fit@hcmus

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

MÃ HÓA ỨNG DỤNG

LAB 01

CRYPTO ENCRYPTION

Giảng viên: Trần Minh Triết

Lớp: 19TN

Người thực hiện:

Họ và tên	MSSV
Đặng Thái Duy	19120491
Hà Chí Hào	19120219

MỤC LỤC

I. Cấu trúc chung

Trong thư mục Source gồm có các file quan trọng sau đây:

- Thư mục Task chứa mã nguồn theo từng thư mục Task tương ứng-
- Thư mục Labsetup được cung cấp sẵn của đề, trong đó thư mục Files ngoài các file cho sẵn còn có output của các task như các file .bin, .bmp, .txt được tạo ra trong quá trình chạy các task
- Makefile: chứa lệnh để chạy các file .sh tương ứng theo từng task. Để chạy task chỉ cần gõ lệnh sau: make <task_name>

II. Task

1. Task 1: Frequency Analysis

Freqency Analysis là một kỹ thuật dựa vào tần suất xuất hiện các ký tự, từ ngữ trong đó và so với tần suất thực tế trong các văn bản ngoài đời, dựa vào đó để phân tích ngược lại được đoạn văn bản đã được mã hóa.

Trong Task 1 này đã cung cấp sẵn file freq.py trong Labsetup/Files. File này sẽ đọc từ ciphertext.txt để lấy ra số liệu thống kê tần suất xuất hiện cho n-grams: gồm có tần số các ký tự đơn, tần số bigram (chuỗi 2 ký tự) và tần số trigram (chuỗi 3 ký tự)

`$ python3 freq/py`: Chạy lệnh này ta có được số liệu như dưới đây

1-gram (top 20):	2-gram (top 20):	3-gram (top 20):
n: 488	yt: 115	ytn: 78
y: 373	tn: 89	vup: 30
v: 348	mu: 74	mur: 20
x: 291	nh: 58	ynh: 18
u: 280	vh: 57	xzy: 16
q: 276	hn: 57	mxu: 14
m: 264	vu: 56	gnq: 14
h: 235	nq: 53	ytv: 13

t: 183	xu: 52	nqy: 13
i: 166	up: 46	vii: 13
p: 156	xh: 45	bxh: 13
a: 116	yn: 44	lvq: 12
c: 104	np: 44	nuy: 12
z: 95	vy: 44	vyn: 12
l: 90	nu: 42	uvy: 11
g: 83	qy: 39	lmu: 11
b: 83	vq: 33	nvh: 11
r: 82	vi: 32	cmu: 11
e: 76	gn: 32	tmq: 10
d: 59	av: 31	vhp: 10

Đồng thời, ta có được tần số xuất hiện các bộ này ngoài đời thật như sau:

+ Đơn ký tự:

E	11.1607%	56.88	M	3.0129%	15.36
A	8.4966%	43.31	H	3.0034%	15.31
R	7.5809%	38.64	G	2.4705%	12.59
I	7.5448%	38.45	B	2.0720%	10.56
O	7.1635%	36.51	F	1.8121%	9.24
T	6.9509%	35.43	Y	1.7779%	9.06
N	6.6544%	33.92	W	1.2899%	6.57
S	5.7351%	29.23	K	1.1016%	5.61
L	5.4893%	27.98	V	1.0074%	5.13
C	4.5388%	23.13	X	0.2902%	1.48
U	3.6308%	18.51	Z	0.2722%	1.39
D	3.3844%	17.25	J	0.1965%	1.00
P	3.1671%	16.14	Q	0.1962%	(1)

+ Chuỗi 2 ký tự:

```
th 1.52       en 0.55       ng 0.18
he 1.28       ed 0.53       of 0.16
in 0.94       to 0.52       al 0.09
er 0.94       it 0.50       de 0.09
an 0.82       ou 0.50       se 0.08
re 0.68       ea 0.47       le 0.08
nd 0.63       hi 0.46       sa 0.06
at 0.59       is 0.46       si 0.05
on 0.57       or 0.43       ar 0.04
nt 0.56       ti 0.34       ve 0.04
ha 0.56       as 0.33       ra 0.04
es 0.56       te 0.27       ld 0.02
st 0.55       et 0.19       ur 0.02
```

+ Chuỗi 3 ký tự:

Rank[1]	Trigram	Frequency[3] (Different source)
1	the	1.81%
2	and	0.73%
3	tha	0.33%
4	ent	0.42%
5	ing	0.72%
6	ion	0.42%
7	tio	0.31%
8	for	0.34%
9	nde	
10	has	
11	nce	
12	edt	
13	tis	
14	oft	0.22%
15	sth	0.21%
16	men	

Từ đó, em bắt đầu dựa trên tần suất các chữ trong văn bản mã hóa ban đầu theo bảng trên đây. Ở đây em dùng lệnh tr để thực hiện việc chuyển đổi ký tự từ file ciphertext.txt sang file output.txt. Để thuận tiện hơn cho việc mã hóa, lúc đầu sẽ dùng các từ in hoa cho văn bản được giải mã, nhờ đó có thể phân biệt được những ký tự nào đã hay chưa được chuyển đổi.

Trong quá trình này, có một số từ không đúng hay sai lệch, lúc này tiếp tục đoán nghĩa của các từ và chuyển đổi các ký tự sao cho nghĩa các từ ngữ phù hợp. Sau một quá trình thử và sai, em ra được kết quả chuyển đổi ngôn ngữ như sau:

$$'abcdefghijklmnopqrstuvwxyz' \rightarrow 'cfmypvbrljxwiejdsgkhnawotu'$$

Sau đó dùng lệnh:

```
$ tr 'abcdefghijklmnopqrstuvwxyz' 'cfmypvbrljxwiejdsgkhnawotu'
<ciphertext.txt> output.txt
```

Từ đó ta rút ra được văn bản sau khi được giải mã:

- ciphertext.txt:

ytn xqavhq yzhu xu qzupvd ltmat qnncq vgxzy hmrty vbynh ytmq ixur qyhvurn

vlvhpq yhme ytn gvrrnh bnniq imsn v uxuvrnuvhmvu yxx

ytn vlvhpq hvan lvq gxxsnupnp gd ytn pncmqn xb tvhfnd lnmuqynmu vy myq xzyqny

vup ytn veevhnuy mceixqmxu xb tmq bmic axcevud vy ytn nup vup my lvq qtvenp gd…

- output.txt:

the oscars turn on sunday which seems about right after this long strange

awards trip the bagger feels like a nonagenarian too

the awards race was bookended by the demise of harvey weinstein at its outset

and the apparent implosion of his film company at the end and it was shaped by…

2. Task 2: Encryption using Different Ciphers and Modes

Trong Task này, em sẽ dùng thử nhiều loại phương thức mã hóa và giải mã khác nhau. Cấu trúc lệnh chung cho các phương thức mã hóa được file hướng dẫn cung cấp như sau

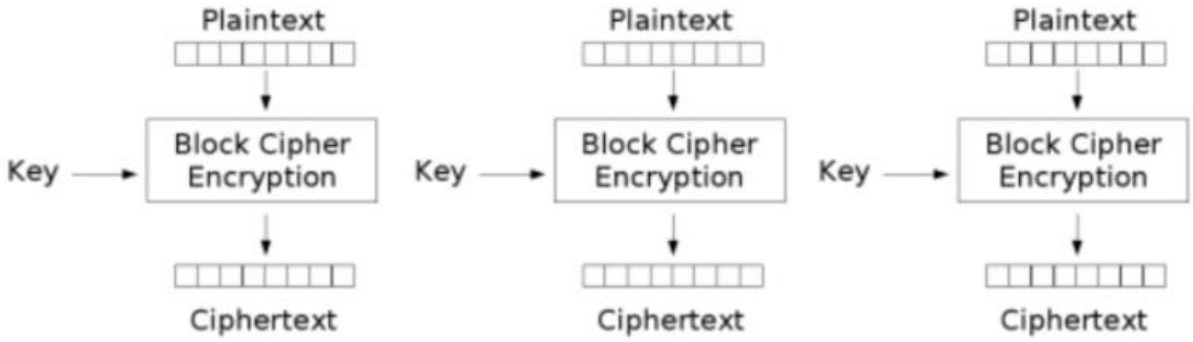
```
$ openssl enc -ciphertype -e  -in plain.txt -out cipher.bin \
              -K  00112233445566778889aabbccddeeff \
              -iv 0102030405060708
```

Trong đó:

- in <file>: input file
- out <file>: output file
- -e: chỉ phương thức đang dùng là để mã hóa (encrypt)
- -d: chỉ phương thức đang dùng là để giải mã (decrypt)
- -K/-iv: Khóa (Key) và vector khởi tạo (Initial vector)

a) ECB: Electronic Codebook

- Là kiểu mã hóa đơn giản, thông điệp cần mã hóa được chia thành từng đoạn trong đó mỗi đoạn được mã hóa độc lập nhau.
- Có một hạn chế lớn khi các khối có cùng nội dung thì sau khi mã hóa sẽ tạo thành các khối có cùng kết quả giống nhau, nên không che giấu được các "mẫu" dữ liệu



Electronic Codebook (ECB) mode encryption

- Lệnh mã hóa:

```
$ openssl enc  -aes-128-ecb  -e -in ciphertext.txt -out
ecb_cipher.bin -K 00112233445566778889aabbccddeef
```

- Lệnh giải mã:

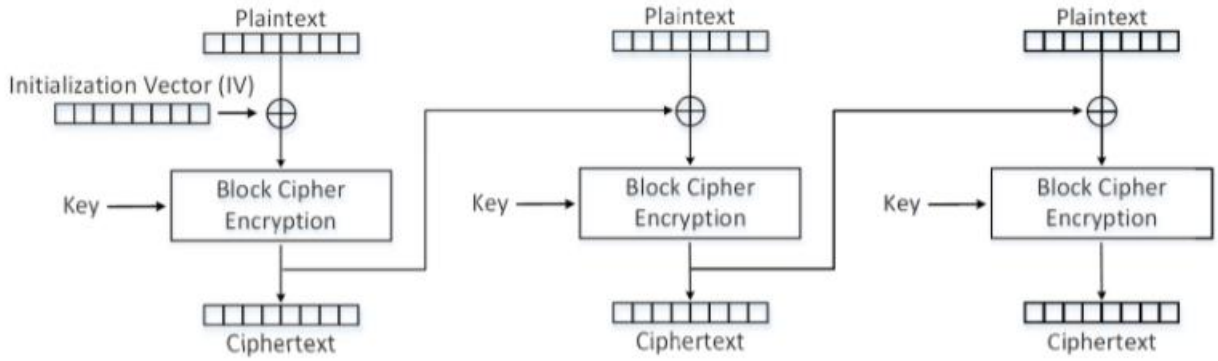
```
$ openssl enc  -aes-128-ecb  -d -in ecb_cipher.bin -out
ecb_plain.txt -K 00112233445566778889aabbccddeef
```

