INFORMATION SECURITY

Secret Key Infrastructure

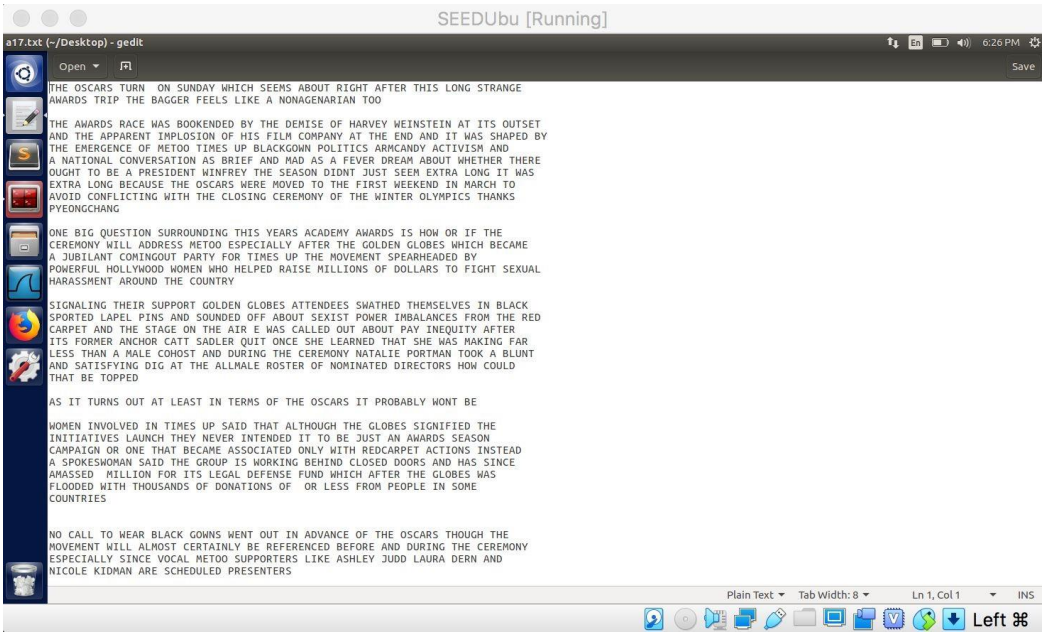
Ambreen Kanwal
University of Sahiwal

UNIVERSITY OF SAHIWAL
A Progressive University for Bright Future

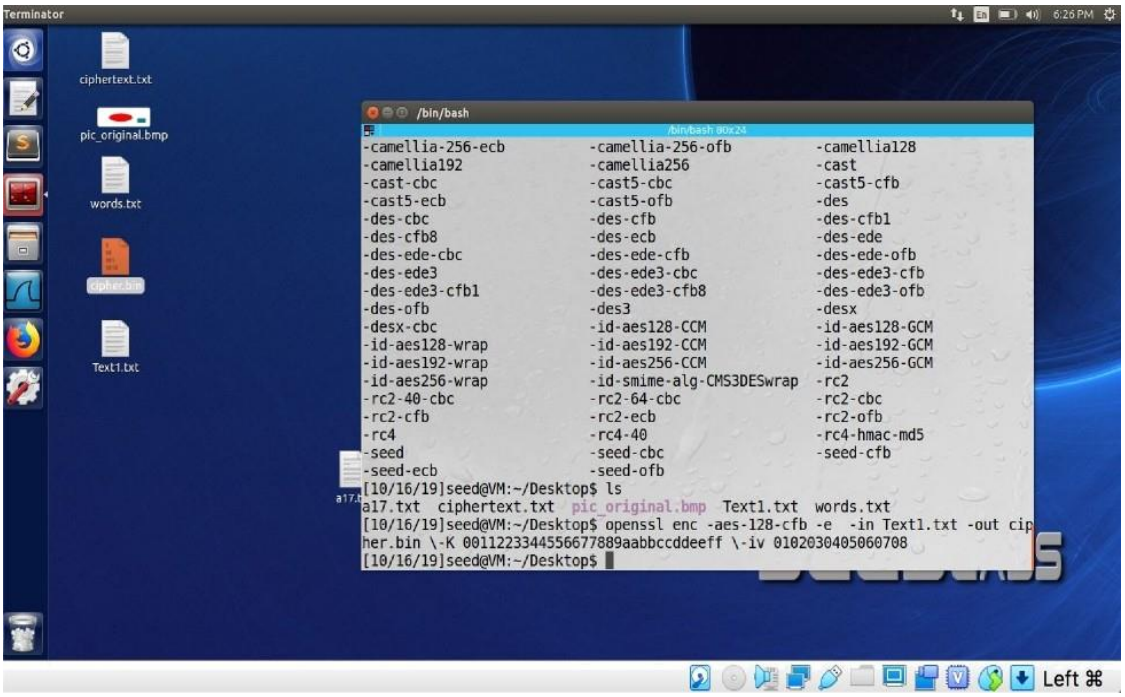
Secret Key Infrastructure

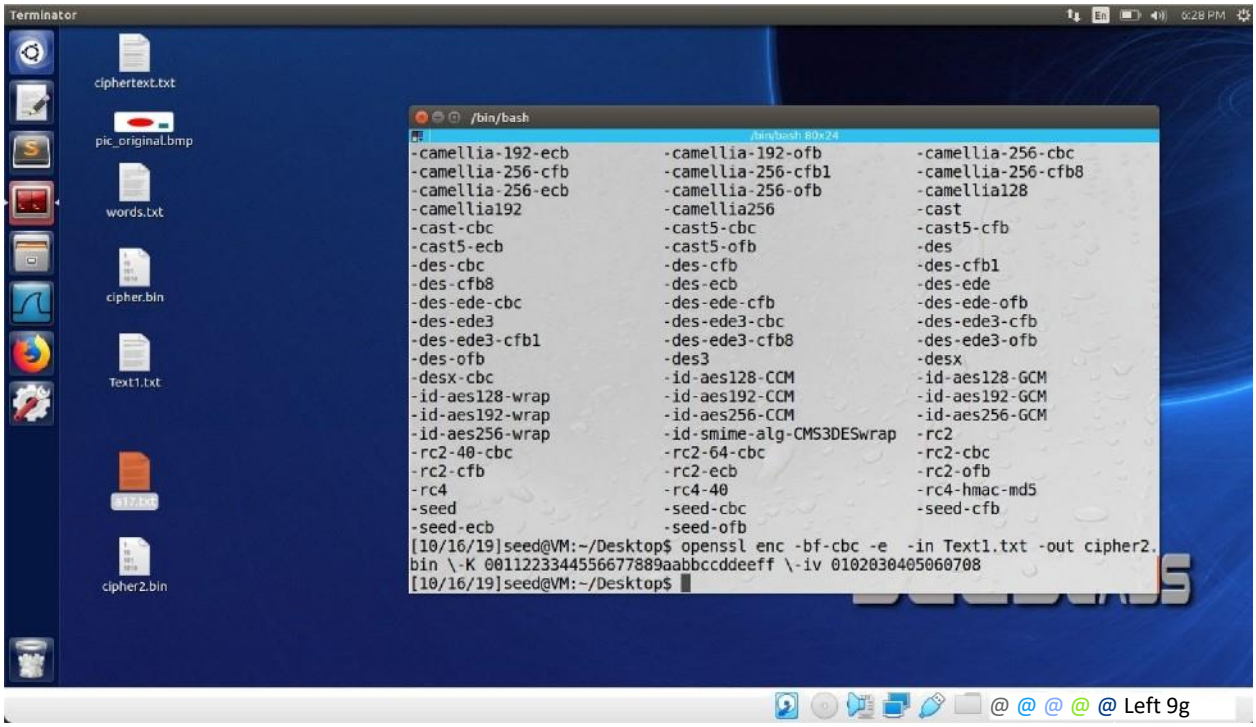
Task 2: Encryption using Different Ciphers and Modes

