Lab 5

CSE-644 INTERNET SECURITY

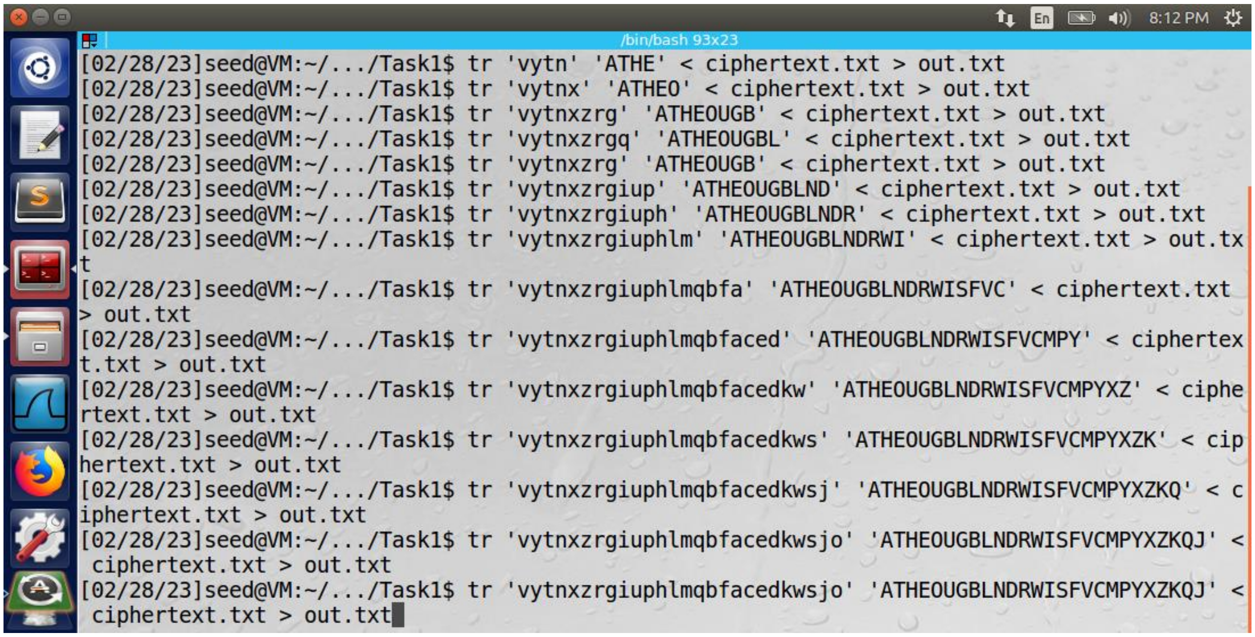
DR. SYED SHAZLI

3/5/2023

Anthony Redamonti
SYRACUSE UNIVERSITY

Task 1: Frequency Analysis

Frequency analysis was used to decrypt the encoded message that was encoded using a monoalphabetic cipher.

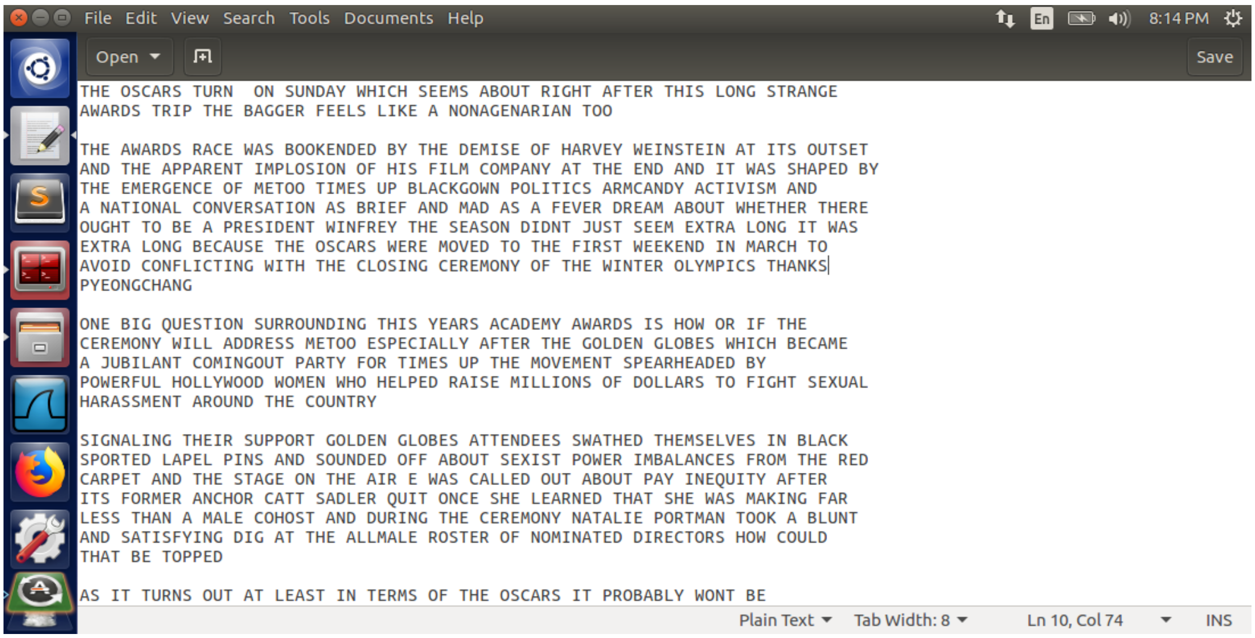


The image above shows how the letter ‘a’ and then the word ‘the’ were guessed correctly to begin the decrypting process. Then the ‘o’ followed by ‘u’, ‘g’, and ‘b’. The monoalphabetic cipher was successfully decrypted and is below.

Plaintext	Ciphertext
A	v
B	g
C	a
D	p
E	n
F	b
G	r
H	t
I	m
J	o
K	s
L	i
M	c
N	u
O	x
P	e
Q	j
R	h
S	q

T	y
U	z
V	f
W	l
X	k
Y	d
Z	w

The complete message is as follows:



THE OSCARS TURN ON SUNDAY WHICH SEEMS ABOUT RIGHT AFTER THIS LONG STRANGE
AWARDS TRIP THE BAGGER FEELS LIKE A NONAGENARIAN TOO

THE AWARDS RACE WAS BOOKENDED BY THE DEMISE OF HARVEY WEINSTEIN AT ITS OUTSET
AND THE APPARENT IMPLOSION OF HIS FILM COMPANY AT THE END AND IT WAS SHAPED BY
THE EMERGENCE OF METOO TIMES UP BLACKGOWN POLITICS ARMCANDY ACTIVISM AND
A NATIONAL CONVERSATION AS BRIEF AND MAD AS A FEVER DREAM ABOUT WHETHER THERE
OUGHT TO BE A PRESIDENT WINFREY THE SEASON DIDNT JUST SEEM EXTRA LONG IT WAS
EXTRA LONG BECAUSE THE OSCARS WERE MOVED TO THE FIRST WEEKEND IN MARCH TO
AVOID CONFLICTING WITH THE CLOSING CEREMONY OF THE WINTER OLYMPICS THANKS
PYEONGCHANG

ONE BIG QUESTION SURROUNDING THIS YEARS ACADEMY AWARDS IS HOW OR IF THE
CEREMONY WILL ADDRESS METOO ESPECIALLY AFTER THE GOLDEN GLOBES WHICH BECAME
A JUBILANT COMINGOUT PARTY FOR TIMES UP THE MOVEMENT SPEARHEADED BY
POWERFUL HOLLYWOOD WOMEN WHO HELPED RAISE MILLIONS OF DOLLARS TO FIGHT SEXUAL
HARASSMENT AROUND THE COUNTRY