- So sánh 2 file ciphertext.txt và ecb_plain.txt: kết quả giống nhau

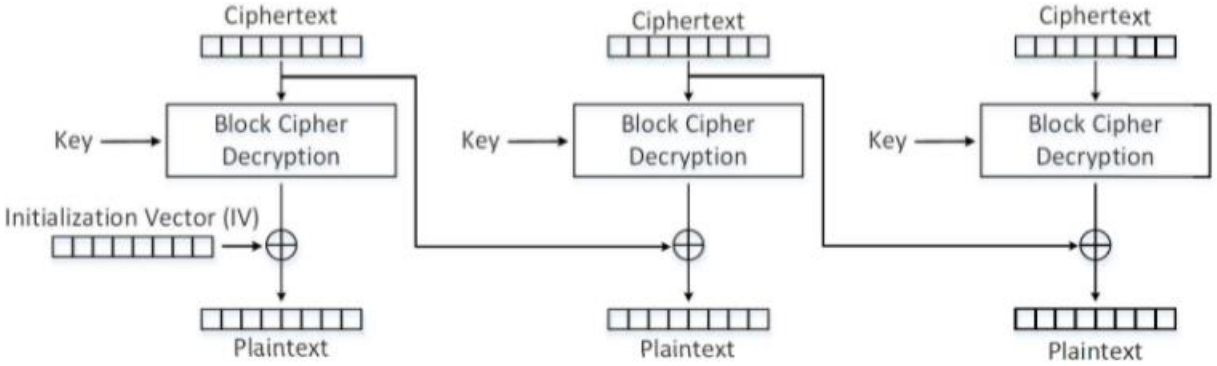
```
$ diff ecb_plain.txt ciphertext.txt
```

b) CBC: Cipher-block chaining

- Mỗi khối plaintext được XOR với khối ciphertext trước khi được mã hóa. Do đó, mỗi khối ciphertext phụ thuộc vào tất cả các khối plaintext xuất hiện từ đầu đến thời điểm đó
- Để đảm bảo tính duy nhất của mỗi thông điệp được mã hóa, ta sử dụng thêm initialization vector.



(a) Cipher Block Chaining (CBC) mode encryption



- Lệnh mã hóa:

```
$ openssl enc  -aes-128-cbc  -e -in ciphertext.txt -out
cbc_cipher.bin -K 00112233445566778889aabbccddeef
```

- Lệnh giải mã:

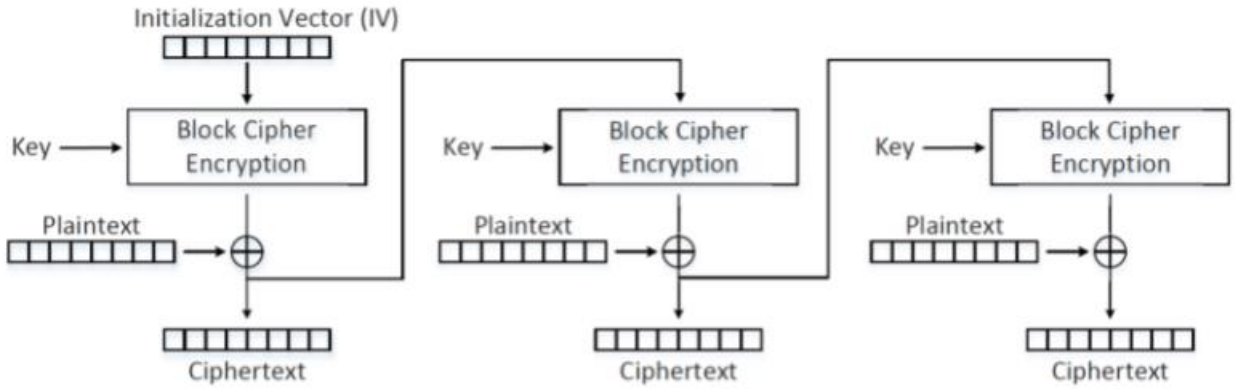
```
$ openssl enc  -aes-128-cbc  -d -in cbc_cipher.bin -out
cbc_plain.txt -K 00112233445566778889aabbccddeef
```

- So sánh 2 file ciphertext.txt và cbc_plain.txt: kết quả giống nhau

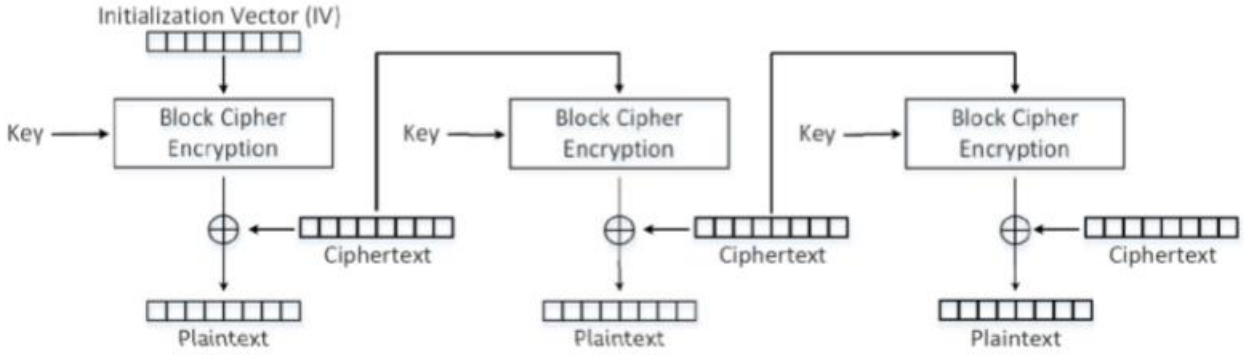
```
$ diff cbc_plain.txt ciphertext.txt
```

c) CFB: Cipher feedback (CFB)

- Plaintext không được mã hóa bằng chính thuật toán đang xét tới mà được mã hóa bằng cách XOR với một chuỗi được tạo ra bằng thuật toán mã hóa
- Ciphertext từ block trước đó được đưa vào block cipher cho việc mã hóa, kết quả trả ra được XOR với plaintext để tạo ra ciphertext thực tế



(a) Cipher Feedback (CFB) mode encryption



- Lệnh mã hóa:

```
$ openssl enc  -aes-128-cfb  -e -in ciphertext.txt -out
cfb_cipher.bin -K 00112233445566778889aabbccddeef
```

- Lệnh giải mã:

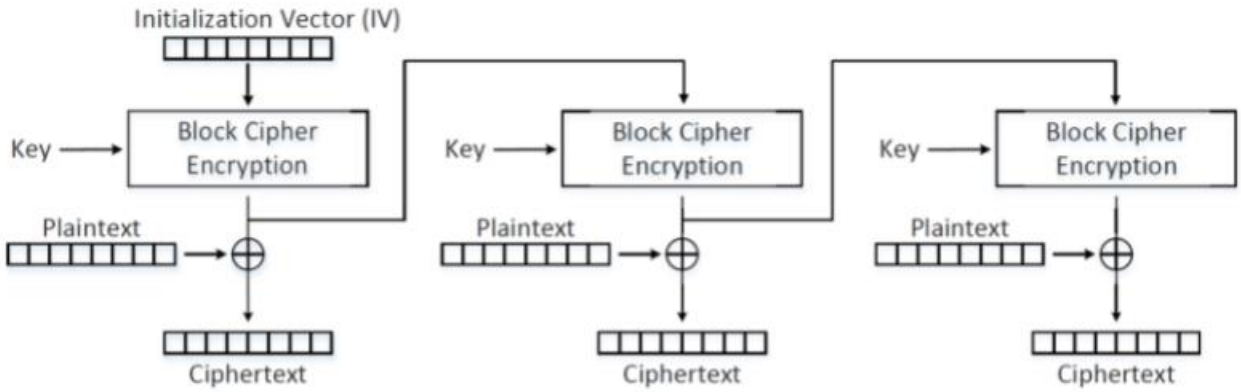
```
$ openssl enc  -aes-128-cfb  -d -in cfb_cipher.bin -out
cfb_plain.txt -K 00112233445566778889aabbccddeef
```

- So sánh 2 file ciphertext.txt và cfb_plain.txt: kết quả giống nhau

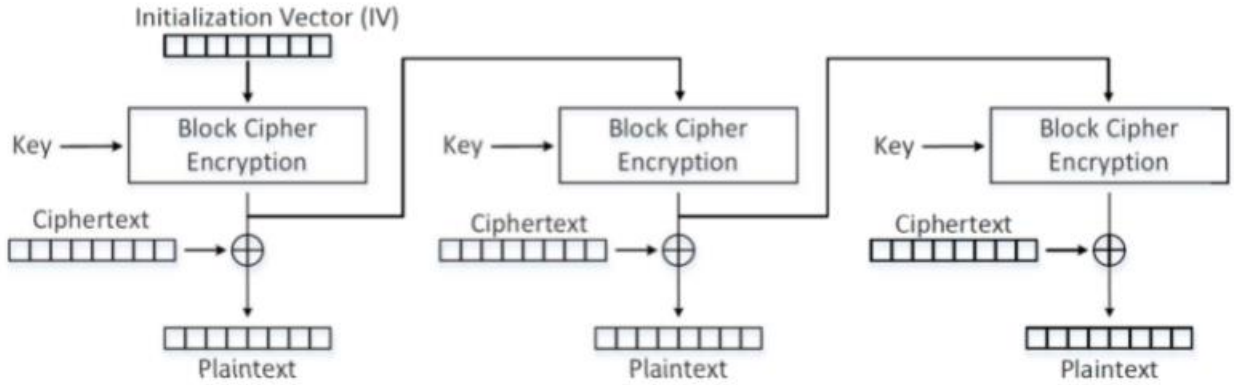
```
$ diff cfb_plain.txt ciphertext.txt
```

d) OFB: Output feedback

- Cũng tương tự như CFB, nhưng trừ việc dữ liệu **trước** (trong khi ở CFB thì là **sau**) khi XOR được đưa đến block tiếp theo



(a) Output Feedback (OFB) mode encryption



- Lệnh mã hóa:

```
$ openssl enc  -aes-128-ofb  -e -in ciphertext.txt -out
ofb_cipher.bin -K 00112233445566778889aabbccddeef
```

- Lệnh giải mã:

```
$ openssl enc  -aes-128-ofb  -d -in ofb_cipher.bin -out
ofb_plain.txt -K 00112233445566778889aabbccddeef
```

- So sánh 2 file ciphertext.txt và ofb_plain.txt: kết quả giống nhau

```
$ diff ofb_plain.txt ciphertext.txt
```

3. Task 3: Encryption Mode – ECB vs. CBC

Trong Task này ta thử 2 phương pháp mã hóa là ECB và CBC để thực hiện việc mã hóa 1 bức ảnh bmp. Để có thể hiển thị được ảnh sau khi mã hóa được thì đối với file .bmp, 54 byte đầu tiên chứa thông tin tiêu đề đặc trưng của hình ảnh vì vậy cần giữ nguyên thông tin tiêu đề của file mã hóa để có thể xem là một file .bmp hợp lệ. Byte thứ 55 trở đi là phần nội dung của bức

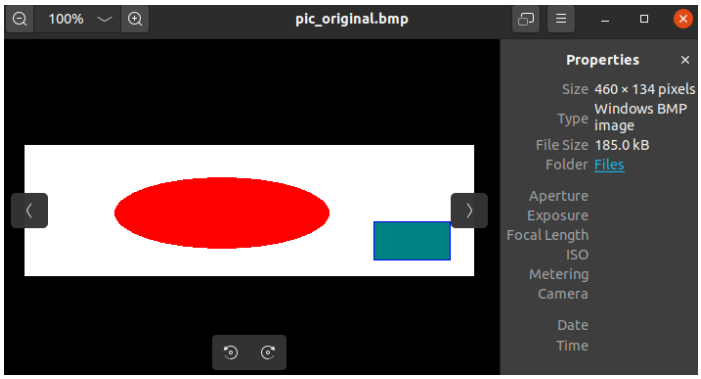
ảnh, ta mã hóa chúng với loại mã hóa tương ứng sau đó kết hợp tiêu đề gốc và phần dữ liệu mã hóa với nhau sẽ được một file mã hóa .bmp hoàn chỉnh có thể xem được.

