Crypto Lab - Secret-Key Encryption

Monday, March 31 2014, 3:57 PM

Task 1: Encryption using different ciphers and modes

Contents of the plain.txt file containing the plaintext to be encrypted:

```
[04/01/2014 08:58] seed@ubuntu:~/Documents$ cat plain.txt
Mark
```

Below are the screenshots of the commands and their respective output to their right seen through gedit, since vim and other CLI editors will mess up some ASCII characters.

Cipher: AES

```
[04/01/2014 11:13] seed@ubuntu:~/Documents$ openssl enc -aes-128-cbc -e -in plai
n.txt -out cipher.bin -K 00112233445566778899aabbccddeeff -iv 0102030405060708
[04/01/2014 11:14] seed@ubuntu:~/Documents$ gedit cipher.bin
[04/01/2014 11:15] seed@ubuntu:~/Documents$ openssl enc -aes-128-cfb -e -in plai
n.txt -out cipher.bin -K 00112233445566778899aabbccddeeff -iv 0102030405060708
[04/01/2014 11:19] seed@ubuntu:~/Documents$ openssl enc -aes-128-ecb -e -in plai
n.txt -out cipher.bin -K 00112233445566778899aabbccddeeff -iv 0102030405060708
```

```
cipher.bin
UkiÞŽô¬ÿÇyù\ï
\EC\F4V
^ÙN_ā{v
```

Cipher: DES

```
[04/01/2014 11:20] seed@ubuntu:~/Documents$ openssl enc -des-cbc -e -in plain.tx
t -out cipher.bin -K 00112233445566778899aabbccddeeff -iv 0102030405060708
[04/01/2014 11:22] seed@ubuntu:~/Documents$ openssl enc -des-cfb -e -in plain.tx
t -out cipher.bin -K 00112233445566778899aabbccddeeff -iv 0102030405060708
[04/01/2014 11:23] seed@ubuntu:~/Documents$ openssl enc -des-ofb -e -in plain.tx
t -out cipher.bin -K 00112233445566778899aabbccddeeff -iv 0102030405060708
```

```
,洠\BA\94}
\A9e\B1\B4
\A9e\B1\B4
```

Cipher: Blowfish

```
[04/01/2014 11:24] seed@ubuntu:~/Documents$ openssl enc -bf-cbc -e -in plain.txt
 -out cipher.bin -K 00112233445566778899aabbccddeeff -iv 0102030405060708
[04/01/2014 11:25] seed@ubuntu:~/Documents$ openssl enc -bf-cfb -e -in plain.txt
 -out cipher.bin -K 00112233445566778899aabbccddeeff -iv 0102030405060708
[04/01/2014 11:27] seed@ubuntu:~/Documents$ openssl enc -bf-ofb -e -in plain.txt
 -out cipher.bin -K 00112233445566778899aabbccddeeff -iv 0102030405060708
```

