Homework 3 Report

Paper and Pencil Problem: (all questions are from the KPS book)

(2.2)

Question: Random J. Protocol-Designer has been told to design a scheme to prevent messages from being modified by an intruder. Random J. decides to append to each message a hash of that message. Why doesn't this solve the problem? (We know of a protocol that uses this technique in an attempt to gain security.)

Answer: Hash function is well-known. The bad guy can try all the hash functions one by one on the message and compare the hashed result with the hash value appended to the original message. If they are the same, the bad guy will know the hash function. He can modify the original message, hash it with the hash function, append it to the modified message and send to receiver.

(2.3)

Question: Suppose Alice, Bob, and Carol want to use secret key technology to authenticate each other. If they all used the same secret key K, then Bob could impersonate Carol to Alice (actually any of the three can impersonate the other to the third). Suppose instead that each had their own secret key, so Alice uses K_A, Bob uses K_B, and Carol uses K_C. This means that each one, to prove his or her identity, responds to a challenge with a function of his or her secret key and the challenge. Is this more secure than having them all use the same secret key K? (Hint: what does Alice need to know in order to verify Carol's answer to Alice's challenge?)

Answer: I don't think this is more secure than having them all use the same secret key K. In the Challenge-Response scheme, Alice has to know Carol's key K_C to decrypt the response sent back from Carol(The response is encrypted using Carol's secret key K_C). Similarly, Bob will also have Carol's key K_C in order to successfully authenticate Carol in Challenge-Response scheme. In this situation, Bob can impersonate Carol to Alice by asking a challenge to Alice, encrypting the challenge with Carol's key K_C and sending back to Alice. Alice has no chance to identify whether the response is from Bob or Carol.

(2.4)

Question: As described in §2.6.4 Downline Load Security, it is common, for performance reasons, to sign a message digest of a message rather than the message itself. Why is it so important that it be difficult to find two messages with the same message digest?

Answer: This is an important property for a good hash function. If it is easy to find two messages with the same message digest, it is more likely that, after the bad guy modified the original message, the faked message and the original message will still have the same message digest. The receiver will not detect the integrity attack.

(3.2)

Question: Token cards display a number that changes periodically, perhaps every minute. Each such device has a unique secret key. A human can prove possession of a particular such device by entering the displayed number into a computer system. The computer system knows the secret keys of each authorized device. How would you design such a device?

Answer: With the principle of CIA, I decide to design such a device as follows,

(1) Each device keeps a table, which contains a list of all User IDs that can possess the device.
(2) The input of such a device should be:

User ID	temp Num	Hash of User ID and temp Num

encrypted by the key of the device in the computer system.

(3) The input will be decrypted by target device using its key. The device will also check whether the Num is a currently valid Num and whether the hash of (User ID + Num) is the same as the received hash.
(4) If the third step passed, the device will check the UID in its stored list. If a match is found, the device will allow the user to use it.

Such a system will have the following pros:

(1) Even if the encrypted input to a device is intercepted and captured, it will not be valid any more in a short period of time.
(2) Centralized key management. Users do not need to know the device key.
(3) Users do not need to reveal their User IDs. All they need to do is to obtain a number from device, specify the device toward computer system and input the encrypted info to desired device.

(3.3)

Question: How many DES keys, on the average, encrypt a particular plaintext block to a particular ciphertext block?

Answer: For a given 56-bit key, the probability of mapping plaintext b to ciphertext c is $\frac{1}{2^{64}}$

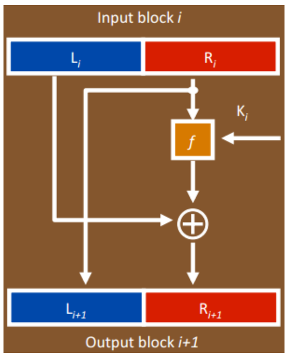
There are 2^{56} possible keys, so the probability of mapping plaintext b to ciphertext c is

$\frac{1}{2^{64}} \times 2^{56} = \frac{1}{256}$

(3.5)

Question: Suppose the DES mangler function mapped every 32-bit value to zero, regardless of the value of its input. What function would DES then compute?

Answer: A normal DES process is initial permutation, 16 DES rounds, swap halves and final permutation.



This is one DES round. If the mangler function f maps every 32-bit input(Ri) to zero, the XOR operation between Li and the result of f will be the same with Li. In other word, each DES round just swaps Li and Ri. Since 16 is an even number, after 16 DES rounds, the initial 64-bit word would be unchanged.

So DES would do the following:

- Initial permutation
- Swap left and right halves
- final permutation

If the swap were not there, DES would have no affect at all.

(4.2)

Question: The pseudo-random stream of blocks generated by 64-bit OFB must eventually repeat (since at most 264 different blocks can be generated). Will K{IV} necessarily be the first block to be repeated?

Answer: I think the answer is yes. Suppose b1 is K{IV}, b2 is K{K{IV}}, bi is the i-fold encryption of IV. bi+1 is the encryption of bi with key K. bi is the decryption of bi+1.Let bi be the first repeat element and bj = bi where j < i.

If j = 1, then the first repeat block is b1(K{IV}). We are done.

If j > 1, then bj-1 = bi-1. So bi is not the first repeat block. Contradiction.

Therefore, the first block b1(K{IV}) must be the first repeat block.

(4.4)

Question: What is a practical method for finding a triple of keys that maps a given plaintext to a given ciphertext using EDE? Hint: It is like the meet-in-the-middle attack of §4.4.1.2 Encrypting Twice with Two Keys.