Đoạn code chung cho phần xử lý này như sau:

```
$ head -c 54 p1.bmp > header

$ tail -c +55 p2.bmp > body

$ cat header body > new.bmp

$ eog -n new.bmp # hiển thị ảnh
```

Bây giờ ta sẽ sử dụng hình ảnh được cho trong Labsetup để thực hiện việc mã hóa là file pic_orginal.bmp

```
$ file = pic_origninal.bmp

$ eog -n ${file} &
```

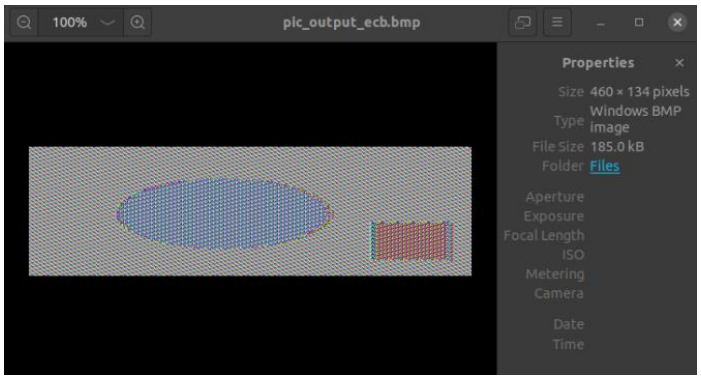


a) Mã hóa với ECB

```
$ openssl enc  -aes-128-ecb  -e -in ${file} -out pic_cipher_ecb.bmp -K
00112233445566778889aabbccddeeff
```

Sau đó thực hiện việc đổi sang 1 ảnh bmp xem được như trên, ảnh đầu ra tương ứng là pic_output_ecb.bmp

```
$ head -c 54 ${file} > header

$ tail -c +55 pic_cipher_ecb.bmp > body

$ cat header body > pic_output_ecb.bmp

$ eog -n pic_output_ecb.bmp &
```



Nhận xét: Bức ảnh được mã hóa bằng ECB nhìn gần giống với bức ảnh gốc. Lý do cho việc đó vì ECB sẽ mã hóa độc lập từng khối với nhau, các block có kích thước là 128 bit. Do đó nếu như có 2 block ban đầu giống nhau thì sau khi mã hóa bằng ECB 2 block đó vẫn giống nhau nên có thể xem đơn giản là tất cả các block đơn giản là được ánh xạ ra những block mới với độ lệch như nhau. Vậy nên ta vẫn có thể xem được bức ảnh mới tương tự hình dáng với bức ảnh gốc.

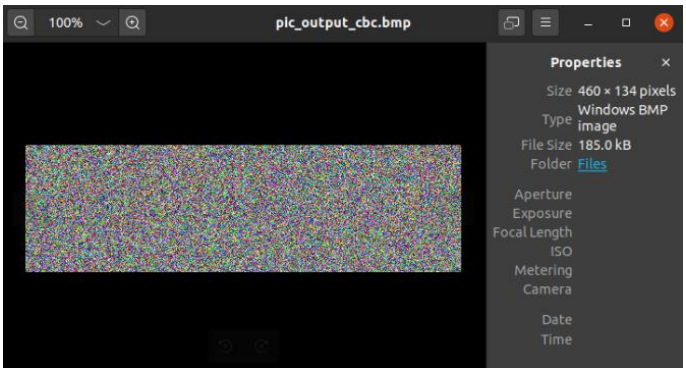
→ Phương thức mã hóa ECB kém an toàn để bảo vệ tính toàn toàn vẹn thông tin (vì ảnh được mã hóa vẫn còn tương tự ảnh gốc)

b) Mã hóa với CBC

```
$ openssl enc  -aes-128-cbc  -e -in ${file} -out pic_cipher_cbc.bmp -K
00112233445566778889aabbccddeeff -iv 0102030405060708
```

Sau đó thực hiện việc đổi sang 1 ảnh bmp xem được như trên, ảnh đầu ra tương ứng là pic_output_ecb.bmp

```
$ head -c 54 ${file} > header

$ tail -c +55 pic_cipher_cbc.bmp > body

$ cat header body > pic_output_cbc.bmp

$ eog -n pic_output_cbc.bmp &
```



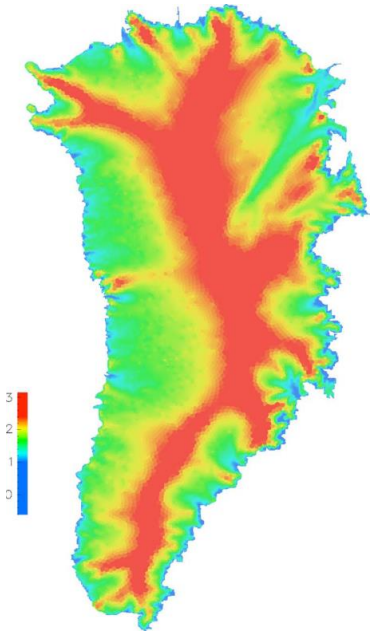
Nhận xét: Bức ảnh được hiển thị là một tổ hợp màu hỗn loạn và không còn dấu vết gì của bức ảnh gốc. Lý do là vì CBC là phương thức mã hóa theo kiểu từng block của plaintext XOR với cipher block trước đó để cho ra cipher block tiếp theo. Có thể nói các khối ciphertext sau phụ thuộc vào tất cả các khối plaintext xuất hiện từ đầu đến thời điểm đó nên thành ra các block được hiển thị. Do đó thành ra bức ảnh hiển thị các block không còn bất kỳ điểm gì tương ứng với bức ảnh cũ

→ Phương thức mã hóa CBC an toàn hơn ECB ở trên rất nhiều (vì ảnh được mã hóa không còn dấu vết gì của ảnh gốc)

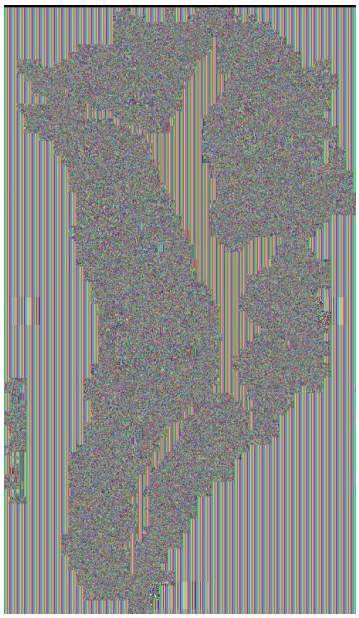
c) Thử trên 1 bức ảnh bmp khác: greenland.bmp

Tương tự như với bức ành pic_original.bmp, ta thay thế file=greenland.bmp

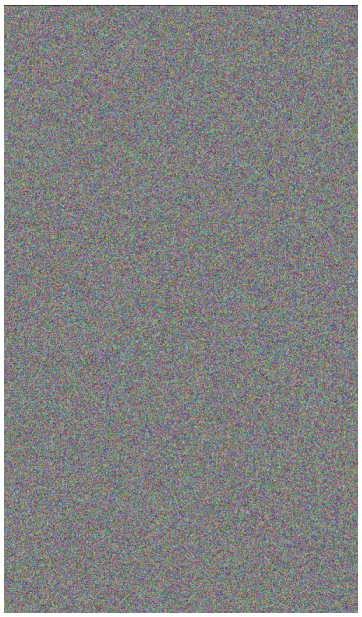
Các bức ảnh in ra lần lượt như sau:



greenland.bmp



pic_output_ecb.bmp



pic_output_cbc.bmp

Nhận xét: có thể thấy với bức ảnh greenland.bmp thì sau khi mã hóa theo dạng ECB vẫn cho ta được một bức ảnh vẫn còn hiển thị những đường nét của bức ảnh gốc trong khi mã hóa theo dạng CBC thì kết quả không hiển thị ra được gì.

4. Task 4: Padding

a) Kiểm tra padding của các phương pháp mã hóa ECB, CBC, CFB, OFB

Tạo 1 file test.txt với nội dung là “hello world”. Sau đó kiểm tra độ dài của file test và các file sau khi được mã hóa với các phương pháp trên. Ta sử dụng lệnh ls -ld <file> để tính ra độ dài

```
$ echo -n "hello world" > test.txt

$ ls -ld test.txt # -> 11 byte
```

- ECB

```
$ openssl enc  -aes-128-ecb  -e -in test.txt -out ecb_cipher.bin -K
00112233445566778889aabbccddeeff -iv 0102030405060708

$ ls -ld ecb_cipher.bin # -> 16 byte
```

- CBC

```
$ openssl enc  -aes-128-cbc  -e -in test.txt -out cbc_cipher.bin -K
00112233445566778889aabbccddeeff -iv 0102030405060708

$ ls -ld cbc_cipher.bin # -> 16 byte
```

- CFB

```
$ openssl enc  -aes-128-cfb  -e -in test.txt -out cfb_cipher.bin -K
00112233445566778889aabbccddeeff -iv 0102030405060708

$ ls -ld cfb_cipher.bin # -> 11 byte
```

- OFB

```
$ openssl enc  -aes-128-ofb  -e -in test.txt -out ofb_cipher.bin -K
00112233445566778889aabbccddeeff -iv 0102030405060708

$ ls -ld ofb_cipher.bin # -> 11 byte
```

```
-rw-rw-rw- 1 root root 11 Apr 10  2022 test.txt
hex string is too short, padding with zero bytes to length
-rwxrwxrwx 1 root root 16 Apr 10  2022 ecb_cipher.bin
hex string is too short, padding with zero bytes to length
-rwxrwxrwx 1 root root 16 Apr 10  2022 cbc_cipher.bin
hex string is too short, padding with zero bytes to length
-rwxrwxrwx 1 root root 11 Apr 10  2022 cfb_cipher.bin
hex string is too short, padding with zero bytes to length
-rwxrwxrwx 1 root root 11 Apr 10  2022 ofb_cipher.bin
hex string is too short, padding with zero bytes to length
```