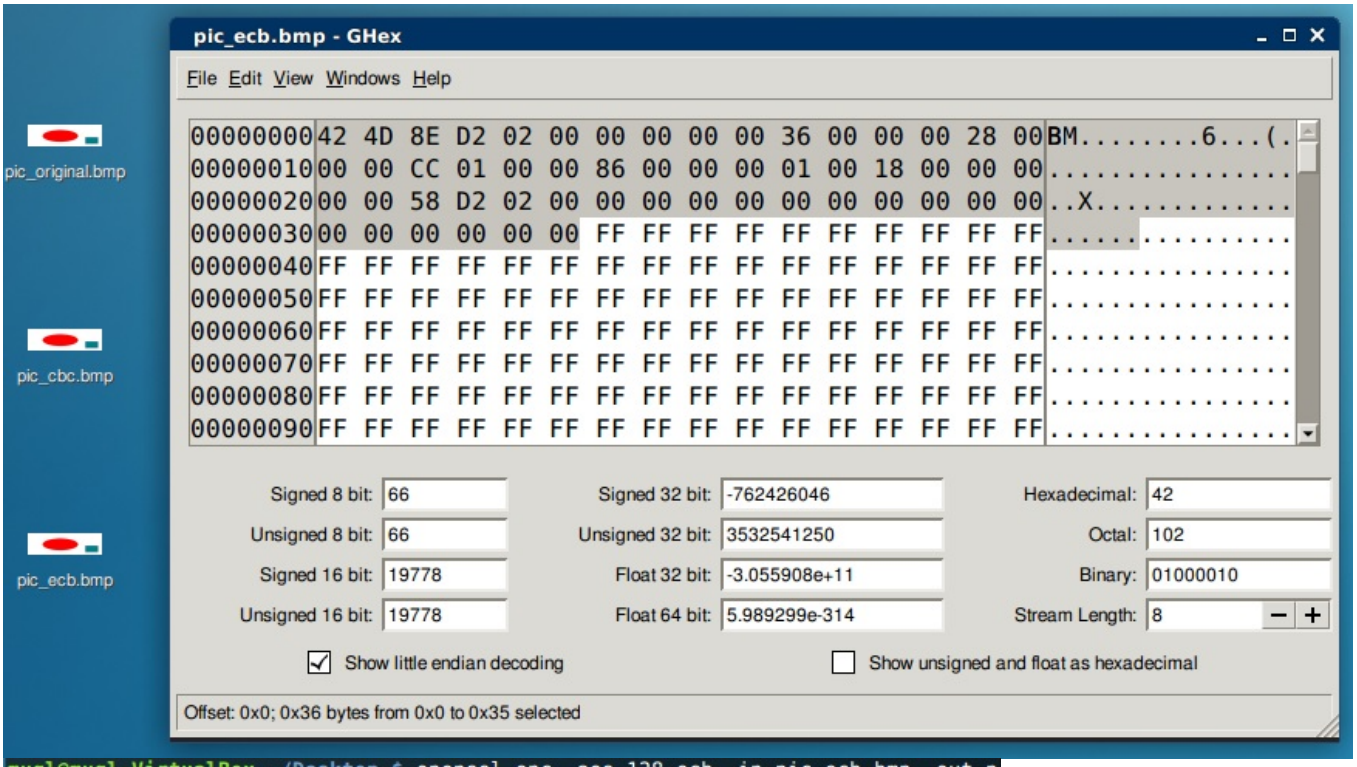
```
:@IÉÃA
\9CT\C1\AF
\9CT\C1\AF
```

Note how CFB and OFB modes, when used with the same cipher, seem to produced the same output in the test cases above.

