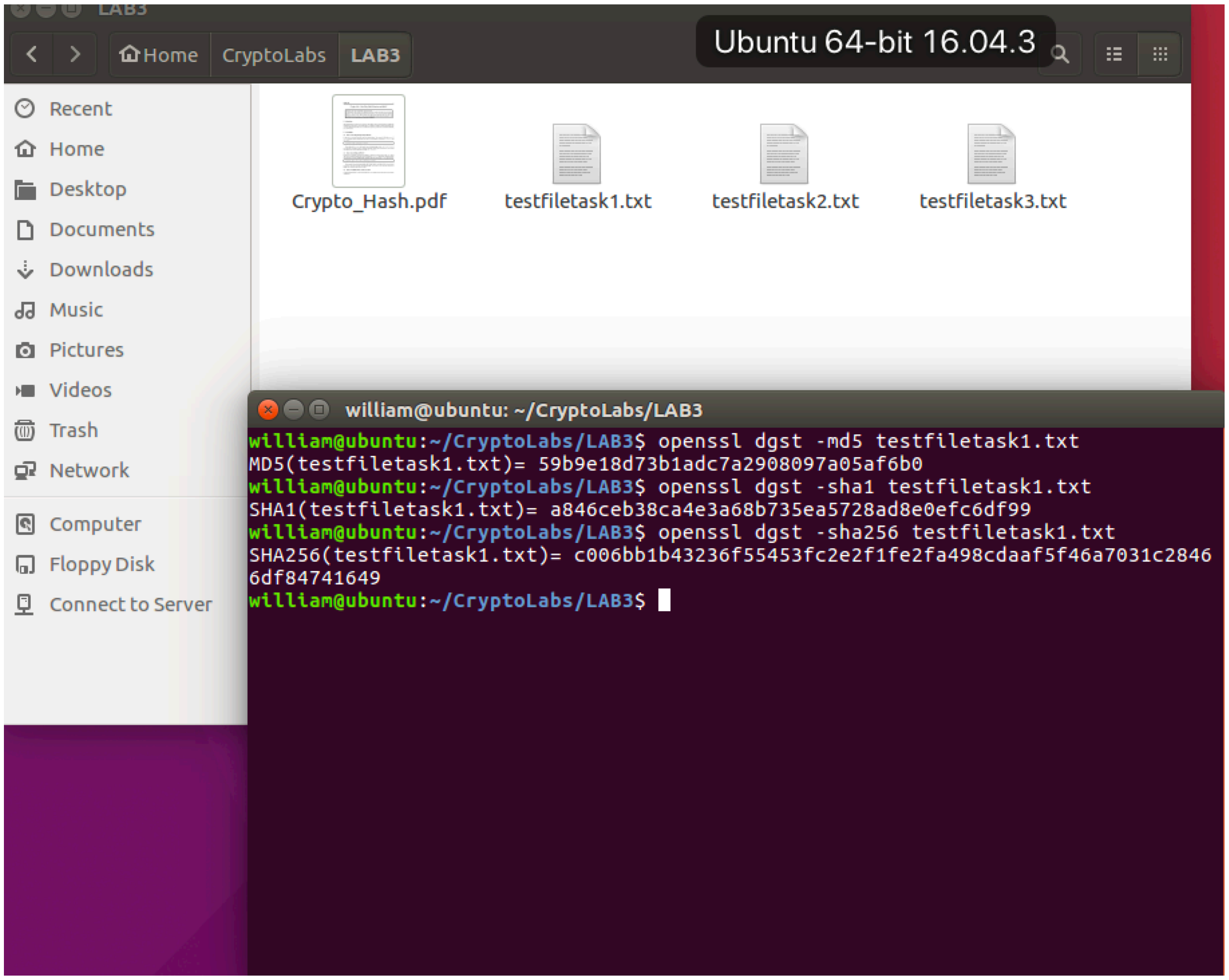
Crypto Lab – One-Way Hash Function and MAC

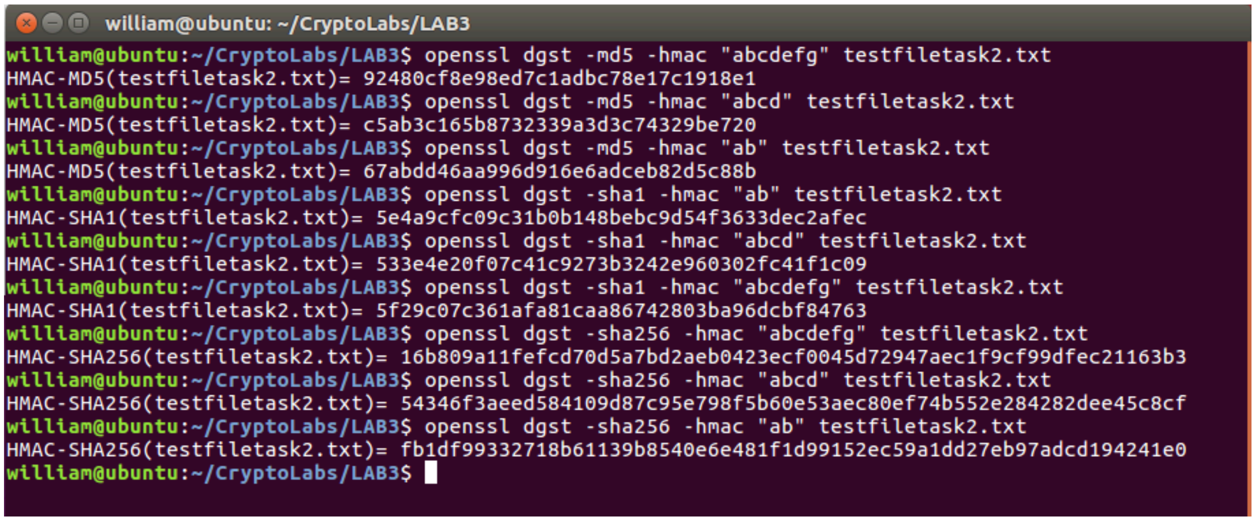
Task 1



```
william@ubuntu:~/CryptoLabs/LAB3$ openssl dgst -md5 testfiletask1.txt
MD5(testfiletask1.txt)= 59b9e18d73b1adc7a2908097a05af6b0
william@ubuntu:~/CryptoLabs/LAB3$ openssl dgst -sha1 testfiletask1.txt
SHA1(testfiletask1.txt)= a846ceb38ca4e3a68b735ea5728ad8e0efc6df99
william@ubuntu:~/CryptoLabs/LAB3$ openssl dgst -sha256 testfiletask1.txt
SHA256(testfiletask1.txt)= c006bb1b43236f55453fc2e2f1fe2fa498cdaaf5f46a7031c2846
6df84741649
william@ubuntu:~/CryptoLabs/LAB3$ 
```

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.