SIGNALING THEIR SUPPORT GOLDEN GLOBES ATTENDEES SWATHED THEMSELVES IN BLACK
SPORTED LAPEL PINS AND SOUNDED OFF ABOUT SEXIST POWER IMBALANCES FROM THE RED
CARPET AND THE STAGE ON THE AIR E WAS CALLED OUT ABOUT PAY INEQUITY AFTER
ITS FORMER ANCHOR CATT SADLER QUIT ONCE SHE LEARNED THAT SHE WAS MAKING FAR
LESS THAN A MALE COHOST AND DURING THE CEREMONY NATALIE PORTMAN TOOK A BLUNT
AND SATISFYING DIG AT THE ALLMALE ROSTER OF NOMINATED DIRECTORS HOW COULD
THAT BE TOPPED

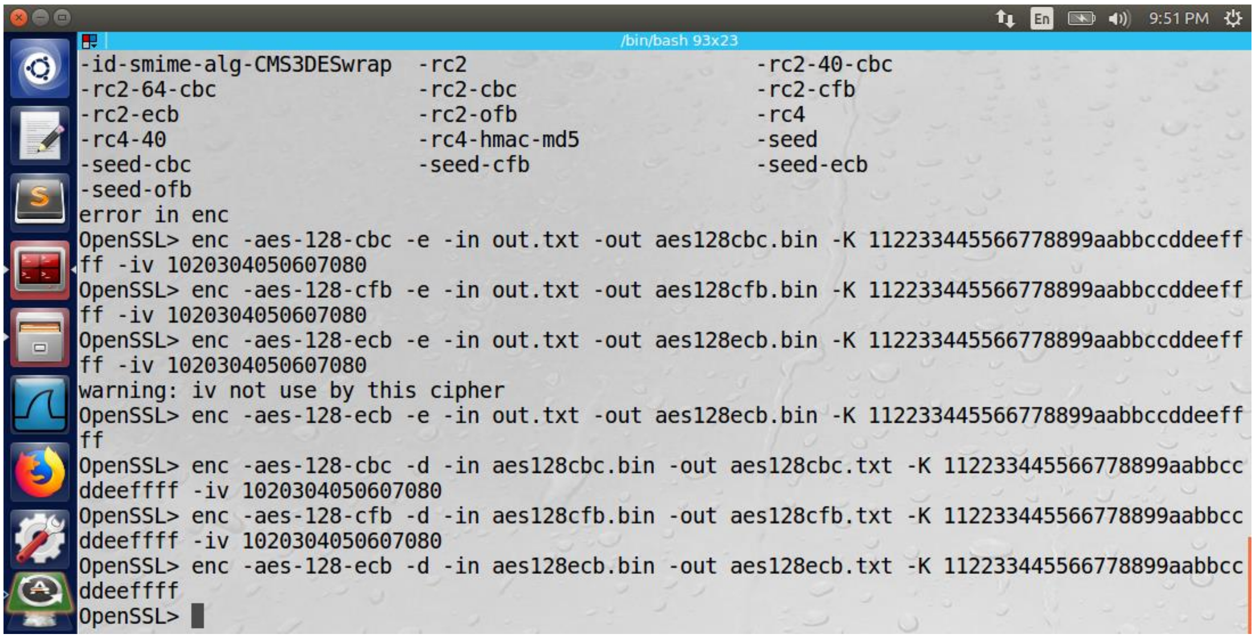
AS IT TURNS OUT AT LEAST IN TERMS OF THE OSCARS IT PROBABLY WONT BE

<u>Observation</u>: The encoded message was able to be decoded using frequency analysis. Vowels are more frequent than consonants. In particular, only the letter ‘a’ can form a word by itself, so it was guessed first. Then the word ‘the’ was guessed.

<u>Explanation</u>: The monoalphabetic ciphers are subject to frequency analysis, which is why they are considered unsecure. Polyalphabetic ciphers are considered more secure.

Task 2: Encryption Using Different Ciphers and Modes

The plaintext from task 1 discussing the Oscars Awards Ceremony was encoded using three different cipher algorithms: aes-128-cbc, aes-128-cfb, and aes-128-ecb. The commands used to encode and decode the plaintext are shown below.

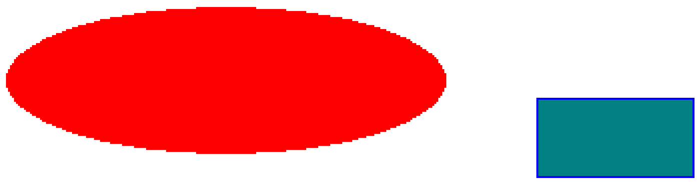


Observation: When encoding the message, using a 128-bit key, the -K argument is used to define the key using hexadecimal representation. The -iv argument is used to define the initialization vector also represented in hexadecimal notation.

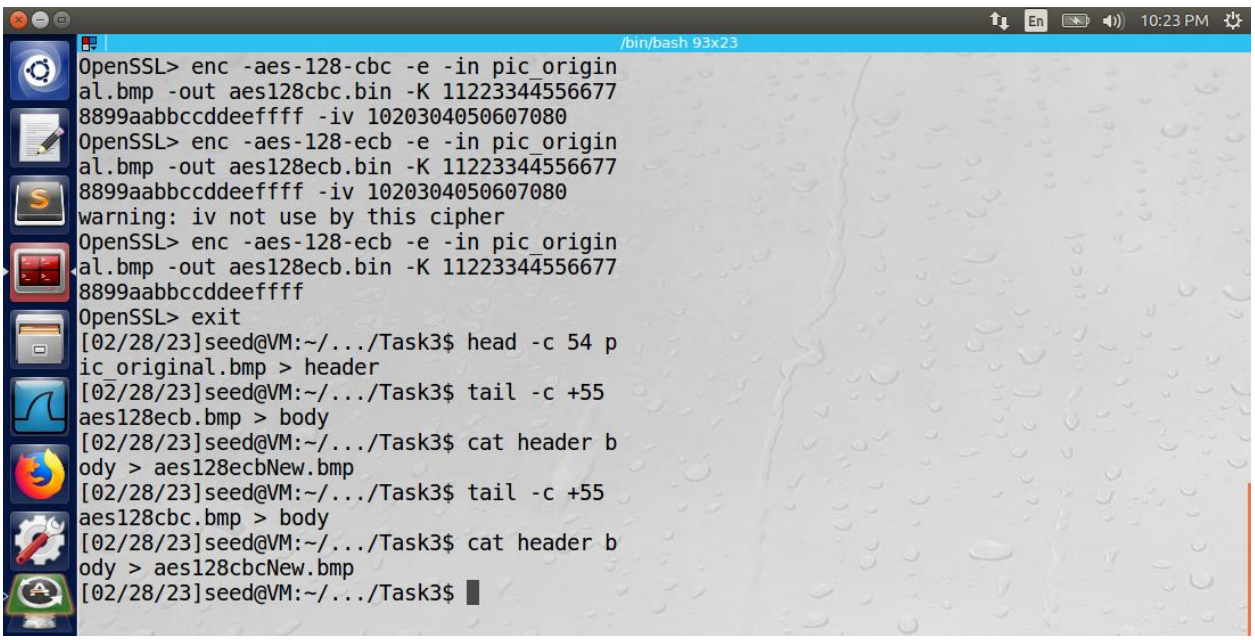
Explanation: The three cipher algorithms are all standard functions in the OpenSSL library. To retrieve a full list of algorithms offered by this library, use the “list-cipher-commands” command. Use the -e or -d argument after the enc command to specify encryption or decryption respectively.

Task 3: Encryption Mode: ECB vs. CBC

The following image (pic_original.bmp) was encoded using two encryption algorithms: Electronic Code Book (ECB) and Cipher Block Chaining (CBC).

