RSA-encrypted k_b, which is

$$\text{which is,} \quad C_b \equiv k_b^e (\bmod n)$$

then we have

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

And let k_b' denote the guessed key of the current round, and $C_b' \equiv {k_b'}^e (\bmod n)$;

The attacker will use k_b' to encrypt a request, suppose the encrypted request is $\mathrm{Re}_{k_b'}$. The attacker sends $\mathrm{Re}_{k_b'}$ combined with C_b as a WUP message to the server, then server will first acquire k_b from C_b with RSA private key, then acquire the request with k_b as the AES communication key. If $k_b = k_b'$, the server will get a legitimate request and respond, which means the attacker's guess is correct for this round. On the contrast, the attack will take the opposite of its guess.

2.3 Input & Output Descriptions of the program

- RSA_p.txt, RSA_q.txt, RSA_Modular.txt, RSA_Private_Key.txt, RSA_Public_Key.txt

  The same RSA key generation files as Task 1;

- History_Message.txt

  The first line is the WUP request (hexadecimal bitstream);

  The second line is AES (decimal);

- AES_key.txt

  Original AES key between the server and the client, without RSA encryption (hexadecimal integer);

- WUP_Request.txt

  Request content in the history message, without AES encryption (hexadecimal);

- AES_Encrypted_WUP.txt

  After obtaining the AES key and the WUP request, the attacker encrypts again and writes to this file, in order to check if the attack has been successfully practiced.

2.4 Result Demonstration

Run CCA2.py and the result is as following, which shows that the attack is successfully realized.

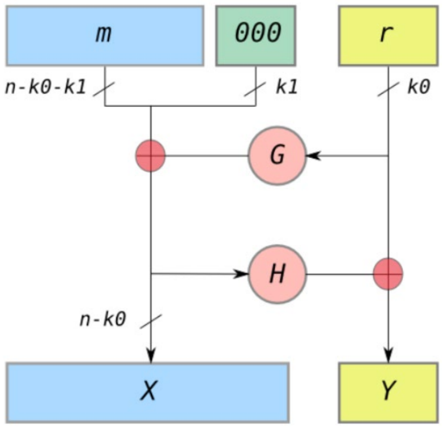
```
D:\Anaconda\python.exe "D:\1PersonalFiles\2022-23seme2\RSA-CCA2\Task 2\CCA2.py"
History information:  this is a history message request
AES_KEY is:  326159362664596819972842436294192047516
Success

Process finished with exit code 0
```

To check more about the result, please run CCA.py and check the txt files as descripted above.

3 OAEP

3.1 Code Logic Explanation

The essential reason why textbook RSA cannot resist CCA2 attack is that the same plaintext is encrypted to get the same ciphertext each time, therefore an attacker can use the History_Message get the AES key. Thus OAEP introduces a random number to improve this. The padding scheme is shown as following:



I used sha384 for G and sha256 for H, and $k_0 = k_1 = 256$.

3.2 Input & Output Description

- RSA_p.txt, RSA_q.txt, RSA_Modular.txt, RSA_Private_Key.txt, RSA_Public_Key.txt

  The same RSA key generation files as Task 1 and Task 2;

- Random_Number.txt

  The generated k-bit random number r (decimal);

- Raw_Message.txt

  Raw message to be encrypted (string);

- Message_After_Padding.txt

  X||Y padded from m and r (hexadecimal);

- Encrypted_Message.txt

  Encrypted message, written after encryption (hexadecimal);

3.3 Result Demonstration

Run OAEP.py and the result is as following, which shows that the encryption and decryption processes with OAEP padding are correspondent with expectation.

```
D:\Anaconda\python.exe "D:\1PersonalFiles\2022-23seme2\RSA-CCA2\Task 3\OAEP.py"
Raw Message:  this is a test towards OAEP padding over textbook RSA

Encrypted Message:
 0x81b3a5e18f72c0065b1a9ffe981625f0196c5d8b5f7ee40d1315438a58a1811ae7e4effcd4a56047d56c627bf93c055fabbc72c77abfe63a5290d692e9884facb6c6e33329ab7567ddd42fc6555c6b9ab
 c30534b83e60932977ede09aa174c06e24a84d704504773b56b1b53a10b56efbf82e0de42c892d8f7df7db5db9485ea

Decrypted Message:  0x746869732069732061207465737420746f7761726473204f41455020706164646696e67206f7665722074657874626f6f6b20525341

Success

Process finished with exit code 0
```

To check more on the result, please run OAEP.py and refer to the files descripted above.

4 CCA2 Attack on OAEP

After implementing OAEP RSA, I have tried to practice CCA2 attack on OAEP RSA (the code is in ./Task_3 named CCA2onOAEP.py).

Run OAEPonRSA.py and the result is as following, which shows that OAEP can successfully defend against CCA2 attacks. Since every round the server couldn't interpret the attacker's message correctly, and every round the attacker guesses 1 for the current bit and gets a negative respond, therefore the final guess result would be 0.

```
D:\Anaconda\python.exe D:\1PersonalFiles\2022-23seme2\RSA-CCA2\Task_3\CCA2onOAEP.py
guessed AES_KEY is:  0

Fail

Process finished with exit code 0
```