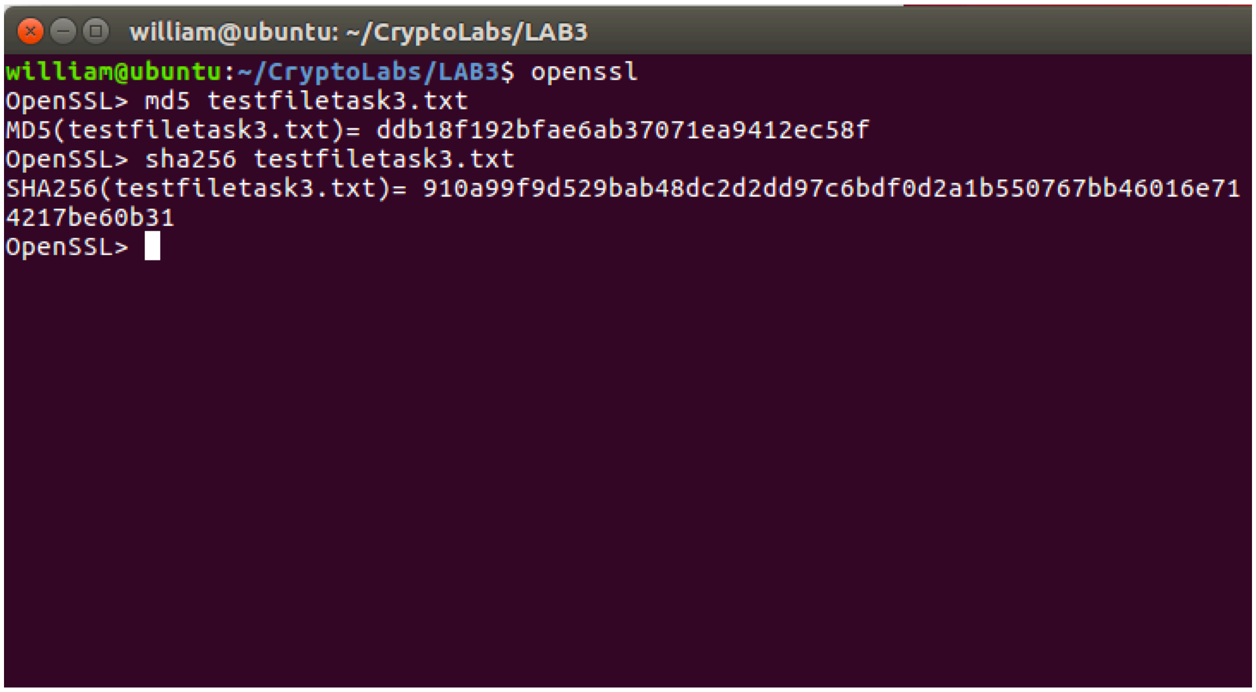
Task 2



```
william@ubuntu: ~/CryptoLabs/LAB3
william@ubuntu:~/CryptoLabs/LAB3$ openssl dgst -md5 -hmac "abcdefg" testfiletask2.txt
HMAC-MD5(testfiletask2.txt)= 92480cf8e98ed7c1adbc78e17c1918e1
william@ubuntu:~/CryptoLabs/LAB3$ openssl dgst -md5 -hmac "abcd" testfiletask2.txt
HMAC-MD5(testfiletask2.txt)= c5ab3c165b8732339a3d3c74329be720
william@ubuntu:~/CryptoLabs/LAB3$ openssl dgst -md5 -hmac "ab" testfiletask2.txt
HMAC-MD5(testfiletask2.txt)= 67abdd46aa996d916e6adceb82d5c88b
william@ubuntu:~/CryptoLabs/LAB3$ openssl dgst -sha1 -hmac "ab" testfiletask2.txt
HMAC-SHA1(testfiletask2.txt)= 5e4a9cfc09c31b0b148bebc9d54f3633dec2afec
william@ubuntu:~/CryptoLabs/LAB3$ openssl dgst -sha1 -hmac "abcd" testfiletask2.txt
HMAC-SHA1(testfiletask2.txt)= 533e4e20f07c41c9273b3242e960302fc41f1c09
william@ubuntu:~/CryptoLabs/LAB3$ openssl dgst -sha1 -hmac "abcdefg" testfiletask2.txt
HMAC-SHA1(testfiletask2.txt)= 5f29c07c361afa81caa86742803ba96dcbf84763
william@ubuntu:~/CryptoLabs/LAB3$ openssl dgst -sha256 -hmac "abcdefg" testfiletask2.txt
HMAC-SHA256(testfiletask2.txt)= 16b809a11fefcd70d5a7bd2aeb0423ecf0045d72947aec1f9cf99dfec21163b3
william@ubuntu:~/CryptoLabs/LAB3$ openssl dgst -sha256 -hmac "abcd" testfiletask2.txt
HMAC-SHA256(testfiletask2.txt)= 54346f3aeed584109d87c95e798f5b60e53aec80ef74b552e284282dee45c8cf
william@ubuntu:~/CryptoLabs/LAB3$ openssl dgst -sha256 -hmac "ab" testfiletask2.txt
HMAC-SHA256(testfiletask2.txt)= fb1df99332718b61139b8540e6e481f1d99152ec59a1dd27eb97adcd194241e0
william@ubuntu:~/CryptoLabs/LAB3$ 
```