Độ dài lần lượt các file

Nhận xét độ dài các byte:

- Với ECB và CBC: độ dài file là 16 byte (thêm bù vào 5 byte so với 11 byte gốc của file test.txt). Lý do được thêm padding là do 2 phương pháp này plaintext sau khi đưa vào

Block Cipher Encrytion thì sẽ được bù vào cho đủ số lượng byte cần thiết để thực hiện (phải là bội số của 16)

- Với CFB và OFB: độ dài file vẫn giữ nguyên giá trị ban đầu so với file test.txt là 11 byte. Lý do là vì plaintext không được đưa vào Block Cipher Block mà được dùng để XOR với kết quả trả ra của nó. Vì vậy từng block của ciphertext sẽ có cùng kích thước với từng block trong plaintext. Do đó kết quả mã hóa vẫn giữ nguyên kích thước cũ.

b) Kiểm tra giá trị của padding trong quá trình mã hóa

Trước hết tạo ra 3 file với kích thước lần lượt và 5, 10 và 16 byte. Sau đó dùng mã hóa theo kiểu cbc để cho ra output có kèm padding như sau:

```
$ echo -n "ABCDE" > f1.txt # 5 byte

$ echo -n "ABCDEFGHIJ" > f2.txt # 10 byte

$ echo -n "ABCDEFGHIJKLMNOP" > f3.txt # 16 byte
```

Mã hóa lần lượt các file cho ra độ dài tương ứng như sau:

```
$ openssl enc  -aes-128-cbc  -e -in f1.txt -out p1.bin -K
00112233445566778889aabbccddeeff -iv 0102030405060708

$ ls -ld p1.bin # 16

$ openssl enc  -aes-128-cbc  -e -in f2.txt -out p2.bin -K
00112233445566778889aabbccddeeff -iv 0102030405060708

$ ls -ld p2.bin # 16

$ openssl enc  -aes-128-cbc  -e -in f3.txt -out p3.bin -K
00112233445566778889aabbccddeeff -iv 0102030405060708

$ ls -ld p3.bin # 32
```

```
hex string is too short, padding with zero bytes to length
-rwxrwxrwx 1 root root 16 Apr 10  2022 p1.bin
hex string is too short, padding with zero bytes to length
-rwxrwxrwx 1 root root 16 Apr 10  2022 p2.bin
hex string is too short, padding with zero bytes to length
-rwxrwxrwx 1 root root 32 Apr 10  2022 p3.bin
```

2 file p1.bin, p2.bin có độ dài là 16 byte do padding bù vào 2 file đó để được bội số của 16. Còn file p3.bin ban đầu có 16 byte đã là bội số của 16 rồi cho nên trong khi mã hóa sẽ thêm vào 1 block mới sau đó có kích thước là 16 byte nên tổng ra 32 byte.

Bây giờ ta cần xem những gì được thêm vào padding trong quá trình mã hóa. Để làm được điều này em sẽ giải mã các file này bằng lệnh "`openssl enc -aes-128-cbc -d`". Tuy nhiên việc giải mã mặc định sẽ tự động loại bỏ đi padding để cho ra kết quả gốc nhưng có một tùy chọn là "-nopad" dùng để vô hiệu hóa việc này. Vậy nên ta có thể xem được padding. Đồng thời, em dùng thêm lệnh hexdump để có thể hiển thị được padding dưới dạng hex.

```
$ openssl enc  -aes-128-cbc  -d -in p1.bin -out p1_decrypt.txt -K
00112233445566778889aabbccddeeff -iv 0102030405060708 -nopad

$ hexdump -C p1_decrypt.txt

$ openssl enc  -aes-128-cbc  -d -in p2.bin -out p2_decrypt.txt -K
00112233445566778889aabbccddeeff -iv 0102030405060708 -nopad

$ hexdump -C p2_decrypt.txt

$ openssl enc  -aes-128-cbc  -d -in p3.bin -out p3_decrypt.txt -K
00112233445566778889aabbccddeeff -iv 0102030405060708 -nopad

$ hexdump -C p3_decrypt.txt
```

Kết quả ra được như sau:

```
hex string is too short, padding with zero bytes to length
00000000  41 42 43 44 45 0b 0b 0b  0b 0b 0b 0b 0b 0b 0b 0b  |ABCDE...........|
00000010
hex string is too short, padding with zero bytes to length
00000000  41 42 43 44 45 46 47 48  49 4a 06 06 06 06 06 06  |ABCDEFGHIJ......|
00000010
hex string is too short, padding with zero bytes to length
00000000  41 42 43 44 45 46 47 48  49 4a 4b 4c 4d 4e 4f 50  |ABCDEFGHIJKLMNOP|
00000010  10 10 10 10 10 10 10 10  10 10 10 10 10 10 10 10  |................|
00000020
```

Nhận xét: kết quả trả về in ra kết quả theo từng cụm 16 byte, giá trị gốc được để trong cặp | |. Ta để ý thấy:

- p1_decrypt: padding thêm vào là 11 ký tự 0b
- p2_decrypt: padding thêm vào là 6 ký tự 06
- p3_decrypt: padding thêm vào là 16 ký tự 10

5. Task 5: Error Propagation – Corrupted Cipher Text

File plain.txt được đề cập ở task này trong phần cài đặt tên là task5_plain.txt, file task5_plain.txt và các file input/output khác được đặt trong Labsetup/Files thay vì trong thư mục task này.

Đầu tiên với công việc là tạo một file có độ dài ít nhất 1000 byte, em chỉ việc viết “abcde” và copy paste lại cho tới khi đủ 1100 byte (hơn 100 byte để an toàn), và đặt tên file là plain.txt.

Để encrypt với AES-128 thì em sẽ dùng phần mềm OpenSSL để encrypt với encryption mode là ECB (Electronic code book):

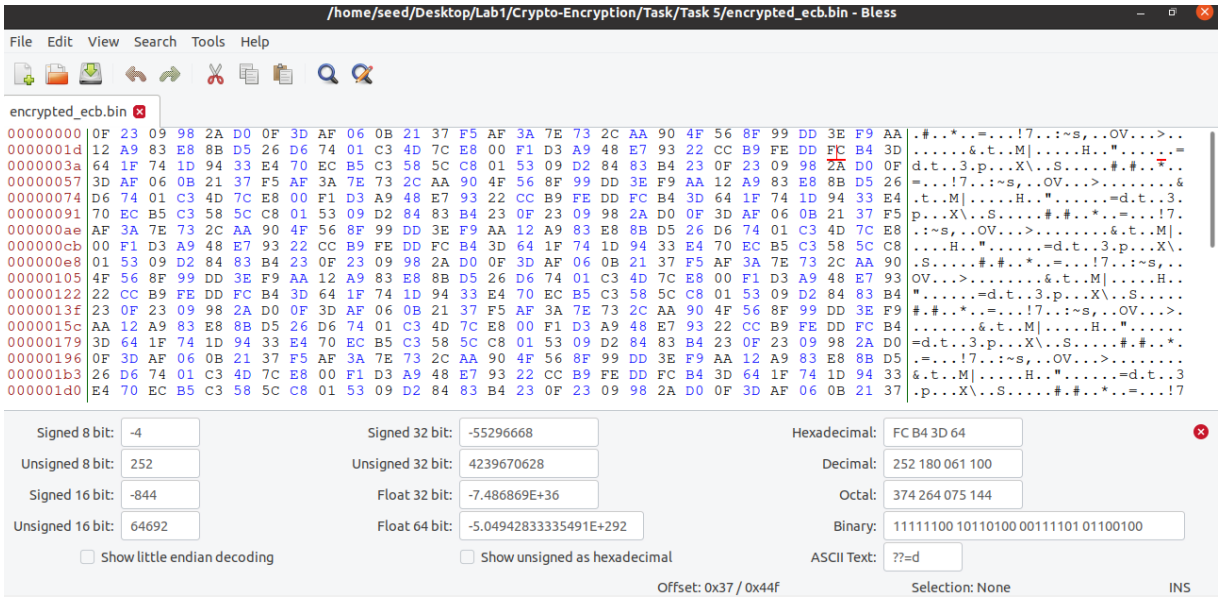
```
$ openssl enc -e -aes-128-ecb -in plain.txt -out encrypted_ecb.bin
-K 01234567890123456789012345678901
```

Dùng lệnh openssl enc cho việc encryption (-e), sử dụng thuật toán AES-128, encryption mode là ECB (-aes-128-ecb), có input file là plain.txt (-in plain.txt), và output ra file encrypted_ecb.bin (-out encrypted_ecb.bin), với key là 01234567890123456789012345678901 (-K 01234567890123456789012345678901, dãy hex phải có độ dài là 32 (một chữ số hex có 16 giá trị => sử dụng 4 bit để thể hiện, 128/4=32)).

Tiếp theo đề bài yêu cầu corrupt 1 bit trong byte thứ 55, bằng cách sử dụng phần mềm chỉnh sửa hex Bless. Sử dụng bless đối với file output ở câu lệnh trước như sau:

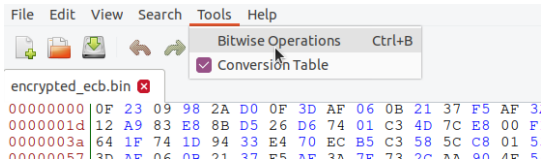
```
$ bless encrypted_ecb.bin
```

Cửa sổ giao diện phần mềm bless sẽ hiện ra với file encrypted_ecb.bin đã mở sẵn:

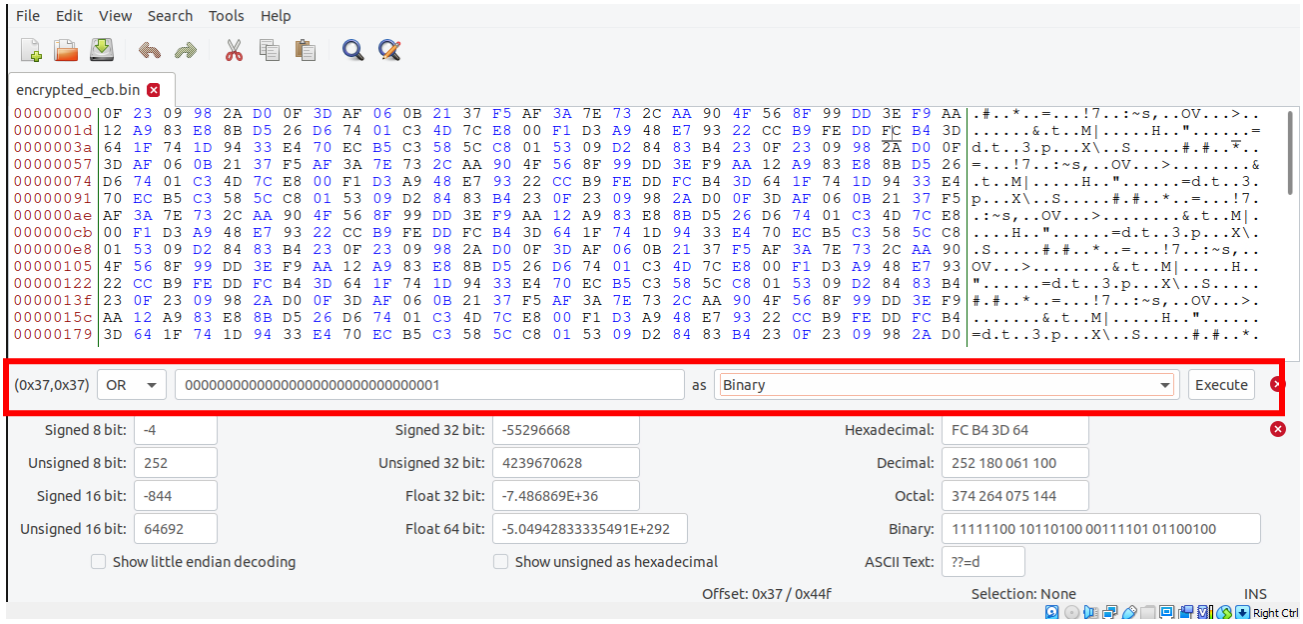


Byte thứ 55 chính là byte có offset là 0x37, đang có giá trị hex là FC và binary là 11111100 10110100 00111101 01100100. Để thực hiện việc corrupt 1 bit, em thấy là bit cuối đang là 0 nên em chỉ việc thực hiện phép OR với dãy bit 00000000000000000000000000000001 để có thể đổi bit cuối thành 1.

