# Assignment 2

- yixi rao u6826541

**Total word**: 2943 (report) + 623 (appendix)

**NOTE**: Appendix is just to make copying code easier, it is not needed to read the appendix to understand the attack strategy.

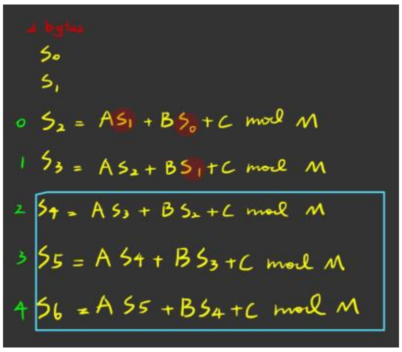
## Question 2 ELCG Cipher

For this ELCG cipher, the $S_0$ and $S_1$ are secret seed since the stream starts with $S_2$ and also the A and B are secret. So, in order to get the value of all the secret keys, we need to explore the equation $S_i = (A \cdot S_{s-1} + B \cdot S_{s-2} + C)\ mod\ M$, there are three unknows coefficients in the equation (A, B, C), therefore we need at least three stream keys equation to solve it. We also need to know the value of three consecutive stream keys and the modular M. In order to get the three consecutive stream keys, the property of stream cipher can be used, which is $Y = S \oplus X, S = Y \oplus X$ and also the fact "every random number is encoded as two bytes in the key stream" can be used. In general, if we can get the first 10 bytes of the plaintext and the first 10 bytes of the ciphertext, then we can use linear equation with three unknows to solve it and finally we can use the keys to decrypt the file.

- **Vulnerability**: Bad cryptographic properties due to the linearity of most PRNGs and the Output can be reproduced and can be predicted.

  We know that the stream cipher is generated using an extended version of linear congruential generator, so this is a Pseudorandom Number Generator which has the property that the PRNGs are not random in a true sense because they can be computed and are thus completely deterministic by using the linear equation it defined. As a result, if the attacker knows 10 bytes of the plaint text and 10 bytes of the cipher text, then the attacker can predict the future number correctly.

The exploitation is divided into three steps, the first step is to get the first 5 stream keys using the given hint.txt and cipher.bin, the reason why I need the first 5 stream keys rather than 3 stream keys is that the first two stream keys contains $S_1$ and $S_0$ that we do not know so it cannot be used to calculate the linear equation (formular in graph 2.1)




graph 2.1

Next, we can use the property of $Y = S \oplus X, S = Y \oplus X$ to get the first 5 stream keys using python script.

```
with open('Q2/cipher.bin', 'rb') as f:
    ct = f.read()

known_pt = b'Now is a good time to buy stock.'

S = strxor(known_pt[0:10], ct[0:10])

S2 = bytes_to_long(S[0:2])
S3 = bytes_to_long(S[2:4])
S4 = bytes_to_long(S[4:6])
S5 = bytes_to_long(S[6:8])
S6 = bytes_to_long(S[8:10])
```

The second step is to calculate the secret key $A, B, C, S_0, S_1$ by using the $S_4, S_5, S_6$ to solve the linear equation with three unknows, and the derivative process of this equation is shown is graph 2.2

$$S_4 - S_5 = A(S_3 - S_4) + B(S_2 - S_3) \bmod M$$

$$S_5 - S_6 = A(S_4 - S_5) + B(S_3 - S_4) \bmod M$$

$$(S_4 - S_5)(S_3 - S_4)^{-1} = A + B(S_2 - S_3)(S_3 - S_4)^{-1}$$

$$(S_5 - S_6)\cdot(S_4 - S_5)^{-1} = A + B(S_3 - S_4)(S_4 - S_5)^{-1}$$

$$(S_4 - S_5)(S_3 - S_4)^{-1} - (S_5 - S_6)\cdot(S_4 - S_5)^{-1} = B\left[(S_2 - S_3)(S_3 - S_4)^{-1} - (S_3 - S_4)(S_4 - S_5)^{-1}\right]$$

$$B = \left[(S_4 - S_5)(S_3 - S_4)^{-1} - (S_5 - S_6)\cdot(S_4 - S_5)^{-1}\right]\cdot\left[(S_2 - S_3)(S_3 - S_4)^{-1} - (S_3 - S_4)(S_4 - S_5)^{-1}\right]^{-1} \,\%\, M$$

$$A = (S_4 - S_5)(S_3 - S_4)^{-1} - B(S_2 - S_3)(S_3 - S_4)^{-1} \,\%\, M$$

$$C = S_4 - AS_3 - BS_2 \,\%\, M$$

$$S_1 = (S_3 - AS_2 - C)B^{-1} \,\%\, M$$

$$S_0 = (S_2 - AS_1 - C)B^{-1} \,\%\, M$$

graph 2.2

The corresponding python script of second step is shown below.

```
B = (((S4 - S5) * inverse(S3 - S4, 64283) - (S5 - S6) * inverse(S4 - S5, 64283))
     * inverse((S2 - S3) * inverse(S3 - S4, 64283) - (S3 - S4) * inverse(S4 - S5, 64283), 64283)) % 64283
A = ((S4 - S5) * inverse(S3 - S4, 64283) - B * (S2 - S3)* inverse(S3 - S4, 64283)) % 64283
C  = (S4 - A * S3 - B * S2) % 64283
S1 = ((S3 - A * S2 - C) * inverse(B, 64283)) % 64283
S0 = ((S2 - A * S1 - C) * inverse(B, 64283)) % 64283
```

