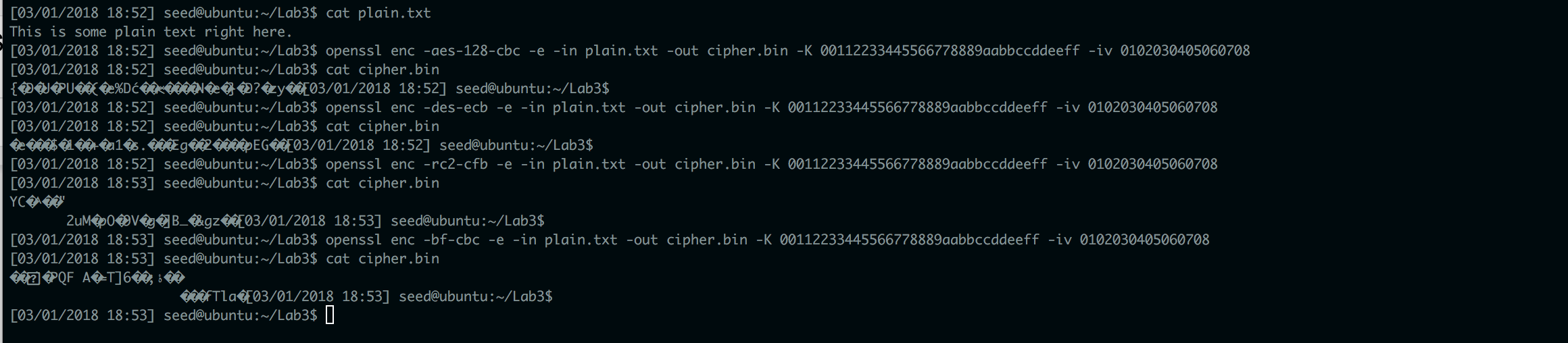
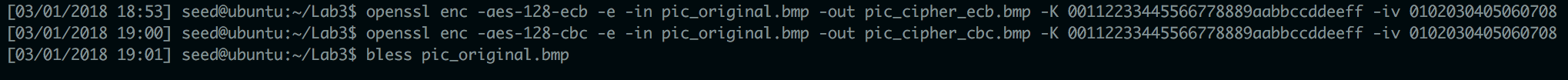
Homework #3

Crypto Lab - Secret Key Encryption

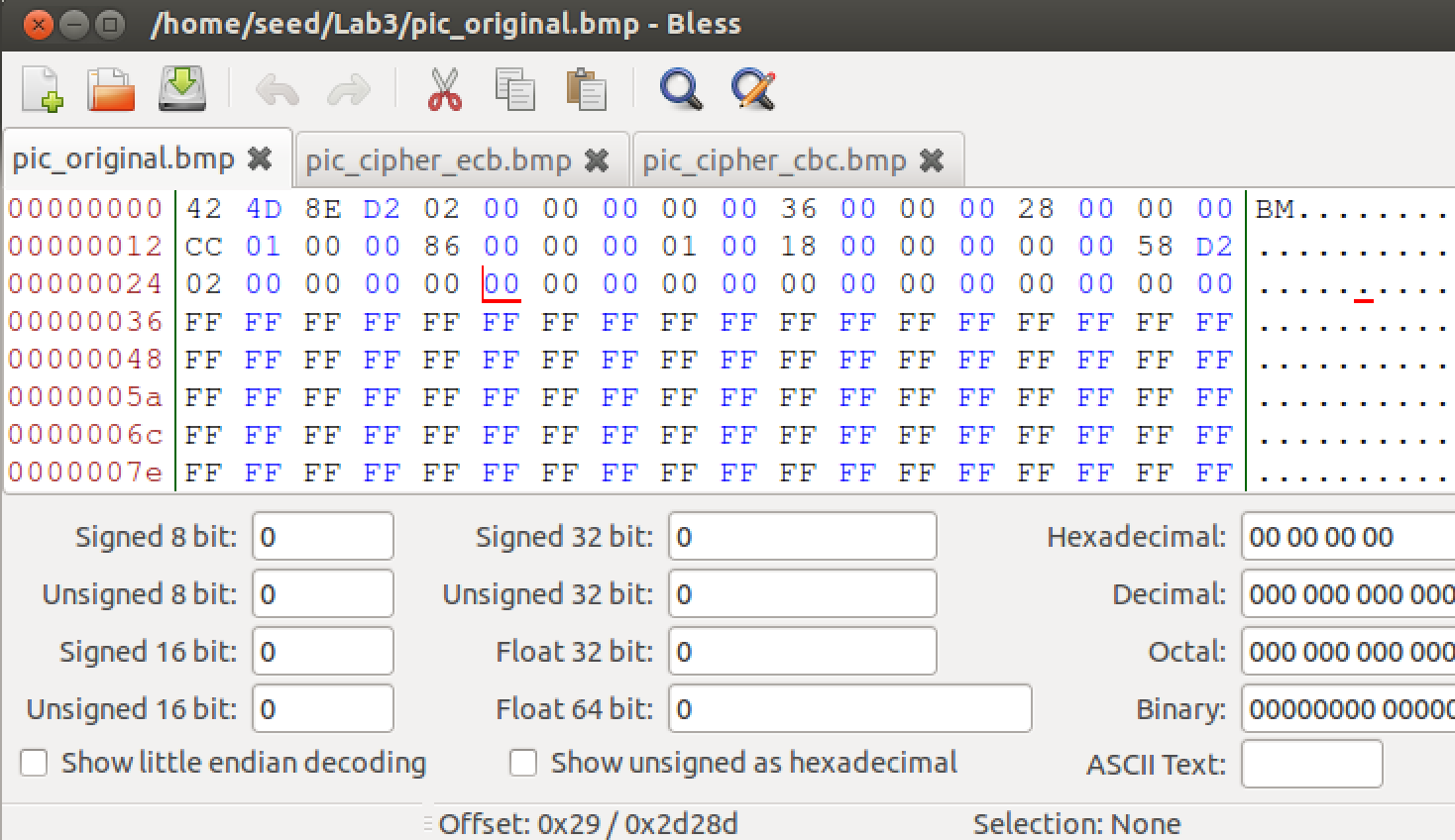
* Task 1:  
  Here is a screenshot of me running openssl to encrypt my plaintext file in various ciphers and modes:

  
 I am encrypting with:

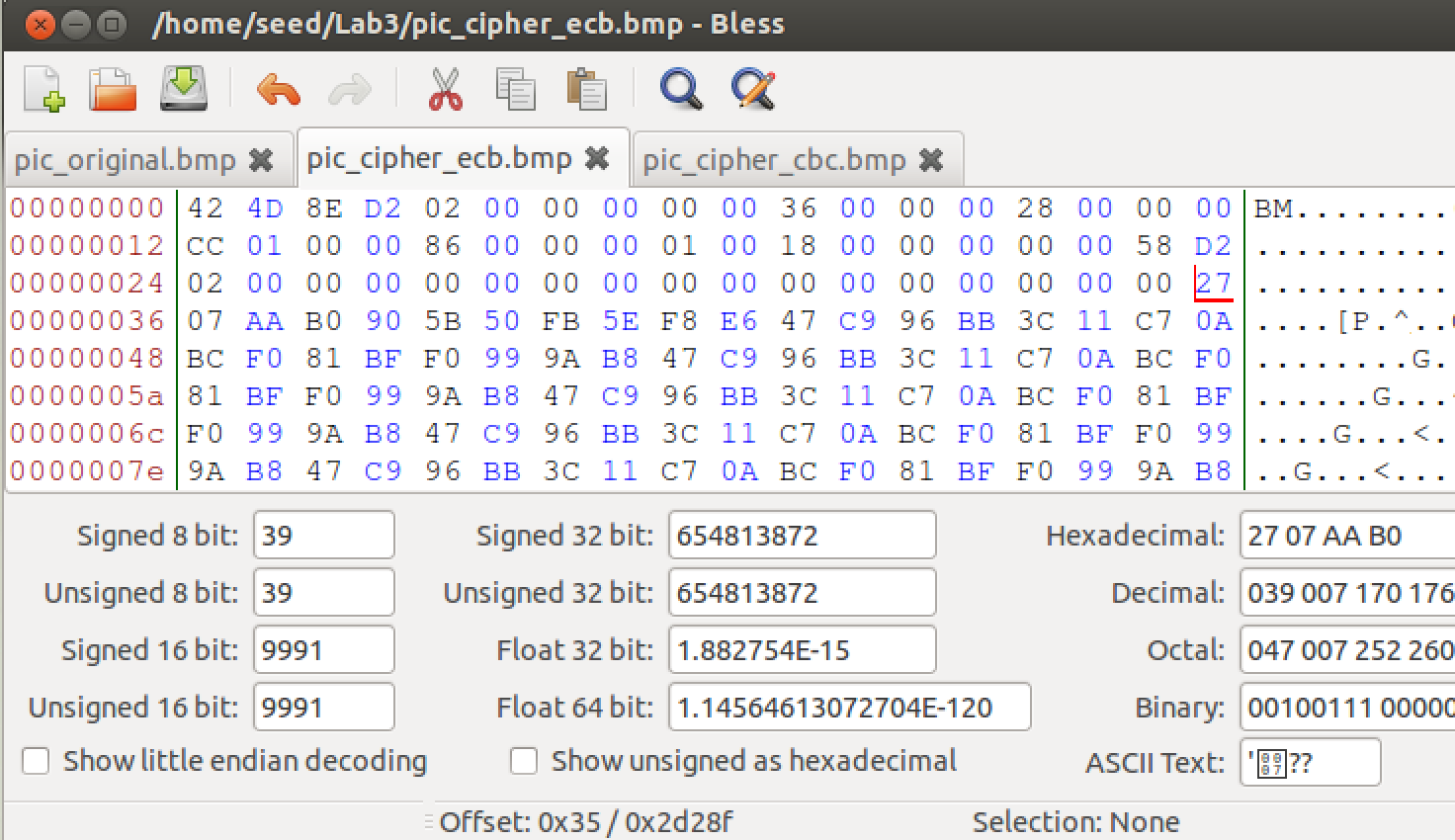
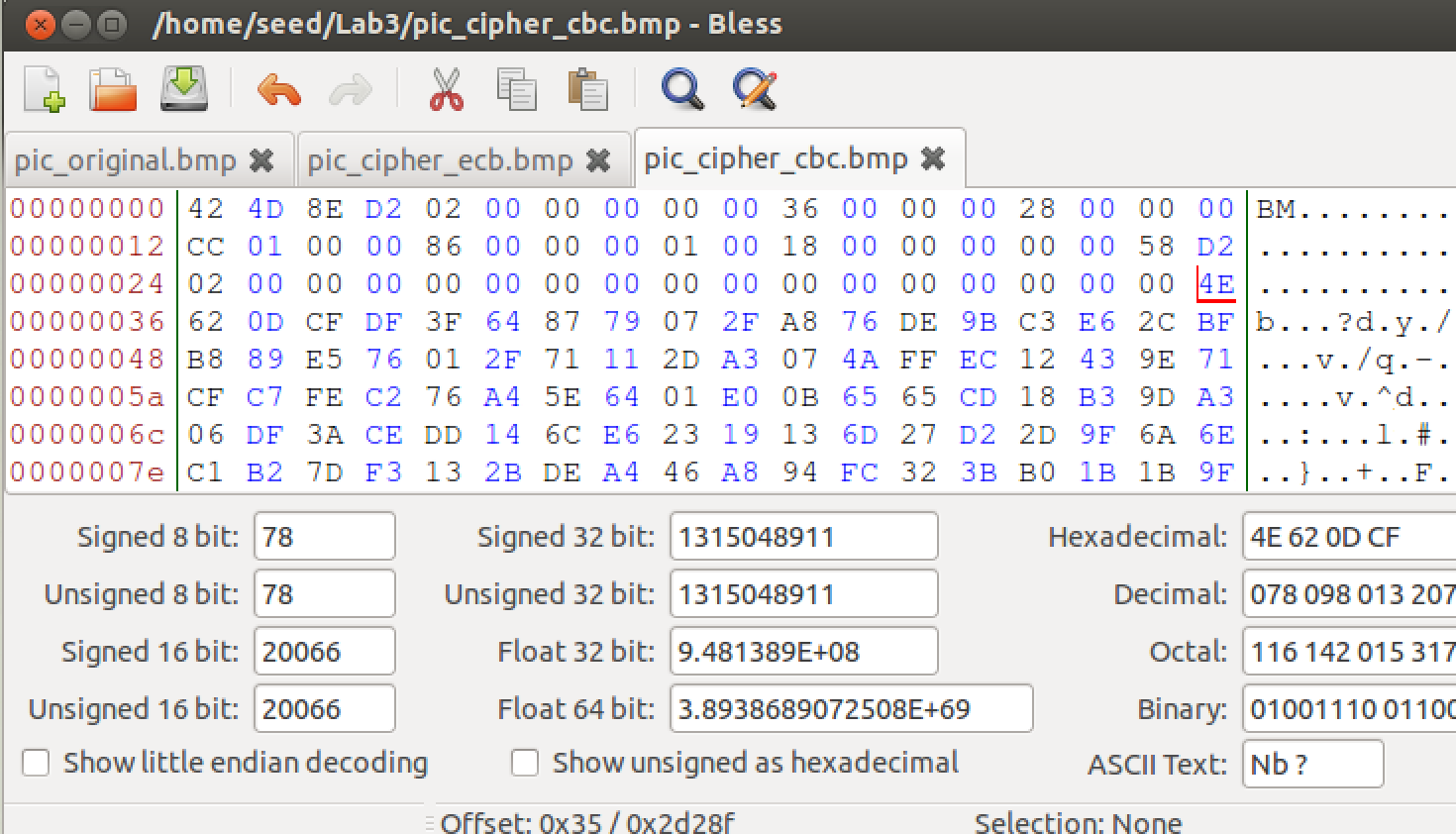
* + -aes-128-cbc (128-bit AES cipher in CBC mode)
  + -des-ecb (DES cipher in ECB mode)
  + -rc2-cfb (RC2 cipher in CFB mode)
  + -bf-cbc (Blowfish cipher in CBC mode)
* Task 2:
  + Part 2.1:  
    Here is a screenshot of me encrypting the pictures:



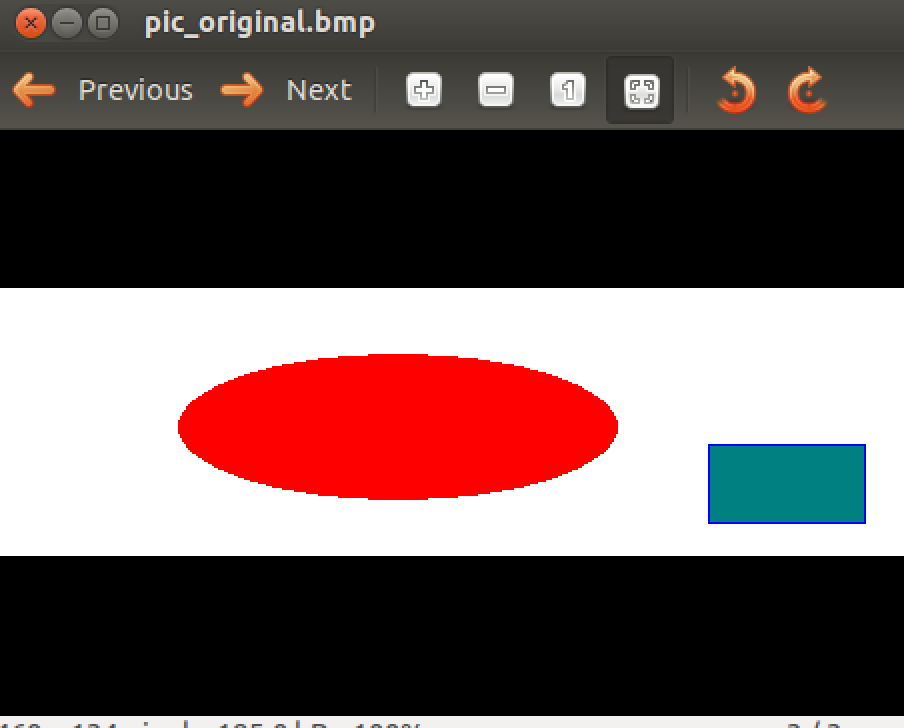
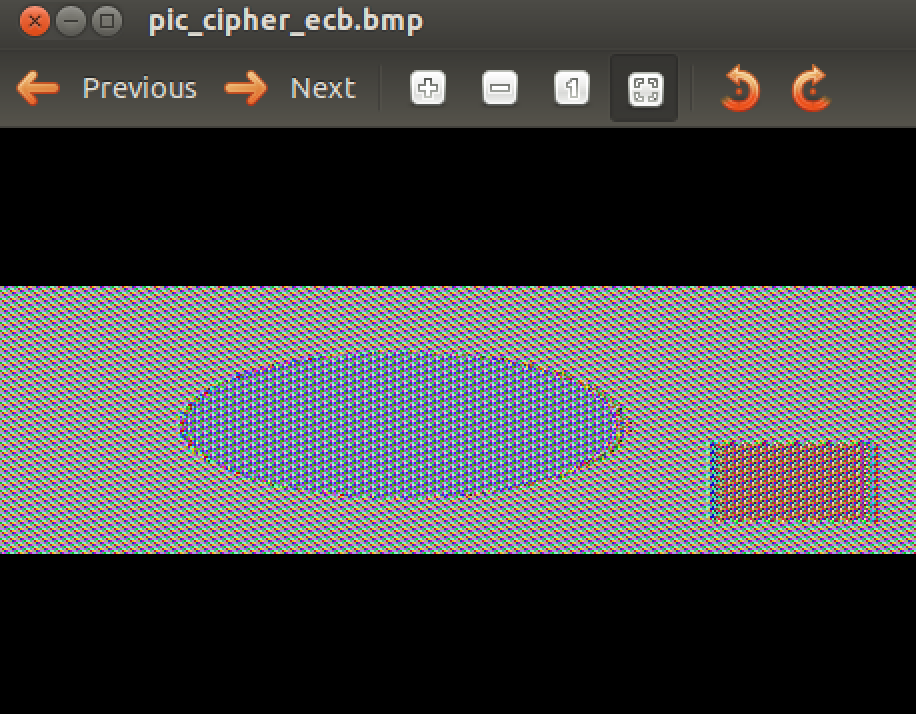
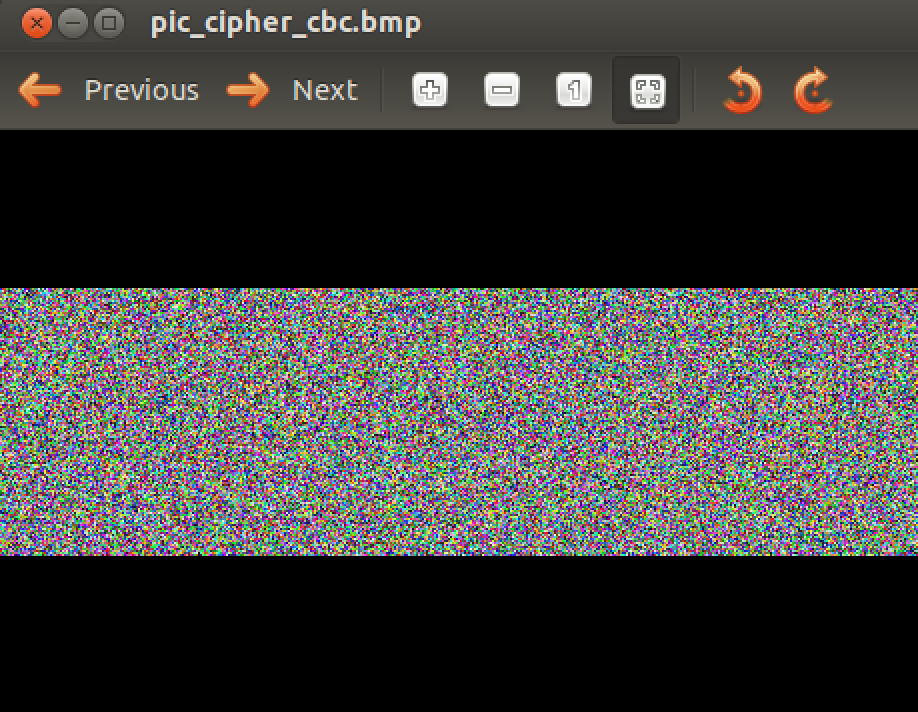
Here is a screenshot of each of the three pictures in the hex editor after I copied the first 54 hex characters from the original file and pasted them into the encrypted files.

