

LABORATORY CEL62: Cryptography and System Security Winter 2021

Experiment 3:	Crypto Encryption
----------------------	--------------------------

Name: Jash Jain

Roll No: 2019130021

TE Comps

Note: Students are advised to read through this lab sheet before doing an experiment. On-the-spot evaluation may be carried out during or at the end of the experiment. Your performance, teamwork effort, and learning attitude will count towards the marks.

Experiment 4: Crypto Lab – Secret-Key Encryption

1 OBJECTIVE

The learning objective of this lab is for students to get familiar with the concepts in the secret-key encryption. After finishing the lab, students should be able to gain first-hand experience on encryption algorithms, encryption modes, paddings, and initial vector (IV). Moreover, students will be able to use tools and write programs to encrypt/decrypt messages.

2 LAB ENVIRONMENT

Installing OpenSSL. In this lab, we will use openssl commands and libraries. We have already installed openssl binaries in our VM. It should be noted that if you want to use openssl libraries in your programs, you need to install several other things for the programming environment, including the header files, libraries, manuals, etc. We have already downloaded the necessary files under the directory

You should read the INSTALL file first:

```
% ./config  
% make  
% make test
```

```
% sudo make install
```

openssl-1.0.1. To configure and install openssl libraries, run the following commands.

Installing GHex. In this lab, we need to be able to view and modify files of binary format. We have installed in our VM GHex a hex editor for GNOME. It allows the user to load data from any file, view and edit it in either hex or ascii.

3 LAB TASKS

3.1 Task 1: Encryption using different ciphers and modes

In this task, we will play with various encryption algorithms and modes. You can use the following openssl enc command to encrypt/decrypt a file. To see the manuals, you can type man openssl and man

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

enc.

Please replace the ciphertype with a specific cipher type, such as -aes-128-cbc, -aes-128-cfb, -bf-cbc, etc. In this task, you should try at least 3 different ciphers and three different modes. You can find the meaning of the command-line options and all the supported cipher types by typing "man enc". We include some common options for the openssl enc command in the following:

-in <file>	input file
-out <file>	output file
-e	encrypt
-d	decrypt
-K/-iv -[pP]	key/iv in hex is the next argument print the iv/key (then exit if -P)

Encryption of plain text using different Ciphers. The plain text is as follows:



The screenshot shows a terminal window with a dark background. At the top left, there are three colored dots (red, yellow, green). At the top right, it says "hello.txt". Below that, the text reads: "Hello World. We are to attack now. If you aren't supposed to read this close it asap before the virus infects your machine and destroys it."

1. Encryption using aes-128-ecb

The Encrypted text and The hexdump for the above cipher text is:

```
[jashjain@Jashs-MacBook-Air temp % openssl enc -aes-128-ecb -e -in hello.txt -out ciphered.bin -K 112233445566778899aabcccd  
deeff  
[jashjain@Jashs-MacBook-Air temp % cat ciphered.bin  
  
yW\??p?x??R~??rY???>>?76=?0]  
[jashjain@Jashs-MacBook-Air temp % hexdump ciphered.bin  
000000 35 90 17 a6 ba 4a b0 80 eb b6 29 1c 1d bf 02 a4  
0000010 2f 51 68 26 be 92 34 dd 05 a2 3e 3e 13 80 37 1c  
0000020 36 3d db be df 4f 5d 0d db 91 9c 8a 52 99 84 70  
0000030 e4 78 9a fb 52 7e b8 a2 72 59 a5 0b 79 02 77 5c  
0000040 89 df 1c 67 3e af f2 df d4 77 ae fb 16 9a 03 15  
0000050 39 3f 8b ad e4 2a 98 47 9d e8 ee 49 79 43 7c 95  
0000060 c5 d1 9c ee 4f f3 ef f9 01 5a 83 53 e9 61 25  
0000070 2f 72 10 07 cd 3e 36 ed 62 40 4b 35 38 16 d0 b8  
0000080 2f 03 a1 b1 4f 90 b6 dd 86 d9 5b 53 75 c5 ee 99  
0000090  
jashjain@Jashs-MacBook-Air temp %
```

2. Encryption using aes-128-cbc

The cipher text and the hexdump of the cipher text is as follows:

```
[jashjain@Jashs-MacBook-Air temp % openssl enc -aes-128-cbc -e -in hello.txt -out ciphered.bin -K 112233445566778899aabcccddeff -iv 01020304050607080  
jashjain@Jashs-MacBook-Air temp % cat ciphered.bin  
u??H??|??_Yh5??As??~NILJa?W?qz?????=a%l1?b???  
%%?Zc ?t????%?h?W?  
????%?{??W?C ?0?  
????e?r?Q&??,h??  
jashjain@Jashs-MacBook-Air temp % hexdump ciphered.bin  
000000 75 e2 df 7d 48 f9 98 7c c3 d9 92 9f f5 5f 59 68  
0000010 35 f9 17 c1 41 73 f4 a7 06 7e 4e 49 4c 4a 61 95  
0000020 57 06 a9 b2 71 7a b2 c6 f3 f7 c2 a8 3d 61 25 6c  
0000030 31 19 f9 62 e1 06 9e 13 d0 0a a9 15 25 a3 5a 11  
0000040 63 1d 09 b9 74 8d 8e aa 1e 06 bd 25 f8 d7 68 99  
0000050 30 57 0b 99 b5 b7 8e 25 82 7b ce e1 57 8a 43 20  
0000060 9d 30 f5 0b f6 83 b7 65 cf cf 84 cb c7 aa 26 e0  
0000070 e9 2c 68 aa 8e ed 93 aa eb e8 31 35 99 6c dd ee  
0000080 f0 ab 70 5e da 13 05 d6 75 0e a8 c1 68 b0 64 af  
0000090  
jashjain@Jashs-MacBook-Air temp %
```

3. Encryption using aes-128-cfb

The cipher text and the hexdump for the same is as follows:

```
[jashjain@Jashs-MacBook-Air temp % openssl enc -aes-128-cfb -e -in hello.txt -out ciphered.bin -K 112233445566778899aabcccddeff -iv 01020304050607080  
jashjain@Jashs-MacBook-Air temp % cat ciphered.bin  
!/?",F?4?????C?#?c?G?????'?KML0eu?  
]/$ ??*?????p?  
ÿ ,/p0?9B?0???[v??.?K/?(^#?????  
#J??v??x5?[e?n??f?5???t??  
jashjain@Jashs-MacBook-Air temp % hexdump ciphered.bin  
000000 21 d0 66 8f 80 10 22 2c 46 9d 34 f3 39 1c d9 da  
0000010 e2 43 e4 23 89 63 96 47 fd 86 be 83 27 dc 4b 4d  
0000020 4c 4f 65 75 a0 0b 5d 2f fd 24 20 85 a4 51 08 2a  
0000030 b3 a0 89 a5 12 fe d4 70 7d ff 0a d2 8b 09 98 2c  
0000040 12 2f d4 97 30 ed 39 42 d5 4f 88 96 fe 5b 76 17  
0000050 ba e3 96 a9 15 e4 1d 4b 2f 00 04 f1 28 5e b9 23  
0000060 fd 1f 1d b2 d6 ed 0c 23 b3 4a b9 c6 76 12 91 a3  
0000070 ee 08 16 78 cc b4 35 dd 5b da 83 07 6e a9 e1 87  
0000080 bf 9c 66 fd 35 96 93 b9 74 99 f5  
000008b  
jashjain@Jashs-MacBook-Air temp %
```

4. Encryption using aes-192-ecb

The cipher text and hexdump of the same is as follows:

```
[jashjain@Jashs-MacBook-Air temp % openssl enc -aes-192-ecb -e -in hello.txt -out ciphered.bin -K 112233445566778899aabccddeff
[jashjain@Jashs-MacBook-Air temp % cat ciphered.bin
m3?*H2????!jv???:??
?g?vr?|?<F??6?~?      3?S??{?H?g?s???|?Po?5rP6[i6]???zA? u??`b?v$??j?q(??"?E?[IH9B?*vrGse'??]??
jashjain@Jashs-MacBook-Air temp % hexdump ciphered.bin
00000000 99 b7 0d 6d 33 9b 2a 48 32 97 ab f8 17 fb 21 05
00000010 6a 76 ed d4 3a 18 3a dd c8 93 ec aa 0a a7 0e 0f
00000020 67 e6 03 76 72 93 7c fb 3c 46 18 8b 02 84 de ab
00000030 08 c0 36 ed 89 8d 7e f1 ac 1e 19 09 d5 85 d9 d7
00000040 ae c8 53 d3 fb 7b b6 48 a5 67 d8 73 c0 ca ef 7c
00000050 ab 50 6f 9c 35 72 50 36 5b 69 36 c3 08 5d b2 a6
00000060 d2 15 7a 41 d5 9a 06 75 c3 d4 cb 98 62 12 05 ce
00000070 76 24 8a d6 6a 85 71 28 ce 22 9c 45 fd 5b 49 48
00000080 39 42 f2 2a 76 72 47 12 73 65 27 a1 17 81 7c 05
00000090
jashjain@Jashs-MacBook-Air temp %
```

5. Encryption using aes-192-cbc

The cipher text and hexdump of the same is as follows:

```
[jashjain@Jashs-MacBook-Air temp % openssl enc -aes-192-cbc -e -in hello.txt -out ciphered.bin -K 112233445566778899aabccddeff -iv 01020304050607080
[jashjain@Jashs-MacBook-Air temp % cat ciphered.bin
??j?j??
?4      ?a?(v?Ur??P????K??h&?w?Tyy$"?a?U3v??????>?P
r?8P1??63wd-??WI?o+n?jy????.yu??????F??
[      ??\?q????`T?.?P&?*?
jashjain@Jashs-MacBook-Air temp % hexdump ciphered.bin
00000000 a1 f1 d3 6a d3 ea 66 04 bb 7f 9d 49 0b 02 8a 61
00000010 cf 28 76 cd 55 72 11 1a df c9 50 fe 14 81 b9 91
00000020 4b c8 d4 68 26 bb 12 77 11 9e 54 79 79 24 22 b1
00000030 22 16 61 11 7f f0 aa 55 33 76 a9 bc a1 93 8e 89
00000040 3e 01 a3 50 0d c1 34 0b 72 ab cf a8 50 1d 31 a0
00000050 ab d7 a9 36 33 64 7e e6 03 9f f1 57 49 a8 6f 2b
00000060 6e ce 6a 79 a4 cd ce dd 2e 79 55 90 cd 16 c4 32
00000070 9b 65 07 ed c9 15 a3 a4 a7 ee 46 a0 02 cc 0c ab
00000080 87 5c dd 71 88 ad f4 60 54 d5 2e 84 50 26 fc 2a
00000090
jashjain@Jashs-MacBook-Air temp %
```

