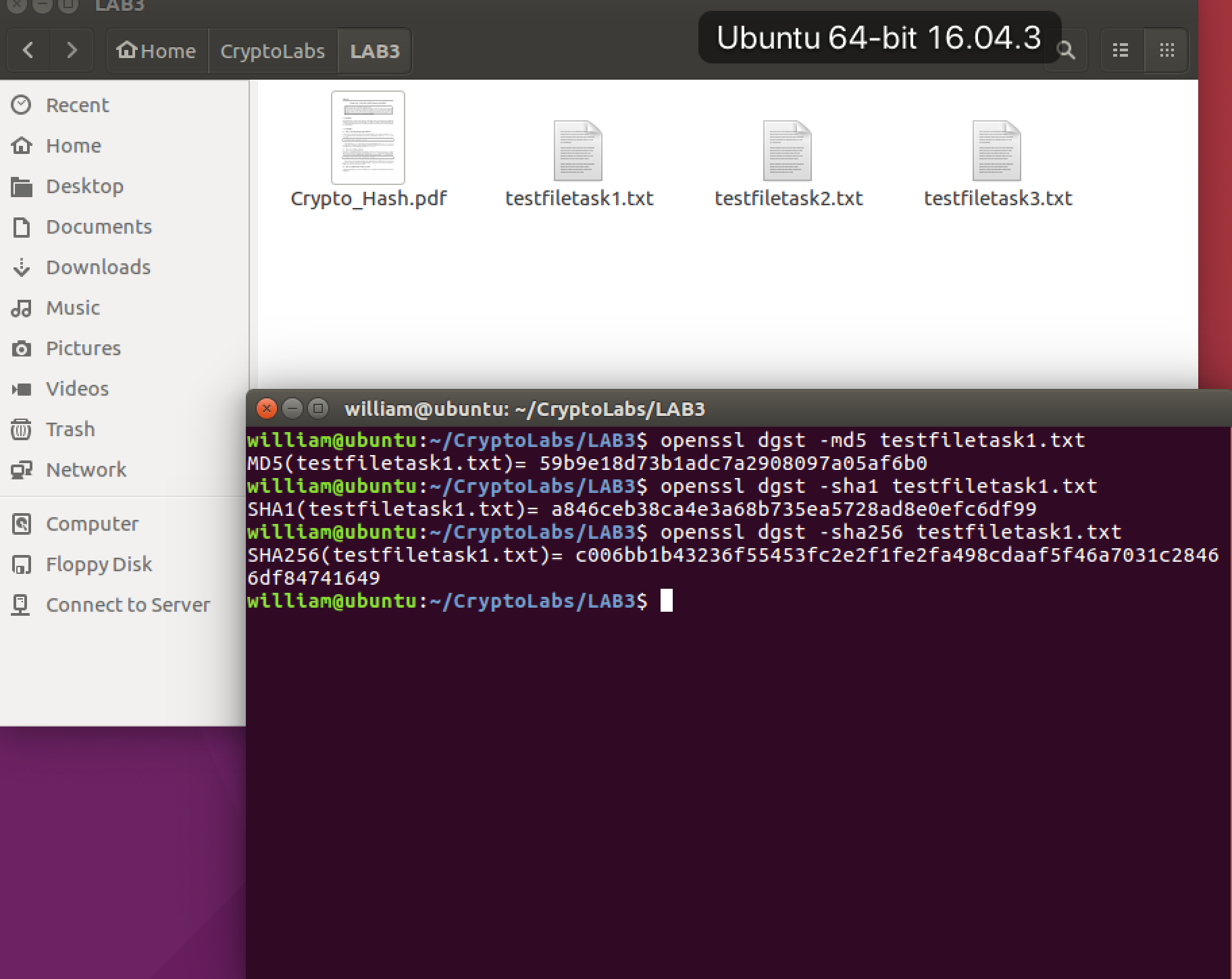
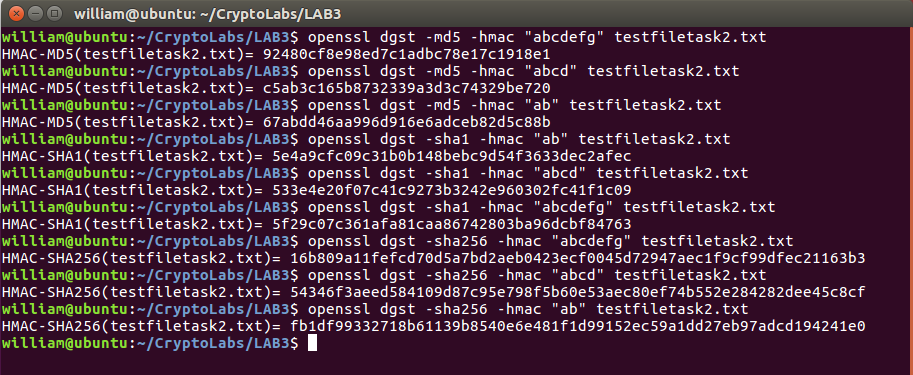
Crypto Lab – One-Way Hash Function and MAC

