1.Experiment purpose

Through practical operation and experience of different encryption mode principles, as well as learning a series of technical operations related to them, we can preliminarily understand the relevant knowledge of Cryptography, improve our interest in learning Cryptography, and further understand and explore the subject of network security.

2.Experiment environment

Ubuntu 20.04 VMWare station

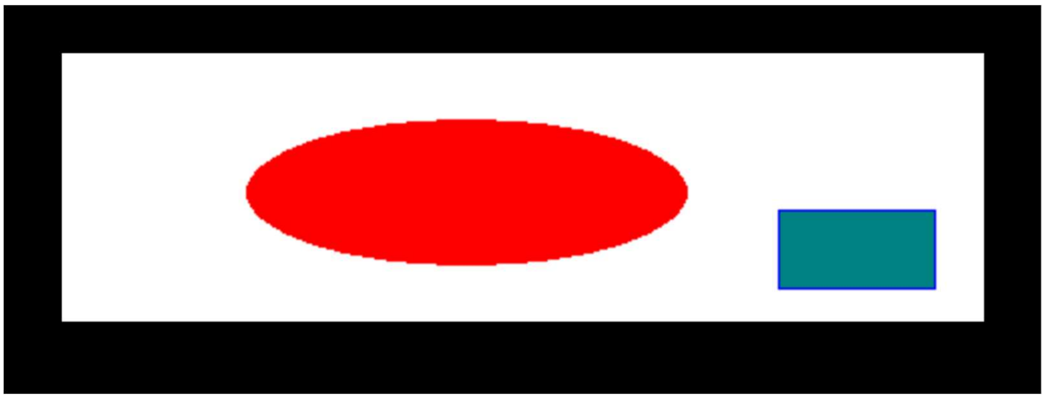
Intel core i7

3.Experiment process and record

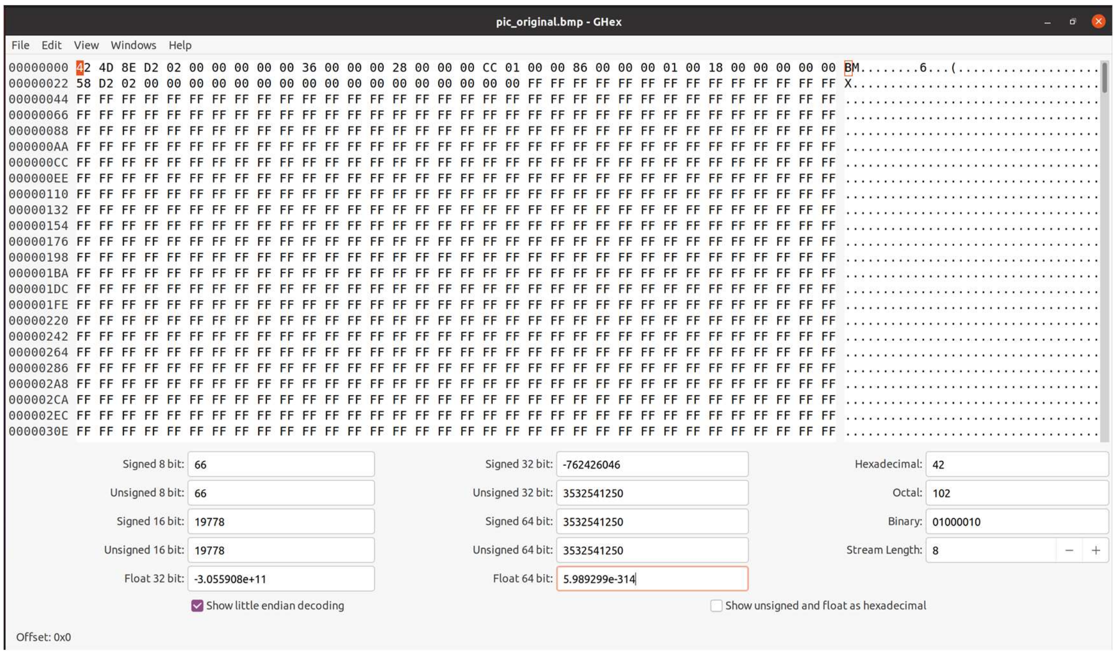
(1) Encryption Mode – ECB vs. CBC

ECB (Electronic Codebook) and CBC (Cipher Block Chaining) are two common symmetric encryption modes that have differences in encryption method, security, and resulting ciphertext.

Firstly, we obtain the image file through the wget command and use the image viewer to view it, as shown in the figure.



Open the image file using a binary file editor, as shown in the figure. The first 54 bytes are key information, which will be used in subsequent experimental steps.



pic_original.bmp - GHex

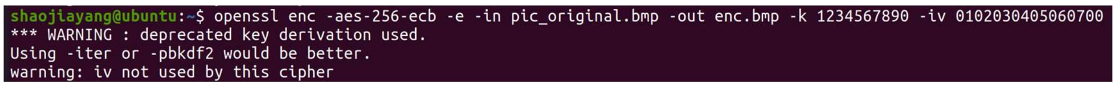
File Edit View Windows Help

```
00000000 42 4D 8E D2 02 00 00 00 00 00 36 00 00 00 28 00 00 00 CC 01 00 00 86 00 00 00 01 00 18 00 00 00 00 00 BM........6...(...................
00000022 58 D2 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 FF FF FF FF FF FF FF FF FF FF FF FF FF FF X.................................
00000044 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ..................................
00000066 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ..................................
00000088 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ..................................
000000AA FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ..................................
000000CC FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ..................................
000000EE FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ..................................
00000110 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ..................................
00000132 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ..................................
00000154 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ..................................
00000176 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ..................................
00000198 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ..................................
000001BA FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ..................................
000001DC FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ..................................
000001FE FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ..................................
00000220 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ..................................
00000242 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ..................................
00000264 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ..................................
00000286 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ..................................
000002A8 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ..................................
000002CA FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ..................................
000002EC FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ..................................
0000030E FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ..................................
```