Answer: For example, if I want to map plaintext p to ciphertext c with triple of keys(k1,k2,k3), the following two relationship needs to be satisfied.

c = Ek1(Dk2(Ek3(p)) and Dk1(c)= Dk2(Ek3(p))

A practical way to find the triple is

(1) Decrypt ciphertext c with a random key k_1 and store the result in a temporary table A.
(2) Encrypt plaintext p with random key k_3, and store the result in another table B(2^{56} records in table B).
(3) For each result in table B, decrypt it with a random key k_2 and store the results in table C(2^{112} records in C).
(4) If table C contains the result in table A, we find the triple of keys(k_1,k_2,k_3)

(4.6)

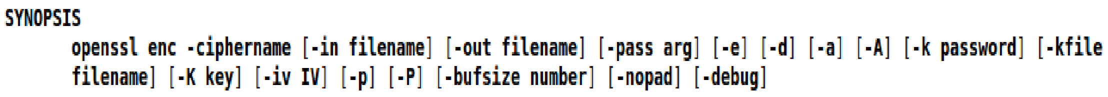
Question: Consider the following alternative method of encrypting a message. To encrypt a message, use the algorithm for doing a CBC decrypt. To decrypt a message, use the algorithm for doing a CBC encrypt. Would this work? What are the security implications of this, if any, as contrasted with the "normal" CBC?

Answer: It would work. However, there will be a security risk with this new scheme. In the new scheme, we can achieve parallel encryption and sequential decryption, just opposite to normal CBC. If the original message blocks are the same, all the ciphertext blocks will be the same except the first ciphertext block because of the XOR operation with IV. Therefore, there will be information leakage in the new scheme.

Besides, if there is one error in certain ciphertext block, all subsequent plaintext will be influenced. The good thing is the elimination of profitably ciphertext manipulation.

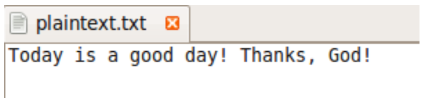
Lab and Programming Tasks:

Task 1: Encryption using different ciphers and modes:



```
SYNOPSIS
       openssl enc -ciphername [-in filename] [-out filename] [-pass arg] [-e] [-d] [-a] [-A] [-k password] [-kfile
       filename] [-K key] [-iv IV] [-p] [-P] [-bufsize number] [-nopad] [-debug]
```

Figure 1 openssl synopsis

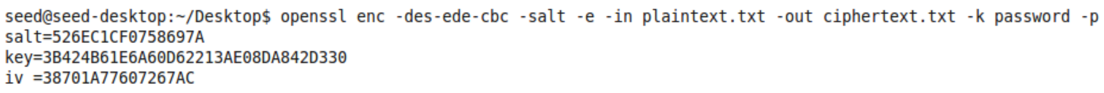


```
Today is a good day! Thanks, God!
```

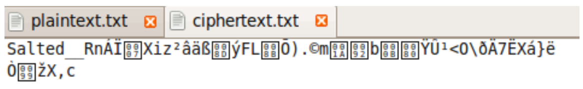
Figure 2 plaintext.txt

I tried the following cipher types:

(1) des-ede-cbc



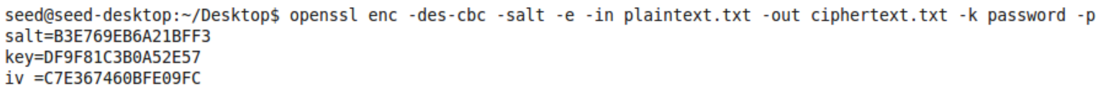
```
seed@seed-desktop:~/Desktop$ openssl enc -des-ede-cbc -salt -e -in plaintext.txt -out ciphertext.txt -k password -p
salt=526EC1CF0758697A
key=3B424B61E6A60D62213AE08DA842D330
iv =38701A77607267AC
```



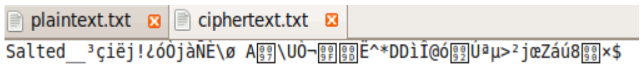
```
Salted__RnÀÏXiz²ââßýFLÕ).©mbŸÙ¹<O\ðÄ7ËXá}ë
ÒžX,c
```

Figure 3 ciphertext in Triple DES CBC mode

(2) des-cbc



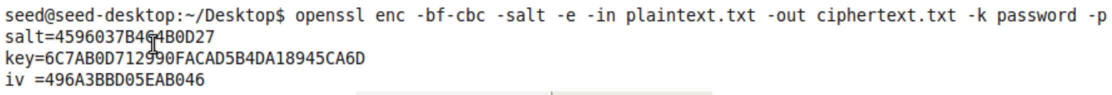
```
seed@seed-desktop:~/Desktop$ openssl enc -des-cbc -salt -e -in plaintext.txt -out ciphertext.txt -k password -p
salt=B3E769EB6A21BFF3
key=DF9F81C3B0A52E57
iv =C7E367460BFE09FC
```



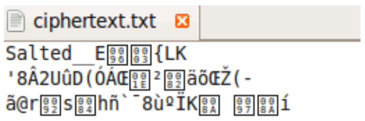
```
Salted__³çiëj!¿óÖjàÑÉ\ø A\UÓ¬Ë^*DDìÏ@óÚªµ>²jœZáü8×$
```

Figure 4 ciphertext in DES CBC mode

(3) bf-cbc



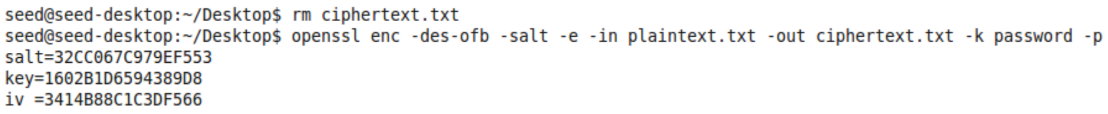
```
seed@seed-desktop:~/Desktop$ openssl enc -bf-cbc -salt -e -in plaintext.txt -out ciphertext.txt -k password -p
salt=4596037B464B0D27
key=6C7AB0D712990FACAD5B4DA18945CA6D
iv =496A3BBD05EAB046
```



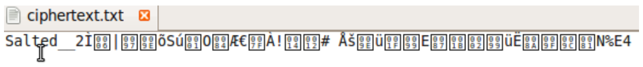
```
Salted__E{LK
'8Â2UûD(ÓÁŒ²äõŒŽ(-
ã@rshñ`¯8ùºÏKí
```