**Task 1**

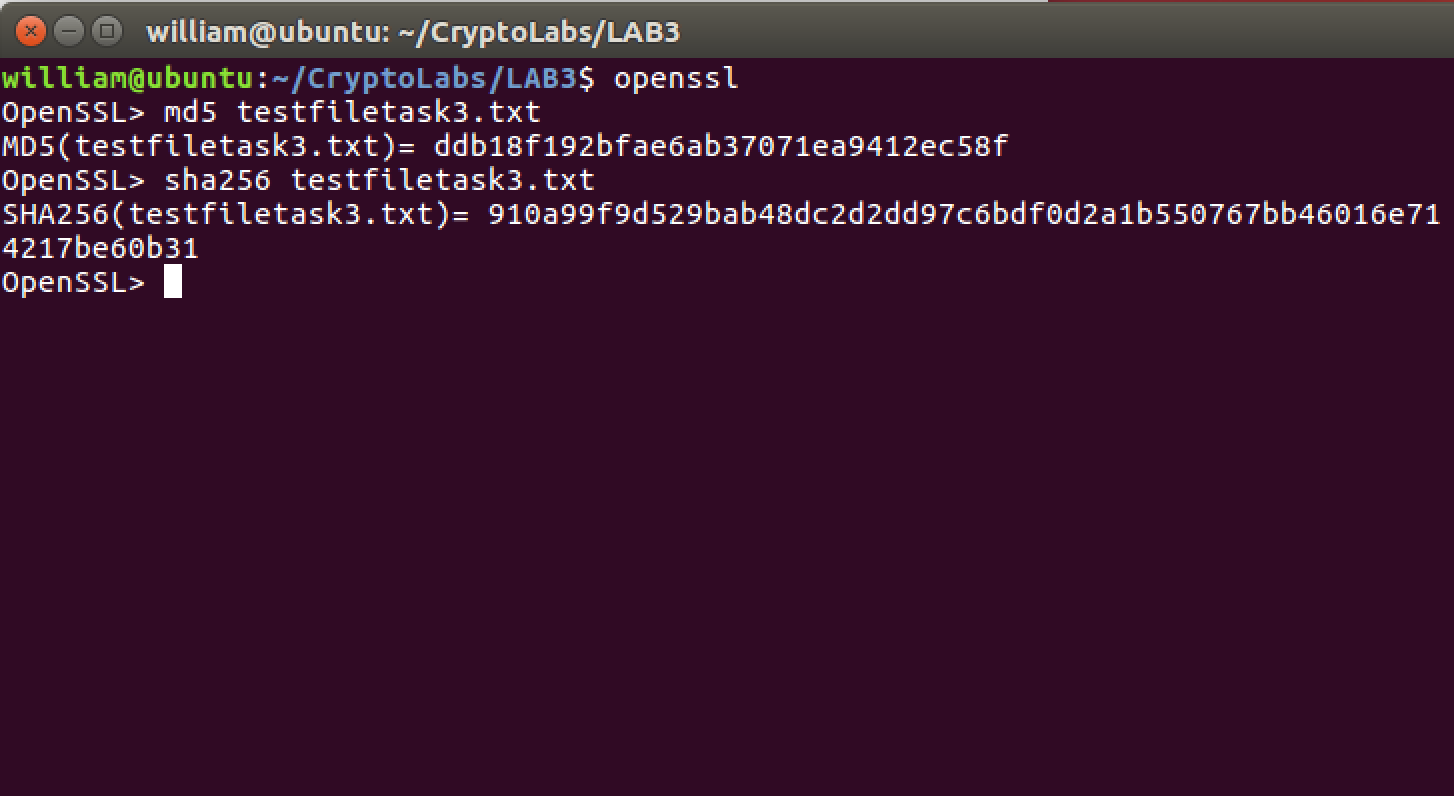
**Observation**: I used the given txt file and implemented 3 algorithms which are MD5, SHA1 and SHA256. The results as the above figure. From the results, I can see that the length of hash value which generated by MD5, SHA-1 and SHA-256 are 32, 40 and 64, correspondingly. It’s a proof that the hash value of MD5 is 128 bits, the hash value of SHA-1 is 160 bits and 256 bits for SHA-256. Also, it shows that SHA-256 is more secure than SHA-1.