6. Encryption using aes-192-cfb

The cipher text and hexdump of the same is as follows:

```
[jashjain@Jashs-MacBook-Air temp % openssl enc -aes-192-cfb -e -in hello.txt -out ciphered.bin -K 112233445566778899aabccddeff -iv 01020304050607080
[jashjain@Jashs-MacBook-Air temp % cat ciphered.bin
'??????P;B$(p?0??)?E?T2\
?eG9?X?u????X??
?V:w????t?
?IC_?i?Dv?~?`#D"63@??+=?hd?@?d?C?g/F??I?#Y??o`?z?af??a??@%
jashjain@Jashs-MacBook-Air temp % hexdump ciphered.bin
00000000 27 c0 a4 b1 c9 11 86 18 81 39 15 2e 50 11 3b 42
00000010 b1 24 28 0e 70 f0 30 96 c5 29 b0 f8 45 ee 54 32
00000020 5c 0c 9d 40 56 3a 77 b3 fa d6 74 ee 0d a4 65 ce
00000030 08 47 39 dc 58 e6 c6 b0 b9 81 ca 58 fa b6 0b dc
00000040 c3 a4 7c bd 88 b2 0d 11 df a4 49 43 ea 5f 33 69
00000050 a4 44 76 98 7e 89 24 5e 44 22 36 33 40 ef 93 d7
00000060 2b 3d b7 68 64 97 e1 16 61 8e 64 fc 43 Bb 67 2f
00000070 0f 46 c6 e9 ba 49 ef 23 59 d3 10 f1 6f 60 cf 7a
00000080 01 8b c7 a3 66 96 d3 61 9f bc 40
0000008b
jashjain@Jashs-MacBook-Air temp %
```

7. Encryption using des-ecb

The cipher text and hexdump of the same is as follows:

```
[jashjain@Jashs-MacBook-Air temp % openssl enc -des-ecb -e -in hello.txt -out ciphered.bin -K 112233445566778899aabccddeff
[jashjain@Jashs-MacBook-Air temp % cat ciphered.bin
??????a??F?i??dpm)9????Z??}??9??q?}??\9&$j?..??
?l{^B?l??p?Q6Px????u?5??^?k?????3?;S}????7HPc*
[      ö?t?l?ý? [+c\?tXM??Iép???
jashjain@Jashs-MacBook-Air temp % hexdump ciphered.bin
00000000 8b 1c 1b 9a 9c db f0 bc f7 f0 c3 a0 d1 c0 46 0f
00000010 ac 69 85 eb a1 07 fe 64 70 6d 29 39 8d 07 fb ab
00000020 a5 5a b5 9c 7d 1d 9f 05 8c 39 f7 84 b8 71 e7 7d
00000030 ae c0 5c 39 26 24 da a8 a1 1e 2e 9b f1 0b 82 6c
00000040 7b 5e 42 f6 6c e8 d6 70 bf 51 d0 b1 50 78 89 b1
00000050 97 be 75 fe da 90 94 e4 5e 93 6b 15 b3 de e0 9e
00000060 d2 33 ef 3b 53 7d fe 00 b2 91 fe 37 48 1b 8b 50
00000070 63 2a 0c c3 b6 ab 74 fe 6c b7 d3 b2 93 5b 2b 63
00000080 5c 1b 85 f8 74 58 4d c1 db 49 ce ad 70 04 b4 de
00000090
jashjain@Jashs-MacBook-Air temp %
```


8. Encryption using des-cfb

The cipher text and hexdump of the same is as follows:

```
[jashjain@Jashs-MacBook-Air temp % openssl enc -des-cfb -e -in hello.txt -out ciphered.bin -K 112233445566778899aabcccddeeff -iv 01020304050607080
[jashjain@Jashs-MacBook-Air temp % cat ciphered.bin
_?5?8?ZP??Gf????V?^?F?Z??*??XQFD??3?D=?0?$$)h]?5|??Uj?oMpM<-?Gv??'
??k??9_?'???" 2??N??u??????rIm
6?<
r?=??x?)?u?%
[
jashjain@Jashs-MacBook-Air temp % hexdump ciphered.bin
00000000 5f ea 35 94 38 ae 5a 50 da d6 47 66 f8 c8 18 3f
00000010 e6 56 de 5e a2 10 46 c4 5a 96 b5 2a e8 c6 ec 58
00000020 1a 51 46 44 c1 c6 33 07 dc 44 3d b9 1b bc 4f df
00000030 e7 bd 24 1e 02 03 29 68 5d b1 35 7c 7c bf cc 55
00000040 13 6a ac 6f 4d 70 4d 3c 2d 99 47 17 76 e7 82 dd
00000050 27 0b 04 ff aa 6b fc e0 39 5f 13 dc ca bb fb 90
00000060 f6 22 20 32 b3 02 cb 4e f7 c0 75 d0 e8 f9 b2 be
00000070 c6 39 85 72 1c 04 49 6d 0b 13 36 8c 3c 0c 72 01
00000080 8c 3d aa e8 78 fa 29 f8 75 01 ba
0000008b
jashjain@Jashs-MacBook-Air temp % ]
```

9. Encryption using des-cbc

The cipher text and hexdump of the same is as follows:

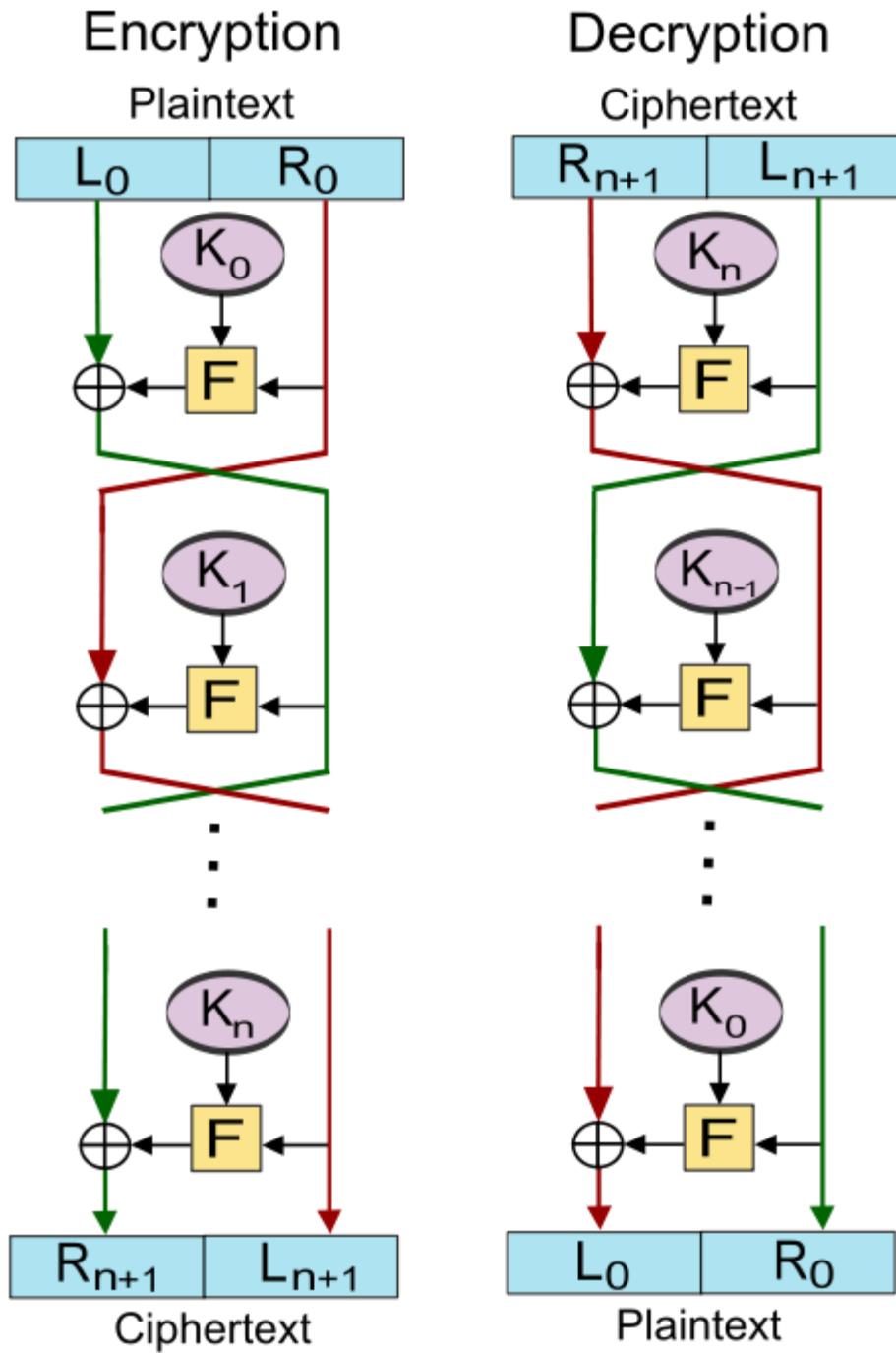
```
[jashjain@Jashs-MacBook-Air temp % openssl enc -des-cbc -e -in hello.txt -out ciphered.bin -K 112233445566778899aabcccddeeff -iv 01020304050607080
[jashjain@Jashs-MacBook-Air temp % cat ciphered.bin
?軌?
???
?q_>N??F'????-?w?/&?
~?9[???@$0?r?'q0xS
??$&l??!"N?}1C".??Da}X?;??
[E:??C?YG=?2????{\?\?f
jashjain@Jashs-MacBook-Air temp % cat ciphered.bin
?軌?
???
?q_>N??F'????-?w?/&?
~?9[???@$0?r?'q0xS
??$&l??!"N?}1C".??Da}X?;??
[E:??C?YG=?2????{\?\?f
jashjain@Jashs-MacBook-Air temp % ]
```

Observations:

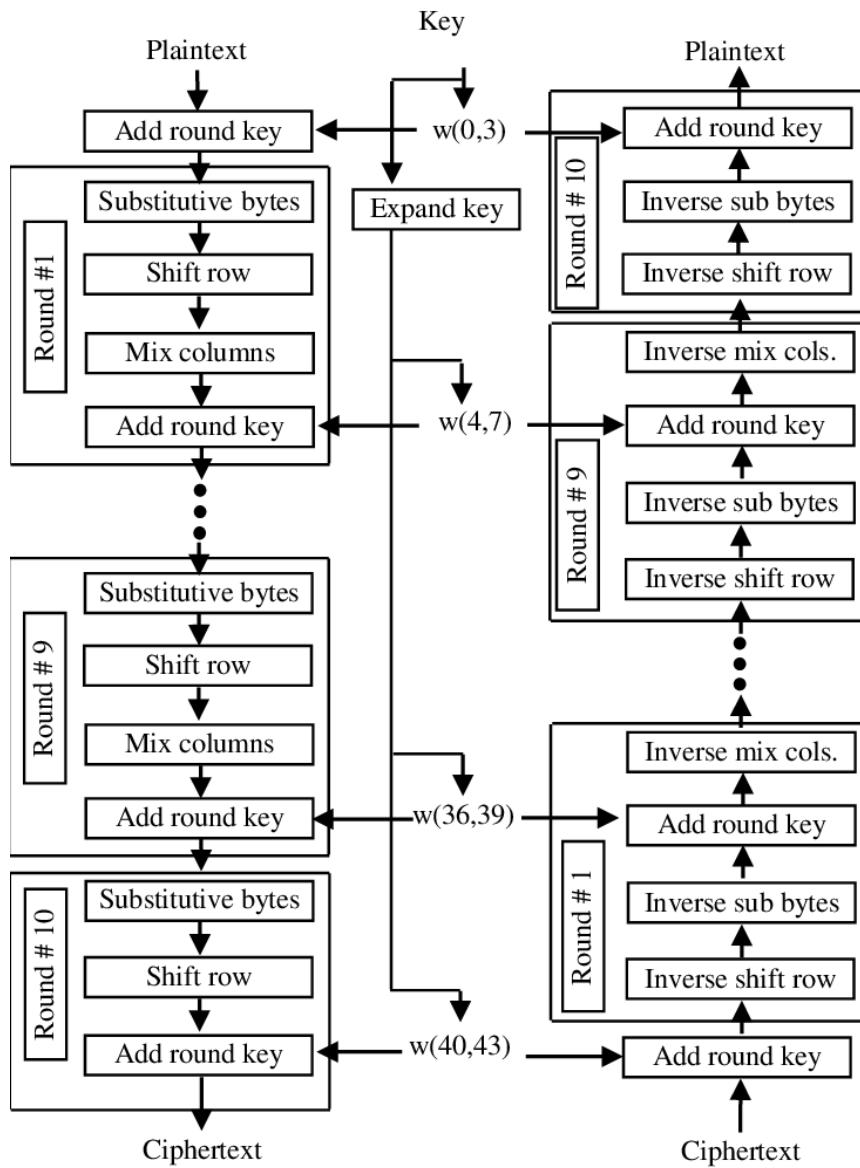
1. AES comprises a series of linked operations, some of which involve replacing inputs by specific outputs (substitutions) and others involve shuffling bits around (permutations). AES performs all its computations on bytes rather than bits. Hence, AES treats the 128 bits of a plaintext block as 16 bytes. AES uses 10 rounds for 128-bit keys, 12 rounds for 192-bit keys and 14 rounds for 256-bit keys.
2. AES adds more redundancy as compared to DES as length of key in AES has to be at least 128 and can go upto 256 bits However DES uses a 64-bit key 8×8 including 1 bit for parity, so the actual key is 56 bits
3. In terms of structure, DES uses the Feistel network(Fig 1.0) which divides the block into two halves before going through the encryption steps. AES(Fig 1.1) on the other hand, uses

permutation-substitution, which involves a series of substitution and permutation steps to create the encrypted block

Fiestel Network(Fig 1.0)



AES(Fig 1.1)



3.2 Task 2: Encryption Mode – ECB vs. CBC

The file pic original.bmp contains a simple picture. We would like to encrypt this picture, so people without the encryption keys cannot know what is in the picture. Please encrypt the file using the ECB (Electronic Code Book) and CBC (Cipher Block Chaining) modes, and then do the following:

1. Let us treat the encrypted picture as a picture, and use a picture viewing software to display it. However, For the .bmp file, the first 54 bytes contain the header information about the picture, we have to set it correctly, so the encrypted file can be treated as a legitimate .bmp file. We will replace the header of the encrypted picture with that of the original picture. You can use the ghex tool to directly modify binary files.
2. Display the encrypted picture using any picture viewing software. Can you derive any useful information about the original picture from the encrypted picture? Please explain your observations.

Original Picture(Fig 1.2)

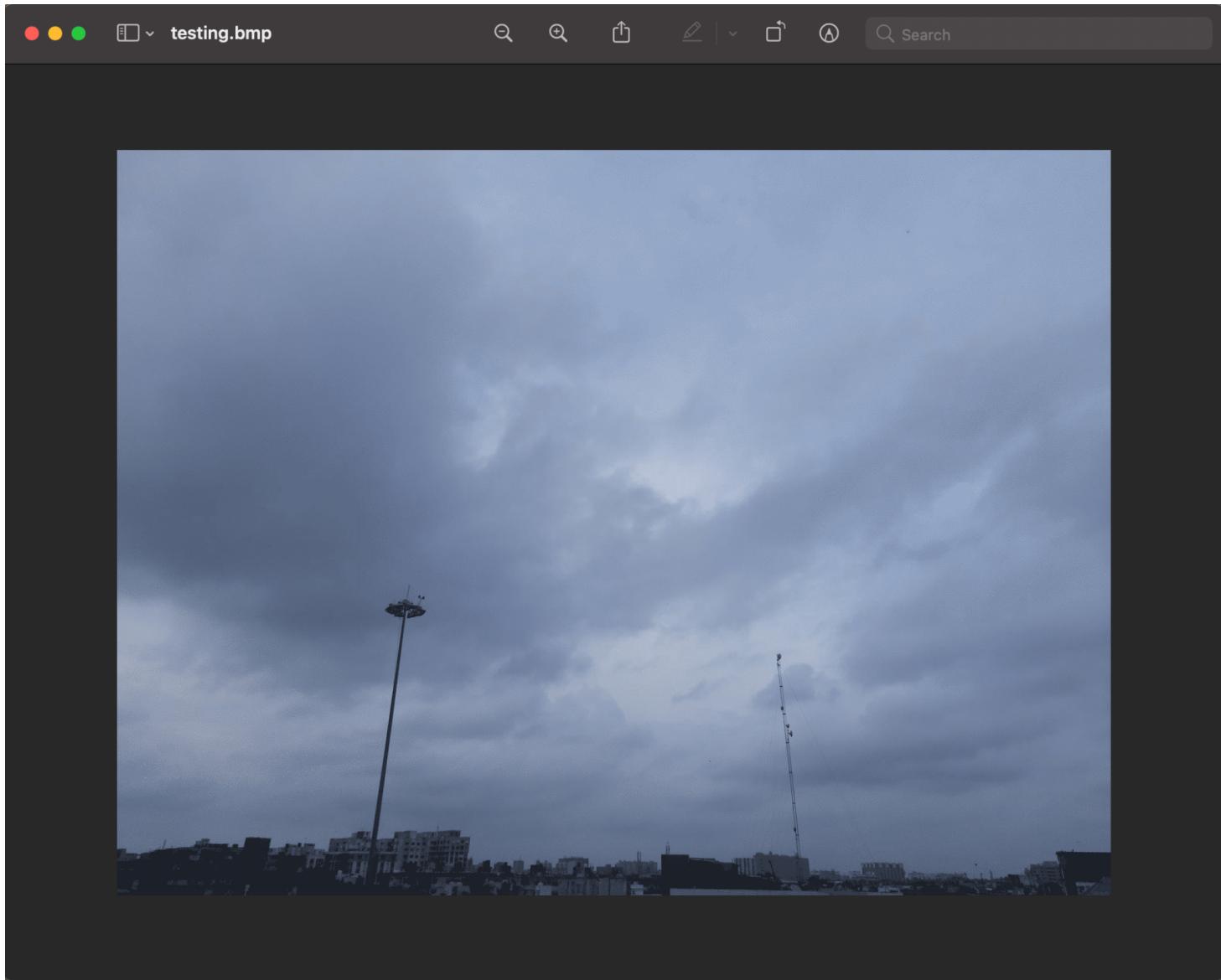


Image Encrypted using ecb(Fig 1.3)