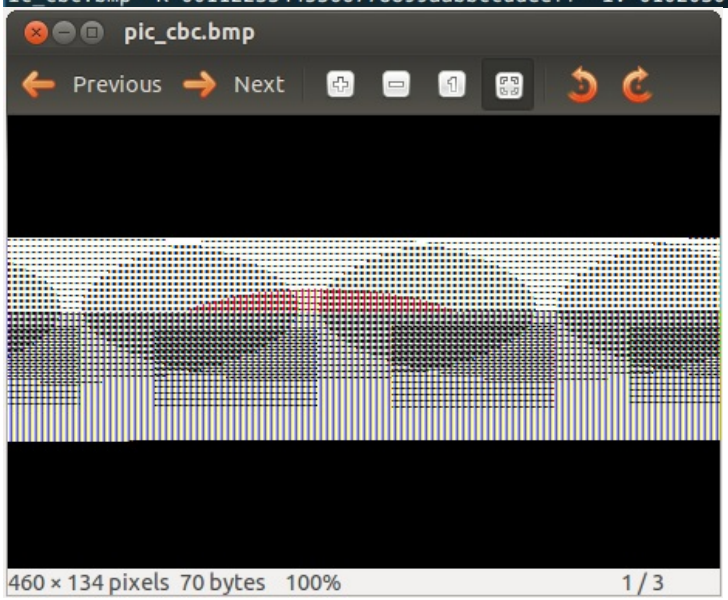
Task 2: Encryption Mode – ECB vs. CBC

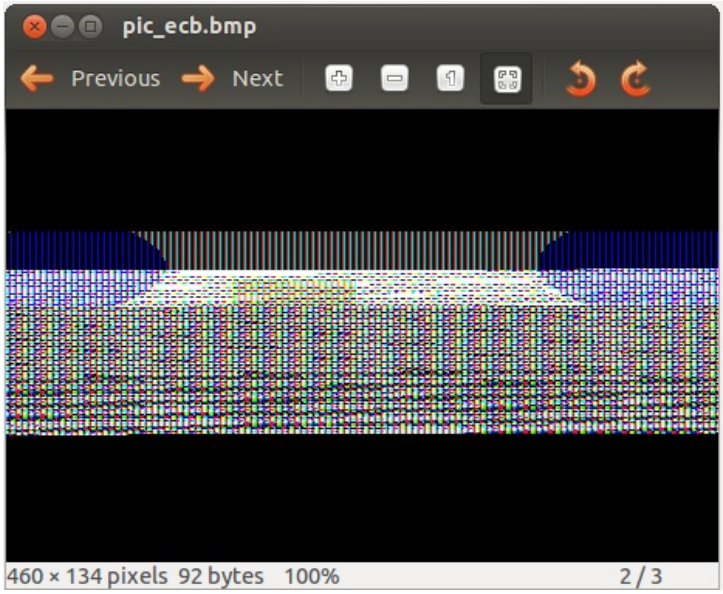
First, since there is header information in the original file too, I removed the header info first and then encrypted the file and returned the header of the original unencrypted picture.
Each time the picture is opened with an image viewer, it is rendered slightly differently. This is assumed to be an inconsistency in header information.
Moreover, the file size was inconsistent after encryption.

Below is the screenshot of one of the bitmaps' header. I opened it on Linux Mint because of problems GHex was having on Ubuntu.