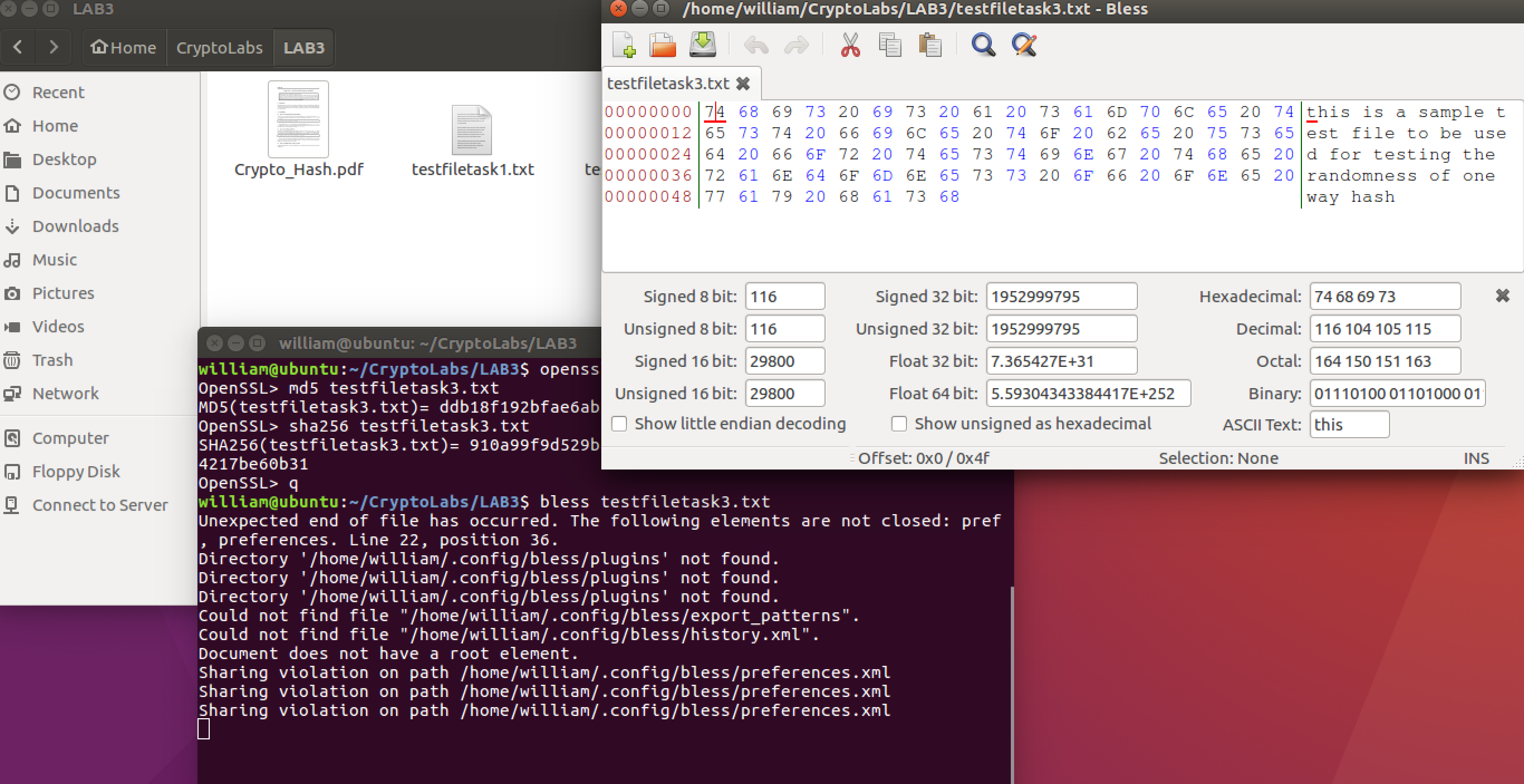
**Task 2**

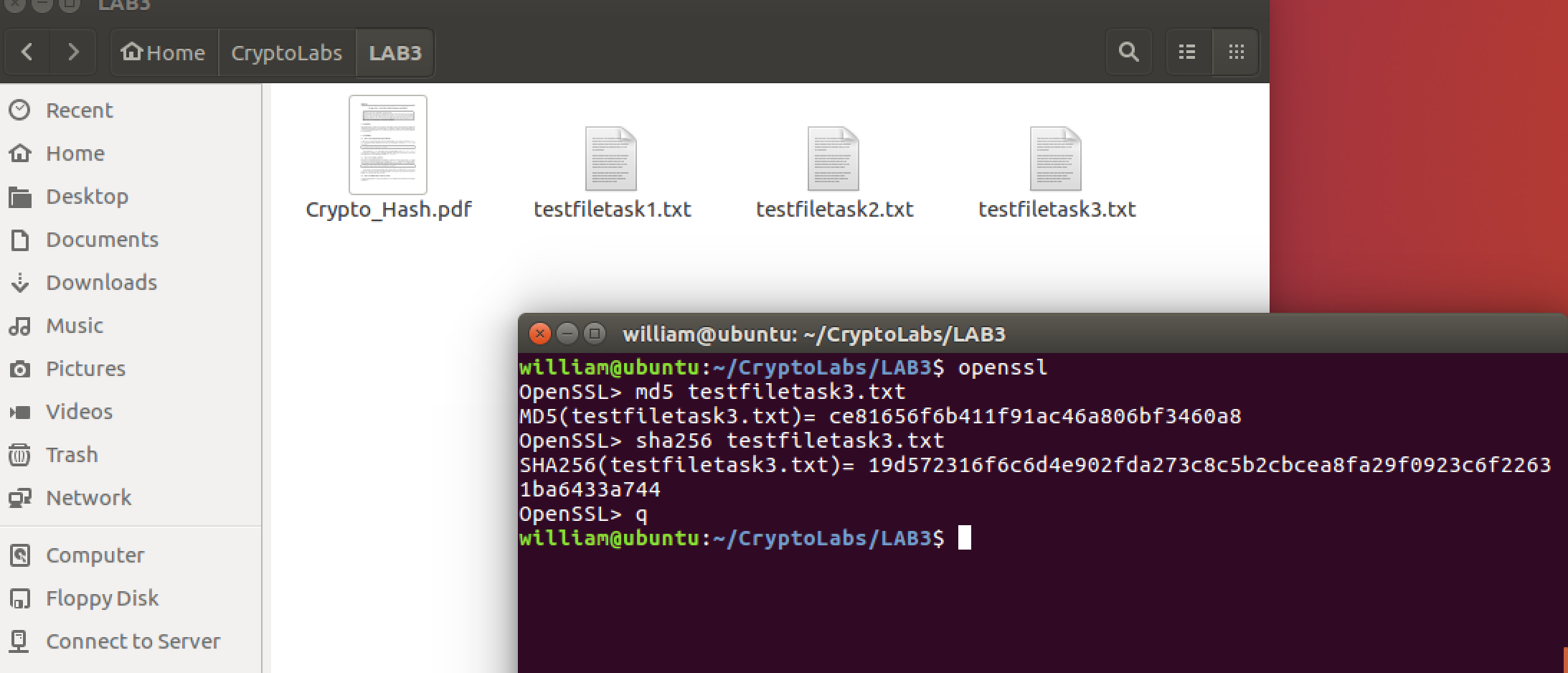
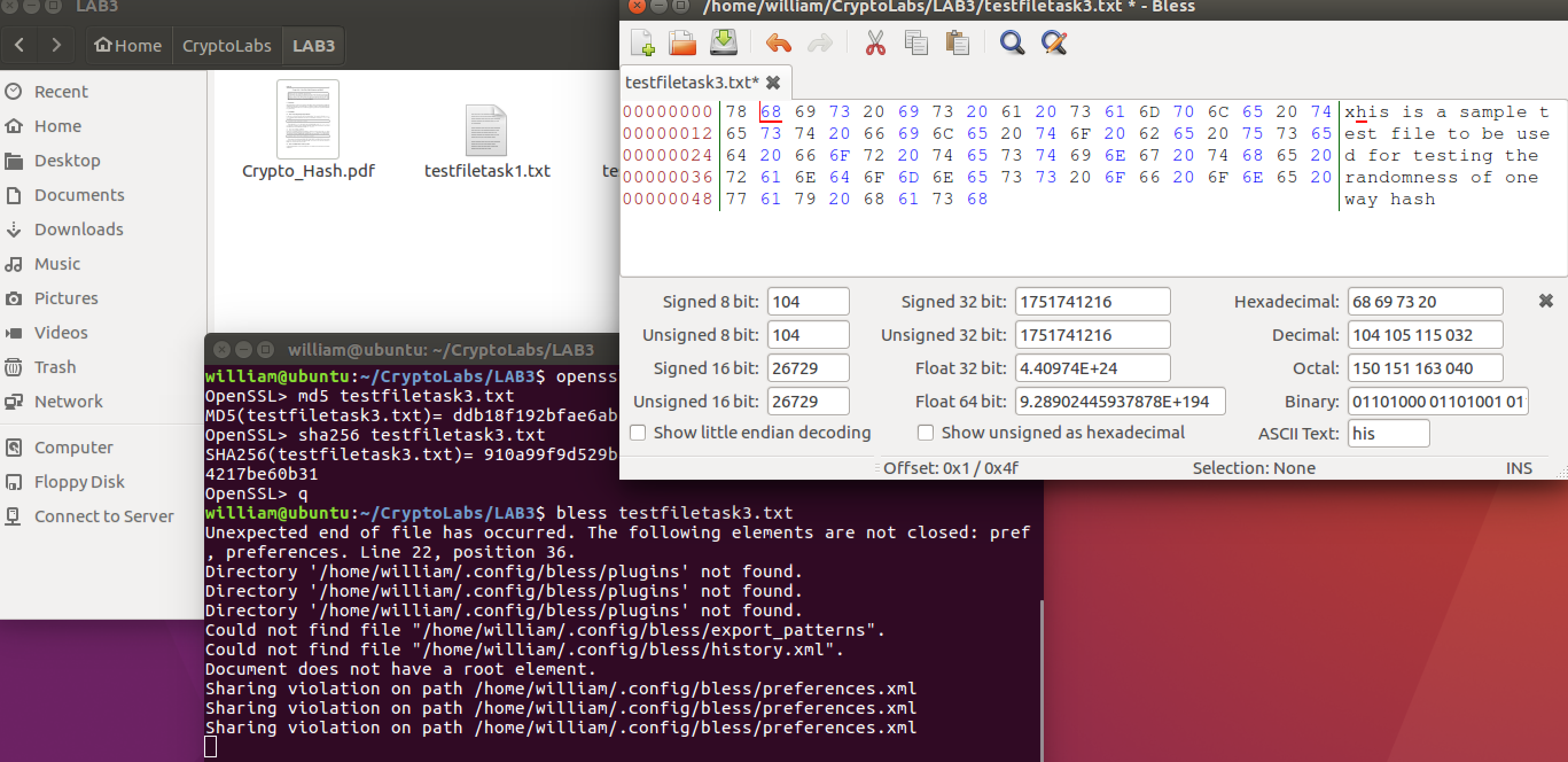