In the last step we only need to use the given elcgcipher.py file and the keys obtained in second step to decrypt the cipher.bin and get the flag. The full python script is provided in appendix.

```
elcgcipher.encfile(S0, S1, A, B, C, "Q2/cipher.bin", "Q2/pt.txt")
```

### Question 3 CTR MAC

In this question, we are required to create a MAC forgery attack by using the fst.bin and the MAC value of it. We first explore the CTRMAC algorithm, it first pads the fst.bin $m_0$ to create $m_1$, and then using the AES-128 to encrypt it using CTR mode to gain $m_2$, next, it rotates each block of $m_2$ by using the index of it, and the $m_3$ will be obtained. The equation of $m_3$ is $m_3[i] = R(m_2[i])^{i+1 \bmod 16}$ where $R(m)^x$ means the $m$ will be rotated to $x$ bytes to the right. And finally, all the blocks in $m_3$ are XORed together to get the final MAC $m_d$:

$$m_d = m_3[0] \oplus m_3[1] \oplus \ldots \oplus m_3[n]$$

Now we explore the fst.bin and the mac1.txt, we find that it is exactly $16 \cdot 18 = 288$ characters, which means it will have 18 blocks. So, if we can find a second pre-image of the fst.bin that has the same MAC value, then we can create a forgery. To find the snd.bin, we need to explore the relation of $m_3, m_2, m_1$ deeply, we first expand the $m_2$ equation as:

$$m_2 = m_1[0] \oplus e_k(IV||CTR_0)||\, m_1[1] \oplus e_k(IV||CTR_1)|| \ldots m_1[n] \oplus e_k(IV||CTR_n)$$

And the $m_3$ is:

$$m_3 = R(m_2[0])^{1\ mod\ 16}||R(m_2[1])^{2\ mod\ 16}|| \ldots R(m_2[n])^{n+1\ mod\ 16}$$

Because the rotation is the byte-wise rotation, and also the $\oplus$ can be considered as performing in byte-wise. Therefore, the linear property will hold:

$$R(X \oplus Y)^i \equiv R(X)^i \oplus R(Y)^i$$

Now apply this property and $m_2$ equation to $m_3$, we will get:

$$m_3[i] = R(m_1[i] \oplus e_k\big(IV\big|\big|CTR_i\big)\big)^{i+1\ mod\ 16}$$

$$= R(m_1[i])^{i+1\ mod\ 16} \oplus R\big(e_k(IV||CTR^i\big)^{i+1\ mod\ 16}$$

Another property of XOR is commutativity property, which can be used in the $m_d$:

$$m_d = R(m_1[0])^1 \oplus R(e_k(IV||CTR_0)^1|| \ldots ||d(m_1[17])^2 \oplus R(e_k(IV||CTR_{17})^{17}$$
$$\equiv [(R(m_1[0])^1 \oplus R(m_1[1])^2 \ldots \oplus R(m_1[17])^2)]$$
$$\oplus [(R(e_k(IV||CTR_0)^1 \oplus R(e_k(IV||CTR_1)^2 \ldots \oplus R(e_k(IV||CTR_{17})^2)]$$

Now we observe that the last part of $m_d$ is $(R(e_k(IV||CTR_0)^1 \oplus R(e_k(IV||CTR_1)^2 \ldots \oplus R(e_k(IV||CTR_{17})^2)$, which is fixed because the $IV$ and the $CTR$ is constant for a specific key. on the other hand, the first part of $m_d$ is manipulable because of the commutativity property of XOR but we have to make sure that the bytes' rotation is correct. I identify two pair of blocks with the same rotation such as $R(m_1)^2, R(m_{17})^{18\ mod\ 16=2}$ and $R(m_0)^1, R(m_{16})^{17\ mod\ 16=1}$ the rotation is the same because they are all rotated to the same number of bytes to the right due to the modulus. The theoretical strategy is exchanging the $2^{th}$ block with $18^{th}$ block, they will produce the same MAC because:

$$(R(m_1[0])^1 \oplus R(m_1[1])^{2\ mod\ 16=2} \ldots \oplus R(m_1[17])^{18\ mod\ 16=2})$$
$$\equiv (R(m_1[0])^1 \oplus R(m_1[17])^{18\ mod\ 16=2} \ldots \oplus R(m_1[1])^{2\ mod\ 16=2})$$

- **Vulnerability**: the vulnerability of this CTRMAC is the use of XOR to chain all the blocks together, which will allow some manipulations to the ciphertext such as blocks exchange. So, if the attacker knows the content of the ciphertext and knows that the same key is used to create MACs, he/she can use the ciphertext to create a forgery attack, and use the same MAC but different plaintext to pass some verification.