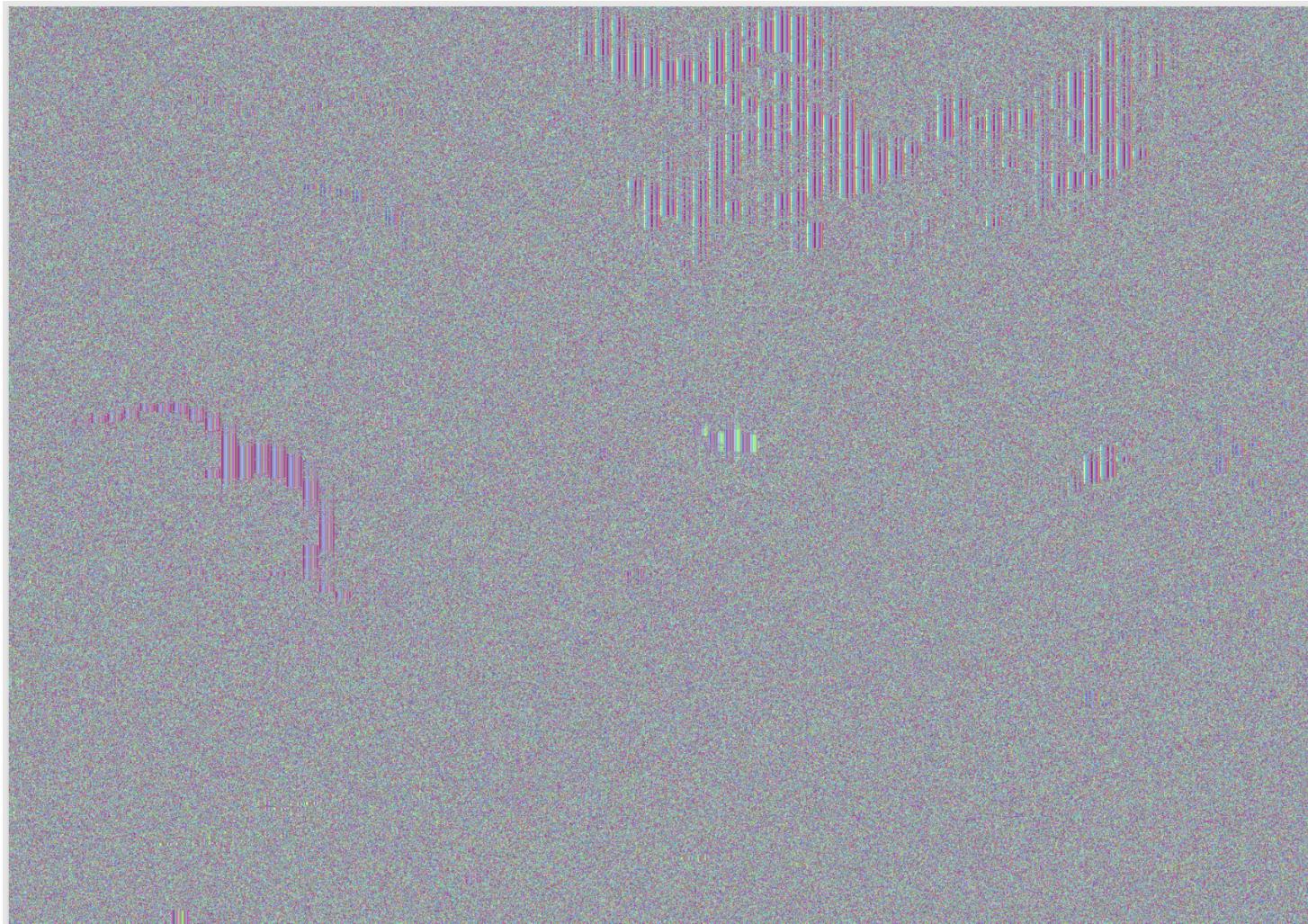
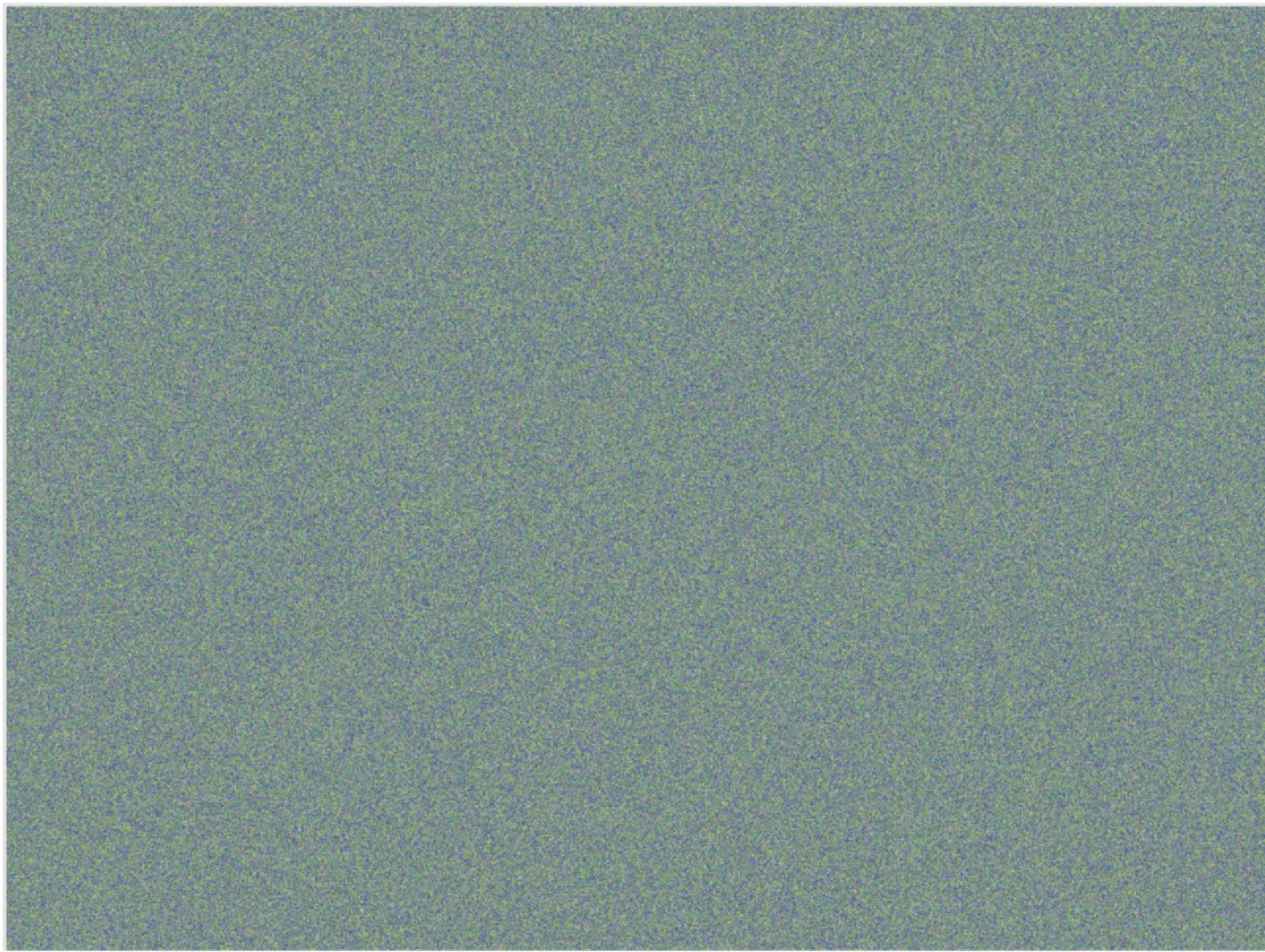
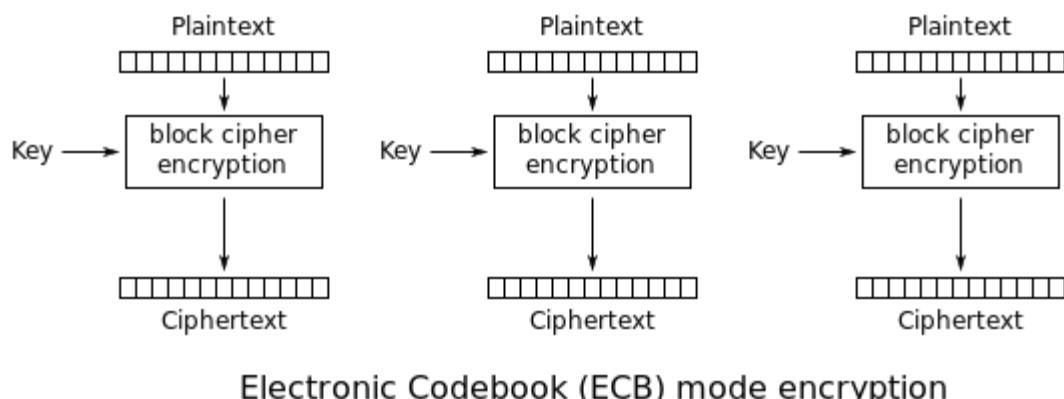


Image encryption using cbc (Fig 1.4)

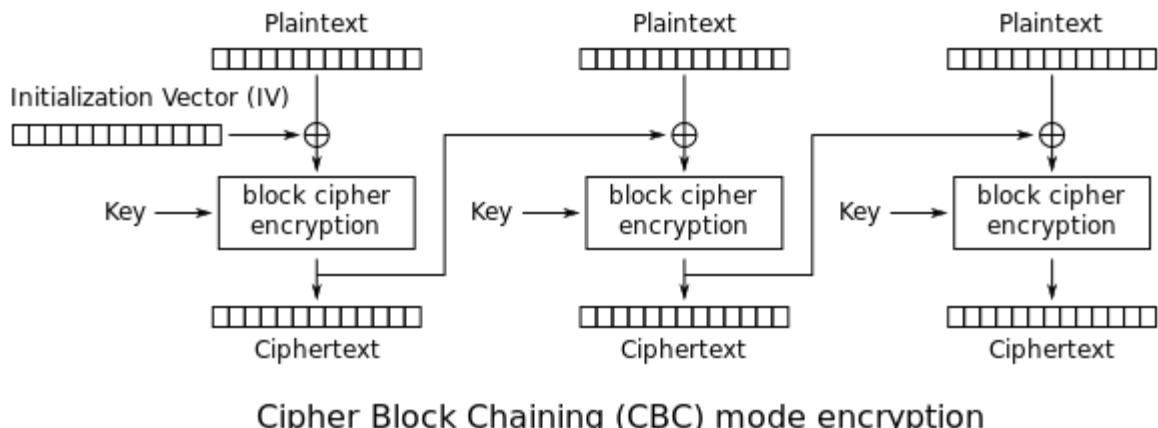


Observations:

- 1) In the image encrypted using ECB(Fig 1.3), we can still see a set of certain colors and shapes from which we can try to figure out what the original picture could be.
- 2) In the image encryption using CBC(Fig 1.4) the entire image is distorted and there is no way by which we can try to figure out the original picture.
- 3) The reason behind the visibility of the overall structure in the image encrypted using ECB is because we use the same key on each and every block and get the output for each of them.



- 4) In CBC, the overall structure of the original image is not transferred to the encrypted image. This means that it is impossible to guess the original image based on the encrypted image. This mainly happens because in CBC, we XOR every block with the last encrypted block and then encrypt the current block, this helps in breaking patterns in the data and makes it more secure.