**Observation**: For the task 2, I used the file textfiletask2.txt to implement. I generated several keyed hash values using HMAC-MD5, HMAC-SHA1 and HMAC-SHA256 with different keys (i.e. “ab”, “abcd” and “abcdefg”). The results can be seen in the above figure. From the results, I can see that the HMAC values’ length are same with the hash code length which is expected. And I think I don’t need to use a key with fixed size in HMAC. Because the HMAC algorithm is quite flexible, I tried different sized keys and the algorithms worked well. Besides, what I should do is to use a key in appropriate length to keep secure.

**Task 3**

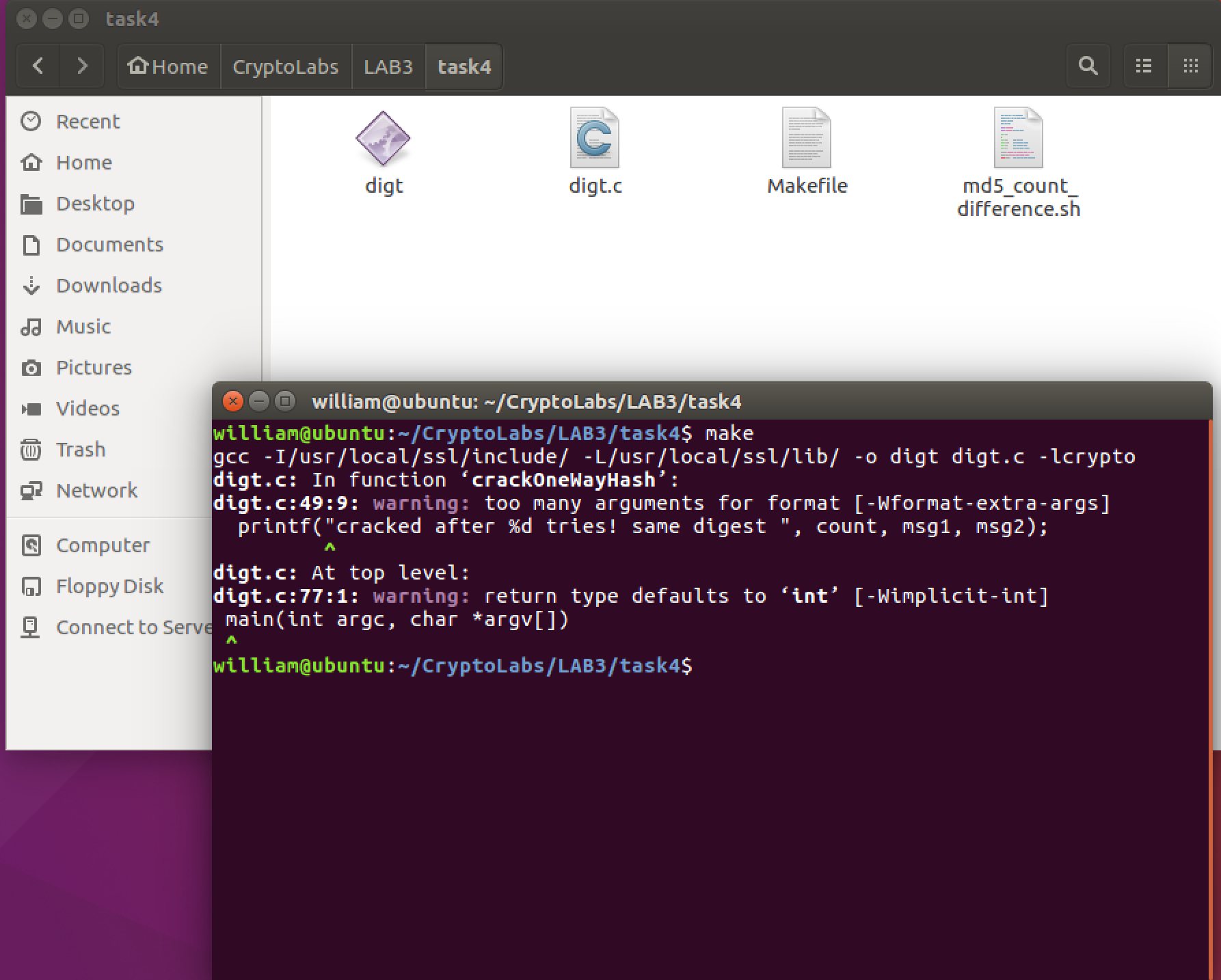
****

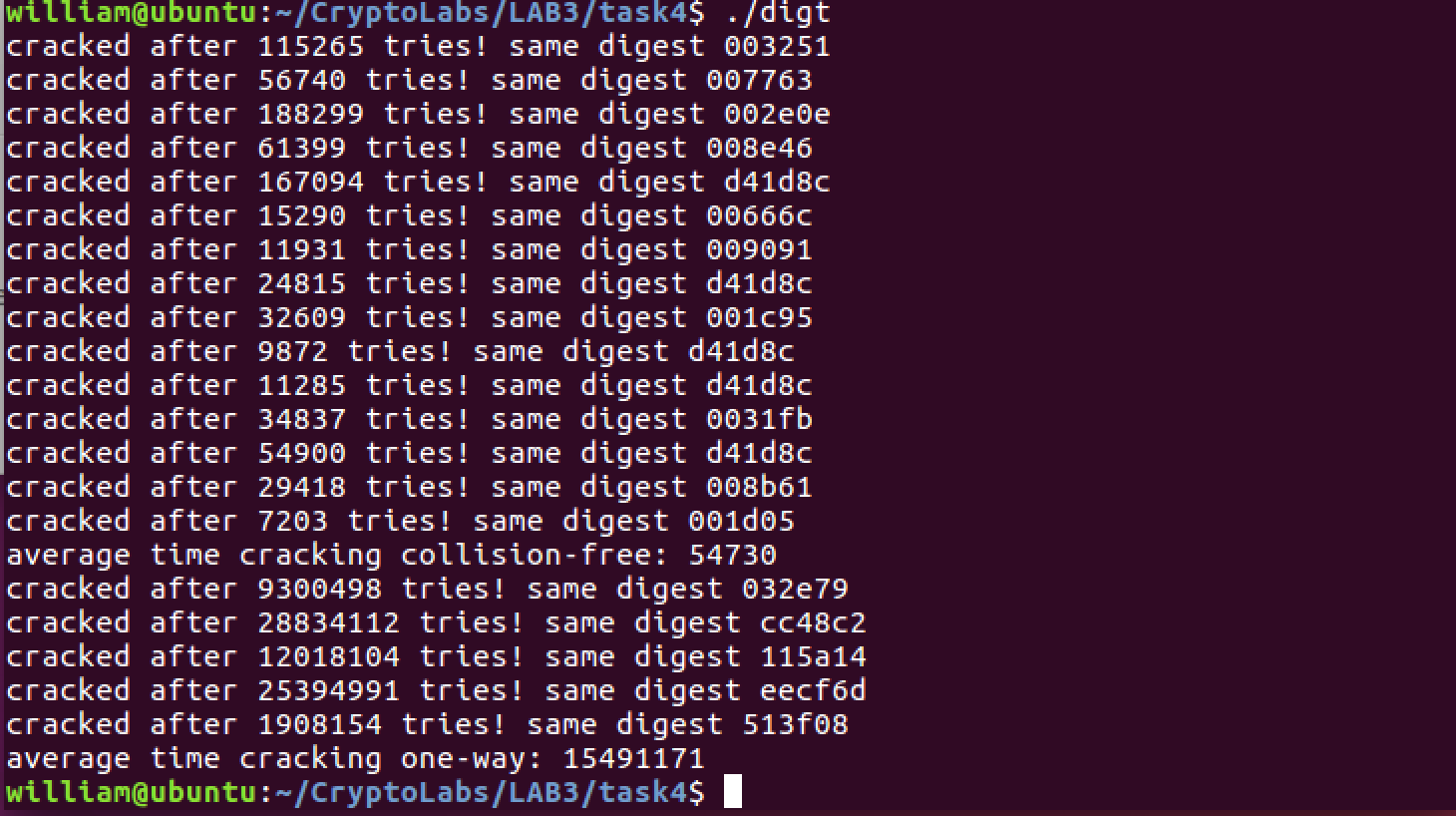


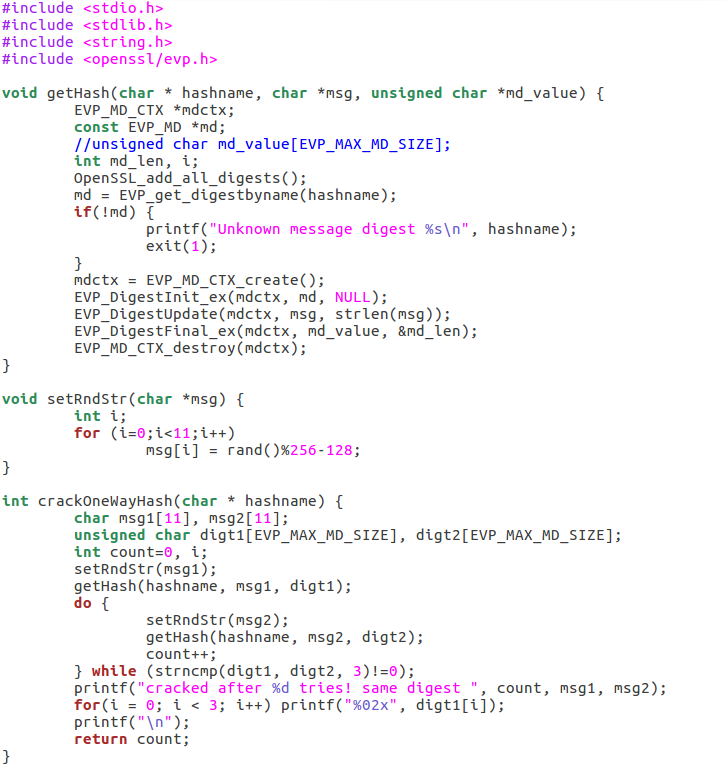


