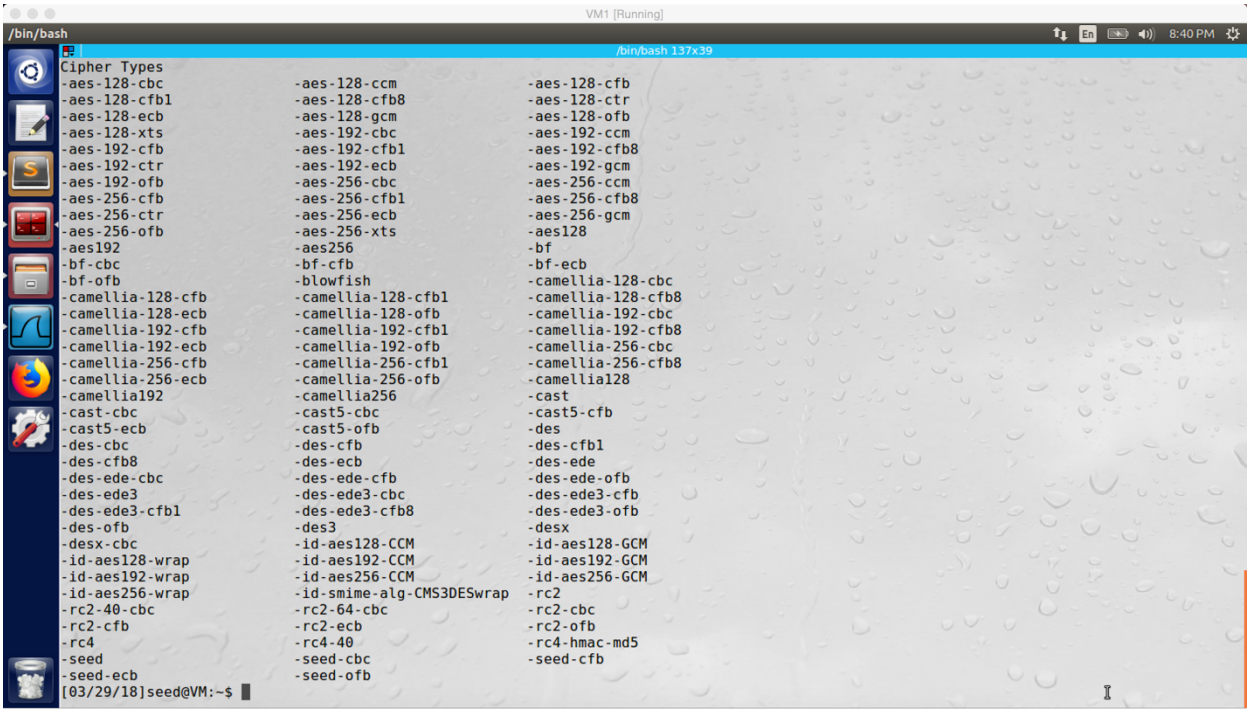
CSE644 Lab7
Yishi Lu
3/30/2018

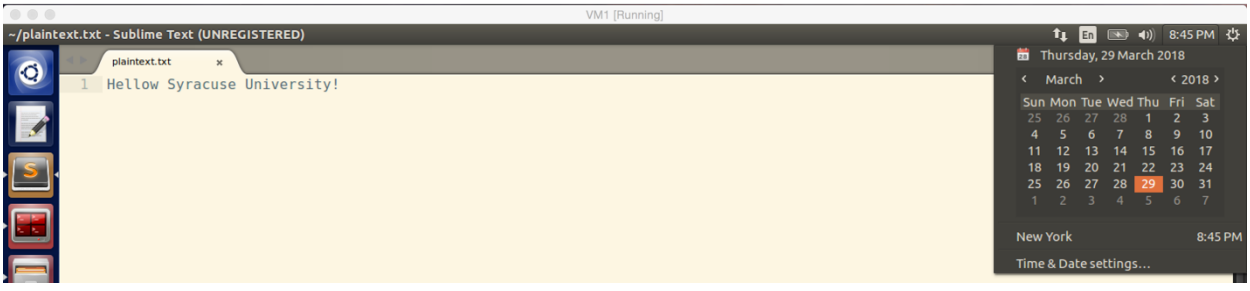
In this task, I use one VM, VM1 with IP address 10.0.2.20

Task1: Encryption using different ciphers and modes

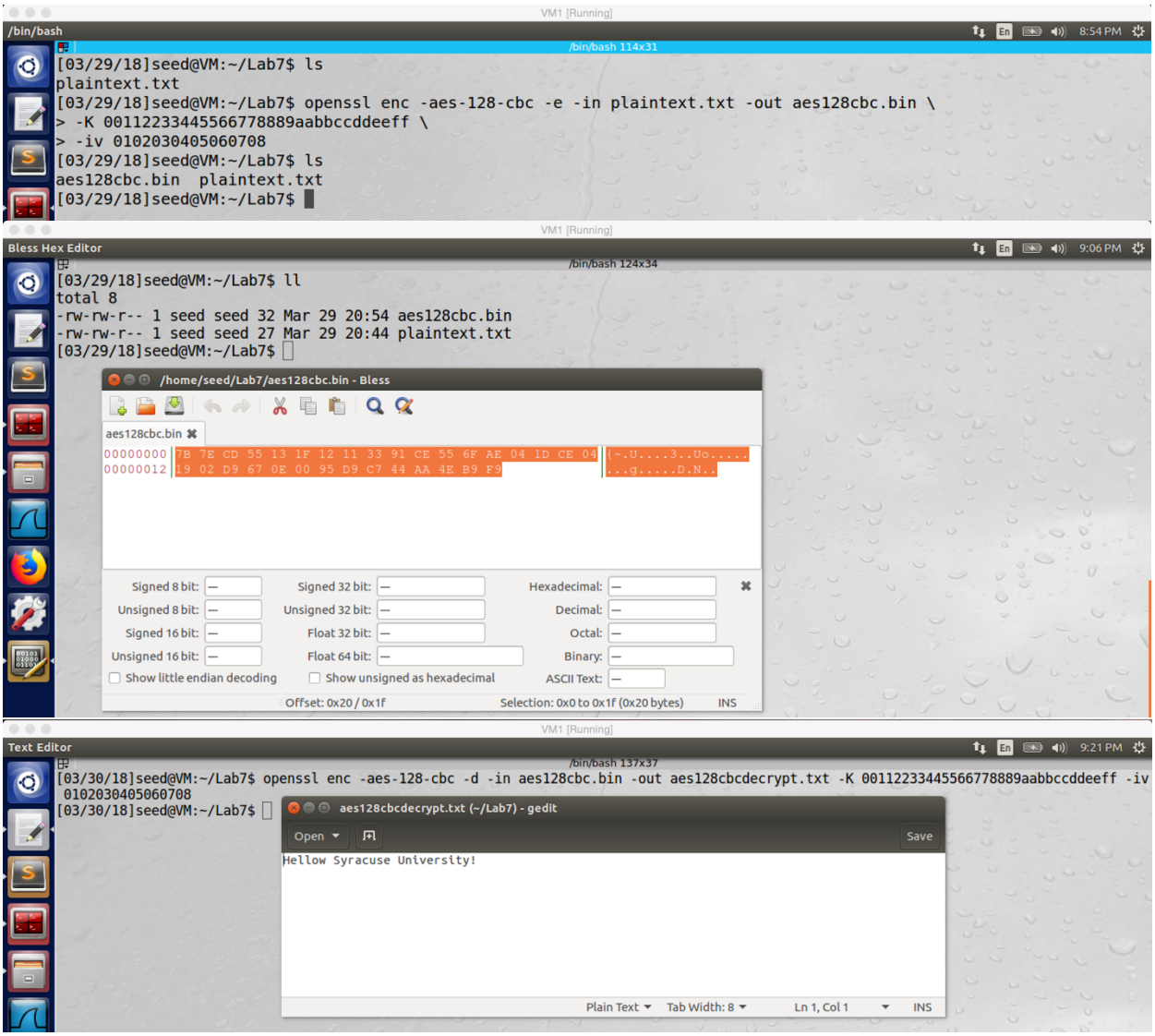
In this task, I will try to use four different ciphers and different modes, as the following screenshot shows:



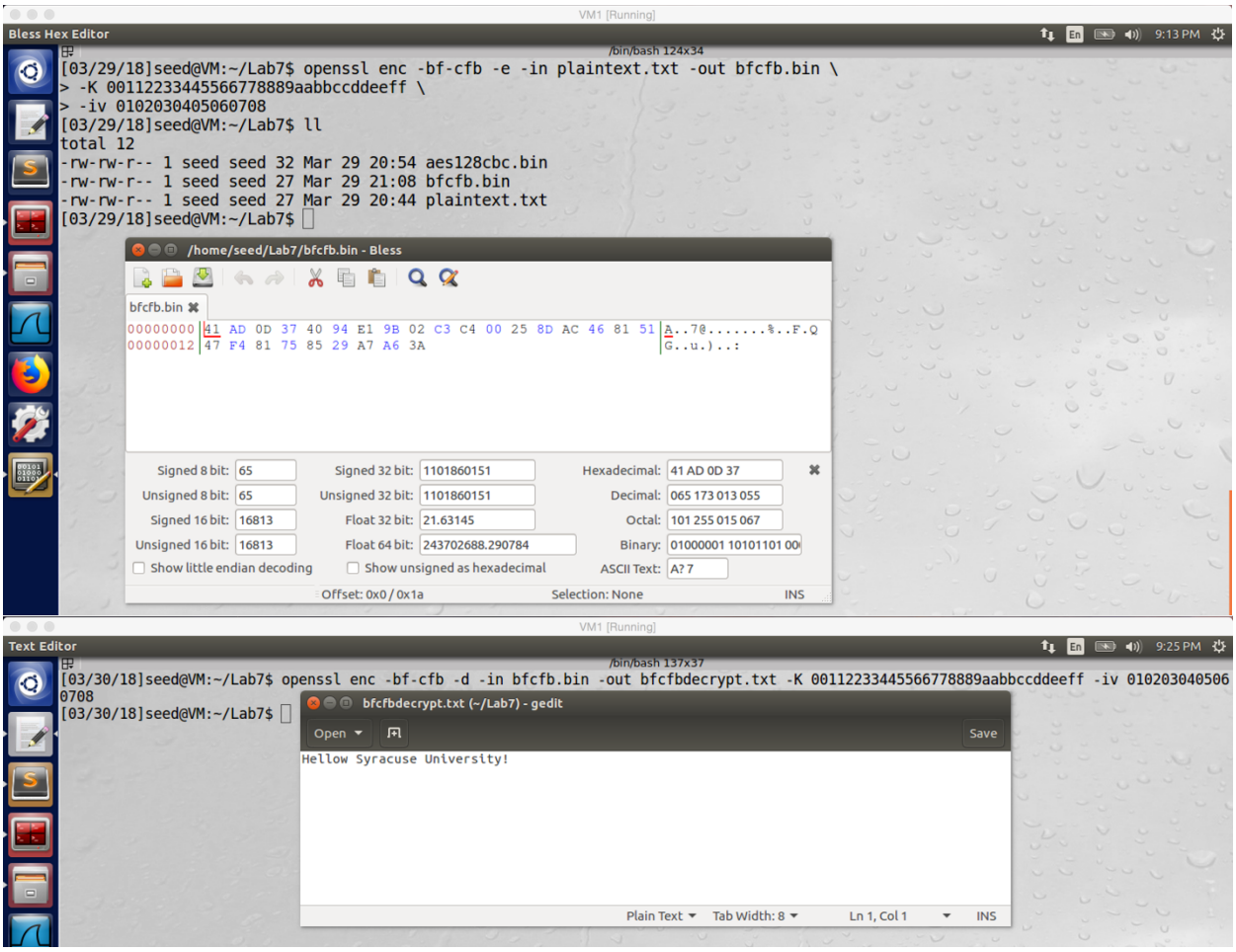
screenshot1. Using “openssl enc -help” command, it shows all ciphers and mode. I want to use –aes-128-cbc, -bf-cfb, –aes-256-ccm, and –cast5-ofb to do this task (four different ciphers with four different modes).



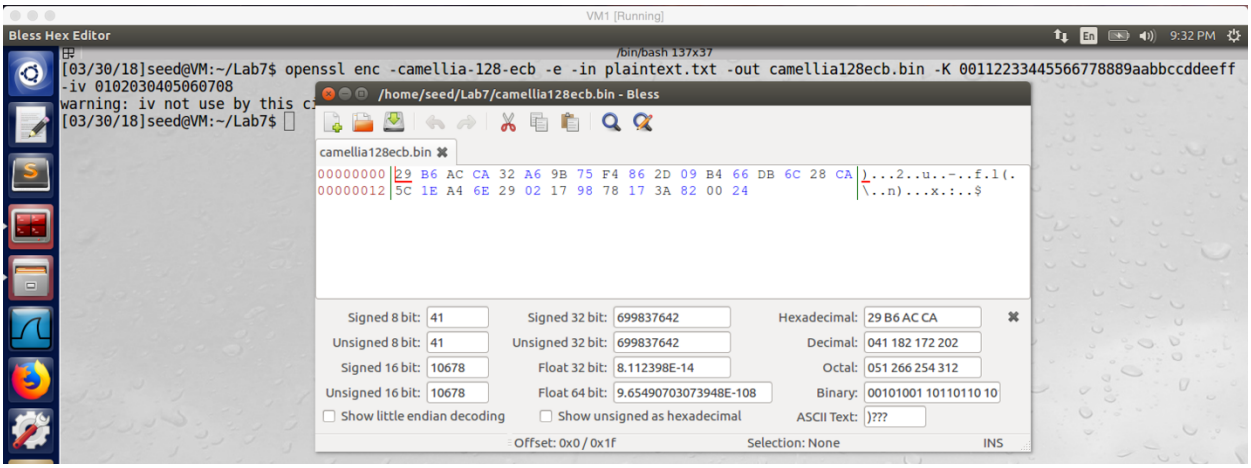
screenshot2. Before I use above ciphers and modes, I create a plaintext

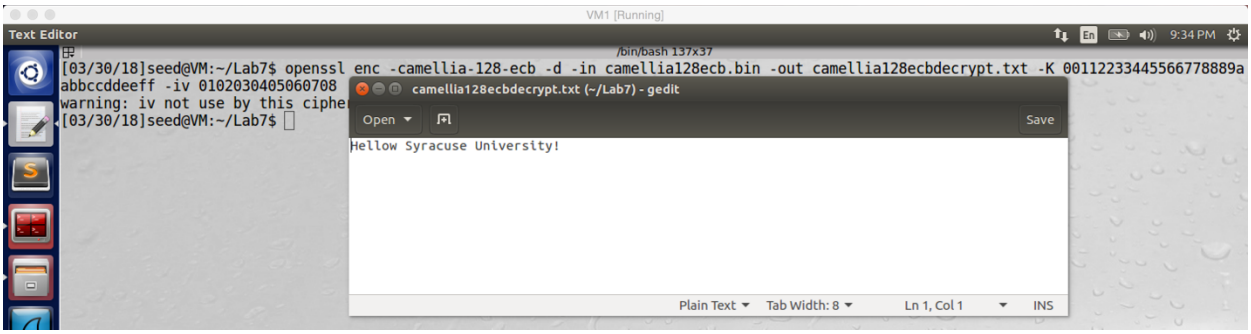


Screenshot3. I use –aes-128-cbc to encrypt plaintext.txt, and then I use hex editor to open the encrypted file. Afterwards, I also decrypt the encrypted file, and I get same content as the original file.

