lab2

Haotian An(han11), xiangjun Ma(xma39), Ziyang Lin(zli32)

Task2

In this task, we chose AES-128-CBC, BF-CFB, and DES to encrypt the file.

For different plaintext, we are encrypting 3 different files. The first two are txt files with different contents, and the third is a png image.

AES-128-CBC

Firstly, under `AES-128-CBC` mode. The encryption using openssl is in command:

```
openssl enc -aes-128-cbc -e -in plain1.txt -out cipher1.bin -K
00112233445566778899aabbccddeeff -iv 01020304050607080910111213141516
```

After Running the command to encrypt three different files, we can see they have all successfully been encrypted. Looking at each file, we cannot tell the original message in each file.

```
parallels@parallels-Parallels-Virtual-Platform:~/Desktop/Intro-Crypto/lab2-2$ op
enssl enc -aes-128-cbc -e -in plain1.txt -out cipher1.bin -K 0011223344556677889
9aabbccddeeff -iv 01020304050607080910111213141516
parallels@parallels-Parallels-Virtual-Platform:~/Desktop/Intro-Crypto/lab2-2$ op
enssl enc -aes-128-cbc -e -in plain2.txt -out cipher2.bin -K 0011223344556677889
9aabbccddeeff -iv 01020304050607080910111213141516
parallels@parallels-Parallels-Virtual-Platform:~/Desktop/Intro-Crypto/lab2-2$ op
enssl enc -aes-128-cbc -e -in image1.png -out cipher3.bin -K 0011223344556677889
9aabbccddeeff -iv 01020304050607080910111213141516
```

For the first two txt file, we can see their message is unreadable:

```
cipher1.bin ×
00000000 C4 C0 32 A8 AC 6F 6C B7 08 06 F2 E5 0E 6E 33 A0 4A 15 71 81 12 ..2..ol......n3.J.q..
00000015 11 36 6F 8D 96 A8 A9 4A AC DF 7E                               .6o....J..~
```

```
cipher2.bin ×
00000000 CF 09 C8 AB 20 8C 51 20 CE D6 40 8D D5 E1 9E 1B D5 BD 3C A4 38 .... .Q ..@.......<.8
00000015 62 22 A6 E2 45 D5 AD 1F 3F AD 69                               b"..E...?.i
```

For the third png file, the cipher file is not readable, also the content makes no sense.

cipher3.bin

```
0000147c 16 93 C7 8E BC 54 08 37 53 31 4C 87 CD F8 E7 6C 31 2F 8F 58 0E 17 EB .....T.7S1L....l1/.X...
00001493 17 12 1E 31 68 8A 32 78 AF 7C 06 5C 20 09 88 7B CC 76 2C 2A F7 EB E3 ...1h.2x.|.\ ..{.v,*...
000014aa 12 C7 4A 9F 07 BF 90 97 F9 89 B9 76 19 BC 41 B0 64 14 56 85 67 23 79 ..J........v..A.d.V.g#y
000014c1 61 07 1D 11 FD E2 6F 4D FF 11 F7 3D 22 FA 11 E8 B6 96 83 8F 57 EB 5E a.....oM...="........W.^
000014d8 37 D7 A3 34 3B 49 58 10 8A 0A 77 A4 16 00 A8 DD 42 F3 20 96 52 D0 F8 7..4;IX...w.....B. .R..
000014ef 49 B4 BD B4 63 F4 65 23 55 26 68 6D A9 7E FE 8D 0B 0C 4B BA 5D 12 19 I...c.e#U&hm.~....K.]..
00001506 73 FE 69 8F E3 AE 6B C4 50 EB 2B 90 68 26 26 AE 6F E8 BA 89 0E 6C 8C s.i...k.P.+.h&&.o....l.
0000151d 64 CE DF 9D 0A 86 47 A8 D5 DD 1B 6F BC 4F DA 97 A2 15 8A A8 9E 91 32 d.....G....o.O........2
00001534 CB D9 10 43 87 51 A4 9F D0 BE 72 07 18 0A E0 A0 8F 5C 31 C2 07 02 95 ...C.Q....r......\1....
0000154b D3 27 E1 A1 B2 80 02 5A 57 B5 03 4C 39 DD 7F 50 78 E4 DE 2B 4B 38 60 .'.....ZW..L9..Px..+K8`
00001562 45 B0 BF EF 3A E7 3F E7 5D 5C 45 07 CD 4F 75 9E B6 57 3A 22 A6 2B BF E...:.?.]\E..Ou..W:".+.
00001579 36 AF D5 5C F1 67 D3 72 DE CD FC 81 89 9A 3E B3 E7 87 5C DF 22 FF E7 6..\.g.r......>...\."..
00001590 2A 1C AD 7A AE 30 1D A1 8D 47 A6 D8 E2 F1 D3 90 D5 8A 79 20 1E BE 61 *..z.0...G........y ..a
000015a7 66 9A 43 A9 EB 9D F3 1E D2 26 2A 26 0D 0D ED A6 03 81 E1 B2 E8 75 6B f.C......&*&.........uk
000015be 1F FD 84 69 99 EE E4 C4 B2 A3 79 D5 AB D8 8B AD A7 6D 36 EA F2 66 38 ...i......y......m6..f8
000015d5 3E F9 B8 5B 31 3B E3 7D 56 14 D9 33 B3 7D 15 56 8D 93 BA 15 8F D5 0B >..[1;.}V..3.}.V.......
000015ec 29 C5 2E 7B 20 92 0D 96 B3 64 9F B3 1A E1 B9 61 DB 03 DA 6D EB 4E F0 )..{ ....d.....a...m.N.
00001603 88 3B 8A 41 68 8D 5E 02 9A 36 76 82 09 D7 42 C0 EB 64 FA 3A B1 A2 7F .;.Ah.^..6v...B..d.:...
0000161a EE AA 48 49 1D E7 14 38 C1 61 BA ED 81 98 42 F1 26 A3 06 F8 79 EA E3 ..HI...8.a....B.&...y..
```