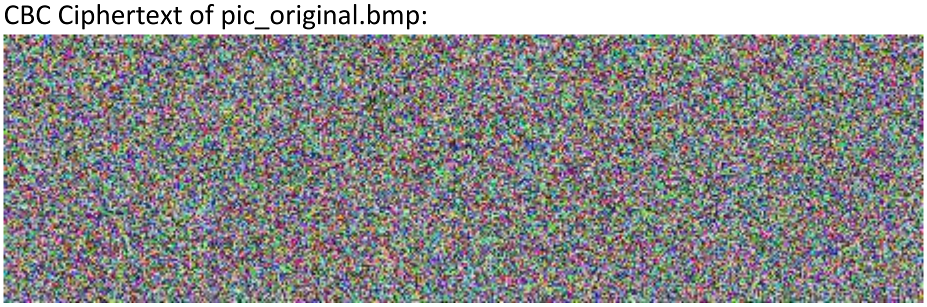


The following commands were used to encode the image and format the ciphertext into proper BMP file format:

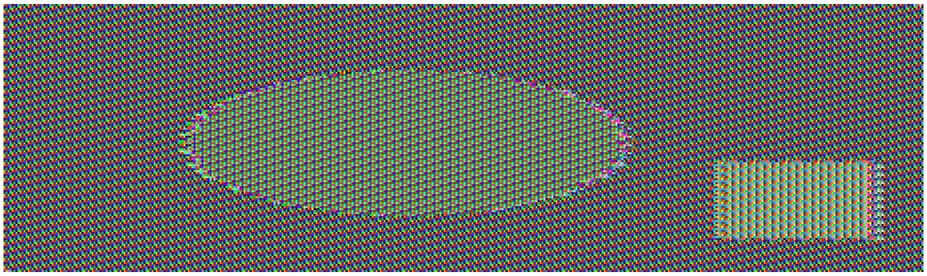


Viewing the ciphertext files as a picture revealed the following:

CBC Ciphertext of pic_original.bmp:

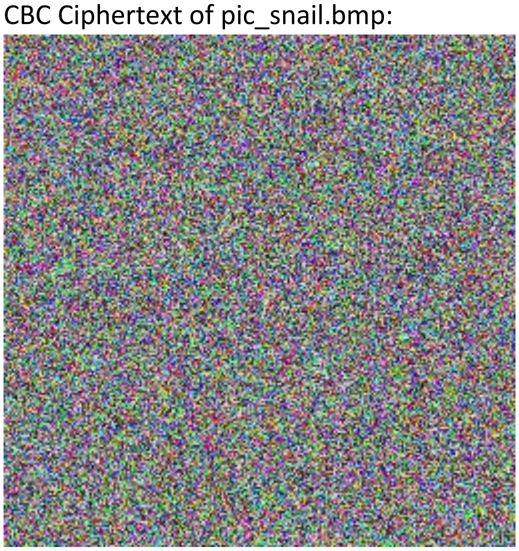


ECB Ciphertext of pic_original.bmp:

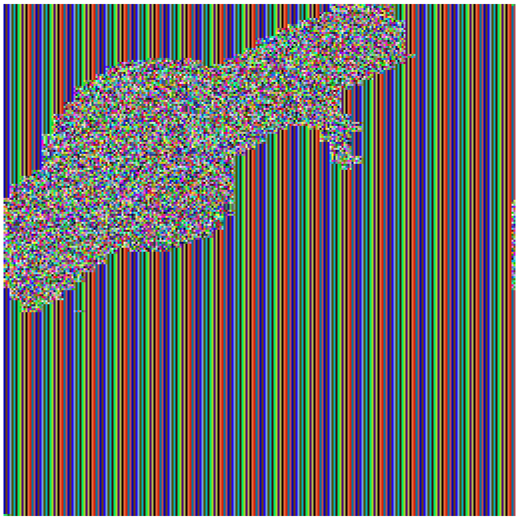


The same process was repeated for another image (pic_snail.bmp) shown below.

CBC Ciphertext of pic_snail.bmp:

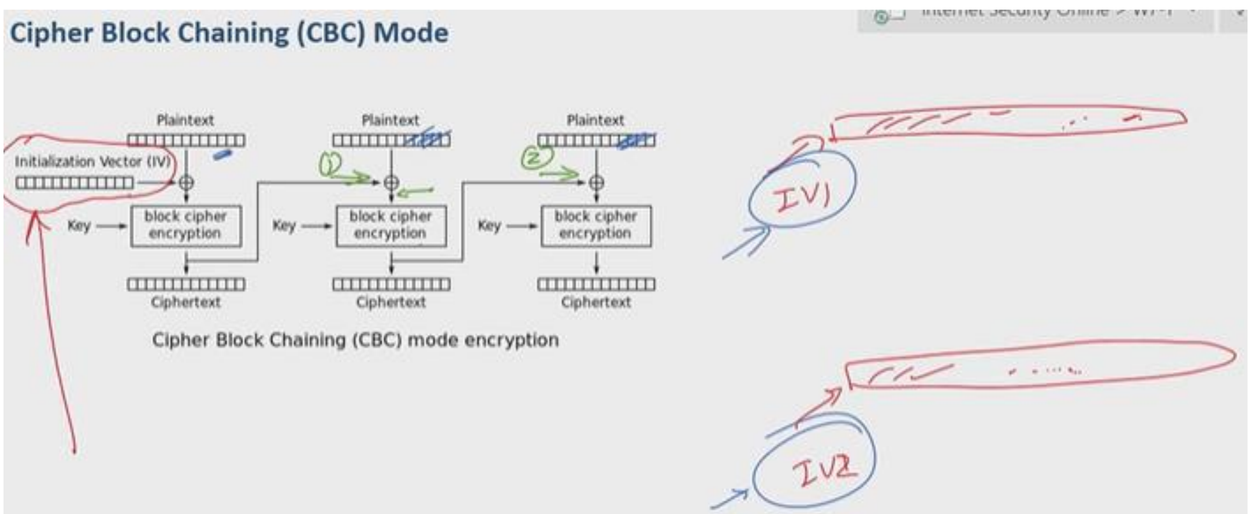


ECB Ciphertext of pic_snail.bmp:



Observation: When encoding the BMP file using the ECB encryption algorithm, an outline of the original image is visible. The same outline is not visible when encoding using the CBC encryption algorithm.

Explanation: The ECB mode of encryption does not use block chaining and is known to be naïve and unsafe. I.e., each block of plaintext is encrypted separately, so if two plaintext blocks are identical, their corresponding ciphertext blocks will also be identical.



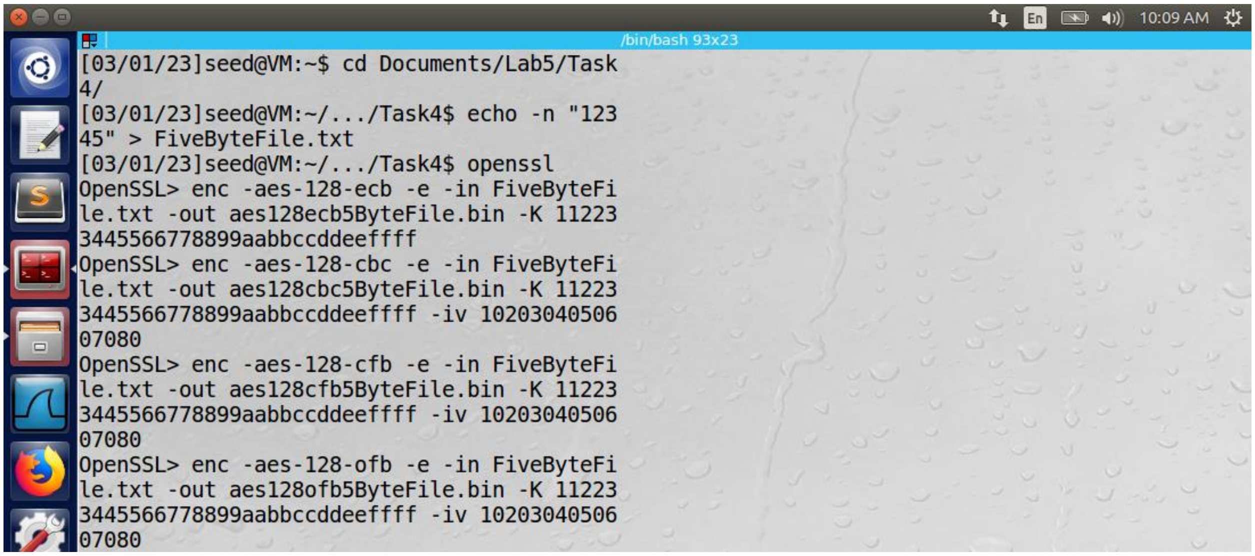
CBC mode shown above uses an initialization vector that is XOR-ed with the plaintext. The output of the XOR operation is fed into a block cipher encryption algorithm, and the output is the ciphertext. The ciphertext is then used in the XOR operator in the next block of encryption.