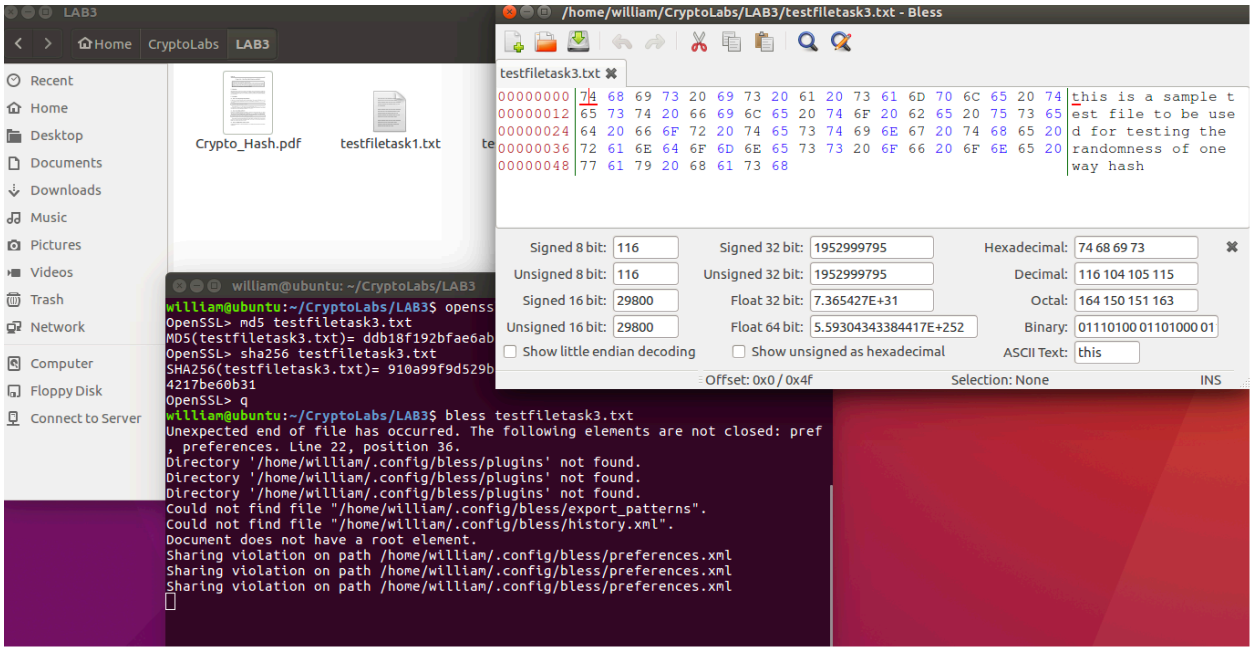
Observation: For the task 2, I used the file textfiletask2.txt to implement. I generated several keyed hash values using HMAC-MD5, HMAC-SHA1 and HMAC-SHA256 with different keys (i.e. “ab”, “abcd” and “abcdefg”). The results can be seen in the above figure. From the results, I can see that the HMAC values’ 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



william@ubuntu: ~/CryptoLabs/LAB3

```
william@ubuntu:~/CryptoLabs/LAB3$ openssl
OpenSSL> md5 testfiletask3.txt
MD5(testfiletask3.txt)= ddb18f192bfae6ab37071ea9412ec58f
OpenSSL> sha256 testfiletask3.txt
SHA256(testfiletask3.txt)= 910a99f9d529bab48dc2d2dd97c6bdf0d2a1b550767bb46016e71
4217be60b31
OpenSSL> 
```



/home/william/CryptoLabs/LAB3/testfiletask3.txt - Bless

testfiletask3.txt

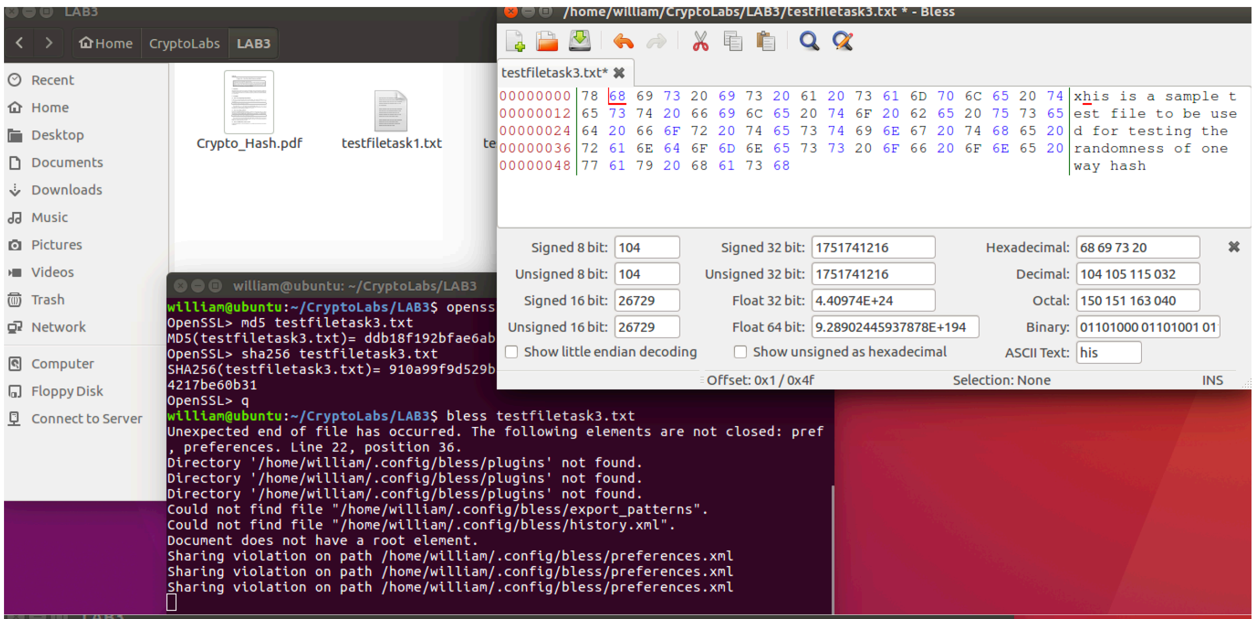
```
00000000 74 68 69 73 20 69 73 20 61 20 73 61 6D 70 6C 65 20 74 this is a sample t
00000012 65 73 74 20 66 69 6C 65 20 74 6F 20 62 65 20 75 73 65 est file to be use
00000024 64 20 66 6F 72 20 74 65 73 74 69 6E 67 20 74 68 65 20 d for testing the
00000036 72 61 6E 64 6F 6D 6E 65 73 73 20 6F 66 20 6F 6E 65 20 randomness of one
00000048 77 61 79 20 68 61 73 68                               way hash
```

Signed 8 bit: 116
Unsigned 8 bit: 116
Signed 16 bit: 29800
Unsigned 16 bit: 29800
☐ Show little endian decoding
Signed 32 bit: 1952999795
Unsigned 32 bit: 1952999795
Float 32 bit: 7.365427E+31
Float 64 bit: 5.59304343384417E+252
☐ Show unsigned as hexadecimal
Hexadecimal: 74 68 69 73
Decimal: 116 104 105 115
Octal: 164 150 151 163
Binary: 01110100 01101000 01
ASCII Text: this