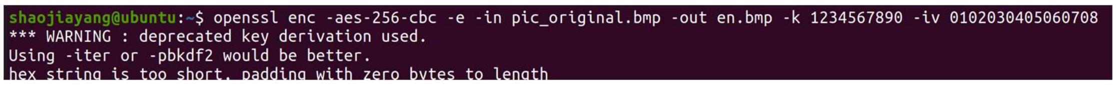
```
  Signed 8 bit: 66              Signed 32 bit: -762426046      Hexadecimal: 42
Unsigned 8 bit: 66            Unsigned 32 bit: 3532541250            Octal: 102
 Signed 16 bit: 19778           Signed 64 bit: 3532541250           Binary: 01000010
Unsigned 16 bit: 19778        Unsigned 64 bit: 3532541250    Stream Length: 8
  Float 32 bit: -3.055908e+11     Float 64 bit: 5.989299e-314
[✓] Show little endian decoding          [ ] Show unsigned and float as hexadecimal

Offset: 0x0
```

Next, we will use encryption mode to encrypt the image file. We will use ecb and cbc modes to encrypt the image, and use OpenSSL for encryption in the terminal to obtain the encrypted image file.

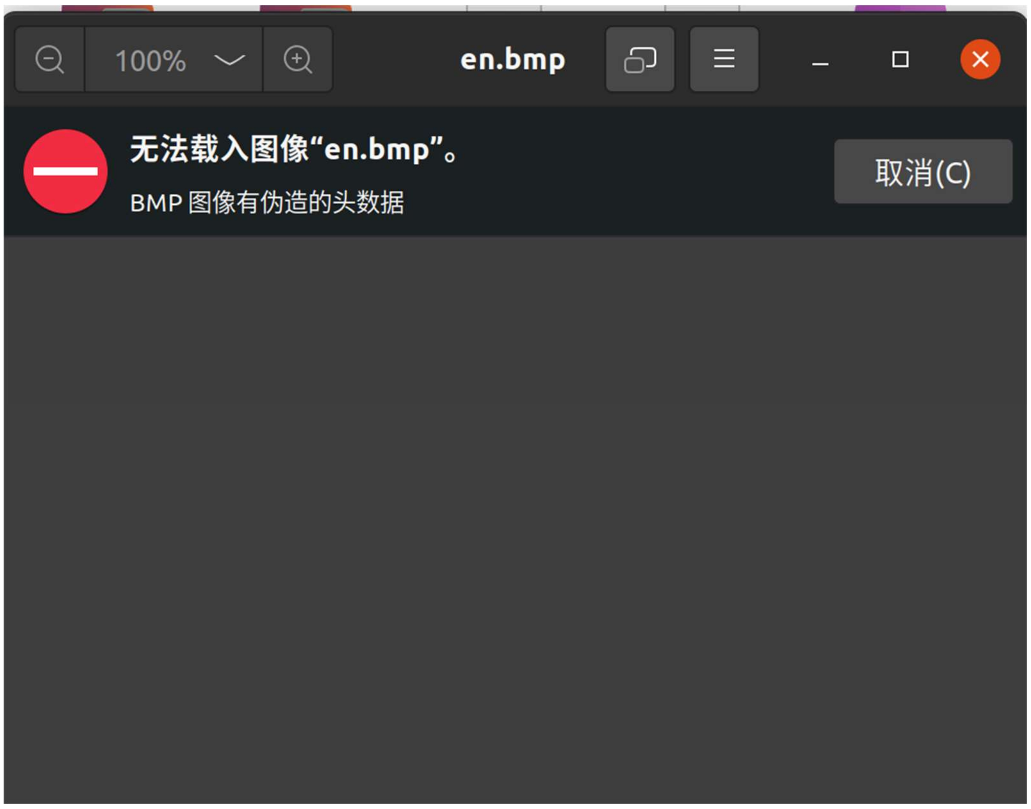


```
shaojiayang@ubuntu:~$ openssl enc -aes-256-ecb -e -in pic_original.bmp -out enc.bmp -k 1234567890 -iv 0102030405060700
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
warning: iv not used by this cipher
```



```
shaojiayang@ubuntu:~$ openssl enc -aes-256-cbc -e -in pic_original.bmp -out en.bmp -k 1234567890 -iv 0102030405060708
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
hex string is too short, padding with zero bytes to length
```

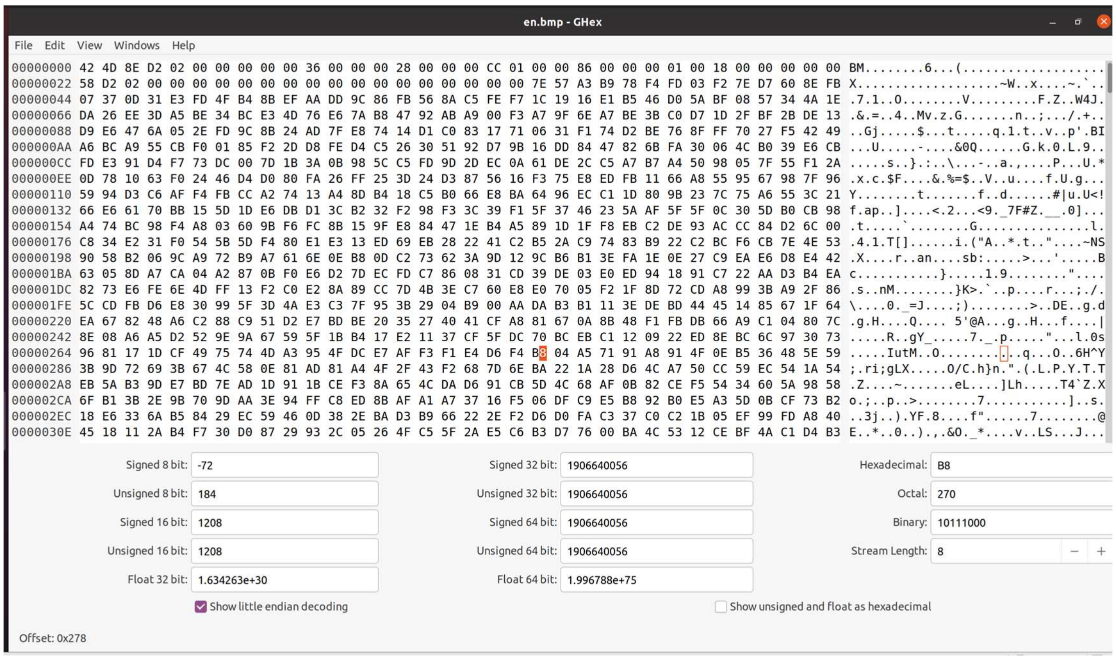
We attempted to open the encrypted image file using an image viewer and found that it could not be opened due to the presence of forged header data.