By using block chaining, the image becomes completely hidden because no clear pattern is kept from the original image. ECB does not use block chaining. Therefore, every block is transformed using the ECB encryption algorithm, but the outline of the image is still visible.

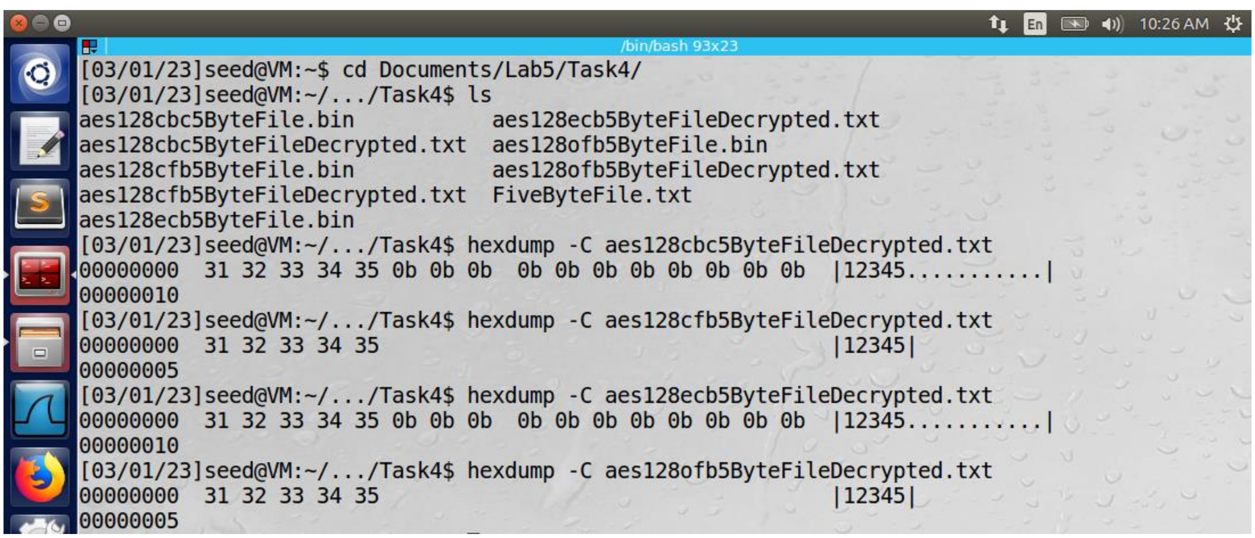
Task 4: Padding

Part 1: Using ECB, CBC, CFB, and OFB to Encrypt a File

The following commands were used to create and encrypt a file of five bytes in size using the ECB, CBC, CFB, and OFB encryption modes.



The following commands were used to view the contents of the ciphertext including the padding.



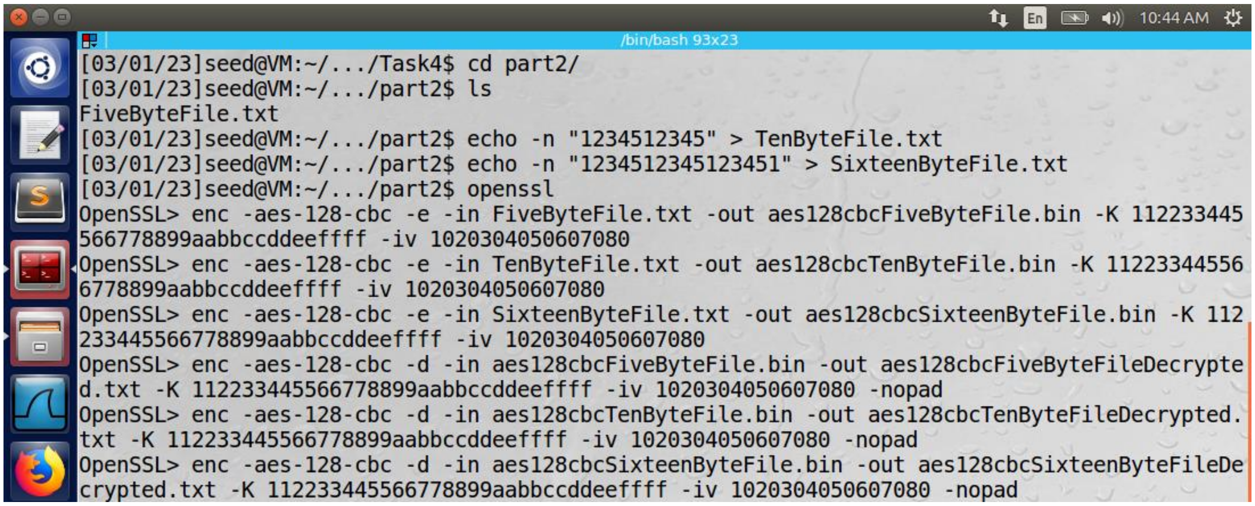
Observation: Notice how padding is used in CBC and ECB modes but not in CFB or OFB modes.

Explanation: CFB and OFB modes do not require padding for the last block because they are stream ciphers. It is not required to wait until enough data are available to fill a cipher block. Each plaintext is encrypted on a bit-by-bit basis because the plaintext is XORed with the output of the previous block (XOR is bit-wise operation). It makes CFB and OFB optimal for encryption of real-time applications.

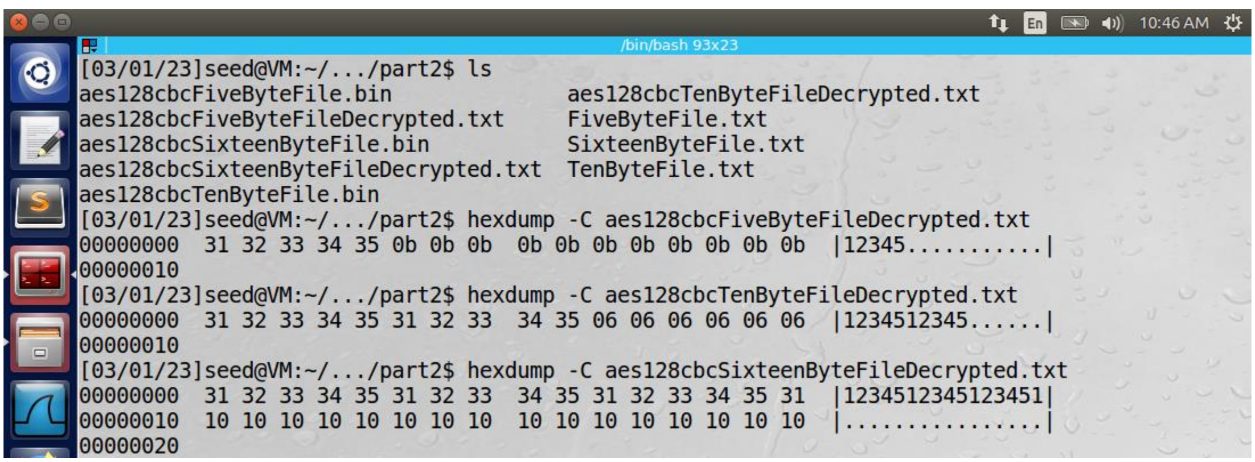
CBC and ECB require padding so that the length of the ciphertext is a multiple of 16 bytes, which is the block size of AES.