Figure 5 ciphertext in blowfish CBC mode

(4) des-ofb



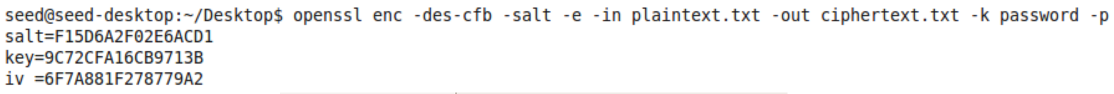
```
seed@seed-desktop:~/Desktop$ rm ciphertext.txt
seed@seed-desktop:~/Desktop$ openssl enc -des-ofb -salt -e -in plaintext.txt -out ciphertext.txt -k password -p
salt=32CC067C979EF553
key=1602B1D6594389D8
iv =3414B88C1C3DF566
```



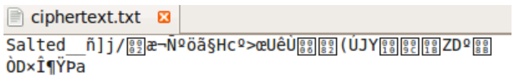
```
Salted__2Ì□|□□õSú□O□Æ€□À!□□# Åš□ü□□E□□□□üË□□□□N%E4
```

Figure 5 ciphertext in DES OFB mode

(5) des-cfb



```
seed@seed-desktop:~/Desktop$ openssl enc -des-cfb -salt -e -in plaintext.txt -out ciphertext.txt -k password -p
salt=F15D6A2F02E6ACD1
key=9C72CFA16CB9713B
iv =6F7A881F278779A2
```



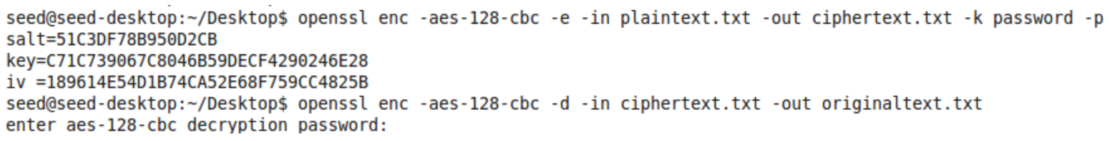
```
Salted__ñ]j/□æ¬Ñºöã§Hcº>œUêÙ□□(ÚJY□□□ZDº□
ÒD×Î¶ŸPa
```

Figure 6 ciphertext in DES CFB mode

My observation:

(1) In DES, the key size is 64 bits. In triple DES(DES-EDE in the first example), the key size is 128 bits.

(2) In all the examples above, I used "-salt" in the openssl synopsis. It guarantees even if I used the same password, the generated key from the password will be different. To decrypt a ciphertext to see whether it is the same as plaintext, I changed the synopsis as follows,



```
seed@seed-desktop:~/Desktop$ openssl enc -aes-128-cbc -e -in plaintext.txt -out ciphertext.txt -k password -p
salt=51C3DF78B950D2CB
key=C71C739067C8046B59DECF4290246E28
iv =189614E54D1B74CA52E68F759CC4825B
seed@seed-desktop:~/Desktop$ openssl enc -aes-128-cbc -d -in ciphertext.txt -out originaltext.txt
enter aes-128-cbc decryption password:
```

After inputing the same password, I found the content of originaltext.txt is the same with plaintext.txt. So the whole encryption and decryption process is successful.

Task 2: Encryption Mode – ECB vs. CBC:

Image Viewing Tool: Image Viewer

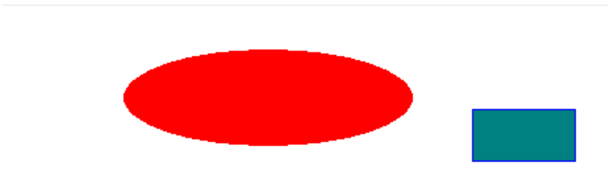


Figure 7 Original bmp file

To make an encrypted bmp file legitimate, I just follow the guidance in HW3 tutorial and replace the first 54 bytes(bmp header) of each encrypted bmp file with the first 54 bytes in the original bmp file. The results are as follows,

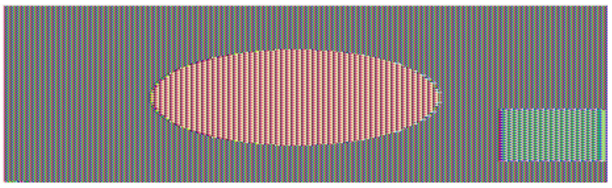


Figure 8 bmp file after encryption using ECB

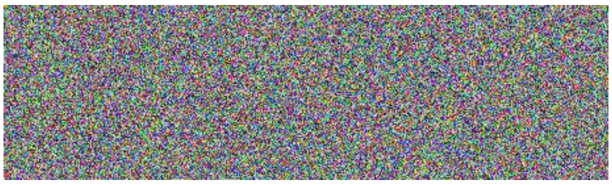
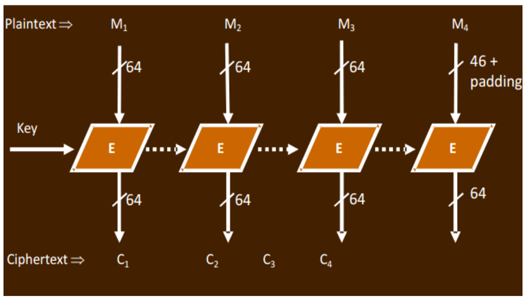


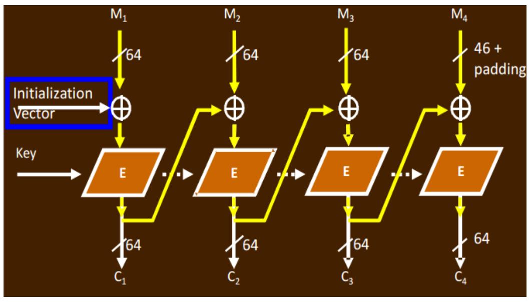
Figure 9 bmp file after encryption using CBC

After encryption with ECB mode, we can derive the basic information of the original picture. However, after encryption with CBC mode, we can hardly derive any information about the original picture.