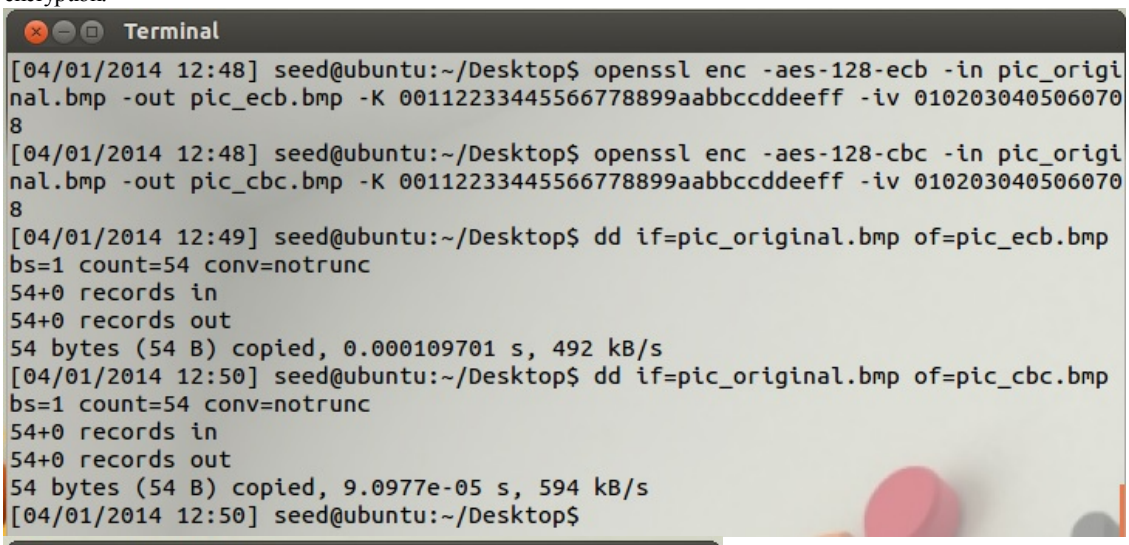


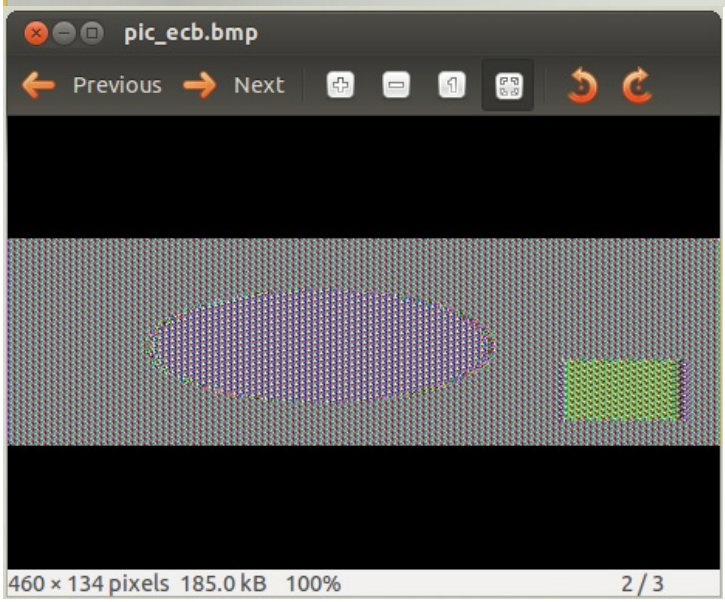

```
mugl@mugl-VirtualBox ~/Desktop $ openssl enc -aes-128-ecb -in pic_ecb.bmp -out p
ic_ecb.bmp -K 00112233445566778899aabbccddeeff -iv 0102030405060708
mugl@mugl-VirtualBox ~/Desktop $ openssl enc -aes-128-cbc -in pic_cbc.bmp -out p
ic_cbc.bmp -K 00112233445566778899aabbccddeeff -iv 0102030405060708
```

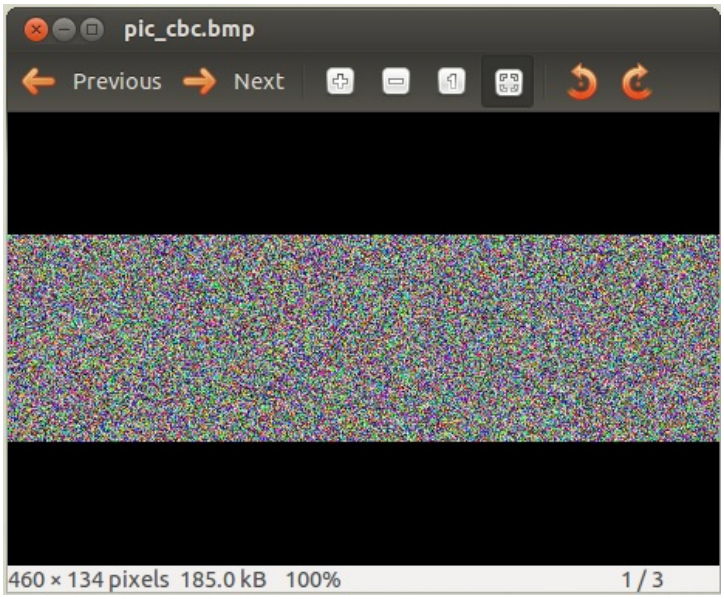




CBC encryption mode was closer to the original file. On the other hand, the ECB picture had more difference. Since this result is inconsistent, the experiment was ran again. But this time, instead of GHex, the dd command was used to copy the header from the original. And the old header was not removed before encryption.