The attack strategy is actually discussed above in the exploration section, which is exchange the $2^{th}$ block with $18^{th}$ block of the fst.bin. And python script is not needed to do this job, we just need to open the fst.bin and separate it by 16 characters, then manually exchange the $2^{th}$ block with $18^{th}$ block and put it on snd.bin (check graph 3.1). And the reason why it works is also discussed detailed above. The verification is shown in graph 3.2.

```
JnfWeOOtxZuDntVq JnfWeOOtxZuDntVq
tSSCGJIXxCuefxyh nsLbkkuCwKgVQVGZ
CHCiVkXvldkpPSAU CHCiVkXvldkpPSAU
UDMImPYKqsDEXCQT UDMImPYKqsDEXCQT
lkvkvQBsBHoFWpUL lkvkvQBsBHoFWpUL
lGskvrJBBWlAZOYa lGskvrJBBWlAZOYa
yEHJgKhIfItXhvwL yEHJgKhIfItXhvwL
DhHHkrOjnClnNSkb DhHHkrOjnClnNSkb
roxRkQQPWkGUnJMQ roxRkQQPWkGUnJMQ
esOiwEXAuEQAKCYR esOiwEXAuEQAKCYR
aZCRrbdJxXigEvLh aZCRrbdJxXigEvLh
gJAAbeeryLSbYIar gJAAbeeryLSbYIar
LFgxeJISvYhlToEs LFgxeJISvYhlToEs
JSJxnjDZUBXqWkzF JSJxnjDZUBXqWkzF
MfxMBlEGsORMgkVi MfxMBlEGsORMgkVi
ZSXdCCDlNxzmuWTR ZSXdCCDlNxzmuWTR
scCTbbekBYWUqmbH scCTbbekBYWUqmbH
nsLbkkuCwKgVQVGZ tSSCGJIXxCuefxyh
```

graph 3.1

```
kyrie@kyrie-rao:/mnt/g/ANU2021_S2/2700/a2/Q3$ ./ctrmac.py 00112233445566778899aabbccddeeff fst.bin > mymac.txt
kyrie@kyrie-rao:/mnt/g/ANU2021_S2/2700/a2/Q3$ diff fst.bin snd.bin
1c1
< JnfWeOOtxZuDntVqtSSCGJIXxCuefxyhCHCiVkXvldkpPSAUUDMImPYKqsDEXCQTlkvkvQBsBHoFWpULlGskvrJBBWlAZOYayEHJgKhIfItXhvwLDhHHkrOjnClnNSkbroxRkQQPWkGUn
JMQesOiwEXAuEQAKCYRaZCRrbdJxXigEvLhgJAAbeeryLSbYIarLFgxeJISvYhlToEsJSJxnjDZUBXqWkzFMfxMBlEGsORMgkViZSXdCCDlNxzmuWTRscCTbbekBYWUqmbHnsLbkkuCwKgV
QVGZ
\ No newline at end of file
---
> JnfWeOOtxZuDntVqnsLbkkuCwKgVQVGZCHCiVkXvldkpPSAUUDMImPYKqsDEXCQTlkvkvQBsBHoFWpULlGskvrJBBWlAZOYayEHJgKhIfItXhvwLDhHHkrOjnClnNSkbroxRkQQPWkGUn
JMQesOiwEXAuEQAKCYRaZCRrbdJxXigEvLhgJAAbeeryLSbYIarLFgxeJISvYhlToEsJSJxnjDZUBXqWkzFMfxMBlEGsORMgkViZSXdCCDlNxzmuWTRscCTbbekBYWUqmbHtSSCGJIXxCue
fxyh
\ No newline at end of file
kyrie@kyrie-rao:/mnt/g/ANU2021_S2/2700/a2/Q3$ ./ctrmac.py 00112233445566778899aabbccddeeff snd.bin
6c0e8e41b8fa688a419d56baf203dbc8
kyrie@kyrie-rao:/mnt/g/ANU2021_S2/2700/a2/Q3$ cat mymac.txt
6c0e8e41b8fa688a419d56baf203dbc8
```

graph 3.2

## Question 4 BBC (Bad Block cipher Chaining)

In this question, we are required to recover the flag in cipher2.bin that is encrypted by the BBC by using the known plain text and ciphertext. In the bbc_encrypt function, it first defines an AES cipher to encrypt the plain1.txt in ECB mode, and then the final ciphertext is obtained by iteratively XOR the previous cipher blocks with the AES encrypted plaintext. This is not like traditional CBC, which first XOR the plaintext block with the previous ciphertext block and then apply the AES encryption. The difference is shown below.

$$BBC: y_{i+1} = e_k(x_{i+1}) \oplus y_i$$
$$CBC: y_{i+1} = e_k(x_{i+1} \oplus yi)$$