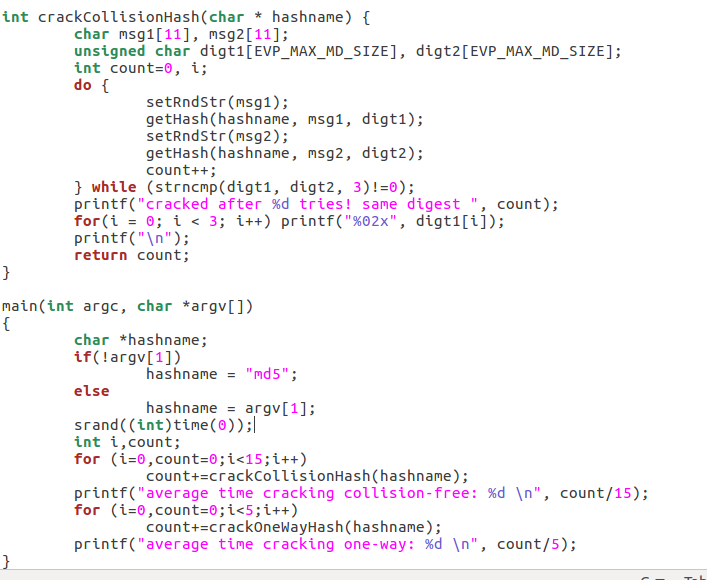
**Observation**: I used the testfiletask3.txt in this task. Firstly, generated the hash values using MD5 and SHA256. Secondly, I changed one bit in the text file (From “this is a sample test file to be used for testing the randomness of one way hash” to “xhis is a sample test file to be used for testing the randomness of one way hash”), and implemented MD5 and SHA256 again to get the new hash values. By comparing them, I found that changing only one bit can cause a totally difference in hash functions. For this example, there is only 1 same bit between the original hash value and the new hash value using MD5 algorithm. While using SHA-256 algorithm, there is no same bit between hash values.