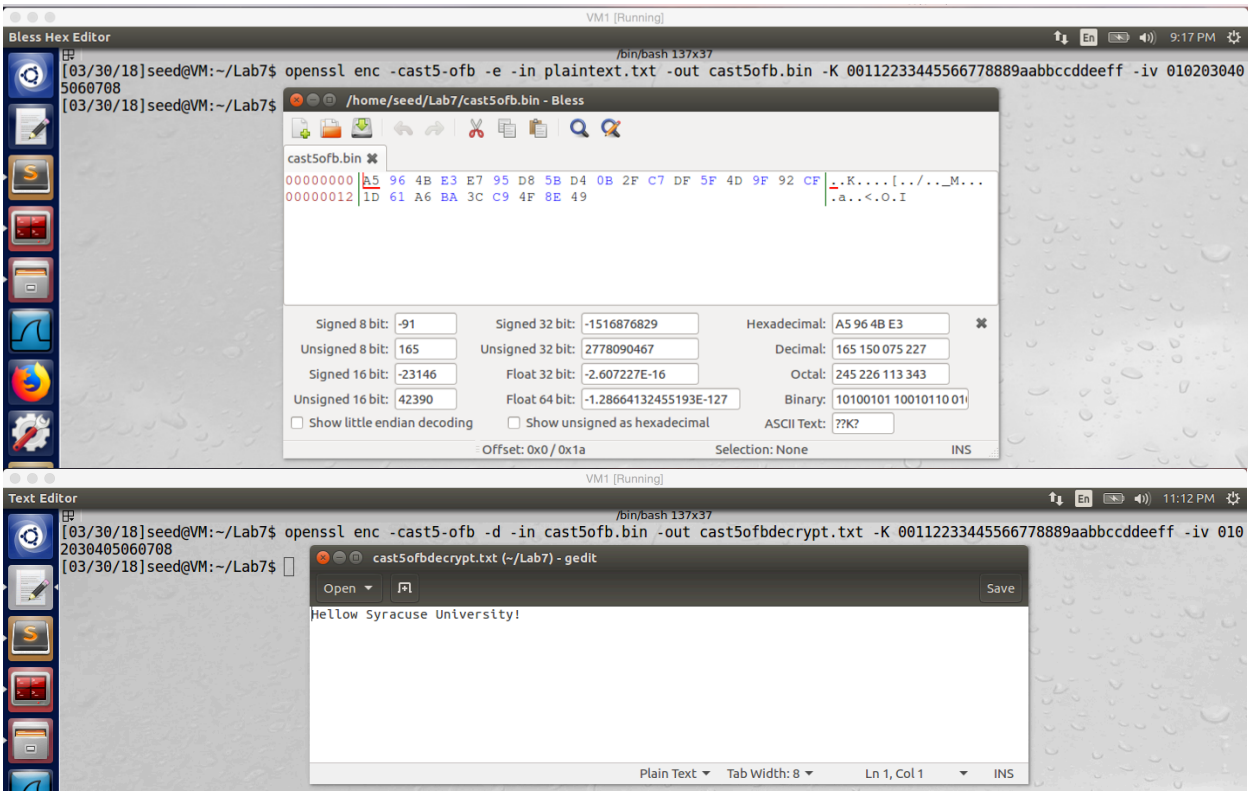


screenshot4. I use –bf-cfb to encrypt encrypt plaintext.txt, and then I use hex editor to open the encrypted file. Afterwards, I also decrypt the encrypted file, and I get same content as the original file.





screenshot5. I use –camellia-128-ecb to encrypt encrypt plaintext.txt, and then I use hex editor to open the encrypted file. Afterwards, I also decrypt the encrypted file, and I get same content as the original file.



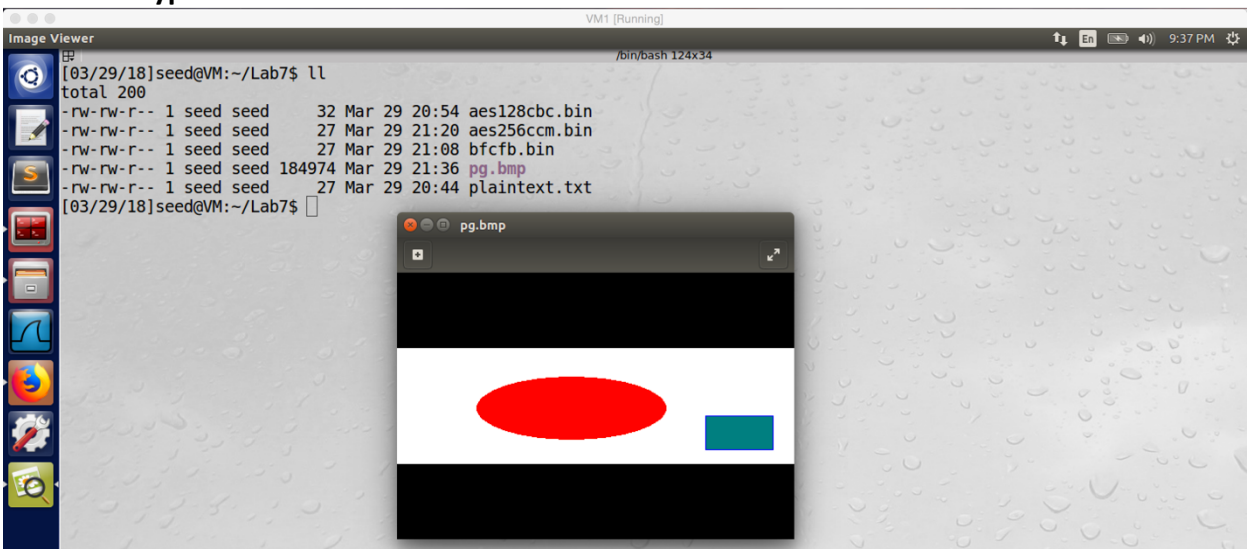
screenshot6. I use –cast5-ofb to encrypt encrypt plaintext.txt, and then I use hex editor to open the encrypted file. Afterwards, I also decrypt the encrypted file, and I get same content as the original file.

Observation and Explanation:

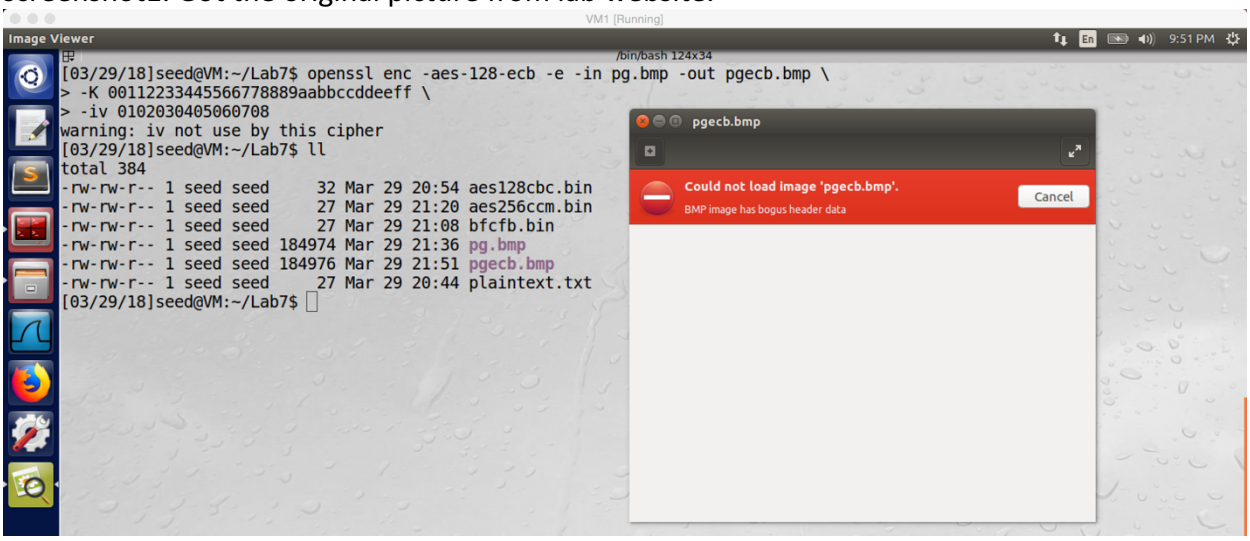
In this task, I used four different ciphers and modes. Before I do the encryption, I create a plaintext file (screenshot1), and this file contains string “Hello Syracuse University”. Firstly, I use aes128 with mode cbc to encrypt the plaintext.txt and decrypt the encrypted file (screenshot3). Secondly, I use bf with mode cfb to encrypt the plaintext.txt and decrypt the encrypted file (screenshot4). thirdly, I use camellia128 with mode ecb to encrypt the

plaintext.txt and decrypt the encrypted file (screenshot5). Finally, I use cast5 with mode ofb to encrypt the plaintext.txt and decrypt the encrypted file (screenshot6). As the screenshot shows, different ciphers and modes produce different encryption codes for same file.

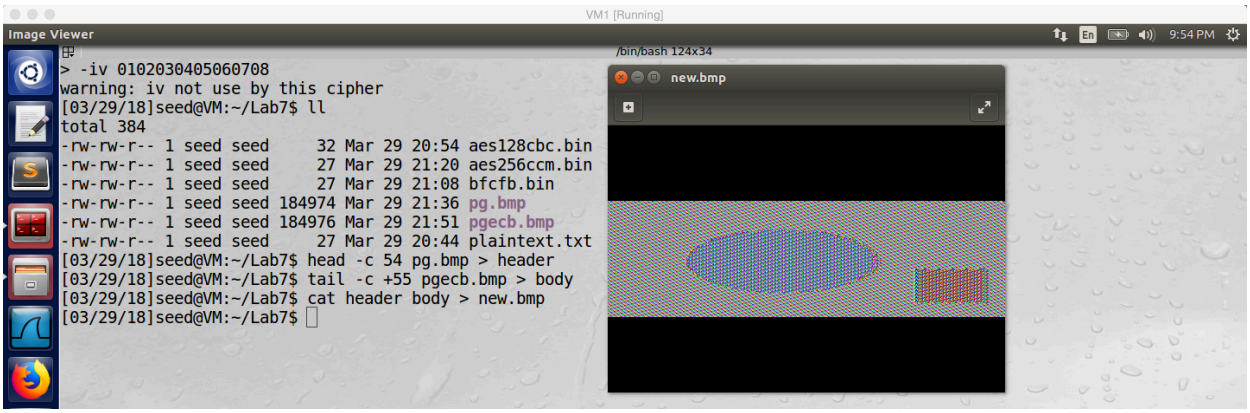
Task2: Encryption Mode – ECB vs. CBC



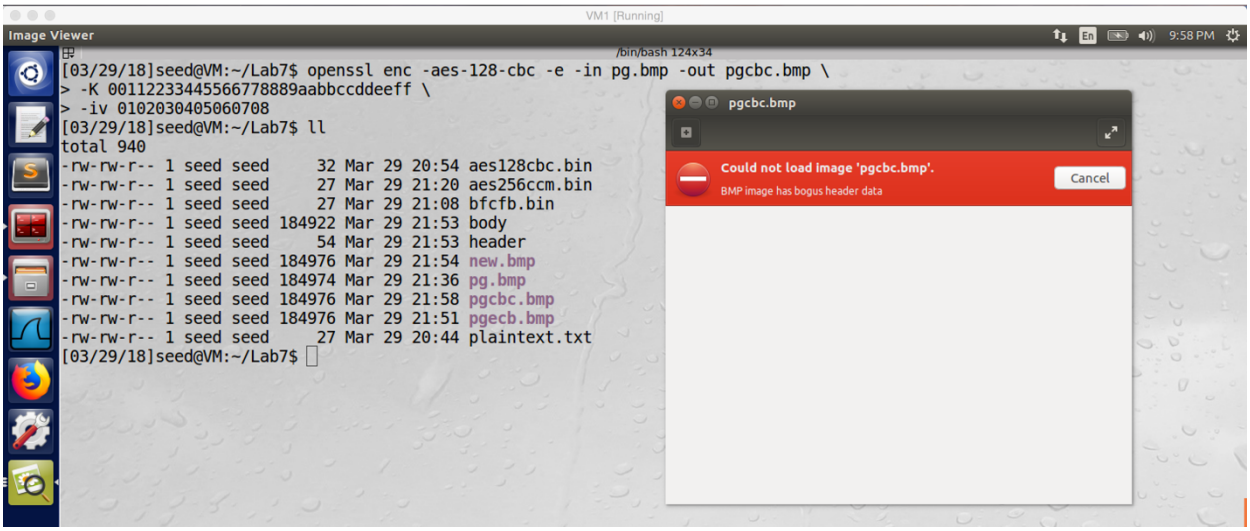
screenshot1. Got the original picture from lab website.