Note that we can use the property of $x \oplus y = z \equiv x = y \oplus z$ to get the AES encryption of the plaintext blocks of this form $e_k(x_{i+1}) = y_{i+1} \oplus y_i$. And all the $yi$ are known so we can easily know all the blocks of the AES encrypted plain1.txt by using the cipher1.bin. The next exploration step is trying to figure out the IV since we need to compare each $e_k(x_{i+1}) \oplus IV = y_{i+1} \oplus y_i \oplus IV$ with the 16 bytes in cipher2.bin, this is reasonable because we know that the two binary files is encrypted with the same key and IV. However, this idea is abandoned because finding the IV is impossible and the cipher2.bin is not 16 bytes it is 48 bytes and the flag is in the second block of cipher2.bin, so the composition of cipher2.bin is

$$f_1||f_2 = e_k(flag) \oplus f_1||padding$$

We can easily get the $e_k(flag)$, by using the $f_1$ and $f_2$ like $e_k(flag) = f_2 \oplus f_1$. Now, the idea is clear, we know all the $e_k(x_{i+1})$ except the $e_k(x_0)$ since the IV is unknown, and we also know the $e_k(flag)$, therefore we can compare each $e_k(x_{i+1})$ with $e_k(flag)$, if they are match then we know the index of the answer flag. And if it cannot find any match for $\forall i,$ then we know the $e_k(x_0)$ will be the correct encrypted flag.

- **Vulnerability**: As discussed in the exploration above, this modified CBC can be turned into ECB easily by stripping away the previous ciphertext block $y_i$ from each block of the ciphertext. So, all the vulnerabilities of the ECB will be applied to this BBC. Such as
  - ECB encrypts highly deterministically
  - Identical plaintexts are mapped to identical ciphertexts
  - an attacker recognizes if the same message has been sent twice
  - plaintext blocks are encrypted independently of previous blocks

  So once a particular plaintext to ciphertext block mapping $xi \rightarrow yi$ is known, a sequence of ciphertext blocks can be easily manipulated, the practical example is the electronic bank transfer attack. For this question, if the BBC applied in the real-world application, the BBC will be more vulnerable because the IV is generally considered public so we even don't need the extra blocks in cipher2.bin to compare the ciphertext blocks. The attacker just needs a 16 bytes flag encryption with the same key and IV to get the flag.

The attack strategy is then clear, we just need to go through all the 16-bytes blocks (except the first one) in the cipher2.bin, and for each loop, it first calculates the $e_k(x_{i+1})$ by using the previous cipher text block $y_i$ and the current one $y_{i+1}$, and compare it with the $e_k(flag) = f_2 \oplus f_1$, if they are match, the flag will be printed using the index and break the loop. If it reaches the end of cipher2.bin, it will return the first index as the correct one. The Python script and result is shown below (full script see appendix).

```python
with open('Q4/cipher1.bin', 'rb') as f:
    flags = f.read()

with open('Q4/cipher2.bin', 'rb') as f:
    flag = f.read()

block_len = len(flags) // 16

with open('Q4/plain1.txt', 'r') as f:
    pt = f.read().split(' ') # Note that the orignal plain1.txt is change to be seperated by one space

for i in range(block_len):
    if i != 0:
        yi   = flags[i * 16 : (i + 1) * 16]
        yi_1 = flags[(i - 1) * 16 : i * 16]
        if (strxor(flag[0 : 16] ,flag[16 : 32]) == strxor(yi ,yi_1)):
            print(f'Done! E_k(x{i + 1}), with index {i}, flag: {pt[i]}')
            break
    if i == block_len - 1:
        print(f'Done! E_k(x{1}), with index {0}, flag: {pt[0]}')
```

graph code

```
(base) PS G:\ANU2021_S2\2700\a2> & E:/anaconda2021/python.exe g:/ANU2021_S2/2700/a2/Q4/findflag.py
Done! E_k(x13), with index 12, flag: flag{rowdy-sap}
```

graph result

## Question 5 Bad AES

From the problem description, we know this modified AES is an AES without the diffusion layer, and the sample.txt is encrypted using this Bad AES in ECB mode, so every block in simple.txt is encrypted independently from other blocks. The flag.enc is encrypted using Bad AES with the same key that was used to encrypt sample.txt. My initial idea was to write the equations of the relation of the plaintext block and the ciphertext block, the equations is:

$$c_1 = S(S(p_1 \oplus k_0) \oplus k_1) \oplus k_2) \dots) \oplus k_{11}$$
$$c_n = S(S(p_n \oplus k_0) \oplus k_1) \oplus k_2) \dots) \oplus k_{11}$$
$$and$$
$$f_{enc} = S\left(S\left(f_p \oplus k_0\right) \oplus k_1\right) \oplus k_2) \dots) \oplus k_{11}$$

The initial idea is to first XOR the $c_1$ and the $f_{enc}$ to get the

$$c_1 \oplus f_{enc} = S\left(S\left(f_p \oplus k_0\right) \oplus k_1\right) \oplus k_2) \dots) \oplus S(S(p_1 \oplus k_0) \oplus k_1) \oplus k_2) \dots)$$

And then to find out whether this property $S(a) \oplus S(b) \equiv S(a \oplus b)$ is true, if this is true then we can eliminate the key $k_{11} \dots k_0$ from outside to inside, and eventually the only unknown is the flag's plaintext $f_p$. However, this idea is not applicable because byte substitution layer is the nonlinear elements of AES which means $S(a) \oplus S(b) \not\equiv S(a+b)$ so this initial idea is abandoned.