To understand this difference, I will firstly give a brief introduction of BMP file format. BMP is also known for bitmap image file or device-independent bitmap (DIB) file format. In typical uncompressed bitmaps, image pixels are generally stored with a color depth of 1, 4, 8, 16, 24, 32, 48, or 64 bits per pixel.

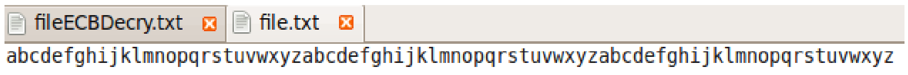
Normally, pixels near to each other are similar in appearance and stores similar binary information. For example, if M1 and M2 are blocks of bits for two pixels. If M1 and M2 are similar, C1 and C2 will be similar as well(ECB has info leakage). This explains why we can obtain basic information of the original picture in Figure 8. However, in CBC mode, info leakage is eliminated. Even if M1 and M2 are similar, C1 and C2 can be totally different. That's the reason why we cannot identify the original picture in Figure 9.





Task3: Encryption Mode – Corrupted Cipher Text:

Plaintext: 3 times traversal of alphabet (78 bytes).


fileECBDecry.txt | file.txt

```
abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz
```

Figure 10 plaintext---file.txt

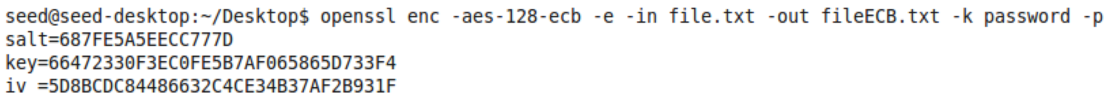
After AES-128 encryption, the encrypted file should be 16×5 = 80 bytes. There will be a 2-byte padding.

Assumption:

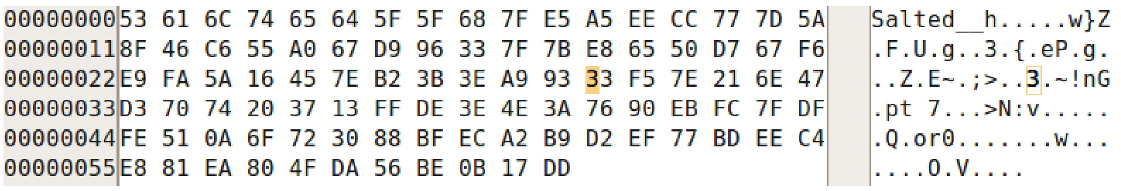
If a single bit of the 30th byte in the encrypted file got corrupted, that means an error occurs in the second block of ciphertext (17th byte – 32nd byte is in C2). Since there is no error propagation in ECB and OFB, only M2 (17th byte – 32nd byte in the plaintext) cannot be recovered after decryption. However, in CBC and CFB mode, both M2 and M3 cannot be recovered because of error propagation.

Reality:

ECB:



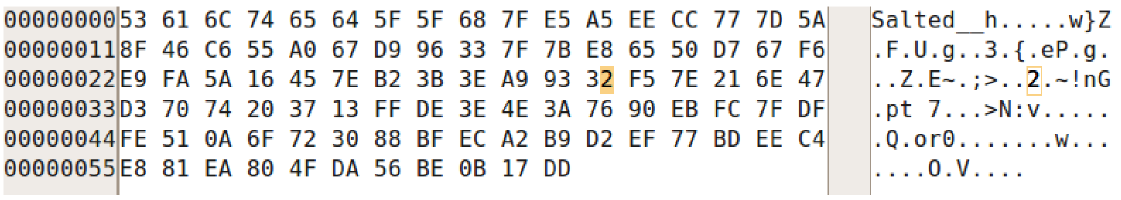
```
seed@seed-desktop:~/Desktop$ openssl enc -aes-128-ecb -e -in file.txt -out fileECB.txt -k password -p
salt=687FE5A5EECC777D
key=66472330F3EC0FE5B7AF065865D733F4
iv =5D8BCDC84486632C4CE34B37AF2B931F
```



```
00000000 53 61 6C 74 65 64 5F 5F 68 7F E5 A5 EE CC 77 7D 5A   Salted__h.....w}Z
00000011 8F 46 C6 55 A0 67 D9 96 33 7F 7B E8 65 50 D7 67 F6   .F.U.g..3.{.eP.g.
00000022 E9 FA 5A 16 45 7E B2 3B 3E A9 93 33 F5 7E 21 6E 47   ..Z.E~.;>..3.~!nG
00000033 D3 70 74 20 37 13 FF DE 3E 4E 3A 76 90 EB FC 7F DF   .pt 7...>N:v.....
00000044 FE 51 0A 6F 72 30 88 BF EC A2 B9 D2 EF 77 BD EE C4   .Q.or0.......w...
00000055 E8 81 EA 80 4F DA 56 BE 0B 17 DD                     ....O.V....
```

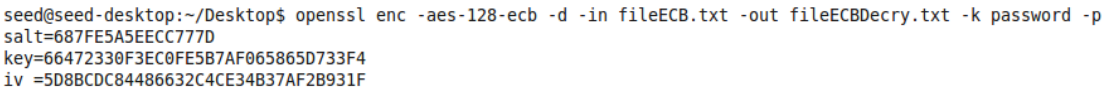
Figure 11 file.txt after encryption in ECB mode

The first 16 bytes are Salted. 0x5A is the first encryption byte. 0x33 is the 30th byte. It is 00110011. I changed the last bit from 1 to 0 to obtain a hex 0x32.