BF-CFB

After the first mode, we tried CFB mode:

```
openssl enc -bf-cfb -e -in image1.png -out cipher3.bin -K 00112233445566778899aabbccddeeff -iv 0102030405060708
```

```
parallels@parallels-Parallels-Virtual-Platform:~/Desktop/Intro-Crypto/lab2-2$ openssl enc -bf-cfb -e -in plain1.txt -out cipher1.bin -K 00112233445566778899aabbccddeeff -iv 0102030405060708
parallels@parallels-Parallels-Virtual-Platform:~/Desktop/Intro-Crypto/lab2-2$ openssl enc -bf-cfb -e -in plain2.txt -out cipher2.bin -K 00112233445566778899aabbccddeeff -iv 0102030405060708
parallels@parallels-Parallels-Virtual-Platform:~/Desktop/Intro-Crypto/lab2-2$ openssl enc -bf-cfb -e -in image1.png -out cipher3.bin -K 00112233445566778899aabbccddeeff -iv 0102030405060708
```

As we can see they also compiled and encrypted successfully. Again, same as the first task, the content is unreadable and makes no sense simply by looking at them. So we are not showing the encrypted files here.

AES-128-CFB

At last, we tried AES-128-CFB mode:

```
openssl enc -aes-128-cfb -e -in plain1.txt -out cipher1.bin -K 00112233445566778899aabbccddeeff -iv 0102030405060708090101112131415 16
```

```
parallels@parallels-Parallels-Virtual-Platform:~/Desktop/Intro-Crypto/lab2-2$ openssl enc -aes-128-cfb -e -in plain1.txt -out cipher1.bin -K 00112233445566778899aabbccddeeff -iv 0102030405060708090101112131415 16
parallels@parallels-Parallels-Virtual-Platform:~/Desktop/Intro-Crypto/lab2-2$ openssl enc -aes-128-cfb -e -in plain2.txt -out cipher2.bin -K 00112233445566778899aabbccddeeff -iv 0102030405060708090101112131415 16
parallels@parallels-Parallels-Virtual-Platform:~/Desktop/Intro-Crypto/lab2-2$ openssl enc -aes-128-cfb -e -in image1.png -out cipher3.bin -K 00112233445566778899aabbccddeeff -iv 0102030405060708090101112131415 16
```

As we can see they all encrypted successfully. As always, the ciphertext is unreadable and looks the same as task1, thus not included here.

Task3

Code snippet

pic_original

```
# ECB encryption
openssl enc --aes-128-ecb -e -in pic_original.bmp -out ECB_ENC.bmp -K
00112233445566778889aabbccddeeff
head -c 54 pic_original.bmp > original_header
tail -c +55 ECB_ENC.bmp > ECB_body
cat original_header ECB_body > ECB_result.bmp
```

```
# CBC encryption
openssl enc --aes-128-cbc -e -in pic_original.bmp -out CBC_ENC.bmp -K
00112233445566778889aabbccddeeff -iv 0102030405060708
head -c 54 pic_original.bmp > original_header
tail -c +55 CBC_ENC.bmp > CBC_body
cat original_header CBC_body > CBC_result.bmp
```

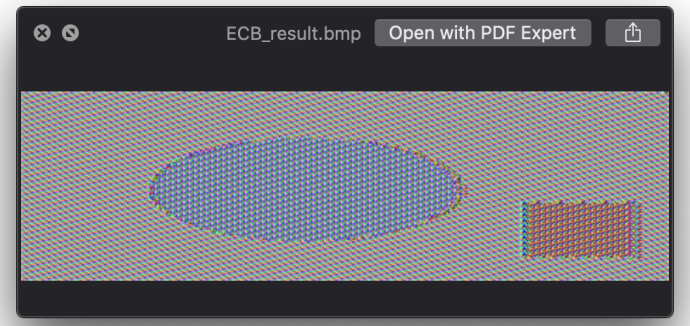
pic_own

```
# ECB encryption
openssl enc --aes-128-ecb -e -in pic_own.bmp -out ECB_ENC.bmp -K
00112233445566778889aabbccddeeff
head -c 54 pic_own.bmp > own_header
tail -c +55 ECB_ENC.bmp > ECB_body
cat own_header ECB_body > ECB_result.bmp
```

```
# CBC encryption
openssl enc --aes-128-cbc -e -in pic_own.bmp -out CBC_ENC.bmp -K
00112233445566778889aabbccddeeff -iv 0102030405060708
head -c 54 pic_own.bmp > own_header
tail -c +55 CBC_ENC.bmp > CBC_body
cat own_header CBC_body > CBC_result.bmp
```