Original Hex:  


ECB Hex:

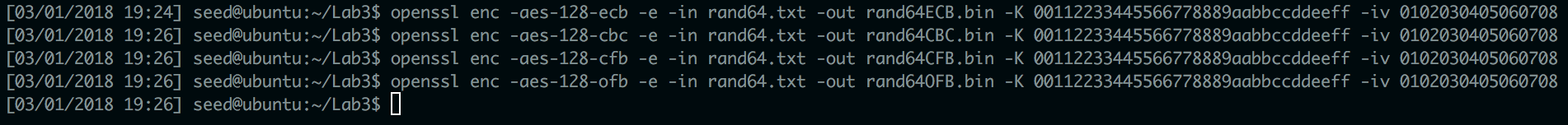
  
  
CBC Hex:  


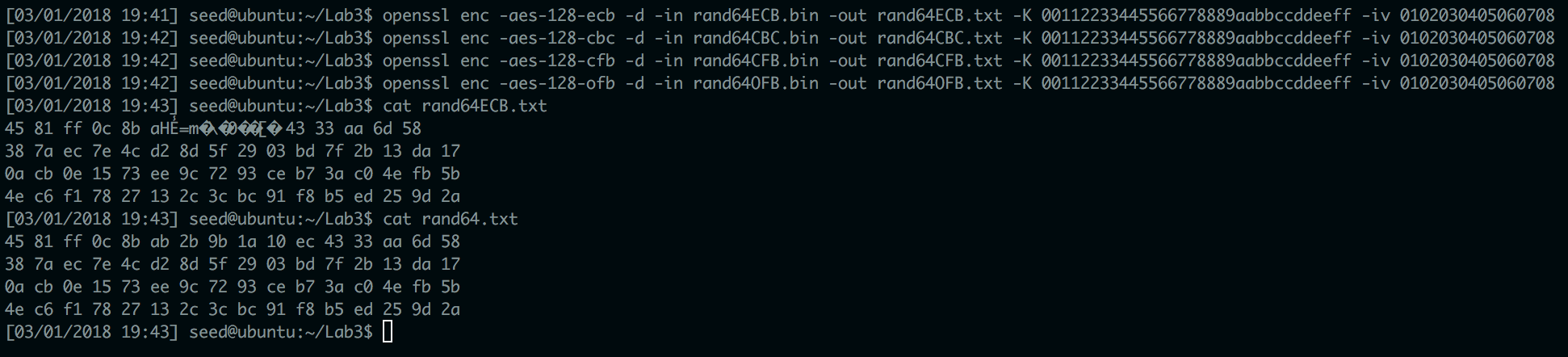
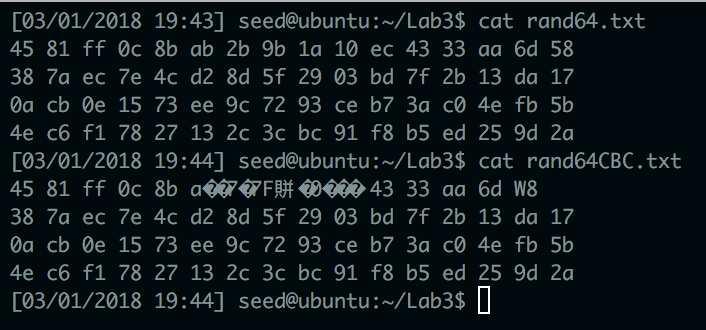
* + Part 2.1:

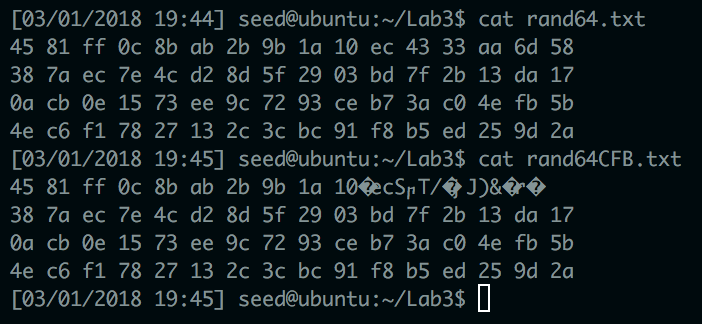
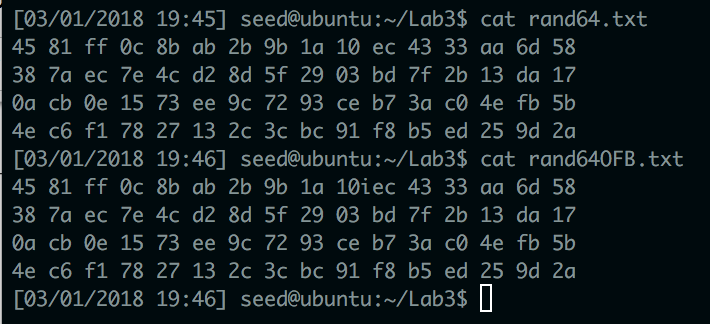
Original Picture:  
  
ECB Encrypted Picture:  
  
CBC Encrypted Picture:  


You cannot derive any useful information about the picture from the CBC encryption, but you can make out the basic form of the objects in the ECB encryption. This is probably because ECB uses a standalone encryption method for each block of plaintext, while CBC uses its ciphertext output from previous blocks as a feedback mechanism to generate a pseudo-random output on the proceeding blocks.

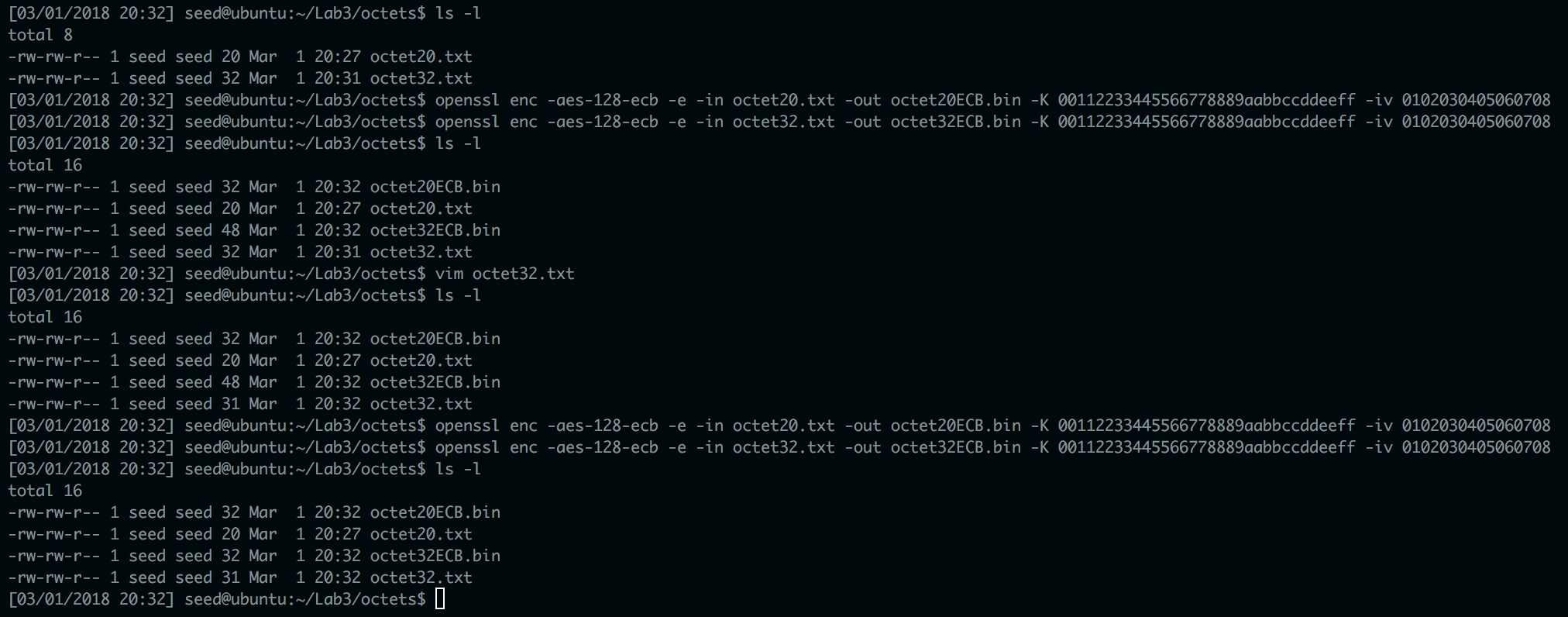
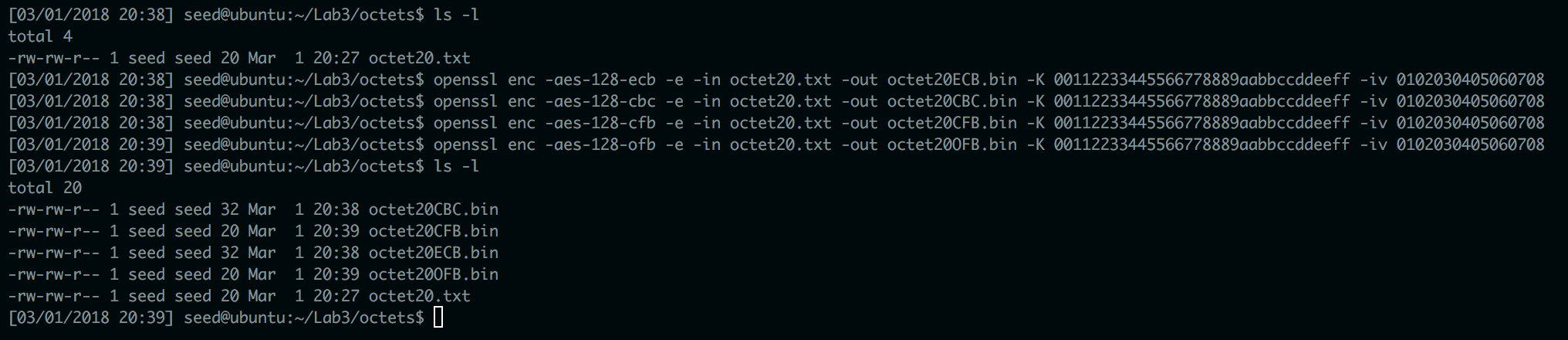
* Task 3:  
  1) I believe that OFB will preserve the most information from the corrupted file, since it was designed to operate over noisy channels. I believe that ECB will preserve the second most information, since its encryption algorithm works on standalone blocks; this means that the corruption will be kept only to a certain block. I believe that CFB will preserve the third most information, since it uses its cipher output as input to the next input stream of bytes, so the corruption should propagate a bit. Finally, I believe that CBC will preserve the least amount of data, because it uses entire blocks as feedback, so the corruption should propagate to the end of the file.  
    