无法载入图像“en.bmp”。

BMP 图像有伪造的头数据

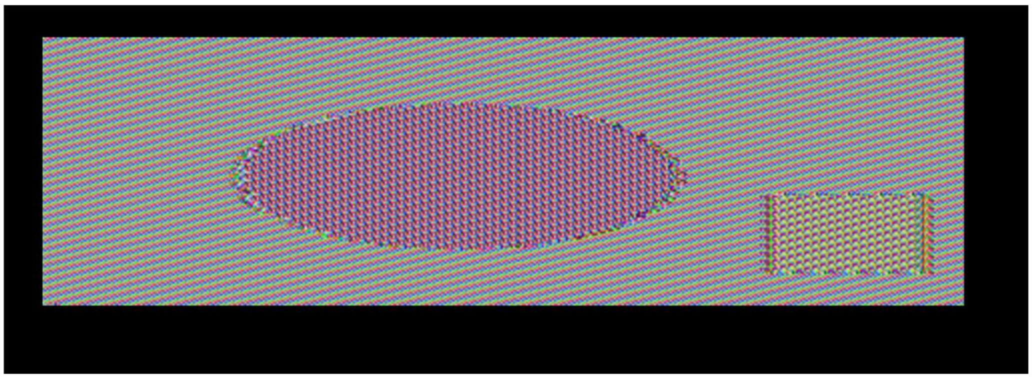
Therefore, we need to modify the header data of the image and use the binary file editor to replace the header information of the first 54 bytes of the encrypted image file with the header information of the original image, as shown in the figure, showing the replaced binary file.