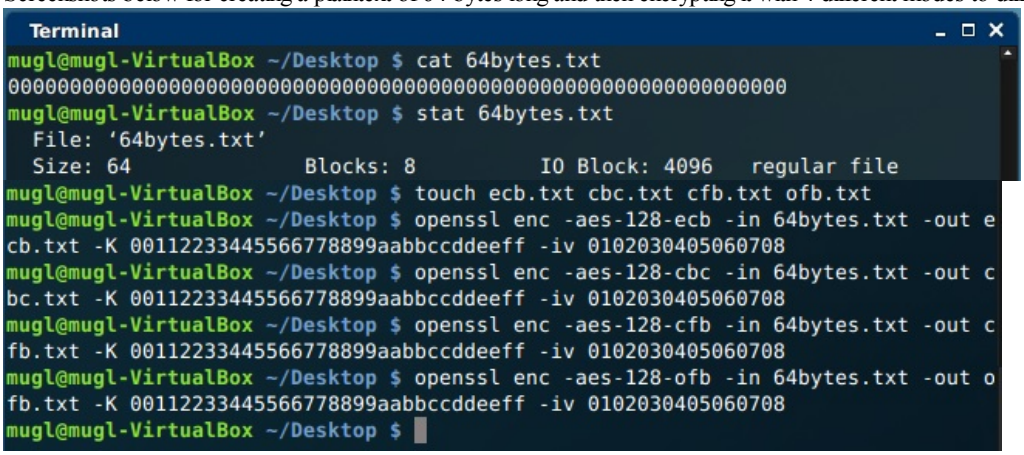




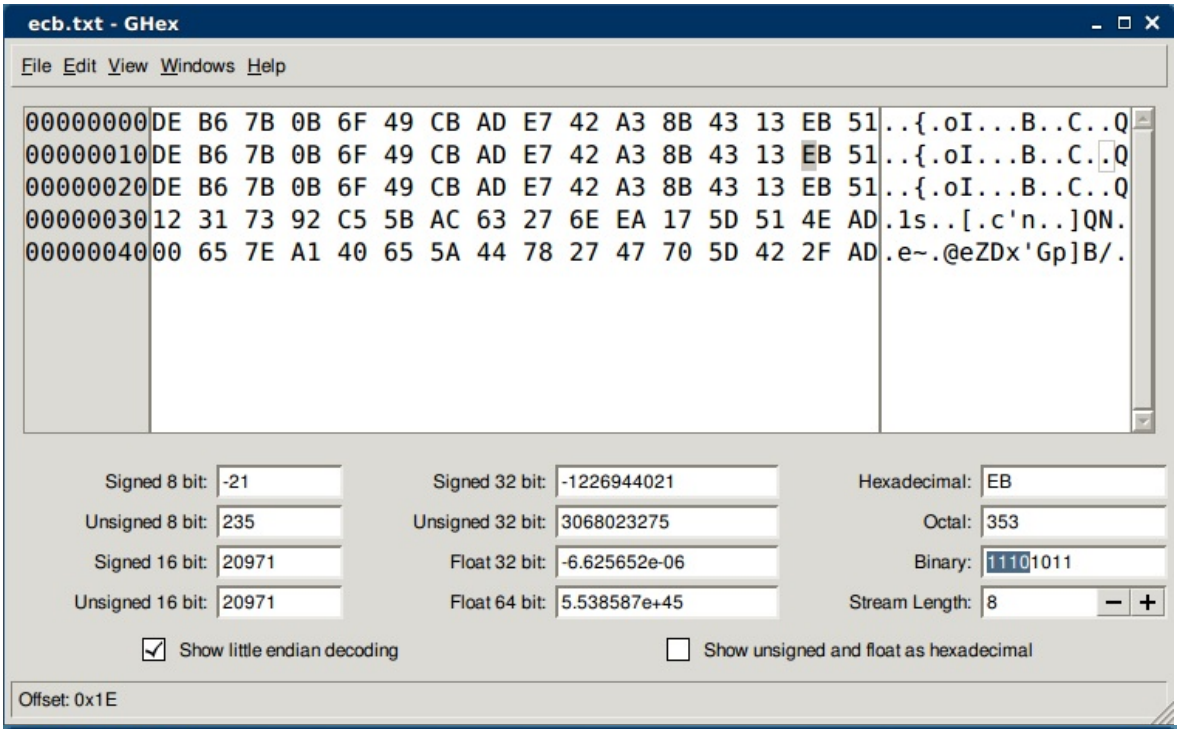
The result is consistent as the images always render the same and they all have the same size as the original. We conclude that CBC is better since it scrambles the whole input, as opposed to ECB which handles blocks of data from input and, as a result, blocks of the same data will be encrypted the same way and therefore have the same resulting block.

