IIT CS458: Introduction to Information Security

Homework 2: Secret-Key Encryption

My Dinh

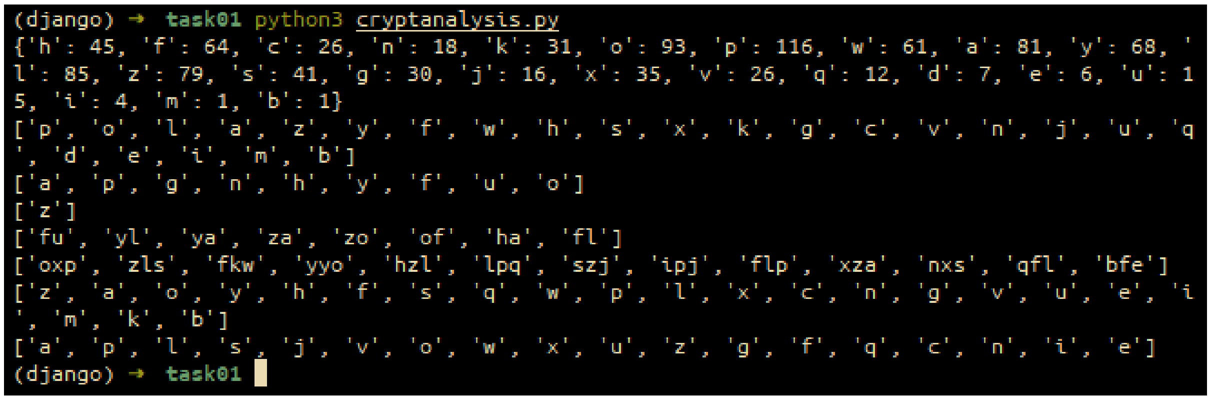
2 Lab Tasks

2.1 Task 1: Frequency Analysis

For the first task, I used Python3 to write a program for cryptanalysis. This program includes functions for preprocessing text (remove special characters and newline), counting letter frequency, doubled letter word, one-letter words, two-letter words, three-letter words, initial letters, final letters.

```python
import re


# remove special characters and new line
def process_text(text) -> str:
    return re.sub('[^A-Za-z0-9 ]+', '', text.replace("\n", " ")).strip()


# sort the keys in given word dictionary d by their frequency
def sort_frequency(d) -> list:
    return sorted(d, key=lambda x: -d[x])


# count letter frequency in text
def count_letters(text) -> dict:
    d = {}

    for c in text:
        if c.isalpha():
            d.setdefault(c, 0)
            d[c] += 1

    return d


# count words in the text with given length n
def count_words(words, n) -> dict:
    d = {}
    for w in words:
        if len(w) == n:
            d.setdefault(w, 0)
            d[w] += 1
    return d


# count initial letter frequency
def count_initial(words) -> dict:
    d = {}

    for w in words:
        d.setdefault(w[0], 0)
        d[w[0]] += 1

    return d


# count final letter frequency
def count_final(words) -> dict:
    d = {}

    for w in words:
        d.setdefault(w[-1], 0)
        d[w[-1]] += 1

    return d


# count doubled letter frequency
def count_doubled(words) -> dict:
    d = {}

    for w in words:
        for i in range(len(w)-1):
            if w[i] == w[i+1]:
                d.setdefault(w[i], 0)
                d[w[i]] += 1

    return d


if __name__ == "__main__":
    f = open("./ciphertext.txt")

    text = process_text(f.read())
    words = text.split(" ")

    letter_freq = count_letters(text)
    doubled_freq = count_doubled(words)
    one_letter_words = count_words(words, 1)
    two_letter_words = count_words(words, 2)
    three_letter_words = count_words(words, 3)
    initial_freq = count_initial(words)
    final_freq = count_final(words)

    print(letter_freq)
    print(sort_frequency(letter_freq))
    print(sort_frequency(doubled_freq))
    print(sort_frequency(one_letter_words))
    print(sort_frequency(two_letter_words))
    print(sort_frequency(three_letter_words))
    print(sort_frequency(initial_freq))
    print(sort_frequency(final_freq))

    f.close()
```

Running the program above would give the following result



```
(django) ➜  task01 python3 cryptanalysis.py
{'h': 45, 'f': 64, 'c': 26, 'n': 18, 'k': 31, 'o': 93, 'p': 116, 'w': 61, 'a': 81, 'y': 68, '
l': 85, 'z': 79, 's': 41, 'g': 30, 'j': 16, 'x': 35, 'v': 26, 'q': 12, 'd': 7, 'e': 6, 'u': 1
5, 'i': 4, 'm': 1, 'b': 1}
['p', 'o', 'l', 'a', 'z', 'y', 'f', 'w', 'h', 's', 'x', 'k', 'g', 'c', 'v', 'n', 'j', 'u', 'q
', 'd', 'e', 'i', 'm', 'b']
['a', 'p', 'g', 'n', 'h', 'y', 'f', 'u', 'o']
['z']
['fu', 'yl', 'ya', 'za', 'zo', 'of', 'ha', 'fl']
['oxp', 'zls', 'fkw', 'yyo', 'hzl', 'lpq', 'szj', 'ipj', 'flp', 'xza', 'nxs', 'qfl', 'bfe']
['z', 'a', 'o', 'y', 'h', 'f', 's', 'q', 'w', 'p', 'l', 'x', 'c', 'n', 'g', 'v', 'u', 'e', 'i
', 'm', 'k', 'b']
['a', 'p', 'l', 's', 'j', 'v', 'o', 'w', 'x', 'u', 'z', 'g', 'f', 'q', 'c', 'n', 'i', 'e']
(django) ➜  task01 
```

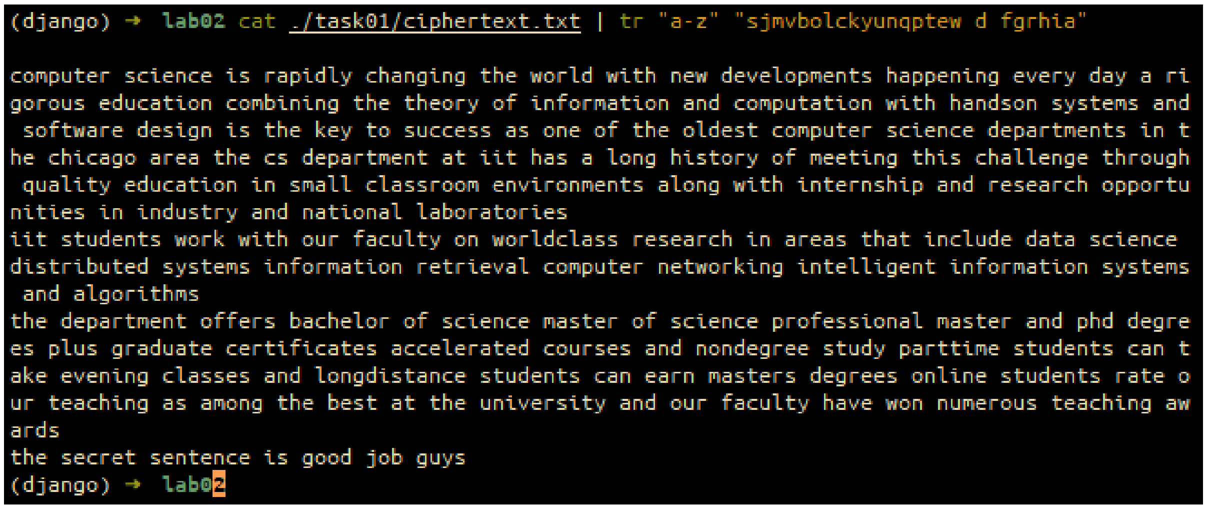
Since the plaintext of the encrypted message is in English, the Cryptanalysis Form from Class Activity 1 can be used again to analyze the ciphertext.

1.
   - Most frequent English letters: **e t a o i n s**
   - Ciphertext frequencies

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
81	1	26	7	6	64	30	46	4	16	31	85	1	18	93	116	12	0	41	0	15	26	61	35	68	79

2.
   - 1-letter English words: **a i**
   - One letter word in ciphertext: **z**
3.
   - Most frequently doubled letters in English: **s e t f l m o**
   - Double letters in ciphertext: **a, p, g, f, y, h, n, u**
4.
   - Most frequent 2-letter words in English: **an, at, as, he, be, in, is, it, on, or, to, of, do, go, no, so, my**
   - Two-letter words in ciphertext: **fu, yl, ya, za, zo, of, ha, fl**
5.
   - Most frequent 3-letter words in English: **the, and, for, was, his, not, but, you, are, her**
   - Three-letter words in ciphertext: **oxp, zls, fkw, yyo, hzl, lpq, szj, ipj, flp, xza, nxs, qfl, bfe**
6.
   - Most frequent initial letters in English: **t a s o i**
   - Initial letters in ciphertext: **z, a, o, y, h, f, s, q, w, p, l, x, c, n, g, v, u, e, i, m, k, b**
7.
   - Most frequent final letters in English: **e s d n t**
   - Final letters in ciphertext: **a, p, l, s, j, v, o, w, x, u, z, g, f, q, c, n, i, e**

Using `tr` tool to map the letters in the ciphertext back to the plaintext, the result is



```
(django) → lab02 cat ./task01/ciphertext.txt | tr "a-z" "sjmvbolckyunqptew d fgrhia"

computer science is rapidly changing the world with new developments happening every day a ri
gorous education combining the theory of information and computation with handson systems and
 software design is the key to success as one of the oldest computer science departments in t
he chicago area the cs department at iit has a long history of meeting this challenge through
 quality education in small classroom environments along with internship and research opportu
nities in industry and national laboratories
iit students work with our faculty on worldclass research in areas that include data science
distributed systems information retrieval computer networking intelligent information systems
 and algorithms
the department offers bachelor of science master of science professional master and phd degre
es plus graduate certificates accelerated courses and nondegree study parttime students can t
ake evening classes and longdistance students can earn masters degrees online students rate o
ur teaching as among the best at the university and our faculty have won numerous teaching aw
ards
the secret sentence is good job guys
(django) → lab02
```

The original message is

computer science is rapidly changing the world with new developments happening every day a rigorous education combining the theory of information and computation with handson systems and software design is the key to success as one of the oldest computer science departments in the chicago area the cs department at iit has a long history of meeting this challenge through quality education in small classroom environments along with internship and research opportunities in industry and national laboratories

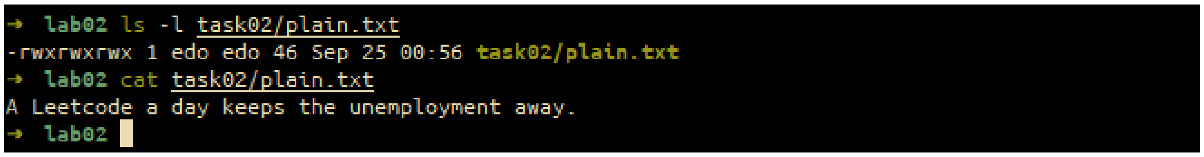
iit students work with our faculty on worldclass research in areas that include data science distributed systems information retrieval computer networking intelligent information systems and algorithms

the department offers bachelor of science master of science professional master and phd degrees plus graduate certificates accelerated courses and nondegree study parttime students can take evening classes and longdistance students can earn masters degrees online students rate our teaching as among the best at the university and our faculty have won numerous teaching awards

the secret sentence is good job guys

2.2 Task 2: Encryption using Different Ciphers and Modes

The ciphertext in `plain.txt` is `"A Leetcode a day keeps the unemployment away."`.



```
→ lab02 ls -l task02/plain.txt
-rwxrwxrwx 1 edo edo 46 Sep 25 00:56 task02/plain.txt
→ lab02 cat task02/plain.txt
A Leetcode a day keeps the unemployment away.
→ lab02
```

Figure 1: Task 2 plaintext information.

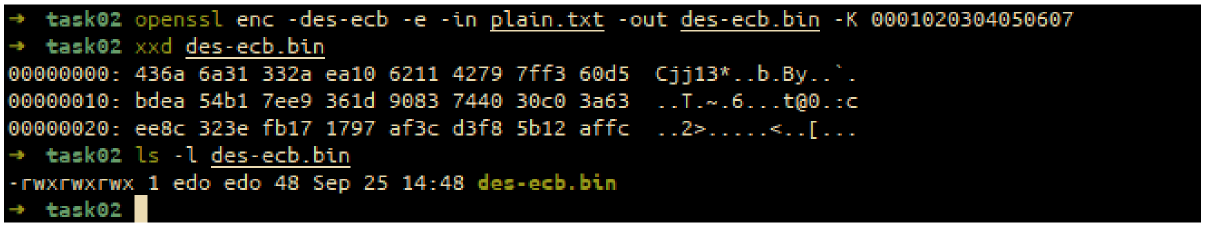
For this task, I am going to use cipher types `des`, `aes-128`, and `aes-256` with mode types `ecb`, `cbc`, and `cfb`.