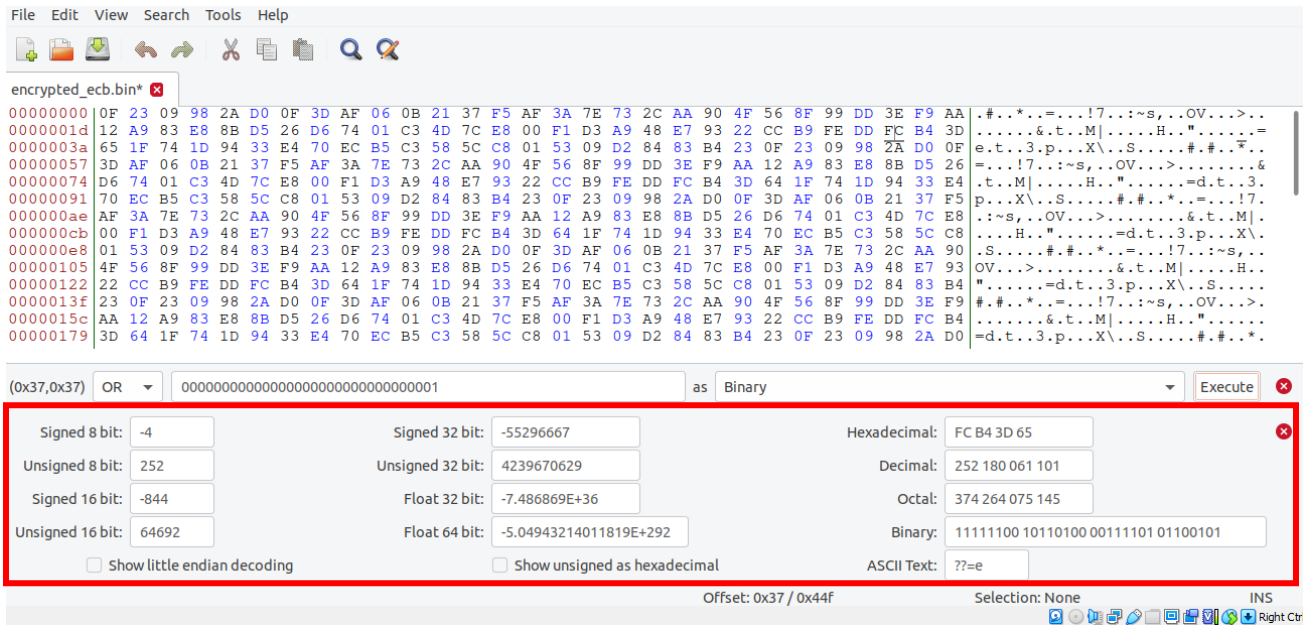
Để thực hiện điều trên trước tiên em chọn mở Tools -> Bitwise Operations



Thanh bên dưới sẽ hiện ra, em sẽ chọn toán tử OR, điền dãy bit ở trên vào, và chọn as Binary



Sau khi bấm Execute byte đó trở thành:



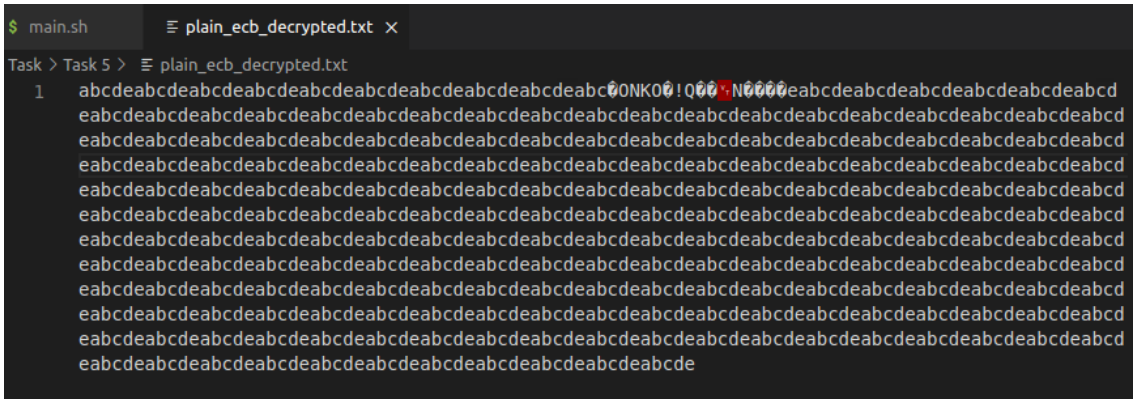
Sau đó em sẽ ctrl+S để lưu file và tắt Bless.

Việc corrupt file đã thực hiện xong, bây giờ em sẽ decrypt lại file ứng với cách encrypt trước đó:

$ openssl enc -d -aes-128-ecb -in encrypted_ecb.bin -out plain_ecb_decrypted.txt -K 01234567890123456789012345678901

Dùng lệnh openssl enc cho việc decryption (-d), sử dụng thuật toán AES-128, encryption mode là ECB (-aes-128-ecb), có input file là encrypted_ecb.bin (-in encrypted_ecb.bin), và output ra file plain_ecb_decrypted.txt (-out plain_ecb_decrypted.txt), với key là 01234567890123456789012345678901 (-K 01234567890123456789012345678901).

Sau khi decrypt thì file output như sau:



Với các encryption mode còn lại mà đề bài đề cập (CBC, CFB, OFB), thì các bước thực hiện cũng sẽ tương tự, chỉ cần để ý byte thứ 55 không phải lúc nào cũng có bit cuối là 0 nên bước corrupt sẽ khác, cụ thể là em sẽ chọn để bit 1 ở vị trí khác thay vì vị trí cuối. Và 3 encryption mode này sẽ cần thêm IV (Initial vector) để có thể encrypt được file, cụ thể với CBC:

$ openssl enc -e -aes-128-cbc -in plain.txt -out encrypted_cbc.bin -K 01234567890123456789012345678901 -iv abcdeabcdeabcdeabcdeabcdeabcdeab

Dùng lệnh openssl enc cho việc encryption (-e), sử dụng thuật toán AES-128, encryption mode là CBC (-aes-128-cbc), có input file là plain.txt (-in plain.txt), và output ra file encrypted_cbc.bin (-out encrypted_cbc.bin), với key là 01234567890123456789012345678901 (-K 01234567890123456789012345678901) và IV là abcdeabcdeabcdeabcdeabcdeabcdeab (-iv abcdeabcdeabcdeabcdeabcdeabcdeab, dãy hex phải có độ dài là 32 (một chữ số hex có 16 giá trị => sử dụng 4 bit để thể hiện, 128/4=32))

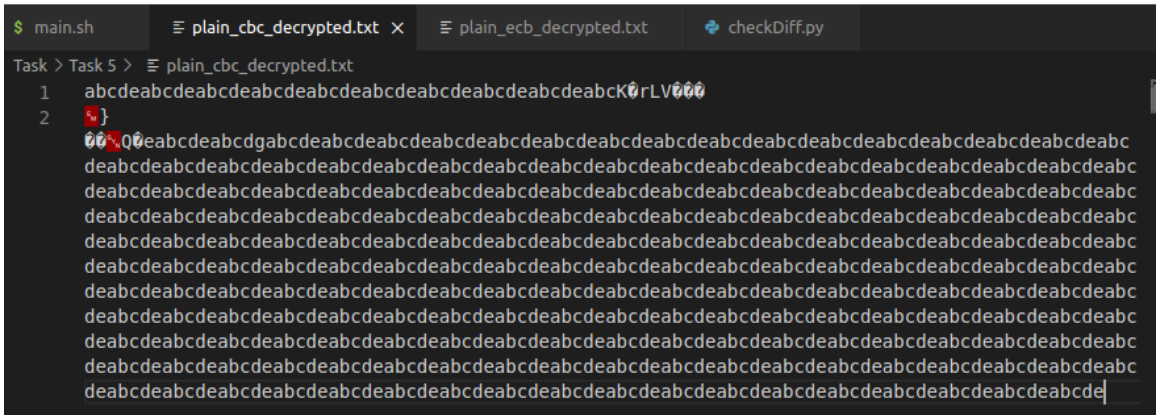
Việc decrypt file cũng sẽ tương ứng:

$ openssl enc -d -aes-128-cbc -in encrypted_cbc.bin -out plain_cbc_decrypted.txt -K 01234567890123456789012345678901 -iv abcdeabcdeabcdeabcdeabcdeabcdeab

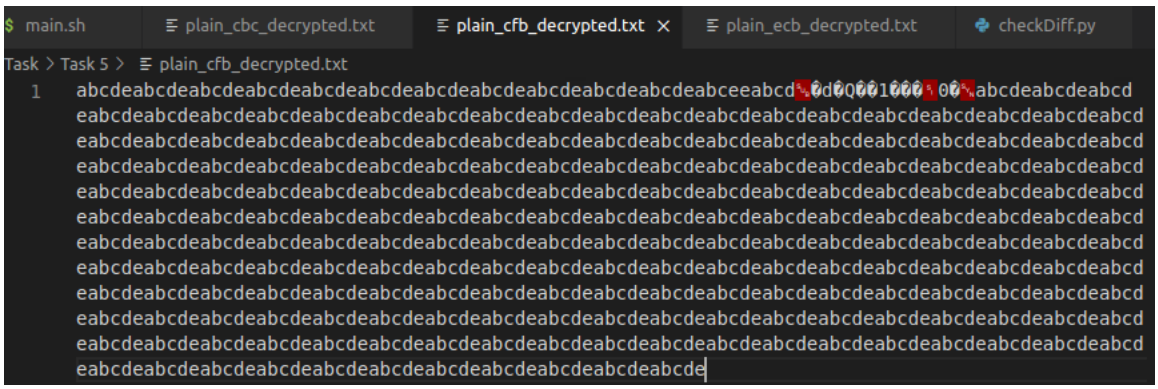
Dùng lệnh openssl enc cho việc decryption (-d), sử dụng thuật toán AES-128, encryption mode là CBC (-aes-128-cbc), có input file là encrypted_cbc.bin (-in encrypted_cbc.bin), và output ra file plain_cbc_decrypted.txt (-out plain_cbc_decrypted.txt), với key là 01234567890123456789012345678901 (-K 01234567890123456789012345678901) và IV là abcdeabcdeabcdeabcdeabcdeabcdeab (-iv abcdeabcdeabcdeabcdeabcdeabcdeab)

