Lab 1

Network Security – Crypto_Encryption

Jeevan Venkataramana

SID : 011917477

Task 1:

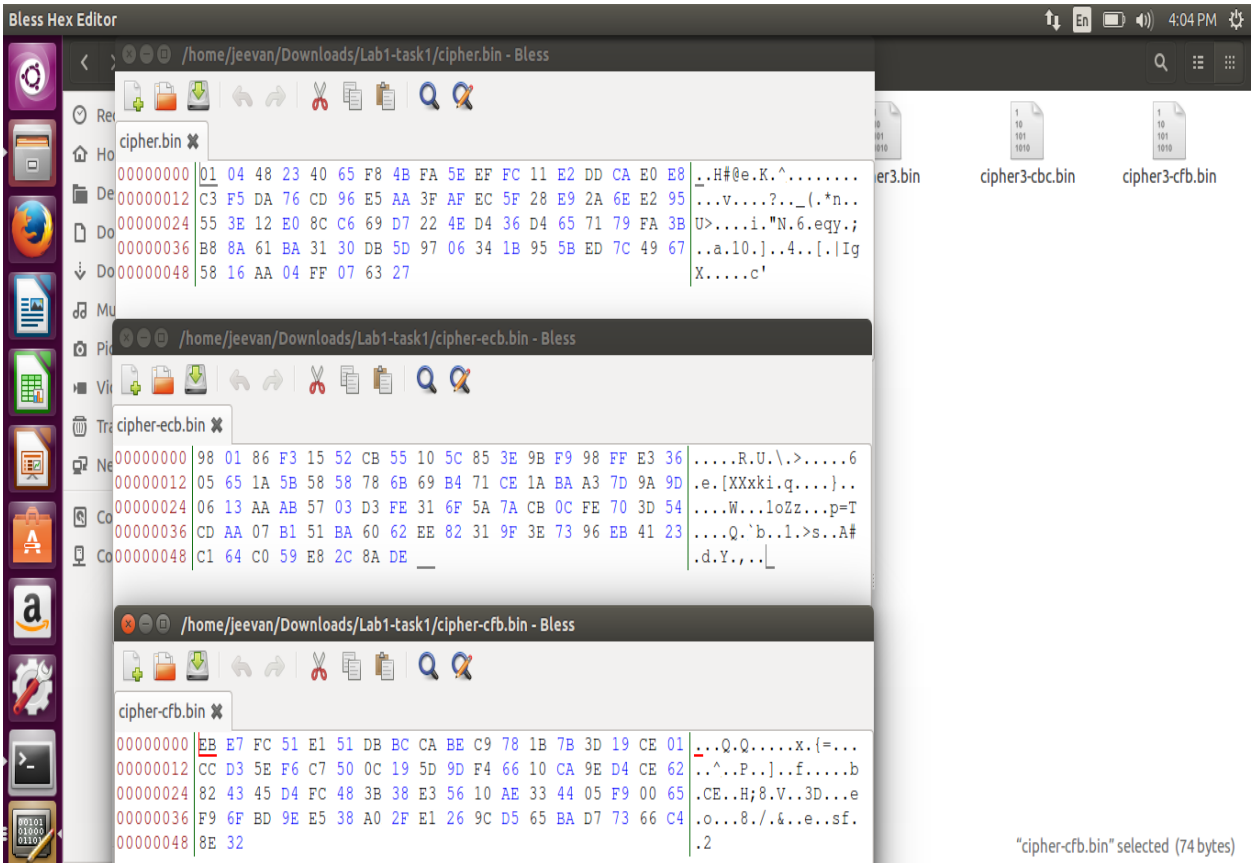
Input file: plaintext.txt

AES Encryption:

Key : 00112233445566778889aabbccddeeff

Iv : 0102030405060708

AES-128 Encryption will all 3 modes.



Commands Executed :

openssl enc -aes-128-cbc -e -in plaintext.txt -out cipher.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708

openssl enc -aes-128-ecb -e -in plaintext.txt -out cipher-ecb.bin -K 00112233445566778889aabbccddeeff

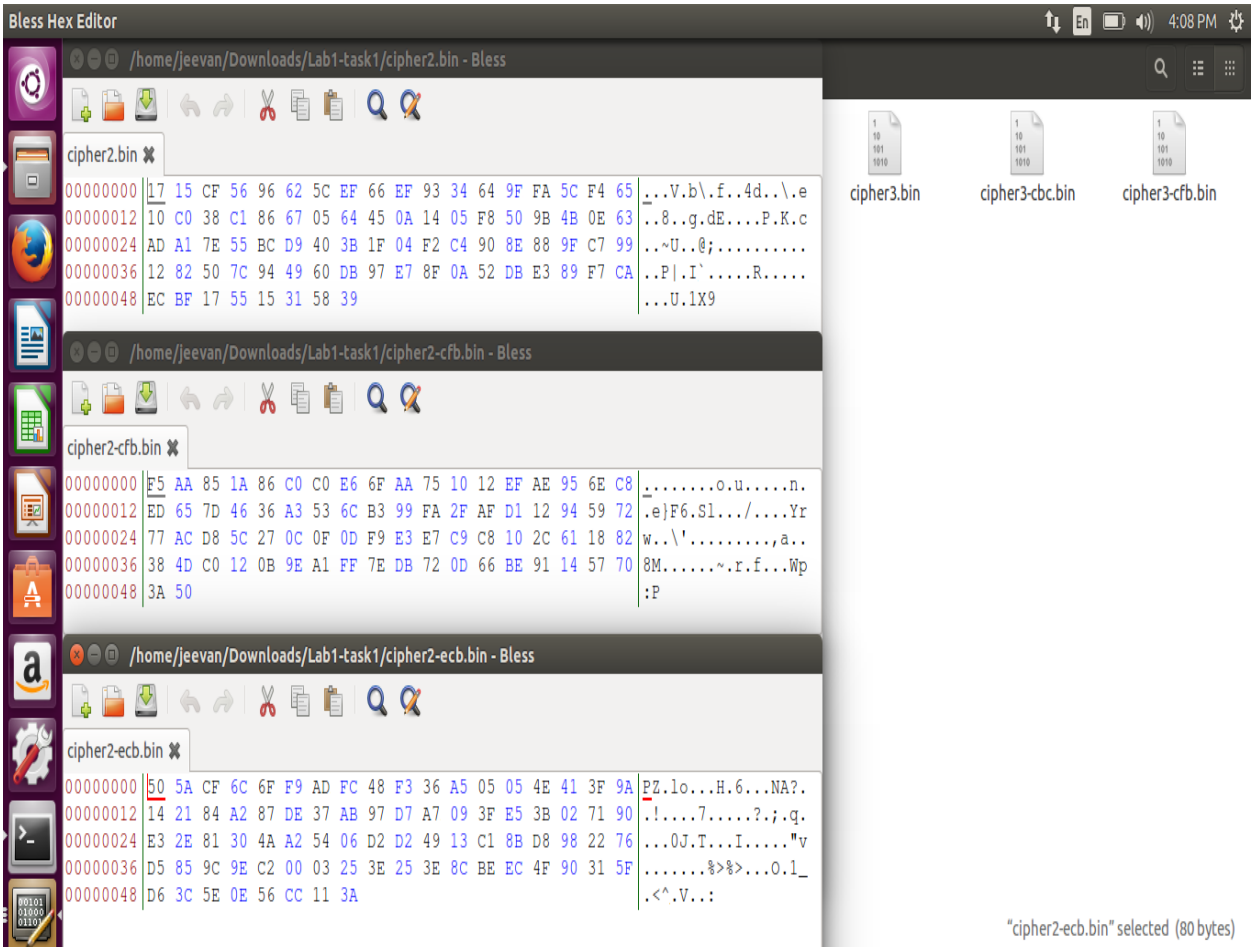
openssl enc -aes-128-cfb -e -in plaintext.txt -out cipher-cfb.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708

DES Encryption:

Key : 00112233445566

Iv : 0102030405060708

DES Encryption with all 3 modes.



Commands Executed:

openssl enc -des-cbc -e -in plaintext.txt -out cipher2.bin -K 00112233445566 -iv 0102030405060708

openssl enc -des-ecb -e -in plaintext.txt -out cipher2-ecb.bin -K 00112233445566

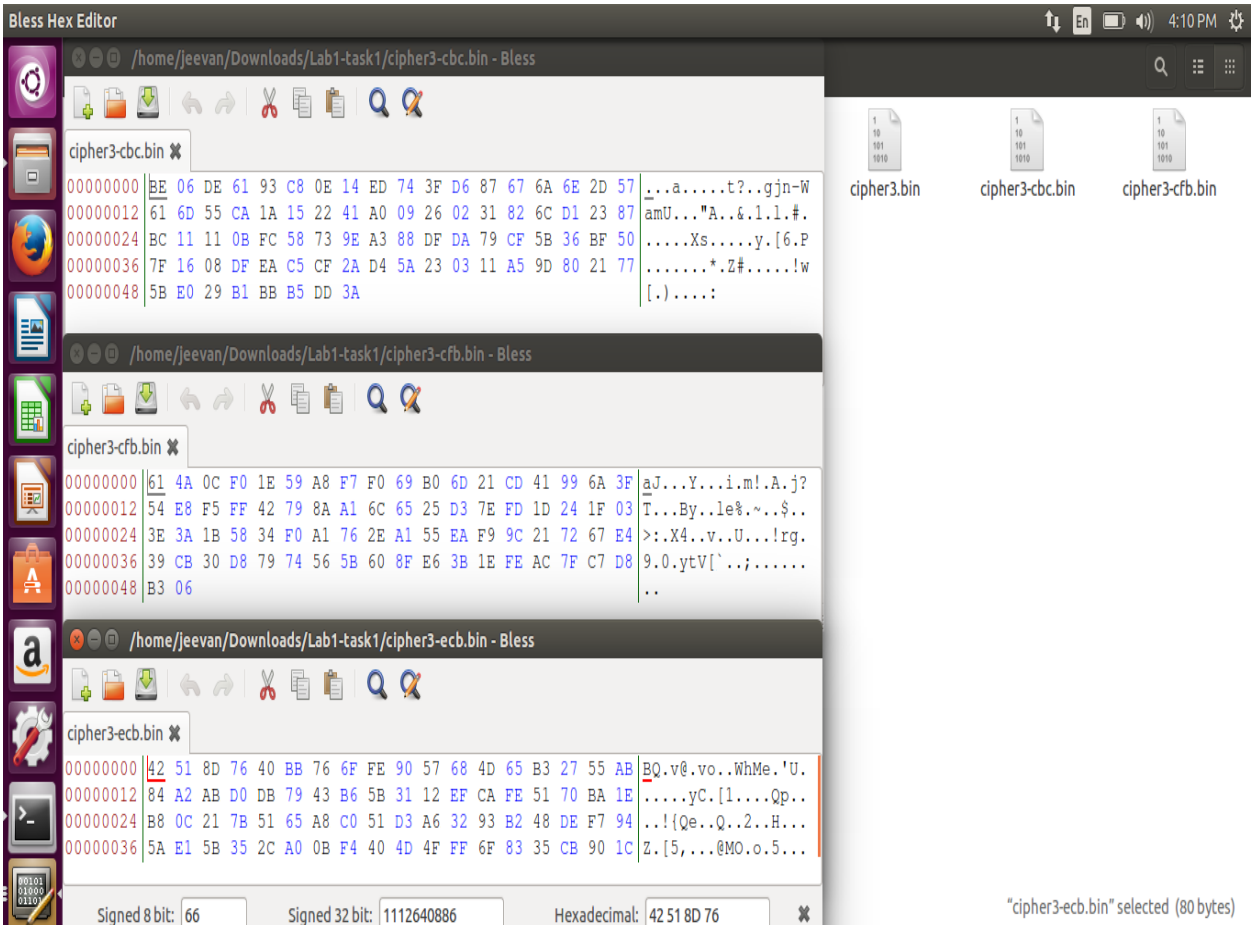
openssl enc -des-cfb -e -in plaintext.txt -out cipher2-cfb.bin -K 00112233445566 -iv 0102030405060708

RC2 Encryption :

Key : 00112233445566

Iv : 0102030405060708

RC2 Encryption with all 3 modes.



Commands Executed:

openssl enc -rc2-cbc -e -in plaintext.txt -out cipher3-cbc.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708

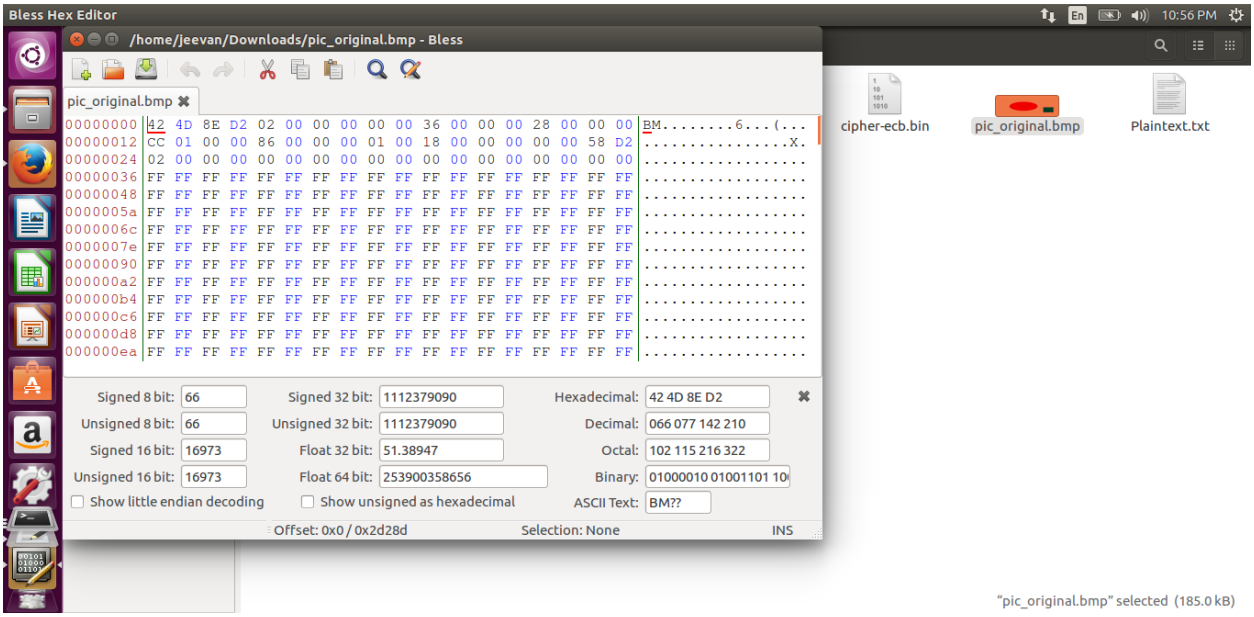
openssl enc -rc2-ecb -e -in plaintext.txt -out cipher3-ecb.bin -K 00112233445566778889aabbccddeeff

openssl enc -rc2-cfb -e -in plaintext.txt -out cipher3-cfb.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708