2.2.1 DES

The **key** used in this encryption is `0001020304050607`.

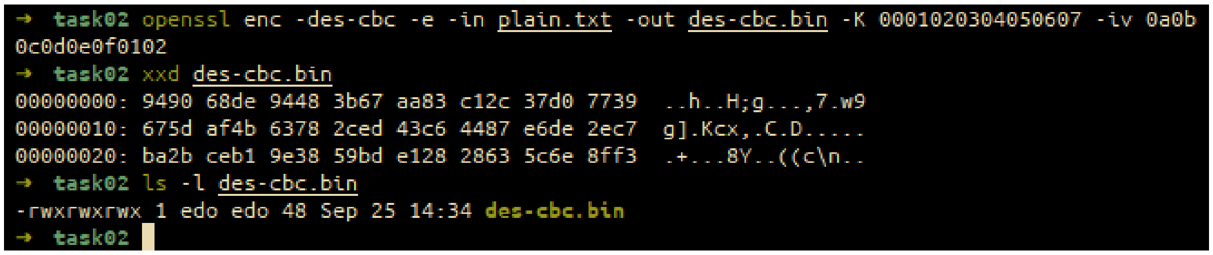
The **initial vector** used in this encryption is `0a0b0c0d0e0f0102` (the initial vector is not needed in ECB mode).

The following figures (*Figure 2, 3, 4*) are the reported results of using encryption type `DES` with mode `ECB`, `CBC`, and `CFB`. The results give us the information about the ciphertext binary and the size of ciphertext.



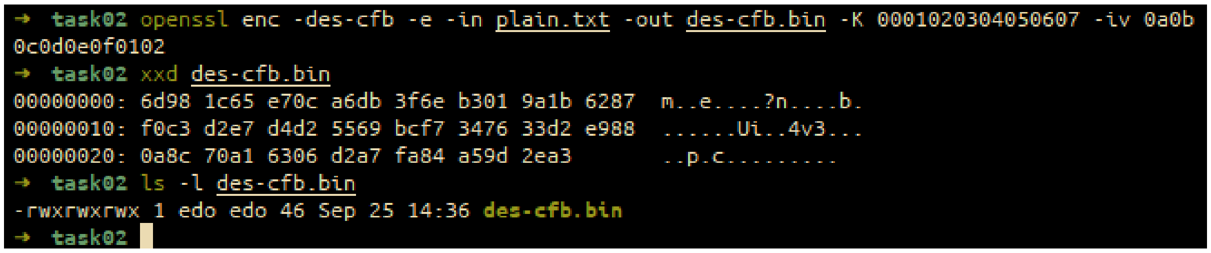
```
➜  task02 openssl enc -des-ecb -e -in plain.txt -out des-ecb.bin -K 0001020304050607
➜  task02 xxd des-ecb.bin
00000000: 436a 6a31 332a ea10 6211 4279 7ff3 60d5  Cjj13*..b.By..`.
00000010: bdea 54b1 7ee9 361d 9083 7440 30c0 3a63  ..T.~.6...t@0.:c
00000020: ee8c 323e fb17 1797 af3c d3f8 5b12 affc  ..2>.....<..[...
➜  task02 ls -l des-ecb.bin
-rwxrwxrwx 1 edo edo 48 Sep 25 14:48 des-ecb.bin
➜  task02
```

Figure 2: Cipher type DES with mode ECB.



```
➜  task02 openssl enc -des-cbc -e -in plain.txt -out des-cbc.bin -K 0001020304050607 -iv 0a0b
0c0d0e0f0102
➜  task02 xxd des-cbc.bin
00000000: 9490 68de 9448 3b67 aa83 c12c 37d0 7739  ..h..H;g...,7.w9
00000010: 675d af4b 6378 2ced 43c6 4487 e6de 2ec7  g].Kcx,.C.D.....
00000020: ba2b ceb1 9e38 59bd e128 2863 5c6e 8ff3  .+...8Y..((c\n..
➜  task02 ls -l des-cbc.bin
-rwxrwxrwx 1 edo edo 48 Sep 25 14:34 des-cbc.bin
➜  task02
```

Figure 3: Cipher type DES with mode CBC.



```
➜  task02 openssl enc -des-cfb -e -in plain.txt -out des-cfb.bin -K 0001020304050607 -iv 0a0b
0c0d0e0f0102
➜  task02 xxd des-cfb.bin
00000000: 6d98 1c65 e70c a6db 3f6e b301 9a1b 6287  m..e....?n....b.
00000010: f0c3 d2e7 d4d2 5569 bcf7 3476 33d2 e988  ......Ui..4v3...
00000020: 0a8c 70a1 6306 d2a7 fa84 a59d 2ea3       ..p.c.........
➜  task02 ls -l des-cfb.bin
-rwxrwxrwx 1 edo edo 46 Sep 25 14:36 des-cfb.bin
➜  task02
```

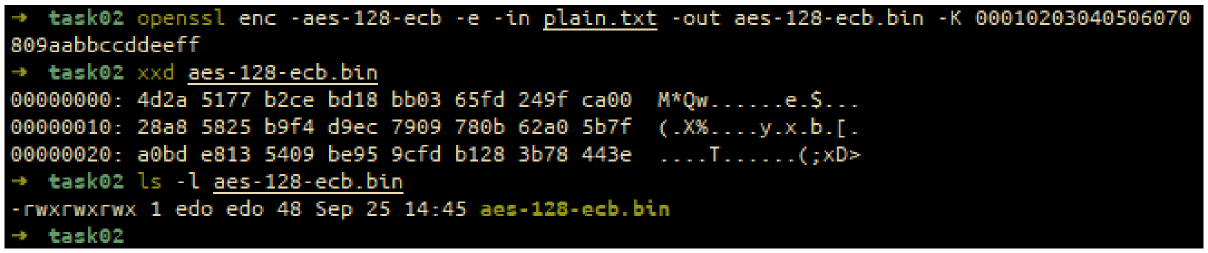
Figure 4: Cipher type DES with mode CFB.

2.2.2 AES-128

The **key** used in this encryption is `00010203040506070809aabbccddeeff`.

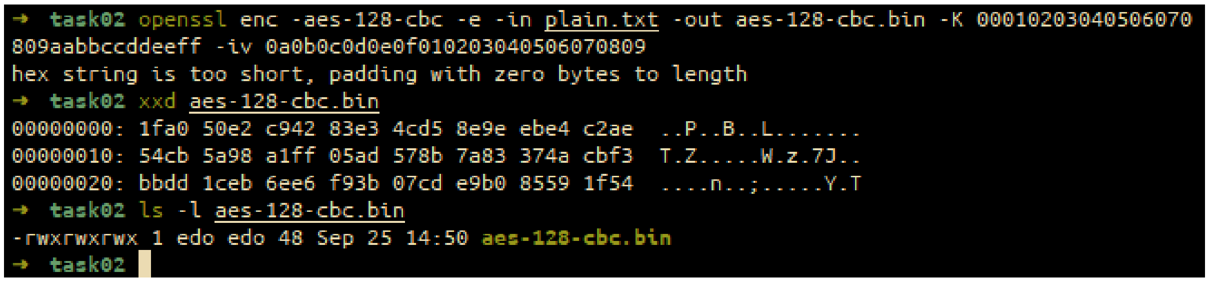
The **initial vector** used in this encryption is `0a0b0c0d0e0f01020304050607080809` (the initial vector is not needed in ECB mode).

The following figures (*Figure 5, 6, 7, 8, 9*) are the reported results of using encryption type `AES-128` with mode `ECB`, `CBC`, and `CFB`. The results give us the information about the ciphertext binary and the size of ciphertext.



```
➜  task02 openssl enc -aes-128-ecb -e -in plain.txt -out aes-128-ecb.bin -K 00010203040506070
809aabbccddeeff
➜  task02 xxd aes-128-ecb.bin
00000000: 4d2a 5177 b2ce bd18 bb03 65fd 249f ca00  M*Qw......e.$...
00000010: 28a8 5825 b9f4 d9ec 7909 780b 62a0 5b7f  (.X%....y.x.b.[.
00000020: a0bd e813 5409 be95 9cfd b128 3b78 443e  ....T......(;xD>
➜  task02 ls -l aes-128-ecb.bin
-rwxrwxrwx 1 edo edo 48 Sep 25 14:45 aes-128-ecb.bin
➜  task02
```

Figure 5: Cipher type AES-128 with mode ECB.

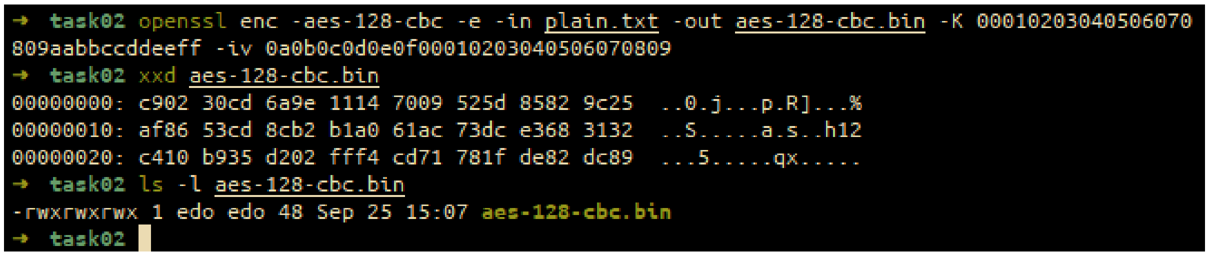


```
➜  task02 openssl enc -aes-128-cbc -e -in plain.txt -out aes-128-cbc.bin -K 00010203040506070
809aabbccddeeff -iv 0a0b0c0d0e0f010203040506070809
hex string is too short, padding with zero bytes to length
➜  task02 xxd aes-128-cbc.bin
00000000: 1fa0 50e2 c942 83e3 4cd5 8e9e ebe4 c2ae  ..P..B..L.......
00000010: 54cb 5a98 a1ff 05ad 578b 7a83 374a cbf3  T.Z.....W.z.7J..
00000020: bbdd 1ceb 6ee6 f93b 07cd e9b0 8559 1f54  ....n..;.....Y.T
➜  task02 ls -l aes-128-cbc.bin
-rwxrwxrwx 1 edo edo 48 Sep 25 14:50 aes-128-cbc.bin
➜  task02
```

Figure 6: Cipher type AES-128 with mode CBC. (1)

(I understood that I should have chosen a better initial vector to match with the size of plaintext but I was too lazy and `openssl` was smart enough to let me do that by doing padding to the initial vector)

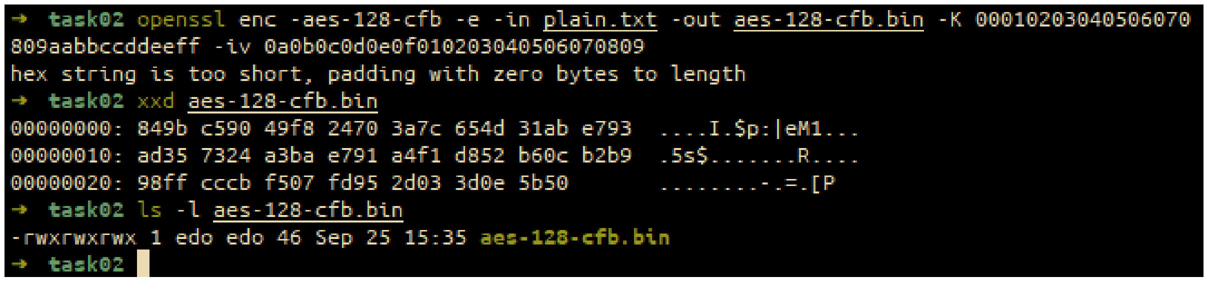
I tried again with a larger size initial vector `0a0b0c0d0e0f00010203040506070809` and it stopped printing out the error.



```
➜  task02 openssl enc -aes-128-cbc -e -in plain.txt -out aes-128-cbc.bin -K 00010203040506070
809aabbccddeeff -iv 0a0b0c0d0e0f00010203040506070809
➜  task02 xxd aes-128-cbc.bin
00000000: c902 30cd 6a9e 1114 7009 525d 8582 9c25  ..0.j...p.R]...%
00000010: af86 53cd 8cb2 b1a0 61ac 73dc e368 3132  ..S.....a.s..h12
00000020: c410 b935 d202 fff4 cd71 781f de82 dc89  ...5.....qx.....
➜  task02 ls -l aes-128-cbc.bin
-rwxrwxrwx 1 edo edo 48 Sep 25 15:07 aes-128-cbc.bin
➜  task02
```