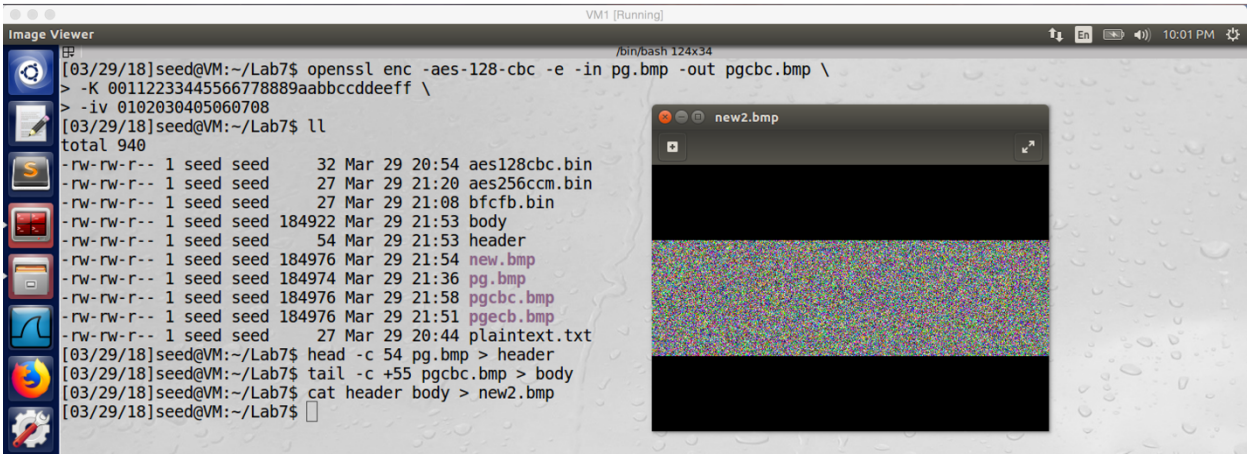
screenshot2. After encrypt pg.bmp with ECB, the image cannot be load because the header information is not correct



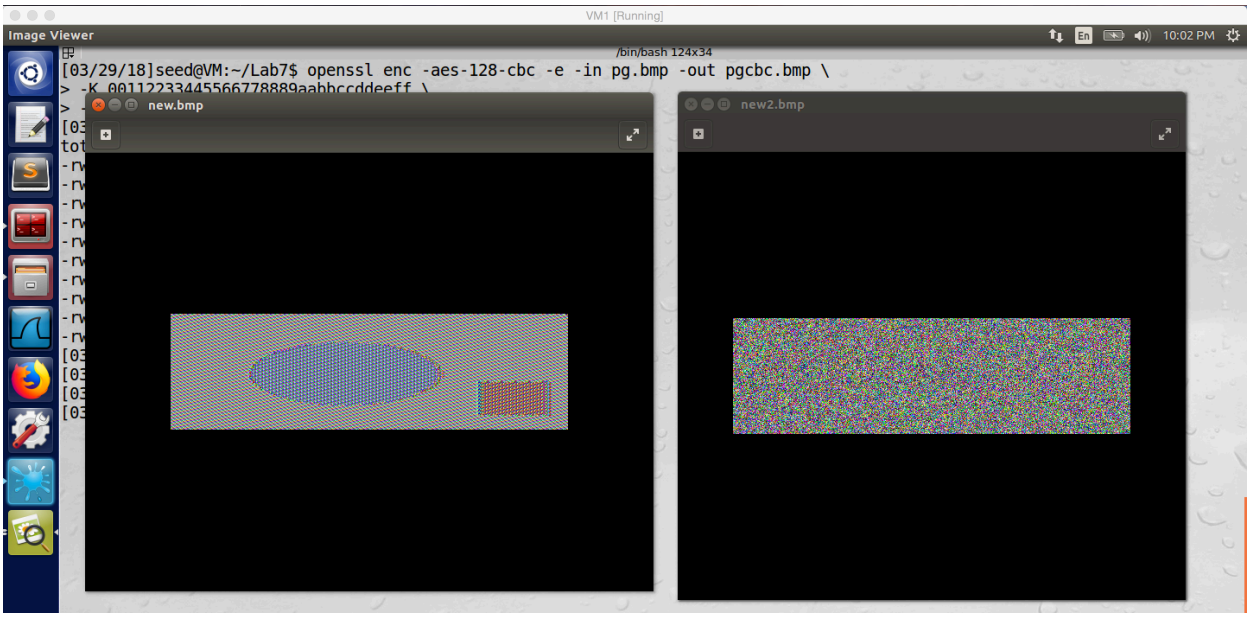
screenshot3. After replacing the header of encrypted image pgecb.bmp with original image bg.bmp, we got a viewable image



screenshot4. After we encrypted pg.bmp with CBC, the same same happened again, the image could not be load because missing header information



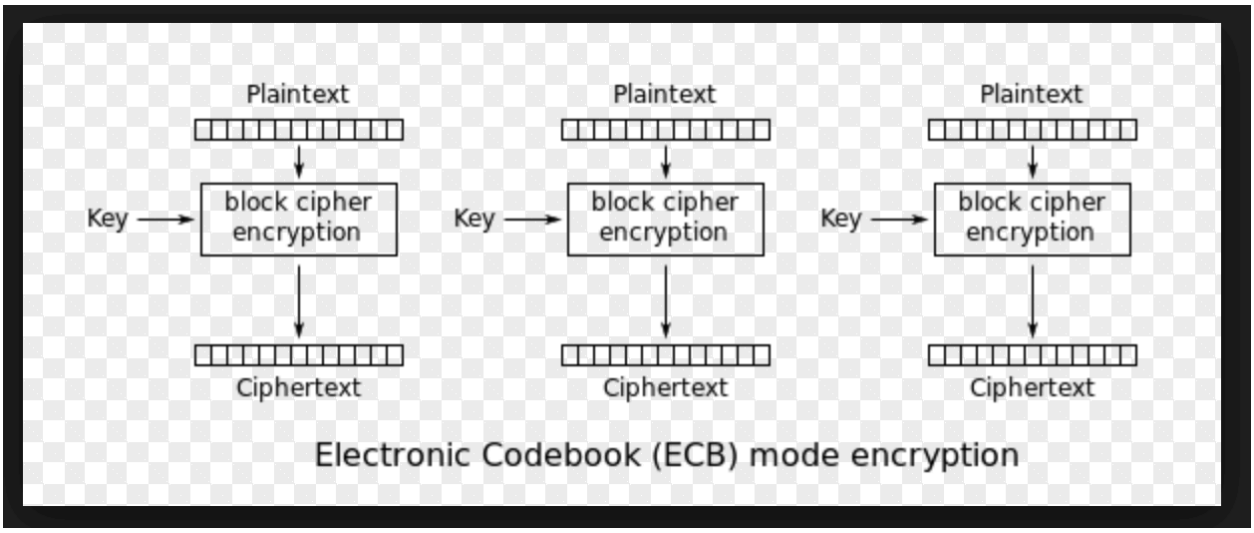
screenshot 5. After we add correct header into the pgcbc.bmp, the image is loadable.



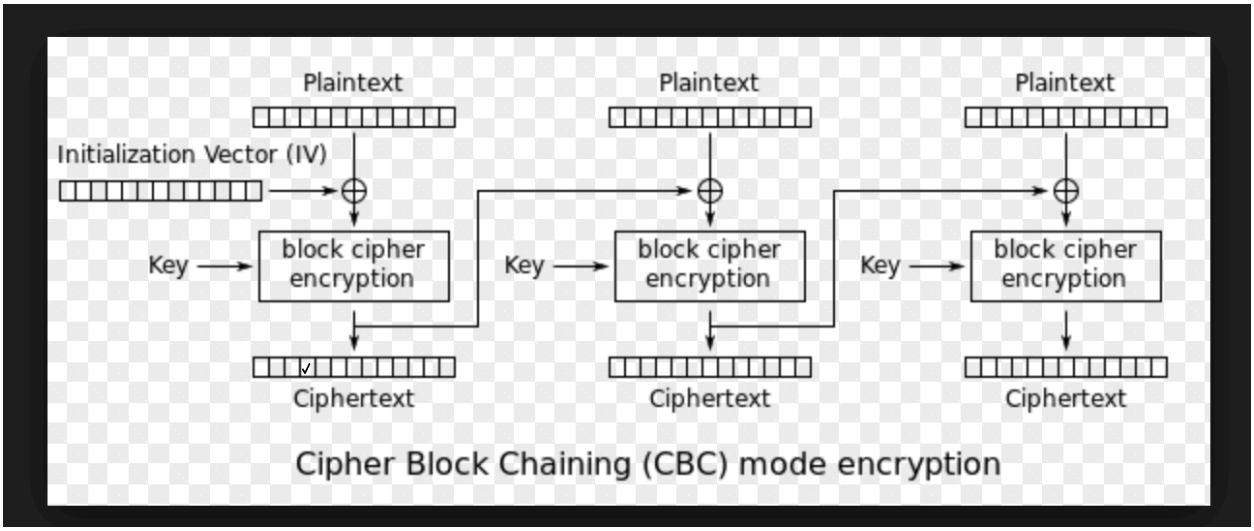
screenshot6. We put these two images together, the left one is encrypted by ECB, and the right one is encrypted by CBC. We can clearly see the pattern of left image, but we just see noise in the right image.

Observation and Explanation:

In this task, we use two different modes of aes128 to encrypt an image. As the first screenshot shows, there are two patterns in the image, one is oval and one is rectangle. We first encrypt the image with ECB mode; after we encrypted, the encrypted file is not loadable (screenshot2). The reason is after we encrypted the image, the header information (first 54 byte) is changed, so we need to put correct header information into it. By achieving this, we can take the first byte of original image and concatenate it with the rest of bytes in the encrypted file. After doing this, the image becomes loadable. And we can see that even the image is encrypted, the pattern is still clear (one oval, one rectangle); so using ECB on an image is not secure (screenshot3). Afterwards, we also use CBC to encrypt the original image. After we encrypted the image and add correct header information, we can open the encrypted image. As screenshot5 shows, there are just noise in the encrypted image, we cannot get any useful information with the encrypted image.



With ECB mode, the key is direct applied into the encryption function, and every block is independent from each other. Therefore, if the plaintexts are same, then the cipher texts will be same as well. Fore the image which we used in the task, because a lot of pixels' value in the image are same, except the edge part of the patterns. Therefore, after we encrypted the image, the edge becomes different, but the other area is almost the same. So we can clearly see the image pattern from the encrypted image. Therefore, ECB is not secure on image



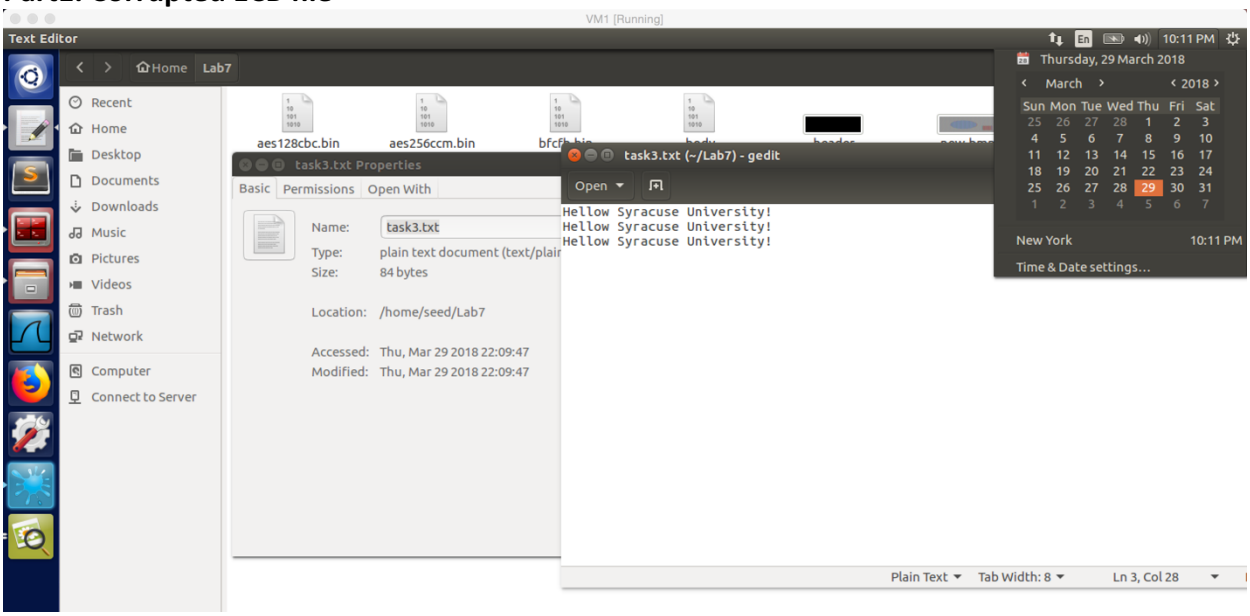
With the CBC mode, every block is not independent from each other. For the first block, IV value is introduced, it is not secret but unpredictable. The IV value XOR with the plaintext, this make the result to be more secure. After cipher text is produced, it will be feed to next block as IV values. Therefore, even the plaintexts are same, the cipher text will never be the same. As a result, after we applied CBC mode on the original image, we cannot get any useful information from the encrypted image file, so CBC mode is more secure than ECB in image encryption.

Task3: Encryption Mode – Corrupted Cipher Text

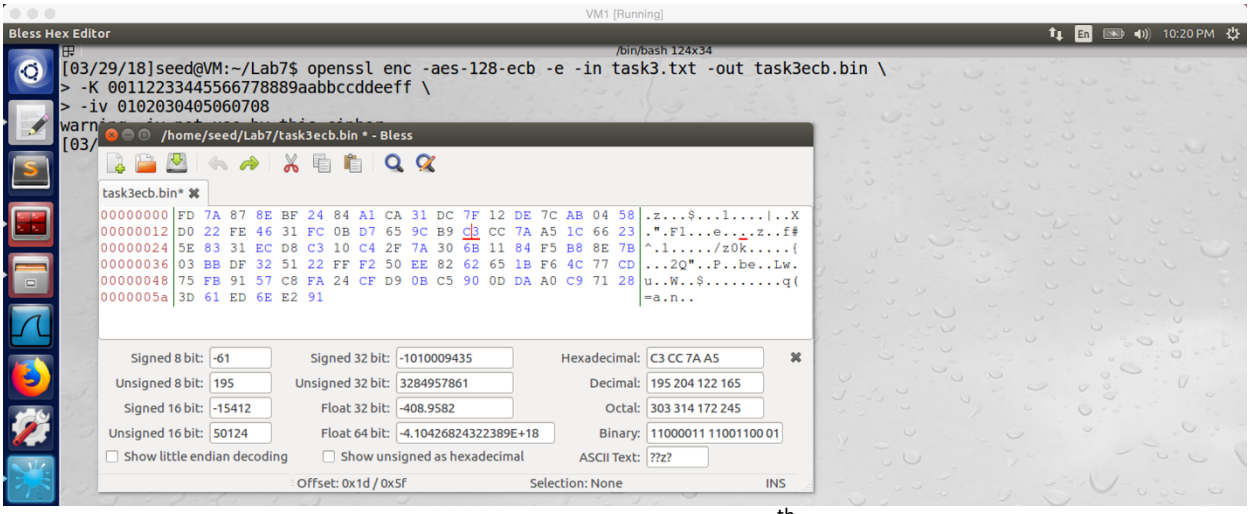
Answer for question1 before do this task:

For ECB, I guess only the block with corrupted byte cannot be recovered, the other block will be recover. For CBC, I guess the block with the corrupted byte and blocks after the corrupted byte cannot be recovered, so blocks before the block with corrupted byte will be recover. For CFB, I guess only the block with the corrupted byte cannot be recovered, other blocks will be recover. For OFB, I guess only the corrupted byte cannot be recovered.