Part 2: Encrypting Differently Sized Files using CBC Mode

Three files of different size were created and encrypted using the commands shown below. The files were 5 bytes, 10 bytes, and 16 bytes in length and were encrypted using the CBC encryption mode. Note the -nopad option to preserve the visibility of the padding.



The ciphertext was viewed using the commands shown below.

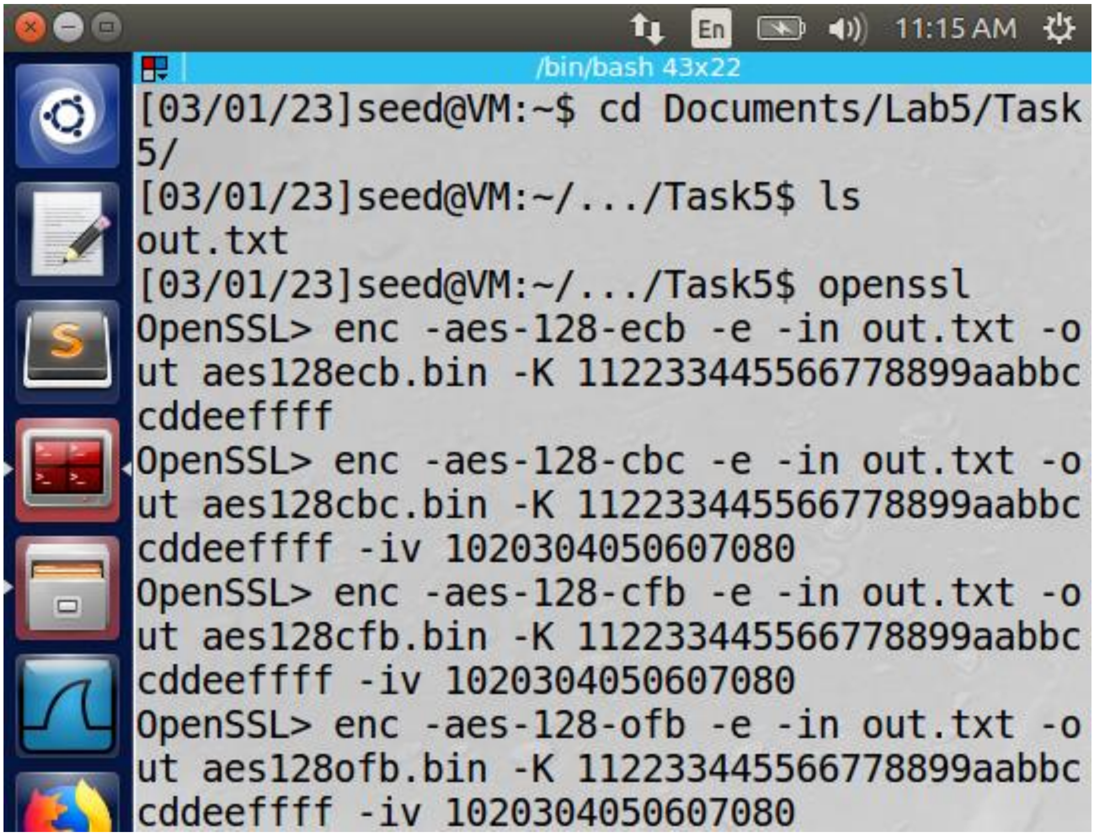


Observation: The number used to pad the 5-byte file was 0xb, or 11 in decimal. The number used to pad the 10-byte file was 6. The number used to pad the 16-byte file was 0x10, or 16 in decimal.

Explanation: CBC and ECB require padding so that the length of the ciphertext is a multiple of 16 bytes, which is the block size of AES. The number used to pad each block represents the number of padded bytes needed to fill the block. I.e., there were 11 padded bytes added to the 5-byte file, so the number 11 (0xb) was used to pad the block. In the case of the 16-byte file, an entire row of padding was added using the number 16.

Task 5: Error Propagation: Corrupted Ciphertext

The plaintext from task 1 discussing the Oscars Awards Ceremony was used as the plaintext for this task. The commands below were used to encrypt the file in ECB, CBC, CFB, and OFB modes.



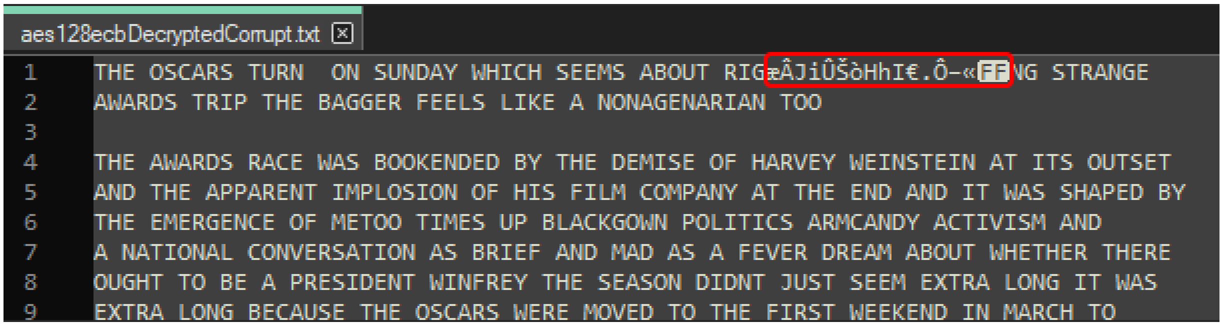
Then, the bless hex editor was used to corrupt one bit in the 55th byte in each binary file of ciphertext. The ciphertext was then decrypted using the correct key and initialization vector.

I predict that upon decryption, both the ECB and CBC modes will corrupt the entire block in which the 55th byte is located. The reason is that the ciphertext is fed back through the block cipher decryption algorithm before being XOR-ed with the output of the previous block.

I predict that upon decryption, CFB mode will corrupt the 55th byte and the entire NEXT block of data (not the block containing the 55th byte). The reason is that during decryption, CFB mode XOR's the ciphertext with the output of the block cipher decryption algorithm and then uses that same ciphertext as input into the next block's cipher decryption algorithm.

I predict that upon decryption, OFB mode will corrupt only the 55th byte. The reason is that upon decryption, OFB mode XOR's the ciphertext with the output of the block cipher decryption algorithm, but before it is XOR'ed with the ciphertext, it is forwarded as input into the next block's cipher decryption algorithm.

Below is the decrypted corrupt file using the ECB mode.