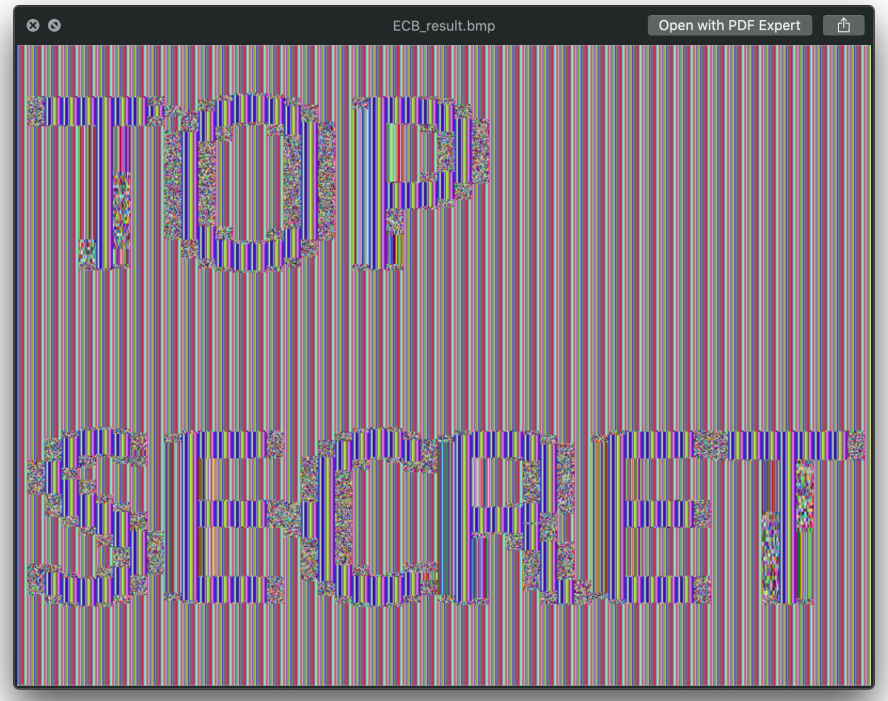
Result

pic_original after ECB encryption

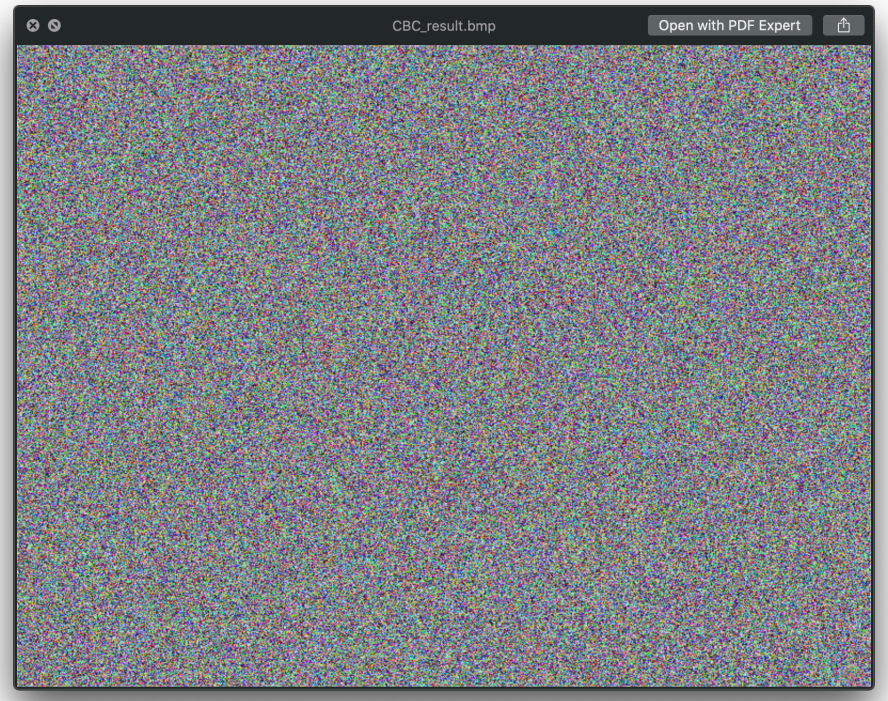


pic_original after CBC encryption

pic_own after ECB encryption



pic_own after CBC encryption



Observation

1. ECB mode encryption is not secure and exposes a lot of information, in the experiment, we can tell the pictures' shape and color distribution of the encrypted pictures, and we can read the word's from the second encrypted picture
2. CBC mode encryption can well protect the pictures, it not only encrypts picture's pixels message like the colors but also makes the distribution of the pixels "random", and it's data pattern can not be observed

explanation

1. ECB mode encrypts data identically, which means, although inside each cipher block, the data is well protected, it expose it's data patterns as a group of blocks, that's why it lacks of diffusion and do not provide confidentiality.
2. CBC mode doesn't have the drawback of lacking confidentiality, if you look at its encryption scheme, it has the "avalanche" effect, which hides the plaintext data patterns

Task 4

First, we created a 7-byte file named “t4.txt”. use this file as material to try the encryption.

```
yang@yang-VirtualBox:~/Desktop/origin/lab/lab2$ echo -n "1234567" > t4.txt
```

Then we encrypted the file with ECB, CBC, CFB, OFB respectively and get the output “t4ecb.bin”,”t4cbc.bin”,”t4cfb.bin”,”t4ofb.bin”.

The key is 00112233445566778899aabbccddeeff and the iv is 01020304050607080910111213141516

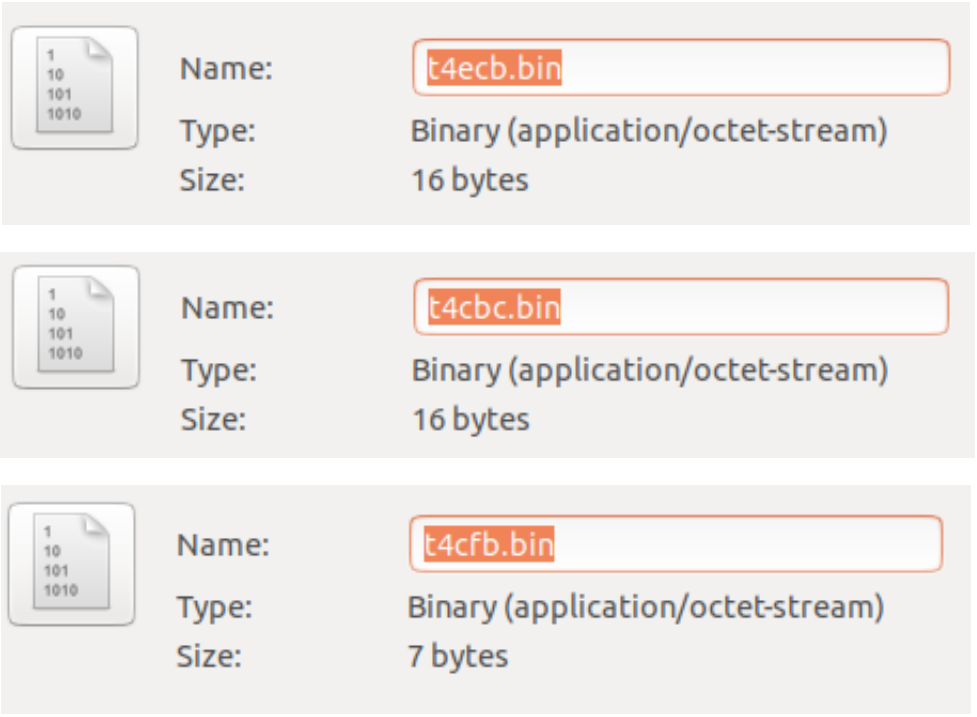
```
yang@yang-VirtualBox:~/Desktop/origin/lab/lab2$ openssl enc -aes-128-ecb -e
 -in t4.txt -out t4ecb.bin -K 00112233445566778899aabbccddeeff
```