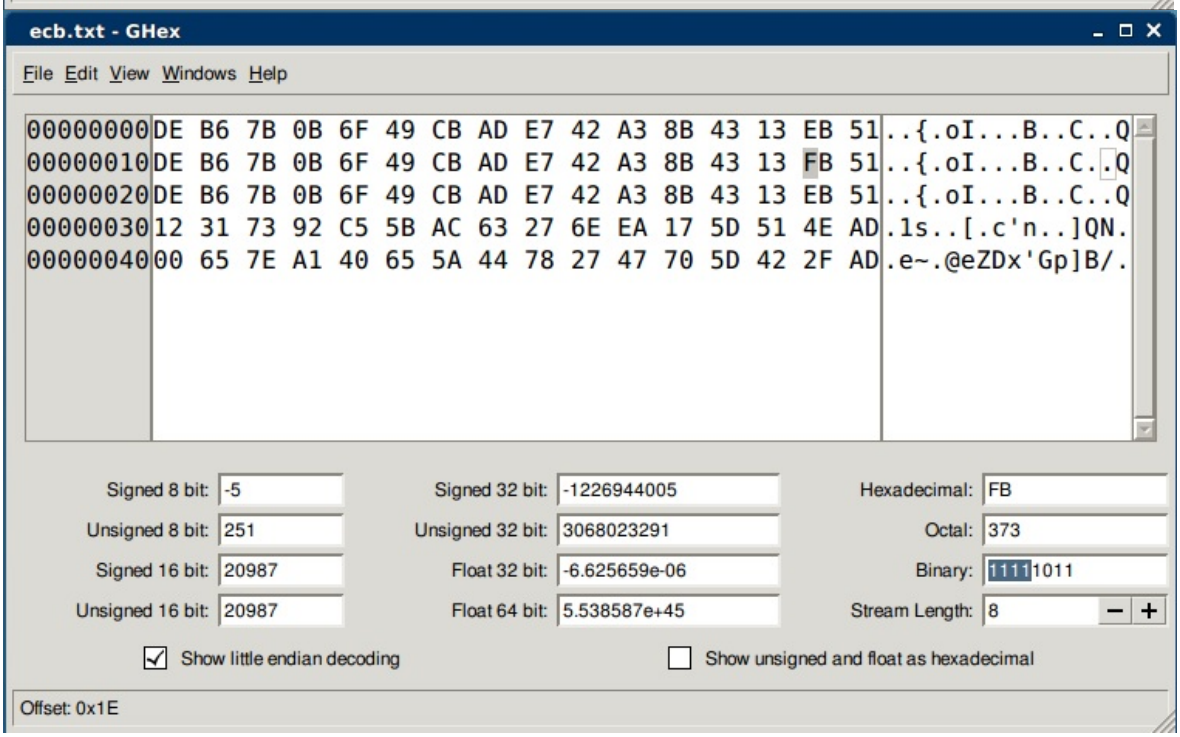
Task 3: Encryption Mode – Corrupted Cipher Text

Screenshots below for creating a plaintext of 64 bytes long and then encrypting it with 4 different modes to different output files.