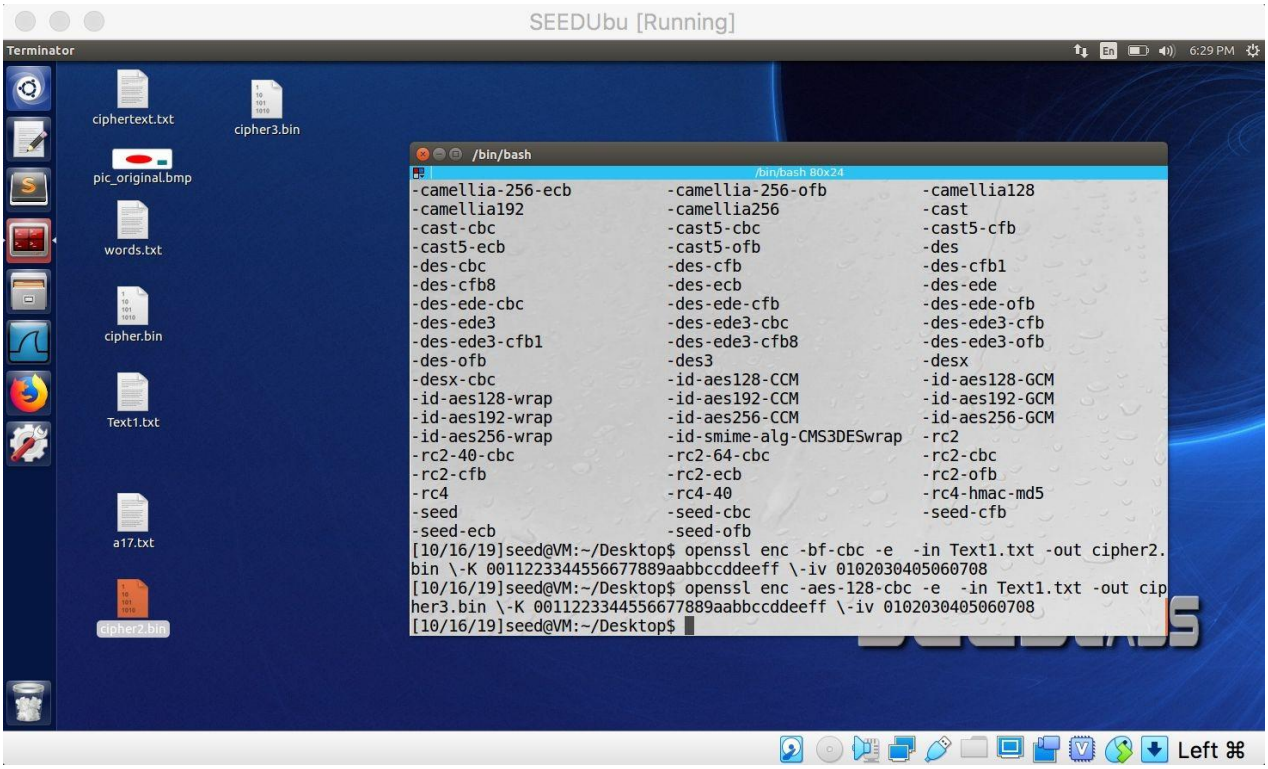
We used the provided ciphertext. Below is the decrypted result in plaintext.



Three different types of encryption below:

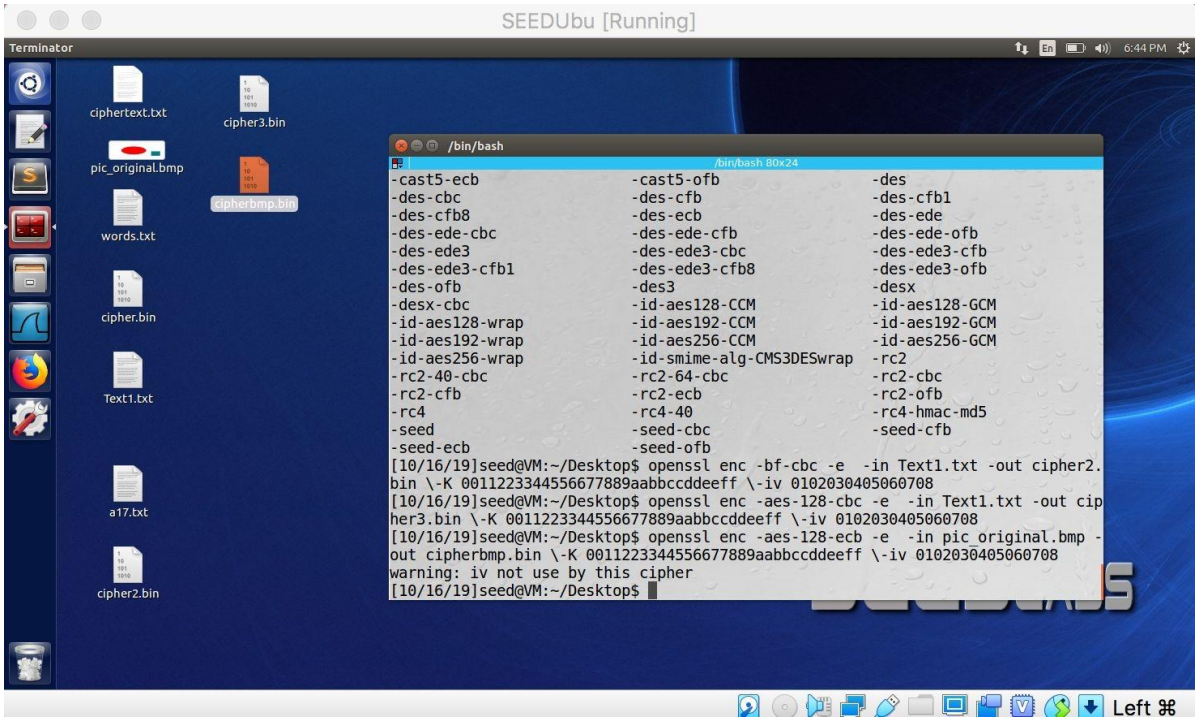




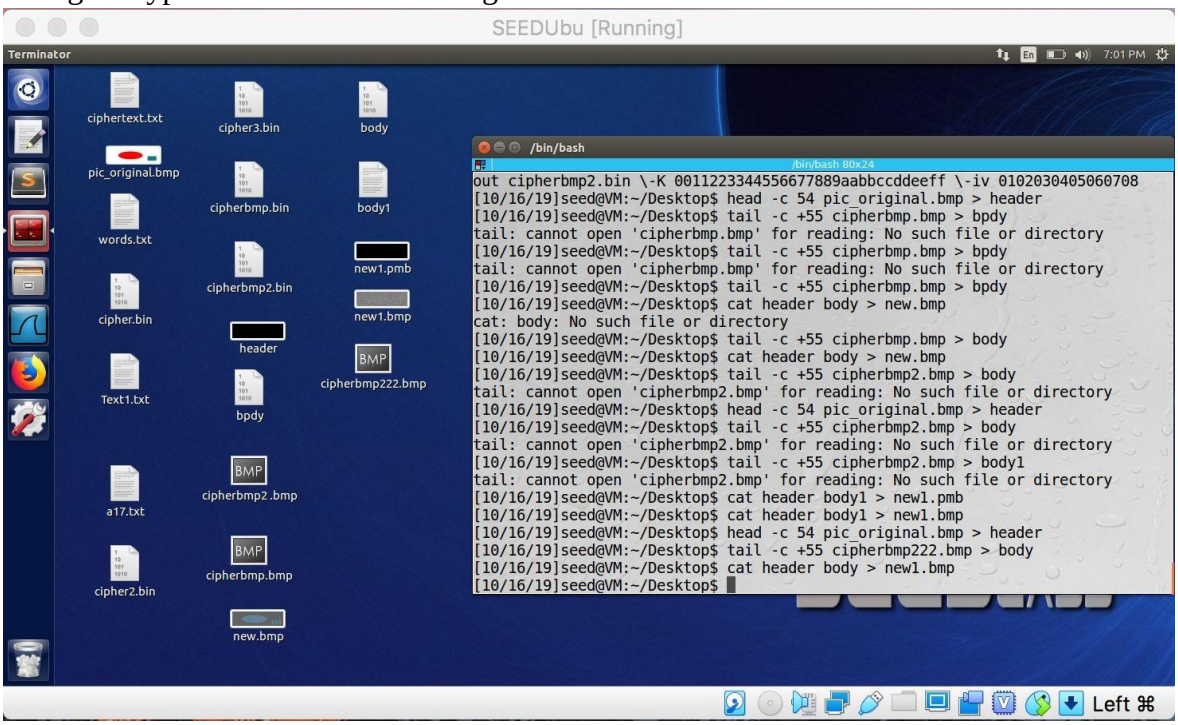


Task 3: Encryption Mode – ECB vs. CBC

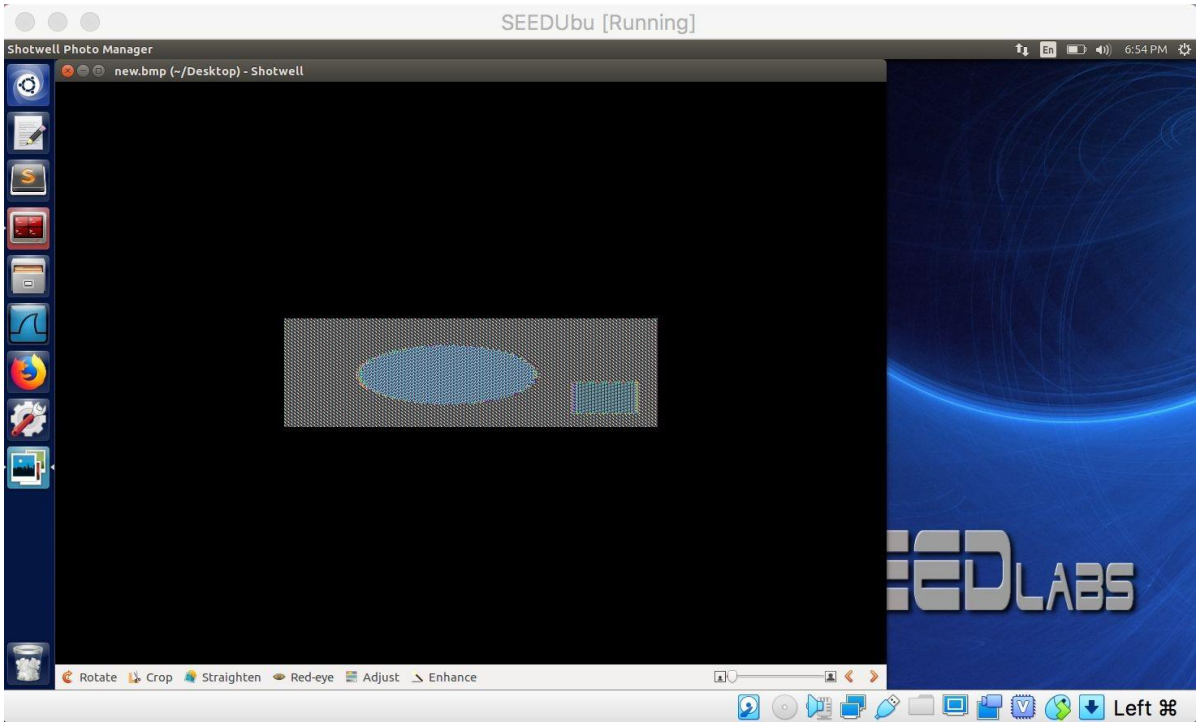
Image encryption using ECB and CBC.



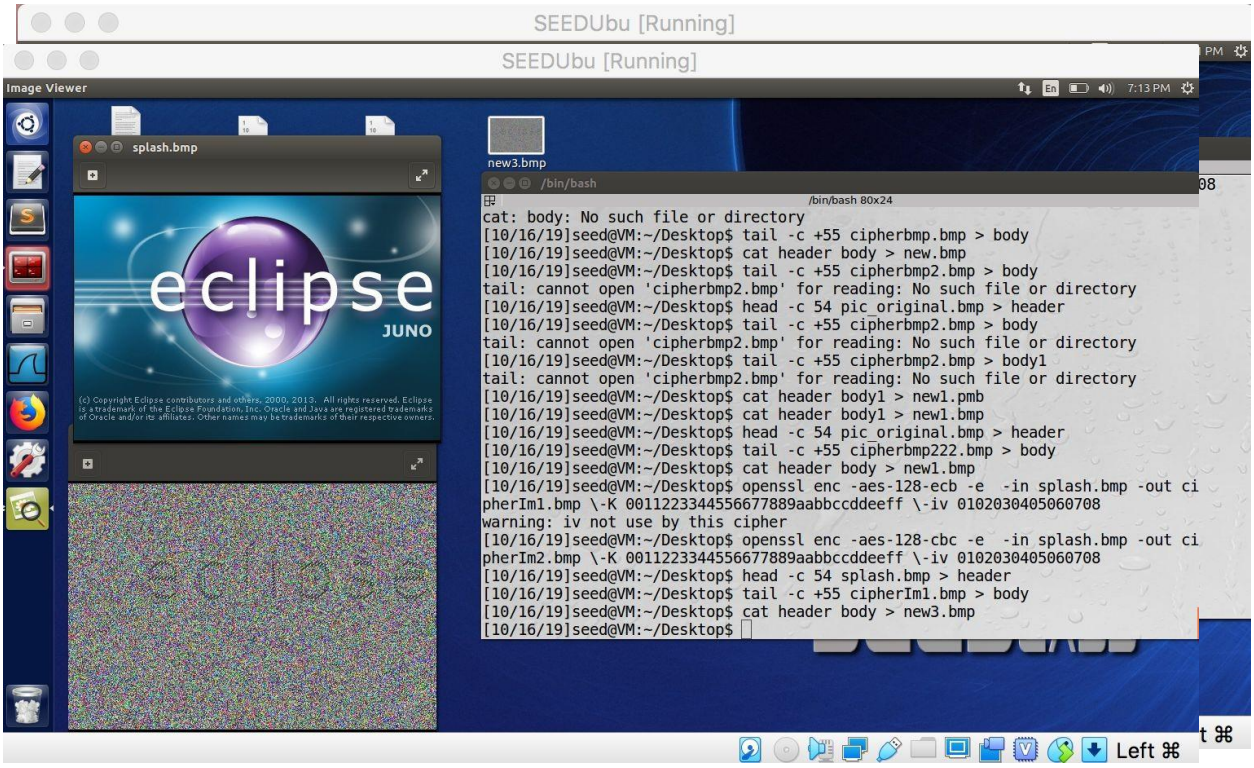
Making encrypted file viewable as image



Viewing ECB encrypted image.

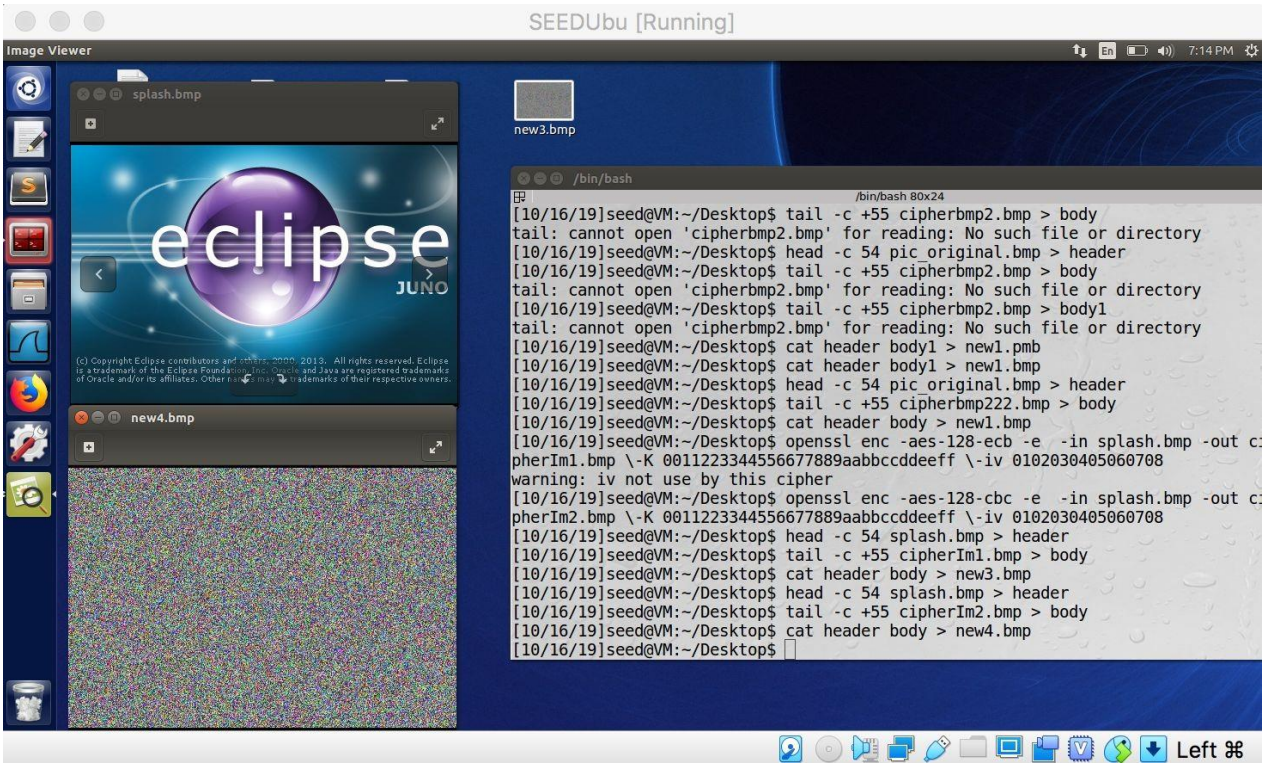


Viewing CBC encrypted image.