Task 2:

Original Image: pic_original.bmp

Binary file of pic_original.bmp

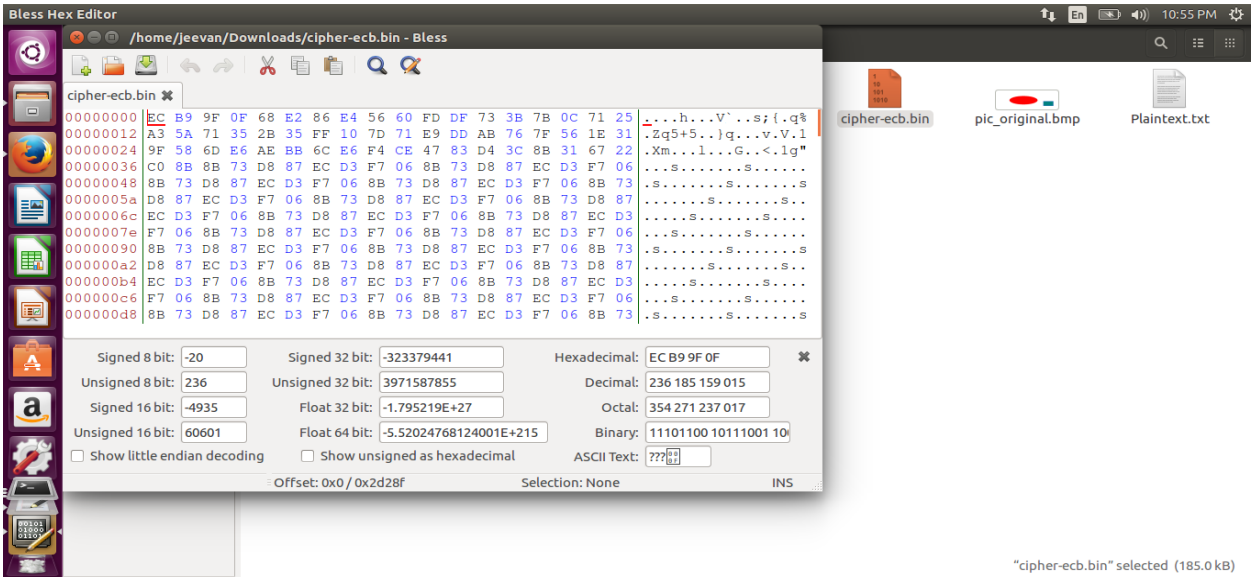


Encryption:

Key: 00112233445566

iv: 01020304050607

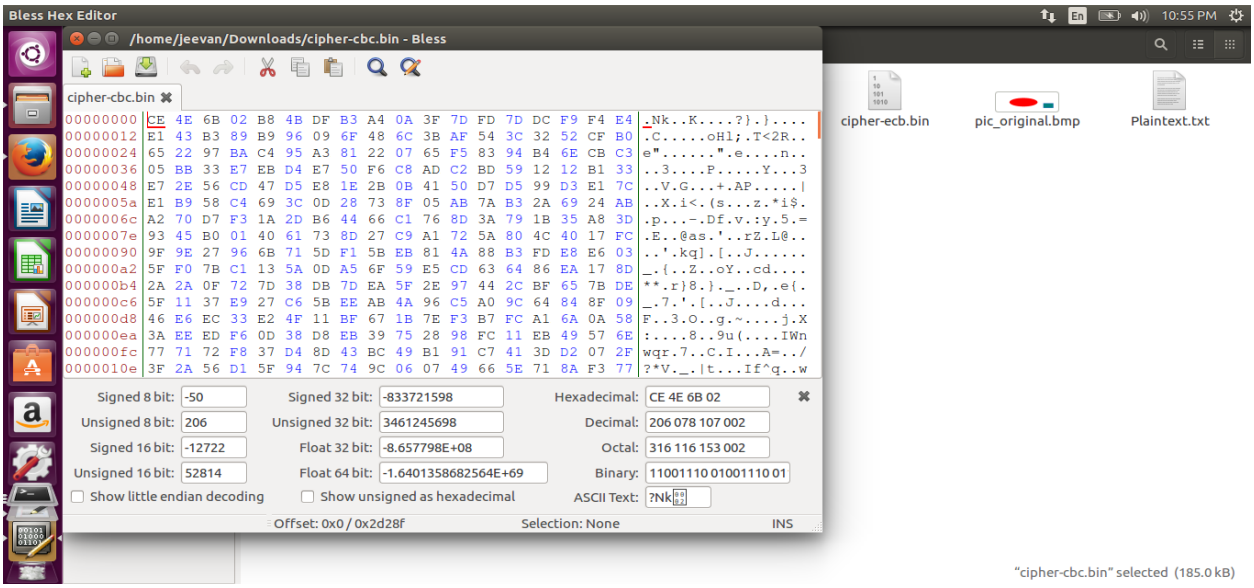
DES – ECB mode:



Command Executed :

openssl enc -des-ecb -e -in pic_original.bmp -out cipher-ecb.bin -K 00112233445566

DES – CBC mode:

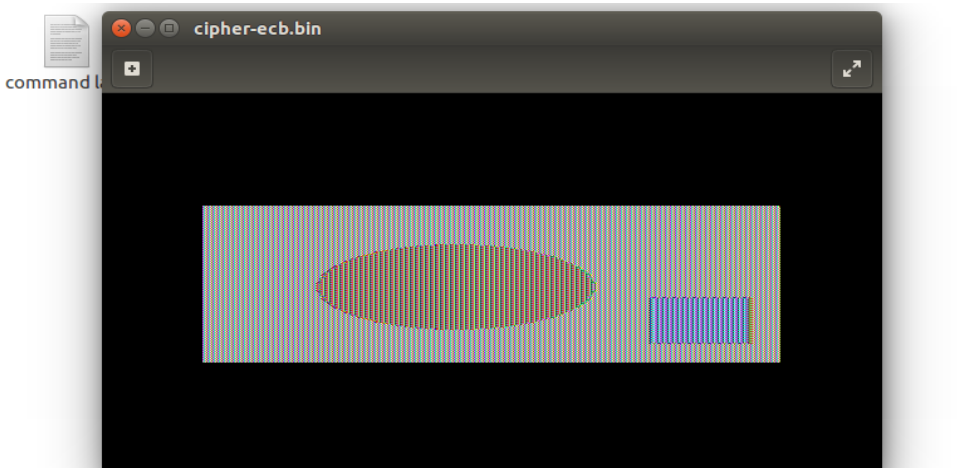


Command Executed:

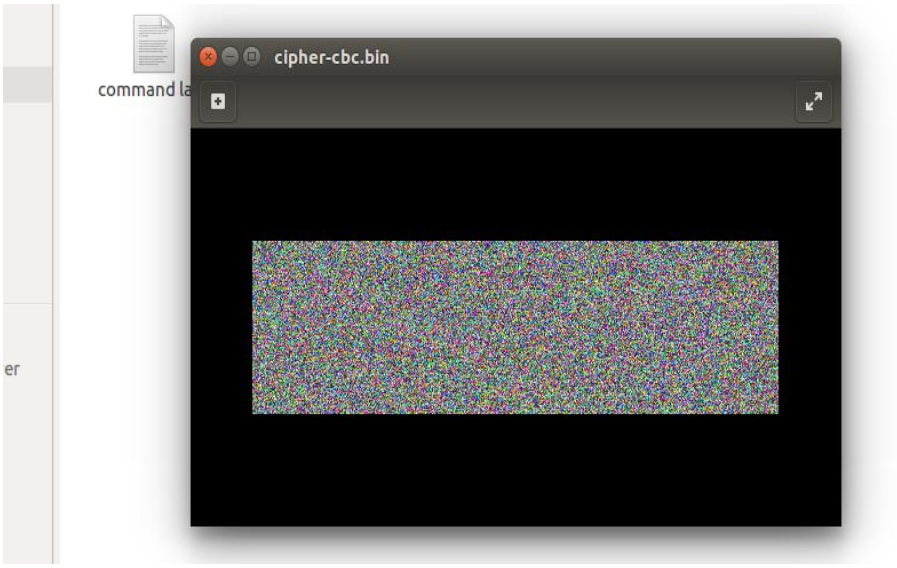
openssl enc -des-cbc -e -in pic_original.bmp -out cipher-cbc.bin -K 00112233445566 -iv 01020304050607

1. Image after replacing header field first 54 bytes of cipher-cbc.bin and cipher-ecb.bin with the first 54 bytes header field of original file pic_original.bmp

cipher-ecb.bin



cipher-cbc.bin



Cannot derive any useful information from the encrypted file in CBC mode. However In ECB mode the overall image may still be discerned as the pattern of identically colored pixels in the original remains in the encrypted version.

Task 3:

1. Text file more than 64 bytes long.

   text file: Task3-input.txt

2. Encryption using AES-128 with all 4 modes.

   Commands Executed:

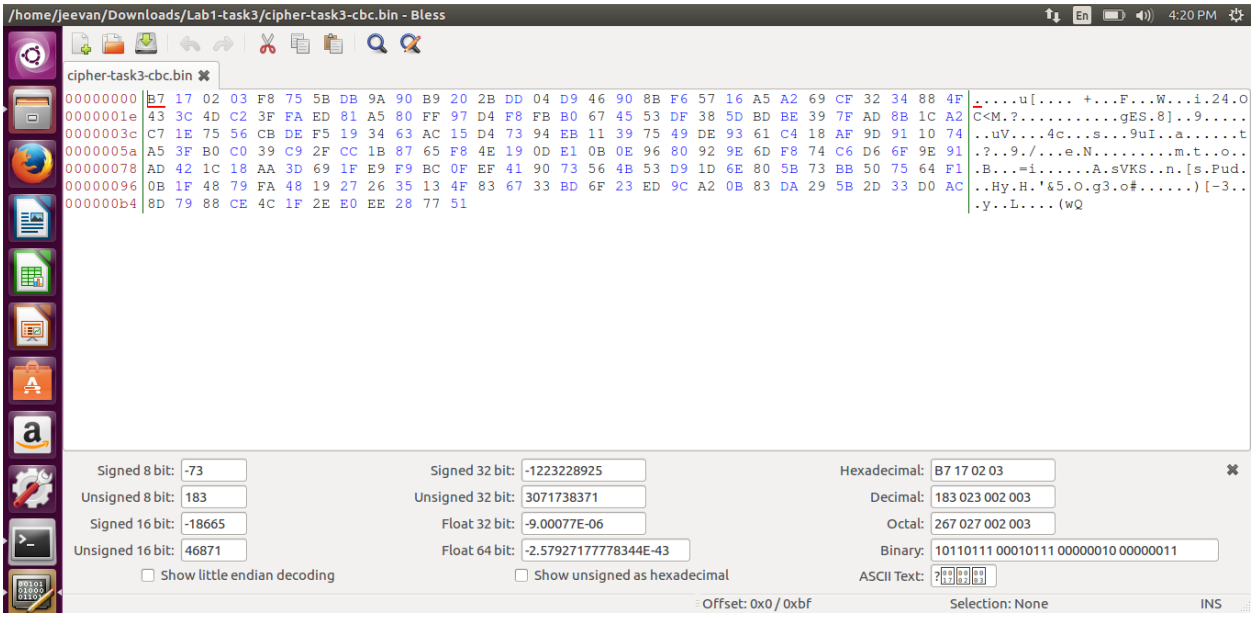
   openssl enc -aes-128-ecb -e -in Task3-input.txt -out cipher-task3-ecb.bin -K 1234512345

   openssl enc -aes-128-cbc -e -in Task3-input.txt -out cipher-task3-cbc.bin -K 1234512345 -iv 5432154321

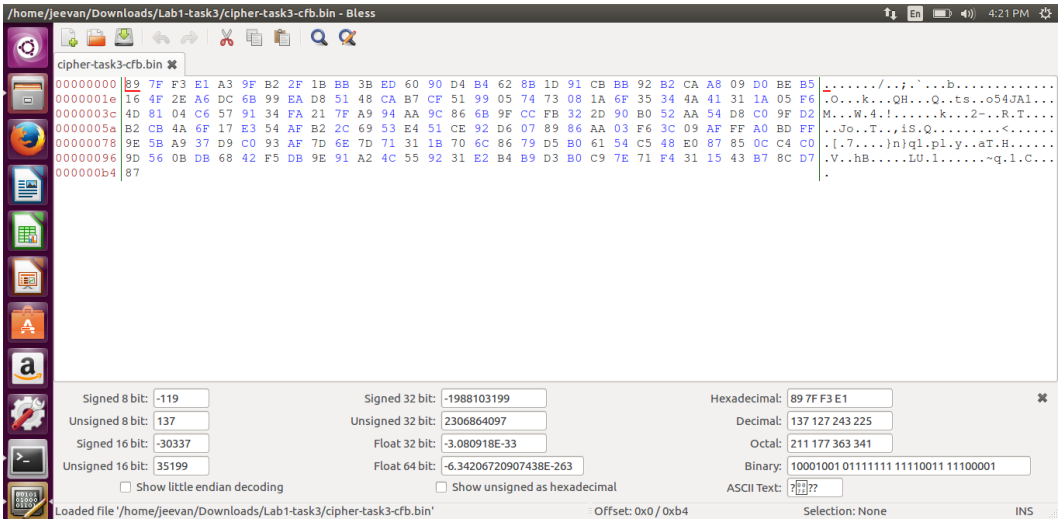
   openssl enc -aes-128-cfb -e -in Task3-input.txt -out cipher-task3-cfb.bin -K 1234512345 -iv 5432154321

   openssl enc -aes-128-ofb -e -in Task3-input.txt -out cipher-task3-ofb.bin -K 1234512345 -iv 5432154321