3.3 Task 3: Encryption Mode – Corrupted Cipher Text

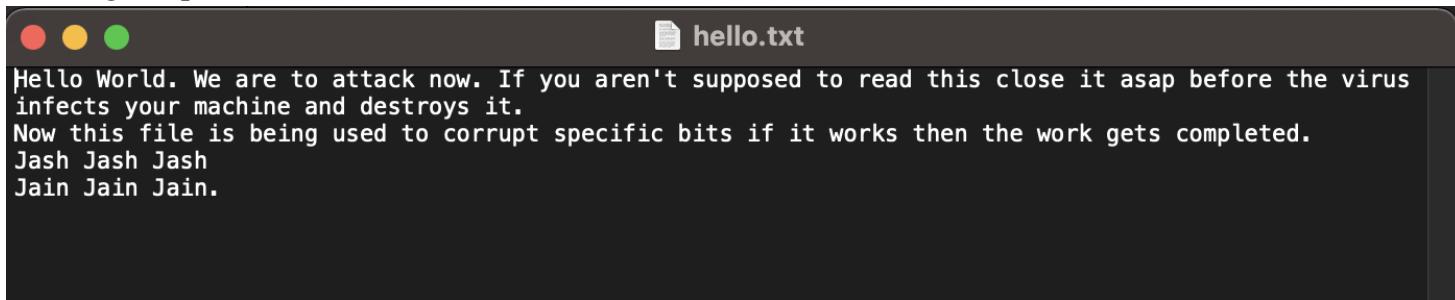
To understand the properties of various encryption modes, we would like to do the following exercise:

1. Create a text file that is at least 64 bytes long.
2. Encrypt the file using the AES-128 cipher.
3. Unfortunately, a single bit of the 30th byte in the encrypted file got corrupted. You can achieve this corruption using ghex.
4. Decrypt the corrupted file (encrypted) using the correct key and IV.

Please answer the following questions: (1) How much information can you recover by decrypting the corrupted file, if the encryption mode is ECB, CBC, CFB, or OFB, respectively? Please answer this question before you conduct this task, and then find out whether your answer is correct or wrong after you finish this task.

(2) Please explain why. (3) What are the implications of these differences?

The original plain text is as follows:



A screenshot of a terminal window titled "hello.txt". The window contains the following text:
Hello World. We are to attack now. If you aren't supposed to read this close it asap before the virus
infects your machine and destroys it.
Now this file is being used to corrupt specific bits if it works then the work gets completed.
Jash Jash Jash
Jain Jain Jain.

The above text is encrypted using aes-128 in 4 different modes: cbc,cfb,ofb and ecb. After corrupting 30th byte of each cbc, cfb, ofb and ecb :

ECB Corruption of file

			cip_ecb.bin
0	359017A6 BA4AB080 EBB6291C 1DBF02A4	5. . .J.....) . . .	
16	2F516826 BE9234DD 05A23E3E 1301371C	/Qh&..4. .>> 7	
32	363DDBBE DF4F5D0D DB919C8A 52998470	6=...0]R..p	
48	E4789AFB 527EB8A2 7259A50B 7902775C	.x..R~..rY. y w\	
64	89DF1C67 3EAFF2DF D477AEFB 169A0315	.. g>....w.. .	
80	393F8BAD E42A9847 9DE8EE49 79437C95	9?...*.G...IyCl.	
96	C5D19CEE 4FF33FEF 39015A83 53E961250.?..9 Z.S.a%	
112	2F721007 CD3E36ED 62404B35 3816D0B8	/r .>6.b@K58 ..	
128	2EE6A312 A5A9540C E2E038C6 8D605992T ..8..`Y.	
144	52AD3779 B86B3EE3 42A6B8A1 0FCC2F94	R.7y.k>.B... ./.	
160	FA0ECE0E 82C5961A DC79594C E3AD7BD9yYL..{.	
176	3D2747CA FD0EBD4E 903C883C FEB07703	='G.. .N.<.<..w	
192	C1A2A291 01A52730 33AF1D1E 1338DC86'03. 8..	
208	E8412EBA DD0FDBD8 7A08C6B2 FFFE137A	.A... .z z	
224	42B65B17 9FF00201 045E2057 BB25E6B5	B.[.. ^ W.%..	
240	05CE7C1E CA40F240 76282FC5 5151F36F	.l .@. @v(./.QQ.o	
256	D9CDD12C D3EBEEBA 9C48E79E D8F89B24	...,.....H.....\$	
272			

Signed Int



le, dec

(select some data)



30 out of 272 bytes

CFB Corruption of file

File: cip_cfb.bin

	Hex	ASCII
0	21D0668F 8010222C 469D34F3 391CD9DA	!.f.. ", F.4.9 ..
16	E243E423 89639647 FD86BE83 270D4B4D	.C.#.c.G....' KM
32	4C4F6575 A00B5D2F FD242085 A451082A	L0eu.]/.\$. .Q *
48	B3A089A5 12FED470 7DFF0AD2 8B09982Cp}. . . .,
64	122FD497 30ED3942 D54F8896 FE5B7617	/..0.9B.0...[v
80	BAE396A9 15E41D4B 2F0004F1 285EB923 K/ .(^.#
96	FD1F1DB2 D6ED0C23 B34AB9C6 761291A3 #.J..v ..
112	EE081678 CCB435DD 5BDA8307 6EA9E187	. x..5.[.. n...
128	BF9C66FD 359693B9 7499F5E9 240320A1	.. f.5...t...\$..
144	EFE48741 1EDBEB39 3CDBF5F9 A5DE1081	...A ..9<..... .
160	BF8C12D2 D25B2947 8DA71F25 152A6D4B [)G.. % *mK
176	46954ECF B6114575 B958BA41 08113155	F.N.. Eu.X.A 1U
192	407DBCE6 158082DE 574847F5 699A80E8	@}.. . . .WHG.i...
208	03B02E4B 9B93050C 8DDF5123 A0CD13D7	.. K.. . . Q#.. .
224	ED73EB50 AD9C15EF 343B13E8 A6AA8C90	.s.P.. .4;
240	1CA52ABF 11906AE9 3E70CC2D B43B0120	.*. .j.>p.-.;
256	7FCF7937 9E628FE4 71	.y7.b..q

Signed Int ▾ le, dec (select some data) - +

265 out of 265 bytes

CFB Corruption of File

File: **cip_cbc.bin**

	Hex	ASCII
0	75E2DF7D 48F9987C C3D9929F F55F5968	u..}H..!.....Yh
16	35F917C1 4173F4A7 067E4E49 4C0B6195	5. .As.. ~NIL a.
32	5706A9B2 717AB2C6 F3F7C2A8 3D61256C	W ..qz.....=a%l
48	3119F962 E1069E13 D00AA915 25A35A11	1 .b. . . . % .Z
64	631D09B9 748D8EAA 1E06BD25 F8D76899	c .t... .%.h.
80	30570B99 B5B78E25 827BCEE1 578A4320	0W%.{..W.C
96	9D30F50B F683B765 CFCF84CB C7AA26E0	.0.e.....&.
112	E92C68AA 8EED93AA EBE83135 996CDDEE	. ,h.....15.l..
128	DD27F812 28CFAB24 8015B623 6EEAE05E	.'. (.\$. .#n..^
144	9BBF525D 8169234A C5992A11 B0102C18	. .R].i#J..* . ,
160	FC5DECA3 0EE2EF5E 46C4D345 AFB5A39D	.].. ..^F..E....
176	3E7261C7 3DC5316C E0ACB413 F0B1298F	>ra.=.1l... .).
192	C592E6A0 F132673D F30D8141 CD94A7A22g=. .A....
208	7C1DAE85 4316C6EB 7942A70A 92141F7C	..C ..yB. .
224	2D077346 0FE0DA82 0082E181 52736765	- sFRsge
240	97281A9A DFAF2388 83939A29 823F9EC5	.(...#....).?..
256	31631526 151887AA B78BBF7C C042539C	1c &l.BS.
272		

Signed Int le, dec (select some data)

30 out of 272 bytes

OFB Encryption of File

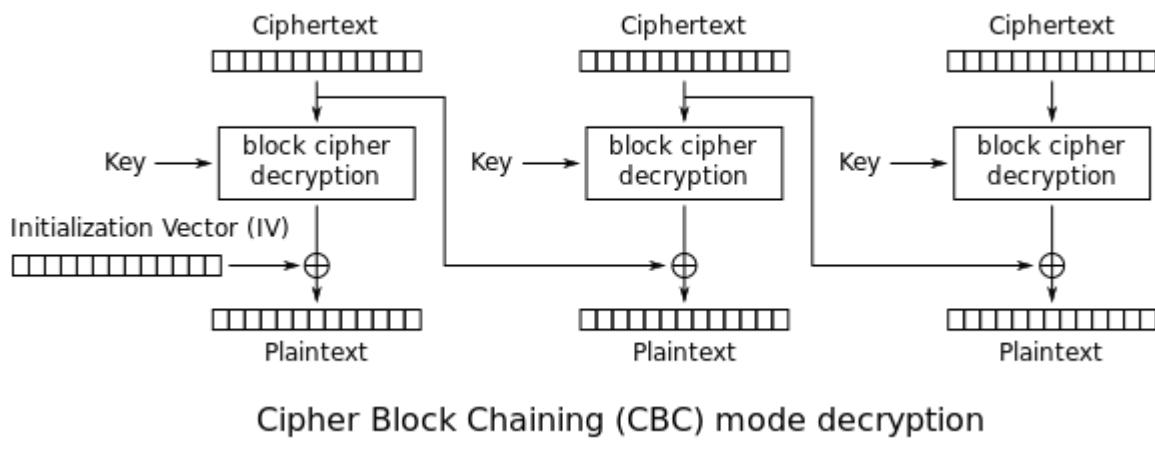
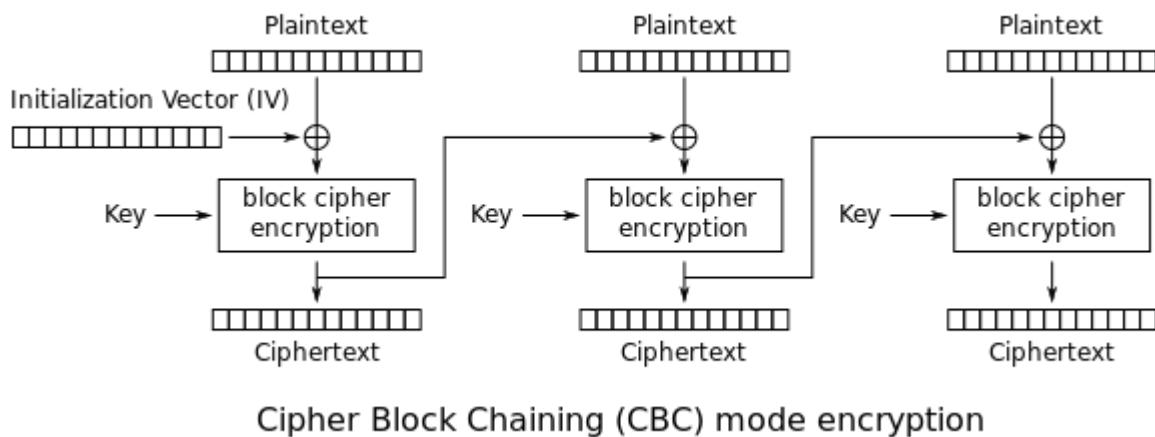
			cip_ofb.bin
0	21D0668F	8010222C	469D34F3 391CD9DA !.f.. ",F.4.9 ..
16	39288CA2	DE09FC96	9B12225E DD04F5E3 9C... ... "^. ..
32	9DA931CD	B8CBB535	F344E7E4 1D2F9BD0 ..1....5.D.. /..
48	C9552101	C56ABCCA	03A19E73 DB3C160D .U! .j.. .s.<
64	CA86F365	B0C7798C	E99EF238 8D0C3E4C ...e...y....8. >L
80	71661B05	FC16EC83	70983F42 3C3C10C8 qf . .p.?B<< .
96	3E733150	AAF98072	886653F9 C50C82D6 >s1P...r.fS.. ..
112	BFE1CA4C	FF889EEA	645D3E71 EDAE8E79 ...L....d]>q...y
128	D0BB8109	EBCCA9D2	A8FDCAEA 58B0E20AX..
144	7298F750	D0BD682F	94DA5D35 FC798246 r..P..h/..]5.y.F
160	A1AF7ACE	5CA9D237	6F1F52A6 588FA968 ..z.\..7o R.X..h
176	3495677B	32BB27A8	44E30148 9D3B863A 4.g{2.'..D. H.;..
192	0880FA8F	7D2A6C0E	4EDF18D9 A40E479C ...}*l N. .. G.
208	FF381A0E	96EF898C	2211AB47 B969A728 .8" .G.i.(
224	0391D871	F98C0204	648895C9 759B44EC ..q.. d...u.D.
240	60A024C3	42B735BE	77BA2BE4 01C4FFE2 `.\$.B.5.w.+. ...
256	3991E656	9C706C86	B5 9..V.pl..