Viewing ECB encrypted image. Can make out some of the original image letters.

Viewing CBC encrypted file. Cannot make out any of the original image’s letters.



Yes, using the ECB encryption it is very easy to make out the shapes of the encrypted image. When using the CBC encryption I cannot see any relationship between the actual image and the encrypted image. When I ran the above encryption again with a newly chosen image, the results were the same. With ECB, I could make out some of the original image in the encrypted image. With CBC, I was unable to recognize any of the original image in the encrypted image.

Task 4: Padding

1. I began by creating a text file of size 27 bytes. After encrypting the 27 byte file I got the following results with different encryption:
ECB - file size became 32 bytes. Padding was used.
CBC - file size became 32 bytes. Padding was used.
CFB - files size remained 27 bytes. No padding.
OFB - file size remained 27 bytes. No padding.

Conclusion is that CFB and OFB do not use padding because they are stream ciphers and therefore the plaintext message does not need to be a multiple of the block-size.

Creating three files size 5, 10, 16 bytes

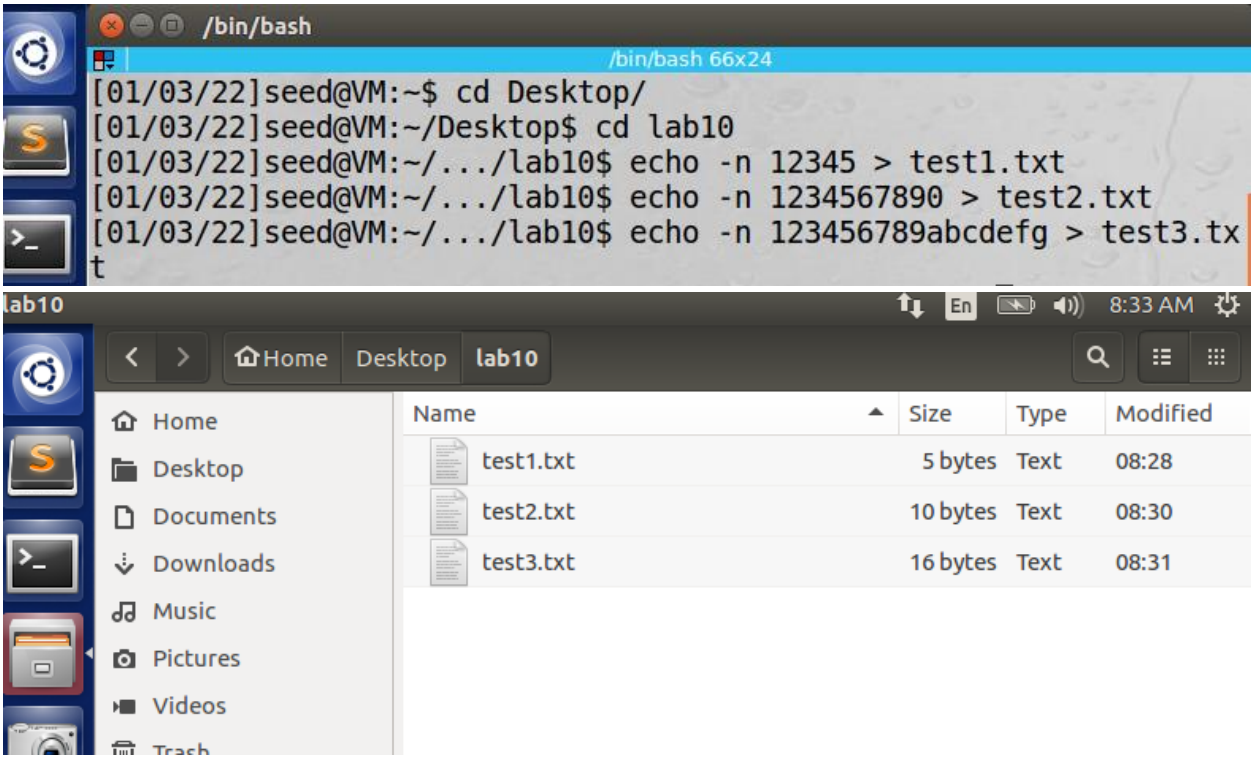
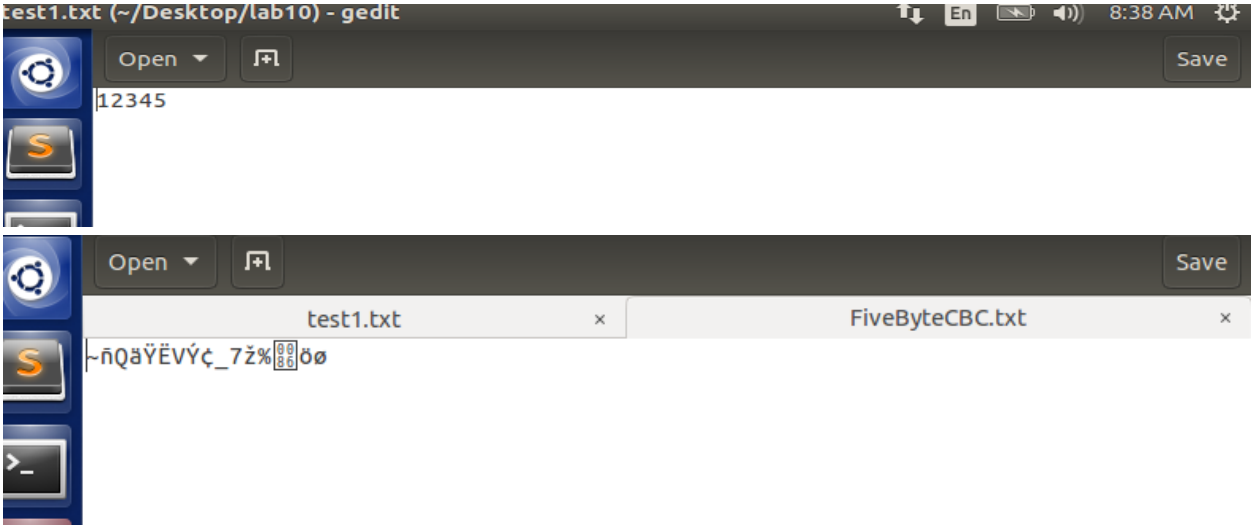


Image of encrypted 5 bytes file, size is 16 bytes.