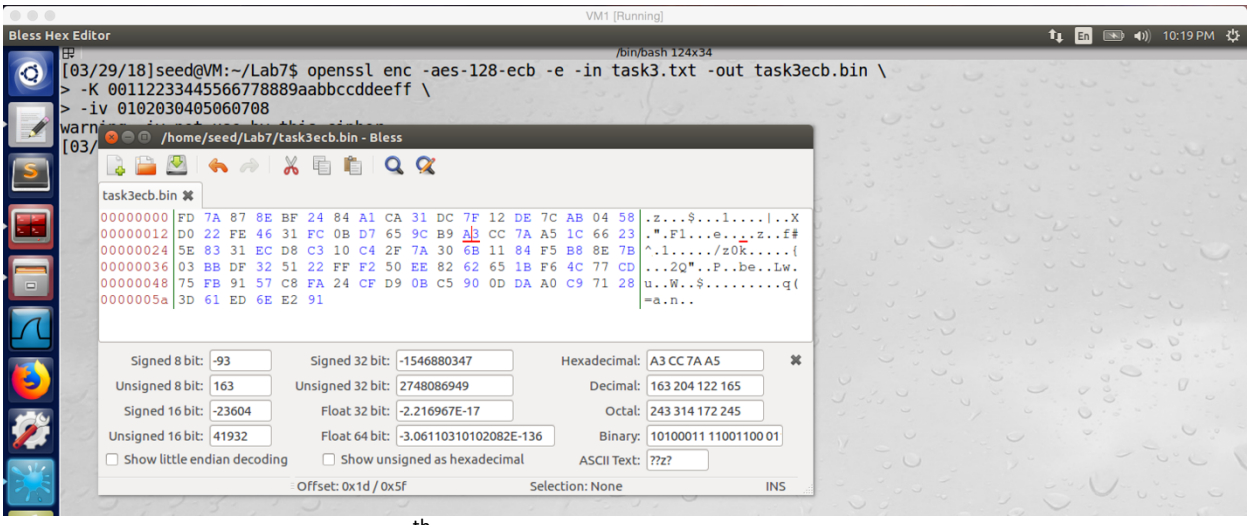
Part1. Corrupted ECB file



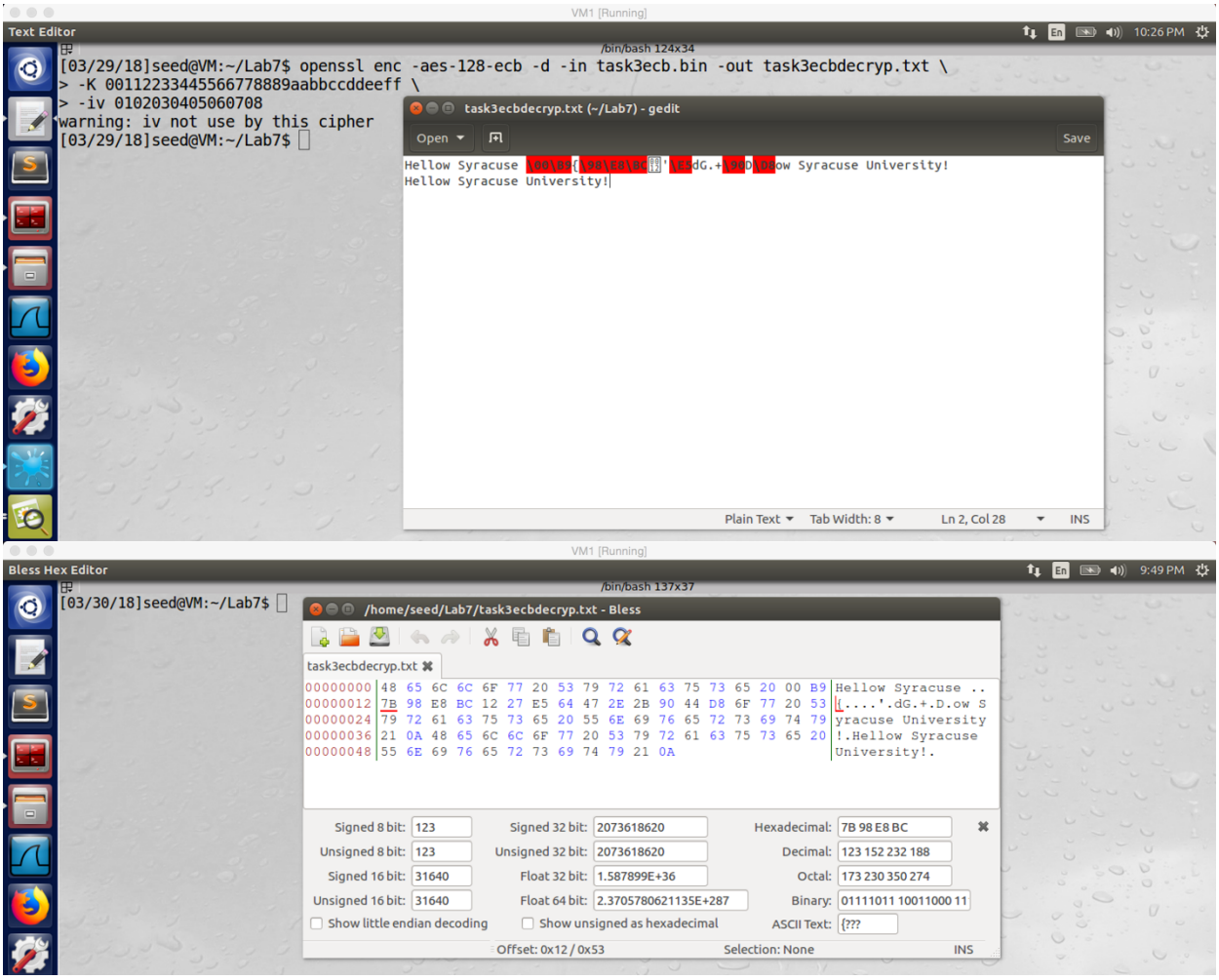
screenshot1. I create a plaintext file (task3.txt) with 84bytes.



screenshot2. Encrypted the file by using aes128 ECB, the 30th byte in original file is C

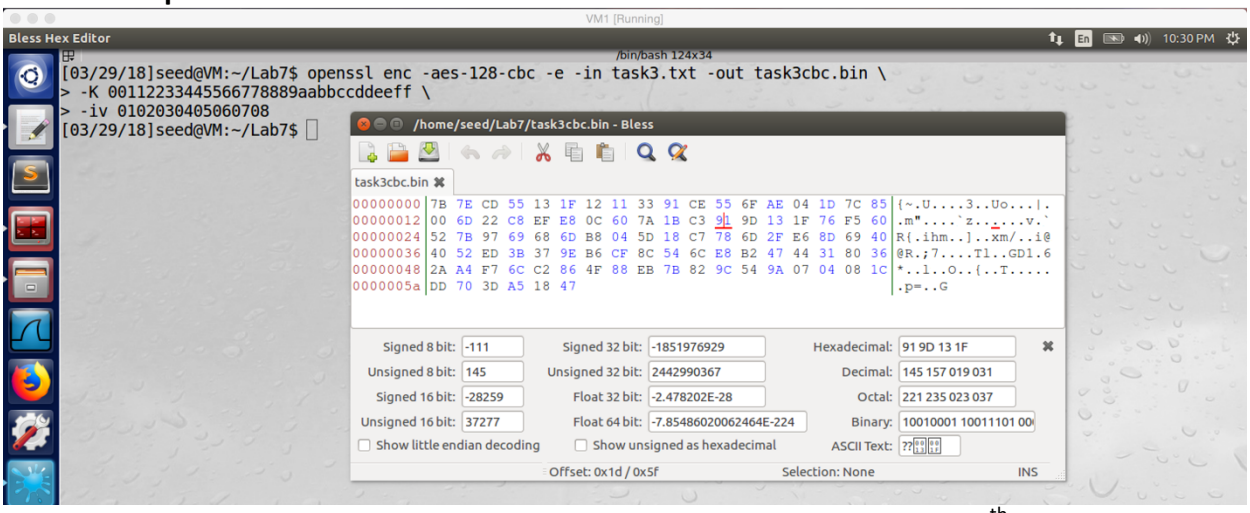


screenshot3. We change the 30th byte from C to A.

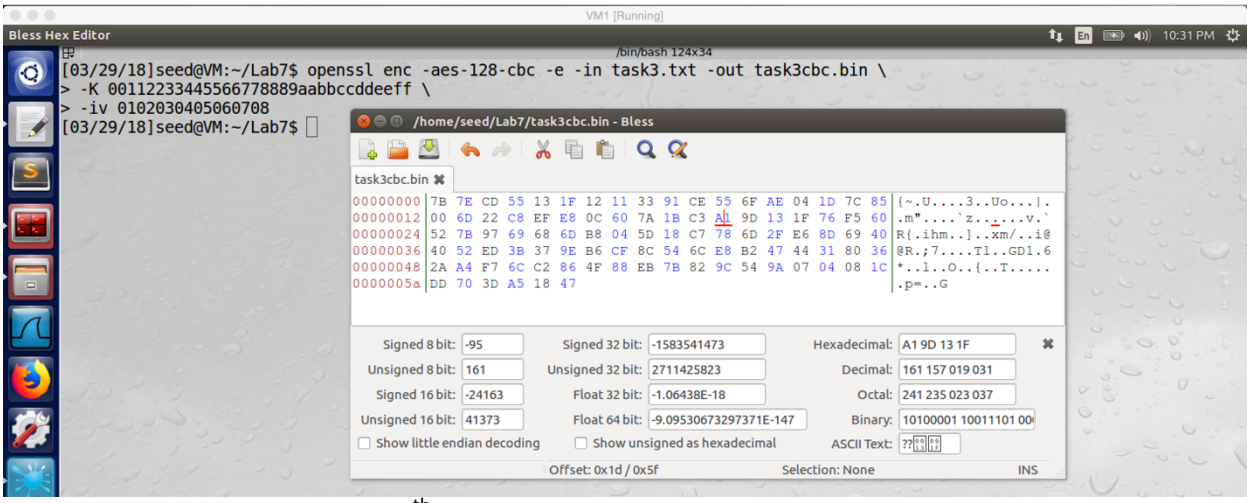


screenshot4. After decrypted the corrupted ECB file, words in the second block cannot be read

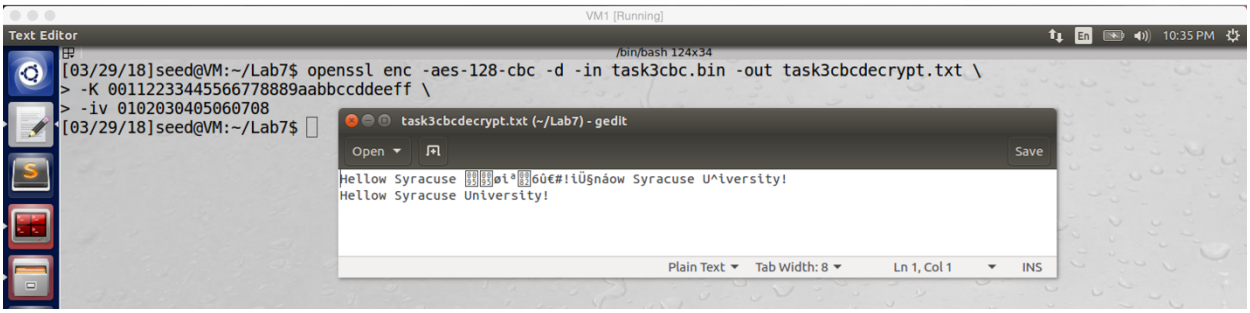
Part2. Corrupted CBC file

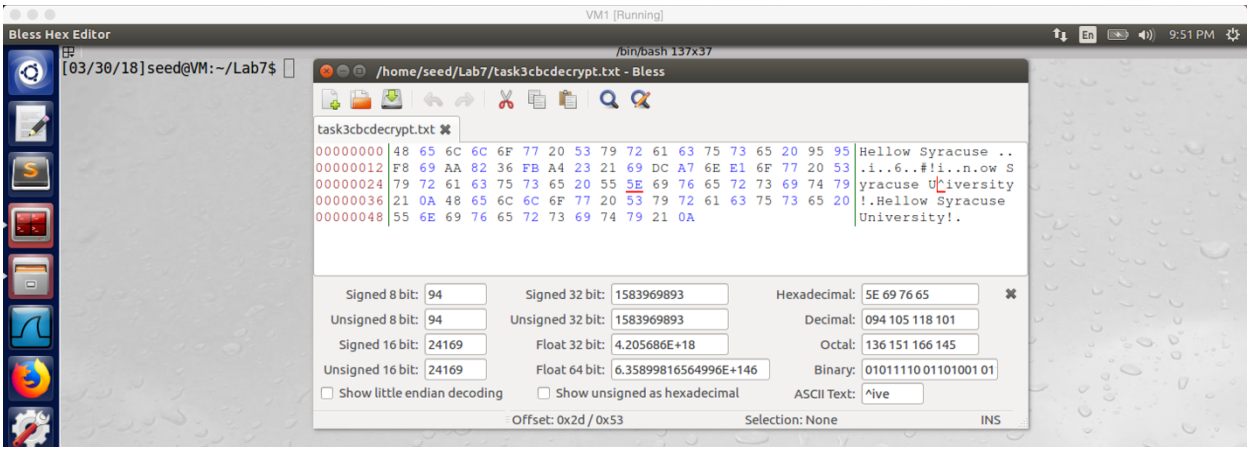


screenshot1. We encrypt the task3.text file by using aes128 CBC, and the 30th byte is 9



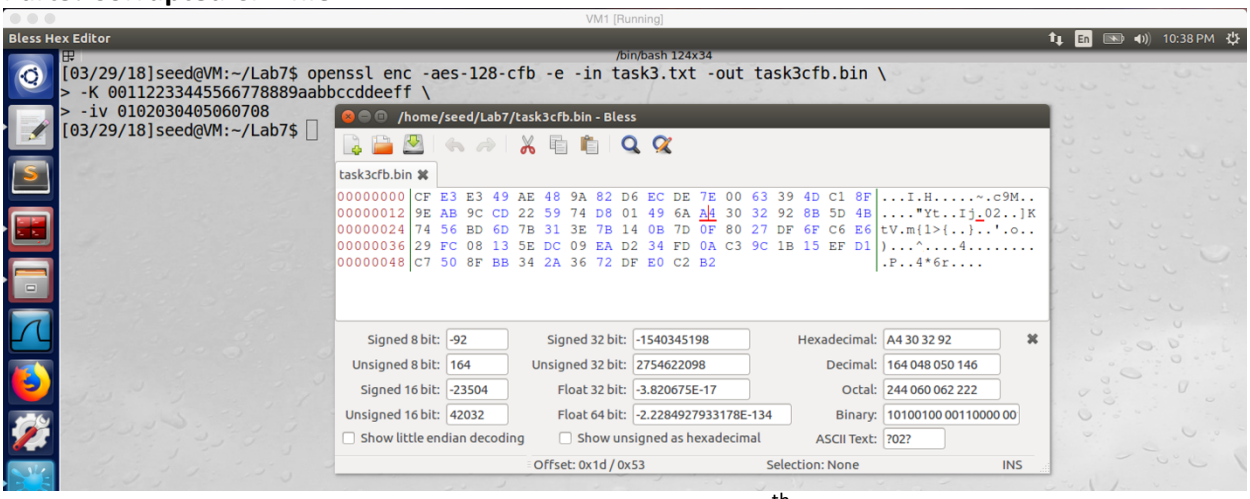
screenshot2. We change the 30th byte 9 with A.