Offset: 0x0 / 0x4f   Selection: None   INS

william@ubuntu: ~/CryptoLabs/LAB3

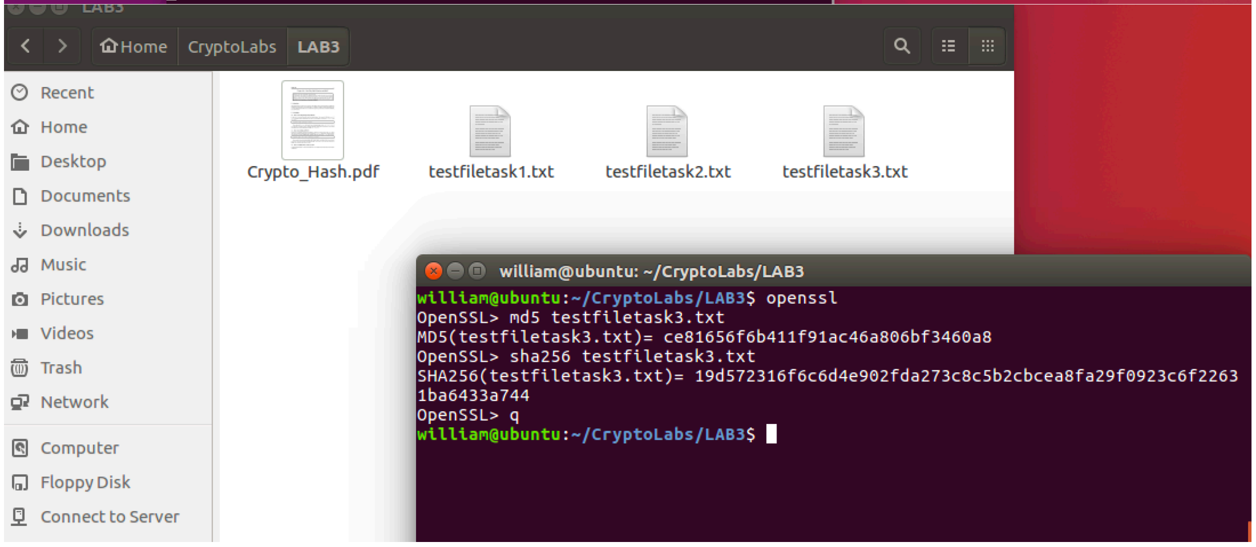
```
william@ubuntu:~/CryptoLabs/LAB3$ openss
OpenSSL> md5 testfiletask3.txt
MD5(testfiletask3.txt)= ddb18f192bfae6ab
OpenSSL> sha256 testfiletask3.txt
SHA256(testfiletask3.txt)= 910a99f9d529b
4217be60b31
OpenSSL> q
william@ubuntu:~/CryptoLabs/LAB3$ bless testfiletask3.txt
Unexpected end of file has occurred. The following elements are not closed: pref
, preferences. Line 22, position 36.
Directory '/home/william/.config/bless/plugins' not found.
Directory '/home/william/.config/bless/plugins' not found.
Directory '/home/william/.config/bless/plugins' not found.
Could not find file "/home/william/.config/bless/export_patterns".
Could not find file "/home/william/.config/bless/history.xml".
Document does not have a root element.
Sharing violation on path /home/william/.config/bless/preferences.xml
Sharing violation on path /home/william/.config/bless/preferences.xml
Sharing violation on path /home/william/.config/bless/preferences.xml
```



```
/home/william/CryptoLabs/LAB3/testfiletask3.txt * - Bless
testfiletask3.txt*
00000000 78 68 69 73 20 69 73 20 61 20 73 61 6D 70 6C 65 20 74 xhis is a sample t
00000012 65 73 74 20 66 69 6C 65 20 74 6F 20 62 65 20 75 73 65 est file to be use
00000024 64 20 66 6F 72 20 74 65 73 74 69 6E 67 20 74 68 65 20 d for testing the
00000036 72 61 6E 64 6F 6D 6E 65 73 73 20 6F 66 20 6F 6E 65 20 randomness of one
00000048 77 61 79 20 68 61 73 68                               way hash

  Signed 8 bit: 104      Signed 32 bit: 1751741216              Hexadecimal: 68 69 73 20
Unsigned 8 bit: 104    Unsigned 32 bit: 1751741216                  Decimal: 104 105 115 032
 Signed 16 bit: 26729     Float 32 bit: 4.40974E+24                   Octal: 150 151 163 040
Unsigned 16 bit: 26729    Float 64 bit: 9.28902445937878E+194        Binary: 01101000 01101001 01
[ ] Show little endian decoding   [ ] Show unsigned as hexadecimal   ASCII Text: his
Offset: 0x1 / 0x4f        Selection: None        INS
```