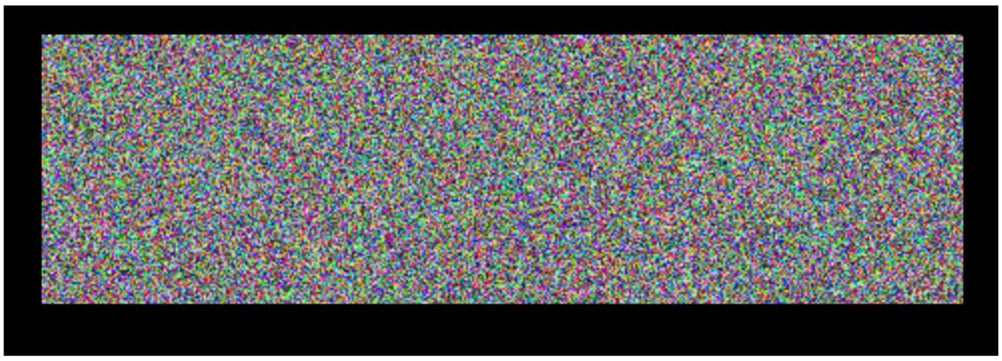
```
00000000 42 4D 8E D2 02 00 00 00 00 00 36 00 00 00 28 00 00 00 CC 01 00 00 86 00 00 00 01 00 18 00 00 00 00 00 BM........6...(..................
00000022 58 D2 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 7E 57 A3 B9 78 F4 FD 03 F2 7E D7 60 8E FB X...................~W..x....~.`..
00000044 07 37 0D 31 E3 FD 4F B4 8B EF AA DD 9C 86 FB 56 8A C5 FE F7 1C 19 16 E1 B5 46 D0 5A BF 08 57 34 4A 1E .7.1..O........V.........F.Z..W4J.
00000066 DA 26 EE 3D A5 BE 34 BC E3 4D 76 E6 7A B8 47 92 AB A9 00 F3 A7 9F 6E A7 BE 3B C0 D7 1D 2F BF 2B DE 13 .&.=..4..Mv.z.G.......n..;.../.+..
00000088 D9 E6 47 6A 05 2E FD 9C 8B 24 AD 7F E8 74 14 D1 C0 83 17 71 06 31 F1 74 D2 BE 76 8F FF 70 27 F5 42 49 ..Gj.....$...t.....q.1.t..v..p'.BI
000000AA A6 BC A9 55 CB F0 01 85 F2 2D D8 FE D4 C5 26 30 51 92 D7 9B 16 DD 84 47 82 6B FA 30 06 4C B0 39 E6 CB ...U.....-....&0Q......G.k.0.L.9..
000000CC FD E3 91 D4 F7 73 DC 00 7D 1B 3A 0B 98 5C C5 FD 9D 2D EC 0A 61 DE 2C C5 A7 B7 A4 50 98 05 7F 55 F1 2A .....s..}.:..\...-..a.,.....P...U.*
000000EE 0D 78 10 63 F0 24 46 D4 D0 80 FA 26 FF 25 3D 24 D3 87 56 16 F3 75 E8 ED FB 11 66 A8 55 95 67 98 7F 96 .x.c.$F....&.%=$..V..u....f.U.g...
00000110 59 94 D3 C6 AF F4 FB CC A2 74 13 A4 8D B4 18 C5 B0 66 E8 BA 64 96 EC C1 1D 80 9B 23 7C 75 A6 55 3C 21 Y........t.......f..d......#|u.U<!
00000132 66 E6 61 70 BB 15 5D 1D E6 DB D1 3C B2 32 F2 98 F3 3C 39 F1 5F 37 46 23 5A AF 5F 5F 0C 30 5D B0 CB 98 f.ap..]....<.2...<9._7F#Z.__.0]...
00000154 A4 74 BC 98 F4 A8 03 60 9B F6 FC 8B 15 9F E8 84 47 1E B4 A5 89 1D 1F F8 EB C2 DE 93 AC CC 84 D2 6C 00 .t.....`........G...............l.
00000176 C8 34 E2 31 F0 54 5B 5D F4 80 E1 E3 13 ED 69 EB 28 22 41 C2 B5 2A C9 74 83 B9 22 C2 BC F6 CB 7E 4E 53 .4.1.T[]......i.("A..*.t..".....~NS
00000198 90 58 B2 06 9C A9 72 B9 A7 61 6E 0E B8 0D C2 73 62 3A 9D 12 9C B6 B1 3E FA 1E 0E 27 C9 EA E6 D8 E4 42 .X....r..an....sb:.....>...'.....B
000001BA 63 05 8D A7 CA 04 A2 87 0B F0 E6 D2 7D EC FD C7 86 08 31 CD 39 DE 03 E0 ED 94 18 91 C7 22 AA D3 B4 EA c...........}.....1.9........"....
000001DC 82 73 E6 FE 6E 4D FF 13 F2 C0 E2 8A 89 CC 7D 4B 3E C7 60 E8 E0 70 05 F2 1F 8D 72 CD A8 99 3B A9 2F 86 .s..nM........}K>.`..p....r...;./.
000001FE 5C CD FB D6 E8 30 99 5F 3D 4A E3 C3 7F 95 3B 29 04 B9 00 AA DA B3 B1 11 3E DE BD 44 45 14 85 67 1F 64 \....0._=J....;)........>..DE..g.d
00000220 EA 67 82 48 A6 C2 88 C9 51 D2 E7 BD BE 20 35 27 40 41 CF A8 81 67 0A 8B 48 F1 FB DB 66 A9 C1 04 80 7C .g.H....Q.... 5'@A...g..H...f....|
00000242 8E 08 A6 A5 D2 52 9E 9A 67 59 5F 1B B4 17 E2 11 37 CF 5F DC 70 BC EB C1 12 09 22 ED 8E BC 6C 97 30 73 .....R..gY_.....7._.p....."...l.0s
00000264 96 81 17 1D CF 49 75 74 4D A3 95 4F DC E7 AF F3 F1 E4 D6 F4 B8 04 A5 71 91 A8 91 4F 0E B5 36 48 5E 59 .....IutM..O..........q...O..6H^Y
00000286 3B 9D 72 69 3B 67 4C 58 0E 81 AD 81 A4 4F 2F 43 F2 68 7D 6E BA 22 1A 28 D6 4C A7 50 CC 59 EC 54 1A 54 ;.ri;gLX.....O/C.h}n.".(.L.P.Y.T.T
000002A8 EB 5A B3 9D E7 BD 7E AD 1D 91 1B CE F3 8A 65 4C DA D6 91 CB 5D 4C 68 AF 0B 82 CE F5 54 34 60 5A 98 58 .Z....~.......eL....]Lh.....T4`Z.X
000002CA 6F B1 3B 2E 9B 70 9D AA 3E 94 FF C8 ED 8B AF A1 A7 37 16 F5 06 DF C9 E5 B8 92 B0 E5 A3 5D 0B CF 73 B2 o.;..p..>.......7...........]..s.
000002EC 18 E6 33 6A B5 84 29 EC 59 46 0D 38 2E BA D3 B9 66 22 2E F2 D6 D0 FA C3 37 C0 C2 1B 05 EF 99 FD A8 40 ..3j..).YF.8....f"......7........@
0000030E 45 18 11 2A B4 F7 30 D0 87 29 93 2C 05 26 4F C5 5F 2A E5 C6 B3 D7 76 00 BA 4C 53 12 CE BF 4A C1 D4 B3 E..*..0..).,.&O._*....v..LS...J...
```

```
Signed 8 bit: -72              Signed 32 bit: 1906640056       Hexadecimal: B8
Unsigned 8 bit: 184            Unsigned 32 bit: 1906640056     Octal: 270
Signed 16 bit: 1208            Signed 64 bit: 1906640056       Binary: 10111000
Unsigned 16 bit: 1208          Unsigned 64 bit: 1906640056     Stream Length: 8
Float 32 bit: 1.634263e+30     Float 64 bit: 1.996788e+75
[✓] Show little endian decoding                                [ ] Show unsigned and float as hexadecimal
Offset: 0x278
```

After the modifications were completed, we used the image viewer again to view the file and found that it could be viewed, and the final displayed file was a blurry mosaic snowflake. However, the final encrypted images presented by the two encryption modes are still different. Images encrypted using ECB mode can still vaguely see some patterns, while images encrypted using CBC mode are completely a mess.



(ecb)



(cbc)

interpretative statement

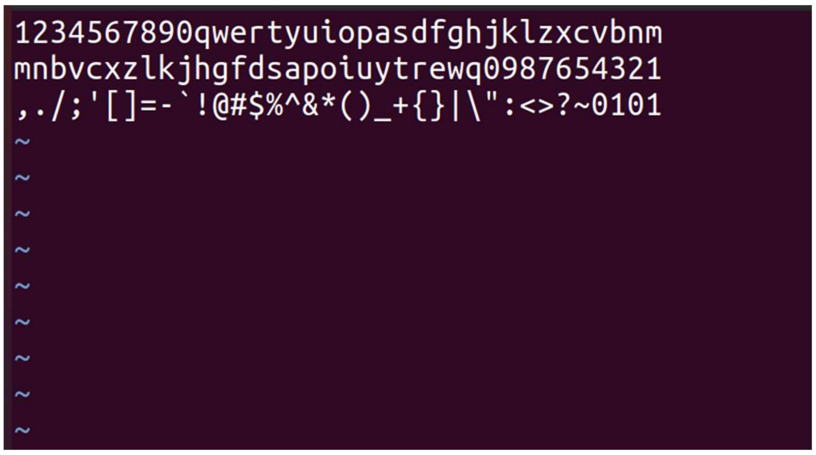
In ECB mode, identical plaintext blocks are replaced with identical ciphertext blocks, which makes it vulnerable to attacks since the patterns in the ciphertext can be observed and used to break the encryption. In contrast, in CBC mode, each plaintext block is XORed with the previous ciphertext block before encryption, so even if there are identical plaintext blocks, they will be replaced with different ciphertext blocks, which increases the difficulty of breaking the encryption.