The decrypted text for the same is as follows:

CBC

```
(base) jashjain@Jashs-MacBook-Air temp % openssl enc -aes-128-cbc -d -in cip_cbc.bin -out cbc.txt -K 112233445566778899aabcccddeeff -iv 01020304050607080  
hex string is too short, padding with zero bytes to length  
hex string is too short, padding with zero bytes to length  
Hello World. We r??!???vr??w. If you are/'t supposed to read this close it asap before the virus infects your machine and destroys it.  
Now this file is being used to corrupt specific bits if it works then the work gets completed.  
Jash Jash Jash  
Jain Jain Jain.%  
(base) jashjain@Jashs-MacBook-Air temp %
```

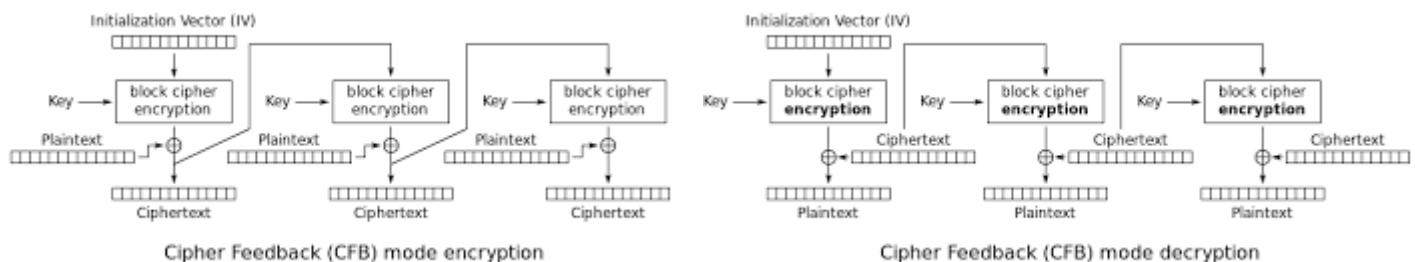
In CBC every encrypted block cipher depends on the previous encrypted block cipher. However while decrypting, we perform XOR on the decrypted block by using the previous cipher text. Hence a one-bit change to the ciphertext causes complete corruption of the corresponding block of plaintext, and inverts the corresponding bit in the following block of plaintext, but the rest of the blocks remain intact.



CFB

```
(base) jashjain@Jashs-MacBook-Air temp % openssl enc -aes-128-cfb -d -in cip_cfb.bin -out cfb.txt -K 112233445566778899aabcccddeeff -iv 01020304050607080  
hex string is too short, padding with zero bytes to length  
hex string is too short, padding with zero bytes to length  
Hello World. We are to attack?no|?u:H?????Py?? supposed to read this close it asap before the virus infects your machine and destroys it.  
Now this file is being used to corrupt specific bits if it works then the work gets completed.  
Jash Jash Jash  
Jain Jain Jain.%  
(base) jashjain@Jashs-MacBook-Air temp %
```

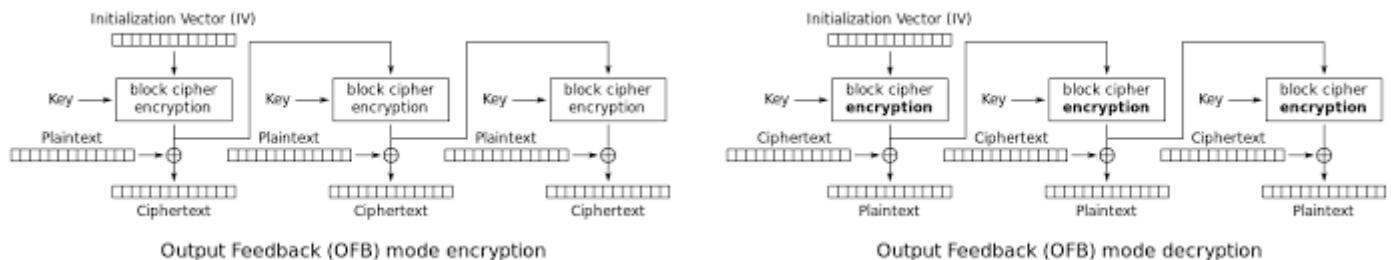
Loss in CFB is much less as compared to ECB and CBC. This is because the same encryption algorithm is used for decryption and the corrupted ciphertext for a block is used for two corresponding blocks hence limiting the damage to two blocks of cipher text.



OFB

```
jashjain@Jashs-MacBook-Air temp % openssl enc -aes-128-ofb -e -in hello.txt -out cip_ofb.bin -K 00112233445566778889aabccddeff -iv 0102030405060708
[jashjain@Jashs-MacBook-Air temp % openssl enc -aes-128-ofb -d -in cip_ofb.bin -out ofb.txt -K 00112233445566778889aabccddeff -iv 0102030405060708
Hello World. We are to attacknow. If you aren't supposed to read this close it asap before the virus infects your machine and destroys it.
Now this file is being used to corrupt specific bits if it works then the work gets completed.
Jash Jash Jash
Jain Jain Jain.% 
jashjain@Jashs-MacBook-Air temp %
```

The data lost in OFB is the least as compared to other block cipher modes. This is mainly because flipping a bit in the ciphertext produces a flipped bit in the plaintext at the same location. This property allows many error-correcting codes to function normally even when applied before encryption.

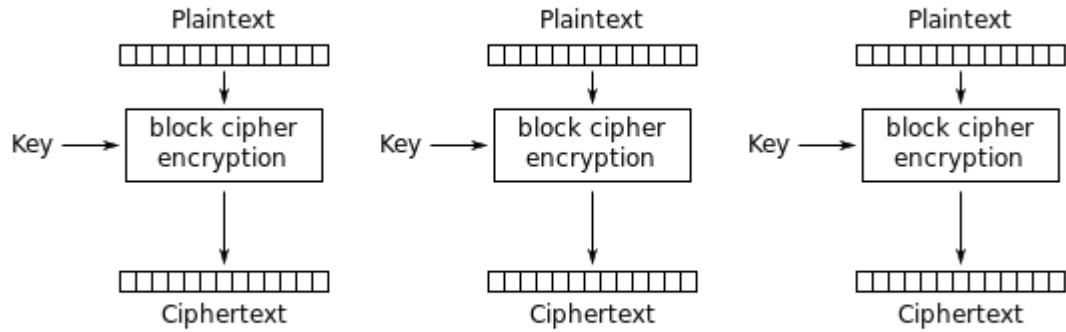


ECB

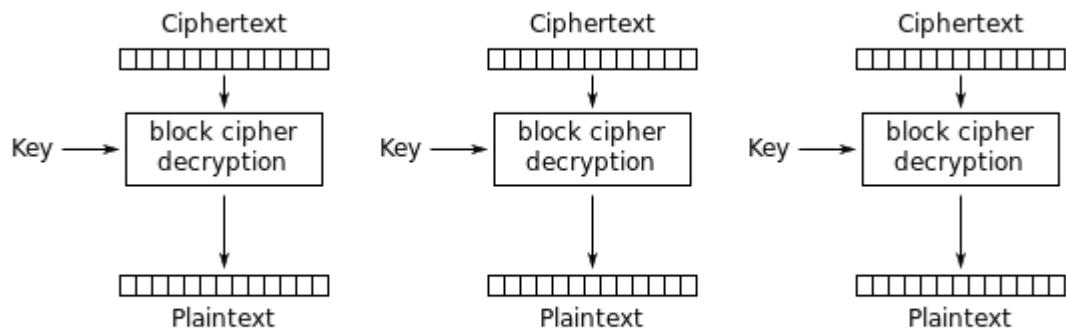
```
(base) jashjain@Jashs-MacBook-Air temp % openssl enc -aes-128-ecb -d -in cip_ecb.bin -out ecb.txt -K 112233445566778899aabccddeff
hex string is too short, padding with zero bytes to length
(base) jashjain@Jashs-MacBook-Air temp % cat ecb.txt
Hello World. We )'?@???????.Rw. If you aren't supposed to read this close it asap before the virus infects your machine and destroys it.
Now this file is being used to corrupt specific bits if it works then the work gets completed.
Jash Jash Jash
Jain Jain Jain.% 
(base) jashjain@Jashs-MacBook-Air temp %
```

From the results above I can conclude that:

In ECB mode, since every block cipher is encrypted separately using the key, when one byte is corrupted and then decrypted, it only affects the corresponding block of data while the other blocks of data are unaffected.