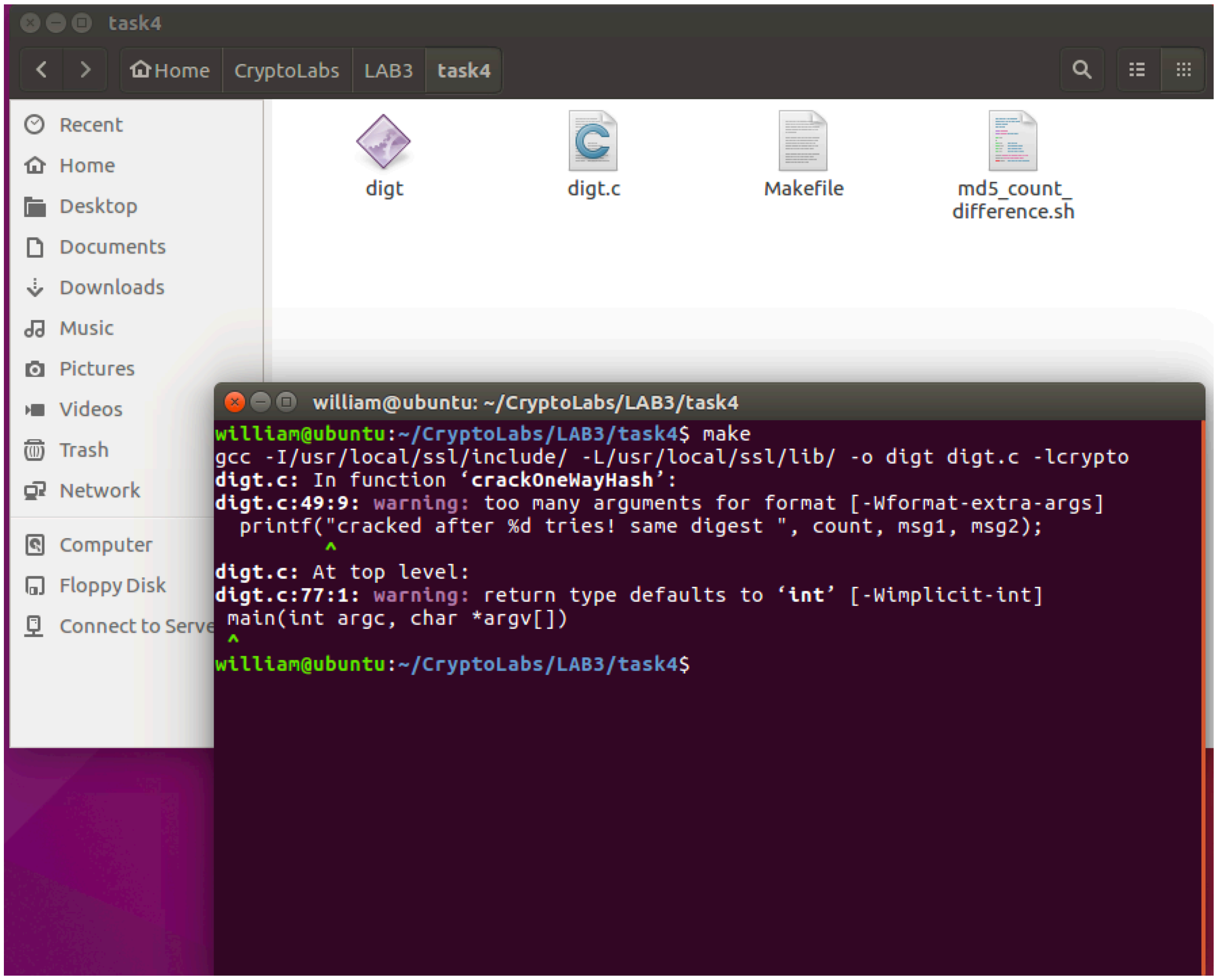
```
william@ubuntu:~/CryptoLabs/LAB3$ openss
OpenSSL> md5 testfiletask3.txt
MD5(testfiletask3.txt)= ddb18f192bfae6ab
OpenSSL> sha256 testfiletask3.txt
SHA256(testfiletask3.txt)= 910a99f9d529b
4217be60b31
OpenSSL> q
william@ubuntu:~/CryptoLabs/LAB3$ bless testfiletask3.txt
Unexpected end of file has occurred. The following elements are not closed: pref
, preferences. Line 22, position 36.
Directory '/home/william/.config/bless/plugins' not found.
Directory '/home/william/.config/bless/plugins' not found.
Directory '/home/william/.config/bless/plugins' not found.
Could not find file "/home/william/.config/bless/export_patterns".
Could not find file "/home/william/.config/bless/history.xml".
Document does not have a root element.
Sharing violation on path /home/william/.config/bless/preferences.xml
Sharing violation on path /home/william/.config/bless/preferences.xml
Sharing violation on path /home/william/.config/bless/preferences.xml
```



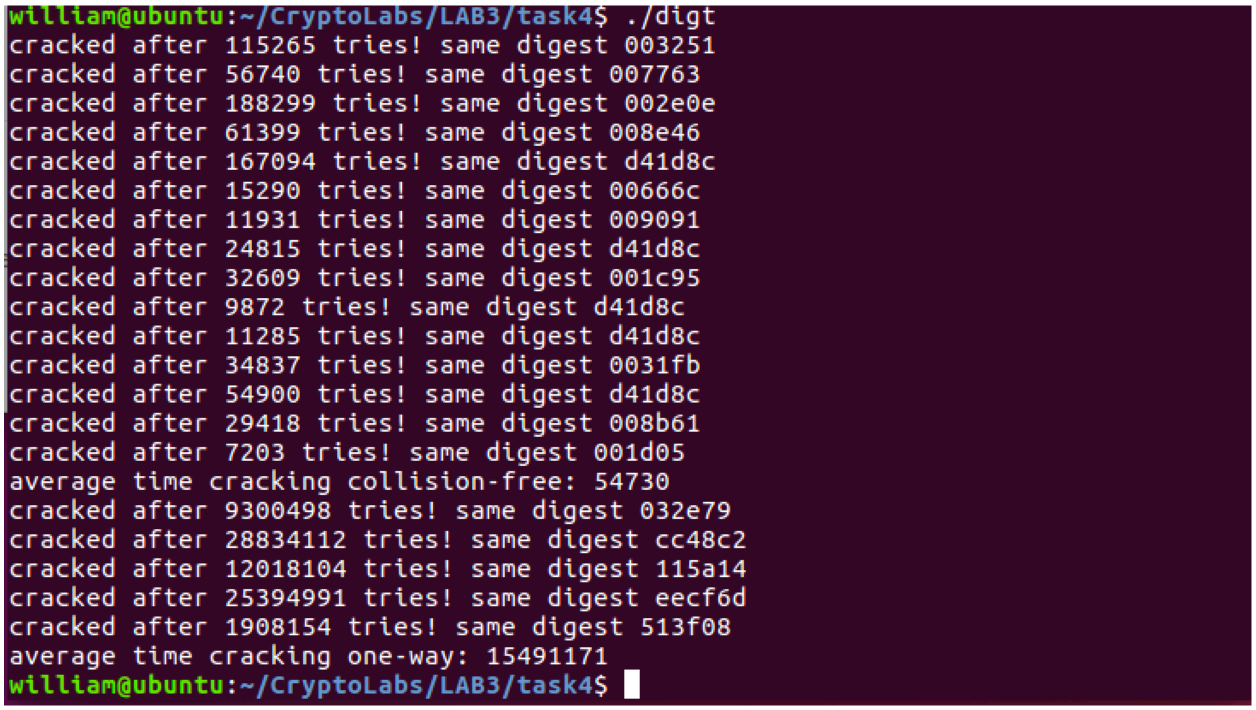
```
william@ubuntu:~/CryptoLabs/LAB3$ openssl
OpenSSL> md5 testfiletask3.txt
MD5(testfiletask3.txt)= ce81656f6b411f91ac46a806bf3460a8
OpenSSL> sha256 testfiletask3.txt
SHA256(testfiletask3.txt)= 19d572316f6c6d4e902fda273c8c5b2cbcea8fa29f0923c6f2263
1ba6433a744
OpenSSL> q
william@ubuntu:~/CryptoLabs/LAB3$
```

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.