When encrypting image files, the differences between ECB and CBC modes can be reflected in the resulting ciphertext. In ECB mode, if there are repeating patterns or color blocks in the image, they will be replaced with identical ciphertext blocks, leading to noticeable patterns in the resulting ciphertext and lower security. On the other hand, in CBC mode, even if there are repeating patterns or color blocks in the image, they will be replaced with different ciphertext blocks, resulting in ciphertext without noticeable patterns and higher security.

Therefore, when encrypting image files or other sensitive data, it is recommended to use CBC mode to improve the security of the resulting ciphertext.

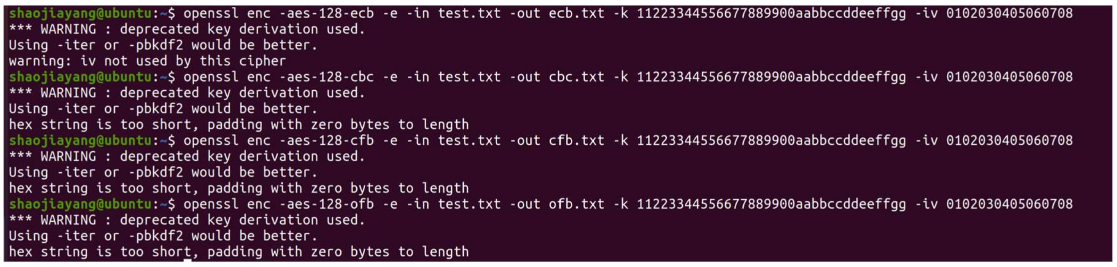
(2) Encryption Mode – Corrupted Cipher Text

Firstly, we create a text file larger than 64 bytes, as shown in the figure.



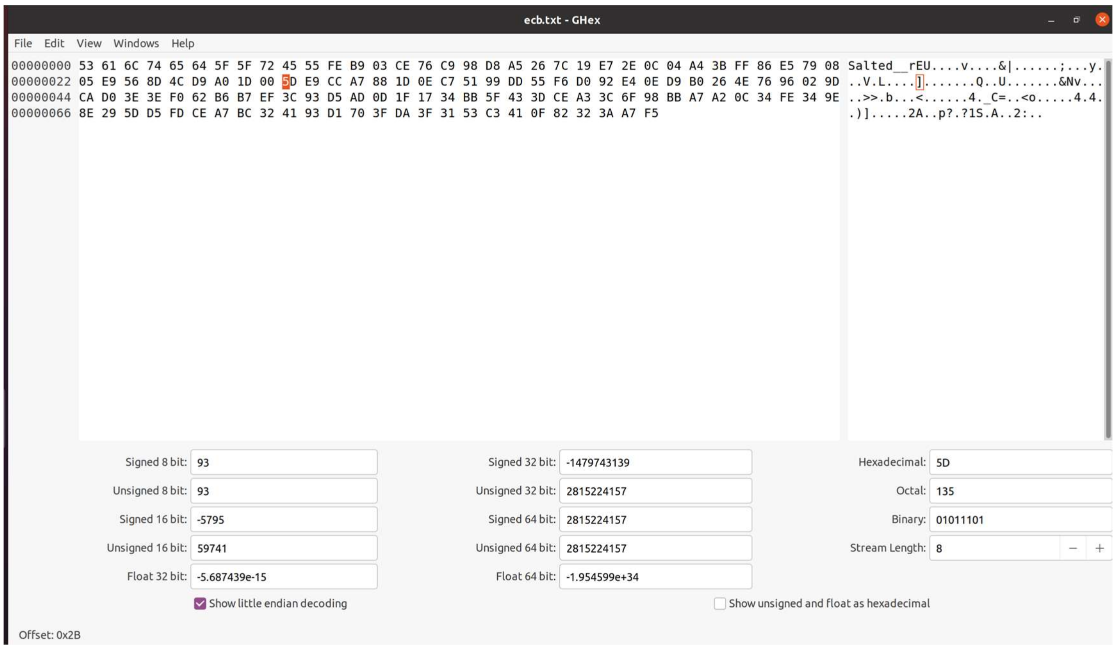
```
1234567890qwertyuiopasdfghjklzxcvbnm
mnbvcxzlkjhgfdsapoiuytrewq0987654321
,./;'[]=-`!@#$%^&*()_+{}|\":<>?~0101
~
~
~
~
~
~
~
~
~
```

Encrypt the original file using different encryption modes to obtain four different encrypted files.



```
shaojiayang@ubuntu:~$ openssl enc -aes-128-ecb -e -in test.txt -out ecb.txt -k 11223344556677889900aabbccddeeffgg -iv 0102030405060708
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
warning: iv not used by this cipher
shaojiayang@ubuntu:~$ openssl enc -aes-128-cbc -e -in test.txt -out cbc.txt -k 11223344556677889900aabbccddeeffgg -iv 0102030405060708
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
hex string is too short, padding with zero bytes to length
shaojiayang@ubuntu:~$ openssl enc -aes-128-cfb -e -in test.txt -out cfb.txt -k 11223344556677889900aabbccddeeffgg -iv 0102030405060708
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
hex string is too short, padding with zero bytes to length
shaojiayang@ubuntu:~$ openssl enc -aes-128-ofb -e -in test.txt -out ofb.txt -k 11223344556677889900aabbccddeeffgg -iv 0102030405060708
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
hex string is too short, padding with zero bytes to length
```

Next, use the binary file editor to simulate the 30th byte of the encrypted file being damaged, and modify the 30th byte of each of the four encrypted files separately