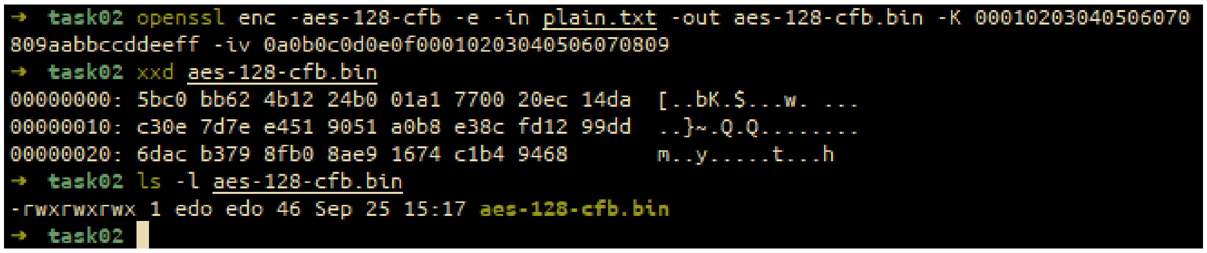
Figure 7: Cipher type AES-128 with mode CBC. (2)

Notice that by changing the position of the zero bytes from the end of the middle initial vector (in the first case with smaller initial vector, zero bytes will be added at the end) to the middle of initial vector, the ciphertext would look completely different.



```
➜  task02 openssl enc -aes-128-cfb -e -in plain.txt -out aes-128-cfb.bin -K 00010203040506070
809aabbccddeeff -iv 0a0b0c0d0e0f010203040506070809
hex string is too short, padding with zero bytes to length
➜  task02 xxd aes-128-cfb.bin
00000000: 849b c590 49f8 2470 3a7c 654d 31ab e793  ....I.$p:|eM1...
00000010: ad35 7324 a3ba e791 a4f1 d852 b60c b2b9  .5s$.......R....
00000020: 98ff cccb f507 fd95 2d03 3d0e 5b50       ........-.=.[P
➜  task02 ls -l aes-128-cfb.bin
-rwxrwxrwx 1 edo edo 46 Sep 25 15:35 aes-128-cfb.bin
➜  task02
```

Figure 8: Cipher type AES-128 with mode CFB. (1)



```
➜  task02 openssl enc -aes-128-cfb -e -in plain.txt -out aes-128-cfb.bin -K 00010203040506070
809aabbccddeeff -iv 0a0b0c0d0e0f00010203040506070809
➜  task02 xxd aes-128-cfb.bin
00000000: 5bc0 bb62 4b12 24b0 01a1 7700 20ec 14da  [..bK.$...w. ...
00000010: c30e 7d7e e451 9051 a0b8 e38c fd12 99dd  ..}~.Q.Q........
00000020: 6dac b379 8fb0 8ae9 1674 c1b4 9468       m..y.....t...h
➜  task02 ls -l aes-128-cfb.bin
-rwxrwxrwx 1 edo edo 46 Sep 25 15:17 aes-128-cfb.bin
➜  task02
```

Figure 9: Cipher type AES-128 with mode CFB. (2)

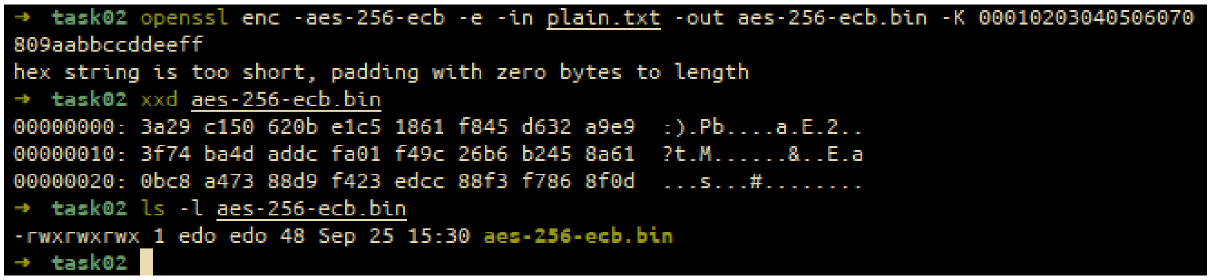
2.2.3 AES-256

The **key** used in this encryption is `00010203040506070809aabbccddeeff`.

Notice that the size of the key in the lab writeup (or the one given above) is much smaller than the requirement size of `aes-256` key (which is 256 bits). `openssl` will fix that by performing padding to add zero bytes at the end of the key.

The **initial vector** used in this encryption is `0a0b0c0d0e0f00010203040506070809` (the initial vector is not needed in ECB mode).

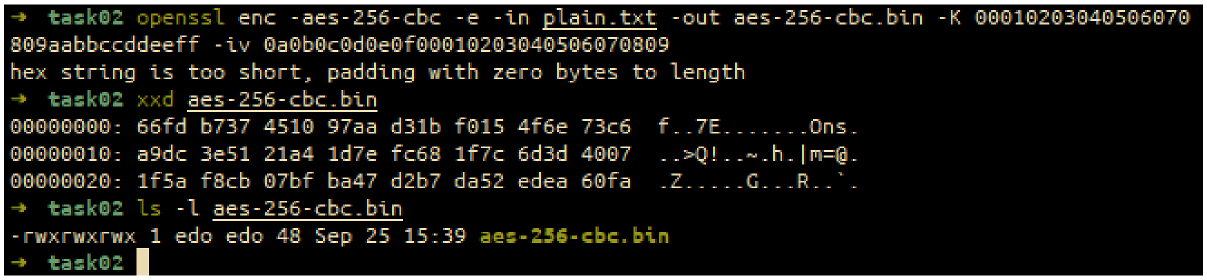
The following figures (*Figure 10, 11, 12*) are the reported results of using encryption type `AES-256` with mode `ECB`, `CBC`, and `CFB`. The results give us the information about the ciphertext binary and the size of ciphertext.



```
➜  task02 openssl enc -aes-256-ecb -e -in plain.txt -out aes-256-ecb.bin -K 00010203040506070
809aabbccddeeff
hex string is too short, padding with zero bytes to length
➜  task02 xxd aes-256-ecb.bin
00000000: 3a29 c150 620b e1c5 1861 f845 d632 a9e9  :).Pb....a.E.2..
00000010: 3f74 ba4d addc fa01 f49c 26b6 b245 8a61  ?t.M......&..E.a
00000020: 0bc8 a473 88d9 f423 edcc 88f3 f786 8f0d  ...s...#........
➜  task02 ls -l aes-256-ecb.bin
-rwxrwxrwx 1 edo edo 48 Sep 25 15:30 aes-256-ecb.bin
➜  task02
```

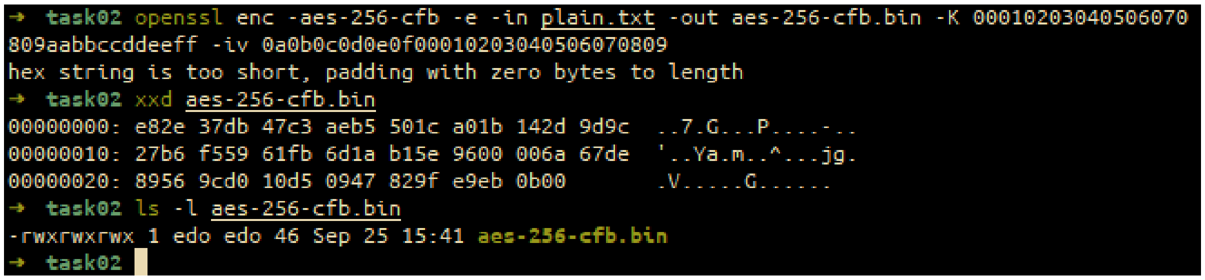
Figure 10: Cipher type AES-256 with mode ECB.

(As expected, it complained about the size of the key being too short, but it fixed the problem by adding zero bytes without crashing the program)



```
➜  task02 openssl enc -aes-256-cbc -e -in plain.txt -out aes-256-cbc.bin -K 00010203040506070
809aabbccddeeff -iv 0a0b0c0d0e0f00010203040506070809
hex string is too short, padding with zero bytes to length
➜  task02 xxd aes-256-cbc.bin
00000000: 66fd b737 4510 97aa d31b f015 4f6e 73c6  f..7E.......Ons.
00000010: a9dc 3e51 21a4 1d7e fc68 1f7c 6d3d 4007  ..>Q!..~.h.|m=@.
00000020: 1f5a f8cb 07bf ba47 d2b7 da52 edea 60fa  .Z.....G...R..`.
➜  task02 ls -l aes-256-cbc.bin
-rwxrwxrwx 1 edo edo 48 Sep 25 15:39 aes-256-cbc.bin
➜  task02
```

Figure 11: Cipher type AES-256 with mode CBC.



```
→ task02 openssl enc -aes-256-cfb -e -in plain.txt -out aes-256-cfb.bin -K 00010203040506070
809aabbccddeeff -iv 0a0b0c0d0e0f000102030405060708090
hex string is too short, padding with zero bytes to length
→ task02 xxd aes-256-cfb.bin
00000000: e82e 37db 47c3 aeb5 501c a01b 142d 9d9c  ..7.G...P....-..
00000010: 27b6 f559 61fb 6d1a b15e 9600 006a 67de  '..Ya.m..^...jg.
00000020: 8956 9cd0 10d5 0947 829f e9eb 0b00       .V.....G......
→ task02 ls -l aes-256-cfb.bin
-rwxrwxrwx 1 edo edo 46 Sep 25 15:41 aes-256-cfb.bin
→ task02 
```

Figure 12: Cipher type AES-256 with mode CFB.

2.2.4 Observations

Only in CFB mode that the size of ciphertext is the same as the size of the plaintext. The two other modes, ECB and CBC, have result ciphertext with size larger than the plaintext.

The ciphertext generated from different cipher types and modes are different despite using the same key and initial vector.

`openssl` automatically patches the wrong the key or initial vector size by adding zero bytes at the end (to grow the size of them) or ignoring the excess bytes if the hex string is too long.

2.3 Task 3: Encryption Mode - ECB vs. CBC

In this task, I tested encrypting the bitmap image with ECB and CBC mode by combining them with DES cipher type.

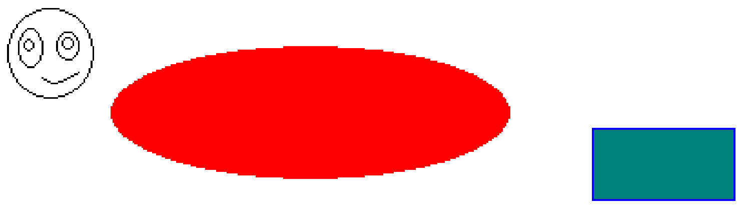
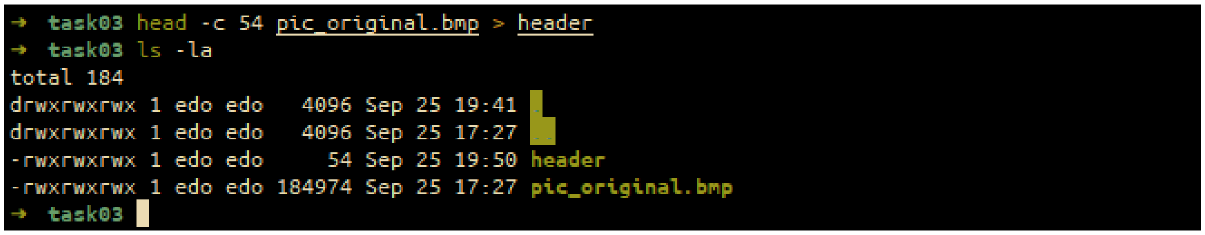


Figure 13: Original bitmap image.

Since the first 54 bytes of bitmap image contains the header information about the picture, I would like to extract and store it in another file for later use.



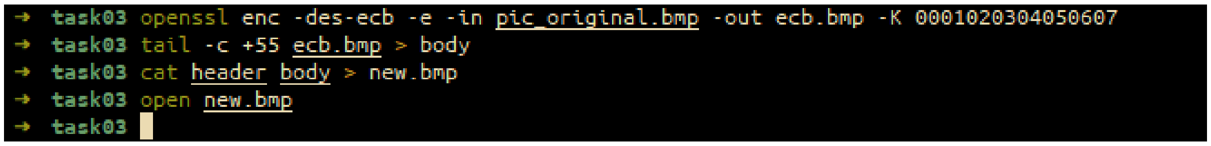
```
→ task03 head -c 54 pic_original.bmp > header
→ task03 ls -la
total 184
drwxrwxrwx 1 edo edo   4096 Sep 25 19:41 .
drwxrwxrwx 1 edo edo   4096 Sep 25 17:27 ..
-rwxrwxrwx 1 edo edo     54 Sep 25 19:50 header
-rwxrwxrwx 1 edo edo 184974 Sep 25 17:27 pic_original.bmp
→ task03 
```

Figure 14: Extract bitmap header information.

The **key** used in this encryption is `0001020304050607`.

The **initial vector** used in this encryption is `0a0b0c0d0e0f0102` (the initial vector is not needed in ECB mode).