Electronic Codebook (ECB) mode encryption



Electronic Codebook (ECB) mode decryption

3.4 Task4 : Padding

For block ciphers, when the size of the plain text is not the multiple of the block size, padding may be required. In this task, we will study the padding schemes. Please do the following exercises:

The openssl manual says that openssl uses PKCS5 standard for its padding. Please design an experiment to verify this. In particular, use your experiment to figure out the paddings in the AES encryption when the length of the plaintext is 20 octets and 32 octets.

1. Please use ECB, CBC, CFB, and OFB modes to encrypt a file (you can pick any cipher). Please report which modes have paddings and which ones do not. For those that do not need paddings, please explain why.

```
[(base) jashjain@Jashs-MacBook-Air temp % cd padding
[(base) jashjain@Jashs-MacBook-Air padding % ls -l
total 16
-rw-r--r--@ 1 jashjain  staff  32 Nov  8 07:23 large.txt
-rw-r--r--@ 1 jashjain  staff  20 Nov  8 07:24 small.txt
(base) jashjain@Jashs-MacBook-Air padding %
```

Two files namely large.txt and small.txt are made with each having 32 and 20 bytes respectively. The files are encrypted using aes-128 in 4 modes namely: ecb, cbc, cfb, ofb

Fig(1.4)

```
[(base) jashjain@Jashs-MacBook-Air padding % ls -l
total 80
-rw-r--r--  1 jashjain  staff  32 Nov  8 07:26 cbc_20.txt
-rw-r--r--  1 jashjain  staff  48 Nov  8 07:27 cbc_32.txt
-rw-r--r--  1 jashjain  staff  20 Nov  8 07:26 cfb_20.txt
-rw-r--r--  1 jashjain  staff  32 Nov  8 07:26 cfb_32.txt
-rw-r--r--  1 jashjain  staff  32 Nov  8 07:28 ecb_20.txt
-rw-r--r--  1 jashjain  staff  48 Nov  8 07:28 ecb_32.txt
-rw-r--r--@ 1 jashjain  staff  32 Nov  8 07:23 large.txt
-rw-r--r--  1 jashjain  staff  20 Nov  8 07:27 ofb_20.txt
-rw-r--r--  1 jashjain  staff  32 Nov  8 07:27 ofb_32.txt
-rw-r--r--@ 1 jashjain  staff  20 Nov  8 07:24 small.txt
(base) jashjain@Jashs-MacBook-Air padding %
```

1. The cipher modes which require encryption are CBC and ECB as we can see that for small as well as large text files, it is adding padding and hence increasing the size of the file to be an exact multiple of the block size, which is 16 Bytes for AES.
2. We can confirm that the padding standard is PKCS5 because, whenever a mode which requires padding is used, PKCS5 adds at least 1 Byte of padding and at most n Bytes of padding where n is the block size. In short, it adds padding to the file so that its size becomes the next closest multiple of block size.
3. We can observe this in figure 1.4, as the size of large.txt is increased to 48 Bytes (next multiple of 16 after 32) and 32 Bytes (next multiple of 16 after 20) for small.txt for CBC and ECB
4. We can observe this in figure 1.4, as the size of large.txt and small.txt for CFB and OFB remains the same as default(32 bytes and 20 bytes respectively) since they are stream ciphers and do not require the size of file to be an exact multiple of block size.

To confirm that openssl uses PKCS5 padding, decrypt the encrypted file with option–nopad. This option turns off the standard block padding. Normally, the padding is included by default during encryption, so if I use the nopad option, I can see the padding in the decrypted file

Finally we can see the decrypted and encrypted files as follows:

Fig(1.5)

```
jash Jain@Jashs-MacBook-Air padding % openssl enc -aes-128-cbc -d -nopad -in cbc_20.txt -out cbc_20_dec.txt -K 00112233445566778899aabcccddeeff -iv 010203040506070809
jash Jain@Jashs-MacBook-Air padding % openssl enc -aes-128-cbc -d -nopad -in cbc_32.txt -out cbc_32_dec.txt -K 00112233445566778899aabcccddeeff -iv 010203040506070809
jash Jain@Jashs-MacBook-Air padding % openssl enc -aes-128-ofb -d -nopad -in ofb_32.txt -out ofb_32_dec.txt -K 00112233445566778899aabcccddeeff -iv 010203040506070809
jash Jain@Jashs-MacBook-Air padding % openssl enc -aes-128-ofb -d -nopad -in ofb_20.txt -out ofb_20_dec.txt -K 00112233445566778899aabcccddeeff -iv 010203040506070809
jash Jain@Jashs-MacBook-Air padding % openssl enc -aes-128-cfb -d -nopad -in cfb_20.txt -out cfb_20_dec.txt -K 00112233445566778899aabcccddeeff -iv 010203040506070809
jash Jain@Jashs-MacBook-Air padding % openssl enc -aes-128-cfb -d -nopad -in cfb_32.txt -out cfb_32_dec.txt -K 00112233445566778899aabcccddeeff -iv 010203040506070809
jash Jain@Jashs-MacBook-Air padding % openssl enc -aes-128-ecb -d -nopad -in ecb_20.txt -out ecb_20_dec.txt -K 00112233445566778899aabcccddeeff
jash Jain@Jashs-MacBook-Air padding % openssl enc -aes-128-ecb -d -nopad -in ecb_32.txt -out ecb_32_dec.txt -K 00112233445566778899aabcccddeeff
jash Jain@Jashs-MacBook-Air padding % ls -l
total 144
-rw-r--r-- 1 jash Jain staff 32 Nov 8 10:45 cbc_20.txt
-rw-r--r-- 1 jash Jain staff 32 Nov 8 10:48 cbc_20_dec.txt
-rw-r--r-- 1 jash Jain staff 48 Nov 8 10:48 cbc_32.txt
-rw-r--r-- 1 jash Jain staff 48 Nov 8 10:48 cbc_32_dec.txt
-rw-r--r-- 1 jash Jain staff 20 Nov 8 10:44 cfb_20.txt
-rw-r--r-- 1 jash Jain staff 20 Nov 8 10:49 cfb_20_dec.txt
-rw-r--r-- 1 jash Jain staff 32 Nov 8 10:44 cfb_32.txt
-rw-r--r-- 1 jash Jain staff 32 Nov 8 10:49 cfb_32_dec.txt
-rw-r--r-- 1 jash Jain staff 32 Nov 8 10:46 ecb_20.txt
-rw-r--r-- 1 jash Jain staff 32 Nov 8 10:50 ecb_20_dec.txt
-rw-r--r-- 1 jash Jain staff 48 Nov 8 10:46 ecb_32.txt
-rw-r--r-- 1 jash Jain staff 48 Nov 8 10:50 ecb_32_dec.txt
-rw-r--r--@ 1 jash Jain staff 32 Nov 8 07:23 large.txt
-rw-r--r-- 1 jash Jain staff 20 Nov 8 10:46 ofb_20.txt
-rw-r--r-- 1 jash Jain staff 20 Nov 8 10:49 ofb_20_dec.txt
-rw-r--r-- 1 jash Jain staff 32 Nov 8 10:46 ofb_32.txt
-rw-r--r-- 1 jash Jain staff 32 Nov 8 10:49 ofb_32_dec.txt
-rw-r--r--@ 1 jash Jain staff 20 Nov 8 07:24 small.txt
jash Jain@Jashs-MacBook-Air padding %
```

The screenshot above shows that the size of CBC and ECB encrypted files with the nopad option is 12

bytes more for the 20 bytes file and 16 bytes larger for the 32 bytes file, however the size of OFB and CFB decrypted files is the same.

On decryption, by default PKCS5 padding is expected to be present and is removed; if the padding is not valid an error occurs.

-nopad means no attempt is made to remove padding; if padding is present, in your ciphertext, it is left in the data.

If encrypt was done (successfully) with -nopad you need -nopad on decrypt so you don't try to remove padding that isn't there and fail.

Result:

1. The experiment shows that padding is needed for ECB and CBC encryption modes. This can be because ECB and CBC are block ciphers and for a block cipher length of input must be an exact multiple of block length. If this is not the case then padding must be added to make it so. This padding is removed after decrypting.
2. In OFB and CFB, the padding is not required because they are stream ciphers and the ciphertext is always the same length as plain text.

3.5 Task 5: Programming using the Crypto Library

So far, we have learned how to use the tools provided by openssl to encrypt and decrypt messages. In this task, we will learn how to use openssl's crypto library to encrypt/decrypt messages in programs. OpenSSL provides an API called EVP, which is a high-level interface to cryptographic functions. Although OpenSSL also has direct interfaces for each individual encryption algorithm, the EVP library provides a common interface for various encryption algorithms. To ask EVP to use a specific algorithm, we simply need to pass our choice to the EVP interface. A sample code is given in http://www.openssl.org/docs/crypto/EVP_EncryptInit.html. Please get yourself familiar with this program, and then do the following exercise.

You are given a plaintext and a ciphertext, and you know that aes-128-cbc is used to generate the ciphertext from the plaintext, and you also know that the numbers in the IV are all zeros (not the ASCII character '0'). Another clue that you have learned is that the key used to encrypt this plaintext is an English word shorter than 16 characters; the word that can be found from a typical English dictionary. Since the word has less than 16 characters (i.e. 128 bits), space characters (hexadecimal value 0x20) are appended to the end of the word to form a key of 128 bits. Your goal is to write a program to find out this key. You can download an English word list from the Internet. We have also linked one on the web

Plaintext (total 21 characters): This is a top secret. Ciphertext
(in hex format): 8d20e5056a8d24d0462ce74e4904c1b5

~~13e10d1df4a2ef2ad4540fae1ca0aa19~~

page of this lab. The plaintext and ciphertext is in the following:

Note 1: If you choose to store the plaintext message in a file, and feed the file to your program, you need to check whether the file length is 21. Some editors may add a special character to the end of the file. If that happens, you can use the ghex tool to remove the special character.