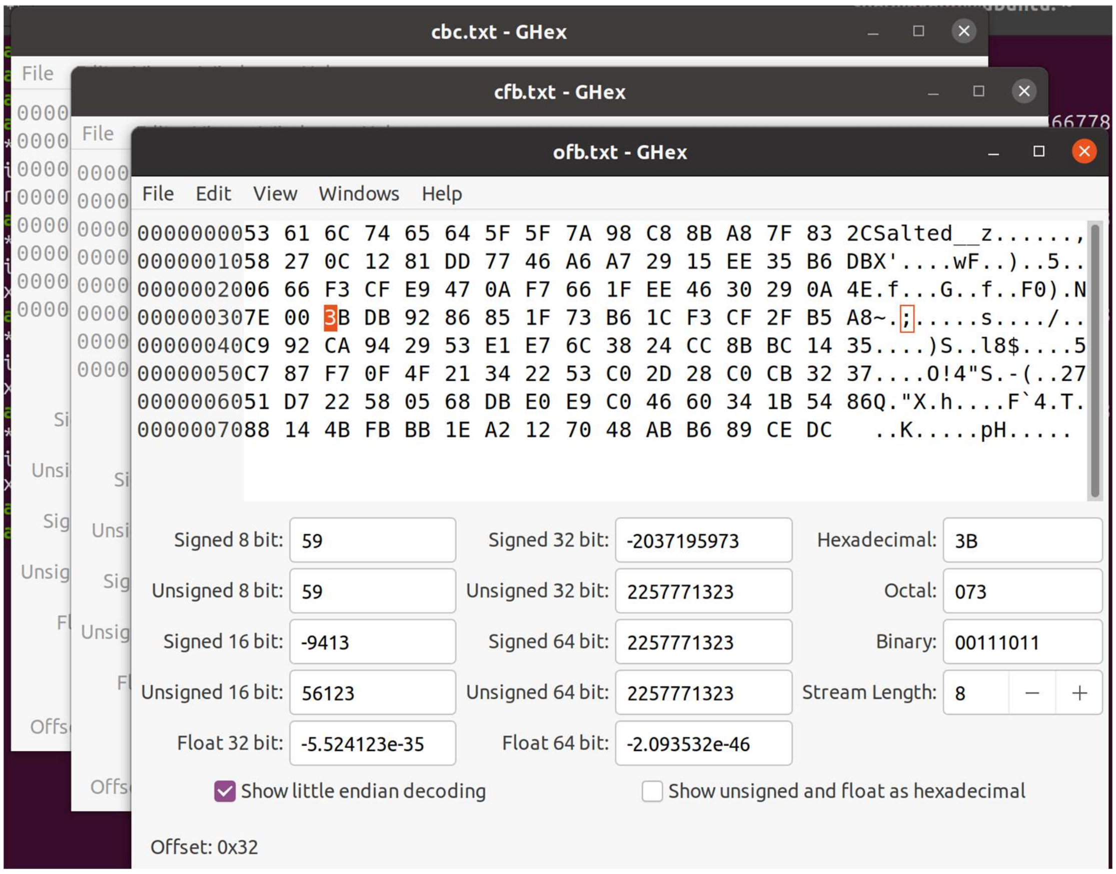


```
ecb.txt - GHex
00000000 53 61 6C 74 65 64 5F 5F 72 45 55 FE B9 03 CE 76 C9 98 D8 A5 26 7C 19 E7 2E 0C 04 A4 3B FF 86 E5 79 08 Salted__rEU....v....&|......;...y.
00000022 05 E9 56 8D 4C D9 A0 1D 00 5D E9 CC A7 88 1D 0E C7 51 99 DD 55 F6 D0 92 E4 0E D9 B0 26 4E 76 96 02 9D ..V.L....].......Q..U.......&Nv...
00000044 CA D0 3E 3E F0 62 B6 B7 EF 3C 93 D5 AD 0D 1F 17 34 BB 5F 43 3D CE A3 3C 6F 98 BB A7 A2 0C 34 FE 34 9E ..>>.b...<......4._C=..<o.....4.4.
00000066 8E 29 5D D5 FD CE A7 BC 32 41 93 D1 70 3F DA 3F 31 53 C3 41 0F 82 32 3A A7 F5                         .)].....2A..p?.?1S.A..2:..
```

Signed 8 bit:	93	Signed 32 bit:	-1479743139	Hexadecimal:	5D
Unsigned 8 bit:	93	Unsigned 32 bit:	2815224157	Octal:	135
Signed 16 bit:	-5795	Signed 64 bit:	2815224157	Binary:	01011101
Unsigned 16 bit:	59741	Unsigned 64 bit:	2815224157	Stream Length:	8
Float 32 bit:	-5.687439e-15	Float 64 bit:	-1.954599e+34		

[✓] Show little endian decoding [ ] Show unsigned and float as hexadecimal

Offset: 0x2B



cbc.txt - GHex

cfb.txt - GHex

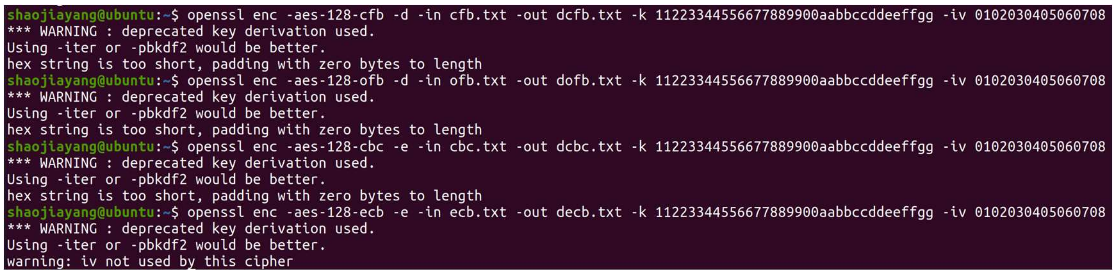
```
ofb.txt - GHex
0000000053 61 6C 74 65 64 5F 5F 7A 98 C8 8B A8 7F 83 2CSalted__z......,
0000001058 27 0C 12 81 DD 77 46 A6 A7 29 15 EE 35 B6 DBX'....wF..)..5..
0000002006 66 F3 CF E9 47 0A F7 66 1F EE 46 30 29 0A 4E.f...G..f..F0).N
000000307E 00 3B DB 92 86 85 1F 73 B6 1C F3 CF 2F B5 A8~.;.....s..../..
00000040C9 92 CA 94 29 53 E1 E7 6C 38 24 CC 8B BC 14 35....)S..l8$....5
00000050C7 87 F7 0F 4F 21 34 22 53 C0 2D 28 C0 CB 32 37....O!4"S.-(..27
0000006051 D7 22 58 05 68 DB E0 E9 C0 46 60 34 1B 54 86Q."X.h....F`4.T.
0000007088 14 4B FB BB 1E A2 12 70 48 AB B6 89 CE DC   ..K.....pH.....
```