- **ECB**



```
→  task03 openssl enc -des-ecb -e -in pic_original.bmp -out ecb.bmp -K 0001020304050607
→  task03 tail -c +55 ecb.bmp > body
→  task03 cat header body > new.bmp
→  task03 open new.bmp
→  task03
```

Figure 15: Encrypt bitmap image with DES cipher type and ECB mode.

The result bitmap image is shown in Figure 16.

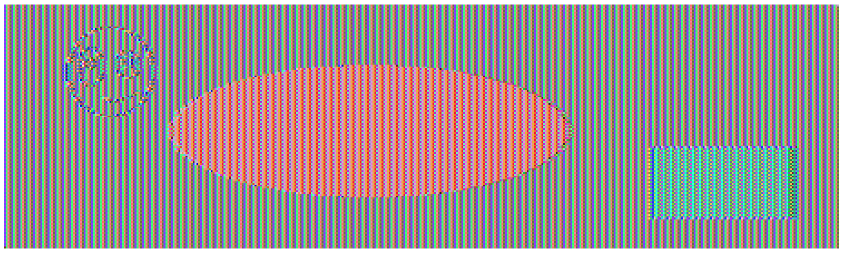
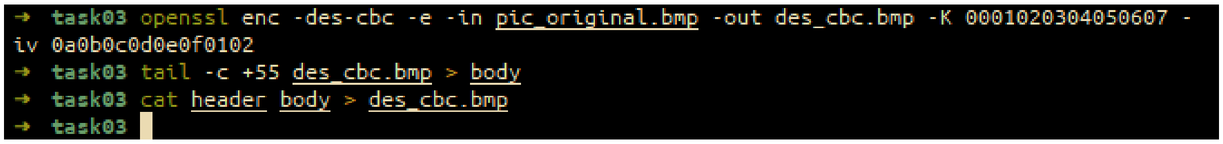


Figure 16: Result bitmap image with DES cipher type and ECB mode.

- **CBC**



```
→  task03 openssl enc -des-cbc -e -in pic_original.bmp -out des_cbc.bmp -K 0001020304050607 -
iv 0a0b0c0d0e0f0102
→  task03 tail -c +55 des_cbc.bmp > body
→  task03 cat header body > des_cbc.bmp
→  task03
```

Figure 17: Encrypt bitmap image with DES cipher type and CBC mode.

The result bitmap image is shown in Figure 18.

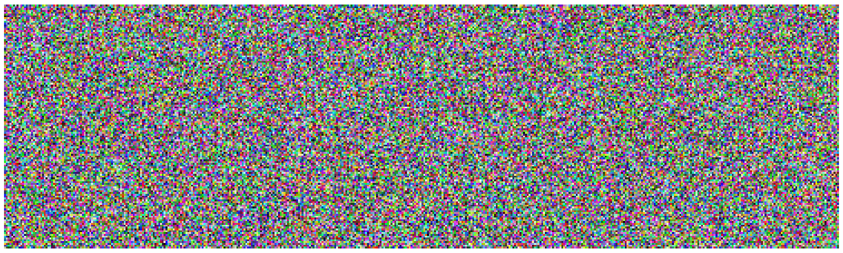


Figure 18: Result bitmap image with DES cipher type and CBC mode.

We can see that the bad guys would definitely get useful information from bitmap image encrypted with ECB mode since it looks like the original bitmap image with some noises. The one encrypted with CBC mode does not give us any useful information from the fact that it looks completely different from the original.

2.3.1 Experiment with another picture

For this experiment, I used the following Illinois Institute of Technology logo as the original message.

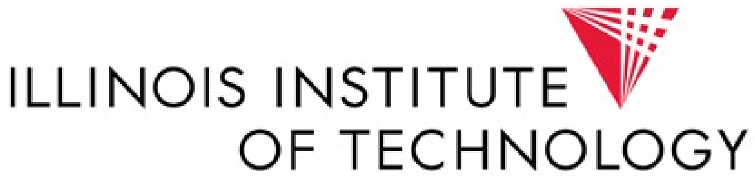
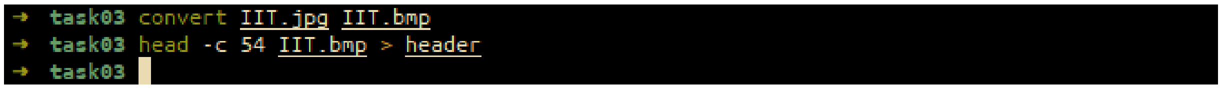


Figure 19: Original message image.

First, I converted the `jpg` format image to `bmp` and repeated the process to get the header information of the original bitmap image and store it `header` file.



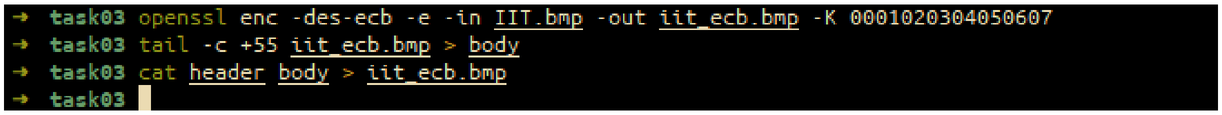
```
➜  task03 convert IIT.jpg IIT.bmp
➜  task03 head -c 54 IIT.bmp > header
➜  task03
```

Figure 20: Extract header information from original bitmap image.

The **key** used in this encryption is `0001020304050607`.

The **initial vector** used in this encryption is `0a0b0c0d0e0f0102` (the initial vector is not needed in ECB mode).

- **ECB**



```
➜  task03 openssl enc -des-ecb -e -in IIT.bmp -out iit_ecb.bmp -K 0001020304050607
➜  task03 tail -c +55 iit_ecb.bmp > body
➜  task03 cat header body > iit_ecb.bmp
➜  task03
```

Figure 21: Encrypt bitmap image with DES cipher type and ECB mode.

The result bitmap image is shown in Figure 22.

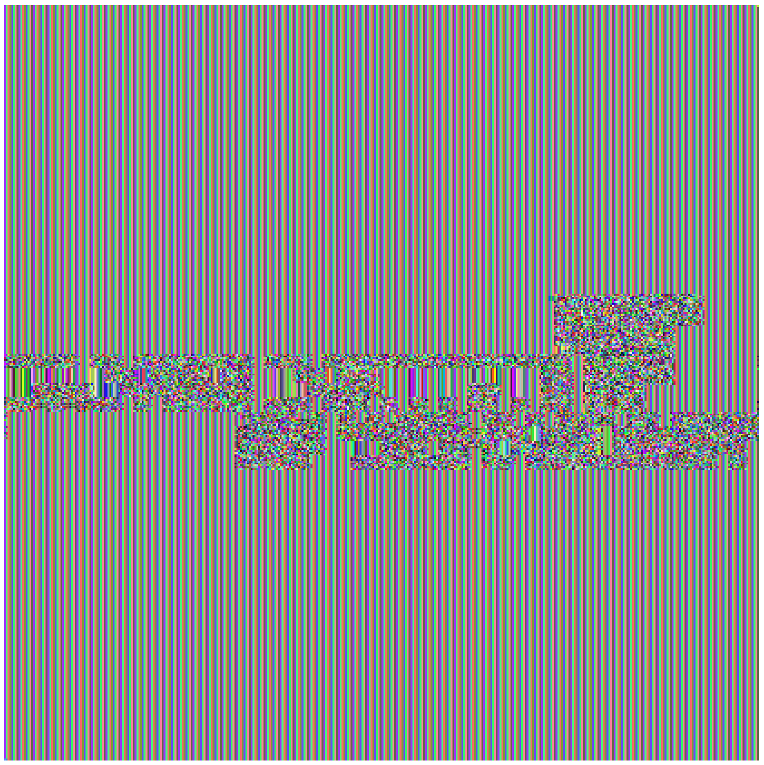
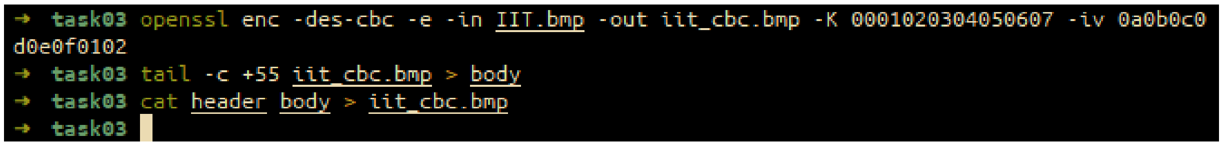


Figure 22: Result bitmap image with DES cipher type and ECB mode.

- **CBC**



```
➜  task03 openssl enc -des-cbc -e -in IIT.bmp -out iit_cbc.bmp -K 0001020304050607 -iv 0a0b0c0
d0e0f0102
➜  task03 tail -c +55 iit_cbc.bmp > body
➜  task03 cat header body > iit_cbc.bmp
➜  task03 
```

Figure 23: Encrypt bitmap image with DES cipher type and CBC mode.

The result bitmap image is shown in Figure 24.

This time, the information given in encrypted image using ECB mode is harder to guess since the block data of the unversity name in the image is not exactly the same so the encrypted data does not look alike. But it still tells us that the noises in the middle are the important information since it looks much different than the encrypted white spaces. And we can see the resemblance in the original and the encrypted image. While the one encrypted with CBC mode looks nothing like the original, which makes it harder to guess the data in the image.

This is because blocks of data with the same message produce the same ciphertext using ECB mode. The blocks of message encrypted in CBC mode are XOR with the initial vector before encryption to create randomnes in the ciphertext.

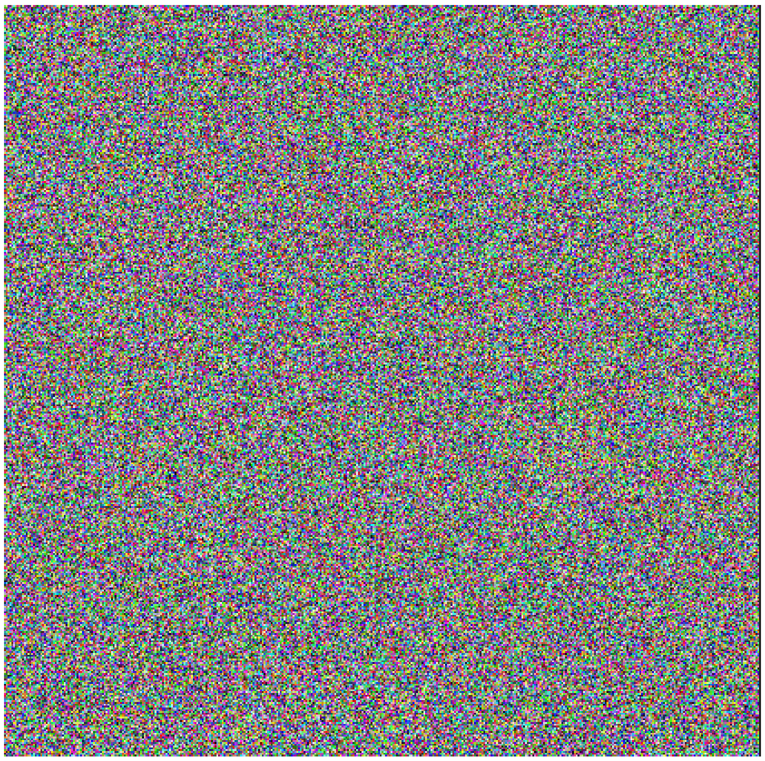
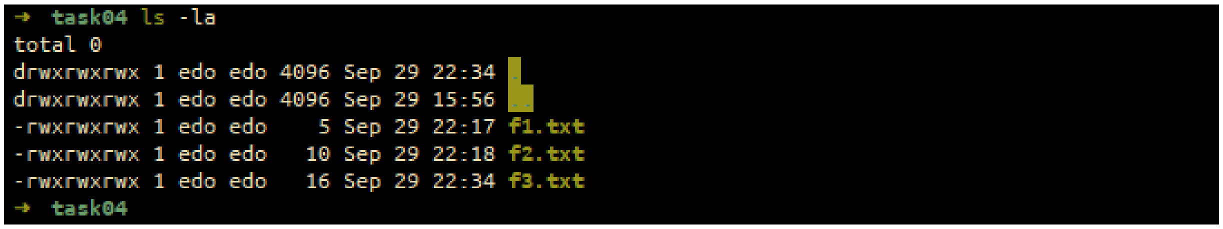


Figure 24: Result bitmap image with DES cipher type and CBC mode.



```
➜  task04 ls -la
total 0
drwxrwxrwx 1 edo edo 4096 Sep 29 22:34 .
drwxrwxrwx 1 edo edo 4096 Sep 29 15:56 ..
-rwxrwxrwx 1 edo edo    5 Sep 29 22:17 f1.txt
-rwxrwxrwx 1 edo edo   10 Sep 29 22:18 f2.txt
-rwxrwxrwx 1 edo edo   16 Sep 29 22:34 f3.txt
➜  task04
```

Figure 25: Plaintext files with their size.