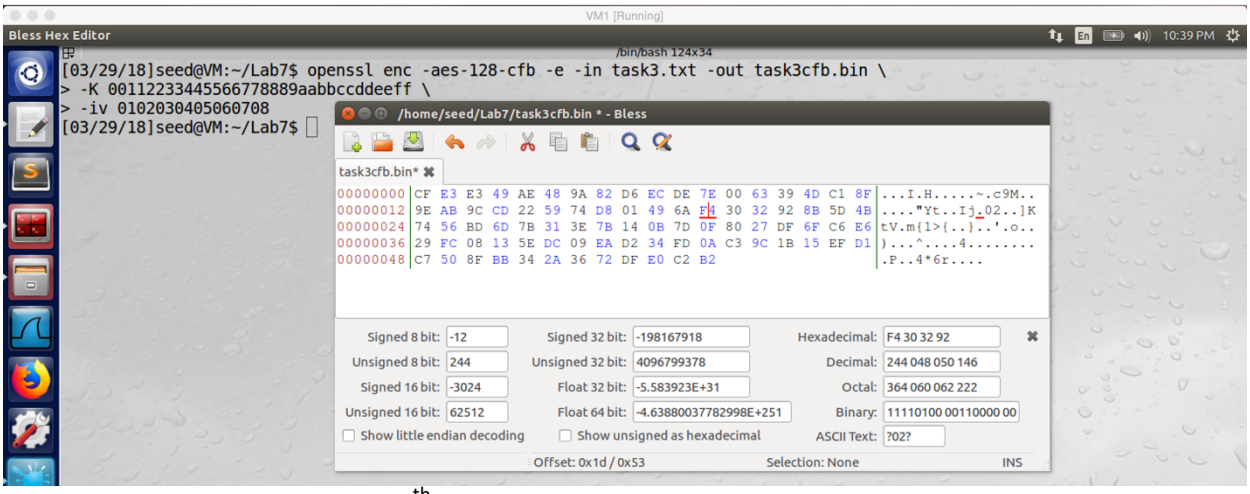


screenshot3. After decrypted the corrupted CBC file, the second block and a byte in the next block are corrupted

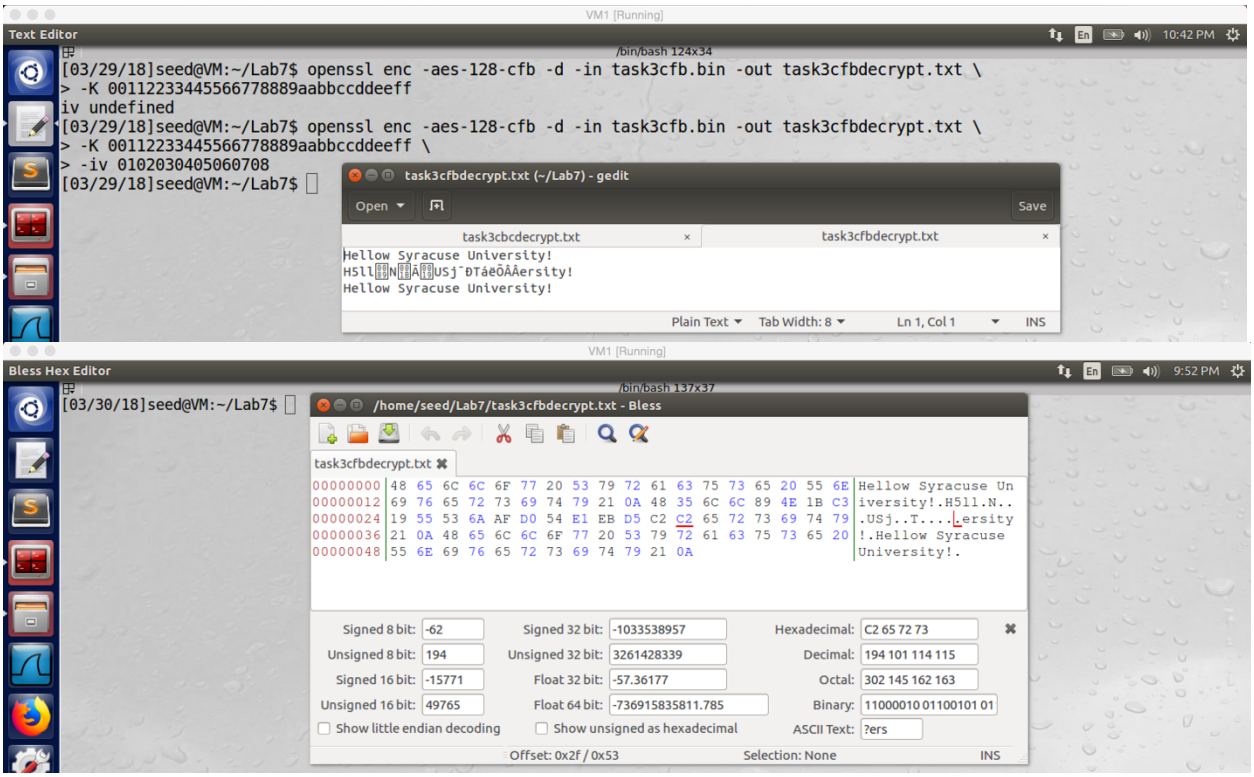
Part3. Corrupted CFB file



screenshot1. Encrypt task3.txt with aes128 CFB, and the 30th byte is A

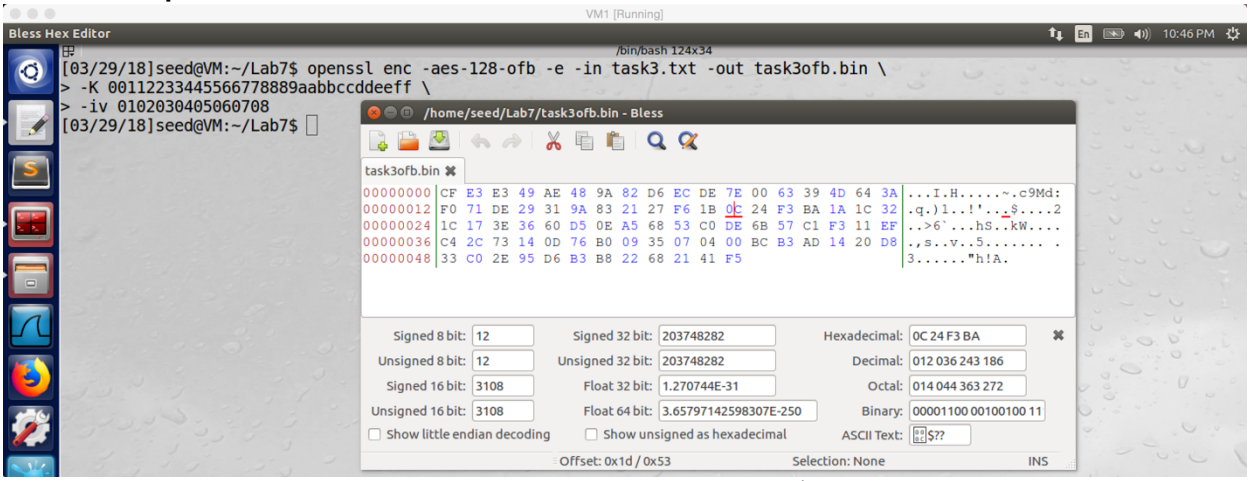


screenshot2. We change the 30th byte to F

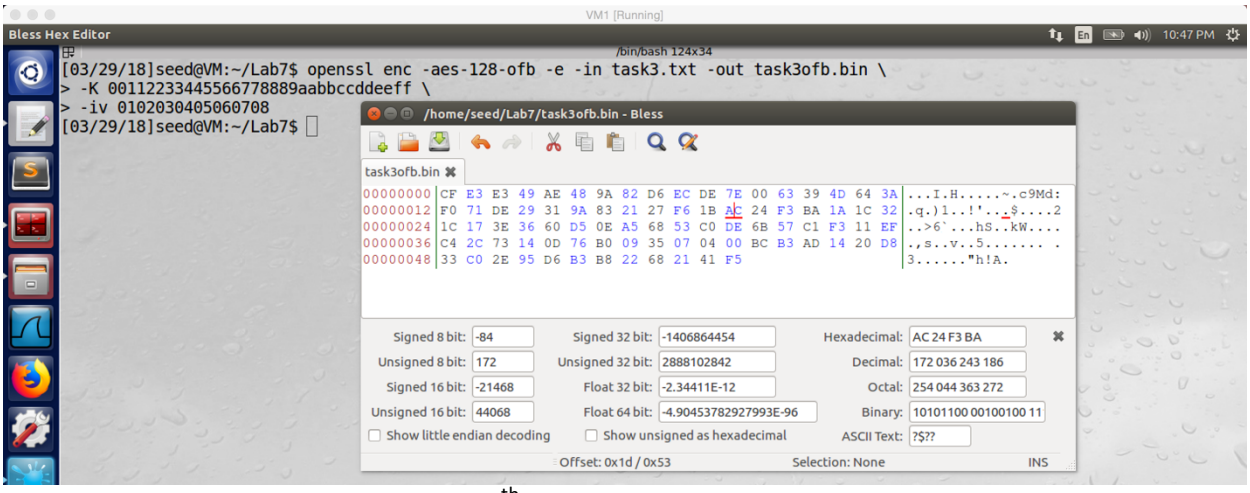


screenshot3. After decrypted the corrupted CFB file, words in the second and third block cannot be read

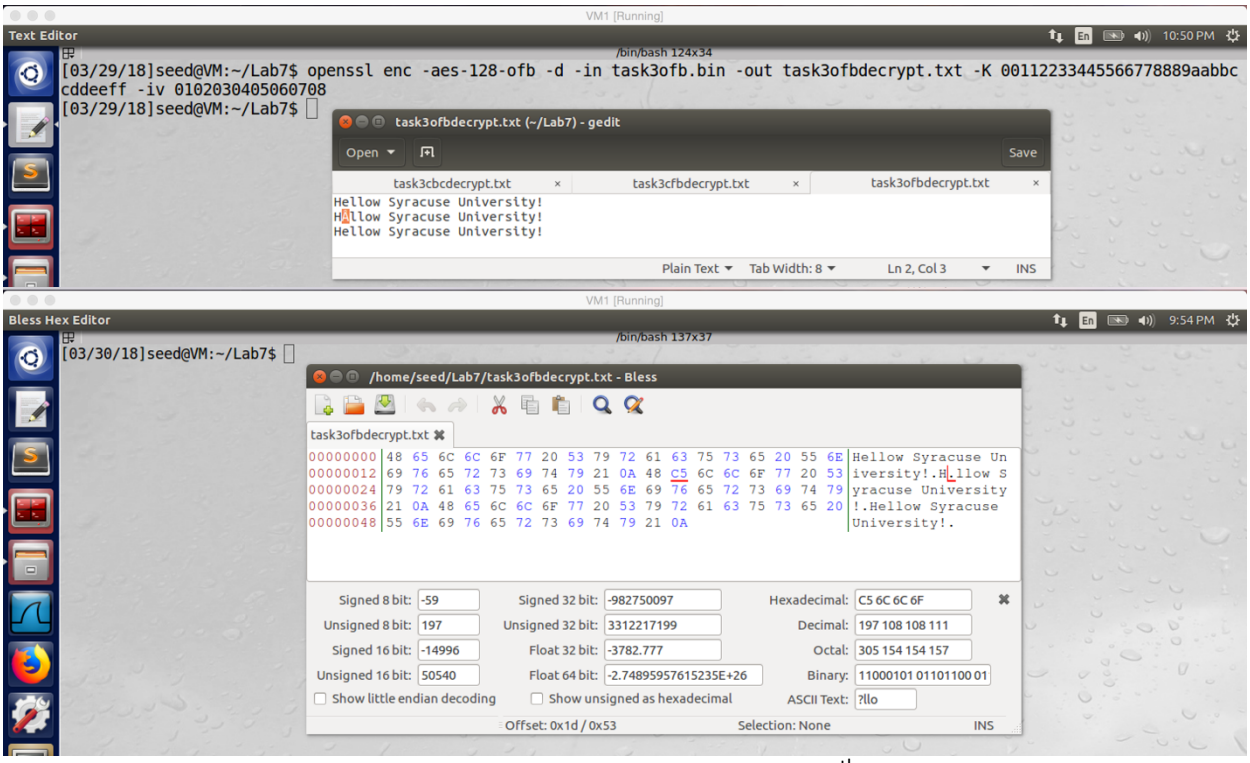
Part4. Corrupted OFB file



screenshot1. After encrypt task3.txt with aes128 OFB, the 30th byte is 0



screenshot2. Using A to change the 30th byte to 0



screenshot3. After decrypted the corrupted OFB file, only the 30th byte is corrupted

Observation and Explanation:

For this task, we have four parts. First part, we encrypted task3.txt with ECB mode; and then we use hex editor to open the encrypted file, we change the 30th byte from C to A. so the file is corrupted. And then I decrypted the corrupted file, we can see (part1 screenshot4) the middle words are unreadable. For the second part to fourth part, we did basically the same thing. The only difference is, for part 2, we use CBC to encrypt the file and change 30th byte to A

with 9. For part 3, we use CFB to encrypt and change 30th byte to F with A. For part 4 we use OFB and change 30th byte to A. With different cipher mode, the results are also different. After finished the task, I examine my answer in question 1 again.

Answer for question 2:

For ECB, I am correct, only current block is corrupted because each block is independent; therefore, only the block with corrupted byte will be corrupted after decryption.

For CBC, I was wrong, I think block after 30th byte will all be corrupted. Actually, only the block with corrupted byte and one byte in the next block are corrupted. The reason is, after we feed corrupted block to the next XOR, because only one byte in the block is corrupted, so only one byte will be affected in the next block. So when we decrypted, only that byte is corrupted as well. For the current block, because we need to send it to cipher function to decrypt, even there are only one byte is different, the whole block will be corrupted after decrypt. That's why we get such result shows in Part2 screenshot3.

For CFB, I guess only current block is corrupted; but, actually second and third block are corrupted. In CFB, if there are x bytes are lost from cipher text, then the cipher will output corrupted plaintext until the shift register equals the state when it doing encryption again. Therefore, n/s block size will be corrupted, the n is block size, and the s is the amount of bits shifted.

For OFB, I'm right, only the corrupted byte is affected. In OFB, the IV is encrypted by the cipher function and join plaintext with XOR to produce cipher text. Therefore, if one byte is corrupted in the cipher text, when we decrypt, the plaintext can be obtained from XOR of corrupted cipher text and the uncorrupted encrypted IV, so only one byte in the plaintext will be affected.

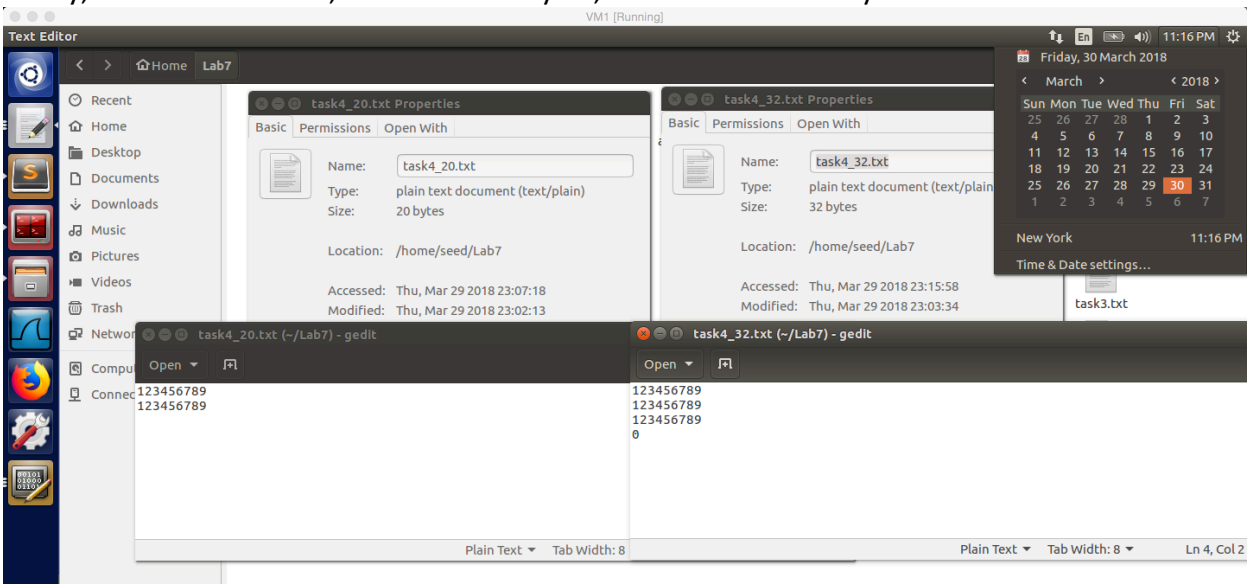
Answer for question 3:

As the above result, we can imply that different mode has different advantage and disadvantage. For the recovery from corrupted cipher text, OFB has the most promising result.