Signed 8 bit:	59	Signed 32 bit:	-2037195973	Hexadecimal:	3B
Unsigned 8 bit:	59	Unsigned 32 bit:	2257771323	Octal:	073
Signed 16 bit:	-9413	Signed 64 bit:	2257771323	Binary:	00111011
Unsigned 16 bit:	56123	Unsigned 64 bit:	2257771323	Stream Length:	8
Float 32 bit:	-5.524123e-35	Float 64 bit:	-2.093532e-46		

[✓] Show little endian decoding [ ] Show unsigned and float as hexadecimal

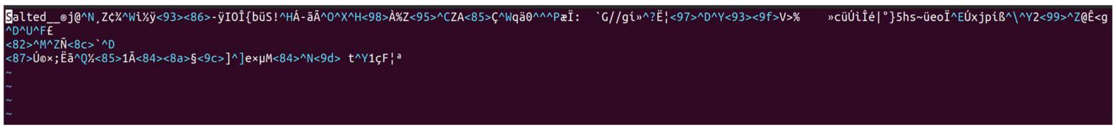
Offset: 0x32

Using the same method to decrypt four encrypted files.



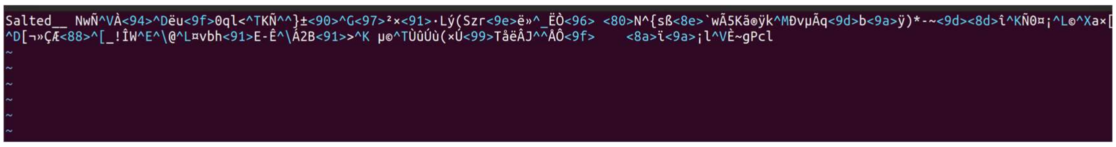
```
shaojiayang@ubuntu:~$ openssl enc -aes-128-cfb -d -in cfb.txt -out dcfb.txt -k 11223344556677889900aabbccddeeffgg -iv 0102030405060708
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
hex string is too short, padding with zero bytes to length
shaojiayang@ubuntu:~$ openssl enc -aes-128-ofb -d -in ofb.txt -out dofb.txt -k 11223344556677889900aabbccddeeffgg -iv 0102030405060708
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
hex string is too short, padding with zero bytes to length
shaojiayang@ubuntu:~$ openssl enc -aes-128-cbc -e -in cbc.txt -out dcbc.txt -k 11223344556677889900aabbccddeeffgg -iv 0102030405060708
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
hex string is too short, padding with zero bytes to length
shaojiayang@ubuntu:~$ openssl enc -aes-128-ecb -e -in ecb.txt -out decb.txt -k 11223344556677889900aabbccddeeffgg -iv 0102030405060708
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.
warning: iv not used by this cipher
```

Looking at the four decrypted files separately and comparing them with the original file, it can be found that the decrypted content of the four files is significantly different.



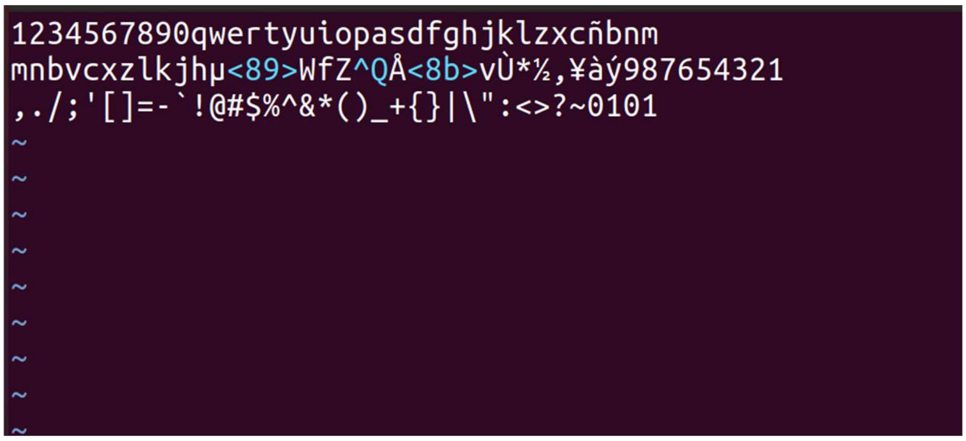
```
Salted__®j@^N¸Z¢¾^Wi½ÿ<93><86>-ÿIOÎ{büS!^HÁ-ãÃ^O^X^H<98>À%Z<95>^CZA<85>Ç^Wqä0^^^PæÏ:  `G//gí»^?Ë¦<97>^D^Y<93><9f>V>%    »cüÚìÎé|°}5hs~üeoÏ^EÚxjpíß^\^Y2<99>^Z@Ê<g
^D^U^F£
<82>^M^ZÑ<8c>`^D
<87>Ú©×;Ëã^Q½<85>1Ã<84><8a>§<9c>]^]e×µM<84>^N<9d> t^Y1çF¦ª
~
~
~
~
```

(ECB)