2.4 Task 4: Padding

The content of the plaintext files for 5 bytes, 10 bytes, and 16 bytes are `12345` in `f1.txt`, `1234567890` in `f2.txt`, and `1234567891011121` in `f3.txt`.

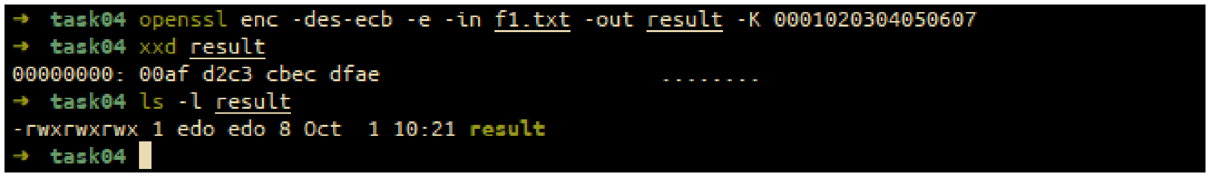
For the first subtask, I will use `DES` cipher type with `ECB`, `CBC`, `CFB`, and `OFB` to encrypt `f1.txt` file (the plaintext is `12345`).

1. The **key** used in this encryption is `0001020304050607` and the **intital vector** is `0a0b0c0d0e0f0102`.

   The encryption modes that need padding are ECB and CBC (the size of plaintext is 5 bytes and the ciphertext is 8 bytes), while CFB and OFB does not need padding (the ciphertext size is 8 bytes).

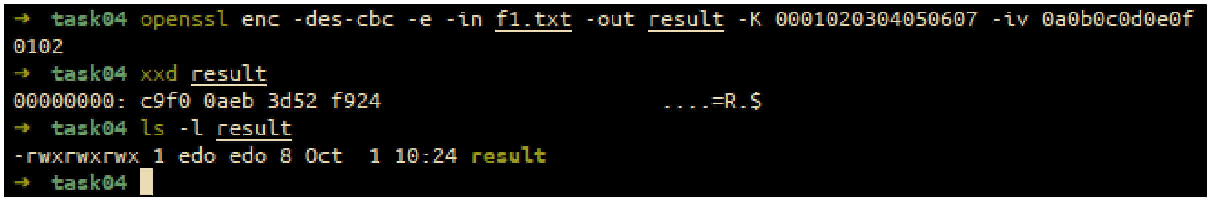
   This is because in CFB and OFB, the blocks of plaintext are not directly used as plaintext for the block cipher encryption, which often requires the plaintext to be 128 bits. Instead, the initial vector

[1]Wikipedia: `https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation`
[2]Wikipedia: `https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation`



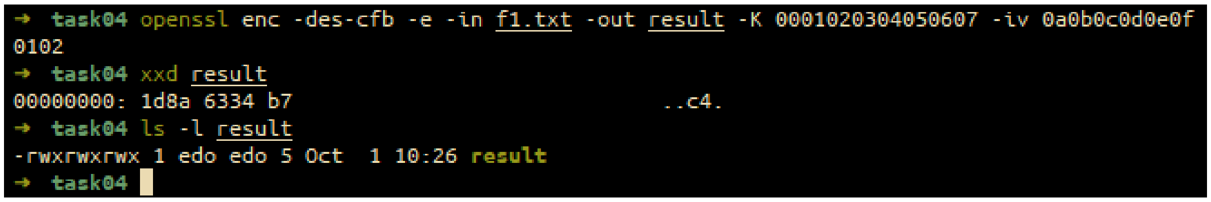
```
➜  task04 openssl enc -des-ecb -e -in f1.txt -out result -K 0001020304050607
➜  task04 xxd result
00000000: 00af d2c3 cbec dfae                      ........
➜  task04 ls -l result
-rwxrwxrwx 1 edo edo 8 Oct  1 10:21 result
➜  task04 
```

Figure 26: Plaintext 12345 encrypted with DES and ECB.



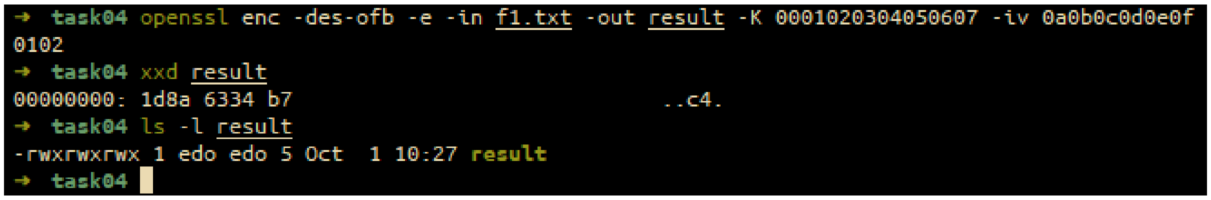
```
➜  task04 openssl enc -des-cbc -e -in f1.txt -out result -K 0001020304050607 -iv 0a0b0c0d0e0f
0102
➜  task04 xxd result
00000000: c9f0 0aeb 3d52 f924                      ....=R.$
➜  task04 ls -l result
-rwxrwxrwx 1 edo edo 8 Oct  1 10:24 result
➜  task04 
```

Figure 27: Plaintext 12345 encrypted with DES and CBC.



```
➜  task04 openssl enc -des-cfb -e -in f1.txt -out result -K 0001020304050607 -iv 0a0b0c0d0e0f
0102
➜  task04 xxd result
00000000: 1d8a 6334 b7                             ..c4.
➜  task04 ls -l result
-rwxrwxrwx 1 edo edo 5 Oct  1 10:26 result
➜  task04 
```

Figure 28: Plaintext 12345 encrypted with DES and CFB.



```
➜  task04 openssl enc -des-ofb -e -in f1.txt -out result -K 0001020304050607 -iv 0a0b0c0d0e0f
0102
➜  task04 xxd result
00000000: 1d8a 6334 b7                             ..c4.
➜  task04 ls -l result
-rwxrwxrwx 1 edo edo 5 Oct  1 10:27 result
➜  task04 
```

Figure 29: Plaintext 12345 encrypted with DES and OFB.

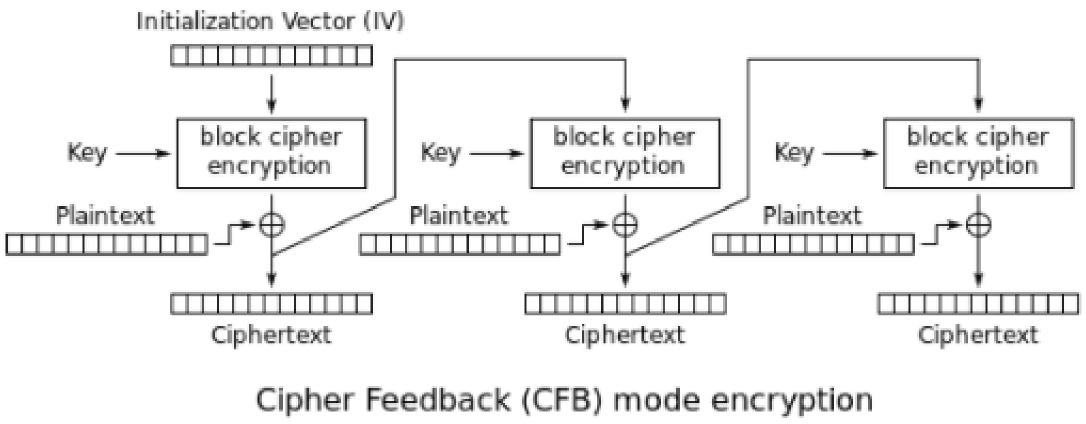


Figure 30: CFB mode encryption.[1]

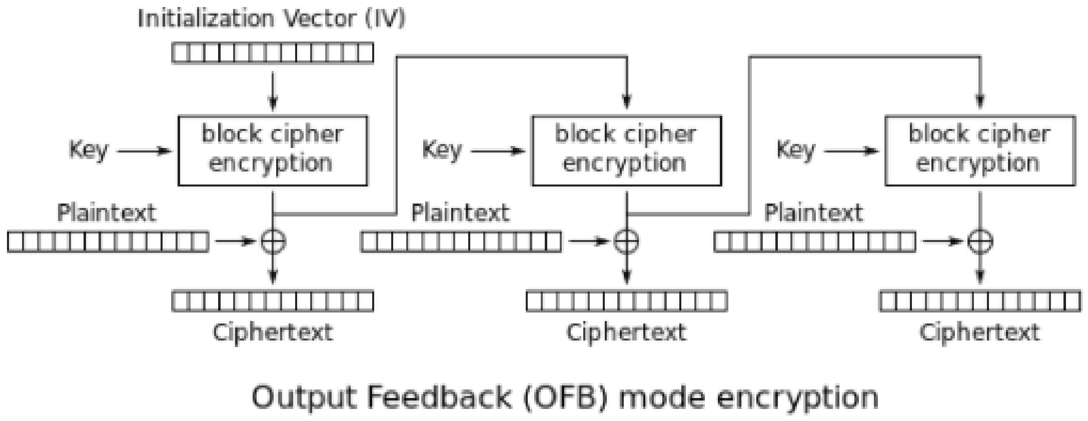
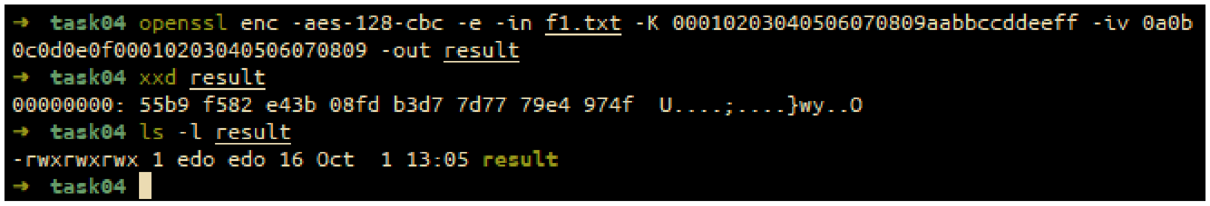

Initialization Vector (IV)
Key
block cipher encryption
Plaintext
Ciphertext
Output Feedback (OFB) mode encryption

Figure 31: OFB mode encryption.[2]

is used as the input for the block cipher encryption and the block of message is only used in XOR operation with the result ciphertext of block cipher encryption and XOR operation performs on bit by bit of data so it does not require any fixed data size.

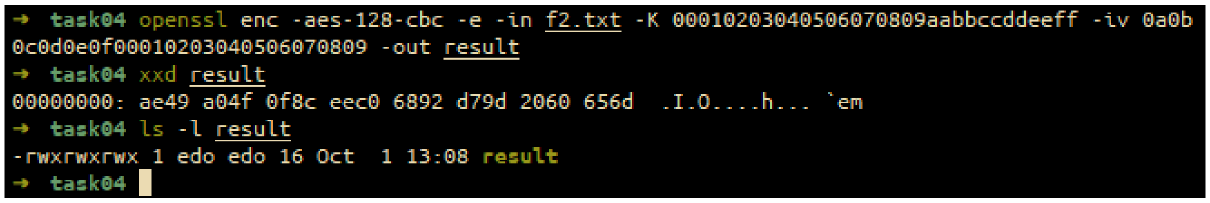
On the other hand, ECB and CBC use blocks of plaintext as the input for block cipher encryption, which requires the block to be in the size of 128 bits. So padding is needed for the blocks of plaintext to be in the right size to perform the encryption.

2. The **key** used in this encryption is `00010203040506070809aabbccddeeff` and the **initial vector** is `0a0b0c0d0e0f00010203040506070809`.



```
➜  task04 openssl enc -aes-128-cbc -e -in f1.txt -K 00010203040506070809aabbccddeeff -iv 0a0b
0c0d0e0f00010203040506070809 -out result
➜  task04 xxd result
00000000: 55b9 f582 e43b 08fd b3d7 7d77 79e4 974f  U....;....}wy..O
➜  task04 ls -l result
-rwxrwxrwx 1 edo edo 16 Oct  1 13:05 result
➜  task04
```

Figure 32: `f1.txt` (5 bytes) encrypted with AES-128 and CBC.



```
➜  task04 openssl enc -aes-128-cbc -e -in f2.txt -K 00010203040506070809aabbccddeeff -iv 0a0b
0c0d0e0f00010203040506070809 -out result
➜  task04 xxd result
00000000: ae49 a04f 0f8c eec0 6892 d79d 2060 656d  .I.O....h... `em
➜  task04 ls -l result
-rwxrwxrwx 1 edo edo 16 Oct  1 13:08 result
➜  task04
```

Figure 33: `f2.txt` (10 bytes) encrypted with AES-128 and CBC.