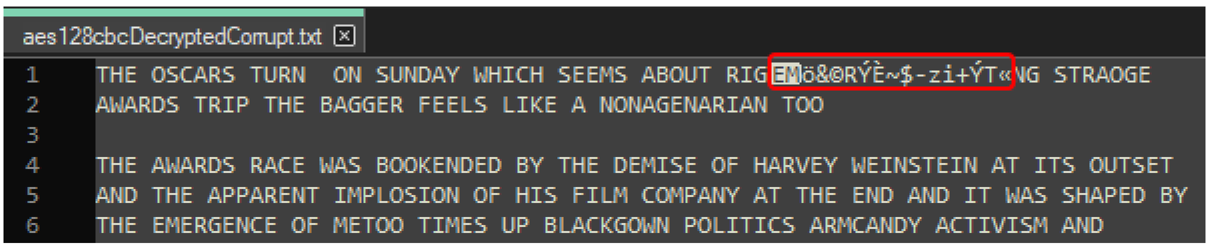

aes128ecbDecryptedCorrupt.txt

```
THE OSCARS TURN  ON SUNDAY WHICH SEEMS ABOUT RIGæÂJiÛŠòHhI€.Ô–«FFNG STRANGE
AWARDS TRIP THE BAGGER FEELS LIKE A NONAGENARIAN TOO

THE AWARDS RACE WAS BOOKENDED BY THE DEMISE OF HARVEY WEINSTEIN AT ITS OUTSET
AND THE APPARENT IMPLOSION OF HIS FILM COMPANY AT THE END AND IT WAS SHAPED BY
THE EMERGENCE OF METOO TIMES UP BLACKGOWN POLITICS ARMCANDY ACTIVISM AND
A NATIONAL CONVERSATION AS BRIEF AND MAD AS A FEVER DREAM ABOUT WHETHER THERE
OUGHT TO BE A PRESIDENT WINFREY THE SEASON DIDNT JUST SEEM EXTRA LONG IT WAS
EXTRA LONG BECAUSE THE OSCARS WERE MOVED TO THE FIRST WEEKEND IN MARCH TO
```

Notice the corrupted block of data outlined in red.

Below is the decrypted corrupt file using the CBC mode.

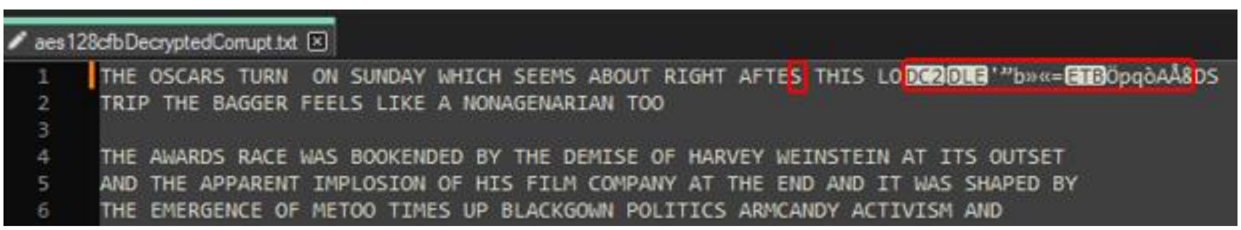

aes128cbcDecryptedCorrupt.txt

```
THE OSCARS TURN  ON SUNDAY WHICH SEEMS ABOUT RIGEMö&©RÝÈ~$-zi+ÝT«NG STRAOGE
AWARDS TRIP THE BAGGER FEELS LIKE A NONAGENARIAN TOO

THE AWARDS RACE WAS BOOKENDED BY THE DEMISE OF HARVEY WEINSTEIN AT ITS OUTSET
AND THE APPARENT IMPLOSION OF HIS FILM COMPANY AT THE END AND IT WAS SHAPED BY
THE EMERGENCE OF METOO TIMES UP BLACKGOWN POLITICS ARMCANDY ACTIVISM AND
```

Notice the corrupted block of data outlined in red.

Below is the decrypted corrupt file using the CFB mode.

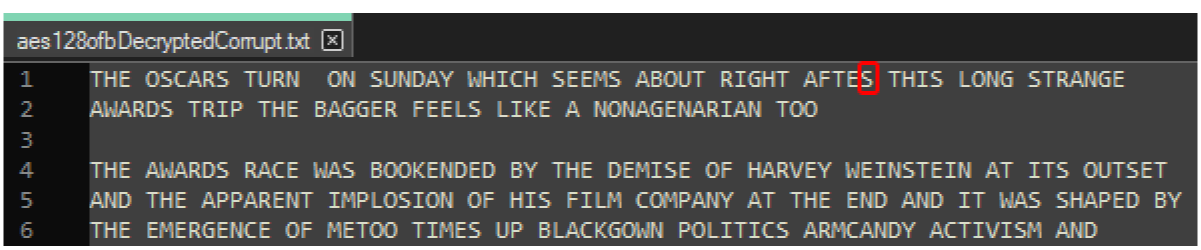

aes128cfbDecryptedCorrupt.txt

```
THE OSCARS TURN  ON SUNDAY WHICH SEEMS ABOUT RIGHT AFTES THIS LODC2DLE'"b»«=ETBÖpqòAÅ8DS
TRIP THE BAGGER FEELS LIKE A NONAGENARIAN TOO

THE AWARDS RACE WAS BOOKENDED BY THE DEMISE OF HARVEY WEINSTEIN AT ITS OUTSET
AND THE APPARENT IMPLOSION OF HIS FILM COMPANY AT THE END AND IT WAS SHAPED BY
THE EMERGENCE OF METOO TIMES UP BLACKGOWN POLITICS ARMCANDY ACTIVISM AND
```

Notice the one corrupted letter 'S' instead of 'R' representing the 55th byte. Also, notice the entire next block has been corrupted.

Below is the decrypted corrupt file using the OFB mode.


aes128ofbDecryptedCorrupt.txt

```
THE OSCARS TURN  ON SUNDAY WHICH SEEMS ABOUT RIGHT AFTES THIS LONG STRANGE
AWARDS TRIP THE BAGGER FEELS LIKE A NONAGENARIAN TOO

THE AWARDS RACE WAS BOOKENDED BY THE DEMISE OF HARVEY WEINSTEIN AT ITS OUTSET
AND THE APPARENT IMPLOSION OF HIS FILM COMPANY AT THE END AND IT WAS SHAPED BY
THE EMERGENCE OF METOO TIMES UP BLACKGOWN POLITICS ARMCANDY ACTIVISM AND
```

Notice how only the 55th byte has been corrupted as the 'S' should be an 'R'.

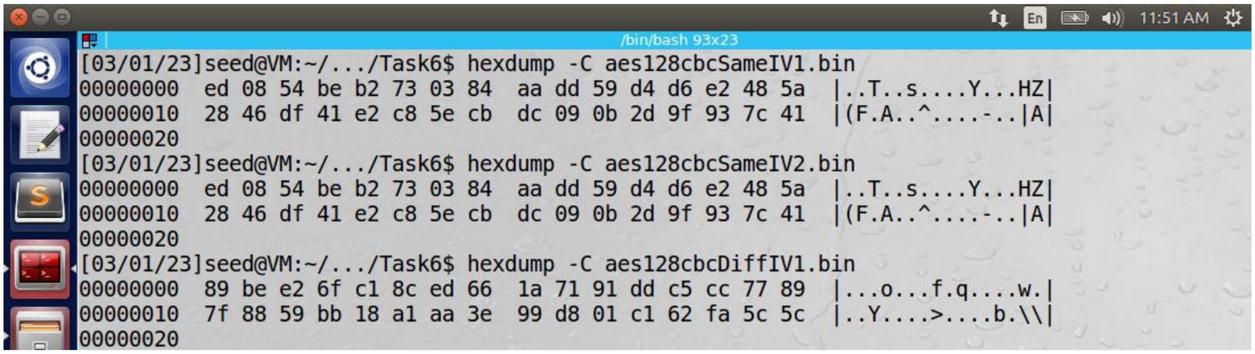
Observation: The decrypted files contained the predicted results. The reasoning behind each prediction has been previously explained.

<u>Explanation</u>: Each mode encrypts and decrypts data using a unique method. Some modes handle data corruption similarly, while others behave differently.

Task 6: Initialization Vector (IV) and Common Mistakes

Part 1: Initialization Vector Uniqueness

If the same Key/IV pair is used to encrypt the same data, the encrypted files will also be the same.



Note above that the two files 'aes128cbcSameIV1.bin' and 'aes128cbcSameIV2.bin' used the same encryption algorithm (CBC) and the same key and initialization vector. The outputs of the encryption algorithm match, which leads to a security risk. When the initialization vector was altered, the output was different (see 'aes128cbcDiffIV1.bin' shown above).