**Task 4**









**Observation**: For this task, I first designed an experiment using C to break the collision-free property. I used a hash function to get the first 24 bits of the hash value. Then in main function, I used a generator to generate random strings in length 32. After that, I compared the first 24 bits of the generated strings’ hash value with the first 24 bits of the hash value. Then I recorded how many trials could we break the collision-free property.

From the results, I can see that, the collision-free property broke in different strings, that’s because I just used the first 24 bits of the hash value. Also, the average number of trials is about 54730times but it is fluctuated greatly. That is probably because of the generator which I used. The generator is not real random; and note that I just pick the first 24 bits of the hash value, thus, the number of trials is large because it’s included the trials that have the same first 24 bits.

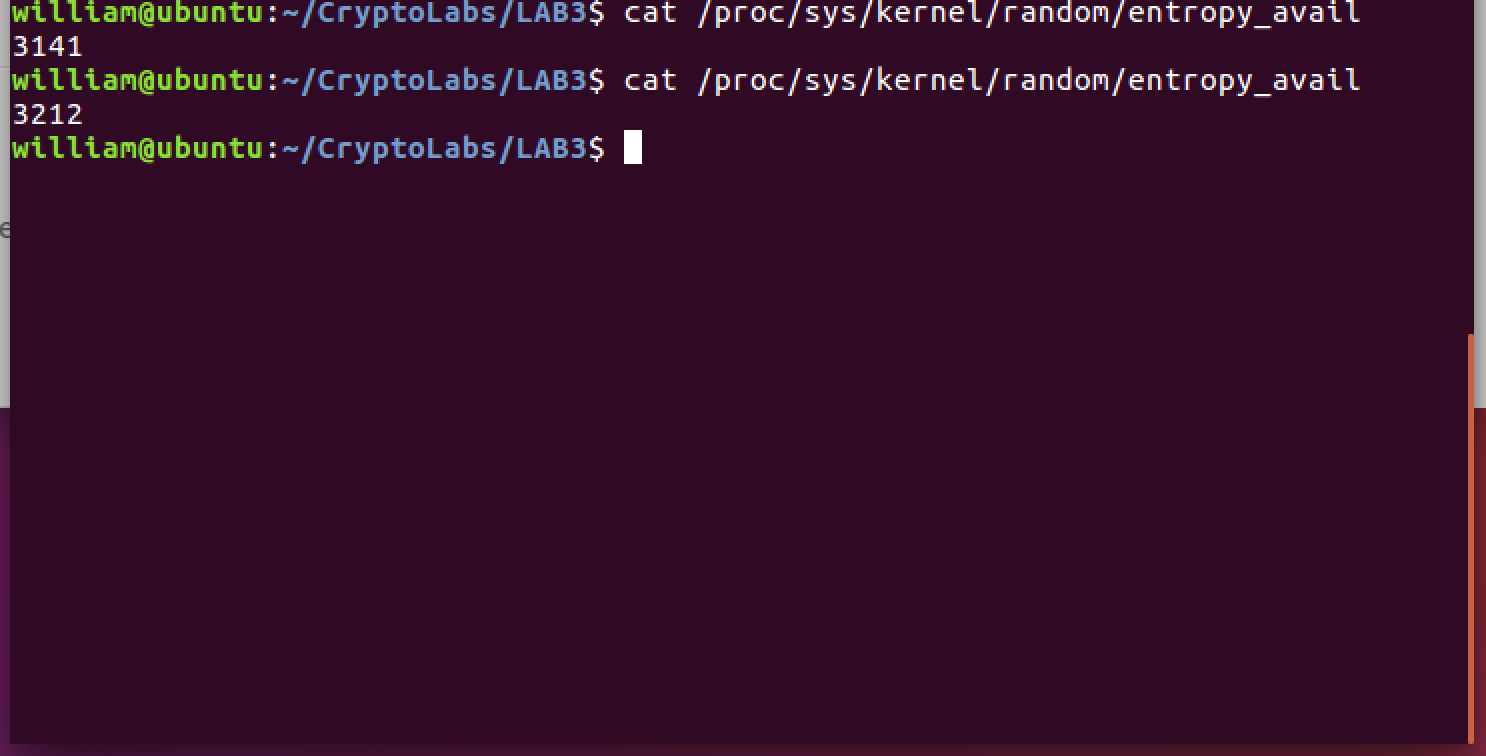
For breaking the one-way property, the code is nearly the same with breaking the collision-free property. I just created the first 24 bits of a hash value, and then I used the generator to generate random strings and find out which string’s hash value can match the hash value we created.

From the result, I can see that, the average number of trials is about 15491171 times, and it’s also fluctuated greatly. The reason is same as above.

Through the observation, I can see that the average numbers of trials for breaking those two properties are almost same using brute force method. However, in my opinion, the one-way property is harder to break. That is because, for breaking the collision-free property, I just need to generate random strings’ hash values and find out two hash values which their first 24 bits are the same.