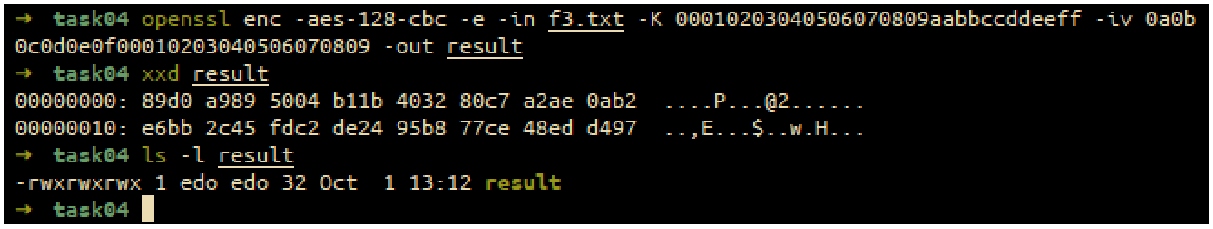
The size of plaintext in `f1.txt` is 5 bytes and its ciphertext is 16 bytes.

The size of plaintext in `f2.txt` is 10 bytes and its ciphertext is 16 bytes.

The size of plaintext in `f3.txt` is 16 bytes and its ciphertext is 32 bytes.

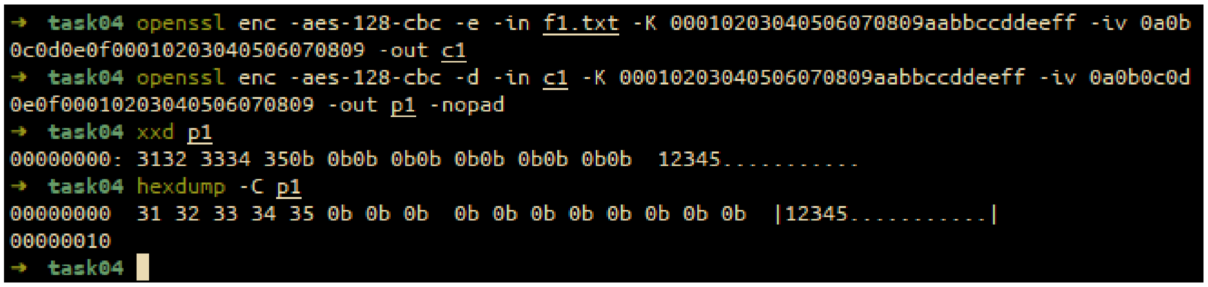
Hex `0b` (which is 11 in decimal) is added to the padding during encryption of `f1.txt` because `f1.txt` block is missing 11 bytes to be 16 bytes (128 bits).

Hex `06` (which is 6 in decimal) is added to the padding during encryption of `f2.txt` because `f2.txt` block is missing 6 bytes to be 16 bytes (128 bits).



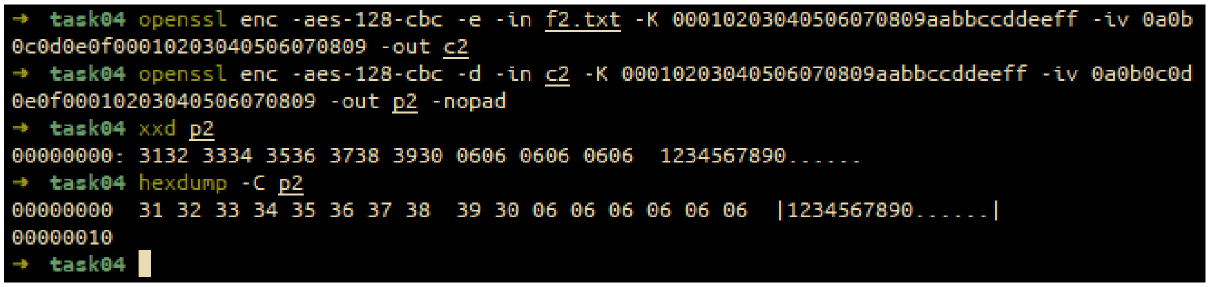
```
➜  task04 openssl enc -aes-128-cbc -e -in f3.txt -K 000102030405060708090aabbccddeeff -iv 0a0b
0c0d0e0f000102030405060708090 -out result
➜  task04 xxd result
00000000: 89d0 a989 5004 b11b 4032 80c7 a2ae 0ab2  ....P...@2......
00000010: e6bb 2c45 fdc2 de24 95b8 77ce 48ed d497  ..,E...$..w.H...
➜  task04 ls -l result
-rwxrwxrwx 1 edo edo 32 Oct  1 13:12 result
➜  task04
```

Figure 34: `f3.txt` (16 bytes) encrypted with AES-128 and CBC.



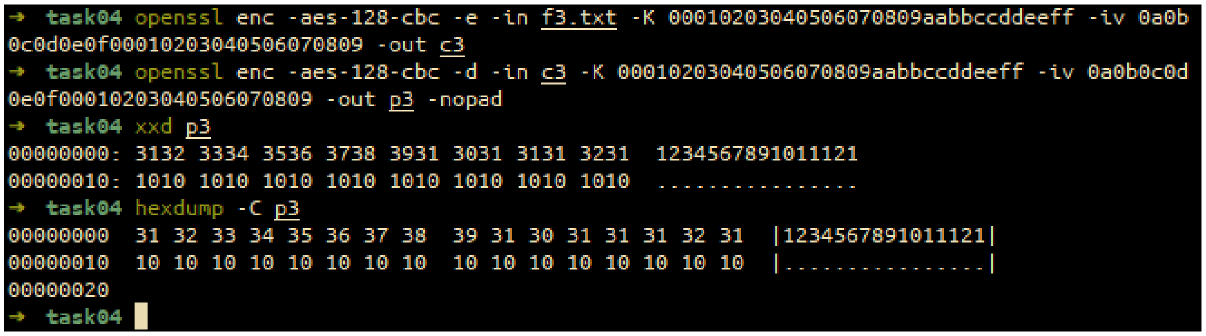
```
➜  task04 openssl enc -aes-128-cbc -e -in f1.txt -K 000102030405060708090aabbccddeeff -iv 0a0b
0c0d0e0f000102030405060708090 -out c1
➜  task04 openssl enc -aes-128-cbc -d -in c1 -K 000102030405060708090aabbccddeeff -iv 0a0b0c0d
0e0f000102030405060708090 -out p1 -nopad
➜  task04 xxd p1
00000000: 3132 3334 350b 0b0b 0b0b 0b0b 0b0b 0b0b  12345...........
➜  task04 hexdump -C p1
00000000  31 32 33 34 35 0b 0b 0b  0b 0b 0b 0b 0b 0b 0b 0b  |12345...........|
00000010
➜  task04
```

Figure 35: `f1.txt` ciphertext decrypted with AES-128 and CBC.



```
➜  task04 openssl enc -aes-128-cbc -e -in f2.txt -K 000102030405060708090aabbccddeeff -iv 0a0b
0c0d0e0f000102030405060708090 -out c2
➜  task04 openssl enc -aes-128-cbc -d -in c2 -K 000102030405060708090aabbccddeeff -iv 0a0b0c0d
0e0f000102030405060708090 -out p2 -nopad
➜  task04 xxd p2
00000000: 3132 3334 3536 3738 3930 0606 0606 0606  1234567890......
➜  task04 hexdump -C p2
00000000  31 32 33 34 35 36 37 38  39 30 06 06 06 06 06 06  |1234567890......|
00000010
➜  task04
```

Figure 36: `f2.txt` ciphertext decrypted with AES-128 and CBC.



```
➜  task04 openssl enc -aes-128-cbc -e -in f3.txt -K 000102030405060708090aabbccddeeff -iv 0a0b
0c0d0e0f000102030405060708090 -out c3
➜  task04 openssl enc -aes-128-cbc -d -in c3 -K 000102030405060708090aabbccddeeff -iv 0a0b0c0d
0e0f000102030405060708090 -out p3 -nopad
➜  task04 xxd p3
00000000: 3132 3334 3536 3738 3931 3031 3131 3231  1234567891011121
00000010: 1010 1010 1010 1010 1010 1010 1010 1010  ................
➜  task04 hexdump -C p3
00000000  31 32 33 34 35 36 37 38  39 31 30 31 31 31 32 31  |1234567891011121|
00000010  10 10 10 10 10 10 10 10  10 10 10 10 10 10 10 10  |................|
00000020
➜  task04
```

Figure 37: `f3.txt` ciphertext decrypted with AES-128 and CBC.

Hex `10` (which is 16 in decimal) is added to the padding during encryption of `f3.txt` because `f3.txt` block is 16 bytes which itself can be a block of message. If we keep the plaintext without the padding like that, there will not be randomness in ciphertext and every plaintext that is like the one in `f3.txt` will result the same ciphertext after encryption. So we have to add another 16 bytes (128 bits) to the padding during the encryption.

2.5 Task 5: Error Propagation - Corrupted Cipher Text

ECB should be the cipher mode that has the most information recovered by decrypting the corrupted file, only the block ciphertext with the corrupted byte will not be fully recovered. This is due to the fact that each block of message is encrypted seperately and merged together at the end in ECB mode. So the corrupted block will not be able to affect other blocks.

CBC, CFB, and OFB should have the same amount of data recovered from the corrupted ciphertext, which is much less than the amount of data recovered from ECB mode, since in each step, it uses the ciphertext or plaintext of the previous step for encryption, so all other blocks after that step will be affected as well.

The plaintext used in this task is as follow:

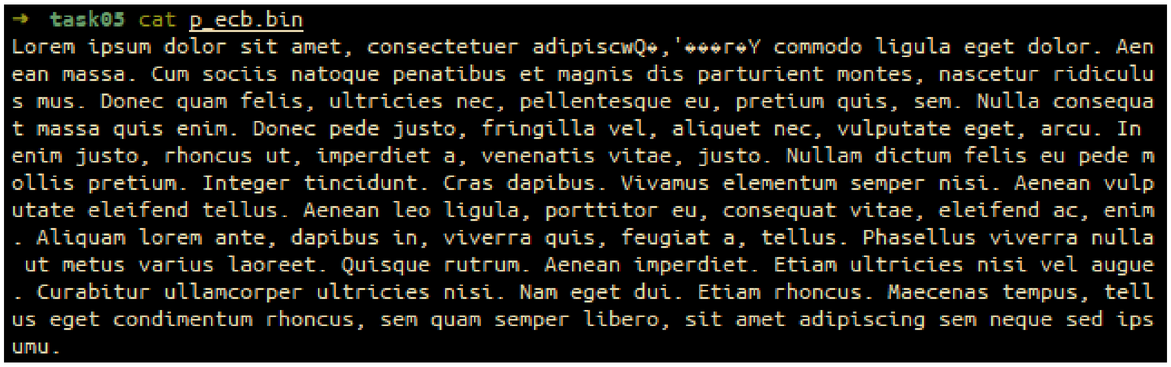
Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. In enim justo, rhoncus ut, imperdiet a, venenatis vitae, justo. Nullam dictum felis eu pede mollis pretium. Integer tincidunt. Cras dapibus. Vivamus elementum semper nisi. Aenean vulputate eleifend tellus. Aenean leo ligula, porttitor eu, consequat vitae, eleifend ac, enim. Aliquam lorem ante, dapibus in, viverra quis, feugiat a, tellus. Phasellus viverra nulla ut metus varius laoreet. Quisque rutrum. Aenean imperdiet. Etiam ultricies nisi vel augue. Curabitur ullamcorper ultricies nisi. Nam eget dui. Etiam rhoncus. Maecenas tempus, tellus eget condimentum rhoncus, sem quam semper libero, sit amet adipiscing sem neque sed ipsumu.

In this task, I wrote a Python program to automatically flip a bit (I chose the least significant bit) in the 55th byte of the ciphertext because I do not have Bless software.

```python
import sys

from subprocess import check_output


# corrupt one bit in given position of 55th byte of ciphertext
def corrupt(ciphertext: bytes, position: int) -> bytes:

    # split the ciphertext into list of decimal numbers converted from the bytes
    # of the ciphertext
    ciphertext_bytes = list(ciphertext)

    # get the 55th byte
    target_byte = ciphertext_bytes[54]

    # flip the bit at the given position
    new_byte = target_byte ^ (1 << position)

    # change the 55th byte in the ciphertext to the new byte
    ciphertext_bytes[54] = new_byte

    return bytes(ciphertext_bytes)


if __name__ == "__main__":

    if len(sys.argv) == 2:
        try:
            filename = sys.argv[1]

            # read bytes of ciphertext from given filename
            ciphertext = check_output(['cat', filename])

            # corrupt the first bit of 55th byte in the ciphertext
            corrupted_ciphertext_bytes = corrupt(ciphertext, 0)

            # write the result bytes into a file
            f_corrupt = open(f'corrupted_{filename}', 'wb')
            f_corrupt.write(corrupted_ciphertext_bytes)
            f_corrupt.close()

        except FileNotFoundError:
            print("Cannot open file.")
    else:
        print("Please provide a filename.")
```

And because I was being lazy, I also wrote a bashscript for encrypting the plaintext, corrupting the ciphertext, and decrypting the corrupted ciphertext.