  Here is a screenshot of me encrypting my random 64-byte file 4 different ways:

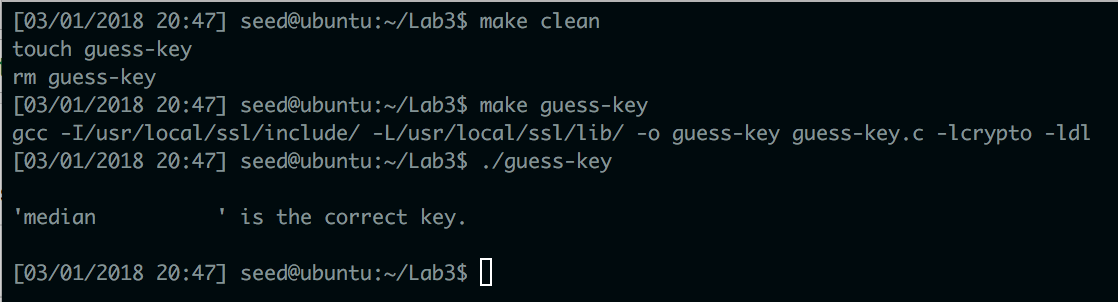
  
 Here is a screenshot of me decrypting the 4 different encrypted files after corrupting the  
 30th byte, and then comparing the ECB output to the original file:

  
 Here is a comparison of the CBC output to the original file:   