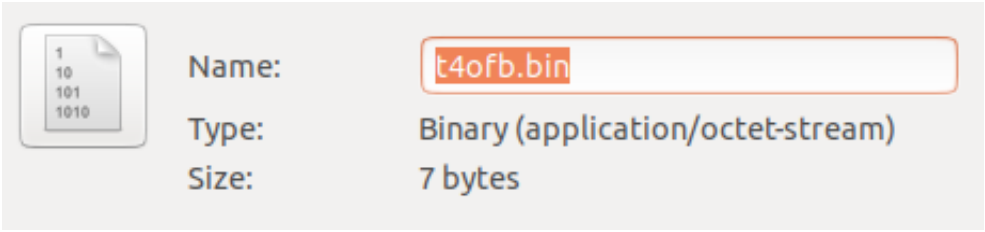
```
yang@yang-VirtualBox:~/Desktop/origin/lab/lab2$ openssl enc -aes-128-cbc -e
 -in t4.txt -out t4cbc.bin -K 00112233445566778899aabbccddeeff -iv 01020304
05060708091011121314151 6
```

```
yang@yang-VirtualBox:~/Desktop/origin/lab/lab2$ openssl enc -aes-128-cfb -e
 -in t4.txt -out t4cfb.bin -K 00112233445566778899aabbccddeeff -iv 01020304
05060708091011121314151 6
```

```
yang@yang-VirtualBox:~/Desktop/origin/lab/lab2$ openssl enc -aes-128-ofb -e
 -in t4.txt -out t4ofb.bin -K 00112233445566778899aabbccddeeff -iv 01020304
05060708091011121314151 6
```

We can see that use ECB and CBC will have 9 bytes padding and use CFB and OFB will not have any padding.





CFB and OFB mode are the mode that first get the encrypted iv and using the encrypted iv XOR the plaintext to encrypt the messages. That means that these two modes can encrypt any length of message. No padding can still encrypt the plaintext messages.

2

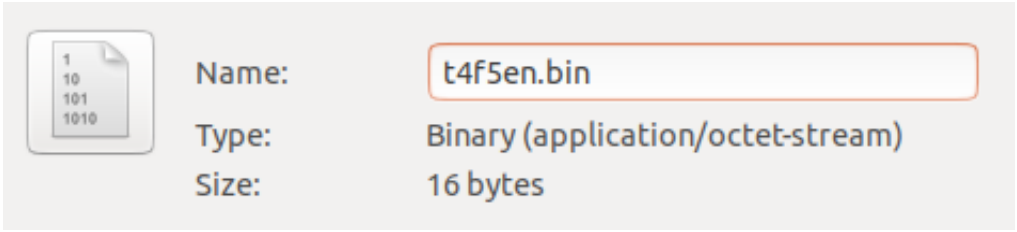
We created the files using “echo -n” command and we got 5-byte file “t4f5.txt”, 10-byte file “t4f10.txt”, and 16-byte file “t4f16.txt”

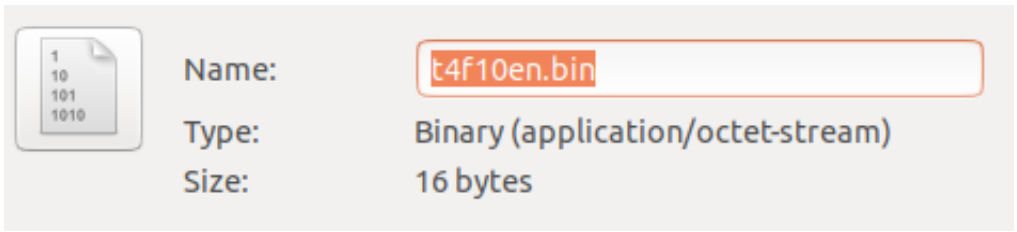
```
yang@yang-VirtualBox:~/Desktop/origin/lab/lab2$ echo -n "12345" > t4f5.txt
yang@yang-VirtualBox:~/Desktop/origin/lab/lab2$ echo -n "0123456789" > t4f10.txt
yang@yang-VirtualBox:~/Desktop/origin/lab/lab2$ echo -n "0123456789012345" > t4f16.txt
```