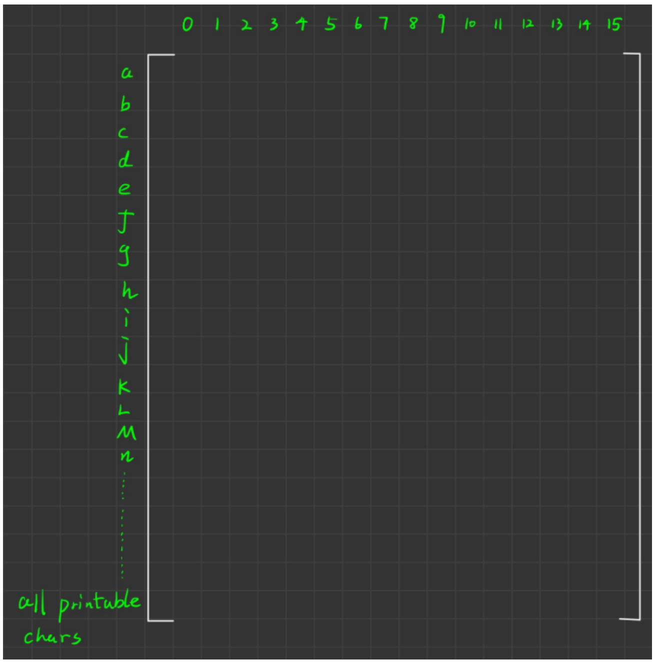
Now I consider what will happen on the AES without the Diffusion layer, the functionality of the diffusion layer is to hide the statistical properties of the plaintext, and the sample.txt is encrypted using ECB mode, which means we might find some mappings from the ciphertext to the plaintext and using that mappings to replace the ciphers in flag.enc and get the plaintext flag. I do some experiments with the sample.txt and sample.enc. some observations are gained.

```
003b6210ce7edfef2a2fa61275e46d13 map with: b'----------------'
003b6210ce7edfef2a2fa61275e46d13 map with: b'----------------'
003b6210ce7edfef2a2fa61275e46d13 map with: b'----------------'
```

This observation reveals that different blocks of 16 bytes “----------------” encryption are the same. And the ‘-’ character in different positions in the block will probably have different encryption’ result but the ‘-’ character in other blocks with the same position will have the same encryption’s result. This is because the key $k_0 \dots k_{11}$ used is the same in different blocks. Now we can make a conclusion:

- the same character in different positions $(0 \le i \le 15)$ in the blocks will have different ciphertext.
- the same character in the same position $(0 \le i \le 15)$ among different blocks will have same ciphertext.

So, the attack strategy will be clear, we just need to build a look-up table by searching the sample.txt and sample.enc where the column is the index of the block, the row is the different printable characters (graph 5.1). And then using the look-up table to find the correct flag.




graph 5.1

- **Vulnerability**: the lack of diffusion layer will not have the influence of one plaintext symbol is spread over many ciphertext symbols, and it will expose the statistical properties of the plaintext. Besides, the ECB mode AES encryption will help the attacker find the mapping of each printable characters to the encrypted one. As a result, the attacker can perform the Substitution Attack. If the attacker manages to get a big plaintext and ciphertext pair, then the attacker can build a look-up matrix and he/she can decrypt all the secret message encrypted with this Bad AES using the same key. This problem can be avoided by using the key once but this is expensive and not practical.

For the attack strategy, it is not necessary to build the look-up table because we only care about the characters of the flag and building the table is expensive. So, the exploitation steps are to create a for loop to go through all the bytes in the flag.enc. For each round, we want to find the ciphertext of the byte in this round, let's call it $flag_i$. For each round, another for loop is created to go through all the plaintext and ciphertext of the sample but we only care about the specific bytes (same position) of it, which is the index i of the $flag_i$, let's call it $txt_i, enc_i$, now we want to check whether $enc_i \equiv flag_i$, if this is true, we find the correct flag's character $txt_i$ and will print it to the screen, and this inner loop will break. If it goes through all the sample and cannot find the mapping, the warning message will be printed. We are lucky that all the encrypted can be matched because the database or corpus (sample.txt) is big enough to matches all the different flag's characters in different 16 positions, and this is the reason why it works. The code and the result are shown below. Full code is in appendix.

```python
with open("Q5/sample.txt", "rb") as f:
    pt = f.read()

with open("Q5/sample.enc", "rb") as f:
    sample = f.read()

with open("Q5/flag.enc", "rb") as f:
    flag = f.read()

result = ""
num_boc = len(sample) // 16
for w in range(32):
    ct_w = flag[w : w + 1]
    for i in range(num_boc):
        s = i * 16 + w
        if ct_w == sample[s : s + 1] :
            print(f'{sample[s : s + 1].hex()} == {pt[s : s + 1]}')
            result = result + str(pt[s : s + 1], encoding = "utf-8")
            break
        if i == num_boc - 1:
            print(f'Cannot find the plaint text')

print(result)
```

```
(base) PS G:\ANU2021_S2\2700
a7 == b'f'
5c == b'l'
75 == b'a'
b6 == b'g'
58 == b'{'
b1 == b'l'
89 == b'i'
ae == b't'
78 == b't'
4e == b'l'
4d == b'e'
12 == b'-'
e7 == b'e'
41 == b'n'
61 == b'v'
bd == b'e'
9c == b'l'
23 == b'o'
b1 == b'p'
7b == b'e'
e6 == b'}'
Cannot find the plaint text
Cannot find the plaint text
Cannot find the plaint text
Cannot find the plaint text
Cannot find the plaint text
Cannot find the plaint text
Cannot find the plaint text
Cannot find the plaint text
Cannot find the plaint text
Cannot find the plaint text
Cannot find the plaint text
flag{little-envelope}
```