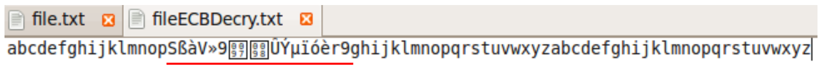
```
00000000 53 61 6C 74 65 64 5F 5F 68 7F E5 A5 EE CC 77 7D 5A   Salted__h.....w}Z
00000011 8F 46 C6 55 A0 67 D9 96 33 7F 7B E8 65 50 D7 67 F6   .F.U.g..3.{.eP.g.
00000022 E9 FA 5A 16 45 7E B2 3B 3E A9 93 32 F5 7E 21 6E 47   ..Z.E~.;>..2.~!nG
00000033 D3 70 74 20 37 13 FF DE 3E 4E 3A 76 90 EB FC 7F DF   .pt 7...>N:v.....
00000044 FE 51 0A 6F 72 30 88 BF EC A2 B9 D2 EF 77 BD EE C4   .Q.or0.......w...
00000055 E8 81 EA 80 4F DA 56 BE 0B 17 DD                     ....O.V....
```

Figure 12 one bit error



```
seed@seed-desktop:~/Desktop$ openssl enc -aes-128-ecb -d -in fileECB.txt -out fileECBDecry.txt -k password -p
salt=687FE5A5EECC777D
key=66472330F3EC0FE5B7AF065865D733F4
iv =5D8BCDC84486632C4CE34B37AF2B931F
```

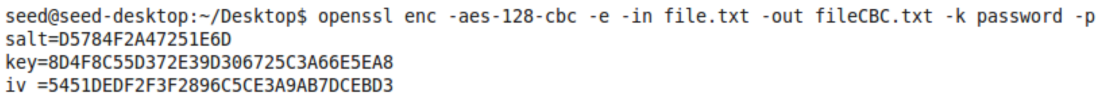
After decryption, we opened fileECBDecry.txt and found out, just like my assumption, the second block of the plaintext (17th byte – 32nd byte, q-f) cannot be obtained any more.



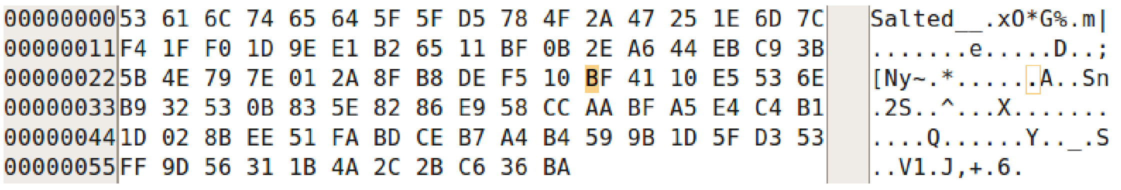
```
file.txt   fileECBDecry.txt
abcdefghijklmnopSßàV»9ÙÝµïóèr9ghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz
```

Figure 13 Decrypted text after corruption---fileECBDecry.txt

CBC:



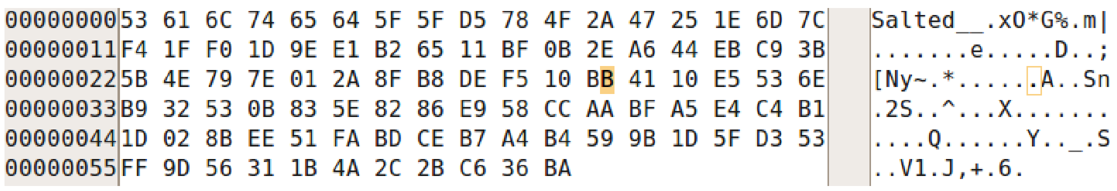
```
seed@seed-desktop:~/Desktop$ openssl enc -aes-128-cbc -e -in file.txt -out fileCBC.txt -k password -p
salt=D5784F2A47251E6D
key=8D4F8C55D372E39D306725C3A66E5EA8
iv =5451DEDF2F3F2896C5CE3A9AB7DCEBD3
```



```
00000000 53 61 6C 74 65 64 5F 5F D5 78 4F 2A 47 25 1E 6D 7C   Salted__.xO*G%.m|
00000011 F4 1F F0 1D 9E E1 B2 65 11 BF 0B 2E A6 44 EB C9 3B   .......e.....D..;
00000022 5B 4E 79 7E 01 2A 8F B8 DE F5 10 BF 41 10 E5 53 6E   [Ny~.*......A..Sn
00000033 B9 32 53 0B 83 5E 82 86 E9 58 CC AA BF A5 E4 C4 B1   .2S..^...X.......
00000044 1D 02 8B EE 51 FA BD CE B7 A4 B4 59 9B 1D 5F D3 53   ....Q......Y.._.S
00000055 FF 9D 56 31 1B 4A 2C 2B C6 36 BA                     ..V1.J,+.6.
```

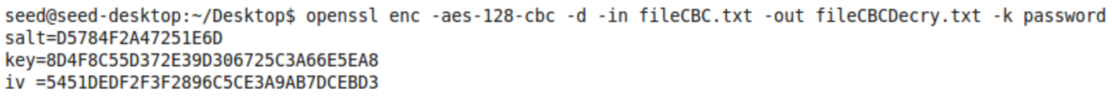
Figure 14 file.txt after encryption in CBC mode

Here, the 30th byte is 0xBF(10111111), I changed it to 0xBB(10111011)



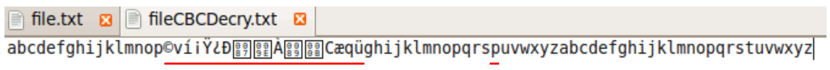
```
00000000 53 61 6C 74 65 64 5F 5F D5 78 4F 2A 47 25 1E 6D 7C   Salted__.xO*G%.m|
00000011 F4 1F F0 1D 9E E1 B2 65 11 BF 0B 2E A6 44 EB C9 3B   .......e.....D..;
00000022 5B 4E 79 7E 01 2A 8F B8 DE F5 10 BB 41 10 E5 53 6E   [Ny~.*......A..Sn
00000033 B9 32 53 0B 83 5E 82 86 E9 58 CC AA BF A5 E4 C4 B1   .2S..^...X.......
00000044 1D 02 8B EE 51 FA BD CE B7 A4 B4 59 9B 1D 5F D3 53   ....Q......Y.._.S
00000055 FF 9D 56 31 1B 4A 2C 2B C6 36 BA                     ..V1.J,+.6.
```