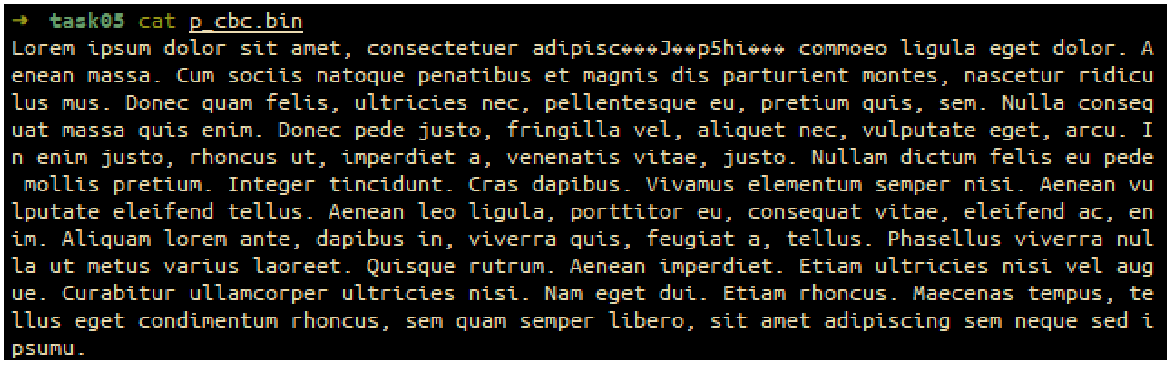
```
#!/bin/bash

openssl enc -aes-128-ecb -e -in plaintext.txt -out ecb.bin \
    -K 000102030405060708090aabbccddeeff
openssl enc -aes-128-cbc -e -in plaintext.txt -out cbc.bin \
    -K 000102030405060708090aabbccddeeff \
    -iv 0a0b0c0d0e0f00010203040506070809
openssl enc -aes-128-cfb -e -in plaintext.txt -out cfb.bin \
    -K 000102030405060708090aabbccddeeff \
    -iv 0a0b0c0d0e0f00010203040506070809
openssl enc -aes-128-ofb -e -in plaintext.txt -out ofb.bin \
    -K 000102030405060708090aabbccddeeff \
    -iv 0a0b0c0d0e0f00010203040506070809

python3 corrupt.py ecb.bin
python3 corrupt.py cbc.bin
python3 corrupt.py cfb.bin
python3 corrupt.py ofb.bin

openssl enc -aes-128-ecb -d -in corrupted_ecb.bin -out p_ecb.bin \
    -K 000102030405060708090aabbccddeeff
openssl enc -aes-128-cbc -d -in corrupted_cbc.bin -out p_cbc.bin \
    -K 000102030405060708090aabbccddeeff \
    -iv 0a0b0c0d0e0f00010203040506070809
openssl enc -aes-128-cfb -d -in corrupted_cfb.bin -out p_cfb.bin \
    -K 000102030405060708090aabbccddeeff \
    -iv 0a0b0c0d0e0f00010203040506070809
openssl enc -aes-128-ofb -d -in corrupted_ofb.bin -out p_ofb.bin \
    -K 000102030405060708090aabbccddeeff \
    -iv 0a0b0c0d0e0f00010203040506070809
```



```
➜  task05 cat p_ecb.bin
Lorem ipsum dolor sit amet, consectetuer adipiscwQ◆,'◆◆◆r◆Y commodo ligula eget dolor. Aen
ean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculu
s mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequa
t massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. In
enim justo, rhoncus ut, imperdiet a, venenatis vitae, justo. Nullam dictum felis eu pede m
ollis pretium. Integer tincidunt. Cras dapibus. Vivamus elementum semper nisi. Aenean vulp
utate eleifend tellus. Aenean leo ligula, porttitor eu, consequat vitae, eleifend ac, enim
. Aliquam lorem ante, dapibus in, viverra quis, feugiat a, tellus. Phasellus viverra nulla
 ut metus varius laoreet. Quisque rutrum. Aenean imperdiet. Etiam ultricies nisi vel augue
. Curabitur ullamcorper ultricies nisi. Nam eget dui. Etiam rhoncus. Maecenas tempus, tell
us eget condimentum rhoncus, sem quam semper libero, sit amet adipiscing sem neque sed ips
umu.
```

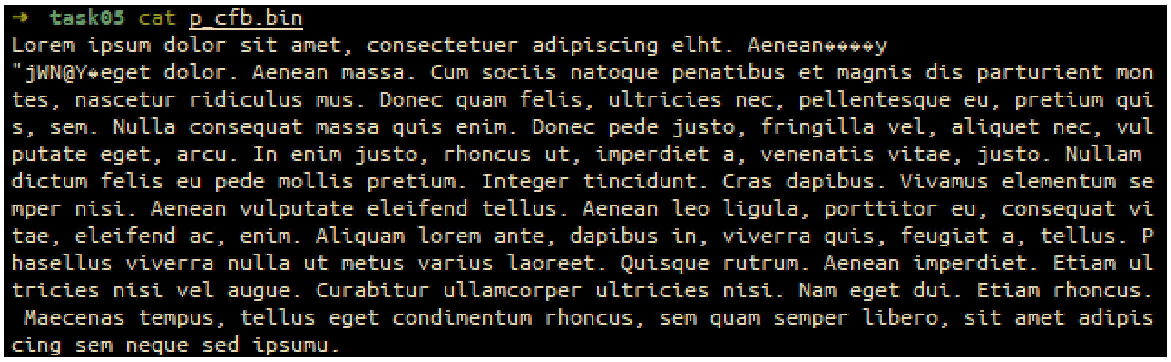
Figure 38: Plaintext decrypted from corrupted ciphertext using AES-128 and ECB.

The plaintext of the corrupted ciphertext decrypt using ECB, CBC, CFB, and OFB mode are shown in Figure 38, 39, 40, and 41.



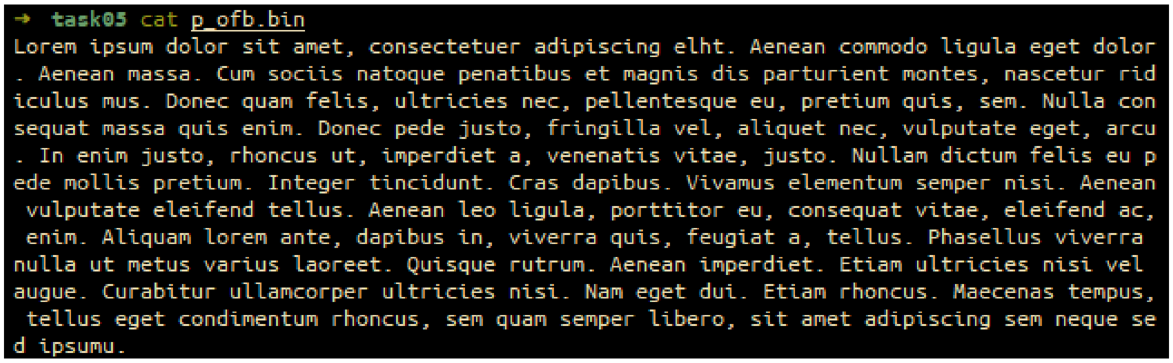
```
➜  task05 cat p_cbc.bin
Lorem ipsum dolor sit amet, consectetuer adipisc���J��p5hi��� commoeo ligula eget dolor. A
enean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridicu
lus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla conseq
uat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu. I
n enim justo, rhoncus ut, imperdiet a, venenatis vitae, justo. Nullam dictum felis eu pede
 mollis pretium. Integer tincidunt. Cras dapibus. Vivamus elementum semper nisi. Aenean vu
lputate eleifend tellus. Aenean leo ligula, porttitor eu, consequat vitae, eleifend ac, en
im. Aliquam lorem ante, dapibus in, viverra quis, feugiat a, tellus. Phasellus viverra nul
la ut metus varius laoreet. Quisque rutrum. Aenean imperdiet. Etiam ultricies nisi vel aug
ue. Curabitur ullamcorper ultricies nisi. Nam eget dui. Etiam rhoncus. Maecenas tempus, te
llus eget condimentum rhoncus, sem quam semper libero, sit amet adipiscing sem neque sed i
psumu.
```

Figure 39: Plaintext decrypted from corrupted ciphertext using AES-128 and CBC.



```
➜  task05 cat p_cfb.bin
Lorem ipsum dolor sit amet, consectetuer adipiscing elht. Aenean����y
"jWN@Y�eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient mon
tes, nascetur ridiculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium qui
s, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vul
putate eget, arcu. In enim justo, rhoncus ut, imperdiet a, venenatis vitae, justo. Nullam
dictum felis eu pede mollis pretium. Integer tincidunt. Cras dapibus. Vivamus elementum se
mper nisi. Aenean vulputate eleifend tellus. Aenean leo ligula, porttitor eu, consequat vi
tae, eleifend ac, enim. Aliquam lorem ante, dapibus in, viverra quis, feugiat a, tellus. P
hasellus viverra nulla ut metus varius laoreet. Quisque rutrum. Aenean imperdiet. Etiam ul
tricies nisi vel augue. Curabitur ullamcorper ultricies nisi. Nam eget dui. Etiam rhoncus.
 Maecenas tempus, tellus eget condimentum rhoncus, sem quam semper libero, sit amet adipis
cing sem neque sed ipsumu.
```

Figure 40: Plaintext decrypted from corrupted ciphertext using AES-128 and CFB.



```
➜  task05 cat p_ofb.bin
Lorem ipsum dolor sit amet, consectetuer adipiscing elht. Aenean commodo ligula eget dolor
. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur rid
iculus mus. Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla con
sequat massa quis enim. Donec pede justo, fringilla vel, aliquet nec, vulputate eget, arcu
. In enim justo, rhoncus ut, imperdiet a, venenatis vitae, justo. Nullam dictum felis eu p
ede mollis pretium. Integer tincidunt. Cras dapibus. Vivamus elementum semper nisi. Aenean
 vulputate eleifend tellus. Aenean leo ligula, porttitor eu, consequat vitae, eleifend ac,
 enim. Aliquam lorem ante, dapibus in, viverra quis, feugiat a, tellus. Phasellus viverra
nulla ut metus varius laoreet. Quisque rutrum. Aenean imperdiet. Etiam ultricies nisi vel
augue. Curabitur ullamcorper ultricies nisi. Nam eget dui. Etiam rhoncus. Maecenas tempus,
 tellus eget condimentum rhoncus, sem quam semper libero, sit amet adipiscing sem neque se
d ipsumu.
```

Figure 41: Plaintext decrypted from corrupted ciphertext using AES-128 and OFB.

It seems that my previous answer is wrong, the amount of information can you recover by decrypting the corrupted file by cipher mode in decreasing order is OFB (only has one character different from plaintext), CBC (loses two words), ECB (loses two words), CFB (loses two words and has one character different).

2.6 Task 6: Inital Vector (IV)

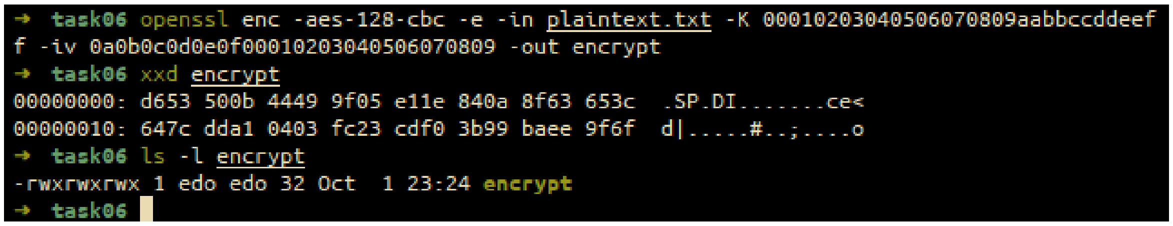
2.6.1 Task 6.1. Uniqueness of the IV

The plaintext is `"This is a secret tool"`.

The **key** is `00010203040506070809aabbccddeef`.

The first **initial vector** is `0a0b0c0d0e0f00010203040506070809`.