## Question 6 CFB Hash

For this question, the target is to create a second-preimage of the file fst.bin so that they have the same hash value of the CFB Hash. First, we look at the implementation of the CFB Hash. The CFB hash first appends the length of the plaintext to itself and pad the plaintext for block size of 32 bytes, then using the AES-128 CFB mode to encrypt the message, finally, extract the last block of the ciphertext as the hash value. There are a few points worth noting:

- It uses the last block of the ciphertext as the hash value, which means we only need to create a second-preimage that has the same last block.
- It appends the length to the plaintext and pads it, which means our second-preimage's length is fixed, the length should be the same as fst.bin.
- It uses the CFB mode to encrypt the plaintext, the CFB chains previous ciphertext block with the plaintext block using XOR, which means we can build the snd.bin based on the property of CFB.

Now we explore the fst.bin, we first check the length of the it, it has 48 bytes, so we can divide it into three blocks and call it $m_1, m_2, m_3$. According to the algorithm, the plaintext will be appended with the length of it and will be padded for block size 32, and padding size will be $32 \cdot n = p + 48 + 4\ (n \in \mathbb{N}) \rightarrow p = 12, n = 2$ so the plaintext will be $m_1, m_2, m_3, lp$. According to the code analysis above, the snd.bin should have the same length as fst.bin, so snd.bin should be $m_1', m_2', m_3', lp$. Now we explore the relation between this four blocks, we can use the CFB definition to write it like:

$$
\begin{gathered}
y_1 = e_k(IV) \oplus m_1 \\
y_2 = e_k(y_1) \oplus m_2 \\
y_3 = e_k(y_2) \oplus m_3 \\
h = y_4 = e_k(y_3) \oplus lp
\end{gathered}
$$

$$
\begin{gathered}
y_1' = e_k(IV) \oplus m_1' \\
y_2' = e_k(y_1') \oplus m_2' \\
y_3' = e_k(y_2') \oplus m_3' \\
h = y_4' = e_k(y_3') \oplus lp
\end{gathered}
$$

Our goal is to let $y_4 = y_4'$ and we know the $lp$ is the same, so we can infer that $y_3 = y_3'$. To achieve this, $m_3'$ should be something like $y_3 \oplus m_p$, the $m_p$ is used to eliminated the $e_k(y_2')$. Now assume that $e_k(y_2') = y_2$, so $m_p$ will be $y_2$ and we can get

$$
\begin{gathered}
y_3' = e_k(y_2') \oplus m_3' = y_2 \oplus y_3 \oplus y_2 = y_3 \\
y_2' = d_k(y_2)
\end{gathered}
$$

The second equation holds because the property of AES symmetric key encryption

$$
d_k(e_k(x)) = x, \qquad e_k(d_k(x)) = x
$$

Now we focus on the $y_2' = d_k(y_2)$, so the $m_2'$ should be $d_k(y_2) \oplus m_p$ where $m_p$ is used to eliminate the $e_k(y_1')$. Now assume that $e_k(y_1') = y_1$, so $m_p$ will be $y_1$ and we can get

$$y_2' = e_k(y_1{}') \oplus m_2' = y_1 \oplus y_1 \oplus d_k(y_2) = d_k(y_2)$$
$$y_1' = d_k(y_1)$$

To get $y_1' = d_k(y_1)$, the $m_1{}'$ should be $d_k(y_1) \oplus m_p$ where $m_p$ is used to eliminate the $e_k(IV)$, therefore the $m_p$ will be $e_k(IV)$ and we get

$$y_1' = e_k(IV) \oplus m_1' = e_k(IV) \oplus e_k(IV) \oplus d_k(y_1) = d_k(y_1)$$

Now we group together the $m_1', m_2', m_3'$ and rewrite the $y_1', y_2', y_3'$ as

$$m_1{}' = d_k(y_1) \oplus e_k(IV)$$
$$m_2{}' = d_k(y_2) \oplus y_1$$
$$m_3{}' = y_3 \oplus y_2$$

$$y_1{}' = e_k(IV) \oplus d_k(y_1) \oplus e_k(IV)$$
$$y_2{}' = e_k(y_1{}') \oplus d_k(y_2) \oplus y_1$$
$$y_3{}' = e_k(y_2{}') \oplus y_3 \oplus y_2$$
$$h = y_4' = e_k(y_3{}') \oplus lp$$
$$y_4 = y_4{}'$$