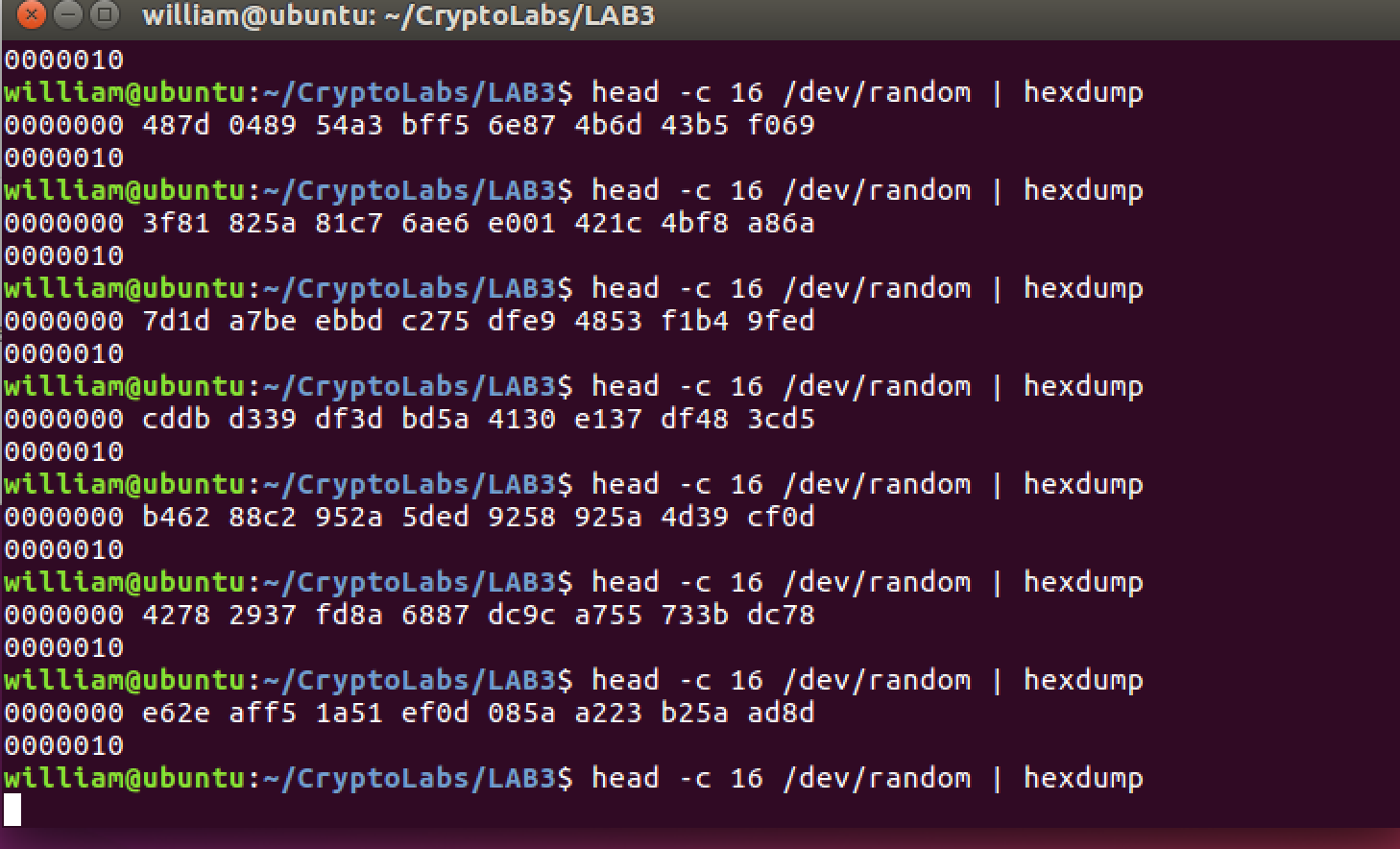
That is comparisons in hexadecimal. But for breaking the one-way property, I should not only find the same hash values, but also ensure the original messages are same. Although the average comparison times are same, breaking one-way property is much harder.

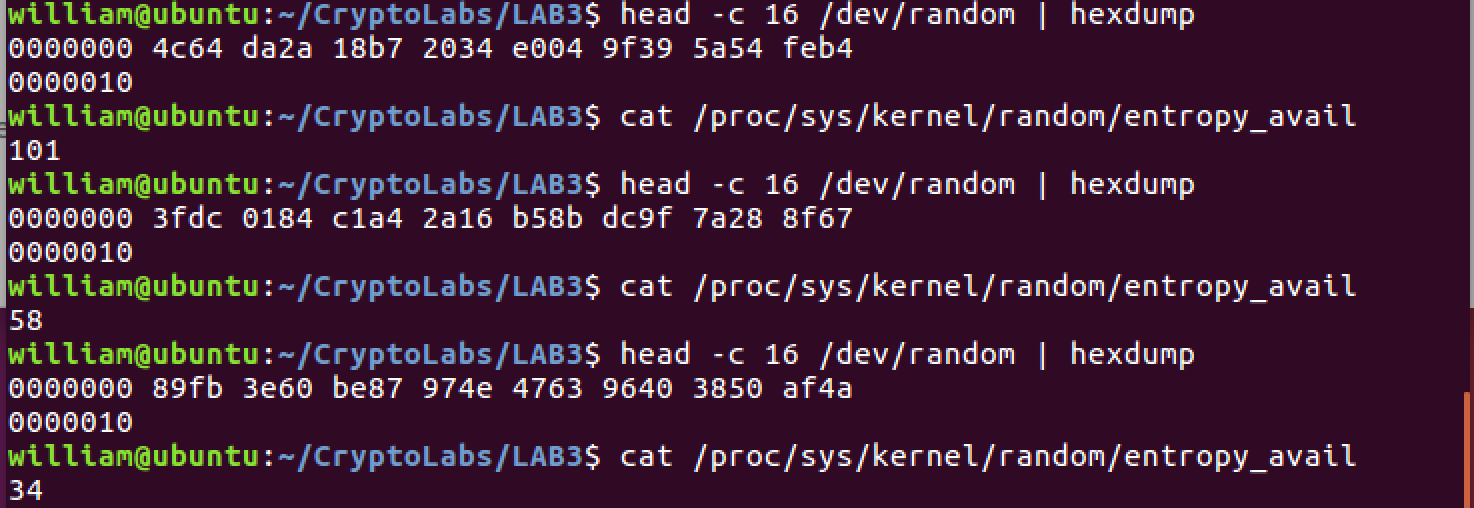
**Task 5.A**

****

**Observation:** I used the given command to get the entropy the kernel has. I can see the second value is larger than the first one. Because I move and click my mouse after the first command.