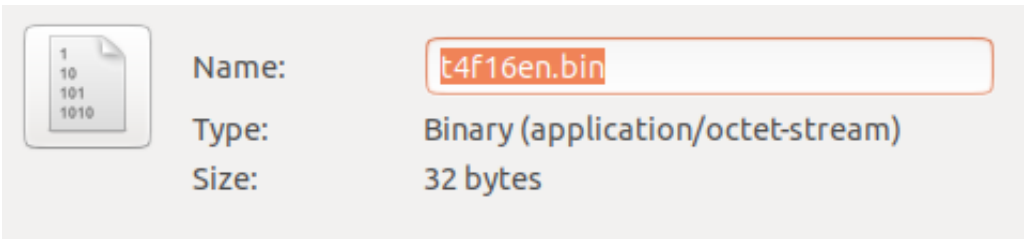
Then we used CBC to encrypt the files respectively and output are “t4f5.bin”, “t4f10.bin”, and “t4f16.bin”, using the same key 00112233445566778899aabbccddeeff and the same iv 0102030405060708091011121314151 6.

```
yang@yang-VirtualBox:~/Desktop/origin/lab/lab2$ openssl enc -aes-128-cbc -e -in t4f5.tx
t -out t4f5en.bin -K 00112233445566778899aabbccddeeff -iv 0102030405060708091011121314
516
yang@yang-VirtualBox:~/Desktop/origin/lab/lab2$ openssl enc -aes-128-cbc -e -in t4f10.t
xt -out t4f10en.bin -K 00112233445566778899aabbccddeeff -iv 0102030405060708091011121
41516
yang@yang-VirtualBox:~/Desktop/origin/lab/lab2$ openssl enc -aes-128-cbc -e -in t4f16.t
xt -out t4f16en.bin -K 00112233445566778899aabbccddeeff -iv 0102030405060708091011121
41516
```

Then we can see that all three files have padding. The 5-byte file and 10-byte became 16-byte, and 16-byte file became 32-byte.



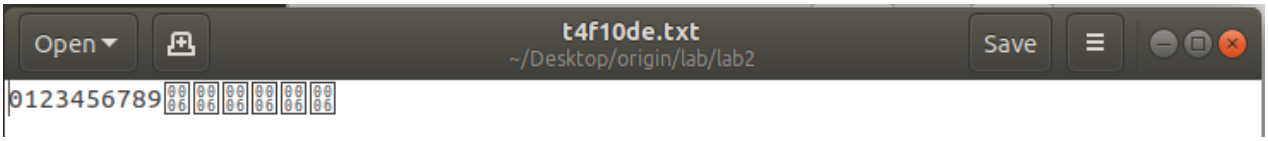




Then we decrypted the files and the output are “t4f5de.bin”, “t4f10de.bin” and “t4f16de.bin”.

```
yang@yang-VirtualBox:~/Desktop/origin/lab/lab2$ openssl enc -aes-128-cbc -d -in t4f5en.
bin -out t4f5de.txt -K 00112233445566778899aabbccddeeff -iv 0102030405060708091011121 31
41516 -nopad
yang@yang-VirtualBox:~/Desktop/origin/lab/lab2$ openssl enc -aes-128-cbc -d -in t4f10en
.bin -out t4f10de.txt -K 00112233445566778899aabbccddeeff -iv 0102030405060708091011121
3141516 -nopad
yang@yang-VirtualBox:~/Desktop/origin/lab/lab2$ openssl enc -aes-128-cbc -d -in t4f16en
.bin -out t4f16de.txt -K 00112233445566778899aabbccddeeff -iv 0102030405060708091011121
3141516 -nopad
```

Open the file we can see these results.



Then we use “hexdump” to open the files. The content of file are displayed like this.

Task 5

Python is used for this task, the backend is default which is same as Openssl. To flip one bit, I first read the 55th bit, and manually add 1 or subtract 1 depending on the result.

code snippet

prepare data, key, iv

```
with open('task5/original1.txt', "rb") as reader:
    data = bytearray(reader.read())
    padding = b"\x00" * (16 - (len(data) % 16) % 16)
    data += padding
    print(data[54:56].hex())
key = bytes.fromhex(
    "00112233445566778889aabbccddeeff00112233445566778889aabbccddeeff")
iv = bytes.fromhex("0102030405060708010203040506070 8")
```

ECB mode experiment

```
ECB_cipher = Cipher(algorithms.AES(key), modes.ECB(),
                    backend=default_backend())
ECB_enc = ECB_cipher.encryptor()
ECB_dec = ECB_cipher.decryptor()
# get cipher text
ct = bytearray(ECB_enc.update(data))
with open("task5/ecb_dec.txt", "wb") as writer:
    ct[55] = ct[55]-1
    pt = ECB_dec.update(ct)
    writer.write(pt)
    print_compare()
```

CBC mode experiment

```
CBC_cipher = Cipher(algorithms.AES(key), modes.CBC(iv),
                    backend=default_backend())
CBC_enc = CBC_cipher.encryptor()
CBC_dec = CBC_cipher.decryptor()
ct = bytearray(CBC_enc.update(data))
with open("task5/cbc_dec.txt", "wb") as writer:
    ct[55] += 1
    pt = CBC_dec.update(ct)
    writer.write(pt)
    print_compare()

```

CFB mode experiemnt

```
CBC_cipher = Cipher(algorithms.AES(key), modes.CBC(iv),
                    backend=default_backend())
CBC_enc = CBC_cipher.encryptor()
CBC_dec = CBC_cipher.decryptor()
ct = bytearray(CBC_enc.update(data))
with open("task5/cbc_dec.txt", "wb") as writer:
    ct[55] += 1
    pt = CBC_dec.update(ct)
    writer.write(pt)
    print_compare()
```

OFB mode experiment

```
OFB_cipher = Cipher(algorithms.AES(key), modes.OFB(iv),
                    backend=default_backend())
OFB_enc = OFB_cipher.encryptor()
OFB_dec = OFB_cipher.decryptor()
ct = bytearray(OFB_enc.update(data))
with open("task5/ofb_enc.txt", "wb") as writer:
    writer.write(ct)
with open("task5/ofb_dec.txt", "wb") as writer:
    ct[55] += 1
    pt = OFB_dec.update(ct)
    writer.write(pt)
    print_compare()