Note that the $m_1', m_2', m_3{}'$ is distinct from original $m_1, m_2, m_3$ (below)

$$m_1 = e_k(IV) \oplus y_1$$
$$m_2 = e_k(y_1) \oplus y_2$$
$$m_3 = e_k(y_2) \oplus y_3$$

Because $y_1 \neq d_k(y_1)$ and then $e_k(IV) \oplus y_1 \neq d_k(y_1) \oplus e_k(IV) \rightarrow m_1' \neq m_1$. Now the second-preimage $m_1', m_2', m_3{}'$ is built.

- **Vulnerability**: This CFB hash program is dangerous when the key and IV are exposed since we need to use the key and IV to find all the ciphertext, and use the key to do some decryptions to create the second-preimage of a specific file. So, if the attacker can get the key and the IV, the attacker can easily create a second-preimage of a file.

For the attack strategy, we need to create the second-preimage by using the given key and IV, the first thing we need to do is to get the value of $y_1, y_2, y_3$, we know that they are obtained by using AES in CFB mode so we create a function cfbAES, this function and the cfbhash in cfbhash.py are almost the same except cfbAES returns the entire ciphertext rather than the last block. Function defined below.

```python
def cfbAES(data):
    cipher = AES.new(key, AES.MODE_CFB, iv=iv, segment_size=128)
    l = len(data)

    if l >= (2 ** 32):
        print("Input too long")
        raise ValueError

    # encode length of data in 4 bytes
    lb = long_to_bytes(l, 4)
    # add length to data and apply PKCS#7 padding

    data = pad(data+lb, 32)
    ct = cipher.encrypt(data)
    return ct
```

Now use that function and the given IV, key, fst.bin to get the $y_1, y_2, y_{3,}y_4$ as follow

```python
with open("Q6/fst.bin", "rb") as f:
    data = f.read()

key  = b'AES-HASH-1234567'
iv   = b'0123456789abcdef'

ori_ct = cfbAES(data)

y1 = ori_ct[0:16]
y2 = ori_ct[16:32]
y3 = ori_ct[32:48]
y4 = ori_ct[48:]
```

Next, we need to create the $m_1', m_2', m_3{}'$ defined above, which needs the AES decryption and AES encryption because we need to calculate the $e_k(IV), d_k(y_1), d_k(y_2)$ so we need to create a AES class with ECB mode using the same key. Finally, we can concatenate all the $m_k{}'$s together to create the second-preimage snd.bin. Process shown below. (full code in appendix)

```python
cipher = AES.new(key, mode=AES.MODE_ECB)

m1 = strxor(cipher.decrypt(y1), cipher.encrypt(iv))
m2 = strxor(y1, cipher.decrypt(y2))
m3 = strxor(y2, y3)

m   = m1 + m2 + m3
with open("Q6/snd.bin",'wb') as f:
    f.write(m)
```

We can further test this snd.bin by comparing the $y_1, y_2, y_3, y_4$ to $y_1', y_2', y_3', y_4'$. The analysis above tells us that $y_1 \neq y_1', y_2 \neq y_2', y_3 = y_3', y_4 = y_4'$. The testing code and result are shown below.

```
with open("Q6/snd.bin",'rb') as f:
    rm = f.read()
new_ct = cfbAES(rm)

y1_snd = new_ct[0:16]
y2_snd = new_ct[16:32]
y3_snd = new_ct[32:48]
y4_snd = new_ct[48:]

print(y1_snd.hex())
print(y2_snd.hex())
print(y3_snd.hex())
print(y4_snd.hex())

print("---------------------------testing---------------------")
print(f'y1 != y1_snd -> {y1 == y1_snd}')
print(f'y2 != y2_snd -> {y2 == y2_snd}')
print(f'y3 == y3_snd -> {y3 == y3_snd}')
print(f'y4 == y4_snd -> {y4 == y4_snd}')
```

```
---------------------------testing---------------------
y1 != y1_snd -> False
y2 != y2_snd -> False
y3 == y3_snd -> True
y4 == y4_snd -> True
```