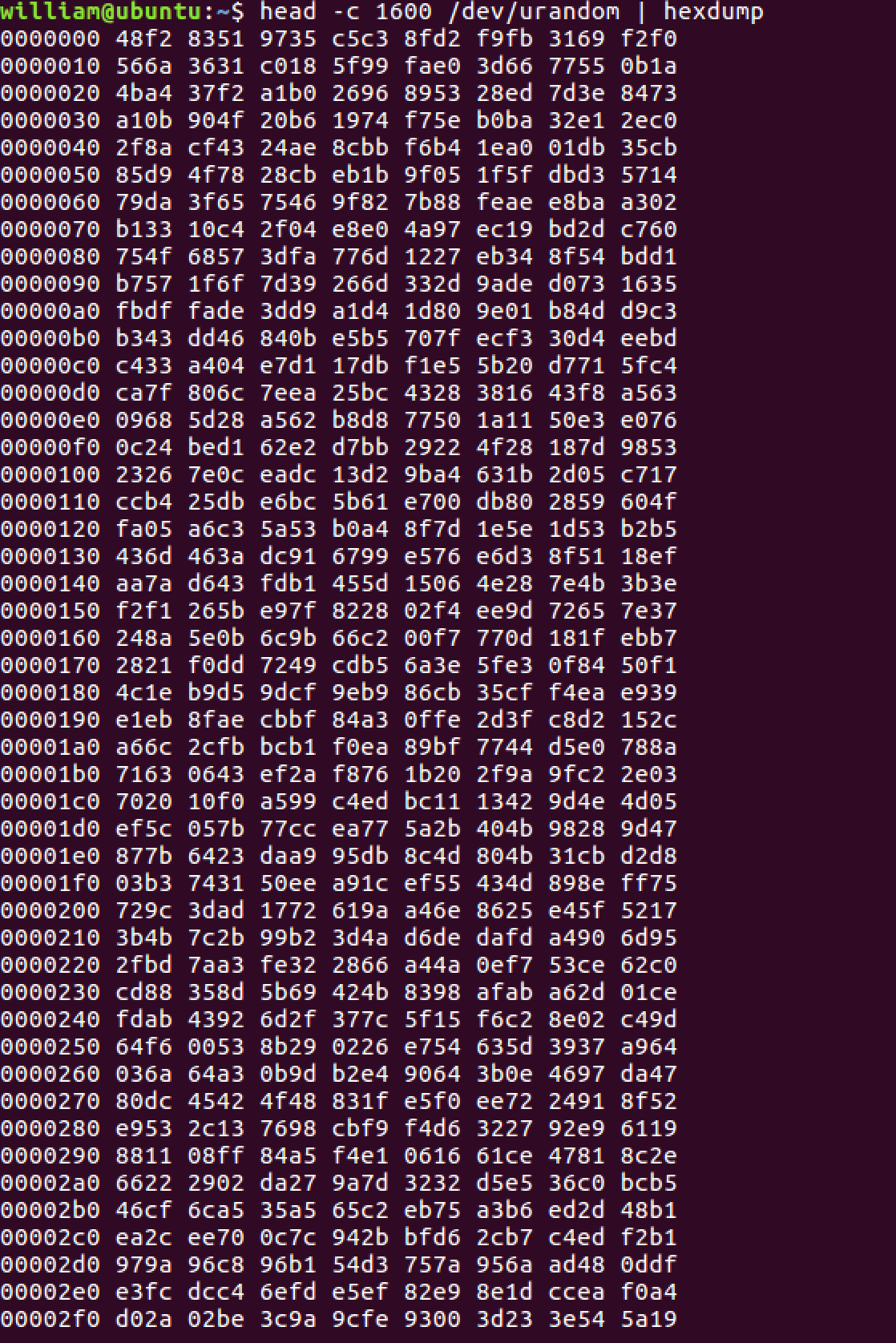
**Task 5.B**

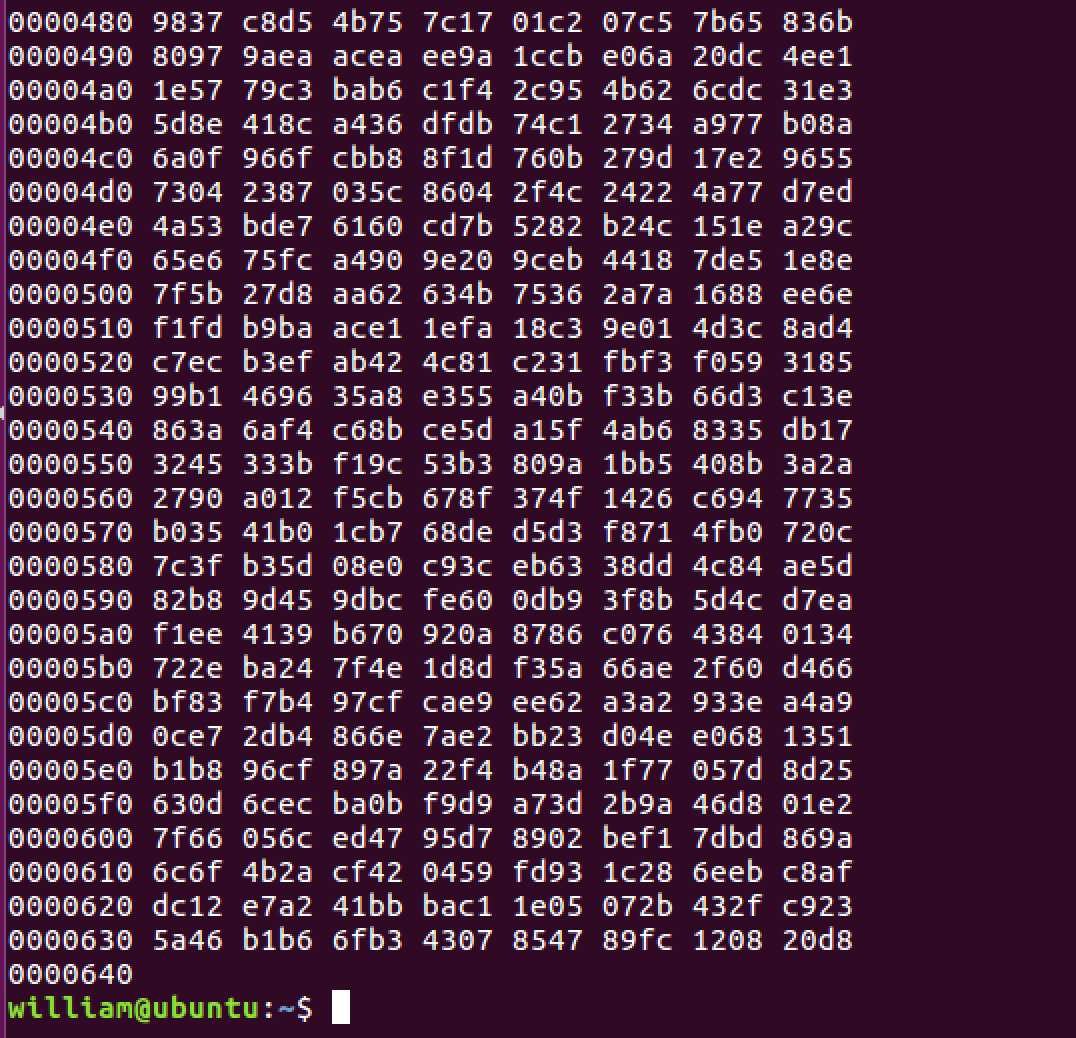
****



**Observation**: After I ran the given command several times, I can see that the program will wait or not print anything. And the entropy number became smaller and smaller after every time I generated the random number until reduce to zero. So, if it is blocked, I can click or move my mouse to make the entropy larger and then generate random number; or I can use /dev/urandom to generate random numbers without waiting.

**Task 5.C**





**Observation**: I used the given command several times to generate 1600 bytes of pseudo random numbers as above figures. It was not blocked.