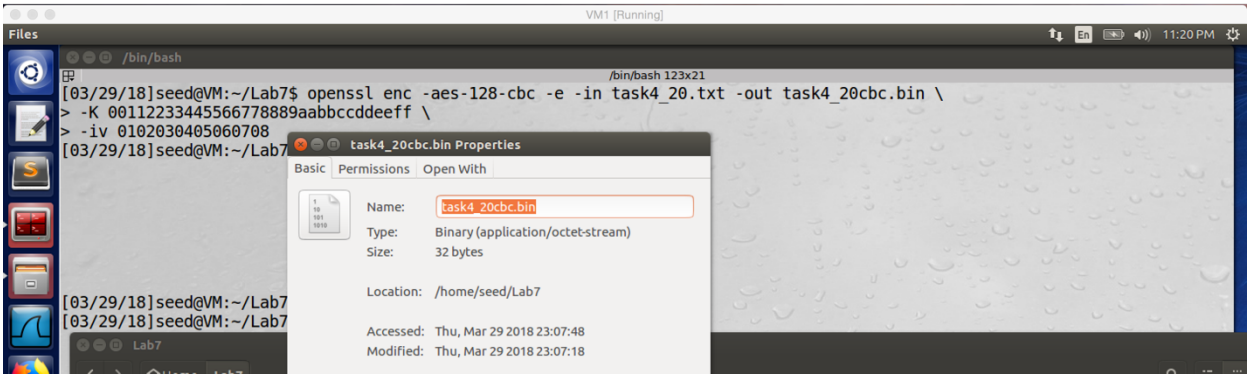
Task4: Padding

Part1:

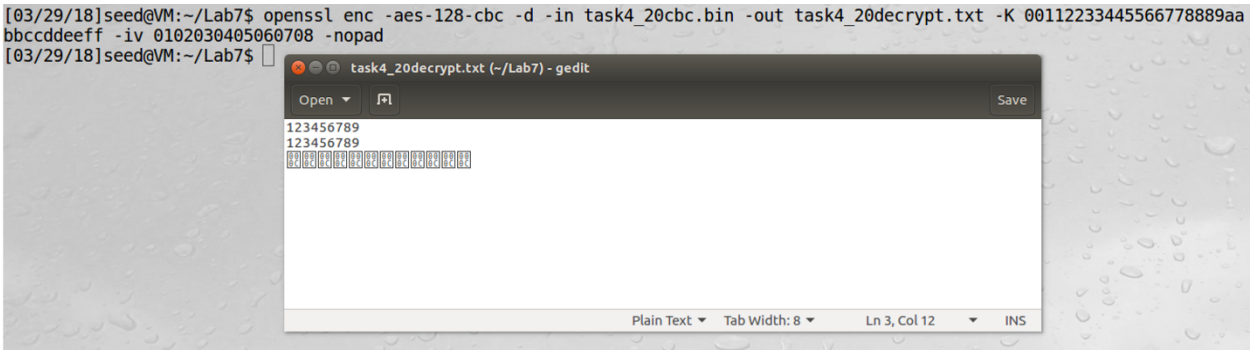
Firstly, I created two files, one size is 20 bytes, another one is 32 bytes.

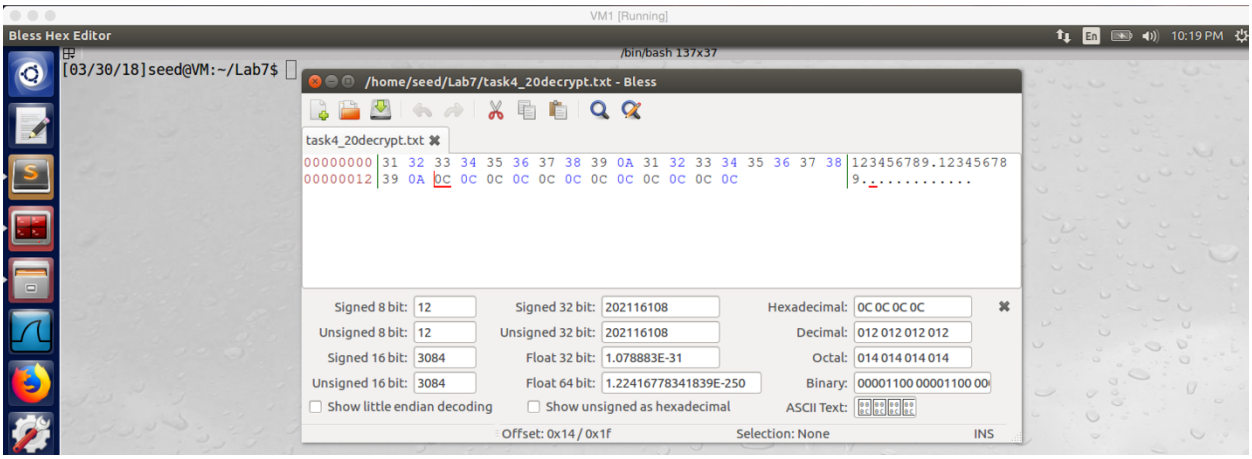


screenshot1. Two file, one 20 bytes, one 32 bytes

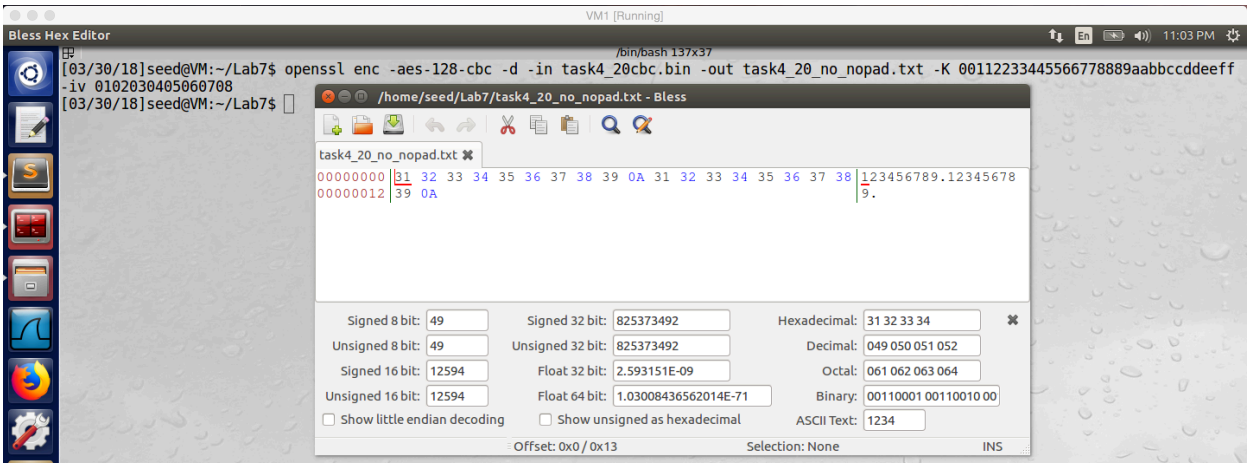


screenshot2. After encrypt task4_20.txt, the size of encrypted file task4_20cbc.bin is 32 bytes. So 12 bytes padding is added to the encrypted file.

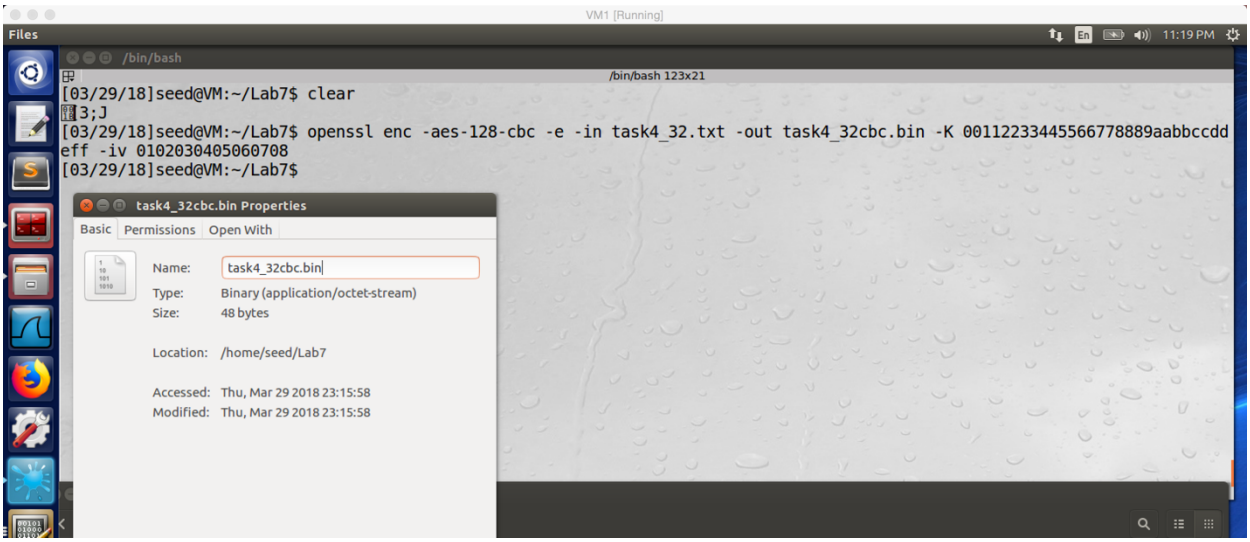




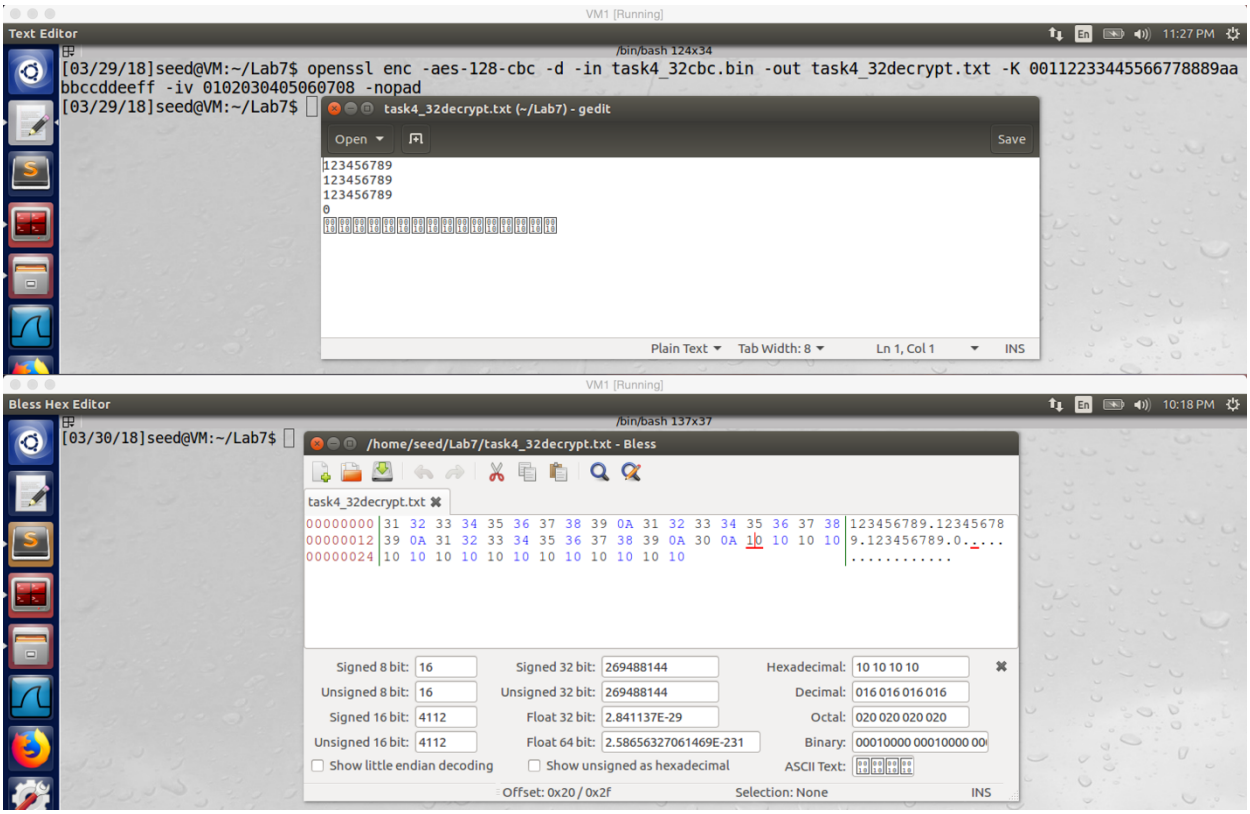
screenshot3. After we decrypted the encrypted file with –nopad, we can see 12 bytes padding are added



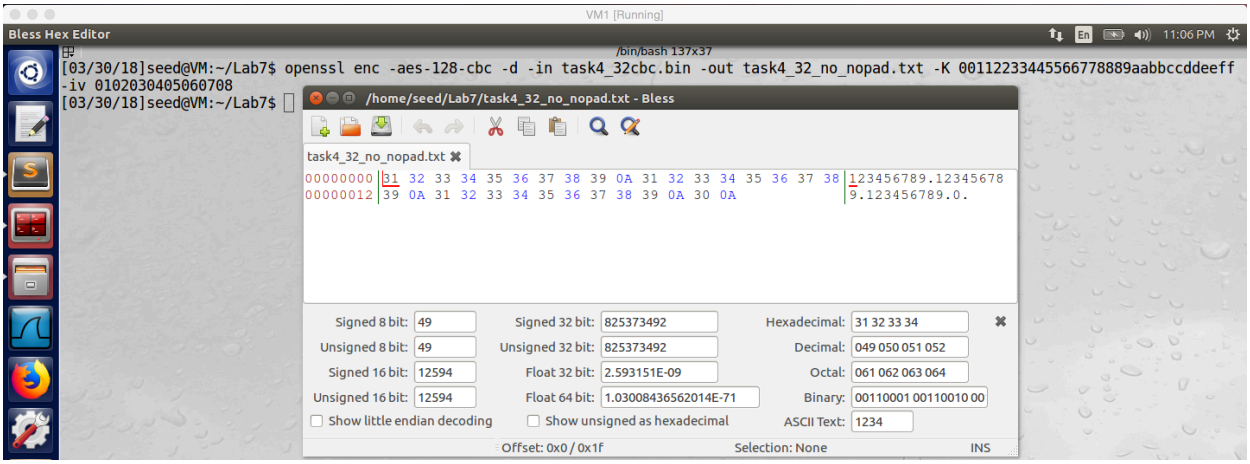
screenshot4. Decrypting without –nopad, there is no padding in the decrypted file



screesshot5. We also encrypted the task3_30.txt (32bytes), the encrypted file is 16 bytes bigger than the original file. So 16 bytes padding are added