Figure 15 one bit error



```
seed@seed-desktop:~/Desktop$ openssl enc -aes-128-cbc -d -in fileCBC.txt -out fileCBCDecry.txt -k password
salt=D5784F2A47251E6D
key=8D4F8C55D372E39D306725C3A66E5EA8
iv =5451DEDF2F3F2896C5CE3A9AB7DCEBD3
```

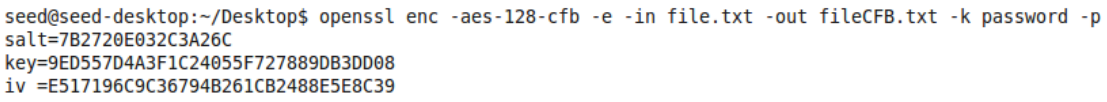
After decryption, we opened fileECBDecry.txt and found out that the second and third blocks of plaintext are influenced by the error. Since I only flipped one bit in C2, after XOR in decryption, it will influence one bit in M3, which changed ‘t’ to ‘p’.



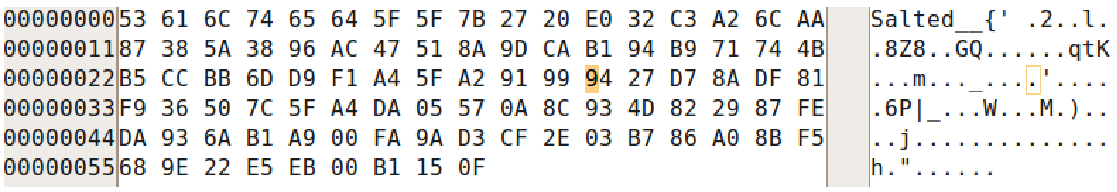
```
file.txt   fileCBCDecry.txt
abcdefghijklmnop©víiŸ¿Ð[87][9E]À[89][88]CæqüghijklmnopqrspuvwxyzabcdefghijklmnopqrstuvwxyZ
```

Figure 16 Decrypted text after corruption---fileCBCDecry.txt

CFB:



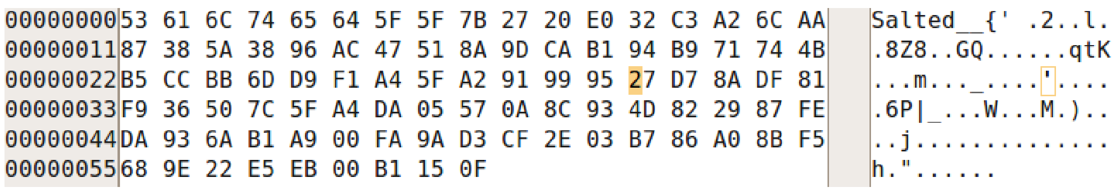
```
seed@seed-desktop:~/Desktop$ openssl enc -aes-128-cfb -e -in file.txt -out fileCFB.txt -k password -p
salt=7B2720E032C3A26C
key=9ED557D4A3F1C24055F727889DB3DD08
iv =E517196C9C36794B261CB2488E5E8C39
```



```
00000000 53 61 6C 74 65 64 5F 5F 7B 27 20 E0 32 C3 A2 6C AA   Salted__{' .2..l.
00000011 87 38 5A 38 96 AC 47 51 8A 9D CA B1 94 B9 71 74 4B   .8Z8..GQ......qtK
00000022 B5 CC BB 6D D9 F1 A4 5F A2 91 99 94 27 D7 8A DF 81   ...m..._....'....
00000033 F9 36 50 7C 5F A4 DA 05 57 0A 8C 93 4D 82 29 87 FE   .6P|_...W...M.)..
00000044 DA 93 6A B1 A9 00 FA 9A D3 CF 2E 03 B7 86 A0 8B F5   ..j..............
00000055 68 9E 22 E5 EB 00 B1 15 0F                           h."......
```

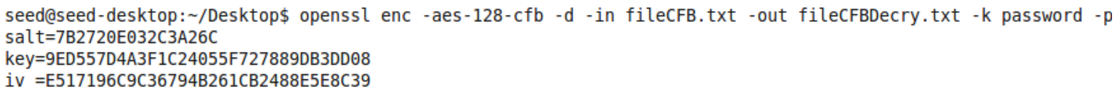
Figure 17 file.txt after encryption in CFB mode

The 30th byte is 0x94(10010100), I changed it to 0x95(10010101)



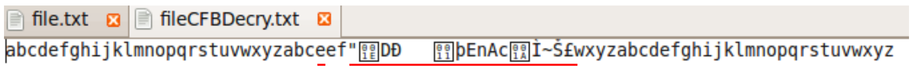
```
00000000 53 61 6C 74 65 64 5F 5F 7B 27 20 E0 32 C3 A2 6C AA   Salted__{' .2..l.
00000011 87 38 5A 38 96 AC 47 51 8A 9D CA B1 94 B9 71 74 4B   .8Z8..GQ......qtK
00000022 B5 CC BB 6D D9 F1 A4 5F A2 91 99 95 27 D7 8A DF 81   ...m..._....'....
00000033 F9 36 50 7C 5F A4 DA 05 57 0A 8C 93 4D 82 29 87 FE   .6P|_...W...M.)..
00000044 DA 93 6A B1 A9 00 FA 9A D3 CF 2E 03 B7 86 A0 8B F5   ..j..............
00000055 68 9E 22 E5 EB 00 B1 15 0F                           h."......
```

Figure 18 one bit error



```
seed@seed-desktop:~/Desktop$ openssl enc -aes-128-cfb -d -in fileCFB.txt -out fileCFBDecry.txt -k password -p
salt=7B2720E032C3A26C
key=9ED557D4A3F1C24055F727889DB3DD08
iv =E517196C9C36794B261CB2488E5E8C39
```

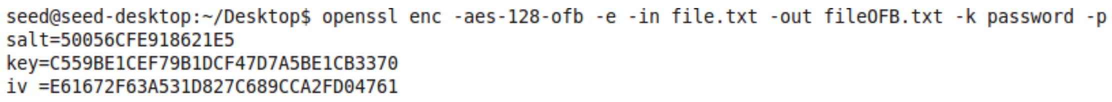
After opening fileCFBDecry.txt, I found it is similar to CBC mode. Flipping one bit in C2 will have a huge influence to M3 and tiny influence to M2. The reason is the same as above.