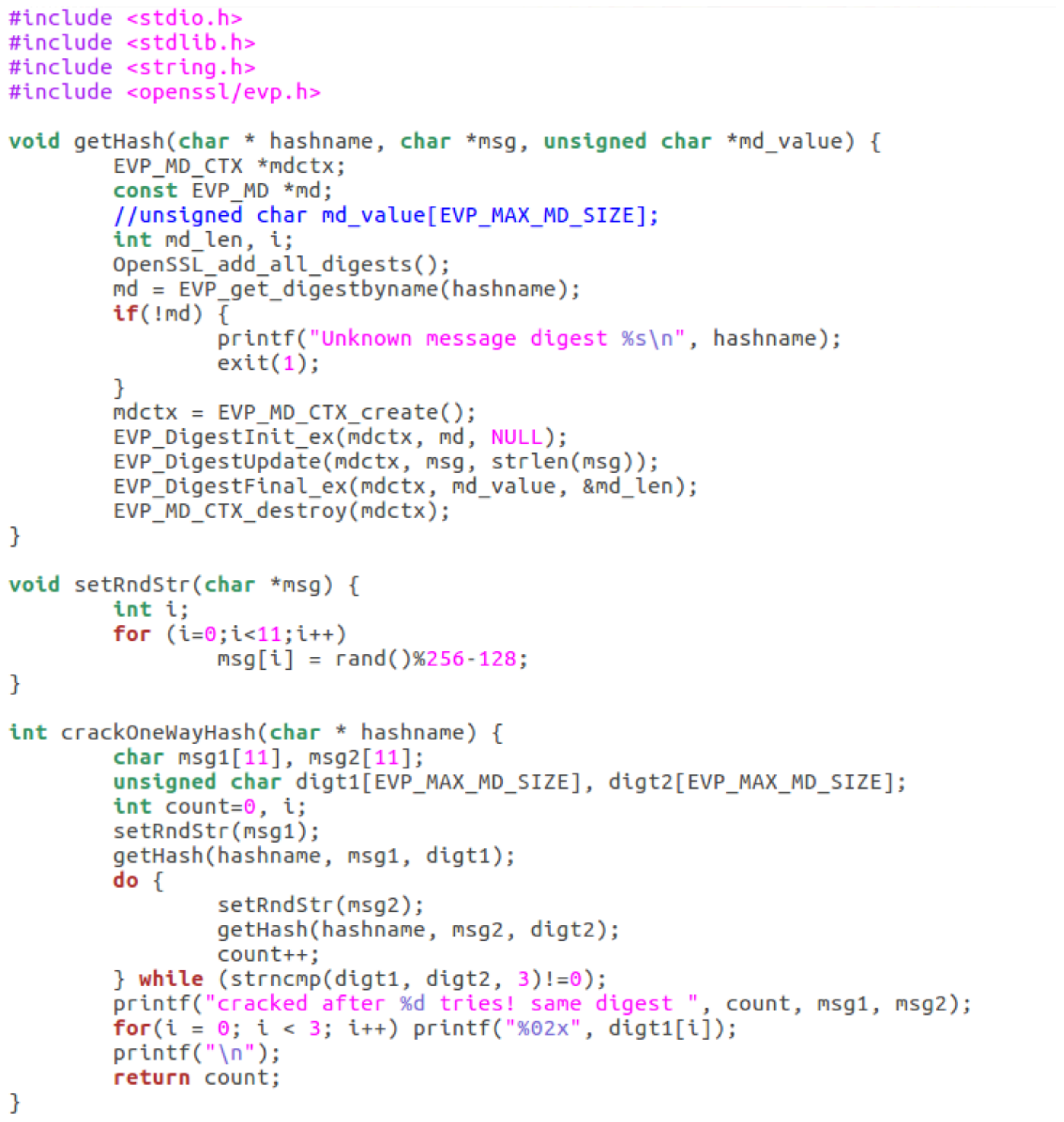
Task 4



```
william@ubuntu:~/CryptoLabs/LAB3/task4$ make
gcc -I/usr/local/ssl/include/ -L/usr/local/ssl/lib/ -o digt digt.c -lcrypto
digt.c: In function ‘crackOneWayHash’:
digt.c:49:9: warning: too many arguments for format [-Wformat-extra-args]
  printf("cracked after %d tries! same digest ", count, msg1, msg2);
         ^
digt.c: At top level:
digt.c:77:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
 main(int argc, char *argv[])
 ^
william@ubuntu:~/CryptoLabs/LAB3/task4$
```



```
william@ubuntu:~/CryptoLabs/LAB3/task4$ ./digt
cracked after 115265 tries! same digest 003251
cracked after 56740 tries! same digest 007763
cracked after 188299 tries! same digest 002e0e
cracked after 61399 tries! same digest 008e46
cracked after 167094 tries! same digest d41d8c
cracked after 15290 tries! same digest 00666c
cracked after 11931 tries! same digest 009091
cracked after 24815 tries! same digest d41d8c
cracked after 32609 tries! same digest 001c95
cracked after 9872 tries! same digest d41d8c
cracked after 11285 tries! same digest d41d8c
cracked after 34837 tries! same digest 0031fb
cracked after 54900 tries! same digest d41d8c
cracked after 29418 tries! same digest 008b61
cracked after 7203 tries! same digest 001d05
average time cracking collision-free: 54730
cracked after 9300498 tries! same digest 032e79
cracked after 28834112 tries! same digest cc48c2
cracked after 12018104 tries! same digest 115a14
cracked after 25394991 tries! same digest eecf6d
cracked after 1908154 tries! same digest 513f08
average time cracking one-way: 15491171
william@ubuntu:~/CryptoLabs/LAB3/task4$
```