Sau khi thực hiện hết thao tác encrypt -> corrupt -> decrypt với cả 3 encryption mode còn lại, thì đây là output của chúng:

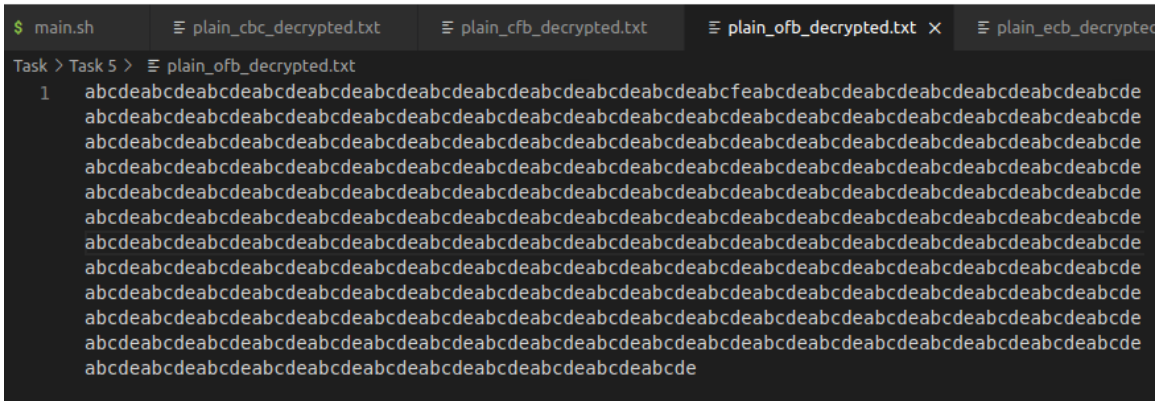
- CBC:

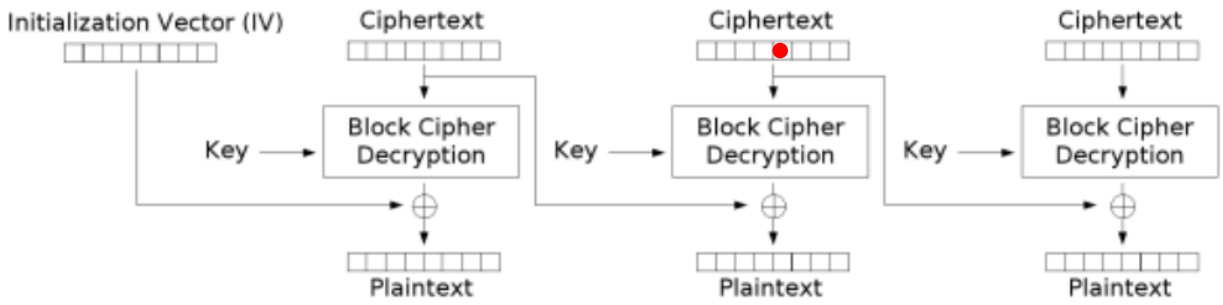


- CFB:



- OFB:

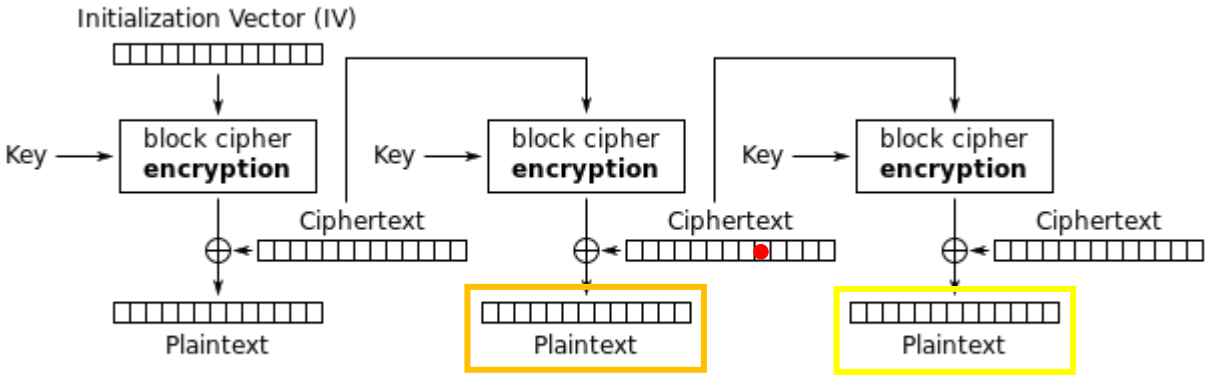

$ main.sh
plain_cbc_decrypted.txt
plain_cfb_decrypted.txt
plain_ofb_decrypted.txt
plain_ecb_decrypted
Task > Task 5 > plain_ofb_decrypted.txt
1
abcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcfeabcdeabcdeabcdeabcdeabcdeabcdeabcde
abcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcd



Cipher Block Chaining (CBC) mode decryption

Ví dụ byte thứ 55 mà ta thay đổi nằm ở block ở chấm màu đỏ trong hình, thì ciphertext của block đó đã bị sai. Khi ciphertext đó được đem đi decrypt thì kết quả decrypt đó đã bị sai hoàn toàn, cho nên nguyên block plaintext đều bị sai.
Ciphertext chứa chấm màu đó ngoài ra còn được đem đi XOR với kết quả decrypt của block kế tiếp, nhưng lúc này do ciphertext đó chỉ sai 1 bit, cho nên kết quả phép XOR (plaintext của block kế tiếp) cũng chỉ sai 1 bit, mà byte bị sai dù chỉ 1 bit thì cũng được xem là khác so với ban đầu.
Từ đó suy ra số byte khác so với ban đầu là 16 + 1 = 17 byte.

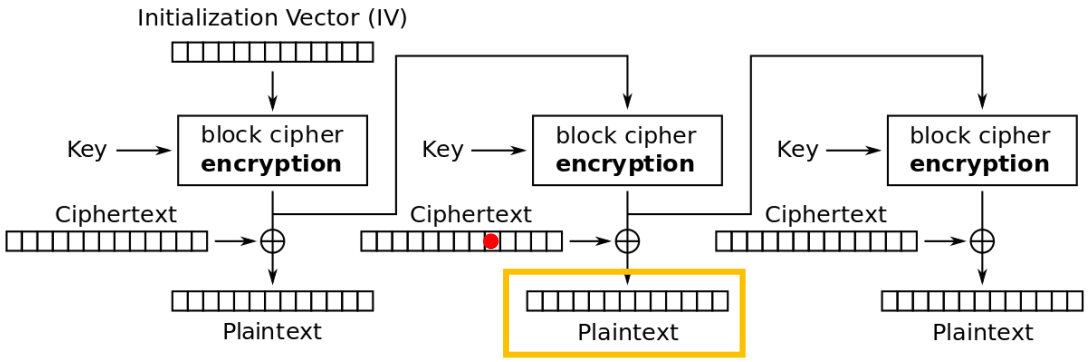
- CFB (Cipher feedback)



Cipher Feedback (CFB) mode decryption

Ví dụ byte thứ 55 nằm ở chấm đỏ, thì trước tiên nó sẽ làm plaintext màu cam bị lỗi 1 bit vì phép tính XOR. Và sẽ làm plaintext màu vàng bị sai hết vì ciphertext bị lỗi đó được đem đi encrypt thì kết quả sẽ ra 1 block mà mọi bit đều sai.
Suy ra số byte khác lúc đầu = 16 + 1 = 17 byte

- OFB (Output feedback)



Output Feedback (OFB) mode decryption

Ví dụ byte thứ 55 nằm ở dấu chấm đỏ, thì ciphertext chỉ được dùng để XOR với kết quả encryption ở trên nên plaintext màu cam chỉ sai 1 bit. Suy ra số byte sai cũng chỉ là 1 byte.

6. Task 6: Initial Vector (IV) and Common Mistakes

a) Task 6.1. IV Experiment

File plain.txt được đề cập ở task này trong phần cài đặt tên là task5_plain.txt, file task5_plain.txt và các file input/output khác được đặt trong Labsetup/Files thay vì trong thư mục của task này.

Trước tiên em tạo 1 file plaintext tên là plain.txt, chỉ chứa string "Ha Chi Hao" bên trong.

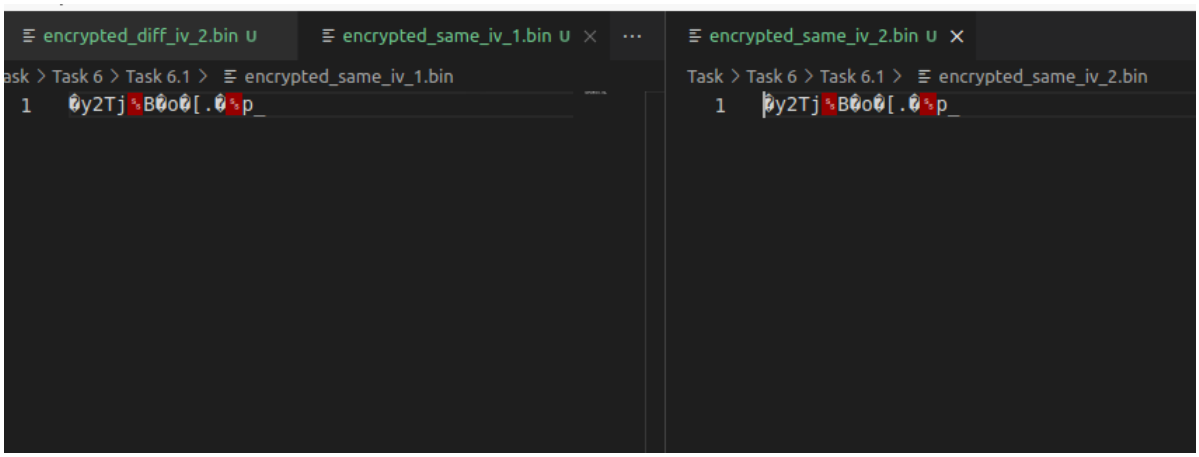
Sau đó em sẽ thực hiện encrypt file đó 2 lần với cùng key và cùng IV:

```
$ openssl enc -e -aes-128-cbc -in plain.txt -out
encrypted_same_iv_1.bin -K 0123456789abcdef0123456789abcdef -iv
11111111111111111111111111111111
```

```
$ openssl enc -e -aes-128-cbc -in plain.txt -out
encrypted_same_iv_2.bin -K 0123456789abcdef0123456789abcdef -iv
11111111111111111111111111111111
```