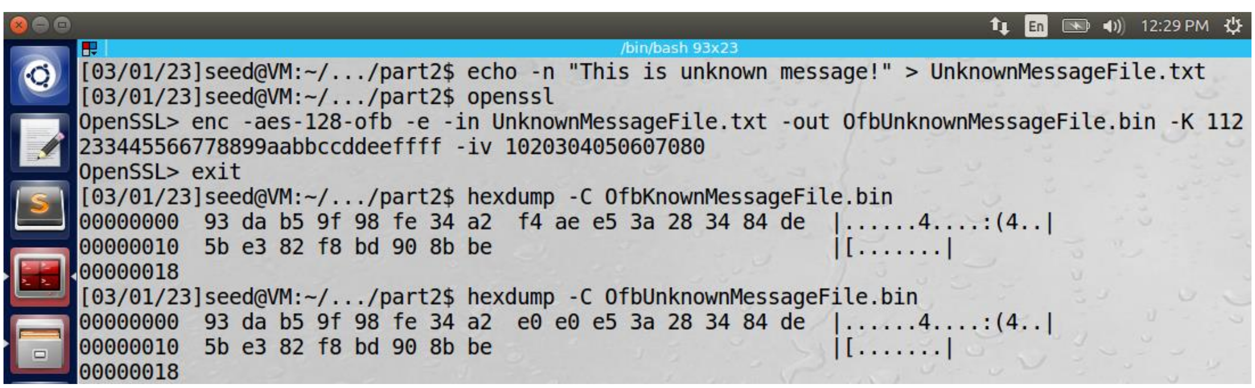
Part 2: Use the Same IV

If the initialization vector is the same, but the plaintext is different, data security issues are still present.

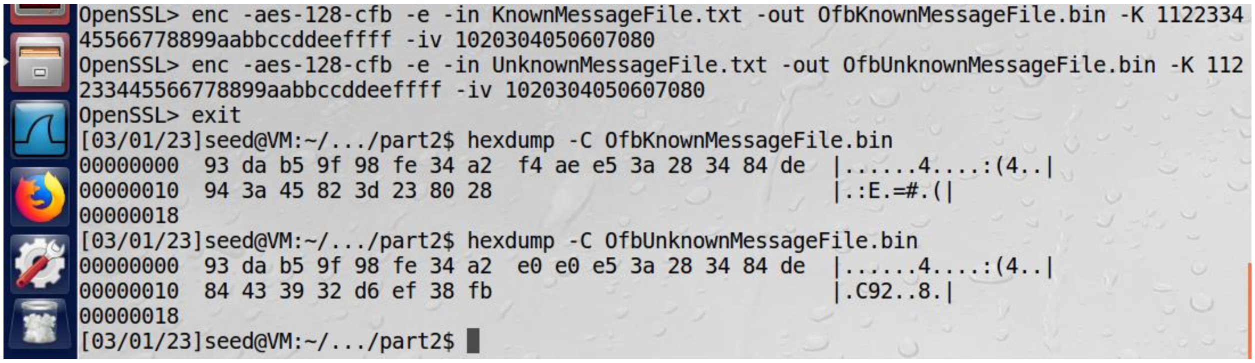
```
Plaintext  (P1): This is a known message!
Ciphertext (C1): a469b1c502c1cab966965e50425438e1bb1b5f9037a4c159

Plaintext  (P2): (unknown to you)
Ciphertext (C2): bf73bcd3509299d566c35b5d450337e1bb175f903fafc159
```

If P2 were "This is unknown message!" then the output of OFB using the same key/IV pair would be very similar to the output of "This is a known message!". See test below.



Notice how the two ciphertexts are almost identical. If the same test is performed for CFB, only the first two blocks of ciphertext are similar.



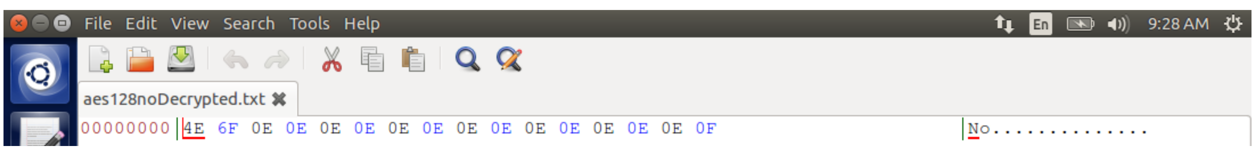
Everything after the second block is unique in CFB mode. The reason is that CFB mode encrypts data by XORing the output of the block cipher encryption algorithm with the plaintext. The output of the XOR operation is fed into the block cipher encryption algorithm of the next block. The next block's encryption algorithm will change the ciphertext to something completely different between messages P1 and P2.

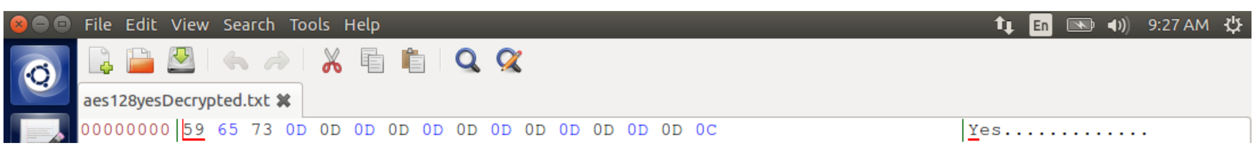
Part 3: Use a Predictable IV

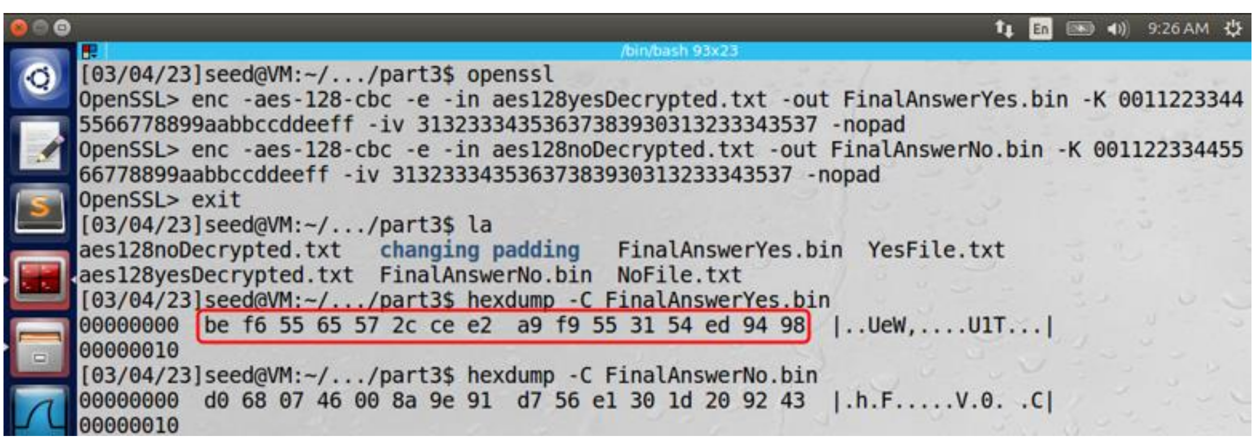
The following guesses were formed by creating a file with the text "Yes" and another with the text "No". Then padding was added to both files to form a 16-byte guess. The last bit was inverted because the first initialization vector was almost the same as the next initialization vector except for the last bit.

$$Ciphertext = Encrypt(IV_1\ XOR\ P_1) = Encrypt\big(IV_2\ XOR\ (IV_2\ XOR\ IV_1\ XOR\ P_1)\big)$$

IV_1 XOR IV_2 = 1. Therefore, the last bit of P_1 will be inverted (flipped), then sent through the aes-128-cbc encryption algorithm.