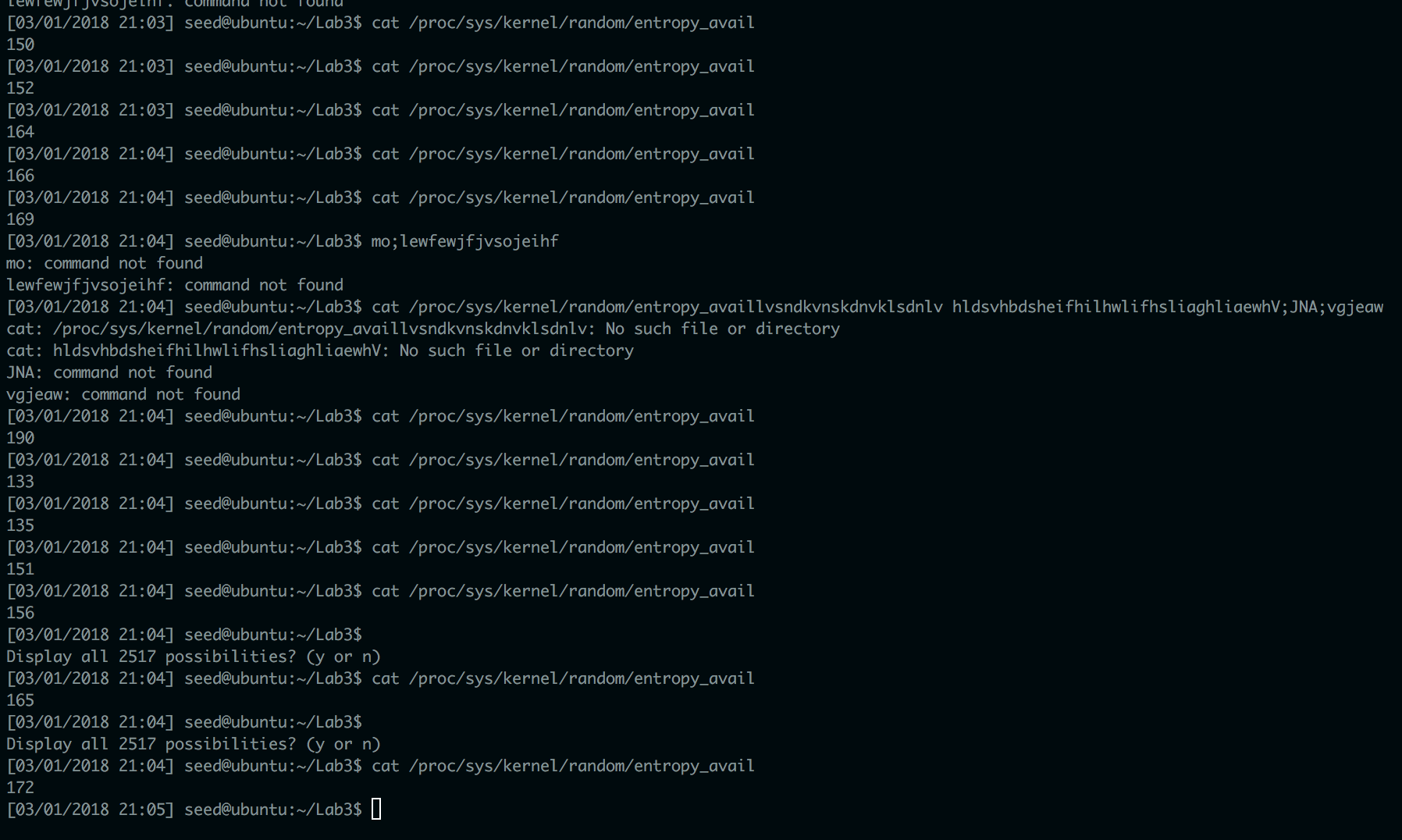
Here is a comparison of the CFB output to the original file:  
  
 Here is a comparison of the OFB output to the original file:  


2) I was correct in assuming that OFB would have the best preservation of the data, but after that, it was hard to tell which mode performed the best. CBC appeared to recover better than I thought it would. The same goes for CFB. This could be due to the fact that I had newline characters in my plaintext file. In any case, ECB did perform the second best, CBC performed third best, and CFB looked to have the worst preservation.  
3) The implication here is that OFB should be the mode that is selected when transmitting data over noisy channels. The other implication made here is that the likelihood of the encrypted data being corrupted should be considered before choosing an encryption mode.

* Task 4:  
  1) PKCS5 standard says that block cipher encryption modes require their input strings to be an exact multiple of their block size. For example, AES-128 uses block sizes that are 16 bytes long, so if it has an input that is not a length that is a multiple of 16 bytes, some zeroes must be added to the input to get it to a length that *is* a multiple of 16 bytes. Therefore, a good way to demonstrate the way that PKCS5 padding works is to encrypt a file of 20 octets and a file of 32 octets.  
  Here is a screenshot of the encryption of a 20 octet file and a 32 octet file, with the lengths of ciphertext being displayed:  
    
  The 20 octet file was padded to 32, while the 32 octet file was padded to 48.  
  This screenshot also shows a 31 octet file being encrypted, with it being padded to 32. For some reason, the encryption algorithm couldn’t allow the 32 octet file to be encrypted without padding, so it resulted in a 48-byte ciphertext.  
  2) Here is a screenshot of me running the 4 different encryption modes on a 20 octet file:  
    
  The ECB and CBC modes used padding and resulted in ciphertexts which were 32 bytes long, which makes sense because they encrypt block-by-block. The CFB and OFB modes did not use padding and resulted in ciphertexts which were 20 bytes long, which makes sense because they encrypt the input as a stream, so they do not need to make the input conform to a certain block size.
* Task 5:  
  The source code is included separately outside of this report in a file called *guess-key.c*. It was designed to load all of the keys from a separate file and try them one by one on the plaintext; the given ciphertext is loaded into main memory, and every output from an attempted key is compared byte-by-byte to the given ciphertext. If the output ciphertext matches the given ciphertext, then the correct key is returned and the program is terminated. Here is a screenshot of it being compiled and run:

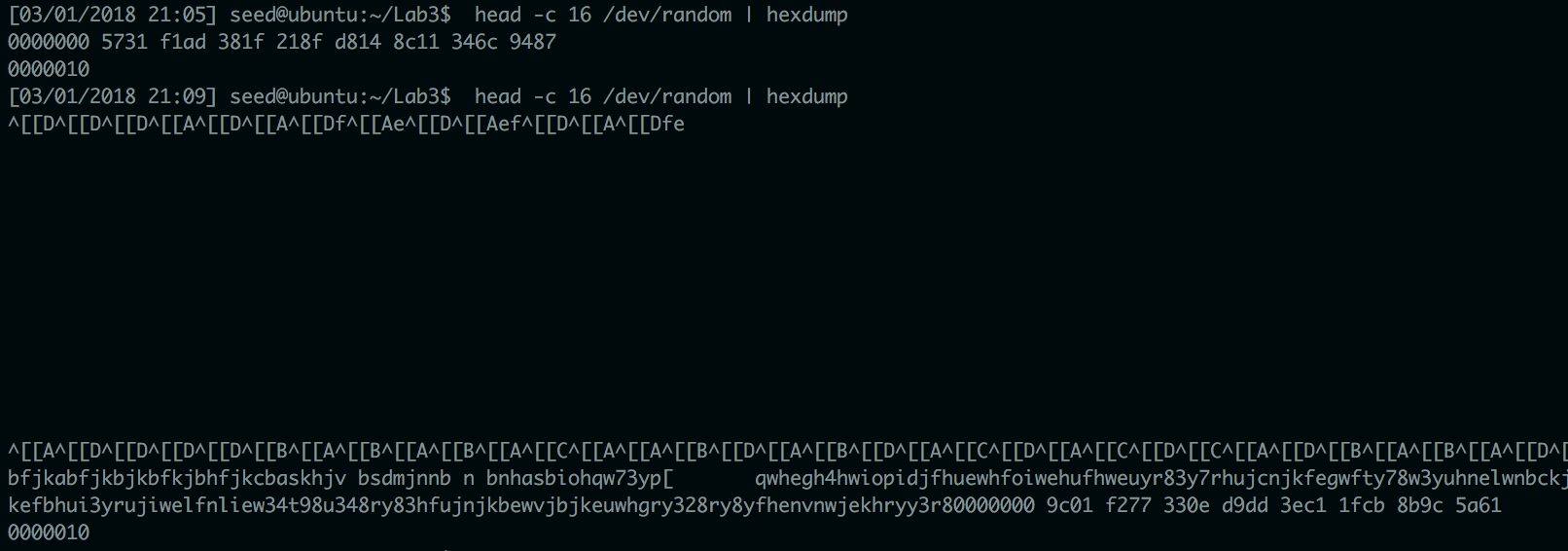


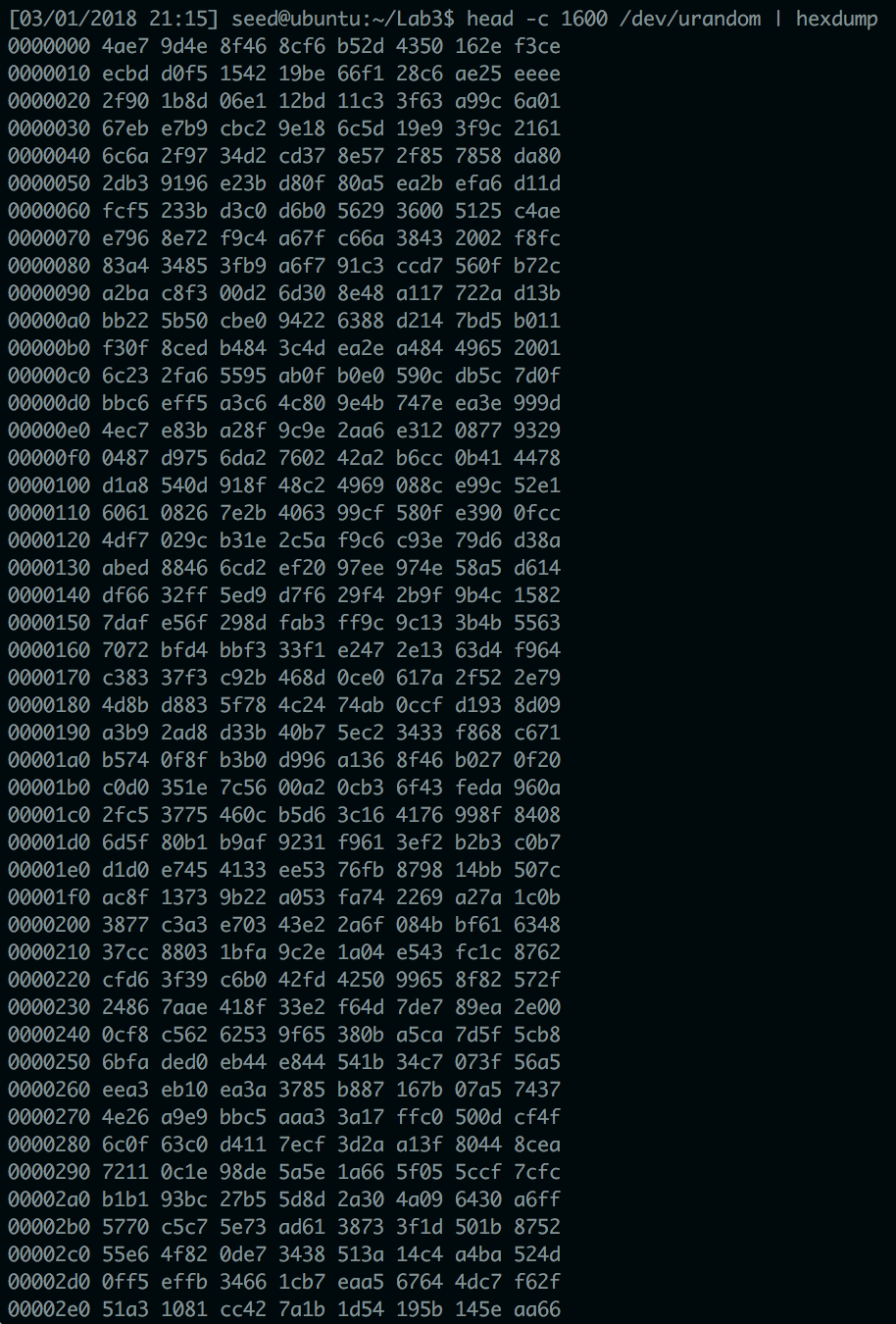
* Task 6:
  + Part 6.A:  
    Here is a screenshot of me attempting to create entropy:



I definitely tried as hard as I could to create entropy, and I noticed that the entropy detected did increase as a result of my banging on the keyboard and moving the mouse around, but it did not last for very long.

* + Task 6.B  
    Here is a screenshot of me using /dev/random:

  
I made it unblock by banging on the keyboard to generate entropy.

* + Task 6.C  
    Here is a screenshot of me running /dev/urandom:  
      
    I ran it at least 20 times, and I can verify that it did not block. The numbers certainly appeared to be random to me, and it ran a lot faster than /dev/random, so I can understand why /dev/urandom would be recommended if timing is a concern when designing a program.