To corrupt one bit from the 30th byte, we open the encrypted file in GHex and look for the 30th byte. Then, in the Binary field, the first 4 bits belong to the 30th byte. We therefore need to corrupt (i.e. flip) one of these bits so that the resulting byte is corrupted by 1 bit. In the screenshots below, we modify the last 0 bit to a 1 to make the relevant byte change from E to F in hexadecimal. We go through this process for all encryption modes.





After corrupting a single bit from every file, we decrypt them and post the results.

```
Terminal
mugl@mugl-VirtualBox ~/Desktop $ touch ecb-d.txt cbc-d.txt cfb-d.txt ofb-d.txt
mugl@mugl-VirtualBox ~/Desktop $ openssl aes-128-ofb -d -in ofb.txt -out ofb-d.t
xt -K 00112233445566778899aabbccddeeff -iv 0102030405060708
mugl@mugl-VirtualBox ~/Desktop $ openssl aes-128-cfb -d -in cfb.txt -out cfb-d.t
xt -K 00112233445566778899aabbccddeeff -iv 0102030405060708
mugl@mugl-VirtualBox ~/Desktop $ openssl aes-128-cbc -d -in cbc.txt -out cbc-d.t
xt -K 00112233445566778899aabbccddeeff -iv 0102030405060708
mugl@mugl-VirtualBox ~/Desktop $ openssl aes-128-ecb -d -in ecb.txt -out ecb-d.t
xt -K 00112233445566778899aabbccddeeff -iv 0102030405060708
mugl@mugl-VirtualBox ~/Desktop $ cat 64bytes.txt ecb-d.txt cbc-d.txt cfb-d.txt o
fb-d.txt
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
mugl@mugl-VirtualBox ~/Desktop $
```

Analyzing the results, we conclude that OFB is the best encryption mode in error-prone environments, since in the result only the corrupted byte was removed, and the rest of the data was decrypted flawlessly. CFB also removed the corrupted byte, but around 12 other bytes were also corrupted in the process. As for ECB and CBC, a lot of the data was corrupted, and there is no way to know which of the bytes is the source of the corruption in this case.

We conclude that in block-chaining modes, data corruption is prone to spread more than in feedback modes.

Task 4: Padding

The official definition of the PKCS#5 padding as defined in its RFC: The padding string PS shall consist of 8 - (||M|| mod 8) octets all having value 8 - (||M|| mod 8).

For a plaintext file of size 20, we get a 32 byte ciphertext output after padding.
12 bytes padded: added 4 + a whole block of 8.

```
mugl@mugl-VirtualBox ~/Desktop $ touch padding-enc.txt
mugl@mugl-VirtualBox ~/Desktop $ openssl enc -aes-128-cbc -in padding.txt -out p
adding-enc.txt -K 00112233445566778899aabbccddeeff -iv 0102030405060708
mugl@mugl-VirtualBox ~/Desktop $ stat padding-enc.txt
  File: 'padding-enc.txt'
  Size: 32              Blocks: 8          IO Block: 4096   regular file
```

For a plaintext file of size 32, the ciphertext size will become 48.
16 bytes padded: two blocks of 8 since 48 is divisible by 8.

```
mugl@mugl-VirtualBox ~/Desktop $ openssl enc -aes-128-cbc -in padding.txt -out p
adding-enc.txt -K 00112233445566778899aabbccddeeff -iv 0102030405060708
mugl@mugl-VirtualBox ~/Desktop $ stat padding-enc.txt
  File: 'padding-enc.txt'
  Size: 48              Blocks: 8          IO Block: 4096   regular file
```