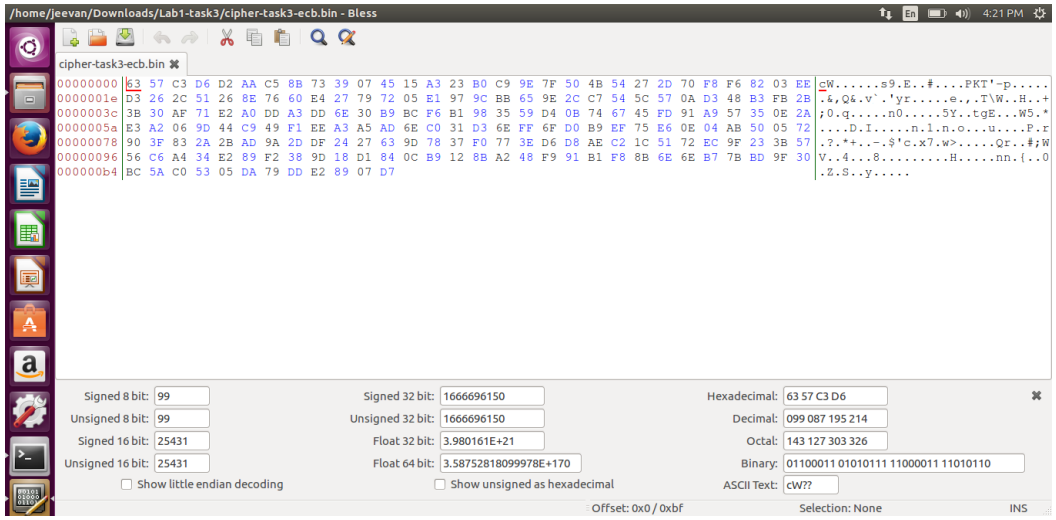
AES-128-CBC



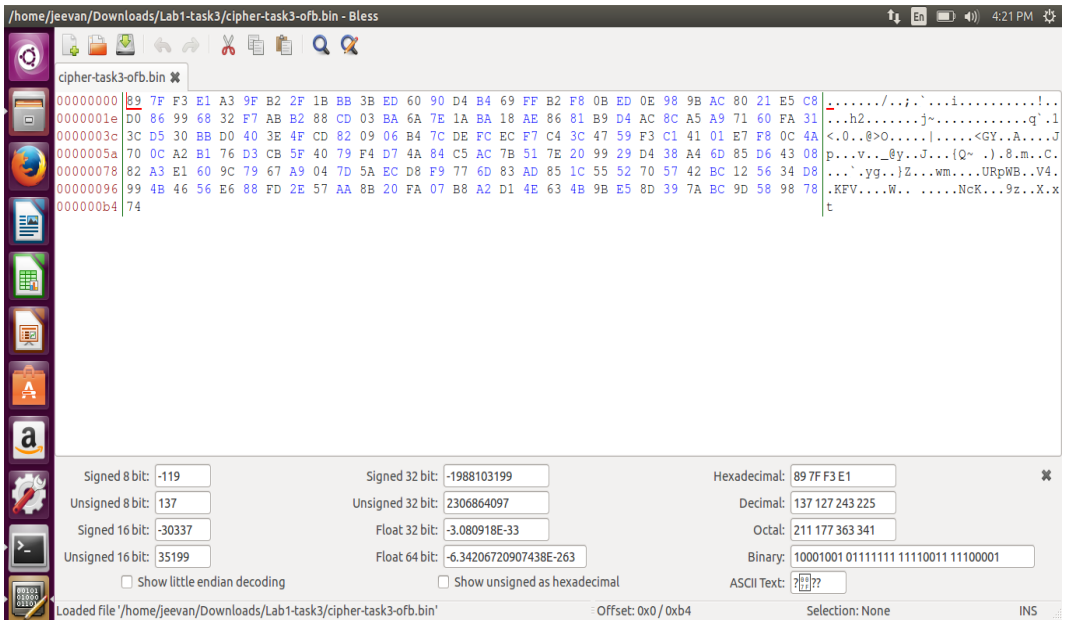
AES-128-CFB



AES-128-ECB

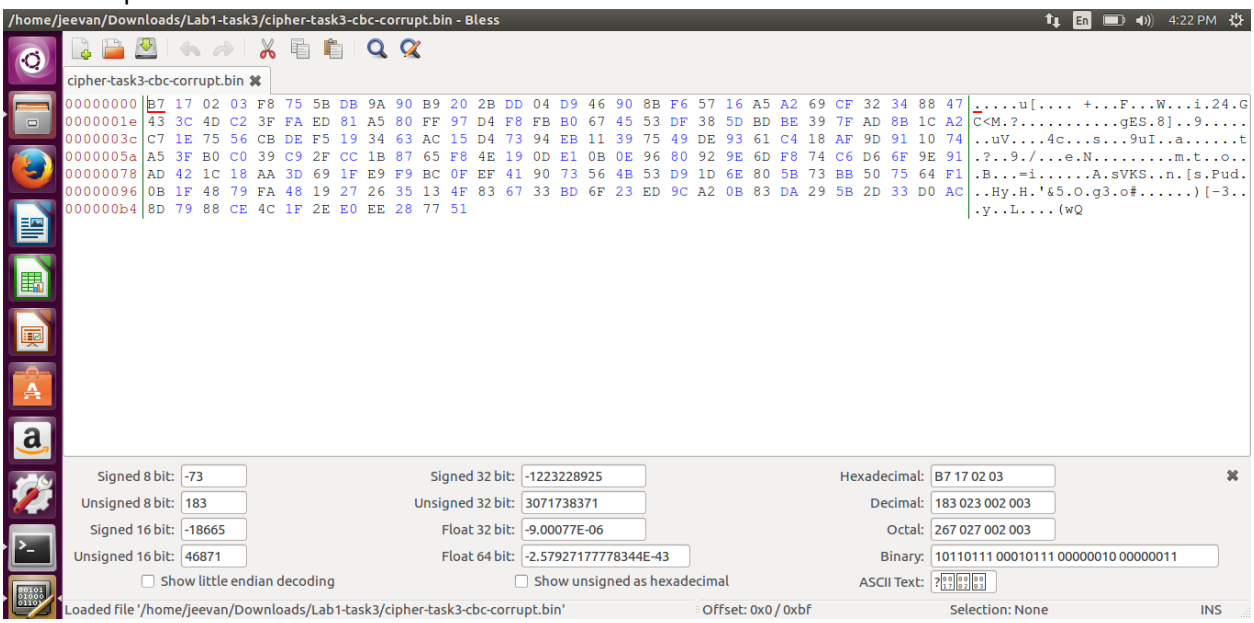


AES-128-OFB

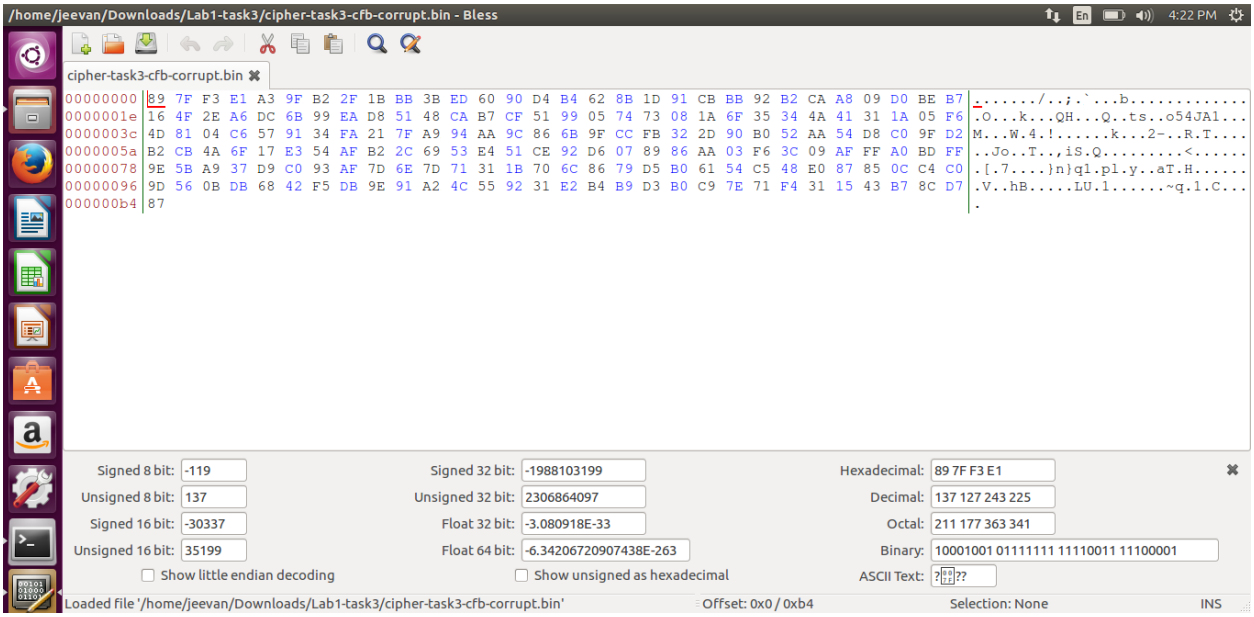


3. Corrupting 30th byte of all encrypted files.

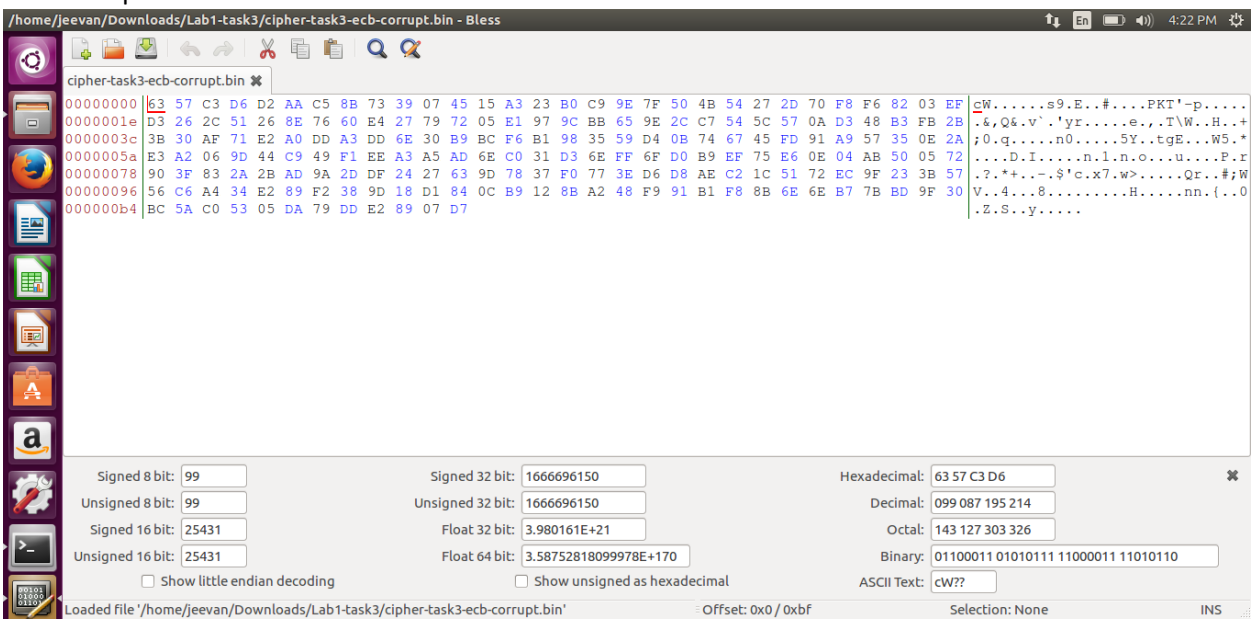
CBC-corrupt



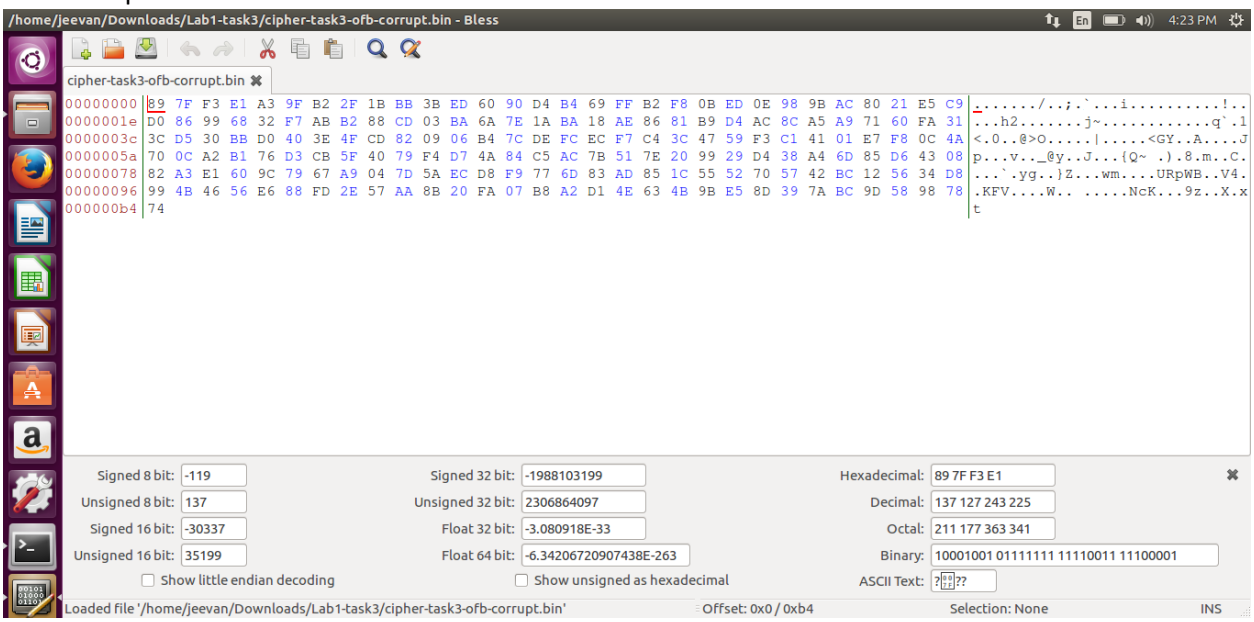
CFB-corrupt



ECB-corrupt

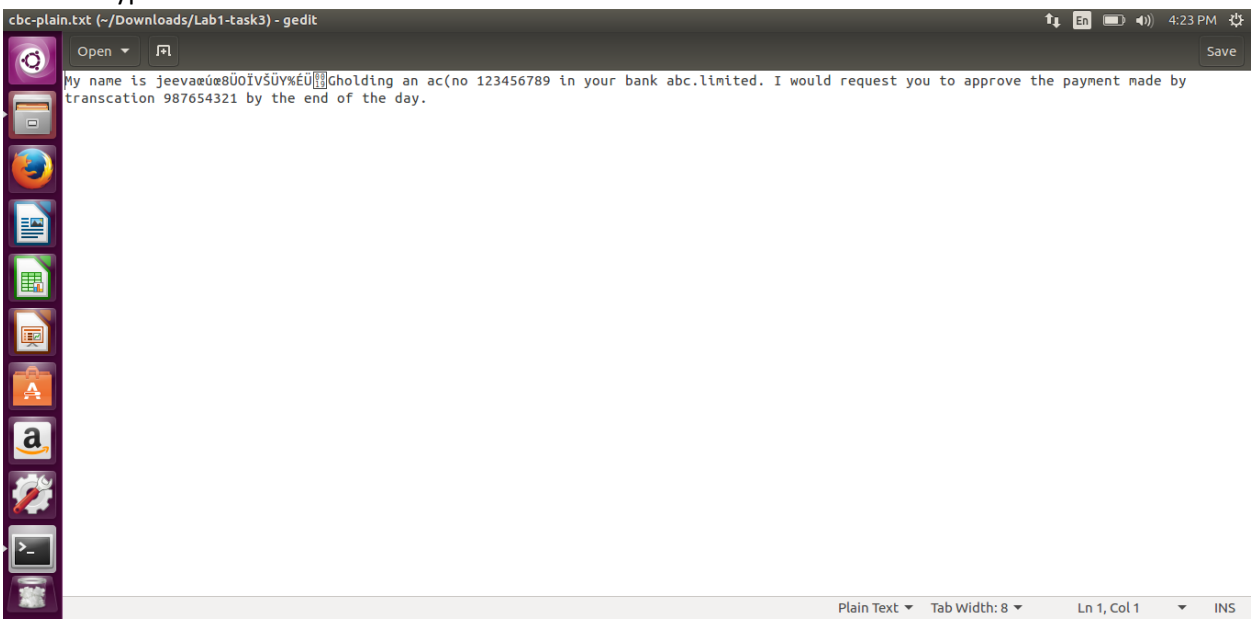


OFB-corrupt

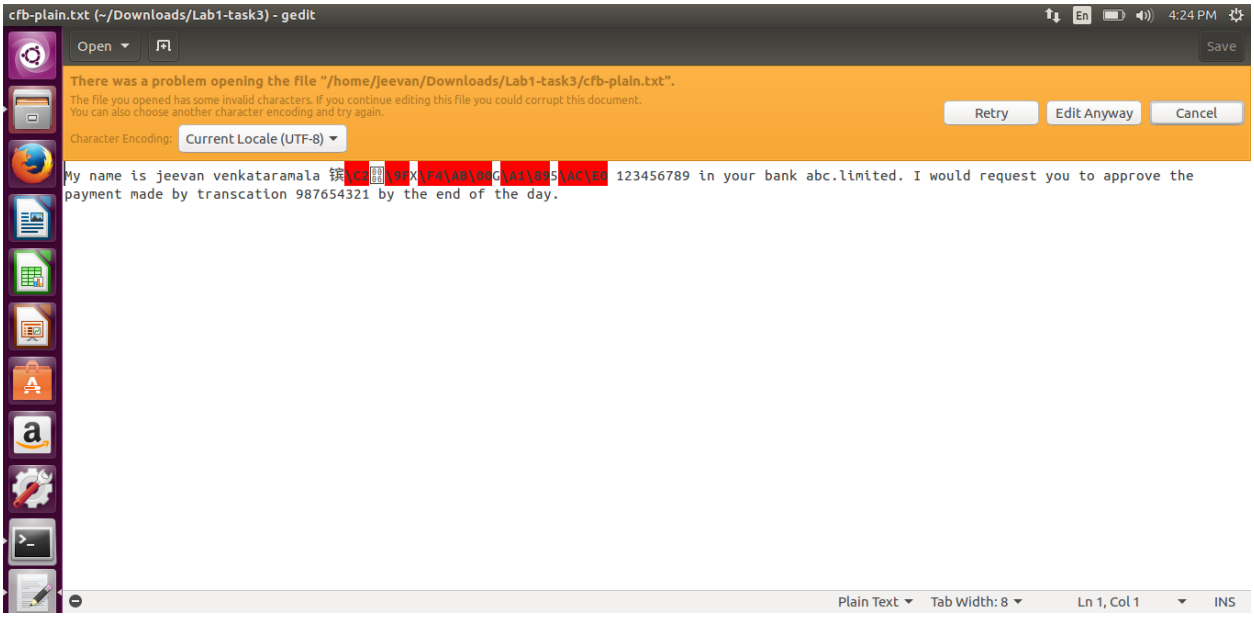


4. Decrypt all above encryption with their correct key and nonce values.

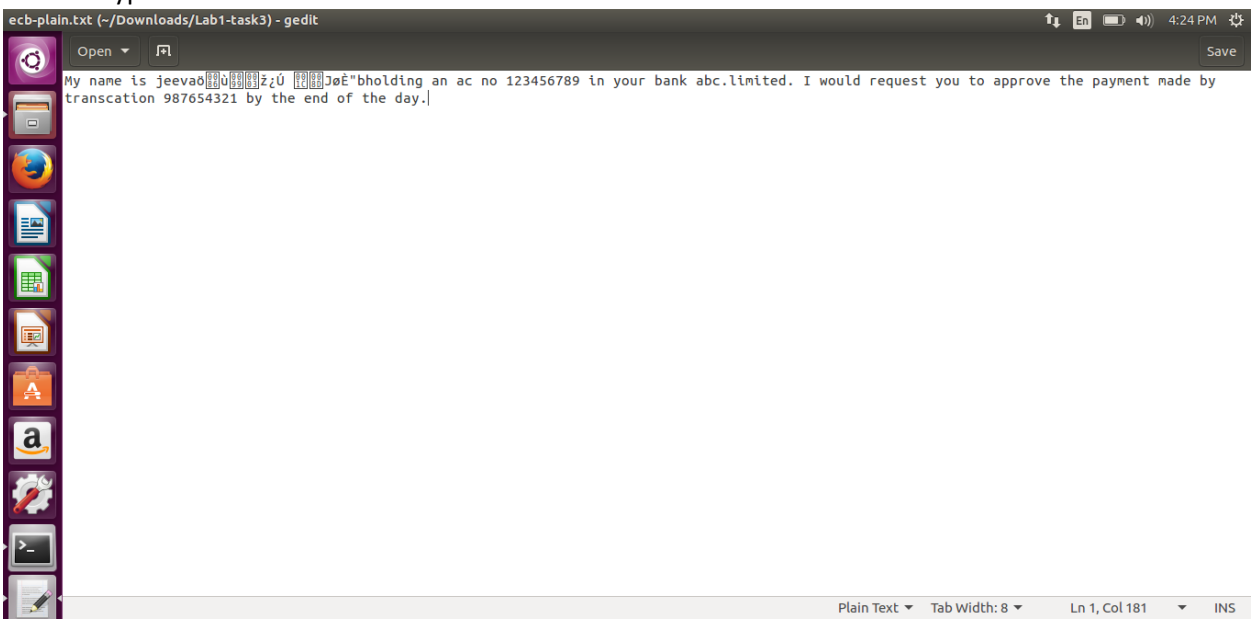
CBC-Decryption



CFB-Decryption



ECB-Decryption



OFB – Decryption

Commands Executed:

openssl enc -aes-128-ecb -d -in cipher-task3-ecb-corrupt.bin -out ecb-plain.txt -K 1234512345

openssl enc -aes-128-cbc -d -in cipher-task3-cbc-corrupt.bin -out cbc-plain.txt -K 1234512345 -iv 5432154321

openssl enc -aes-128-cfb -d -in cipher-task3-cfb-corrupt.bin -out cfb-plain.txt -K 1234512345 -iv 5432154321

openssl enc -aes-128-ofb -d -in cipher-task3-ofb-corrupt.bin -out ofb-plain.txt -K 1234512345 -iv 5432154321

1. How much information can you recover by decrypting the corrupted file, if the encryption mode is ECB, CBC, CFB, or OFB, respectively?

Sol.

Before Task:

ECB: Data will be encrypted in blocks. The encrypted block whose bit is corrupted should map to some other plaintext. Resulting in failure to recognize only that plaintext block.

CBC: Bit change to the cipher text should cause complete corruption of the corresponding block of plaintext and also invert the corresponding bit in the following block of plaintext, but rest of the blocks should remain intact.

CFB: One-bit change in the ciphertext should affect two plaintext blocks: a one-bit change in the corresponding plaintext block, and complete corruption of the following plaintext block. Rest of the blocks intact.

OFB: One-bit change in ciphertext should affect only corresponding single bit in plaintext.

After Task:

ECB: Satisfies as stated. Each block is of size 16 bytes and the block containing corrupted bit has been mapped to a random plaintext.