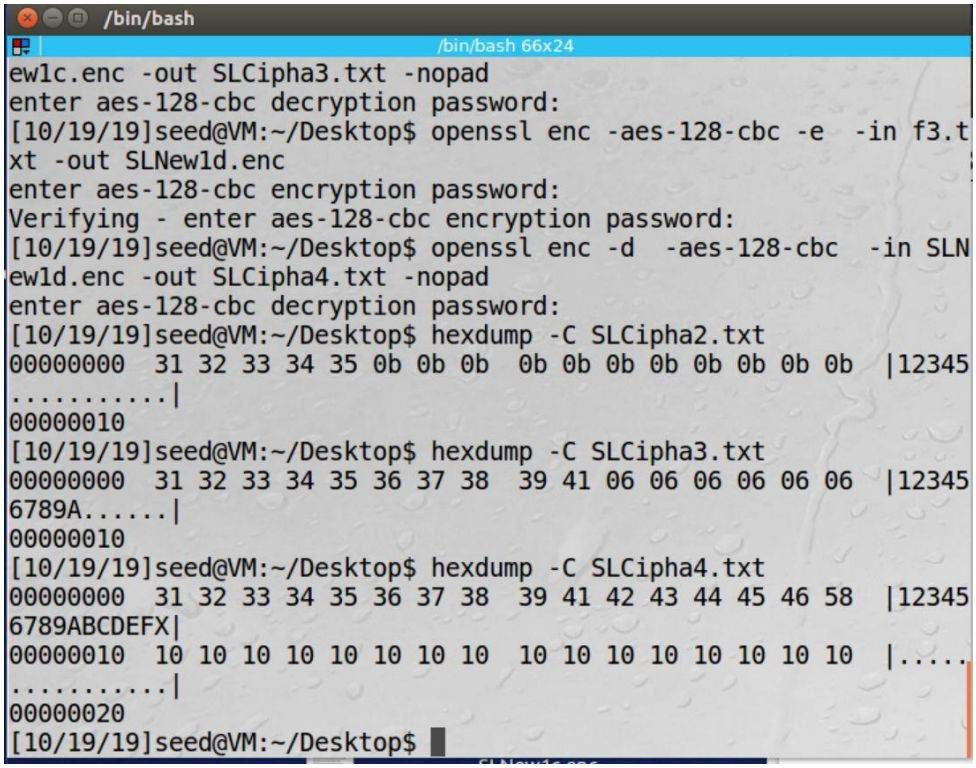
Image of encrypted 10 bytes file, size is 16bytes.

Image of encrypted 16 bytes file, size is 32 bytes.

FiveByteFile.txt and Encrypted FiveByteFile.txt :

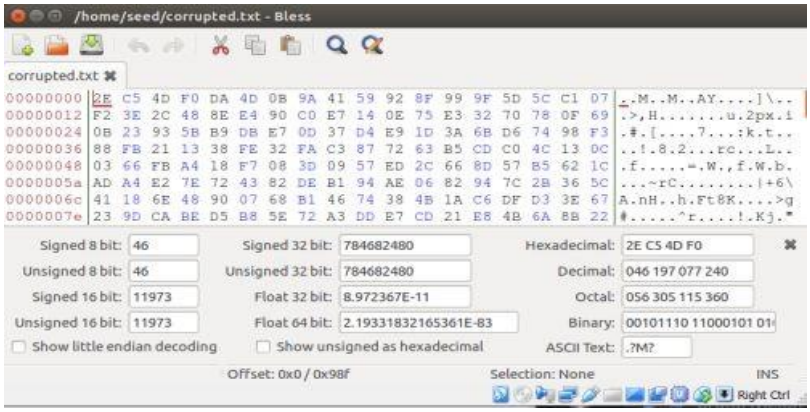


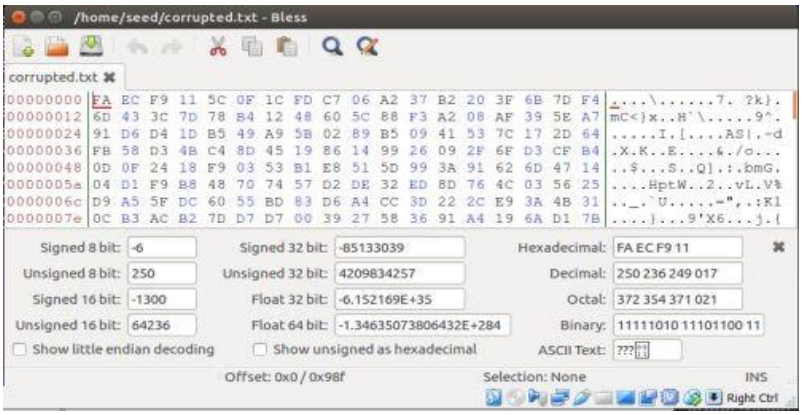
Hex Dump results for finding how each file was padded

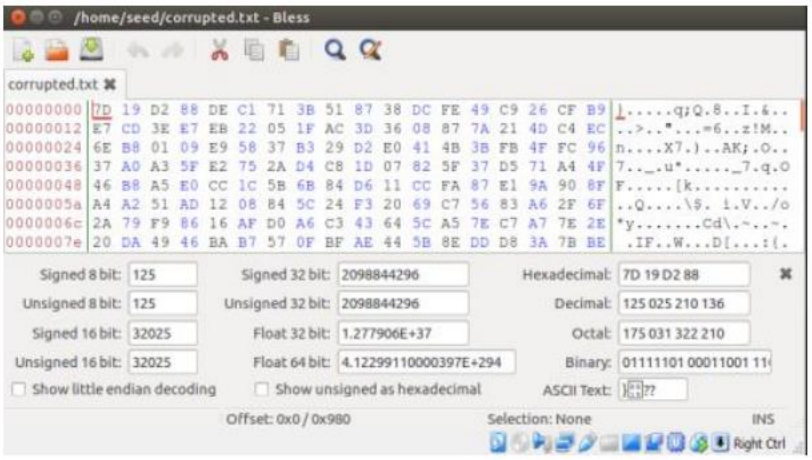


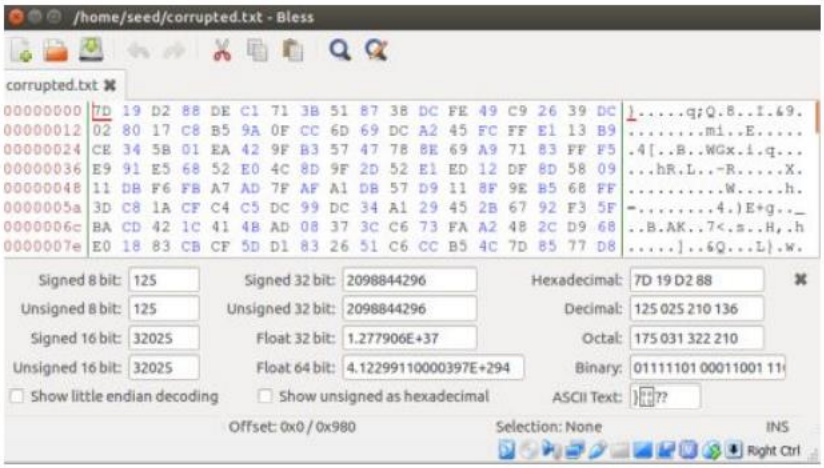
Task 5: Error Propagation – Corrupted Cipher Text

Created a text file greater than 1000 bytes in size. The file is essentially a repeat of NazimZerrouki and GregGertsen (our full names) until we reached or exceeded 1000 bytes. We then encrypted the file using AES 128 bit block ciphers using ECB, CBC, CFB, and OFB. The key and IV that were used was '00112233445566778899aabbccddeeff' and 'aabbccddeeff99887766554433221l' unless stated otherwise. The files prior to corrupting the 55th bit for ECB, CBC, CFB, and OFB respectively which are shown below:

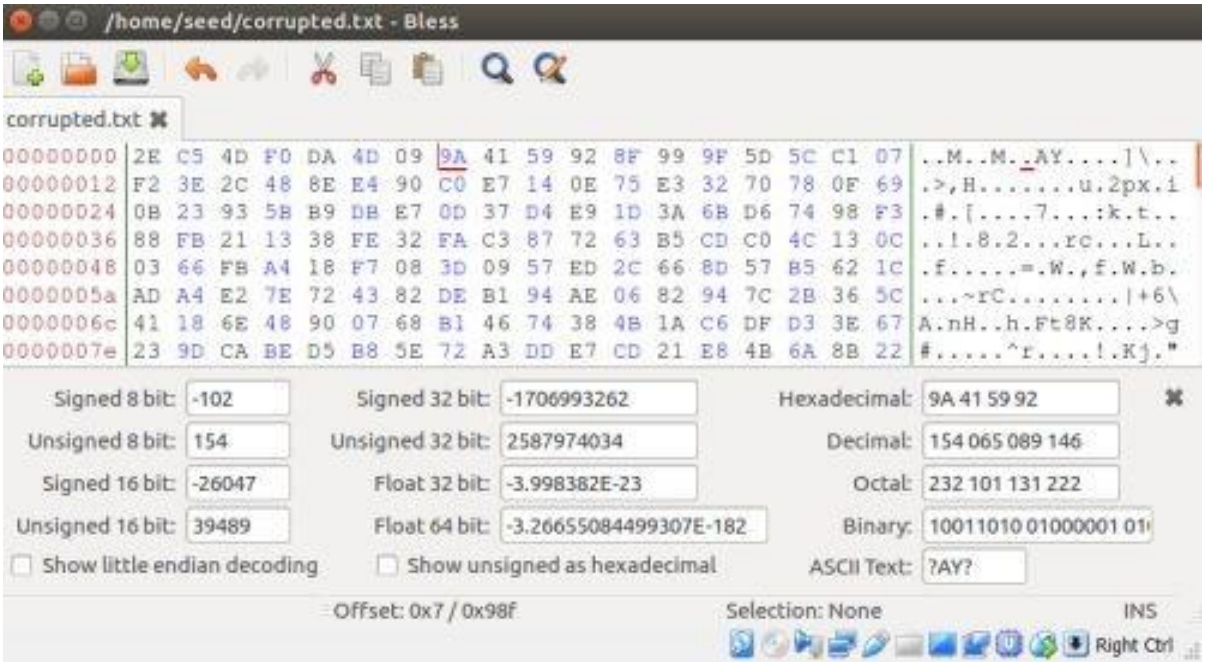



/home/seed/corrupted.txt - Bless
corrupted.txt
00000000 FA EC F9 11 5C 0F 1C FD C7 06 A2 37 B2 20 3F 6B 7D F4
00000012 6D 43 3C 7D 78 B4 12 48 60 5C 88 F3 A2 08 AF 39 5E A7
00000024 91 D6 D4 1D B5 49 A9 5B 02 89 B5 09 41 53 7C 17 2D 64
00000036 FB 58 D3 4B C4 8D 45 19 86 14 99 26 09 2F 6F D3 CF B4
00000048 0D 0F 24 18 F9 03 53 B1 E8 51 5D 99 3A 91 62 6D 47 14
0000005a 04 D1 F9 B8 48 70 74 57 D2 DE 32 ED 8D 76 4C 03 56 25
0000006c D9 A5 5F DC 60 55 BD 83 D6 A4 CC 3D 22 2C E9 3A 4B 31
0000007e 0C B3 AC B2 7D D7 D7 00 39 27 58 36 91 A4 19 6A D1 7B
Signed 8 bit: -6
Signed 32 bit: -85133039
Hexadecimal: FA EC F9 11
Unsigned 8 bit: 250
Unsigned 32 bit: 4209834257
Decimal: 250 236 249 017
Signed 16 bit: -1300
Float 32 bit: -6.152169E+35
Octal: 372 354 371 021
Unsigned 16 bit: 64236
Float 64 bit: -1.34635073806432E+284
Binary: 11111010 11101100 11
Show little endian decoding
Show unsigned as hexadecimal
ASCII Text:
Offset: 0x0 / 0x98f
Selection: None
INS
Right Ctrl

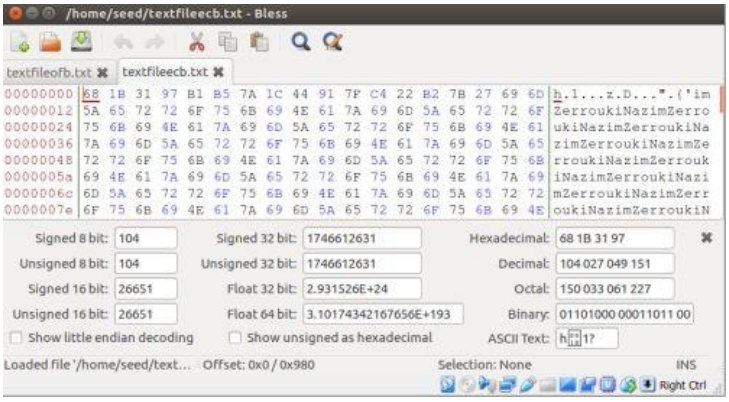

/home/seed/corrupted.txt - Bless
corrupted.txt
00000000 7D 19 D2 88 DE C1 71 3B 51 87 38 DC FE 49 C9 26 CF B9
00000012 E7 CD 3E E7 EB 22 05 1F AC 3D 36 08 87 7A 21 4D C4 EC
00000024 6E B8 01 09 E9 58 37 B3 29 D2 E0 41 4B 3B FB 4F FC 96
00000036 37 A0 A3 5F E2 75 2A D4 C8 1D 07 82 5F 37 D5 71 A4 4F
00000048 46 B8 A5 E0 CC 1C 5B 6B 84 D6 11 CC FA 87 E1 9A 90 8F
0000005a A4 A2 51 AD 12 08 84 5C 24 F3 20 69 C7 56 83 A6 2F 6F
0000006c 2A 79 F9 86 16 AF D0 A6 C3 43 64 5C A5 7E C7 A7 7E 2E
0000007e 20 DA 49 46 BA B7 57 0F BF AE 44 5B 8E DD D8 3A 7B BE
Signed 8 bit: 125
Signed 32 bit: 2098844296
Hexadecimal: 7D 19 D2 88
Unsigned 8 bit: 125
Unsigned 32 bit: 2098844296
Decimal: 125 025 210 136
Signed 16 bit: 32025
Float 32 bit: 1.277906E+37
Octal: 175 031 322 210
Unsigned 16 bit: 32025
Float 64 bit: 4.12299110000397E+294
Binary: 01111101 00011001 11
Show little endian decoding
Show unsigned as hexadecimal
ASCII Text:
Offset: 0x0 / 0x980
Selection: None
INS
Right Ctrl


/home/seed/corrupted.txt - Bless
corrupted.txt
00000000 7D 19 D2 88 DE C1 71 3B 51 87 38 DC FE 49 C9 26 39 DC
00000012 02 80 17 C8 B5 9A 0F CC 6D 69 DC A2 45 FC FF E1 13 B9
00000024 CE 34 5B 01 EA 42 9F B3 57 47 78 8E 69 A9 71 83 FF F5
00000036 E9 91 E5 68 52 E0 4C 8D 9F 2D 52 E1 ED 12 DF 8D 58 09
00000048 11 DB F6 FB A7 AD 7F AF A1 DB 57 D9 11 8F 9E B5 68 FF
0000005a 3D C8 1A CF C4 C5 DC 99 DC 34 A1 29 45 2B 67 92 F3 5F
0000006c BA CD 42 1C 41 4B AD 08 37 3C C6 73 FA A2 48 2C D9 68
0000007e E0 18 83 CB CF 5D D1 83 26 51 C6 CC B5 4C 7D 85 77 D8
Signed 8 bit: 125
Signed 32 bit: 2098844296
Hexadecimal: 7D 19 D2 88
Unsigned 8 bit: 125
Unsigned 32 bit: 2098844296
Decimal: 125 025 210 136
Signed 16 bit: 32025
Float 32 bit: 1.277906E+37
Octal: 175 031 322 210
Unsigned 16 bit: 32025
Float 64 bit: 4.12299110000397E+294
Binary: 01111101 00011001 11
Show little endian decoding
Show unsigned as hexadecimal
ASCII Text:
Offset: 0x0 / 0x980
Selection: None
INS
Right Ctrl

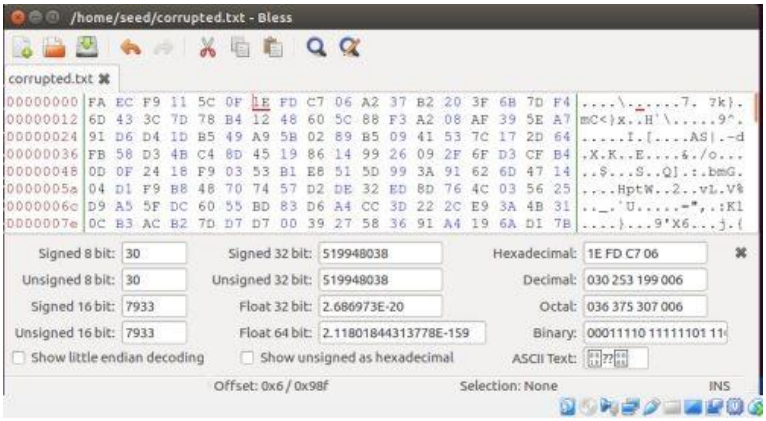
Corrupted encryption by changing 1 bit of the encrypted file for each type of encryption are shown below in the same order.