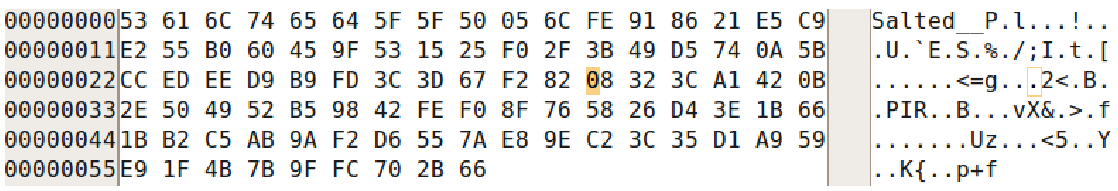
```
file.txt   fileCFBDecry.txt
abcdefghijklmnopqrstuvwxyzabceef"[1E]DÐ    [11]þEnAc[1A]İ~Š£wxyzabcdefghijklmnopqrstuvwxyz
```

Figure 19 Decrypted text after corruption---fileCFBDecry.txt

OFB:



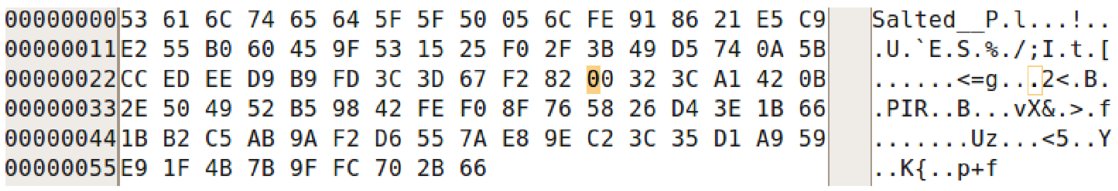
```
seed@seed-desktop:~/Desktop$ openssl enc -aes-128-ofb -e -in file.txt -out fileOFB.txt -k password -p
salt=50056CFE918621E5
key=C559BE1CEF79B1DCF47D7A5BE1CB3370
iv =E61672F63A531D827C689CCA2FD04761
```



```
00000000 53 61 6C 74 65 64 5F 5F 50 05 6C FE 91 86 21 E5 C9   Salted__P.l...!..
00000011 E2 55 B0 60 45 9F 53 15 25 F0 2F 3B 49 D5 74 0A 5B   .U.`E.S.%./;I.t.[
00000022 CC ED EE D9 B9 FD 3C 3D 67 F2 82 08 32 3C A1 42 0B   ......<=g...2<.B.
00000033 2E 50 49 52 B5 98 42 FE F0 8F 76 58 26 D4 3E 1B 66   .PIR..B...vX&.>.f
00000044 1B B2 C5 AB 9A F2 D6 55 7A E8 9E C2 3C 35 D1 A9 59   .......Uz...<5..Y
00000055 E9 1F 4B 7B 9F FC 70 2B 66                           ..K{..p+f
```

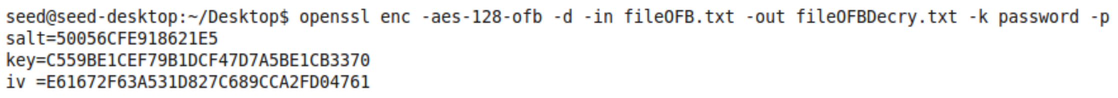
Figure 20 file.txt after encryption in OFB mode

The 30th byte is 0x08(00001000), I changed it to 0x00(00000000)



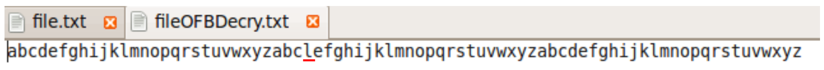
```
00000000 53 61 6C 74 65 64 5F 5F 50 05 6C FE 91 86 21 E5 C9   Salted__P.l...!..
00000011 E2 55 B0 60 45 9F 53 15 25 F0 2F 3B 49 D5 74 0A 5B   .U.`E.S.%./;I.t.[
00000022 CC ED EE D9 B9 FD 3C 3D 67 F2 82 00 32 3C A1 42 0B   ......<=g...2<.B.
00000033 2E 50 49 52 B5 98 42 FE F0 8F 76 58 26 D4 3E 1B 66   .PIR..B...vX&.>.f
00000044 1B B2 C5 AB 9A F2 D6 55 7A E8 9E C2 3C 35 D1 A9 59   .......Uz...<5..Y
00000055 E9 1F 4B 7B 9F FC 70 2B 66                           ..K{..p+f
```

Figure 21 one bit error



```
seed@seed-desktop:~/Desktop$ openssl enc -aes-128-ofb -d -in fileOFB.txt -out fileOFBDecry.txt -k password -p
salt=50056CFE918621E5
key=C559BE1CEF79B1DCF47D7A5BE1CB3370
iv =E61672F63A531D827C689CCA2FD04761
```

Because of the decryption scheme of OFB mode, when one bit error occurs in C2, there will only be one bit error in M2. No error propagation. The one bit error in M2 will lead to one letter error, just like the one with a red underline in Figure 22.



```
abcdefghijklmnopqrstuvwxyzabclefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz
```

Figure 22 Decrypted text after corruption---fileOFBDecry.txt

Implication of these differences:

Just like what I mentioned in the assumption, these differences indicate that if there is one bit error in ciphertext, only the block, where the error bit is, will be influenced in ECB and OFB modes(No error propagation). The extent of influence is different in these two modes. However, in CBC and CFB modes, the block where the error bit is and its first following block will be influenced and cannot be recovered due to error propagation.