Below are screenshots of an experiment run twice, once with a plaintext file of size 32, and the other when the same file was of size 29. The encryption was made using cipher AES-128 on four different modes: CBC, CFB, ECB and OFB.

```
Terminal
mugl@mugl-VirtualBox ~/Desktop/Padding $ touch ecb.txt cbc.txt cfb.txt ofb.txt
mugl@mugl-VirtualBox ~/Desktop/Padding $ openssl enc -aes-128-ecb -in plain.txt
-out ecb.txt -K 00112233445566778899aabbccddeeff -iv 0102030405060708
mugl@mugl-VirtualBox ~/Desktop/Padding $ openssl enc -aes-128-cbc -in plain.txt
-out cbc.txt -K 00112233445566778899aabbccddeeff -iv 0102030405060708
mugl@mugl-VirtualBox ~/Desktop/Padding $ openssl enc -aes-128-cfb -in plain.txt
-out cfb.txt -K 00112233445566778899aabbccddeeff -iv 0102030405060708
mugl@mugl-VirtualBox ~/Desktop/Padding $ openssl enc -aes-128-ofb -in plain.txt
-out ofb.txt -K 00112233445566778899aabbccddeeff -iv 0102030405060708
mugl@mugl-VirtualBox ~/Desktop/Padding $ ls -l
total 20
-rw-r--r-- 1 mugl mugl 48 Mar 31 21:16 cbc.txt
-rw-r--r-- 1 mugl mugl 32 Mar 31 21:16 cfb.txt
-rw-r--r-- 1 mugl mugl 48 Mar 31 21:16 ecb.txt
-rw-r--r-- 1 mugl mugl 32 Mar 31 21:17 ofb.txt
-rw-r--r-- 1 mugl mugl 32 Mar 31 21:12 plain.txt
mugl@mugl-VirtualBox ~/Desktop/Padding $
```

```
Terminal
mugl@mugl-VirtualBox ~/Desktop/Padding $ openssl enc -aes-128-ecb -in plain.txt
-out ecb.txt -K 00112233445566778899aabbccddeeff -iv 0102030405060708
mugl@mugl-VirtualBox ~/Desktop/Padding $ openssl enc -aes-128-cbc -in plain.txt
-out cbc.txt -K 00112233445566778899aabbccddeeff -iv 0102030405060708
mugl@mugl-VirtualBox ~/Desktop/Padding $ openssl enc -aes-128-cfb -in plain.txt
-out cfb.txt -K 00112233445566778899aabbccddeeff -iv 0102030405060708
mugl@mugl-VirtualBox ~/Desktop/Padding $ openssl enc -aes-128-ofb -in plain.txt
-out ofb.txt -K 00112233445566778899aabbccddeeff -iv 0102030405060708
mugl@mugl-VirtualBox ~/Desktop/Padding $ ls -l
total 20
-rw-r--r-- 1 mugl mugl 32 Mar 31 21:19 cbc.txt
-rw-r--r-- 1 mugl mugl 29 Mar 31 21:19 cfb.txt
-rw-r--r-- 1 mugl mugl 32 Mar 31 21:19 ecb.txt
-rw-r--r-- 1 mugl mugl 29 Mar 31 21:19 ofb.txt
-rw-r--r-- 1 mugl mugl 29 Mar 31 21:19 plain.txt
mugl@mugl-VirtualBox ~/Desktop/Padding $
```

OFB and CFB modes will output a ciphertext with its file size equal to that of the plaintext. They therefore do not use padding. The reason is they will XOR the result with the plaintext, so they do not work in blocks, and therefore do not need to add padding. CBC and ECB (block chaining modes) use padding since the ciphertext size is larger and a multiple of 8.

Task 5: Programming using the Crypto Library

In this task, we will write a C program that will brute force an English word dictionary: Since we know the IV and encryption type and mode, we need to encrypt the plaintext with every word in the dictionnary as a key for the encryption and then compare every instance of the encrpytion to the ciphertext we already have. If they match, then the corresponding dictionary word is the correct key.

Another way to do it is to decrypt the ciphertext with each word from the dictionary as the key, and we should eventually get the plaintext.