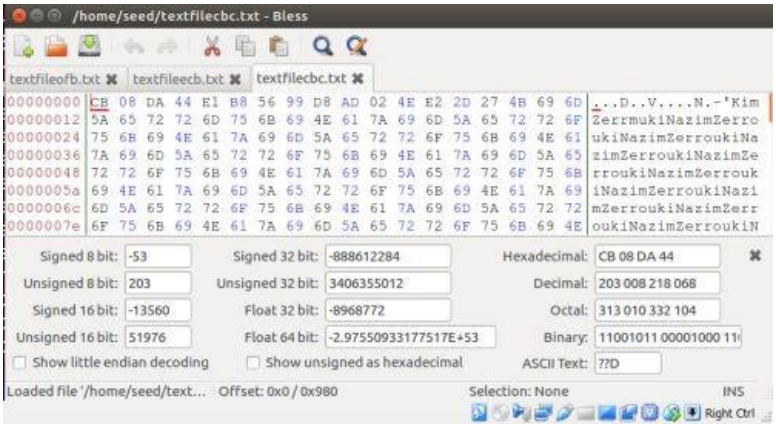
Observation:



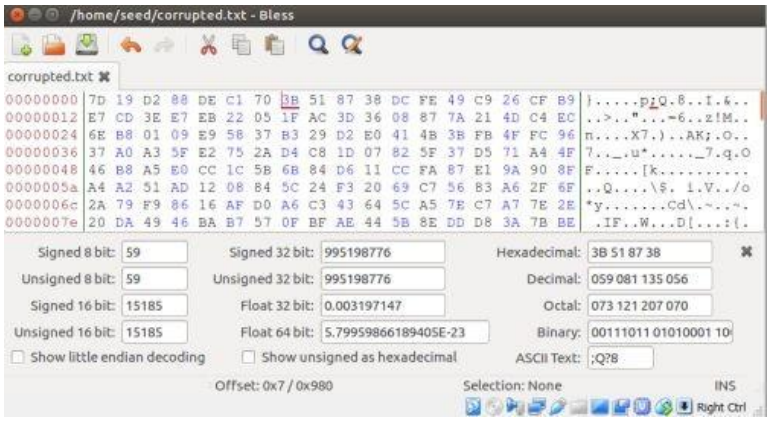
Using ECB to decrypt the corrupted file, we noticed that the first 16 bytes (64 bits) were corrupted while the rest of the bits were completely unaffected.



Observation:

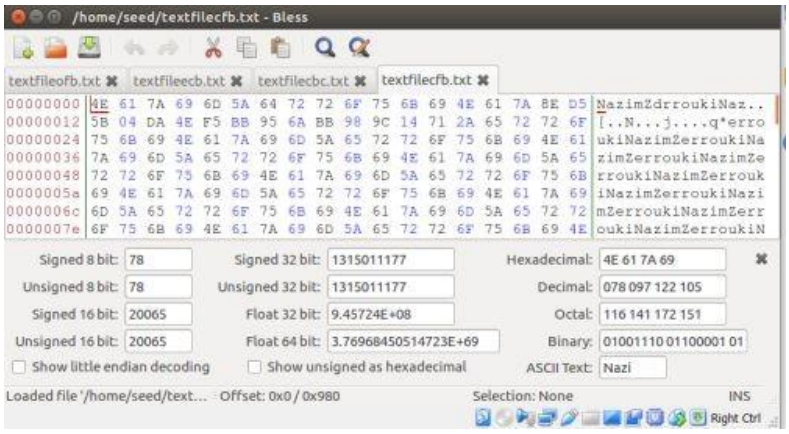


For CBC, after decrypting the corrupted file, we noticed that the majority of the first 23 bytes

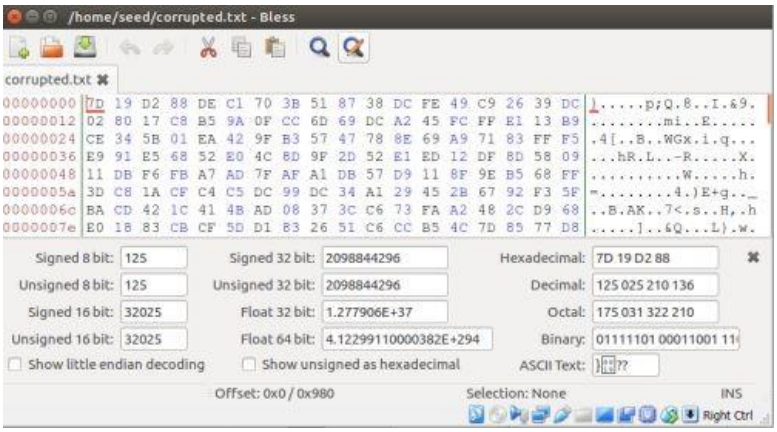


(roughly first 72 bits) were corrupted while the rest of the bits remained unaffected.

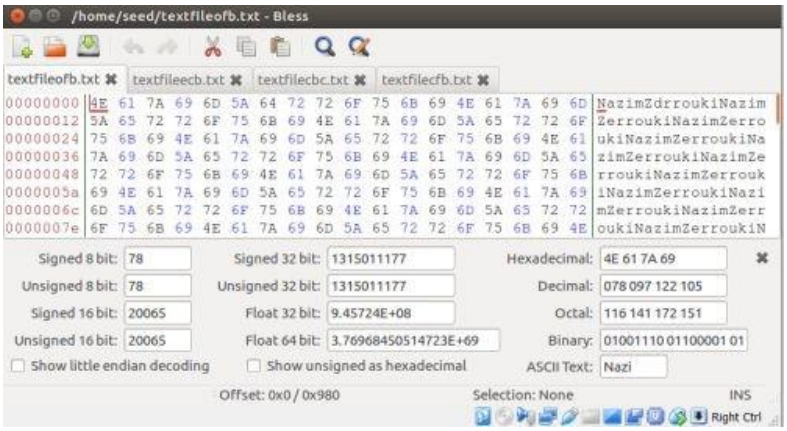
Observation:



When decrypting using CFB, we noticed that the 7th byte was corrupted and bytes 17-21 and bytes 23-32 were corrupted as well. The remaining file was left completely unaffected.



Observation:

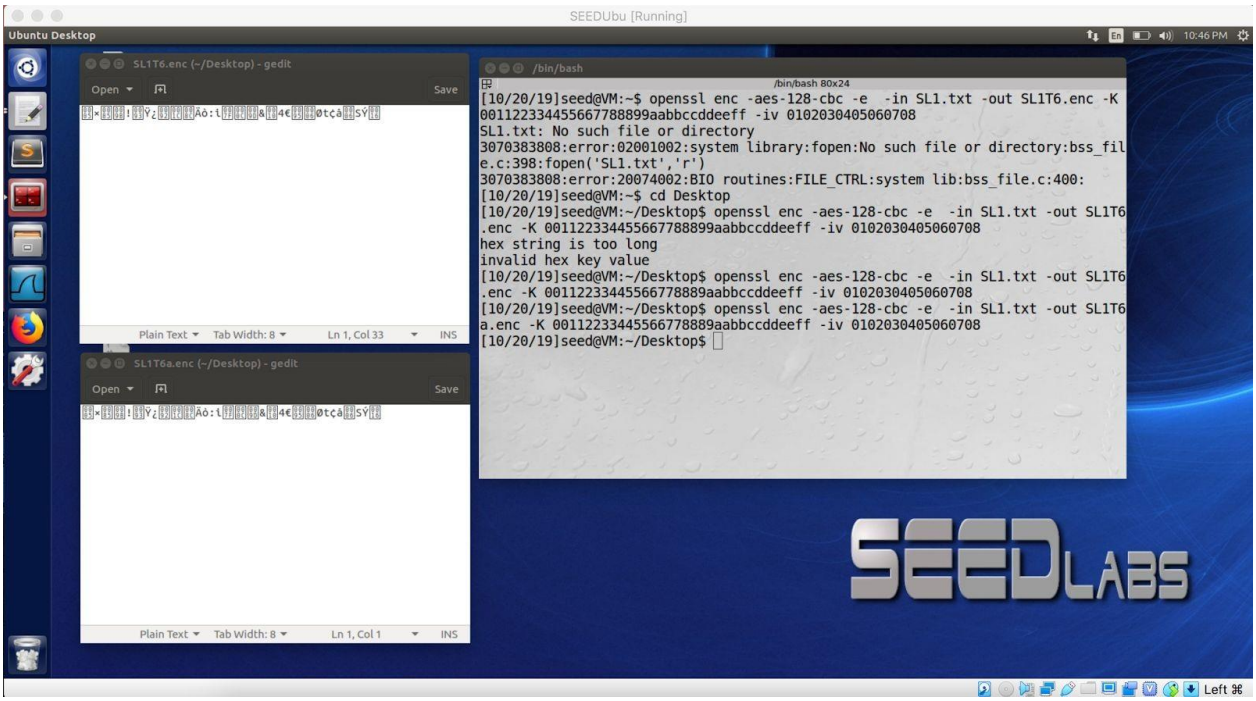


When decrypting using OFB, we noticed that only the 7th byte was corrupted while the rest of the bytes remained intact.