## Appendix

### Q2

```
from Crypto.Util.number import *
from Crypto.Util.strxor import *

import elcgcipher

with open('Q2/cipher.bin', 'rb') as f:
    ct = f.read()

known_pt = b'Now is a good time to buy stock.'

S = strxor(known_pt[0:10], ct[0:10])

S2 = bytes_to_long(S[0:2])
S3 = bytes_to_long(S[2:4])
S4 = bytes_to_long(S[4:6])
S5 = bytes_to_long(S[6:8])
S6 = bytes_to_long(S[8:10])
```

```
print("S2 = %d, S3 = %d, S4 = %d, S5 = %d, S6 = %d" %
(S2,S3,S4,S5,S6))
print("--------------------------------------------------------------------
-----------")

B = (((S4 - S5) * inverse(S3 - S4, 64283) - (S5 - S6) * inverse(S4 -
S5, 64283))
     * inverse((S2 - S3) * inverse(S3 - S4, 64283) - (S3 - S4) *
inverse(S4 - S5, 64283), 64283)) % 64283
A = ((S4 - S5) * inverse(S3 - S4, 64283) - B * (S2 - S3)* inverse(S3
- S4, 64283)) % 64283
C  = (S4 - A * S3 - B * S2) % 64283
S1 = ((S3 - A * S2 - C) * inverse(B, 64283)) % 64283
S0 = ((S2 - A * S1 - C) * inverse(B, 64283)) % 64283

print("The key is (S0 = %d, S1 = %d, A = %d, B = %d, C = %d)" % (S0,
S1, A, B, C))
print("--------------------------------------------------------------------
-----------")

elcgcipher.encfile(S0, S1, A, B, C, "Q2/cipher.bin", "Q2/pt.txt")
```

**Q4**

```
from Crypto.Util.number import *
from Crypto.Util.strxor import *
from Crypto.Cipher import AES


with open('Q4/cipher1.bin', 'rb') as f:
    flags = f.read()

with open('Q4/cipher2.bin', 'rb') as f:
    flag = f.read()

block_len = len(flags) // 16
# ref.txt is the orignal plain1.txt changed to be seperated by one
space amoung flags
with open('Q4/ref.txt', 'r') as f:
    pt = f.read().split(' ')

for i in range(block_len):
    if i != 0:
        yi   = flags[i * 16 : (i + 1) * 16]
        yi_1 = flags[(i - 1) * 16 : i * 16]
```

```
        if (strxor(flag[0 : 16] ,flag[16 : 32]) == strxor(yi ,yi_1)):
            print(f'Done! E_k(x{i + 1}), with index {i}, flag:
{pt[i]}')
            break
    if i == block_len - 1:
        print(f'Done! E_k(x{1}), with index {0}, flag: {pt[0]}')
```

**Q5**

```
from Crypto.Cipher import AES
from Crypto.Util.number import *
from Crypto.Util.strxor import *
from Crypto.Util.Padding import pad

with open("Q5/sample.txt", "rb") as f:
    pt = f.read()

with open("Q5/sample.enc", "rb") as f:
    sample = f.read()

with open("Q5/flag.enc", "rb") as f:
    flag = f.read()

result = ""
num_boc = len(sample) // 16
for w in range(32):
    ct_w = flag[w : w + 1]
    for i in range(num_boc):
        s = i * 16 + w
        if ct_w == sample[s : s + 1] :
            print(f'{sample[s : s + 1].hex()} == {pt[s : s + 1]}')
            result = result + str(pt[s : s + 1], encoding = "utf-8")
            break
        if i == num_boc - 1:
            print(f'Cannot find the plaint text')

print(result)
```

**Q6**

```
from Crypto.Cipher import AES
from Crypto.Util.number import *
from Crypto.Util.strxor import *
from Crypto.Util.Padding import pad
```

```python
def cfbAES(data):
    cipher = AES.new(key, AES.MODE_CFB, iv=iv, segment_size=128)
    l = len(data)

    if l >= (2 ** 32):
        print("Input too long")
        raise ValueError

    lb = long_to_bytes(l, 4)

    data = pad(data+lb, 32)
    ct = cipher.encrypt(data)
    return ct

with open("Q6/fst.bin", "rb") as f:
    data = f.read()

key = b'AES-HASH-1234567'
iv  = b'0123456789abcdef'

ori_ct = cfbAES(data)

y1 = ori_ct[0:16]
y2 = ori_ct[16:32]
y3 = ori_ct[32:48]
y4 = ori_ct[48:]

print(y1.hex())
print(y2.hex())
print(y3.hex())
print(y4.hex())

cipher = AES.new(key, mode=AES.MODE_ECB)

m1 = strxor(cipher.decrypt(y1), cipher.encrypt(iv))
m2 = strxor(y1, cipher.decrypt(y2))
m3 = strxor(y2, y3)

m  = m1 + m2 + m3
with open("Q6/snd.bin",'wb') as f:
    f.write(m)

print("testing whether new_m == ori_m: " + str(m == data))
```

```
print("--------------------------------------------------------")

with open("Q6/snd.bin",'rb') as f:
    rm = f.read()
new_ct = cfbAES(rm)

y1_snd = new_ct[0:16]
y2_snd = new_ct[16:32]
y3_snd = new_ct[32:48]
y4_snd = new_ct[48:]

print(y1_snd.hex())
print(y2_snd.hex())
print(y3_snd.hex())
print(y4_snd.hex())

print("----------------------------testing----------------------")
print(f'y1 != y1_snd -> {y1 == y1_snd}')
print(f'y2 != y2_snd -> {y2 == y2_snd}')
print(f'y3 == y3_snd -> {y3 == y3_snd}')
print(f'y4 == y4_snd -> {y4 == y4_snd}')
```