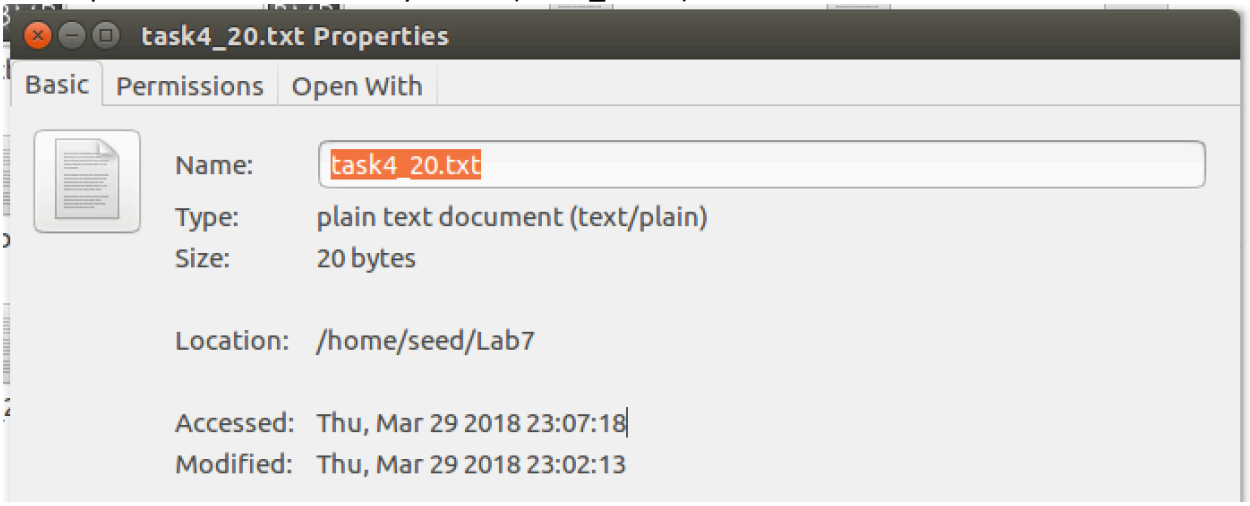
Screenshot6. After we decrypted the encrypted file with –nopad, we can see 16 bytes padding are added



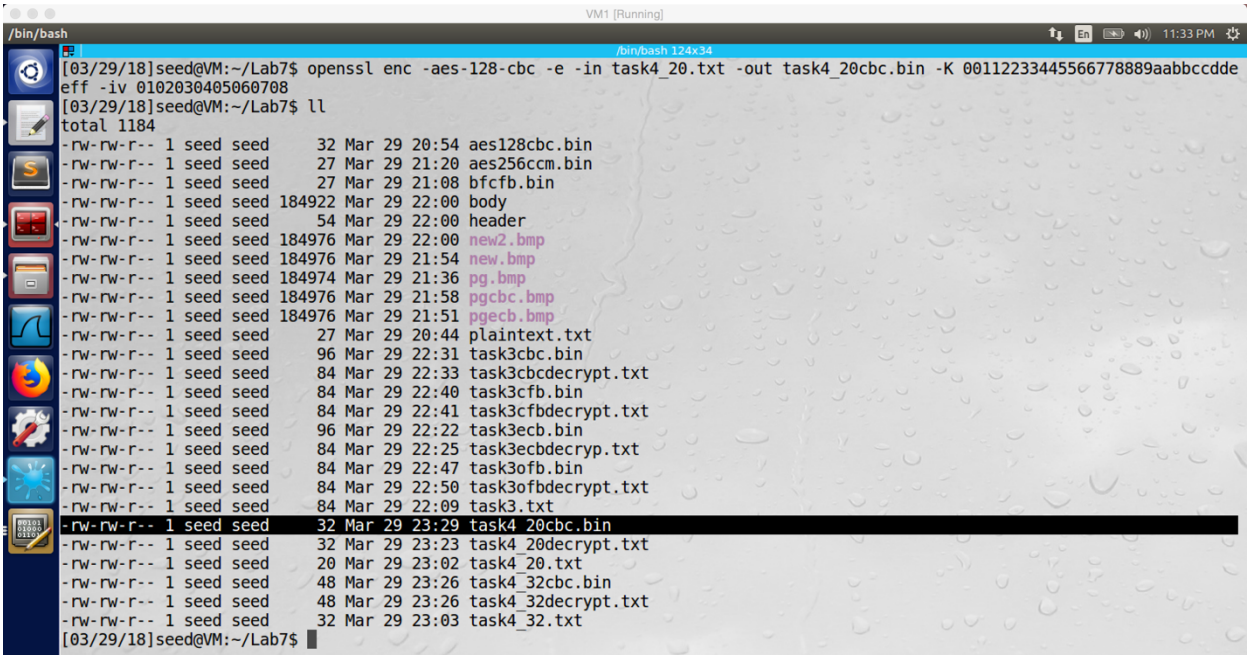
screenshot7. Decrypting without –nopad, there is no padding in the decrypted file

Part2:

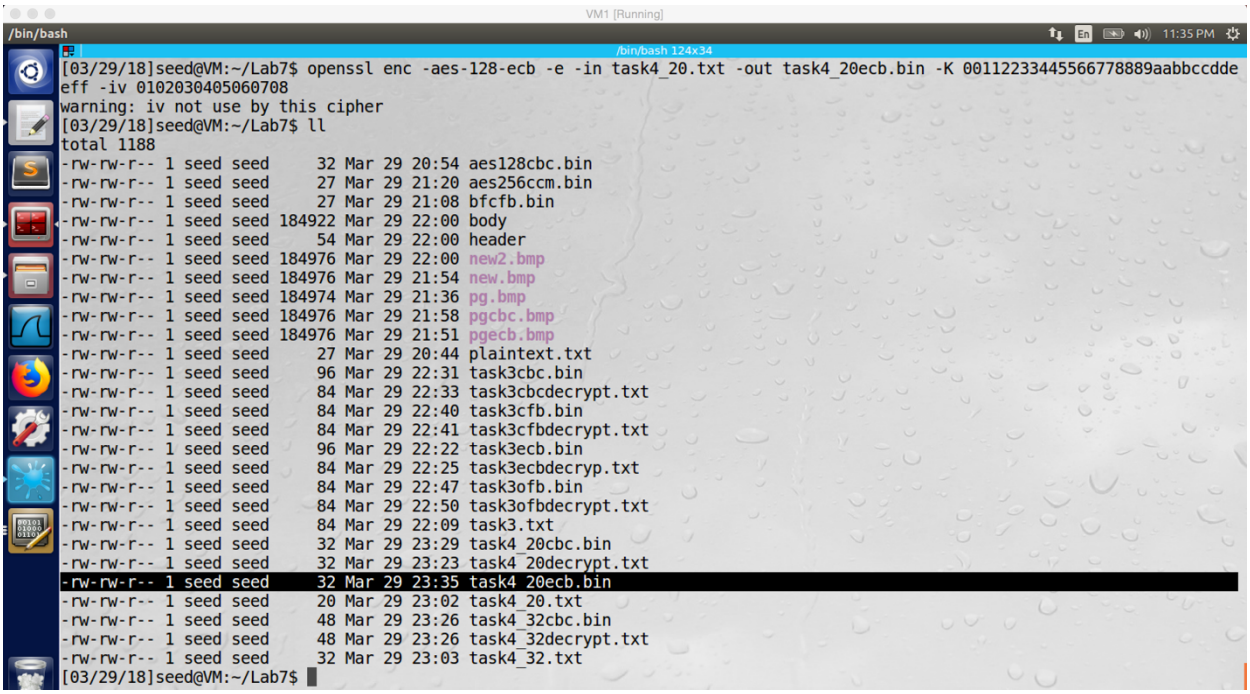
In this part we still use the 20bytes file (task4_20.txt) to do the task.



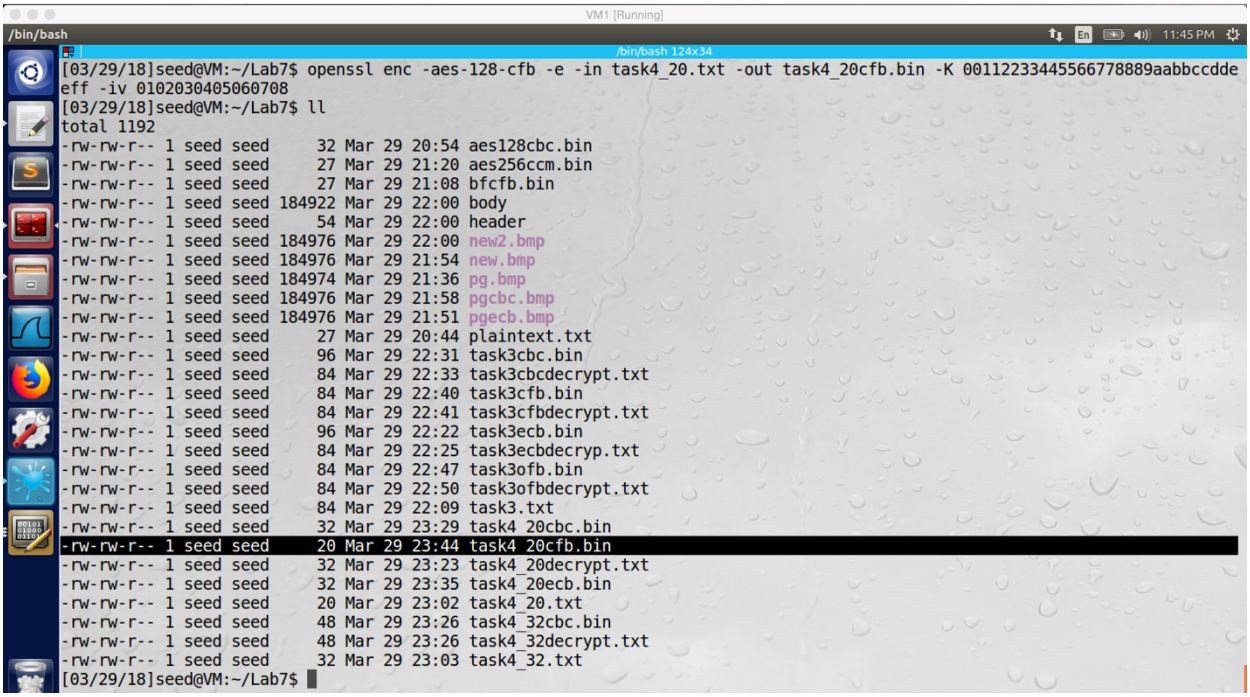
screenshot1. The20 byte file: task_20.txt



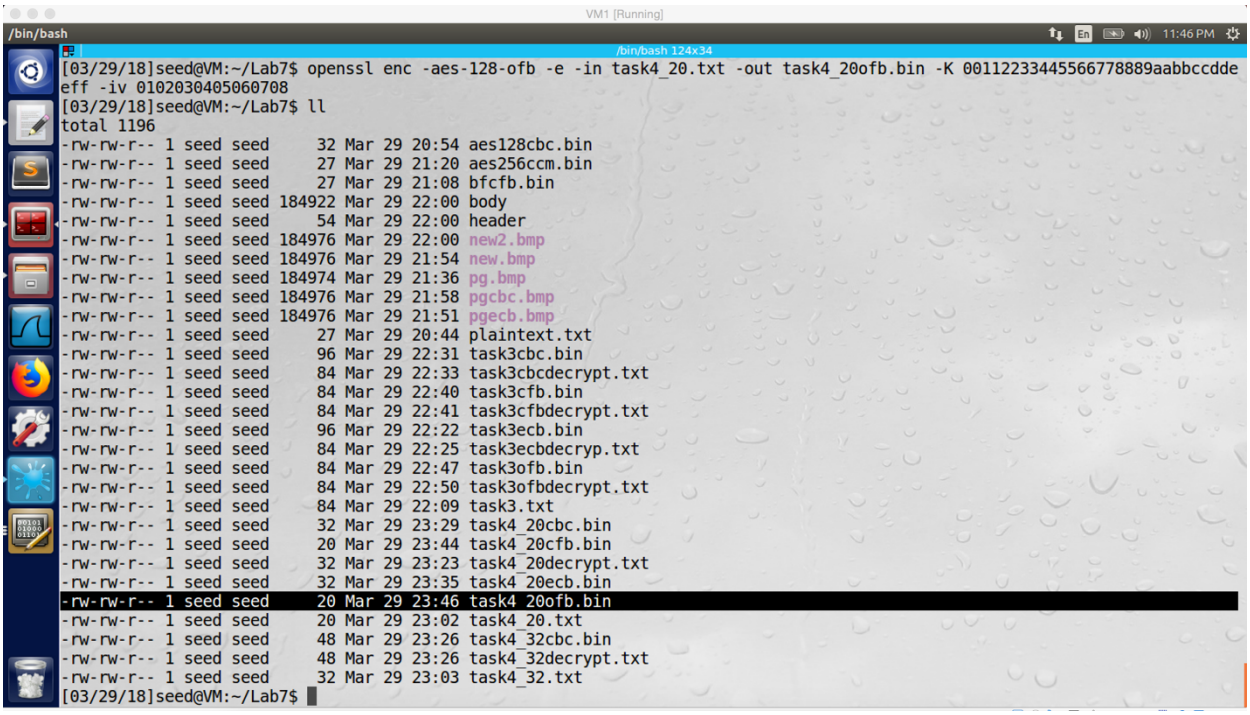
screenshot2. After encrypt the 20byte file with CBC, the encrypted file size becomes 32byte, so 12byte padding is added to the encrypted file



screenshot3. After encrypt the 20byte file with ECB, the encrypted file size becomes 32byte, so 12byte padding is added to the encrypted file



screenshot4. After encrypt the 20byte file with CFB, the encrypted file size is still 20byte, so no padding is added to the encrypted file



screenshot5. After encrypt the 20byte file with OFB, the encrypted file size is still 20byte, so no padding is added to the encrypted file

Observation and Explanation:

This task has two parts. In part one, we need to figure out padding in AES encryption. For this part, we firstly create two files, one 20 bytes, and one 32 bytes (Part1, screenshot1). We encrypt the 20bytes file first. After we encrypted it, we see the size of encrypted file is 32bytes, so 12 bytes padding are added to the encrypted file (screenshot2). The reason is because we use aes128 CBC, it requires padding and each block in the encrypted file should be 16bytes. However, for the 20bytes file, the second block is only 4bytes; therefore, 12bytes padding are added to the encrypted file to form a 16bytes block. To give more evidence, we use –nopad to decrypt the file, it will keep padding in the decrypted file. As the screenshot3 shows, there are 12bytes padding are added in the end of the file. Of course, if we do not use –nopad to decrypt the file, the decrypted file size should be 20bytes (screenshot4).