CBC: Satisfies as stated. The corresponding block in plaintext is corrupted as well as the corresponding bit in next plaintext block. Where a space is mapped to symbol "(".

CFB: Satisfies as stated. The corresponding corrupt bit is mapped to a different character in plaintext and the next block is also mapped to some other plaintext.

OFB: Only one bit has changed also satisfies as expected.

2) Reasons:

ECB: It decrypts identical cipher blocks in to identical plaintext blocks and each block is independent of other block.

CBC: Corrupt bit block should map to some other plain text. And the corresponding bit in next block also corrupted because the corrupted ciphertext block is fed as input.

CFB: Next block of corresponding corrupted block maps to some other plaintext because the corrupted ciphertext block is fed as input to encryption along with key.

OFB: Error does not propagate because each decryption block is independent of ciphertext of previous block. Which results in error of the only bit that is corrupted.

3) Implications:

Single bit error in ciphertext would result in 1 corrupted block in ECB, 2 blocks in CBC and CFB(1 whole block and 1 bit in other block). And only a single bit in OFB.

Error propagation: ECB and OFB do not propagate error. CBC and ECB does.

Lab 2

Crypto Lab – One-Way Hash Function and MAC

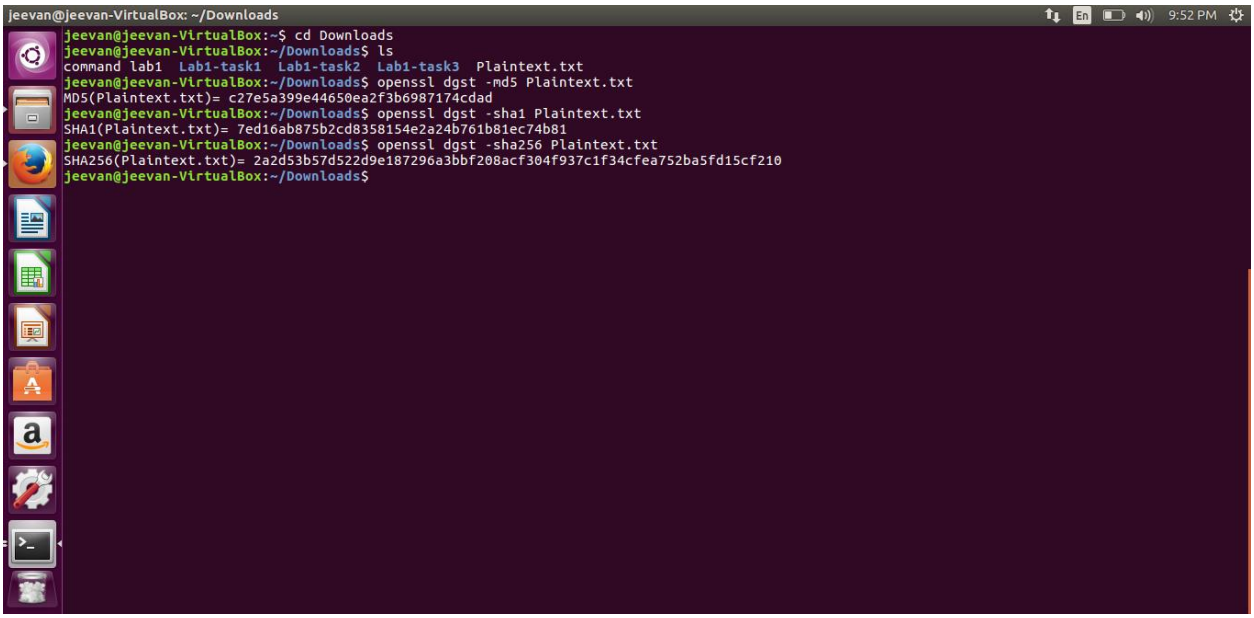
Task 1: Generating Message Digest and MAC

In this task, we will play with various one-way hash algorithms. You can use the following openssl dgst command to generate the hash value for a file. To see the manuals, you can type man openssl and man dgst.

% openssl dgst dgsttype filename

Please replace the dgsttype with a specific one-way hash algorithm, such as -md5, -sha1, -sha256, etc. In this task, you should try at least 3 different algorithms, and describe your observations. You can find the supported one-way hash algorithms by typing "man openssl".

observations



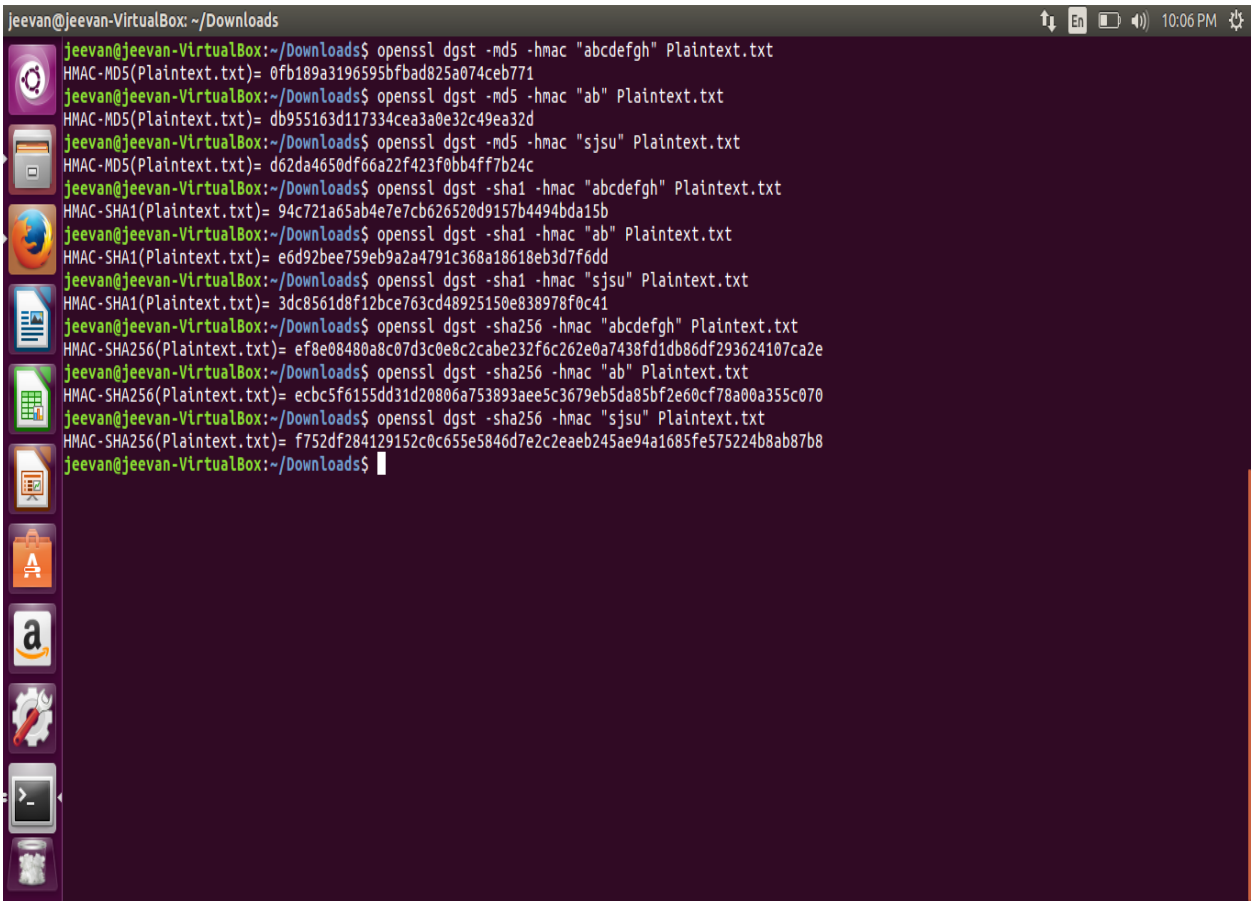
I have created a file called “Plaintext.txt” in Downloads directory. After that, we’ve implemented the three algorithms, md5, sha-1 and sha-256. The results can be seen in the figure.