Dùng lệnh openssl enc cho việc encryption (-e), sử dụng thuật toán AES-128, encryption mode là CBC (-aes-128-cbc), có input file là plain.txt (-in plain.txt), và output ra file encrypted_same_iv.bin (-out encrypted_same_iv.bin), với key là 0123456789abcdef0123456789abcdef (-K 0123456789abcdef0123456789abcdef) và IV là 11111111111111111111111111111111 (-iv 11111111111111111111111111111111)

Cả 2 đều có kết quả giống nhau:



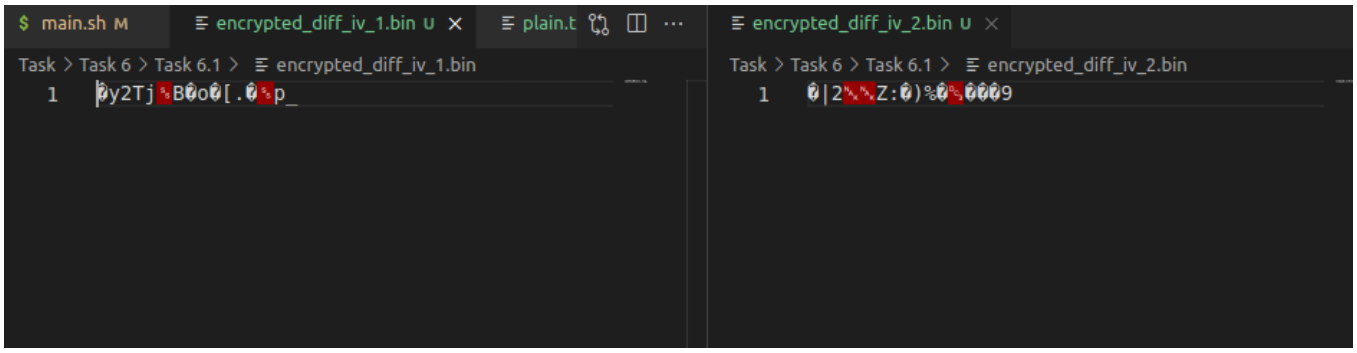
Rồi sau đó em thực hiện encrypt thêm 2 lần với 2 IV khác nhau:

$ openssl enc -e -aes-128-cbc -in plain.txt -out encrypted_diff_iv_1.bin -K 0123456789abcdef0123456789abcdef -iv 11111111111111111111111111111111

$ openssl enc -e -aes-128-cbc -in plain.txt -out encrypted_diff_iv_2.bin -K 0123456789abcdef0123456789abcdef -iv 22222222222222222222222222222222

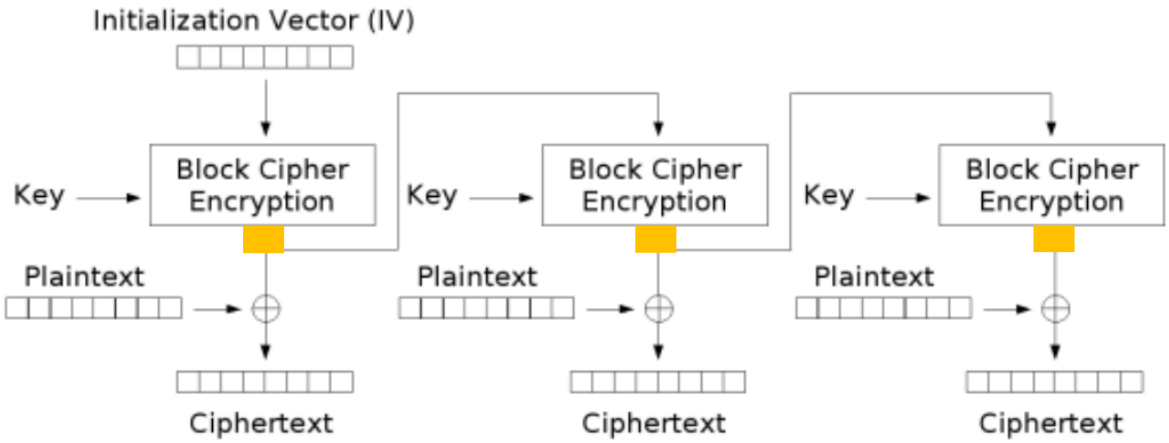
Dùng lệnh openssl enc cho việc encryption (-e), sử dụng thuật toán AES-128, encryption mode là CBC (-aes-128-cbc), có input file là plain.txt (-in plain.txt), và output ra file encrypted_diff_iv.bin (-out encrypted_diff_iv.bin), với key là 0123456789abcdef0123456789abcdef (-K 0123456789abcdef0123456789abcdef) và IV là 11111111111111111111111111111111 (-iv 11111111111111111111111111111111), và 22222222222222222222222222222222 (-iv 22222222222222222222222222222222)

Kết quả:



Kết luận: sử dụng IV giống nhau với cùng 1 plaintext với cùng 1 key sẽ ra kết quả encrypt giống nhau, cho nên sẽ rất dễ bị tấn công. Không bao giờ sử dụng lại 1 IV.

b) Task 6.2. Common Mistake: Use the Same IV



Output Feedback (OFB) mode encryption

Ta xem sơ đồ quá trình encrypt của OFB, ta thấy rằng việc tạo ra kết quả encrypt của block hiện tại (các khối màu cam) thật ra không phụ thuộc vào plaintext hay ciphertext của block trước, mà phụ thuộc vào khối màu cam trước đó, mà khối màu cam đầu tiên phụ thuộc vào Key và IV. Cho nên ta chỉ cần biết được Key và IV, ta có thể suy ra được mọi khối màu cam, và khi **Key và IV giống nhau**, các **khối màu cam cũng sẽ giống nhau**. Một khối màu cam cũng có thể được suy ra khi ta biết cả plaintext và ciphertext (vì tính chất Associative của XOR). Từ đó suy ra điều sau:

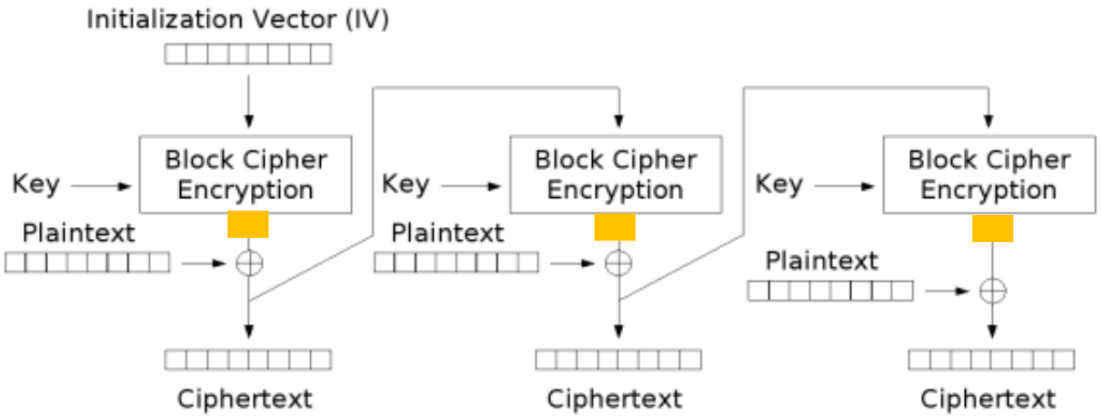
$$khối\ cam = p_1 \oplus c_1$$

$$p_2 = khối\ cam \oplus c_2 \quad //khối\ cam\ này = khối\ cam\ ở\ dòng\ trên$$

$$=> p_2 = p_1 \oplus c_1 \oplus c_2$$

Để có thể thực hiện những điều ở trên, đặc biệt là việc dùng p_1, c_1 để suy ra khối cam, thì việc biết trước p_1 là một dữ kiện quan trọng và em đã lợi dụng nó để có thể truy xuất được thêm những thông tin khác từ thuật toán encrypt, việc lợi dụng như vậy được gọi là *known-plaintext attack*.

Để có thể thực hiện công thức được suy ra ở trên, em viết ra 1 script python, sử dụng hàm bytes và hàm bytearray.fromhex để chuyển plaintext và 2 ciphertext thành 3 dãy byte, sau đó em sử dụng hàm XOR 2 dãy byte bên trong file sample_code.py được đính kèm trong LabSetup.zip để XOR theo như công thức trên. Ra được kết quả là “Order: Launch a missile!”.



Cipher Feedback (CFB) mode encryption

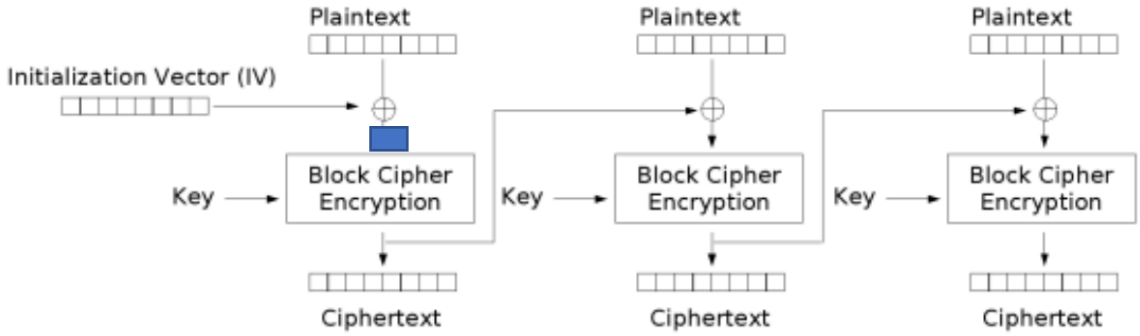
Đối với CFB, thì nó sẽ không có tính chất như OFB là Key, IV giống nhau thì mọi block màu cam giống nhau, vì lúc này các khối màu cam đã phải phụ thuộc vào kết quả của việc XOR khối màu cam và **plaintext** của block trước. Cho nên dù Key, IV có giống nhau, thì ta chỉ suy ra được khối màu cam đầu tiên là giống nhau chứ không phải những khối màu cam sau.

Khi Key, IV giống nhau, việc biết trước 1 cặp plaintext p_1 và ciphertext c_1 và 1 ciphertext c_2 khác chỉ có thể suy ra được p_2 khi p_1, c_1 nằm ở block đầu và c_2 nằm ở block 2, hoặc c_2 nằm ở block đầu, p_1, c_1 nằm ở block 2.

c) Task 6.3. Common Mistake: Use a Predictable IV

Gọi thông điệp đầu tiên mà Bob gửi (thông điệp cần tìm) là p_1, ciphertext của thông điệp đó là c_1, và IV dùng để encrypt thông điệp đó là iv_1. Một plaintext khác bất kỳ là p_2, ciphertext của nó là c_2 và IV dùng để encrypt nó là iv_2.