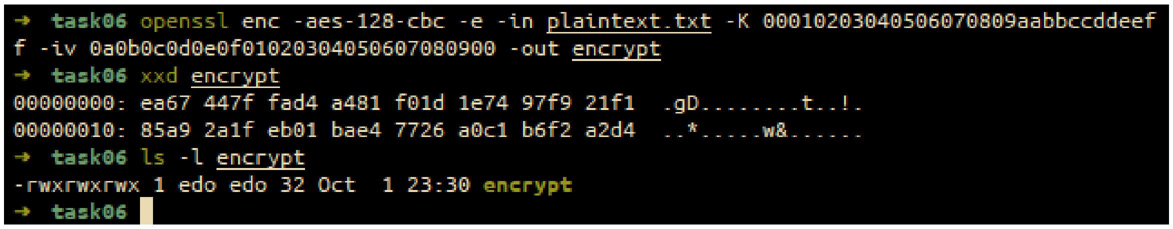
The second **initial vector** is `0a0b0c0d0e0f01020304050607080900`.



```
→ task06 openssl enc -aes-128-cbc -e -in plaintext.txt -K 00010203040506070809aabbccddeef
f -iv 0a0b0c0d0e0f00010203040506070809 -out encrypt
→ task06 xxd encrypt
00000000: d653 500b 4449 9f05 e11e 840a 8f63 653c  .SP.DI.......ce<
00000010: 647c dda1 0403 fc23 cdf0 3b99 baee 9f6f  d|.....#..;....o
→ task06 ls -l encrypt
-rwxrwxrwx 1 edo edo 32 Oct  1 23:24 encrypt
→ task06
```

Figure 42: Encrypted plaintext with AES-128 and CBC with first initial vector.

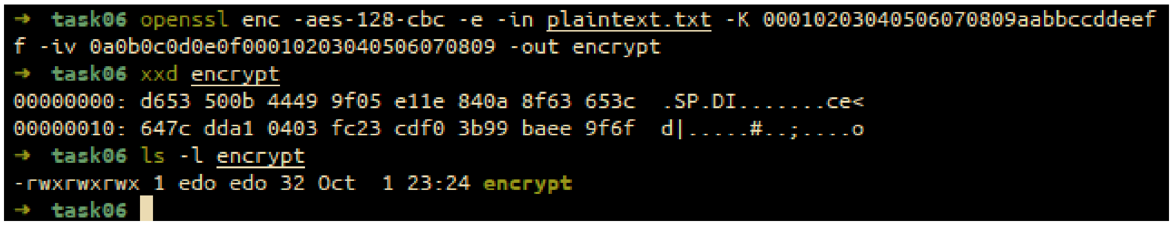
1. Figure 43 shows the result of encryption using the different IVs.



```
→ task06 openssl enc -aes-128-cbc -e -in plaintext.txt -K 00010203040506070809aabbccddeef
f -iv 0a0b0c0d0e0f01020304050607080900 -out encrypt
→ task06 xxd encrypt
00000000: ea67 447f fad4 a481 f01d 1e74 97f9 21f1  .gD........t..!.
00000010: 85a9 2a1f eb01 bae4 7726 a0c1 b6f2 a2d4  ..*.....w&......
→ task06 ls -l encrypt
-rwxrwxrwx 1 edo edo 32 Oct  1 23:30 encrypt
→ task06
```

Figure 43: Encrypted plaintext with AES-128 and CBC with different initial vector.

2. Figure 44 shows the result of encryption using the same IVs.



```
→ task06 openssl enc -aes-128-cbc -e -in plaintext.txt -K 00010203040506070809aabbccddeef
f -iv 0a0b0c0d0e0f00010203040506070809 -out encrypt
→ task06 xxd encrypt
00000000: d653 500b 4449 9f05 e11e 840a 8f63 653c  .SP.DI.......ce<
00000010: 647c dda1 0403 fc23 cdf0 3b99 baee 9f6f  d|.....#..;....o
→ task06 ls -l encrypt
-rwxrwxrwx 1 edo edo 32 Oct  1 23:24 encrypt
→ task06
```

Figure 44: Encrypted plaintext with AES-128 and CBC with same initial vector.

Using the same IVs to decrypt the message results the same ciphertext. Using different IVs would result different ciphertexts.

Thus, IVs should be different while encrypting the same message to add randomness to the ciphertext and makes it harder to guess the plaintext based on the ciphertext.

2.6.2 Task 6.2. Common Mistake: Use the Same IV

For OFB mode, If the key and IV keep unchanged, known-plaintext attack is feasible.

Output stream can be obtained by XORing plaintext and ciphertext block by block. Similarly, to get plaintext, I can XOR plaintext and ciphertext. When sharing the same key and IV for OFB mode, the output streams are identical among encryptions.

We know that `C1` = `P1 XOR IV`, then `IV` = `P1 XOR C1`. Thus, `C2` = `P2 XOR IV` tells us that `P2` = `C2 XOR IV` = `C2 XOR P1 XOR C1`.

```
from binascii import unhexlify

# initialize p1, c1, c2
p1 = "This is a known message!"
c1 = "a469b1c502c1cab966965e50425438e1bb1b5f9037a4c15913"
c2 = "bf73bcd3509299d566c35b5d450337e1bb175f903fafc15913"

# extract the given p1, c1, c2 to array of bytes
p1 = list(p1.encode('utf-8'))
c1 = list(unhexlify(c1))
c2 = list(unhexlify(c2))

# perform each byte of p1, c1, c2, decode the byte and join it to find p2
p2 = bytes([p1[i] ^ c1[i] ^ c2[i] for i in range(len(p1))]).decode('utf-8')

print(p2)
```

The result of `P2` is found to be `Order:  Launch a missle!`.

We can also see that same characters in `P1` and `P2` are encrypted the same. For example, the substring `e!` at the end of the sentence of `P1` and `P2` is encrypted the be `c15913`.

`CFB` would be attacked with the same method above and `P2` will be fully revealed.

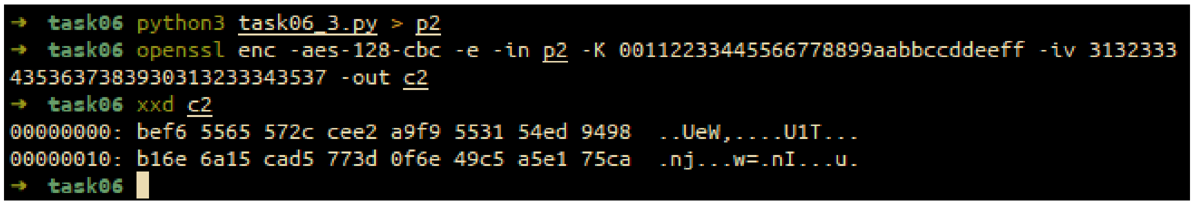
2.6.3 Task 6.3. Common Mistake: Use a Predictable IV

Let the original plaintext sent by Bob is `P1` and the the next plaintext is `P2`.

Construct `P2` = `P1 XOR IV XOR IV_next`.

```
from binascii import unhexlify

# guess that the original plaintext is "Yes"
p1 = "Yes"
iv = "31323334353637383930313233343536"
iv_next = "31323334353637383930313233343537"

# convert string to hexadecimal and split them into block of 1 byte
p1 = list(p1.encode('utf-8'))
iv = list(unhexlify(iv))
iv_next = list(unhexlify(iv_next))

# check if plaintext can be split into 128bit-block. If not, add padding to
# the plaintext before encryption
padding = 16 - len(p1) % 16
p1 += [padding] * padding

# calculate p2
p2 = bytes([p1[x] ^ iv[x] ^ iv_next[x] for x in range(len(p1))]).decode("utf-8")
print(p2)
```

We then check if the ciphertext of P2 is the same as the ciphertext of P1.



```
→ task06 python3 task06_3.py > p2
→ task06 openssl enc -aes-128-cbc -e -in p2 -K 00112233445566778899aabbccddeeff -iv 3132333
4353637383930313233343537 -out c2
→ task06 xxd c2
00000000: bef6 5565 572c cee2 a9f9 5531 54ed 9498  ..UeW,....U1T...
00000010: b16e 6a15 cad5 773d 0f6e 49c5 a5e1 75ca  .nj...w=.nI...u.
→ task06
```

Figure 45: Check ciphertext generated from P2.

We can see that the first few bytes of C2 (ciphertext of P2) are the same as the ones in C1 (ciphertext of P1). Therefore, the original message P1 is "Yes".

2.7 Task 7: Programming using the Crypto Library

For this task, because there is not any implementation of openssl encrypt/decrypt library for Python, I used pycrypto [3] and pycryptodome [4] instead.

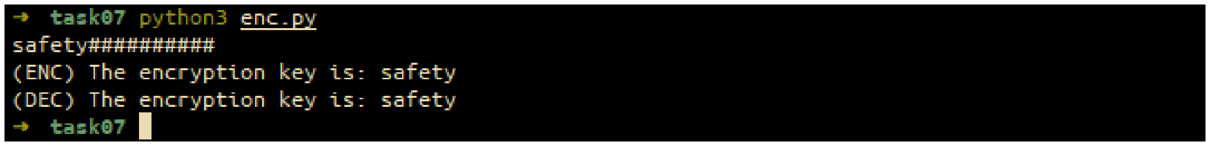
```python
from subprocess import check_output

from binascii import unhexlify

from Crypto.Cipher import AES
from Crypto.Util.Padding import pad
from Crypto.Util.Padding import unpad

# read english words from file and store them in an arraylist
f = open("english_word_list.txt")
english_words = f.readlines()
english_words = [x.strip("\n") for x in english_words]
f.close()

# add padding to words so their size is 128 bits
english_words_padding = []

for w in english_words:
    wb = w.encode('utf-8')
    if len(wb)*8 < 128:
        english_words_padding.append(w + "#"*(128//8 - len(wb)))

# to make sure it works right, loop through the list and test if the words are
# 128 bits
for w in english_words_padding:
    if len(w.encode('utf-8'))*8 != 128:
        raise Exception("Key size is too small (less than 128 bits)!!!")

# initialize the plaintext, ciphertext, and initial vector
plaintext = "This is a secret tool"
iv = "0102030405060708090000a0b0c0d0e0f"
ciphertext = "ece6753e938f8f903cabbbe12d395bf5f7eae38ad918a2d3e1c3a832476d5c7a"


# in this function, I used openssl to get the key for to compare with the result
# of encryption without using openssl
def enc_openssl(plaintext: str, key: str, iv: str) -> bytes:

    encrypted = check_output(['echo', '-n', plaintext, '|', 'openssl', 'enc',
                              '-aes-128-cbc', '-e', '-K', key, '-iv', iv])
```

[3]pycrypto: https://www.dlitz.net/software/pycrypto/api/current/
[4]pycryptodome: https://pycryptodome.readthedocs.io/en/latest/src/introduction.html

```
    return encrypted


# encrypt message using aes-128-cbc with pycrypto and pycryptodome library
def enc(plaintext: str, key: str, iv: str) -> bytes:

    key = unhexlify(key)
    iv = unhexlify(iv)

    # initialize the cipher type and mode
    cipher = AES.new(key, AES.MODE_CBC, iv)

    # encrypt the message using the cipher type and mode declared above.
    # also add padding to the plaintext in case the size of blocks of message
    # from plaintext is smaller than 128 bits.
    encrypted = cipher.encrypt(pad(plaintext.encode('utf-8'), AES.block_size))

    return encrypted


# decrypt ciphertext using aes-128-cbc with pycrypto and pycryptodome library
def dec(ciphertext: str, key: str, iv: str) -> bytes:

    ciphertext = unhexlify(ciphertext)
    key = unhexlify(key)
    iv = unhexlify(iv)

    # initialize the cipher type and mode
    decipher = AES.new(key, AES.MODE_CBC, iv)

    # decrypt the ciphertext using the cipher type and mode declared above.
    decrypted = decipher.decrypt(ciphertext)

    # check if the decrypted message has padding. If yes, remove the padding and
    # return the unpadded plaintext. Else, return the decrypted message.
    try:
        decrypted = unpad(decrypted, AES.block_size)
    except Exception:
        pass

    return decrypted


if __name__ == "__main__":

    # bruteforce: go through every possible key and encrypt the plaintext.
    # compare the result with the given ciphertext.
    for k in english_words_padding:
        key = k.encode('utf-8').hex()
        encrypted = enc(plaintext, key, iv)

        if encrypted.hex() == ciphertext:
            print(k)
            print("(ENC) The encryption key is:", k.strip("#"))
            break

    # bruteforce: go through every possible key and decrypt the ciphertext.
    # compare the result with the given plaintext.
    for k in english_words_padding:
        key = k.encode('utf-8').hex()
        decrypted = dec(ciphertext, key, iv)

        if decrypted == plaintext.encode('utf-8'):
            print("(DEC) The encryption key is:", k.strip("#"))
            break
```

I used two methods to find the encryption key: go through every key to encrypt the plaintext and compare the result with the given ciphertext, go through every key to decrypt the ciphertext and compare the result with the given plaintext. More explanation can be found in the comments in the code.

Both methods result the same key: `safety##########` with padding and `safety` without padding.



```
→ task07 python3 enc.py
safety##########
(ENC) The encryption key is: safety
(DEC) The encryption key is: safety
→ task07
```

Figure 46: Result from running the program above.

The result from using `openssl` is also `safety##########` with padding and `safety` without padding.

Therefore, I can conclude that the encryption key for this cipher is `safety`.

This method only works if we already know a fixed set of keys that used to encrypt the message. In real life, the set of possible keys is large and the key are combination of words and special characters, sometimes the keys do not have to have meaning, they can be total random. This would make it impossible to bruteforce AES.