From the results, it can be inferred that the length of hash value generated by md5, sha-1 and sha-256 are 32, 40 and 64 respectively. It follows from the fact that hash value of md5, sha-1 and sha-256 are 128, 140 and 256 bits each. It can also be inferred that SHA-256 algorithm is the most secure of all the three algorithms.

Task 2: Keyed Hash and HMAC

Please generate a keyed hash using HMAC-MD5, HMAC-SHA256, and HMAC-SHA1 for any file that you choose. Please try several keys with different length. Do we have to use a key with a fixed size in HMAC? If so, what is the key size? If not, why?

Observation:



For this task, we created a keyed hash for Plaintext.txt file. We generated several keyed hash values using HMAC –md5, HMAC –sha1 and HMAC -256 with several different keys. The results of this can be seen in the figure above.

From the results above, it is clear that HMAC values length is same with the corresponding hash code length. Based on this, it is easily established that we do not need to use a key with fixed size and in any way the HMAC algorithm is quite flexible.

Task 3: The Randomness of One-way Hash

To understand the properties of one-way hash functions, we would like to do the following exercise for MD5 and SHA256

1. Create a text file of any length.

2. Generate the hash value H1 for this file using a specific hash algorithm.

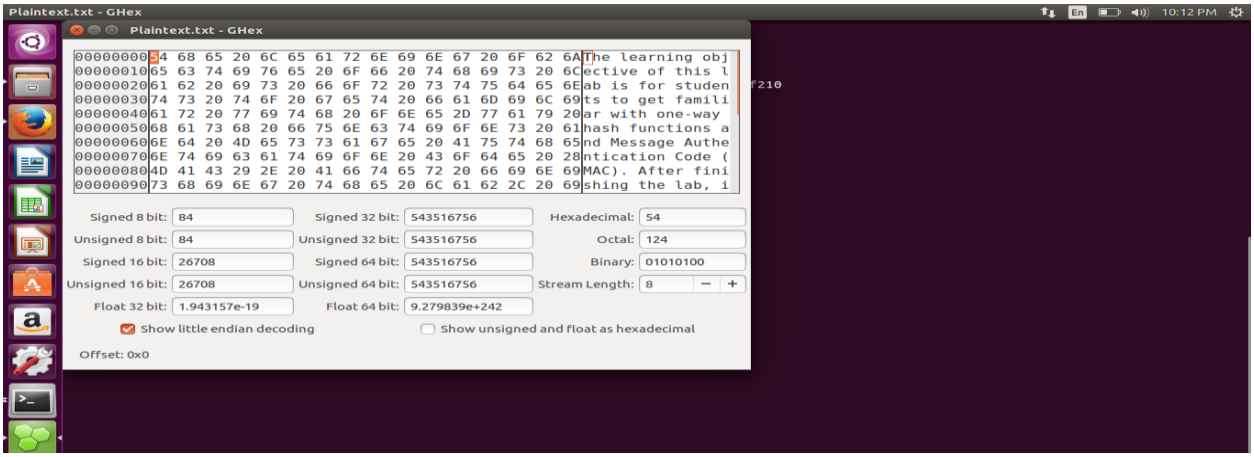
3. Flip one bit of the input file. You can achieve this modification using ghex or Bless.

4. Generate the hash value H2 for the modified file.

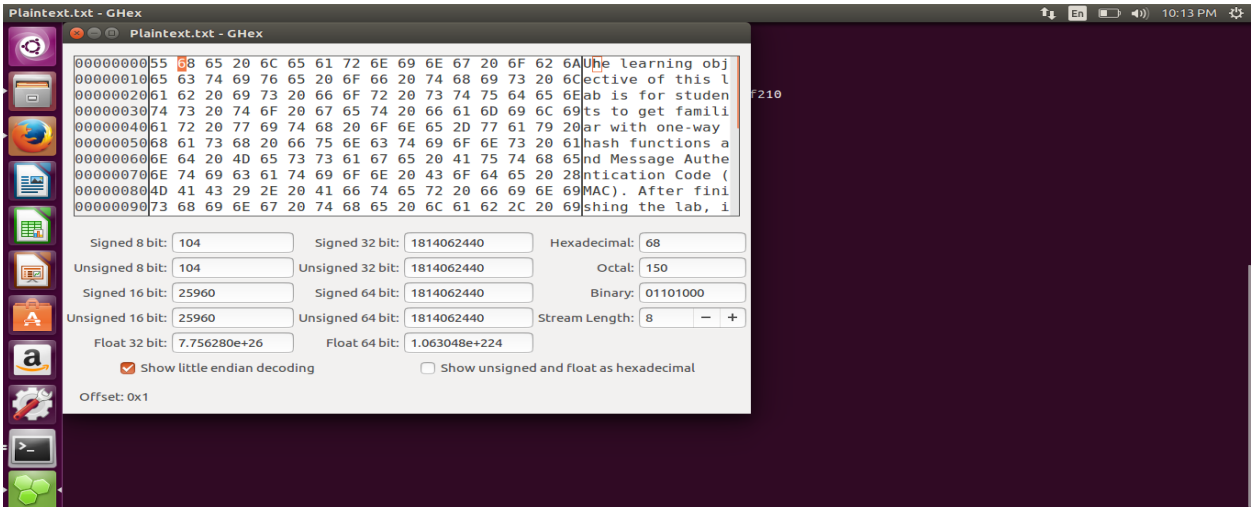
5. Please observe whether H1 and H2 are similar or not. Please describe your observations in the lab report. You can write a short program to count how many bits are the same between H1 and H2.

Observation:

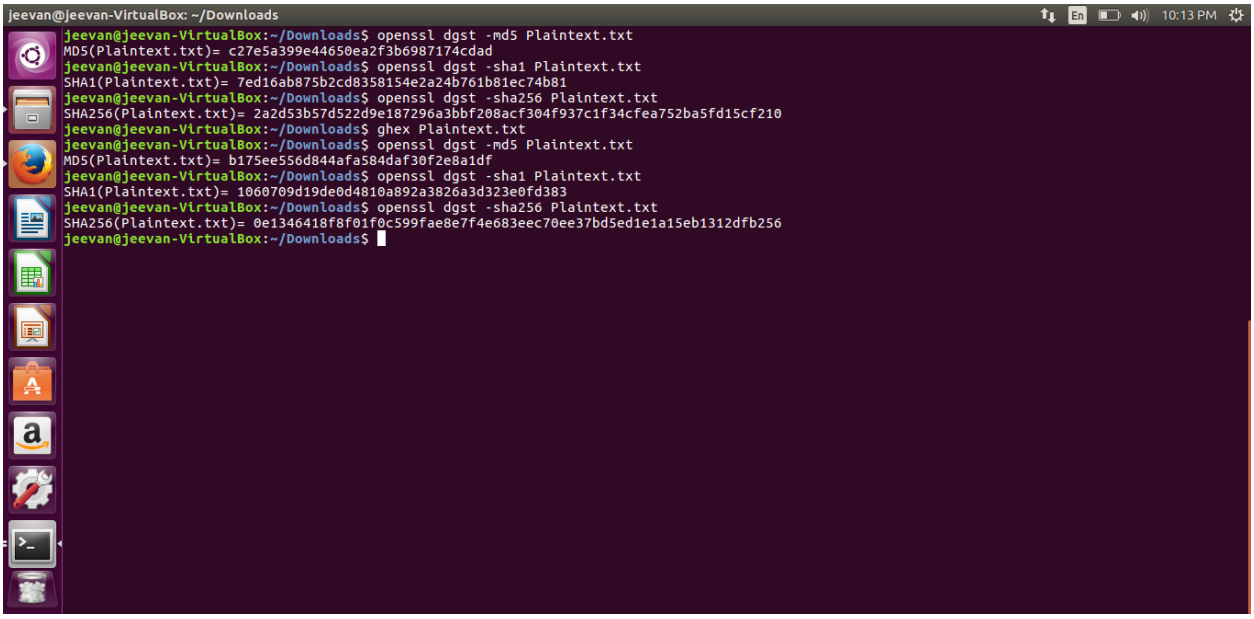
Original file:



After flip:



Comparison before and after flip



Generated hash values for plaintext.txt. Flipped a single bit of first byte and again generated hash value. It can be concluded a single bit change produces a completely different hash value. H1 and H2 are not similar.

md5

H1: c27e5a399e44650ea2f3b6987174cdad

H2: b175ee556d844afa584daf30f2e8a1df

no of similar bits: 61

sha1

H1: 7ed16ab875b2cd8358154e2a24b761b81ec74b81

H2: 1060709d19de0d4810a892a3826a3d323e0fd383

No of similar bits: 89

sha256

H1: 2a2d53b57d522d9e187296a3bbf208acf304f937c1f34cfea752ba5fd15cf210

H2: 0e1346418f8f01f0c599fae8e7f4e683eec70ee37bd5ed1e1a15eb1312dfb256

No of bits similar: 125

Program: To count similar bits.

```
/* Created by Jeevan Venkataramana */

#include<iostream>
#include<string>
int to_decimal(char a);
using namespace std;
int main()
{
string a, b;
cout<<endl<<"enter hash 1: ";
cin>>a;
cout<<endl<<"enter hash 2: ";
cin>>b;
int n= a.length();
int count=0;
for(int i=0; i<n; i++)
{
	int x= to_decimal(a[i]);
	int y= to_decimal(b[i]);
	int xor_result = x^y;

	for(int j=0; j<4; j++)
	{
		if(xor_result % 2 == 0)
		{
			count++;
		}
		xor_result = xor_result/2;
	}
}
cout<<endl<<" No of bits similar: "<<count;
return 0;
}

int to_decimal(char a)
{
	if(a =='0')return 0;
	if(a =='1')return 1;
	if(a =='2') return 2;
	if(a =='3') return 3;
	if(a =='4') return 4;
	if(a =='5') return 5;
	if(a =='6') return 6;
	if(a =='7') return 7;
	if(a =='8') return 8;
	if(a =='9') return 9;
	if(a =='a') return 10;
	if(a =='b') return 11;
	if(a =='c') return 12;
	if(a =='d') return 13;
	if(a =='e') return 14;
	if(a =='f') return 15;
}
```

Task 4: One-Way Property versus Collision-Free Property

1) How many trials it will take you to break the one-way property using the brute-force method? You should repeat your experiment for multiple times and report your average number of trials.

   For any given h, computationally infeasible to find y such that H(y) = h With m-bit hash code, effort required proportional to 2^m We have 24 bit hash code, effort required is 2^24 (theoretically)

2) How many trials it will take you to break the collision-free property using the brute-force method? Similarly, you should report the average.

   It is hard to find two inputs that hash to the same output; that is, two inputs a and b such that H(a) = H(b) With m-bit hash code, effort required proportional to 2^(m/2). We have 24 bit hash code, effort required is 2^12 (theoretically).

3) Based on your observation, which property is easier to break using the brute-force method?

   collision resistant is easier to break then one way property

```python
import numpy as np


number = 0
hashfile = []


    # initializing with 32 zeroes
h = np.binary_repr(0).zfill(24)
for m in range(0,1000):
    for i in range(0, m):
        # converting
        a = int(h[16:], 2)
        # xoring least significant byte with byte read
        b1 = np.bitwise_xor(int(h[16:], 2), m)
        # converting integer into binary string
        b2 = "{0:b}".format(b1)
        # padding with 0's on left until 8 bits
        c = b2.zfill(8)
        # adding the converted bit to
        h = h[0:16] + c

        # left circular shift by 4 bits
        h = h[4:] + h[0:4]
    hashfile.append(h)
    number = number + 1
# checking for collisions
count=0
for k in range(0, 1000):
    for l in range(k + 1, 1000):
        if hashfile[k] == hashfile[l]:
            count = count + 1

print("number of two way collsiion")
print(count)

count1=0
for l in range(1, 1000):
    if hashfile[55] == hashfile[l]:
        count1 = count1 + 1
print("number of one way collsiion")
print(count1)
```

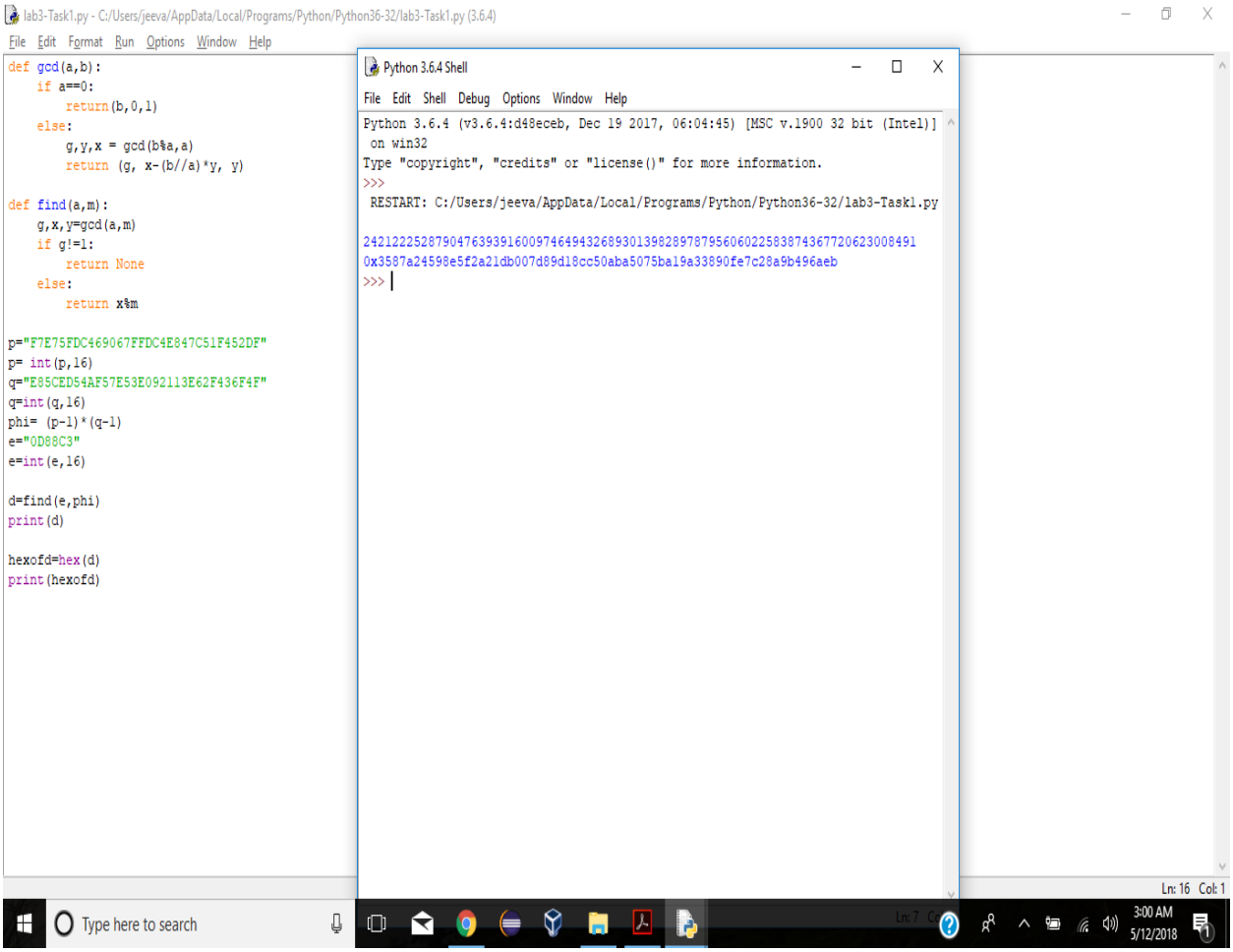
```
Usage

Here you can get help of any object by pressing Ctrl+I
in front of it, either on the Editor or the Console.

Help can also be shown automatically after writing a left
parenthesis next to an object. You can activate this
behavior in Preferences > Help.

New to Spyder? Read our tutorial

Variable explorer  File explorer  Help

IPython console

Console 1/A

Python 3.6.3 |Anaconda custom (64-bit)| (default, Oct 15 2017, 03:27:45) [MSC v.1900 64
bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 6.1.0 -- An enhanced Interactive Python.
number of two way collsiion
43
number of one way collsiion
4

In [2]:

In [2]:

IPython console  History log
```

Numbers are from 0 to 1000, Two-way collision are 43. Thus, one-way property is satisfied on four instances.