Task 4: Programming using the Crypto Library:

To successfully compile my source code, the following preparation work needs to be done.

(1) Fix apt-get

As this version of ubuntu is EOL, apt-get doesn't work.

To fix:

append

""

deb http://old-releases.ubuntu.com/ubuntu/ jaunty main restricted universe multiverse

deb http://old-releases.ubuntu.com/ubuntu/ jaunty-updates main restricted universe multiverse

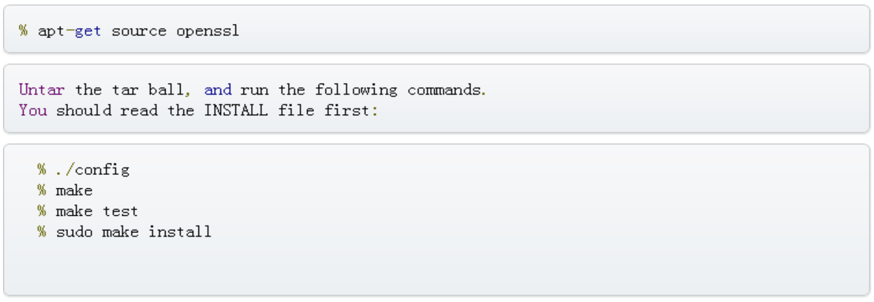
deb http://old-releases.ubuntu.com/ubuntu/ jaunty-security main restricted universe multiverse

""

to /etc/apt/sources.list as root (There are two ways to change the file. One is change the limits of authority using "chmod 777". The other is "vi sources.list" as root, append the three lines above and ":q" to save and quit.)

then do sudo apt-get update

(2) apt-get and installation:

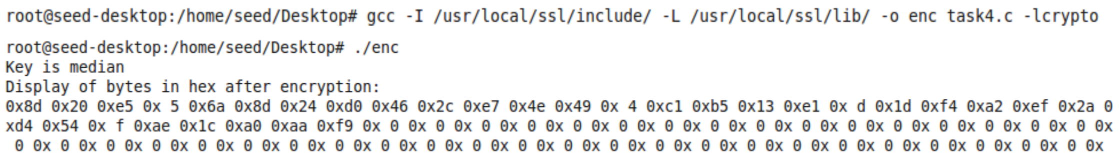


```
% apt-get source openssl
```

```
Untar the tar ball, and run the following commands.
You should read the INSTALL file first:
```

```
  % ./config
  % make
  % make test
  % sudo make install
```

Then I can compile and run my code successfully.



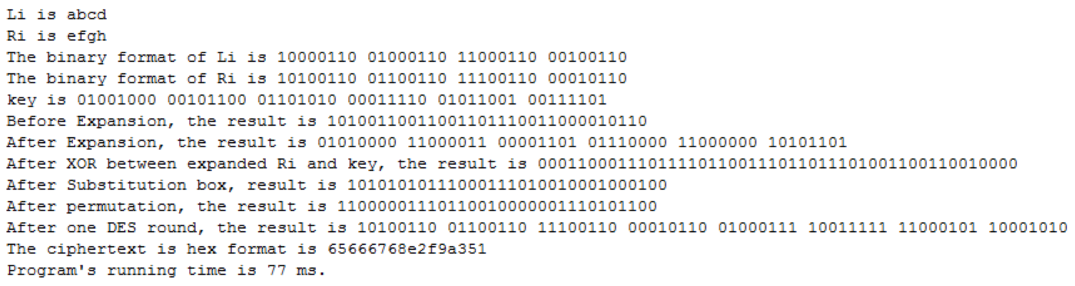
```
root@seed-desktop:/home/seed/Desktop# gcc -I /usr/local/ssl/include/ -L /usr/local/ssl/lib/ -o enc task4.c -lcrypto

root@seed-desktop:/home/seed/Desktop# ./enc
Key is median
Display of bytes in hex after encryption:
0x8d 0x20 0xe5 0x 5 0x6a 0x8d 0x24 0xd0 0x46 0x2c 0xe7 0x4e 0x49 0x 4 0xc1 0xb5 0x13 0xe1 0x d 0x1d 0xf4 0xa2 0xef 0x2a 0
xd4 0x54 0x f 0xae 0x1c 0xa0 0xaa 0xf9 0x 0 0x 0 0x 0 0x 0 0x 0 0x 0 0x 0 0x 0 0x 0 0x 0 0x 0 0x 0 0x 0 0x 0 0x 0 0x
 0 0x 0 0x 0 0x 0 0x 0 0x 0 0x 0 0x 0 0x 0 0x 0 0x 0 0x 0 0x 0 0x 0 0x 0 0x 0 0x 0 0x 0 0x 0 0x 0 0x 0 0x 0 0x 0 0x
```

Figure 23 Program Result

It can be clearly found out the key is “median”.

Task 5: Write your own DES encryption code (one round):



```
Li is abcd
Ri is efgh
The binary format of Li is 10000110 01000110 11000110 00100110
The binary format of Ri is 10100110 01100110 11100110 00010110
key is 01001000 00101100 01101010 00011110 01011001 00111101
Before Expansion, the result is 10100110011001101110011000010110
After Expansion, the result is 01010000 11000011 00001101 01110000 11000000 10101101
After XOR between expanded Ri and key, the result is 000110001110111101100111011011101001100110010000
After Substitution box, result is 10101010111000111010010001000100
After permutation, the result is 11000001110110010000001110101100
After one DES round, the result is 10100110 01100110 11100110 00010110 01000111 10011111 11000101 10001010
The ciphertext is hex format is 65666768e2f9a351
Program's running time is 77 ms.
```

Figure 24 Program Result with plaintext block: abcdefgh