Note 2: In this task, you are supposed to write your own program to invoke the crypto library. No credit will be given if you simply use the openssl commands to do this task.

Note 3: To compile your code, you may need to include the header files in openssl, and link to openssl libraries. To do that, you need to tell your compiler where those files are. In your Makefile, you may want to specify the following:

```
INC=/usr/local/ssl/
include/
LIB=/usr/local/ssl/
lib/
```

```
all:
gcc -IS(INC) -LS(LIB) -o enc yourcode.c -lcrypto -lssl
```

Code:

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

plain_text = b"This is a top secret."
cipher_hex = "8d20e5056a8d24d0462ce74e4904c1b513e10d1df4a2ef2ad4540fae1ca0aaf9"

result_k = []
file = open('words.txt', 'r')
lines = file.readlines()
words = [str.strip(line) for line in lines]
for word in words:
    if len(word) >= 16:
        continue
    word = word.lower()
    key = word.encode() + b' '*(16-len(word))
    cipher = AES.new(key, AES.MODE_CBC, iv=bytes.fromhex('0'*32))
    ciphertext = cipher.encrypt(pad(plain_text, AES.block_size))
    if bytes.hex(ciphertext) == cipher_hex:
        result_k.append(word)
print(result_k)
```

Fig 1.6

~/Desktop/college/sem5/LAB/temp/last ---zsh

```
zulu 3090ef5bf9a986ce0c28a3d35d516c22377c124b6eea579cb586049352ff2978 NOT MATCHED
zuludom 060bb5ebbla0719f034c6c2703c23d931e0736ef97773227f81eb0279b10477c NOT MATCHED
zuluize af315a33460644b5a4cd4029a5b52dbb1b37098f42277fc2e1c459ddca1e8cdd NOT MATCHED
zulu-kaffir aeaba7cb7a3525ca1eb47a9c088e32973799088c36483ad468c52df00426ff8d NOT MATCHED
zululand b47603dcf503f3374dad30284778687553be77433abb89491fcfa564f33a462 NOT MATCHED
zulus f92bdd505e2cc13b5201f62d2f5181cd49ec7add0ebf3ded60ef228276dd901e NOT MATCHED
zumatic a5d3d18290eabae7a1e96a2507100f9a42465718f150edd9dd9b57f02697d36 NOT MATCHED
zumbooruk 7a4058623d974edb22703e1bc4de6a631127d3441d3c5fc7d66a8ed5dd001928 NOT MATCHED
zumbrota d665978da1f581f6e120f99500e7c0b19739b1d74e1a696c487196e35b35e1f4 NOT MATCHED
zumstein ff4a890cd2d77d0646ea154352e24ccbf67b143e134c4850fb6b7cd50229de81 NOT MATCHED
zumwalt dc86c3624570377ac53b1f7836790a165a124592c07f940d3581083e5 NOT MATCHED
zungaria 337263f4915e715e79a317b01aa9728d55ff98a103df33737e7837b3937c70c NOT MATCHED
zuni 2fc23a78fef6f9ddb90ac09b16ebfc77117e62cee6e01e4baec5558ece3b8079 NOT MATCHED
zunian d71f79cb186995854dead267405b3fc52658fd409b9a37ca3c48bc360a9f84 NOT MATCHED
zunyite 97ad3e3b425b54831fc321777dcdfb6139cb5b63a7b7a187b39f797c2d546c6 NOT MATCHED
zunis 02e18da9f296811acee8e538a9b6b81025ecd8af352867727ddfeafab7e83f2 NOT MATCHED
zupanate 0192f3b5f3f8c4f0e76bd2e0ca6dedae0e4bd4d8108573c5b5dd54205386d202 NOT MATCHED
zupus 32fed27e04e33a0e90f133ebd2016829566632ae256cad419db240a41f4 NOT MATCHED
zurbar 0068cbada380c0c9b135541bb7e319646115e9a1820c4a7cf5cbe27deb310d6c NOT MATCHED
zurbaran 47d5dff9cd8903fc457817b3449f59e52f8ba4a478cc7b47daecd1e1fbf31db NOT MATCHED
zurek 16eb55ab1034690c5cb17a4e6e0296bcc61f277e4dcbb1c46a87a946dd152ae6 NOT MATCHED
zurheide 9d5e3c8c2ffcc28c5573126526c540949b3445fb794ee491884792645c5f4e6 NOT MATCHED
zurich 138ab3471de32ac3ba9529f86cbd388f8d036075578768f35468f3fc91d1f5a6 NOT MATCHED
zurkow 3e7f148cebba64d1e6bb6f9abcaceab5eaef965b0ed9def8fc848c0b738c5f0d4 NOT MATCHED
zurlite 336080c6a50b3eb47f6d1a7bbd702a9fd6b122414942c14bc77915bed77615 NOT MATCHED
zurn 02474b9a2fc07dc7e5d6eb6a1ec2a088b9914ef5e566cf9d6d08c031c837f8 NOT MATCHED
zurvan 6b3e75619622e2f9afc4068b49bcc8311ace4b7b9a839eb81b98eb6322a339e7 NOT MATCHED
zusman 8cbd1f67bf3a5c87321c24ba468e06d0d609d733dc89dcde56ae0a43c496b NOT MATCHED
zutugil f5b9c23ad3645da2bda13e1090cf84d2b5e945f9635a522957e4a2316940c36 NOT MATCHED
zuurveldt 7c76dcc12c049ed0ea34aaffbd02f6963fa8117c1382c9978a205a1e71b0d22 NOT MATCHED
zuza 49259e25ddfdeac0b070282f10d98b25bbdfc97080b1d85a3c6bfb1e40d440a08 NOT MATCHED
zuzana 4b317a3a18dd02ac63191e6aae08397a1507f6f861a0e9e342930278d26b24bc NOT MATCHED
zu-zu 04eb3d47bcc2fce2df97928758cd418272fdbc4866ffd4ed6227c80ad861f8ef NOT MATCHED
zwanziger 70b52532147330fb9b718d64b0cd144300ad9e4fe5ab875cef8ac74d587621ff NOT MATCHED
zwart 7b8693c89e594c35fec42bd33ec0f76a318e4c6096c8fb73b0a65346d7ee79627 NOT MATCHED
zwei 965faf9245114fb64c0a0c1bc2f5213f69e12199a7ad6dea685b5d62ac5acb7 NOT MATCHED
zweig 43f25db77f3a0363dff7617755855365cf2ff071b50729b97cc5e6b7f3e5618cc NOT MATCHED
zwick 96ea8987d2b5d0ff773555b5cc318cf3d71a5bde00f25c556e65b1b6 NOT MATCHED
zwickau 7c1b8bf63bfd5b6675224556182a099835623f2f8debe2ec604f9d82cb08e4 NOT MATCHED
zwicky c6c25a3e4cafdbded440529fb9f3f18ac2521d0b4e44afe0046173430b61f57 NOT MATCHED
zwieback 31b67ea013c06813e26e0466ef22aac00d0742615c3a8127e438e87b4b20c5464 NOT MATCHED
zwiebacks cf4df730792d4bc427c4ee046a226f3362cc657e8ceb9515c1f61c3a8a7 NOT MATCHED
zwibel 080d15f65c1fb580a38e0f99e565869cc98fb9b3773d4ad0f2e05422ab31c41ee NOT MATCHED
zwieselite 2f2342c4e1c99673b54aca30655a35d89767864c3f5e9718f90365d92360dfa0 NOT MATCHED
zwingle c99210cef7014a9a948d5270a6d3cd99fdafbc8cf5a68ca2b81877bea74f39b NOT MATCHED
zwingli b05d6713ea431a27239a6d191adc9d5521c2ae2596a8bc1dbf2bfab102 NOT MATCHED
zwinglian 5f2b18df93d7fa277fc4a2c7581a2728e64cf25a1e71652b86979ee6b5bac NOT MATCHED
zwinglianism aedf7feabae8a15a22334d5cd43844b60f946f03882d13a8acd60b84dd80348b0 NOT MATCHED
zwinglianist 17158ad1fefe7d3e6fc2d801019d22b3f64a37e2ddff68ae9d877cf8a084 NOT MATCHED
zwitter 6df12d857a5b3461a67a6f4548250d705497c73c48cd5242f226e38077d18b9 NOT MATCHED
zwitterion 0b794ab2da476a76aaeb642ad9fe582dfd79cc0042113790d30adef2da8b51b5 NOT MATCHED
zwitterionic 43d9a3b78420070e6b0143c3e9495c296487814a3c4d4c8747d630c7de3 NOT MATCHED
zwolle 327589a13f6940f1b17e3f38e5826724e8b2bb8cebccef4c151e7f91e625c635 NOT MATCHED
zworykin 770797ec4f4bd24b34e54ad3f7e958c05cc673631b47623b372e4473519e1305 NOT MATCHED
zz 5d7a15c6b07909667c79feac6346d16c02204fc1c63389721179c2015cb21c2bb NOT MATCHED
ztt 58dae019524c7d7d79ceaa3a43186ce4a5f64d9ad1817ac4b06a649500de2ef8 NOT MATCHED
zzz 2bfc0c1cb86d489616d5eaf811512b493869e2b14dfe420eaba6a4a4f431e94b NOT MATCHED
```

The key used to encrypt is median.

Conclusion:

- 1) AES, DES are symmetric key algorithms using the same keys to encrypt and decrypt the data.
- 2) ECB mode of encryption is the weakest form of encryption in comparison to CBC, CFB and OFB. If it is used to encrypt an image, an outline of the main contents of the image is still visible in the encrypted image.
- 3) ECB and CBC use padding while encryption while the other two(OFB and CFB) don't. This proves that ECB and CBC are block ciphers while CFB and OFB are stream ciphers. I could also verify that openssl uses PKCS5 standard for padding as it adds padding till the file size becomes

the next closest multiple of the block size.

- 4) I learned how different modes react to a corrupted bit of a cipher text. The best decryption in such a case is provided by OFB where only the corrupted bit of cipher text is affected while encrypting.
- 5) I could conclude from this experiment that, If we know the plaintext, ciphertext and iv, I can easily find the key using brute force method. In this case, we tried out each word with less than 16 characters from the dictionary and found the key to be “median” as seen in figure 1.6.