Lab 3

RSA Public-Key Encryption and Signature Lab

Task1: Deriving the Private Key



Private key:

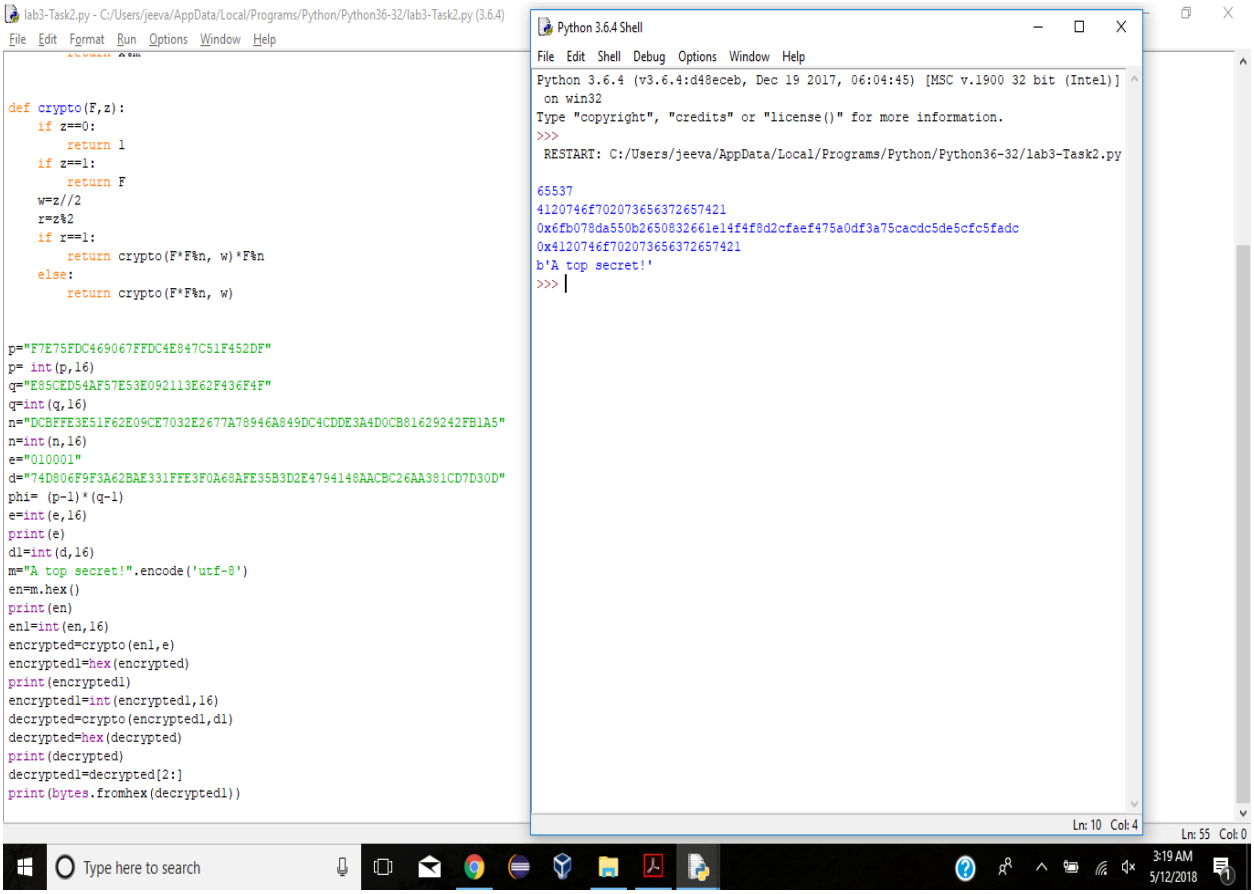
Integer

d = 24212225287904763939160097464943268930139828978795606022583874367720623008491

Hexadecimal

d = 0x3587a24598e5f2a21db007d89d18cc50aba5075ba19a33890fe7c28a9b496aeb

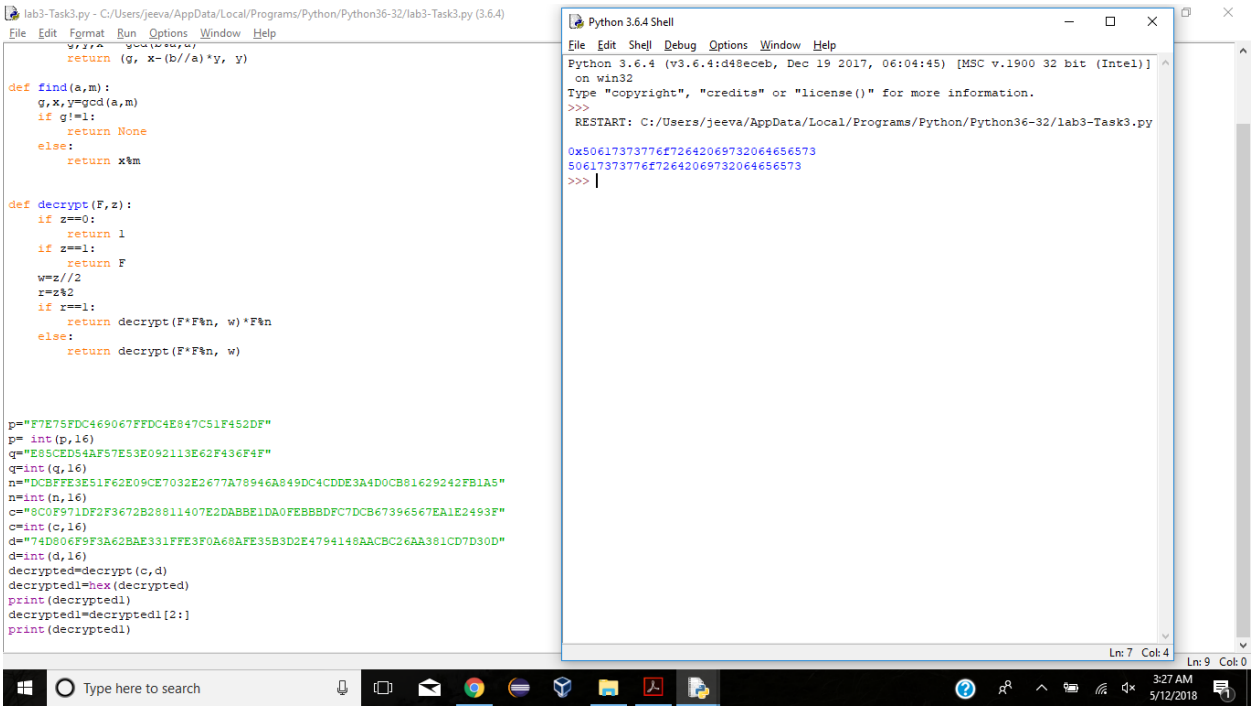
Task 2: Encrypting a Message



Ciphertext = 0x6fb078da550b2650832661e14f4f8d2cfaef475a0df3a75cacdc5de5cfc5fadc

Decryption: A top secret! got the plaintext back.

Task 3: Decrypting a Message

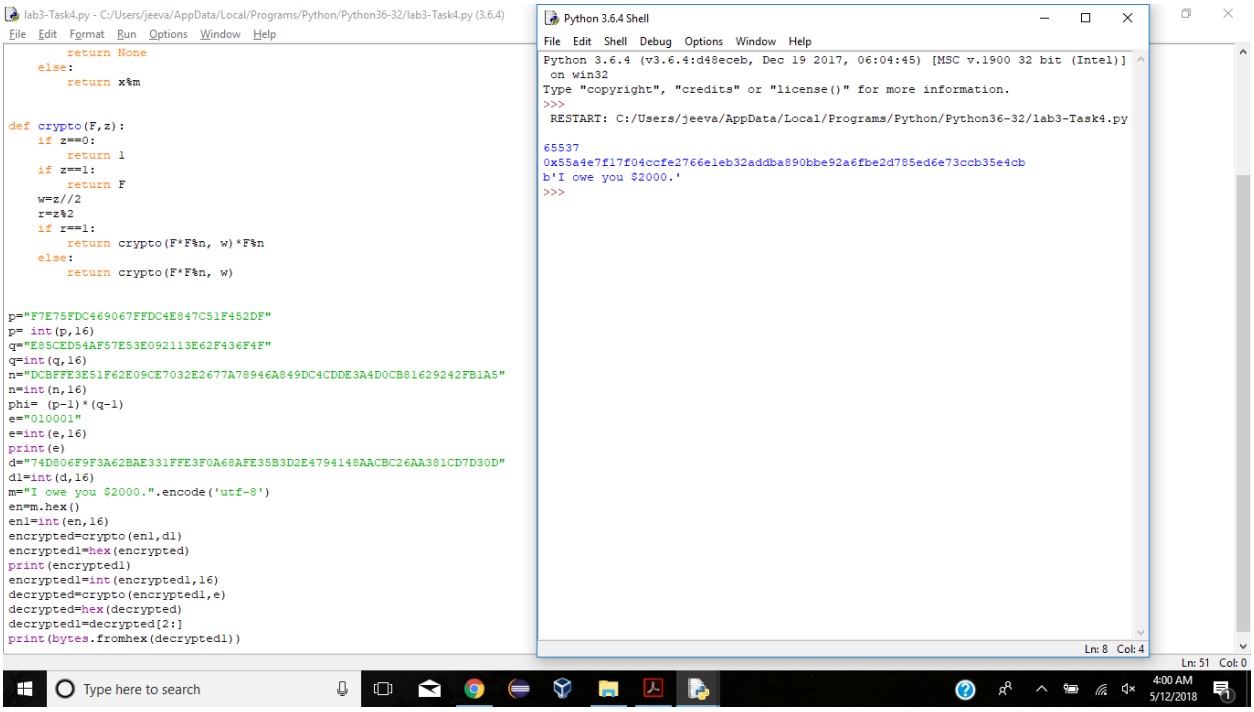


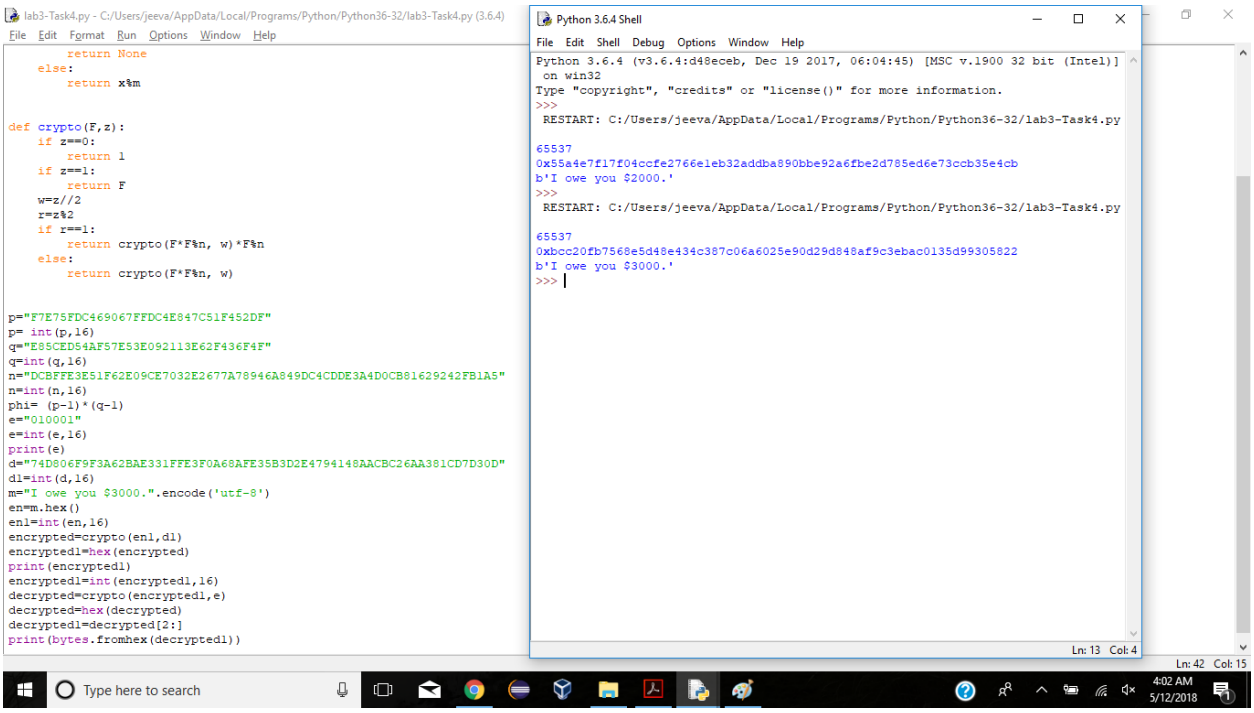
Plaintext hexadecimal string: 0x50617373776f72642069732064656573

Converting in ASCII format:

Plaintext: “Password is dees”