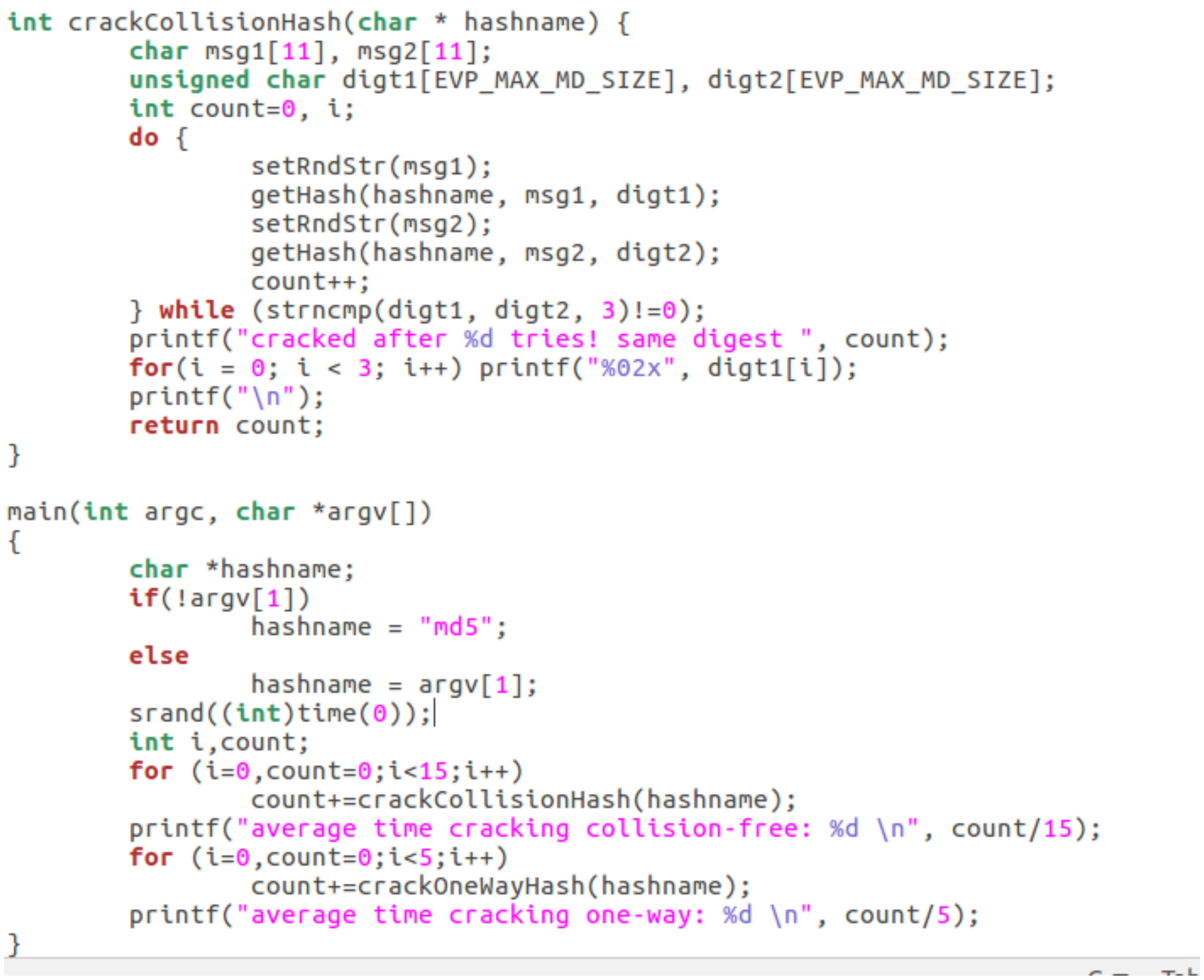


```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <openssl/evp.h>

void getHash(char * hashname, char *msg, unsigned char *md_value) {
        EVP_MD_CTX *mdctx;
        const EVP_MD *md;
        //unsigned char md_value[EVP_MAX_MD_SIZE];
        int md_len, i;
        OpenSSL_add_all_digests();
        md = EVP_get_digestbyname(hashname);
        if(!md) {
                printf("Unknown message digest %s\n", hashname);
                exit(1);
        }
        mdctx = EVP_MD_CTX_create();
        EVP_DigestInit_ex(mdctx, md, NULL);
        EVP_DigestUpdate(mdctx, msg, strlen(msg));
        EVP_DigestFinal_ex(mdctx, md_value, &md_len);
        EVP_MD_CTX_destroy(mdctx);
}

void setRndStr(char *msg) {
        int i;
        for (i=0;i<11;i++)
                msg[i] = rand()%256-128;
}

int crackOneWayHash(char * hashname) {
        char msg1[11], msg2[11];
        unsigned char digt1[EVP_MAX_MD_SIZE], digt2[EVP_MAX_MD_SIZE];
        int count=0, i;
        setRndStr(msg1);
        getHash(hashname, msg1, digt1);
        do {
                setRndStr(msg2);
                getHash(hashname, msg2, digt2);
                count++;
        } while (strncmp(digt1, digt2, 3)!=0);
        printf("cracked after %d tries! same digest ", count, msg1, msg2);
        for(i = 0; i < 3; i++) printf("%02x", digt1[i]);
        printf("\n");
        return count;
}
```



```c
int crackCollisionHash(char * hashname) {
        char msg1[11], msg2[11];
        unsigned char digt1[EVP_MAX_MD_SIZE], digt2[EVP_MAX_MD_SIZE];
        int count=0, i;
        do {
                setRndStr(msg1);
                getHash(hashname, msg1, digt1);
                setRndStr(msg2);
                getHash(hashname, msg2, digt2);
                count++;
        } while (strncmp(digt1, digt2, 3)!=0);
        printf("cracked after %d tries! same digest ", count);
        for(i = 0; i < 3; i++) printf("%02x", digt1[i]);
        printf("\n");
        return count;
}

main(int argc, char *argv[])
{
        char *hashname;
        if(!argv[1])
                hashname = "md5";
        else
                hashname = argv[1];
        srand((int)time(0));
        int i,count;
        for (i=0,count=0;i<15;i++)
                count+=crackCollisionHash(hashname);
        printf("average time cracking collision-free: %d \n", count/15);
        for (i=0,count=0;i<5;i++)
                count+=crackOneWayHash(hashname);
        printf("average time cracking one-way: %d \n", count/5);
}
```

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 54730 times 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.

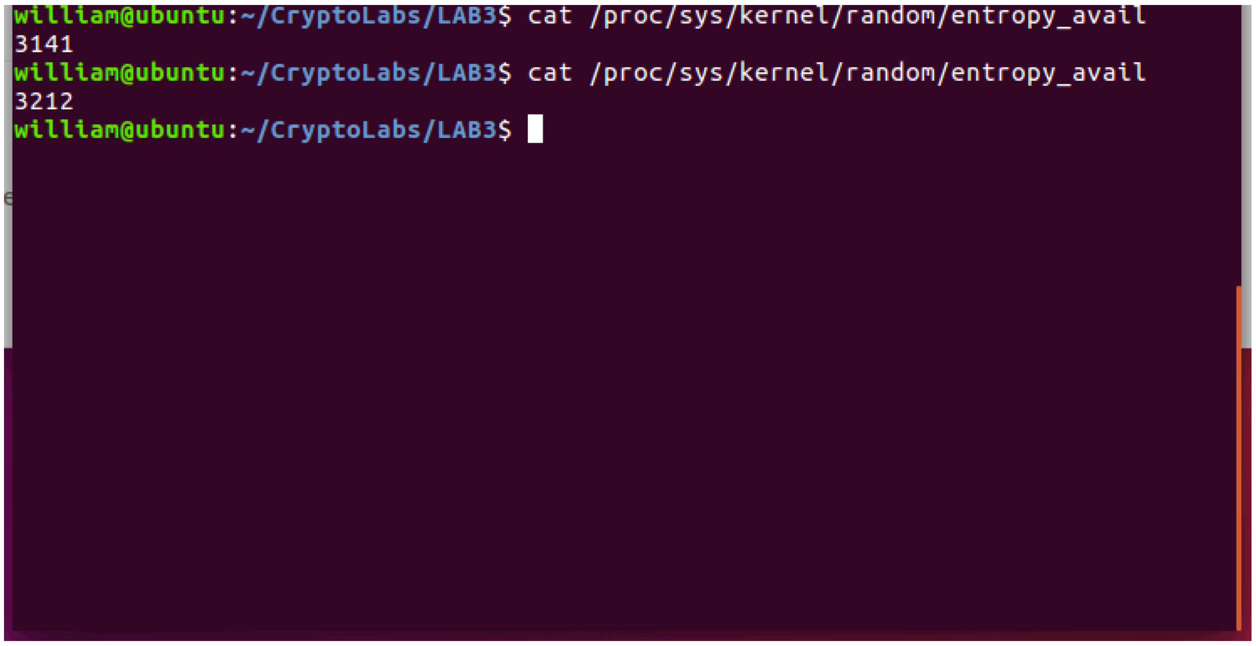
For breaking the one-way property, the code is nearly the same with breaking the collision-free property. I just created the first 24 bits of a hash value, and then I used the generator to generate random strings and find out which string's hash value can match the hash value we created.