```

How much information can recover after corruption

ECB

Only 1 block(16 bytes) of information is corrupted ,and that's because, in ECB mode's decryption, there's no interaction between the blocks when decrypting in this mode

CBC

In CBC mode, if 1 block is broken, then 2 blocks will be affected, that's because to decrypt a current block, it needs the previous cipher block as the input for this message block, so the 3rd block will affect both 3rd and 4th blocks in decryption. Also, while the whole 3rd block is corrupted, the 4th block has only 1 byte affected, that's because the 3rd cipher block is only evolved in the XOR operation and then output block is generated, thus only one byte in 4th block is affected

CFB

In CFB mode, the 3rd block's corruption will affect both the 3rd and 4th block, that's because one cipher block is involved in the computation of current and next block's decryption. Also, while the whole 4th block is corrupted, the 3rd block has only 1 byte broken, because to generate the 3rd output block, the broken 3rd cipher block is only involved in the last operation, which is XOR, so only one byte is affected

OFB

In OFB mode, only one byte in the 3rd block is affected, since each cipher block is separately involved in the computation of correlated output block, so it's the situation is the same as ECB mode decryption, and even with less influence: only one byte is wrong.

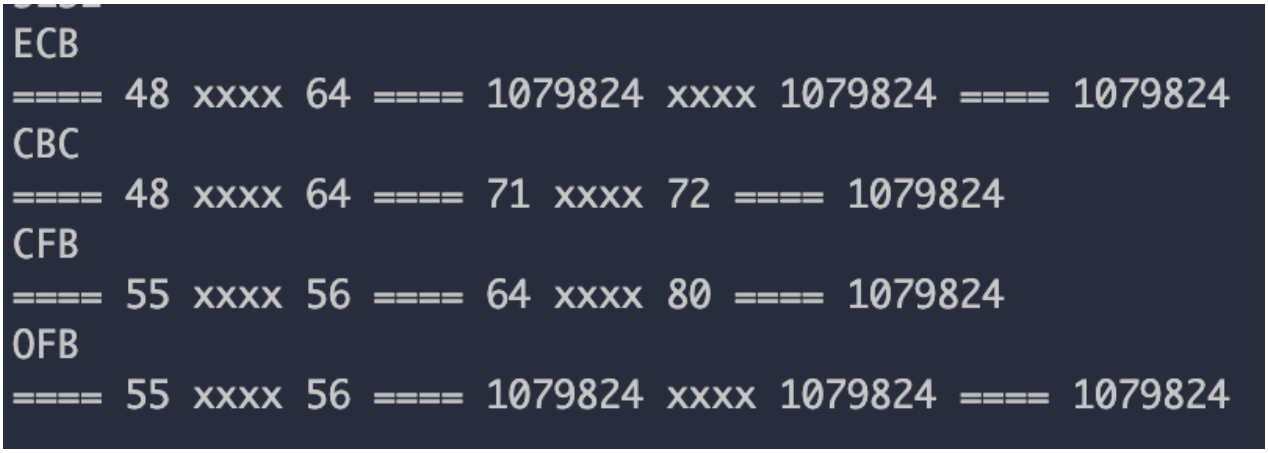
justification

code snippet

I use the following code to compare the "correct" result and the results from each experiment

```
def print_compare():
    i = 0
    while i < len(data) and data[i] == pt[i]:
        i += 1
    a = i
    while i < len(data) and data[i] != pt[i]:
        i += 1
    b = i
    while i < len(data) and data[i] == pt[i]:
        i += 1
    c = i
    while i < len(data) and data[i] != pt[i]:
        i += 1
    d = i
    while i < len(data) and data[i] == pt[i]:
        i += 1
    print("==== {} xxxx {} ==== {} xxxx {} ==== {} ".format(a, b, c, d, i))
```

outcome



This shows that:

1. For ECB mode, the bytes in 3rd block (bytes from 48 to 63) are corrupted
2. For CBC mode, the bytes in 3rd block and one byte in 4th block(71th) are corrupted
3. For CFB mode, one byte in 3rd block and bytes in 4th block(from 64 to 79) are corrupted
4. For OFB mode, one byte at 55th position is corrupted

Task 6

6.1 Encryption with different IV

IV is required in CBC, OFB and CTR mode, in either case it is used for the initial operation for the first block. Here we use CBC mode for example to show IV should be unique in each encryption.

The plaintext we are using:

```
A basic requirement for IV is uniqueness, which means that no IV may be reused u
nder the same key.
~
```

1. Encrypt the same plaintext using the same IV twice:

```
1  openssl enc -aes-128-cbc -e -in plaintext.txt -out cipher_different_iv_2.bin -K
   00112233445566778899aabbccddeeff -iv 0102030405060708
```

cipher1.bin

```
00000000 6C DC 55 27 4C BB 7E 4F 7D BA 90 52 E1 81 EF EE 06 BB 2F A9 BC 2D F0 D5 52 96 7F 87
0000001c 42 36 46 36 FE 40 20 C6 C0 AF E7 A2 A3 A3 F4 1C C5 18 AD CA C5 39 EF F4 F4 8C 01 43
00000038 B8 49 BE 31 AA 0E 62 CD 2C 00 FE 8D 97 59 5C 87 DD 5E 04 F8 36 E7 BD 8E EC EF 72 D8
00000054 D8 56 F0 40 B7 29 1C 82 29 02 9D AF 20 FB 4C AA AA 96 F6 FD 51 D5 71 F1 10 35 71 35
00000070
```

cipher2.bin

```
00000000 6C DC 55 27 4C BB 7E 4F 7D BA 90 52 E1 81 EF EE 06 BB 2F A9 BC 2D F0 D5 52 96 7F 87
0000001c 42 36 46 36 FE 40 20 C6 C0 AF E7 A2 A3 A3 F4 1C C5 18 AD CA C5 39 EF F4 F4 8C 01 43
00000038 B8 49 BE 31 AA 0E 62 CD 2C 00 FE 8D 97 59 5C 87 DD 5E 04 F8 36 E7 BD 8E EC EF 72 D8
00000054 D8 56 F0 40 B7 29 1C 82 29 02 9D AF 20 FB 4C AA AA 96 F6 FD 51 D5 71 F1 10 35 71 35
00000070
```

As we can see `cipher1` and `cipher2` has the exact same value.