The encrypted message (C_2) outlined in red above from the Yes file matches the known Ciphertext (C_1). Therefore, P_1 was "Yes". Note: when encoding P_2 (testing the guess), it was important to use the -nopad option in openssl, since the padding was provided (and altered) in the plaintext.

Observation: The 3 exercises of this task demonstrate that initialization vectors must be unique and unpredictable.

Explanation: Using the same IV/Key pair can lead to similar plaintext producing similar ciphertext. Predictable IV's leave users vulnerable to chosen-plaintext attacks.

Task 7: Programming using the Crypto Library

The following C program was written to find the hidden key used to encrypt the message: “This is a top secret.” The key was taken from a word list and the hash sign (#) was used as padding to form a 16-byte key. Each word in the word list was provided in a text file “WordsList.txt”. The initialization vector and ciphertext were both provided.

The following output was displayed to the console when running the program. The wordSelector variable in the program was used to print out a particular word in the file and all of its important details, such as its key representation (including the # padding), the formatted key representation in hexadecimal, and the formatted cipher output for that word in hexadecimal.

```
[03/05/23]seed@VM:~/.../Task7$ gcc -o findKey LatestKeyNew.c -lcrypto
[03/05/23]seed@VM:~/.../Task7$ ./findKey
key: accession#######
formatted key: 616363657373696f6e23232323232323
formatted cipher output: b4c3897c6fc256d3d0dd7e187f41694b7b63dbb3db4c7b380063db777b000031
max num of bytes matching in a row: 3
ciphertext solution: 764aa26b55a4da654df6b19e4bce00f4ed05e09346fb0e762583cb7da2ac93a2
[03/05/23]seed@VM:~/.../Task7$
```

Unfortunately, no key was found that produced ciphertext that matched the given ciphertext. The highest number of matching bytes in a row that was found by testing the entire words list was 3. If the number of matching bytes in a row were 64, then a match would have been found.

Observation: The Crypto library is a powerful tool that can be used to test many different encryption algorithms. The library provides easy to use methods that are implemented in the C programming language.

Explanation: By knowing the plaintext, ciphertext, initialization vector, and encryption algorithm, it is possible to find the key that was used to encrypt the plaintext. Brute force programming methods were used to test each possible key and compare the output ciphertext with the known ciphertext.

```c
# Anthony Redamonti
# Dr. Syed Shazli
# CSE-644: Internet Security
# 3-5-2023

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <openssl/bio.h> /* BasicInput/Output streams */
#include <openssl/err.h> /* errors */
#include <openssl/ssl.h> /* core library */
#include <openssl/evp.h>
#include <ctype.h>
#include <unistd.h>

#define BuffSize 1024
#define WordSize 16
#define InitVectorSize 32
```

```
int encrypt(unsigned char *plaintext, int plaintext_len, unsigned char *key, unsigned
char *iv, unsigned char *ciphertext)
{
   EVP_CIPHER_CTX *ctx;
   int len;
   int ciphertext_len;
   /* Create and initialise the context */
   if(!(ctx = EVP_CIPHER_CTX_new())){
      EVP_CIPHER_CTX_cleanup(ctx);
      printf("error occurred\n");
      return 0;
   }
    /*
    * Initialise the encryption operation. IMPORTANT - ensure you use a key
    * and IV size appropriate for your cipher
    * In this example we are using 256 bit AES (i.e. a 256 bit key). The
    * IV size for *most* modes is the same as the block size. For AES this
    * is 128 bits
    */
    if(1 != EVP_EncryptInit_ex(ctx, EVP_aes_128_cbc(), NULL, key, iv)){
      EVP_CIPHER_CTX_cleanup(ctx);
      printf("error occurred\n");
      return 0;
    }

    /*
    * Provide the message to be encrypted, and obtain the encrypted output.
    * EVP_EncryptUpdate can be called multiple times if necessary
    */
    if(1 != EVP_EncryptUpdate(ctx, ciphertext, &len, plaintext, plaintext_len)){
      EVP_CIPHER_CTX_cleanup(ctx);
      printf("error occurred\n");
      return 0;
    }

    ciphertext_len = len;

    /*
    * Finalise the encryption. Further ciphertext bytes may be written at
    * this stage.
    */
   if(1 != EVP_EncryptFinal_ex(ctx, ciphertext + len, &len)){
      EVP_CIPHER_CTX_cleanup(ctx);
      printf("error occurred\n");
      return 0;
   }

   ciphertext_len += len;
   return ciphertext_len;
}

int main(){

   unsigned char plainText[] = "This is a top secret.";
```

```
   unsigned char cipherText[] = 
"764aa26b55a4da654df6b19e4bce00f4ed05e09346fb0e762583cb7da2ac93a2";
   unsigned char* initializationVector = (unsigned 
char*)"aabbccddeeff00998877665544332211";

   int plainTextLen = strlen((const char*)plainText);
   unsigned char word[WordSize];
   unsigned char cipherOutput[64];
   int outputBufferLength;

   FILE *wordsListFile = fopen("WordsList.txt", "r");

   int max = 0;
   int count = 0;
   int wordSelector = 150; // select a word to print to the console

   while(fgets(word, WordSize, wordsListFile)){

      int wordLen = strlen(word);
      word[wordLen-2] = '\0'; // end the word with the NULL character
      wordLen = strlen(word);

      // adjust the padding of each word if needed
      // all words should be size 16
      while(wordLen < WordSize){
         word[wordLen] = '#';
         wordLen++;
         if(wordLen == WordSize){
            word[wordLen] = '\0';
         }
      }

      unsigned char* key = (unsigned char*)word;
      if(count == wordSelector){printf("key: ");}

      // print the key for debugging purposes
      for(int j = 0; j < wordLen; j++){
          if(count == wordSelector){
         printf("%c", key[j]);
         if(j == wordLen-1){printf("\n");}
          }
      }

      // length of key is now 16 bytes
      // convert key to hexadecimal
      unsigned char formattedKey[33];

      for(int i = 0; i < 16; i++){
            sprintf(formattedKey+2*i, "%.2x", key[i]);
      }

      formattedKey[32] = '\0';

      if(count == wordSelector){printf("formatted key: ");}
      // print the formatted key for debugging purposes
```

```
      for(int j = 0; j < 32; j++){
          if(count == wordSelector){
         printf("%c", formattedKey[j]);
         if(j == 31){printf("\n");}
          }
      }

      int cipher_len = encrypt(plainText, plainTextLen, formattedKey,
initializationVector, cipherOutput);

      unsigned char formattedOutput[65];

      // format the ciphertext to hexadecimal
      for(int i = 0; i < 64; i++){
         unsigned char ciphOutputByte[1];
         sprintf(ciphOutputByte, "%02x", cipherOutput[i]);
         unsigned char ciphTextByte = cipherText[i];
         formattedOutput[i] = ciphOutputByte[0];
      }
      formattedOutput[64] = '\0';
      if(count == wordSelector){printf("formatted cipher output: ");}

      // print the formatted key for debugging purposes
      for(int j = 0; j < 64; j++){
          if(count == wordSelector){
         printf("%c", formattedOutput[j]);
         if(j == 63){printf("\n");}
          }
      }

      int index = 0;
      while((index < 64) && (formattedOutput[index] == cipherText[index])){
          index = index + 1;
      }

      // record the maximum bytes we were able to match in a row
      if(index > max){max = index;}

      // if we matched all 64 bytes, we have found a solution
      if(max == 64){
          printf("match found!\n");
          printf("word: %s\n", word);
      }
      else{
         // nothing to do at this time.
      }

      count = count + 1;
   }

   printf("max num of bytes matching in a row: %d\n", max);
   printf("ciphertext solution: %s\n", cipherText);
   fclose(wordsListFile);
   return 0;
}
```