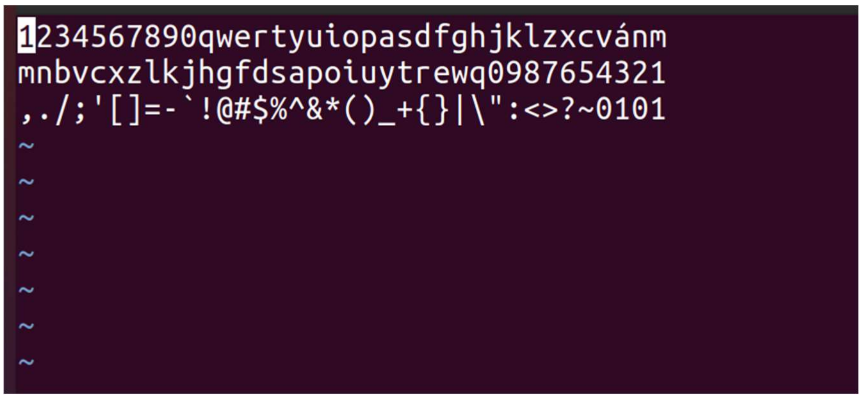
```
Salted__ NwÑ^VÀ<94>^Dëu<9f>0ql<^TKÑ^^}±<90>^G<97>²×<91>·Lý(Szr<9e>ë»^_ËÒ<96> <80>N^{sß<8e>`wÃ5Kã®ÿk^MÐvµÃq<9d>b<9a>ÿ)*-~<9d><8d>î^KÑ0¤¡^L©^Xa×[
^D[¬»ÇÆ<88>^[_!ÎW^E^\@^L¤vbh<91>E-Ê^\Á2B<91>>^K µ©^TÙûÚù(×Ú<99>TåëÂJ^^ÅÔ<9f>     <8a>ï<9a>¡l^VÈ~gPcl
~
~
~
~
~
~
```

(CBC)



```
1234567890qwertyuiopasdfghjklzxcñbnm
mnbvcxzlkjhµ<89>WfZ^QÅ<8b>vÙ*½,¥àý987654321
,./;'[]=-`!@#$%^&*()_+{}|\":<>?~0101
~
~
~
~
~
~
~
~
~
```

(CFB)



```
1234567890qwertyuiopasdfghjklzxcvánm
mnbvcxzlkjhgfdsapoiuytrewq0987654321
,./;'[]=-`!@#$%^&*()_+{}|\":<>?~0101
~
~
~
~
~
~
~
```

(OFB)

It can be found that the decrypted content of ECB and CBC is vastly different from the original file, and the original file content cannot be seen at all.

In CFB and OFB modes, the original file content can be roughly seen, and the degree of damage and modification is relatively small.

Concise answers to these three questions

The amount of information that can be recovered by decrypting a corrupted file depends on several factors, including the type of encryption mode used.

When using ECB mode encryption, the same plaintext block is replaced by the same ciphertext block, making it vulnerable to certain types of attacks. If the corrupted file only affects one ciphertext block, then it is possible to fully recover the corresponding plaintext block. However, if the corruption affects multiple ciphertext blocks, only the parts of the information that have already been decrypted can be recovered.

In CBC mode encryption, each plaintext block is XORed with the previous ciphertext block before encryption. If the corrupted file only affects one ciphertext block, then it is possible to fully recover the corresponding plaintext block. However, if the corruption affects multiple ciphertext blocks, only the parts of the information that have already been decrypted can be recovered.

In CFB and OFB mode encryption, a pseudorandom bit stream is generated using the ciphertext block as input, which is then XORed with the plaintext block. If the corrupted file only affects one ciphertext block, then it is possible to fully recover the corresponding plaintext block. However, if the corruption affects multiple ciphertext blocks, only the parts of the information that have already been decrypted can be recovered.

These differences in encryption modes have implications for the security and efficiency of file encryption and decryption. ECB mode encryption is vulnerable to certain types of attacks, while CBC, CFB, and OFB modes can avoid these issues. Additionally, these modes have different efficiencies in processing large files, with ECB mode being faster due to its ability to process multiple blocks in parallel, while CBC, CFB, and OFB modes require sequential processing of each block, making them slower.

In summary, the amount of information that can be recovered from a corrupted file depends on the type of encryption mode used and the extent of the corruption. The choice of encryption mode is an important factor in ensuring data security and performance.

4.Experiment result and personal experiences

In this experiment, we compared the differences between the ECB and CBC encryption modes, and performed two tasks to verify their properties.

In Task 1, we encrypted a simple picture using both ECB and CBC encryption modes, and then displayed the encrypted picture as a legitimate BMP file by replacing the header information. We found that in ECB mode, the encrypted picture displayed noticeable repeating patterns, which made it easier to infer the original content of the picture. However, in CBC mode, the encrypted picture did not display any repeating patterns, making it harder to infer the original content.

In Task 2, we created a text file that was at least 64 bytes long and encrypted it using AES-128 algorithm. We then corrupted a single bit of the 30th byte in the encrypted file using a hex editor to simulate data corruption during transmission. Finally, we compared the decryption results of ECB, CBC, CFB, and OFB encryption modes with the correct key and IV values. We found that in ECB and CBC mode, since the entire block was destroyed due to the corruption of a single byte, no information could be recovered. However, in CFB, and OFB modes, only one block was affected, so most of the information could be recovered except for the last block.

If a file encrypted using CBC mode is corrupted by a single byte, the result of comparing the decrypted and original files depends on the position of the corrupted byte within the block.

After obtaining the experimental results, I have doubts about the experimental results of CBC mode, which seem to not meet expectations. After checking the data, I have the following explanation:

If the corrupted byte is in the middle of a block, all bytes in that block will be destroyed, causing the decrypted file to differ completely from the original file for that block. However, other blocks may still be decrypted correctly and match the original file.

If the corrupted byte is at the end of a block, only the last byte in that block will be destroyed, and other blocks can still be decrypted correctly and match the original file. However, the last byte in that block may differ from the original file, and this difference will be observed when comparing the decrypted and original files.

Since CBC mode uses the ciphertext of the previous block as the IV for the next block, corruption of one block may affect the decryption of subsequent blocks. If only one byte is corrupted, then only the affected block and subsequent blocks may be destroyed, and other blocks can still be decrypted correctly. However, the position of the corrupted byte has different effects on the decryption result.

Overall, this experiment deepened our understanding of the differences between the ECB and CBC encryption modes, and verified their performance in data transmission and decryption. We also learned that choosing the appropriate encryption mode based on the characteristics and security requirements of the data is crucial for ensuring data security and integrity in practical applications.