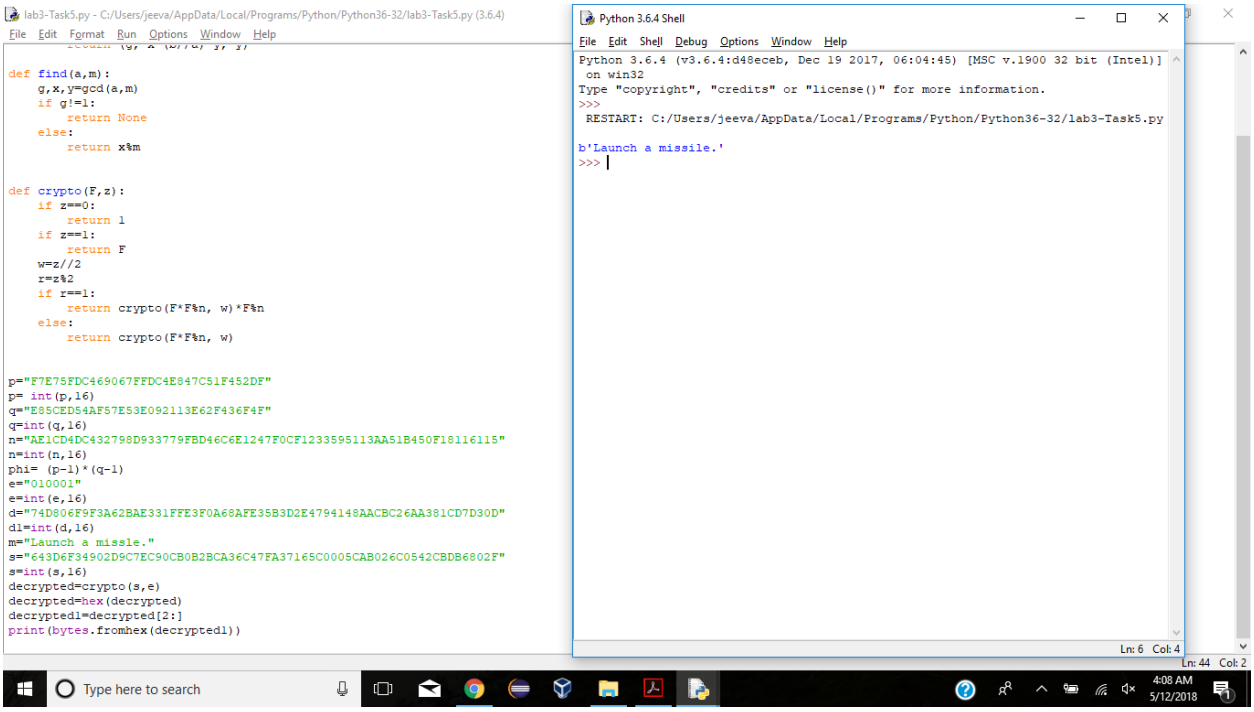
Task 4: Signing a Message

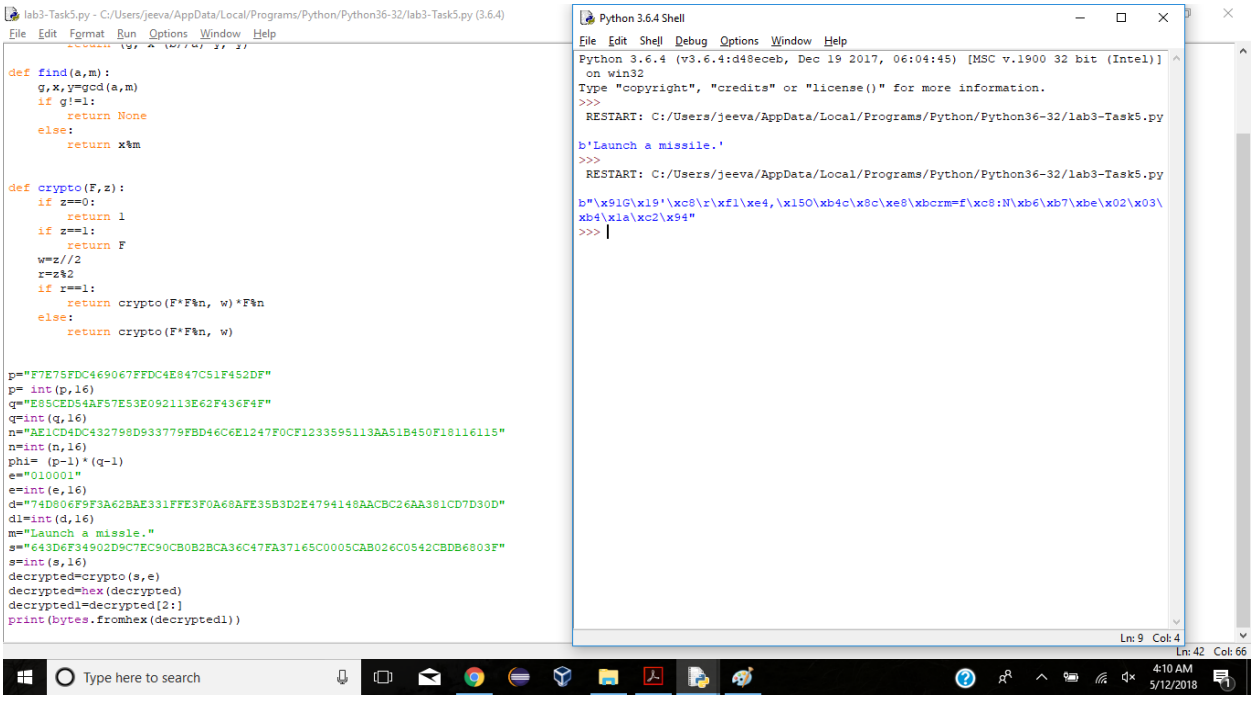




The difference in signature is 253 bits as hash function is used.

Task 5: Verifying a Signature

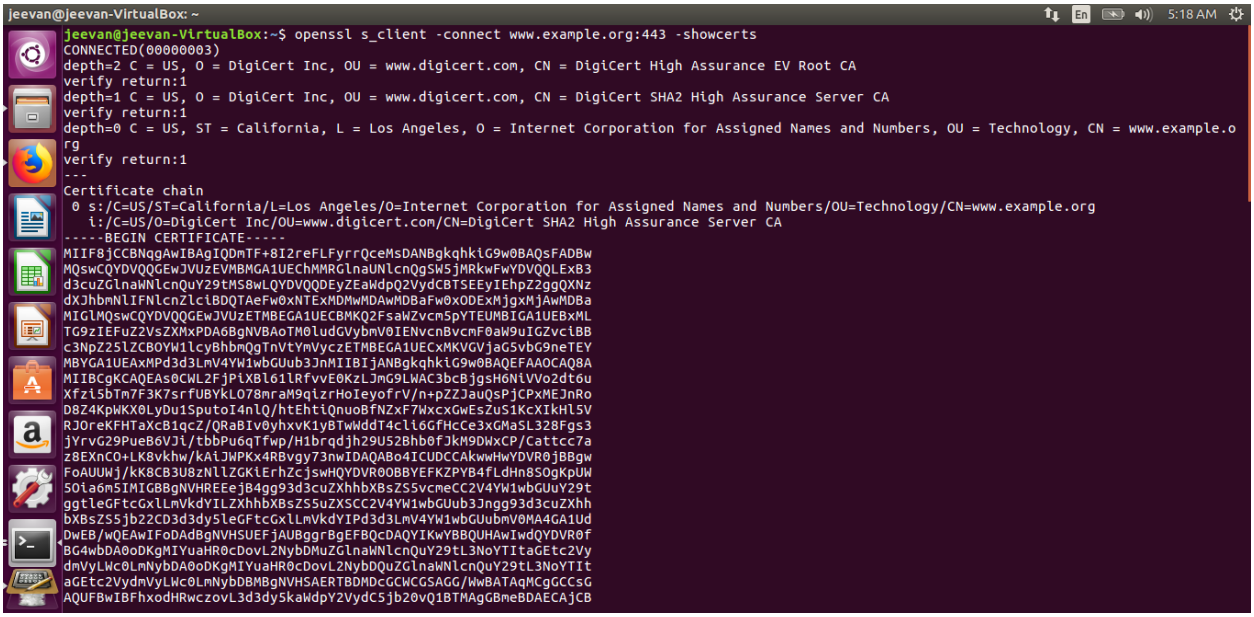


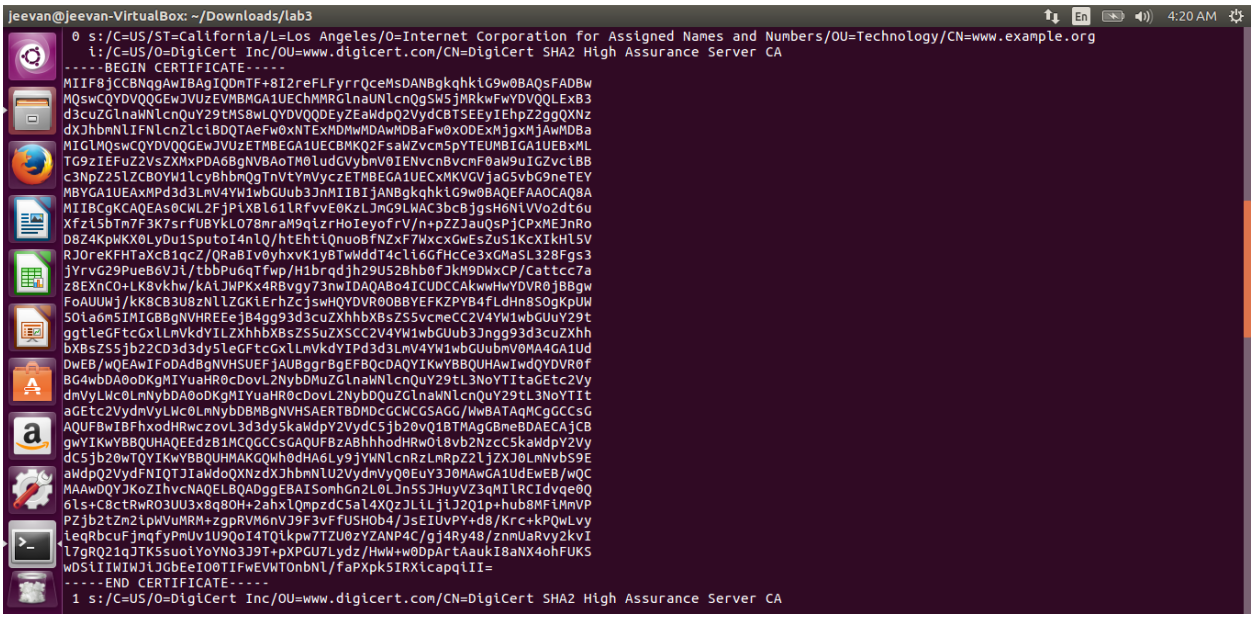


When signed with 2F it gives back plaintext. When signed with 3F it gives a different string of different length.

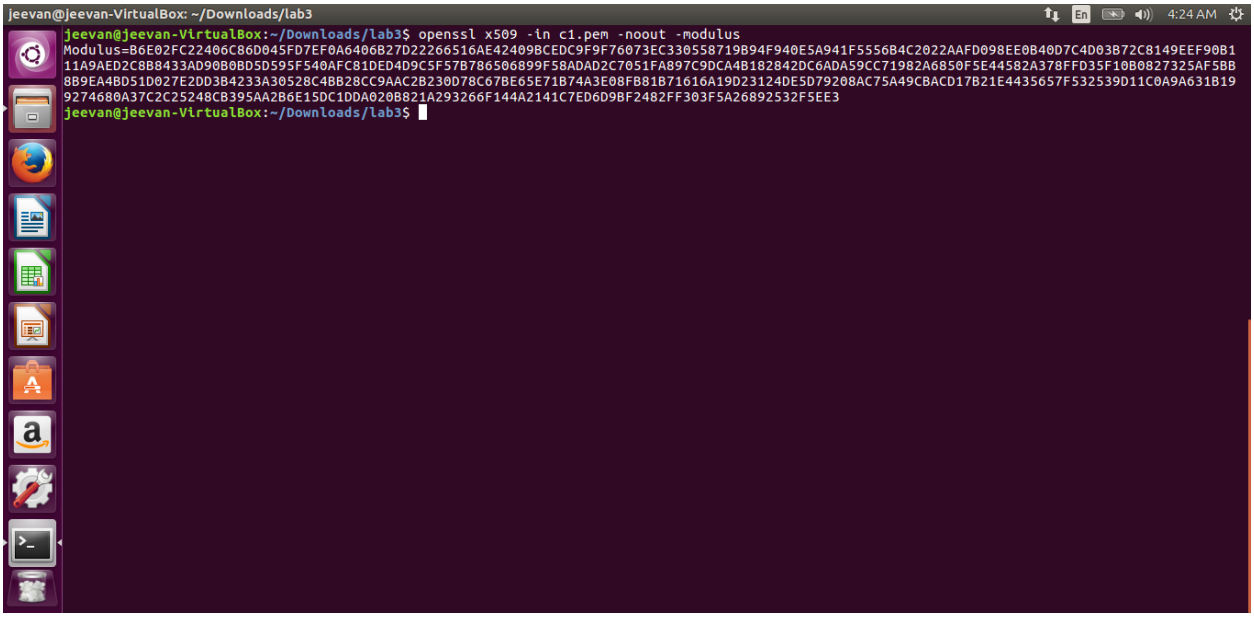
Task 6: Manually Verifying an X.509 Certificate