2) Encrypt the same plaintext using different IV:

```
1  openssl enc -aes-128-cbc -e -in plaintext.txt -out cipher_different_iv_2.bin -K
   00112233445566778899aabbccddeeff -iv 0102030405060708
```

```
1  openssl enc -aes-128-cbc -e -in plaintext.txt -out cipher_different_iv_2.bin -K
   00112233445566778899aabbccddeeff -iv 1122334455667788
```

cipher_different_iv_1.bin

```
00000000 6C DC 55 27 4C BB 7E 4F 7D BA 90 52 E1 81 EF EE 06 BB 2F A9 BC 2D F0 D5 52 96 7F 87
0000001c 42 36 46 36 FE 40 20 C6 C0 AF E7 A2 A3 A3 F4 1C C5 18 AD CA C5 39 EF F4 F4 8C 01 43
00000038 B8 49 BE 31 AA 0E 62 CD 2C 00 FE 8D 97 59 5C 87 DD 5E 04 F8 36 E7 BD 8E EC EF 72 D8
00000054 D8 56 F0 40 B7 29 1C 82 29 02 9D AF 20 FB 4C AA AA 96 F6 FD 51 D5 71 F1 10 35 71 35
00000070
```

cipher_different_iv_2.bin

```
00000000 BD 76 02 46 46 26 9A C1 84 05 9B 23 E2 CE 40 3B 28 85 3D A6 CA 88 F6 A3 13 0E B9 8B
0000001c AC 16 4F 95 65 6F 49 AB 71 6D 56 C7 6B F4 BD EC 11 0C 39 38 43 87 68 E8 21 59 B3 2A
00000038 76 2D 26 2C E3 BF 64 69 9B A8 DE 41 3F 98 73 D1 D5 85 75 2A A6 30 28 73 16 84 B5 B5
00000054 0B 74 05 D9 BE 2B 69 EA 14 60 60 E7 44 8B 99 32 A6 2C 4D A9 8F EF 85 31 2A FA 7D 67
00000070
```

As we can see, even we are encrypting the same message under same mode, but different IV would provide complete different results in the ciphertext.

Bases on the above observation, using the same IV to encrypt message is not feasible since same plaintext messages would come our exactly the same under same IVs. This is because the same IV on the same encryption mode would provide exactly same output in the first block, which would yield exactly same result on the second block and so on, thus IV should be random and unique in each encrypting.

6.2 Gussing the plaintext

When we are using OFB mode, if we keep using the same IV/key pairs, getting two cipher text and one plaintext can fully recover the paired plaintext. Since in OFB, we are encrypting each message using the XOR of message and IV/Key stream, i.e.

$$\begin{aligned} \text{Plaintext1} \oplus F_k(IV) &= \text{Ciphertext1} \\ \text{Plaintext2} \oplus F_k(IV) &= \text{Ciphertext2} \end{aligned} \tag{1}$$

Thus for each block we have the same encrypting stream, in total we can recover based on:

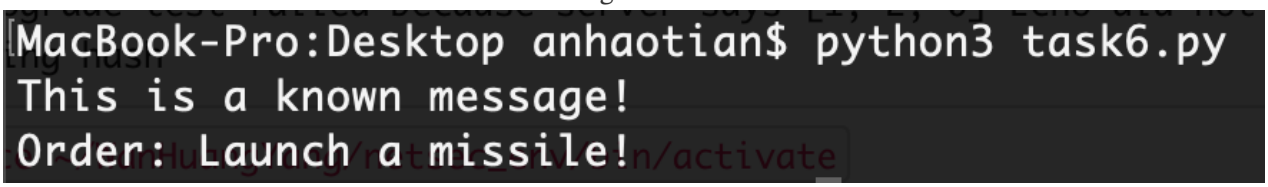
$$\text{Plaintext2} = \text{Ciphertext2} \oplus \text{Ciphertext1} \oplus \text{Plaintext1} \tag{2}$$

As the Python script as below, we can recover plaintext2:

```
plaintext = 'This is a known message!'
cipher1 = 'a469b1c502c1cab966965e50425438e1bb1b5f9037a4c159'
cipher2 = 'bf73bcd3509299d566c35b5d450337e1bb175f903fafc159'

def extend(text):
  while(len(text) < 8):
    text = '0' + text
  return text

plaintext_ascii = [hex(ord(c)) for c in plaintext]
plaintext_binary = []
cipher2_binary = []
cipher1_binary = []
for i in plaintext_ascii:
  temp = "{0:b}".format(int(i, 16))
  plaintext_binary.append(extend(temp))

for i in range(len(cipher1)):
  if i % 2 == 1:
    continue
  temp1 = "{0:b}".format(int('0x' + cipher1[i:i+2], 16))
  temp2 = "{0:b}".format(int('0x' + cipher2[i:i+2], 16))
  cipher1_binary.append(extend(temp1))
  cipher2_binary.append(extend(temp2))

plaintext2 = []
for i in range(len(plaintext_binary)):
  result = int(plaintext_binary[i], 2) ^ int(cipher1_binary[i], 2) ^ int(cipher2_binary[i],
2)
  plaintext2.append(result)

```

```
res = ''.join(chr(int(i)) for i in plaintext2)
print(plaintext)
print(res)
```

We can get the result:



which the plaintext2 is `Order: Launch a missile!`.