For the 32 bytes file we did same thing, we encrypted it and the encrypted file size becomes 48bytes (screenshot5). In fact, aes128 CBC needs to add padding in the end of the encrypted file; but in the 32bytes file, there are already two blocks, and all of them has full block size 16bytes. So aes128 CBC add padding in the whole size of a block, which is 16 bytes. Therefore, the encrypted file size becomes 48bytes. We can also use –nopad to decrypt the file, and we can see 16bytes padding are added (screenshot6). So the above experiments verified that openssl uses PKCS5 standard for padding.

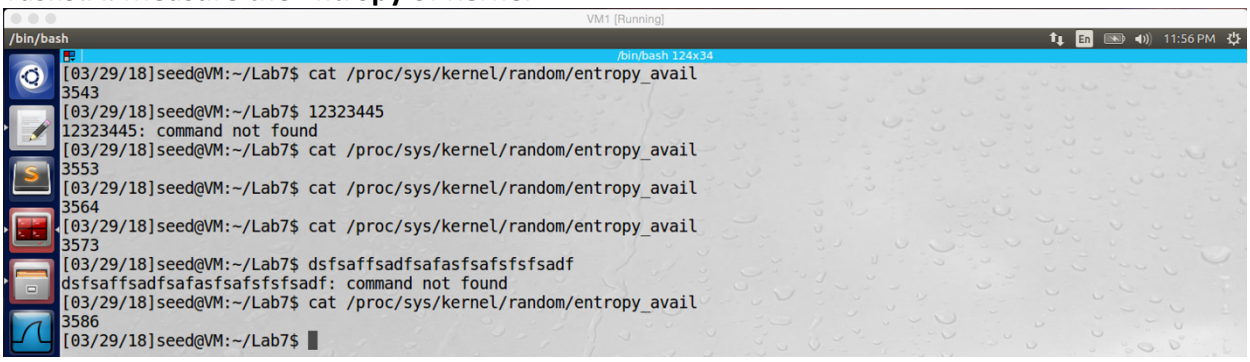
For part 2, I still use the 20bytes file which we created in part one to do this task (screenshot1). I will encrypt the 20bytes file with 4 different cipher mode (ECB, CBC, CFB, OFB) to see which one requires padding, which do not. For CBC, after I use it to encrypt a file, the

encrypted file size is 32bytes (screenshot2), 12bytes padding are added. For ECB, the encrypted file size is 32bytes as well (screenshot3), 12 bytes padding are added. For CFB, the encrypted file size is still 20bytes (screenshot4), no padding is added. For OFB, the encrypted file size is 20bytes (screenshot5), no padding is added.

So we can see CFB and OFB do not require padding. The reason is that they are stream ciphers, so their block size usually is not fixed. Therefore, they do not need to add padding in encrypted file.

Task5: Pseudo Random Number Generation

Task5.A: Measure the Entropy of Kernel

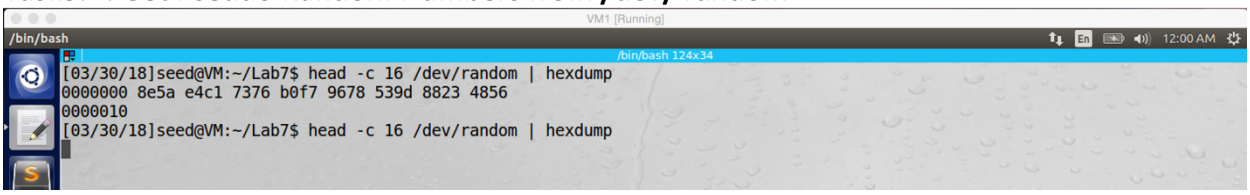


screenshot1. After I moving mouse or typing something, the value become greater.

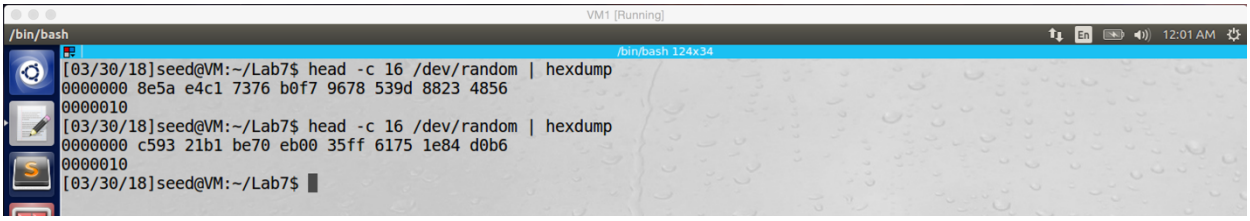
Observation and Explanation:

As I move my mouse, or type something on the keyboard, or do other operation in the VM, the value becomes bigger. In other words, as I operate on the computer, the randomness which measured by entropy is increased.

Task5.B: Get Pseudo Random Numbers from /dev/ random



Screenshot1. After running the program several times, the program cannot print anything

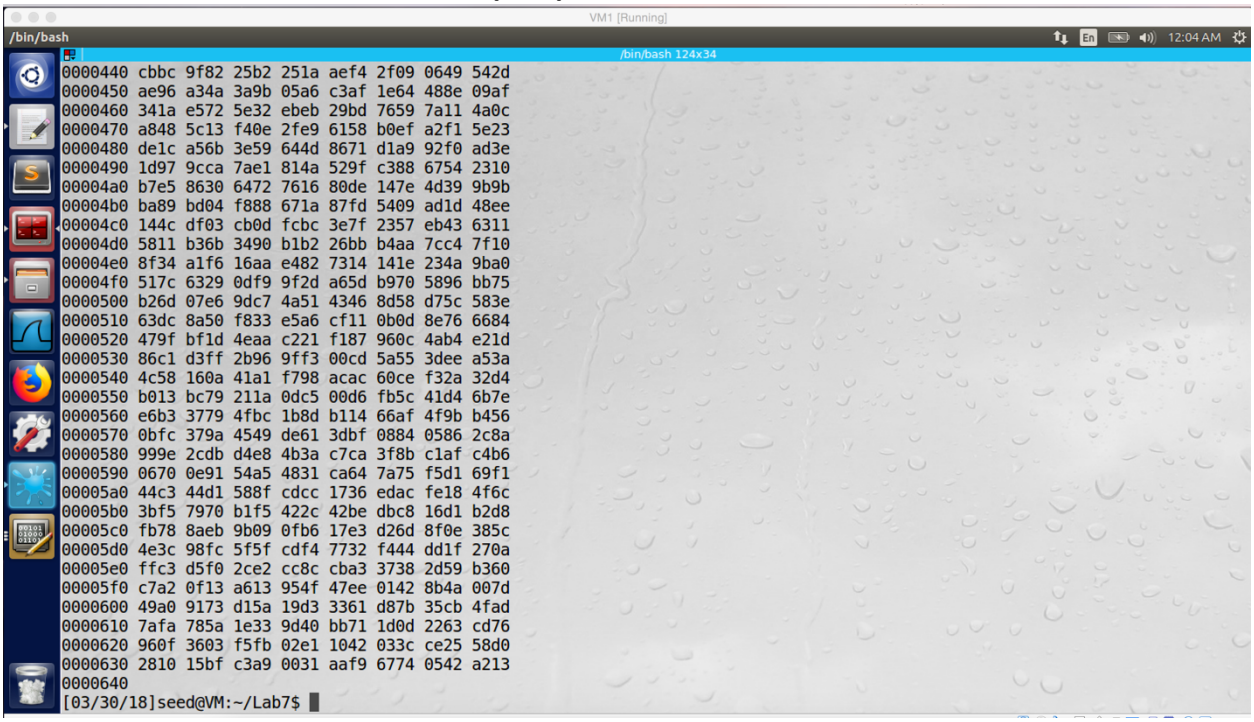


screenshot2. After the program stacked, I just need to move my mouse, and then the program can print something again

Observation and Explanation:

After I running several times of the program, the program does not print anything, because its randomness pool is exhausted (screenshot1). After I move my mouse several times, the program can get some data to add into its randomness pool. So it prints some random value again (screenshot2).

Task5.C: Get Random Numbers from /dev/urandom



screenshot1. I run the program more than twenty times, but it does not stack

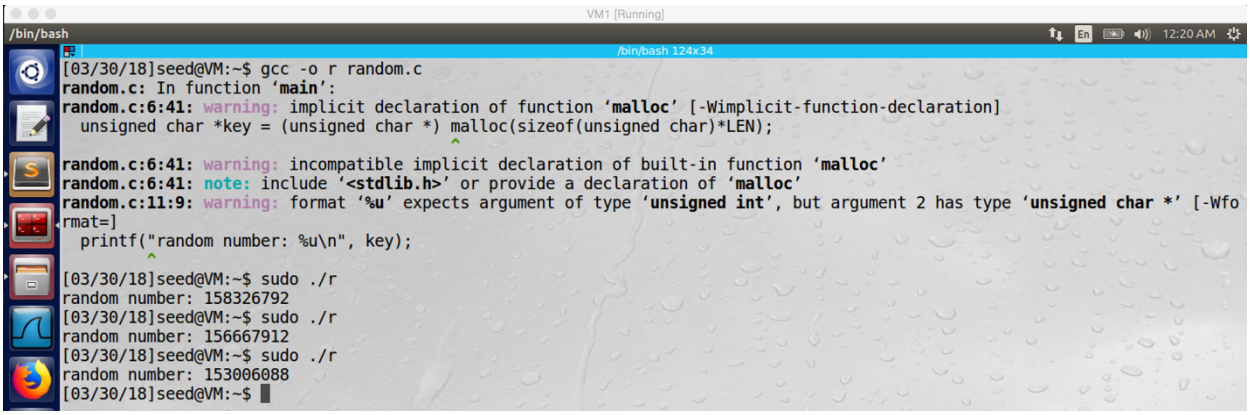
I compile and run the random number generation program successfully, the following is the program. The program is copped from lab description; I just add one line to print out the random number we got from /dev/urandom file.

```
#include <stdio.h>

#define LEN 16 // 128 bits

int main(){
        unsigned char *key = (unsigned char *) malloc(sizeof(unsigned char)*LEN);
        FILE* random = fopen("/dev/urandom", "r");
        fread(key, sizeof(unsigned char)*LEN, 1, random);
        fclose(random);

        printf("random number: %u\n", key); //print the random number
}
```

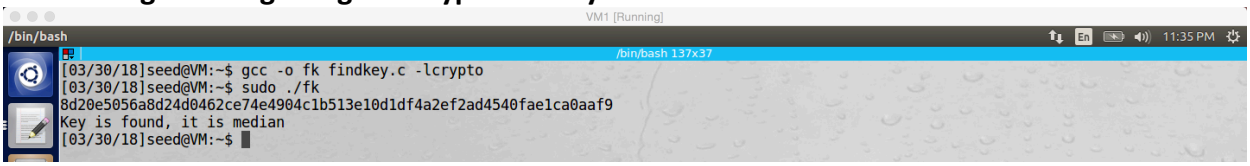


screenshot2. After I run the program, the random number is printed

Observation and Explanation:

In this task, I first observer the result printed by "head –c 1600 /dev/urandom | hexdump", the program keeps print random value and never stop (screenshot1). And then I also compile and run the program in the lab description, the program gets random numbere form /dev/urandom successfully (screenshot2).

Task6: Programming using the Crypto Library



screenshot1. After I run my findkey.c program, it gets the corresponding key from the English word list for the given cipher text, plaintext, and IV.

Observation and Explanation:

In this task, we need to write a program. For a given cipher text (encrypted by aes128 CBC), plaintext, and IV, the program can find the key, and this key is an English word. For my findkey.c program, it will read every word in the English word list, and then it will encrypt these words by using aes128 CBC. If the cipher text generated by my program is equal to the given cipher text, then we find the key which is the current English word. After my program find the key, it will print the key with the cipher text generated by my program.

For the given cipher text in the lab description, after I run my program, I found out the answer key is **median**.

The following is the findkey.c code:

```
#include <openssl/evp.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>

//given plaintext
char plaintext[] = "This is a top secret.";
```

```
//given ciphertext
char givenCipher[] = "8d20e5056a8d24d0462ce74e4904c1b513e10d1df4a2ef2ad4540fae1ca0aaf9";

unsigned char iv[16] = {0}; //as lab description, IV are all 0
unsigned char key[16]; //key size is 16

//this function is used to fill up a key word, if it size is less than 16 bytes
void fillUp(char *key){
		int length = strlen(key);
		while(length < 16){
			key[length] = ' ';
			length++;
		}
		//add '\0' to end the key words
		key[length] = '\0';
}

//this function is used to compare the encrypted code we get by runing the program with
//the given cipher code; if they are same, then we find the key; otherwise, we keep running the program
void valueCompare(unsigned char *buf, char*s, int len, FILE *out, FILE *in){

		//I try a lot of methods, finally I choose this one.
		//after I got the encrypted code, I try to put it in char array
		//but I cannot get correct hex value. So I try to write it in a file, and this works
		//Therefore, after I get an encrypted code, I write it into a file, and then
		//I just read the file again, and put the hex value in a char array.
		// Then we can compare it with the given ciphertext
		//if they are equal, then we find the key

		//write the encrypted code into a file
		out = fopen("testResult.txt", "wb");
		for(int i=0;i<len;i++){
			fprintf(out, "%02x", buf[i]);
		}
		fclose(out);

		//read the file, and put the hex value into a char array
		in = fopen("testResult.txt", "r");
		unsigned char pw[65];
		//if they are equal, then we find the key
		if(fgets(pw, 65, in)){
			if(strcmp(pw, givenCipher)==0){
				for(int i=0; i<strlen(pw);i++)
					printf("%c", pw[i]);
				printf("\n");
				printf("Key is found, it is %s \n", key);
			}
		}
		fclose(in);
}

int main(){
```

```
		int inlen, outlen, tmplen;
		unsigned char outbuf[1024];

		FILE *in1, *in2, *out;
		in1 = fopen("wordList.txt", "r");

		EVP_CIPHER_CTX ctx;
		EVP_CIPHER_CTX_init(&ctx);

		//read word from the word list one by one
		while(fgets(key, 16, in1)){

				//text editor will add null to the end, so we should remove it
				key[strlen(key) - 1] = '\0';
				//fillup words to be 16 bytes
				if(strlen(key)<16) fillUp(key);

				//I follow an example in the
https://www.openssl.org/docs/man1.0.2/crypto/EVP_EncryptInit.html to use the following encryption method
				//it is basic method to encryt plaintext with given key and iv
	EVP_EncryptInit_ex(&ctx, EVP_aes_128_cbc(), NULL, key, iv);

	if(!EVP_EncryptUpdate(&ctx, outbuf, &outlen, plaintext, strlen(plaintext)))
	{
	   /* Error */
	   EVP_CIPHER_CTX_cleanup(&ctx);
	   return 0;
	}
	/* Buffer passed to EVP_EncryptFinal() must be after data just
	 * encrypted to avoid overwriting it.
	 */
	if(!EVP_EncryptFinal_ex(&ctx, outbuf + outlen, &tmplen))
	{
	   /* Error */
	   EVP_CIPHER_CTX_cleanup(&ctx);
	   return 0;
	}
	outlen += tmplen;

	//compare the generated encypted code with given ciphertext
	valueCompare(outbuf, key, outlen, out, in2);

		}
		fclose(in1);

		return 0;
}
```