Vì p_1 chỉ có thể là “Yes” hoặc “No” nên ta hoàn toàn có thể kiểm chứng từng trường hợp xem nó có phải là thông điệp đúng hay không. Như vậy vấn đề còn lại mà ta cần giải quyết đó chính là làm cách nào để kiểm chứng, thì ta quan sát sơ đồ quá trình encrypt của CBC:



Cipher Block Chaining (CBC) mode encryption

Ta thấy rằng ta có thể "giả vờ" cho input là $p_1 \oplus iv_1$ khi ta đang thực hiện với p_2 và iv_2 bằng cách lợi dụng tính chất của phép tính XOR, ta có thể bỏ đi iv_2 bằng cách cho nó XOR với chính nó trong p_2, và khi iv_2 mất đi, ta muốn input còn lại chỉ là $p_1 \oplus iv_1$, nên suy ra ta nên xây dựng p_2 như sau:

$$p_2 = p_1 \oplus iv_1 \oplus iv_2$$

Khi ta thế p_2 như vậy thì khối màu xanh dương sẽ bằng $p_1 \oplus iv_1$. Kiểm chứng lại:

$p_2 \oplus iv_2 = p_1 \oplus iv_1 \oplus iv_2 \oplus iv_2 = p \oplus iv_1$ (tính chất Self-inverse của XOR)

Lúc này thì c_2 sẽ bằng c_1, vì input của Block Cipher Encryption của 2 trường hợp đều là $p_1 \oplus iv_1$.

Những gì mà em vừa thực hiện ở trên được gọi là *chosen-plaintext attack*, vì em vừa cố ý xây dựng một plaintext để có thể dùng nó để trích xuất thêm thông tin từ server.

Vì 2 kết quả encrypt bằng nhau, nên ta chỉ cần lần lượt thế "Yes" và "No" vào bên trong p_1, để vào công thức trên để ra p_2 và cho Bob encrypt p_2 đó, nếu kết quả encrypt giống với kết quả encrypt thông điệp của Bob thì chứng tỏ p_1 mà chúng ta chọn là đúng.

Đối với task này thì em phải sử dụng docker-compose để chạy 1 encryption oracle, tượng trưng cho Bob, đầu tiên thì em sẽ vào folder Labsetup (được giải nén từ Labsetup.zip) và chạy câu lệnh sau đây để xây dựng môi trường:

```
docker-compose build
```

Sau đó em chạy câu lệnh dưới đây để bắt đầu chạy server:

```
docker-compose up
```

Khi gõ xong câu lệnh trên thì kết quả ra như sau:

```
[04/08/22]seed@VM:~/.../Labsetup$ docker-compose up
Recreating oracle-10.9.0.80 ... done
Attaching to oracle-10.9.0.80
oracle-10.9.0.80 | Server listening on 3000 for known_iv
oracle-10.9.0.80 | Connect to 3000, launching known_iv
oracle-10.9.0.80 | Connect to 3000, launching known_iv
```

Sau đó em phải để cho nó chạy liên tục để có thể nhận plaintext và trả lại ciphertext, em mở một tab terminal mới và kết nối tới nó bằng câu lệnh:

```
nc 10.9.0.80 3000
```

```
[04/08/22]seed@VM:~/.../Labsetup$ nc 10.9.0.80 3000
Bob's secret message is either "Yes" or "No", without quotations.
Bob's ciphertex: ff59264944adff46d17bfd4858aff4c4
The IV used    : b88438959c9f52269f3fb73e8ed7829d

Next IV        : 427332019d9f52269f3fb73e8ed7829d
Your plaintext :
```

Trước khi có thể kiểm chứng thì em cần viết 1 script python nhỏ để thực hiện các phép toán XOR ở biểu thức trên để có thể ra p_2 và từ đó em có thể dùng để nhập vào plaintext cho Bob encrypt.

Đầu tiên em định nghĩa hàm XOR 2 dãy byte mà đã được cung cấp trong sample_code.py mà em đã sử dụng ở Task 6.2.

```
def xor(first, second):
   return bytearray(x^y for x,y in zip(first, second))
```

Sau đó em sử dụng argv trong thư viện sys để có thể lấy các tham số từ terminal khi script được chạy:

```
from sys import argv
_, first, second, third = argv
```

Em định nghĩa p_1, iv_1, iv_2, với p_1 thì em sẽ chuyển tham số đầu tiên sang dãy byte trước, sau đó em sẽ tính padding và thêm vào dãy byte đó bằng hàm extend, vì padding scheme là PKCS5 nên những byte padding sẽ có giá trị bằng với kích thước padding chứ không phải các số 0. Với iv_1, iv_2 thì em chỉ cần chuyển nó từ hex sang dãy byte bằng hàm fromhex.

```
p1 = bytearray(first, encoding='utf-8')
padding = 16 - len(p1) % 16
p1.extend([padding]*padding)

iv1 = bytearray.fromhex(second)
```

```
iv2 = bytearray.fromhex(third)
```

Sau cùng chỉ là bước XOR 3 tham số theo công thức và in ra p_2 dưới dạng hex (vì encryption oracle đó nhận plaintext là hex):

```
p2 = xor(xor(p1, iv1), iv2)

print("".join("{:02x}".format(x) for x in p2))
```

Như vậy thì script để XOR 3 số input theo công thức đã xong, em kết nối lại oracle:

```
[04/08/22]seed@VM:~/.../Labsetup$ nc 10.9.0.80 3000
Bob's secret message is either "Yes" or "No", without quotations.
Bob's ciphertex: 110510ec2209c589f9bf639ccda39e32
The IV used    : 740537020a116f4315a75d58b843f466

Next IV        : 32c8a43a0a116f4315a75d58b843f466
Your plaintext :
```

Lúc này thì c_1=110510ec2209c589f9bf639ccda39e32, iv_1=740537020a116f4315a75d58b843f466, iv_2=32c8a43a0a116f4315a75d58b843f466

Bây giờ em sẽ giả sử p_1=”Yes” và chạy script python trên với các input trên:

```
[04/08/22]seed@VM:~/.../Task63$ python3 calP2.py "Yes" 740537020a116f4315a75d58b843f466 32c8a43a0a116f4315a75d58b843f4
66
1fa8e0350d0d0d0d0d0d0d0d0d0d0d0d
[04/08/22]seed@VM:~/.../Task63$
```

Thì ta ra p_2=1fa8e0350d0d0d0d0d0d0d0d0d0d0d0d, ta lấy nhập vào oracle cho Bob encrypt:

```
[04/08/22]seed@VM:~/.../Labsetup$ nc 10.9.0.80 3000
Bob's secret message is either "Yes" or "No", without quotations.
Bob's ciphertex: 110510ec2209c589f9bf639ccda39e32
The IV used    : 740537020a116f4315a75d58b843f466

Next IV        : 32c8a43a0a116f4315a75d58b843f466
Your plaintext : 1fa8e0350d0d0d0d0d0d0d0d0d0d0d0d
Your ciphertext: 110510ec2209c589f9bf639ccda39e3265c52406a4f017439ae433a7a03b367
c

Next IV        : 9bf9d1740a116f4315a75d58b843f466
Your plaintext :
```

Ta thấy rằng 32 kí tự đầu của ciphertext kết quả (c_2) bằng với ciphertext lúc đầu (c_1) chứng tỏ rằng p_1=”Yes” là thông điệp ban đầu của Bob.

Lí do vì sao c_2 lại có độ dài 64 (32 byte) là vì plaintext đầu vào có đủ 16 byte nên phải padding thêm 16 byte nữa cho plaintext xong mới encrypt (PKCS5). Còn thông điệp của Bob là “Yes” chỉ có 3 byte nên chỉ padding tới 16 byte là đủ.

7. Task 7: Programming using the Crypto Library

Để có thể tìm được key khi biết plaintext, ciphertext và IV, và key là 1 từ nằm trong từ điển có độ dài nhỏ hơn 16 kí tự. Thì ý tưởng thuật toán đó chính là brute force qua mọi từ trong từ điển, tìm những từ nhỏ hơn 16 kí tự và thử encrypt plaintext với IV và key đó xem có ra lại ciphertext hay không.

Để cài đặt thuật toán trên thì em sử dụng python, mới vô em sẽ khai báo 3 biến plaintext, ciphertext và IV để sử dụng:

```
plain_text = bytearray("This is a top secret.", encoding='utf-8')
cipher_text =
bytearray.fromhex("764aa26b55a4da654df6b19e4bce00f4ed05e09346fb0e762583cb7da2ac93a
2")
iv = bytearray.fromhex("aabbccddeeff00998877665544332211")
```

Về từ điển thì đã có file words.txt được đính kèm sẵn trong LabSetup.zip nên em không cần phải tìm ở chỗ nào khác, em sẽ thực hiện đọc mọi dòng bên trong words.txt. Em không dùng hàm readlines() vì các dòng sẽ bị dư dấu xuống hàng:

```
with open('./words.txt') as f:
      words = f.read().splitlines()
```

Sau đó em tạo 1 hàm để thực hiện mục đích chính (sử dụng hàm vì có thể dùng return để dừng hàm), trong đó có 1 vòng for để chạy qua hết mọi từ trong words và chỉ xét những từ có độ dài nhỏ hơn 16:

```
def findKey():
    for w in words:
        if len(w) < 16:
```

Bên trong if, trước tiên em sẽ xây dựng key hiện tại mà thuật toán sẽ xử lí, sẽ lấy từ hiện tại và cộng thêm số kí tự ‘#’ còn thiếu để trở thành đủ độ dài 16:

```
currentKey = w + '#'*(16-len(w))
```

Sau đó sẽ định nghĩa một object Cipher mới bằng thư viện Crypto.Cipher.AES, với key là key được tạo ra ở trên (chuyển thành dãy byte), mode là CBC, và IV là iv được định nghĩa lúc đầu.

```
aes_cbc = AES.new(key=bytearray(currentKey,encoding='utf-8'), mode=AES.MODE_CBC, iv=iv)
```

Bây giờ thì em sẽ thử encrypt plaintext bằng key hiện tại với object Cipher trên để xem nó có bằng ciphertext lúc đầu hay không, nhưng trước khi encrypt thì plaintext sẽ được thêm padding bằng hàm pad trong thư viện Crypto.Util.Padding:

```
currentC = aes_cbc.encrypt(pad(plain_text, 16))
```

Sau cùng là bước so sánh kết quả encrypt hiện tại với ciphertext ở lúc đầu, nếu chúng bằng nhau thì chứng tỏ được key hiện tại chính là key mà ta đang tìm, chương trình sẽ in ra key và dừng hàm. Nếu không bằng nhau thì hàm sẽ được thực hiện tiếp cho tới khi nào hết từ thì thôi, nếu hết từ mà không tìm được key thì chương trình sẽ dừng mà không in ra gì.

```
if currentC == cipher_text:
      print(w)
      return
```

Kết quả: **Syracuse**