CFB mode

If we replace OFB with CFB mode, only the first block of P2 can be revealed, since we can still using the same method as OFB on the first block. However, in the following blocks, we cannot recover P2 since we are using different stream to do XOR on P2 and P1. As long as we cannot get access to Oracle F_k, we cannot decrept the following blocks.

6.3 CPA Under Predictable IV

Under this scenario, we are performing CPA under predictable IVs. Our goal is to construct a P2 to ask Bob encrypt it and get the ciphertext. Before that, we know:

```
Encryption method: 128-bit AES with CBC mode.
Key (in hex):    00112233445566778899aabbccddeeff (known only to Bob)
Ciphertext (C1): bef65565572ccee2a9f9553154ed9498 (known to both)
IV used on P1 (known to both)
     (in ascii): 1234567890123456
(in hex)  : 31323334353637383930313233343536
Next IV (known to both)
     (in ascii): 1234567890123457
     (in hex)  : 31323334353637383930313233343537
```

In CBC mode, each message is encrypted as:

$$\begin{aligned} F_k(\text{Plaintext1} \oplus IV) &= \text{Ciphertext1} \\ F_k(\text{Plaintext2} \oplus \text{Ciphertext1}) &= \text{Ciphertext2} \\ &\cdots \end{aligned} \tag{3}$$

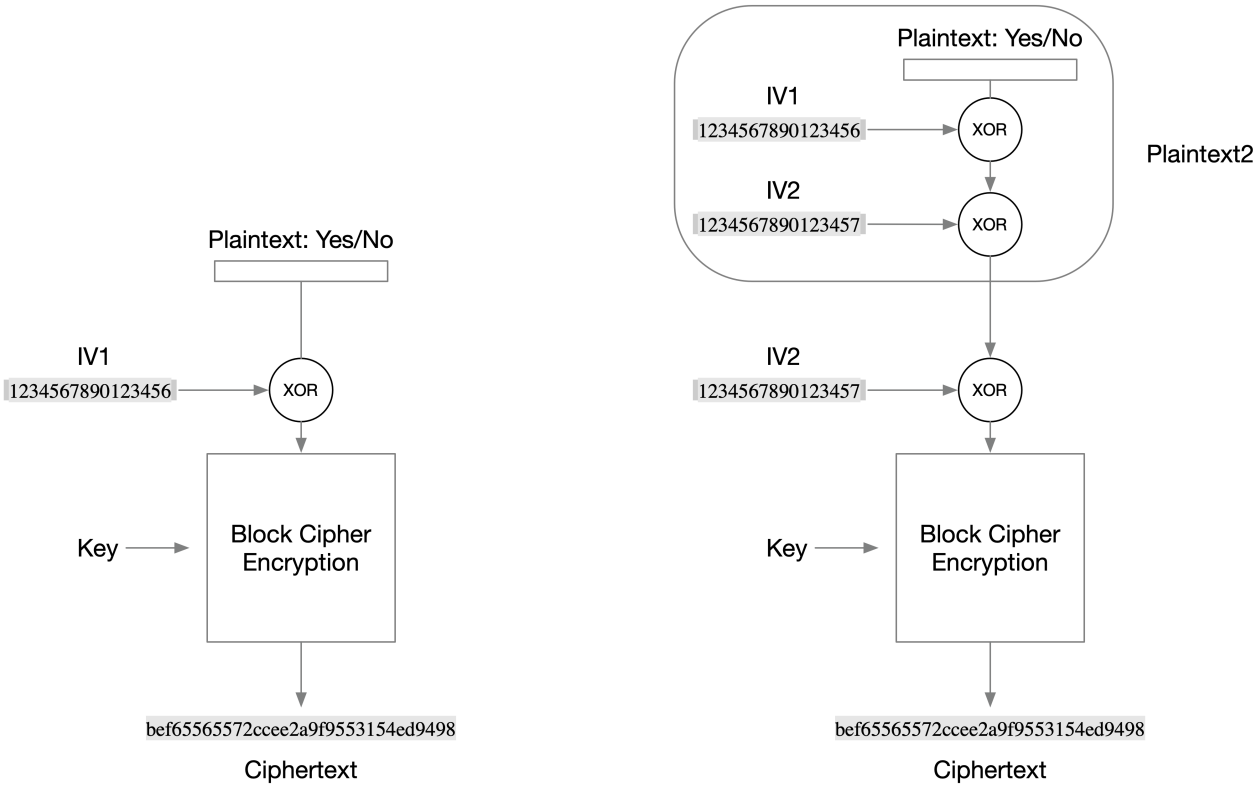
Since we already know the message is either `Yes` or `No`, we have only two choices:

$$\text{Ciphertext1} = F_k(\text{“Yes”} \oplus IV1) \tag{4}$$

or

$$\text{Ciphertext1} = F_k(\text{“No”} \oplus IV1) \tag{5}$$

Suppose the message is `Yes`, then as long as we can let Bob encrypt `Yes` with the same IV1, we should get the same cipher text. Thus we can construct the inpue message `P2` as follows:



Use "Yes" $\oplus IV_1 \oplus IV_2$ as the input string in `P2`, as long as we are using the same `Yes` or `No` as the first time, we should get the same ciphertext output. This is because

$$\text{"Yes"} \oplus IV_1 \oplus IV_2 \oplus IV_2 = \text{" } Yes \text{ "} \oplus IV_1 \tag{6}$$

thus

$$F_k(\text{Plaintext1} \oplus IV_1 \oplus IV_2) \oplus IV_2) = F_k(\text{Plaintext1} \oplus IV_1) = \text{Ciphertext1} \tag{7}$$

Using `Yes` to construct `P2`, if we are getting the same ciphertext from Bob, then we can guarantee the first time Bob is encrypting `Yes`. Otherwise, if we are getting different ciphertext, we can say Bob was encrypting `No` at the first time.Î