Step 1: Download a certificate from a real web server

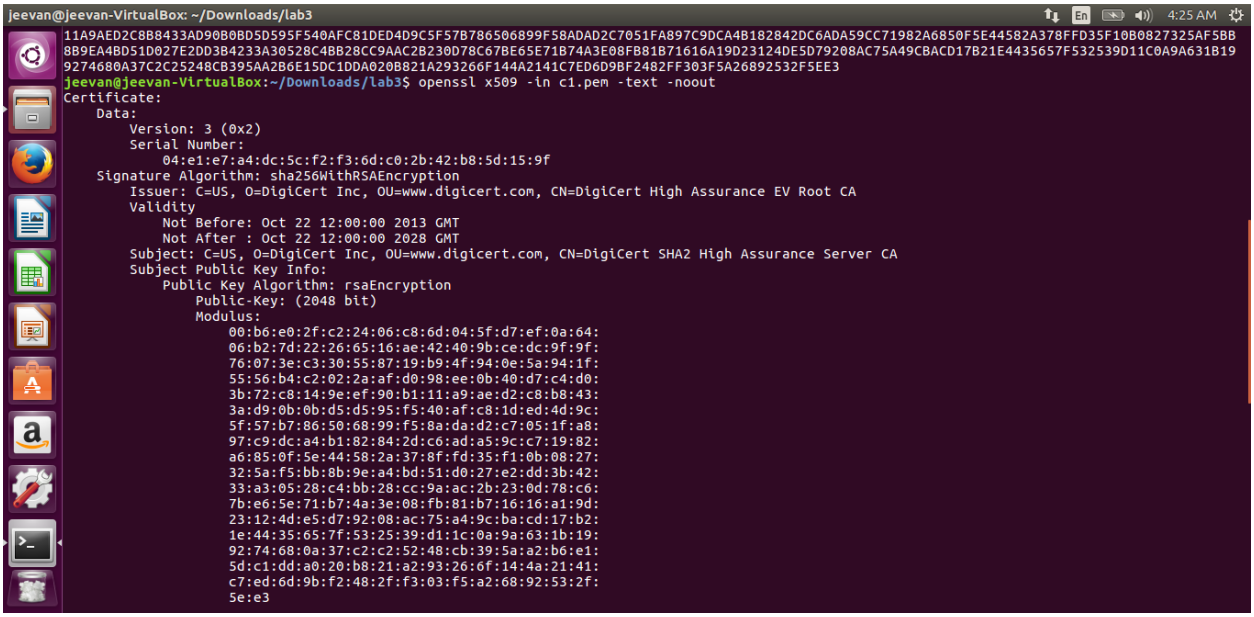


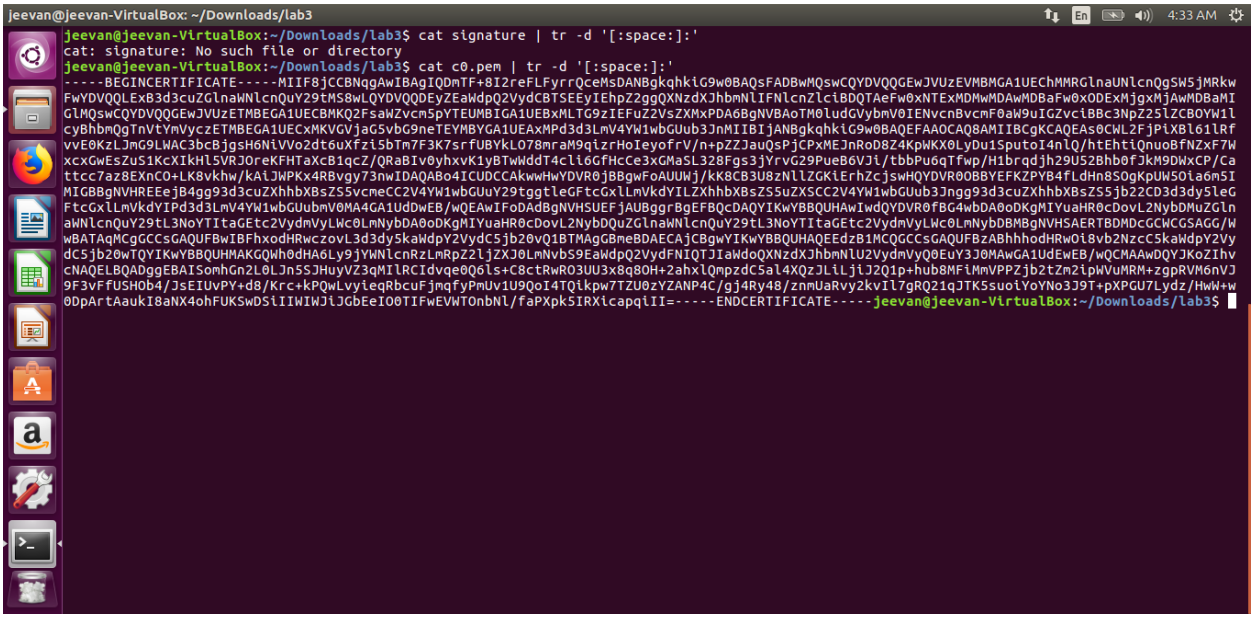


Step 2: Extract the public key (e, n) from the issuer's certificate.

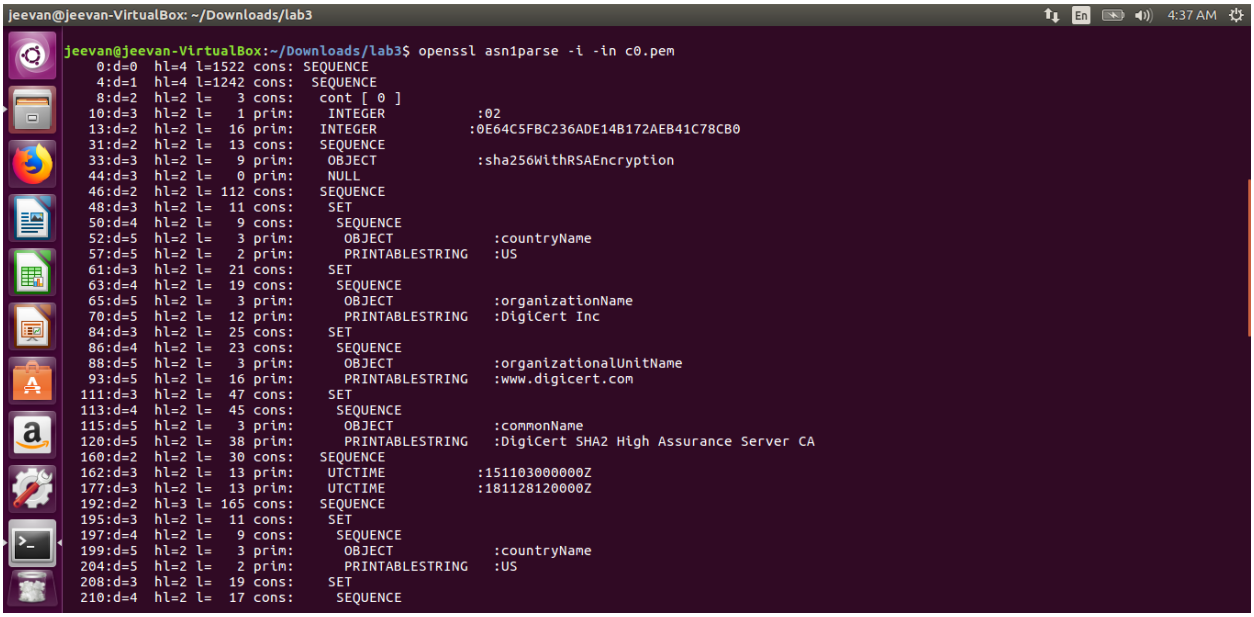


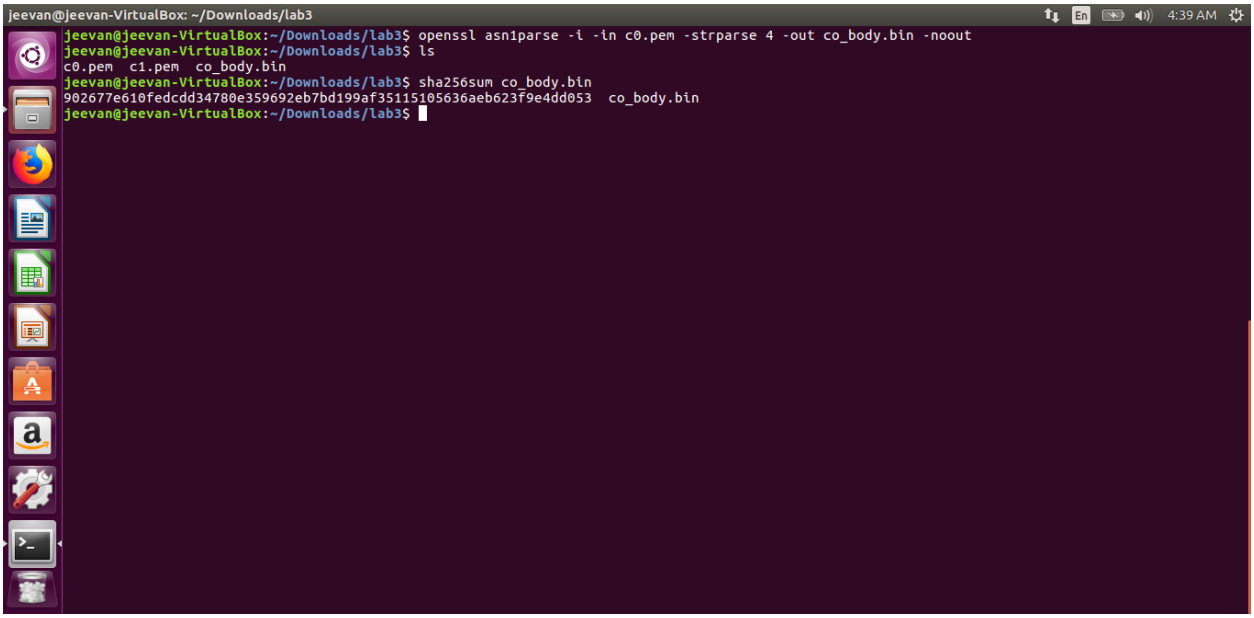
Step 3: Extract the signature from the server's certificate.

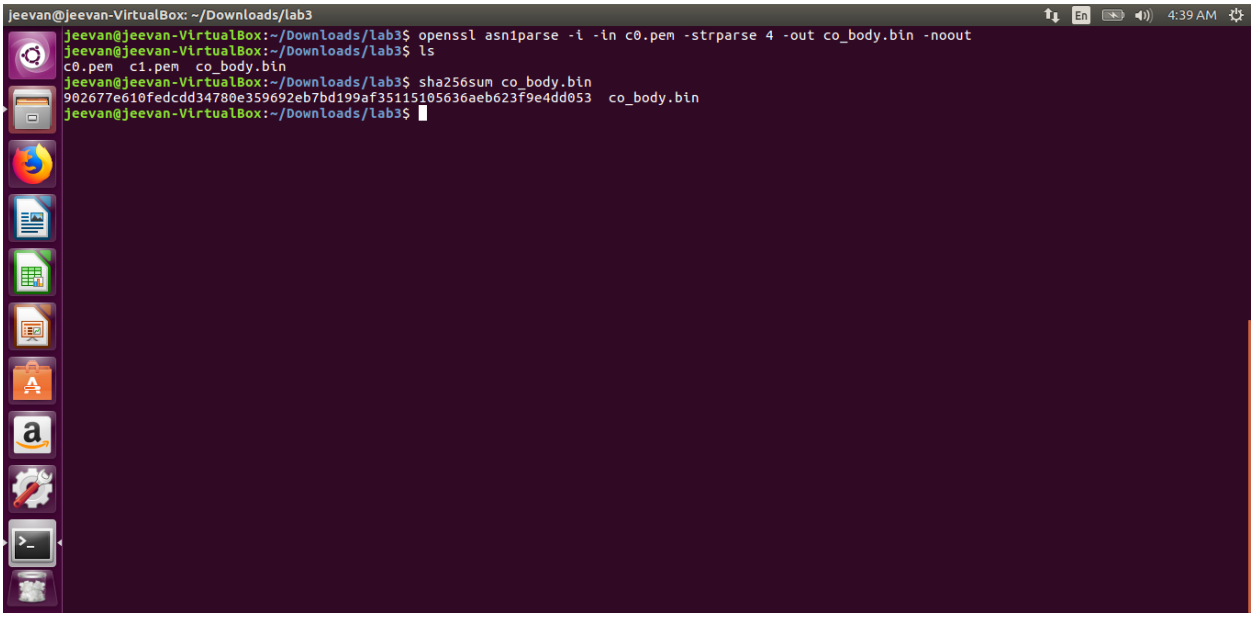




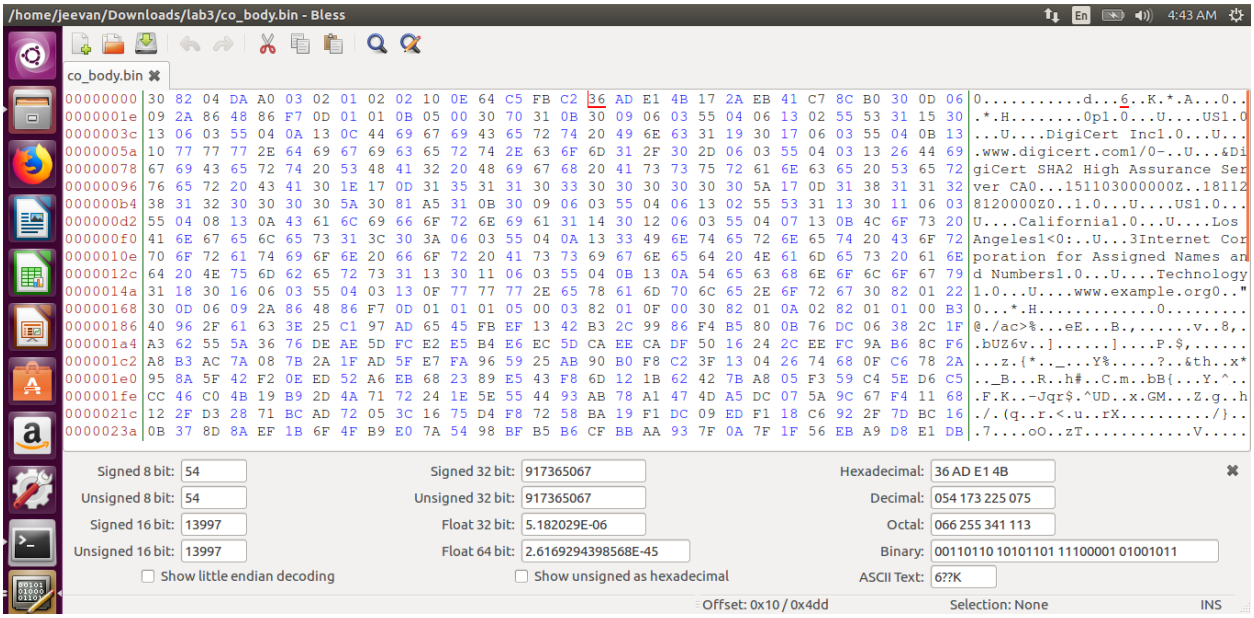
Step 4: Extract the body of the server's certificate.




jeevan@jeevan-VirtualBox: ~/Downloads/lab3
4:39 AM
jeevan@jeevan-VirtualBox:~/Downloads/lab3$ openssl asn1parse -i -in c0.pem -strparse 4 -out co_body.bin -noout
jeevan@jeevan-VirtualBox:~/Downloads/lab3$ ls
c0.pem c1.pem co_body.bin
jeevan@jeevan-VirtualBox:~/Downloads/lab3$ sha256sum co_body.bin
902677e610fedcdd34780e359692eb7bd199af35115105636aeb623f9e4dd053 co_body.bin
jeevan@jeevan-VirtualBox:~/Downloads/lab3$


jeevan@jeevan-VirtualBox: ~/Downloads/lab3
4:39 AM
jeevan@jeevan-VirtualBox:~/Downloads/lab3$ openssl asn1parse -i -in c0.pem -strparse 4 -out co_body.bin -noout
jeevan@jeevan-VirtualBox:~/Downloads/lab3$ ls
c0.pem c1.pem co_body.bin
jeevan@jeevan-VirtualBox:~/Downloads/lab3$ sha256sum co_body.bin
902677e610fedcdd34780e359692eb7bd199af35115105636aeb623f9e4dd053 co_body.bin
jeevan@jeevan-VirtualBox:~/Downloads/lab3$

Step 5: Verify the signature.



We have all the information and the certificate is verified.