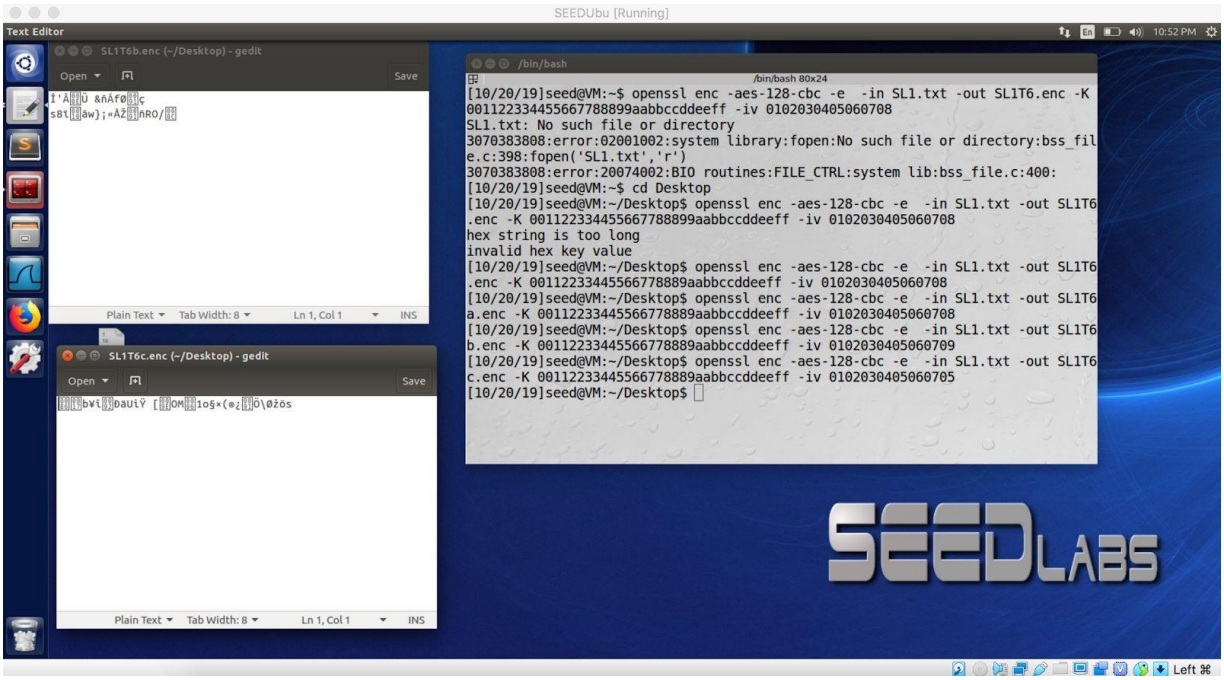
Task 6: Initial Vector (IV) and Common Mistakes

Task 6.1. IV Experiment

Using the same IV twice on the same plaintext file resulted in and identical output for the encryption.



With different IVs the encryption results were completely different.



The IV needs to be unique because if we are using the same plaintext more than once, not having a unique IV each time will result in an identical ciphertext and a pattern will start to emerge if there is an adversary observing our encryption.

Task 6.2. Common Mistake: Use the Same IV

When reusing OFB and same IV for encryption, even if our two plaintext messages to be encrypted are not identical there patterns will start to emerge and the algorithm will become deterministic. IN the case of P1> C1, P2>C2 there are some repeats in the ciphertext between C1 and C2, this can be used to crack the encryption. By looking at the two ciphertext it looks like maybe both plaintext messages end with an '!' mark.

When using CFB the attacker can only know the first block of the message.

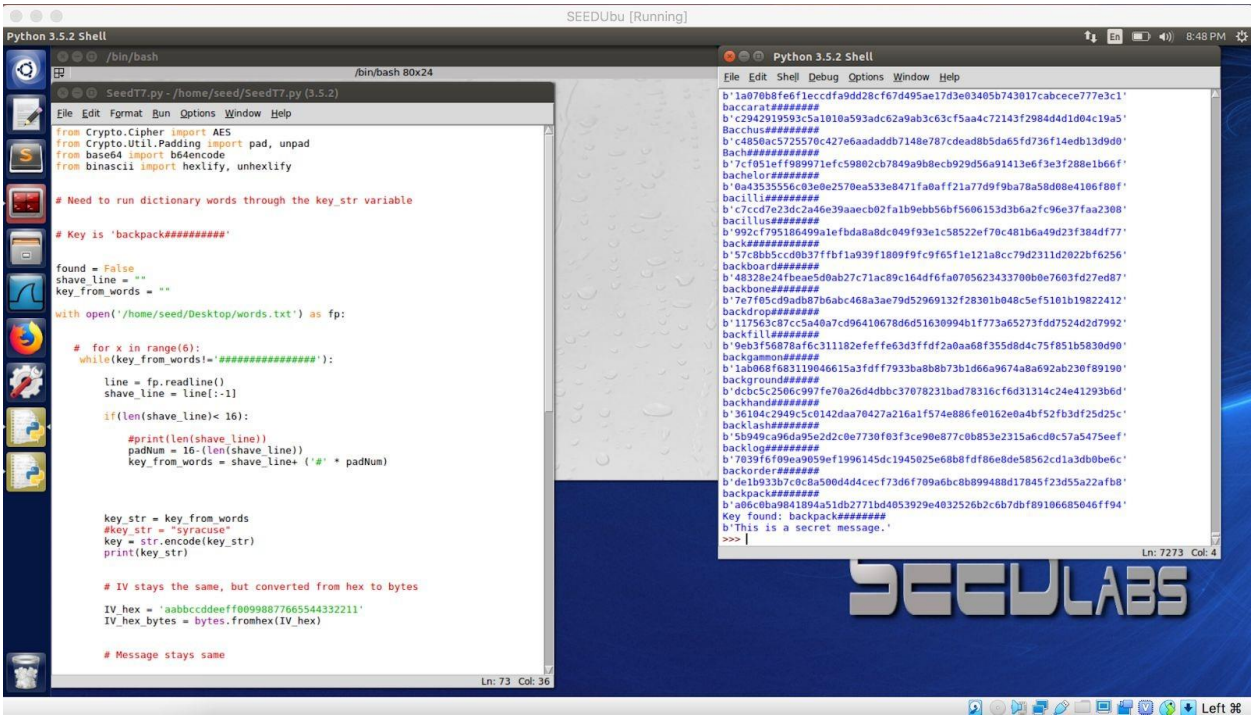
Task 6.3. Common Mistake: Use a Predictable IV

If you know that the plaintext is either 'yes' or 'no' and you need to guess which one it is from cipher text and you can predict the IV. Then it is possible to verify your guess by submitting either 'yes' or 'no' with a known IV.

Task 7: Programming using the Crypto Library

For task 7 we wrote a program in Python using the Pycrypto library formaking a dictionary attack on the AES-128-CBC encryption. The program works and I have demonstrated this below with screenshots. Basically, when given a ciphertext, message, and IV. It will run through a word list + padding them with #s to be 16 bytes long, and when it outputs a match to the desired ciphertext then program ends and then displays the key that generated the result. The only issue was that perhaps due to the different implementation of the encryption library as compared to Openssl, I was unable to generate the same key that was given to us in the lab.
Therefore, I just generated a new ciphertext as the target and ran the program until it matched it. In this regard it worked fine. Please see below images

Program and output after finding the key, word was 'backpack' ('backpack########'). This image shows top half of program



This image shows remaining bottom half of program. And demonstrates that we were able to run a successful dictionary attack on AES-128-CBC encryption when given the IV, message, and ciphertext.