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 16^3 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.

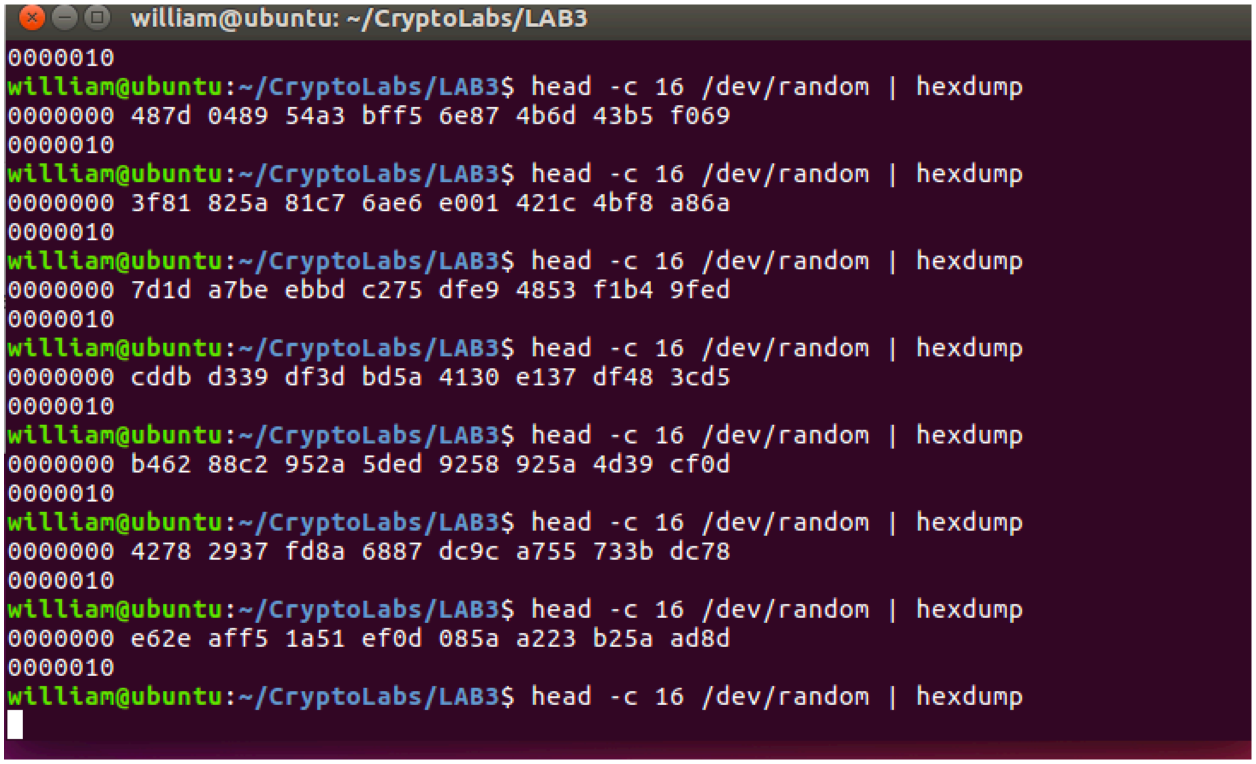
Task 5.A



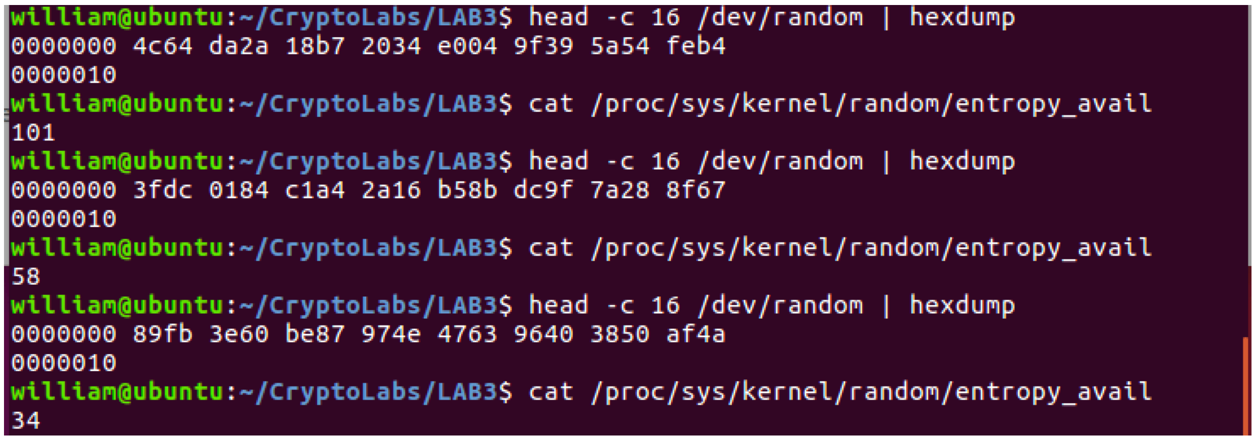
```
william@ubuntu:~/CryptoLabs/LAB3$ cat /proc/sys/kernel/random/entropy_avail
3141
william@ubuntu:~/CryptoLabs/LAB3$ cat /proc/sys/kernel/random/entropy_avail
3212
william@ubuntu:~/CryptoLabs/LAB3$ 
```

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



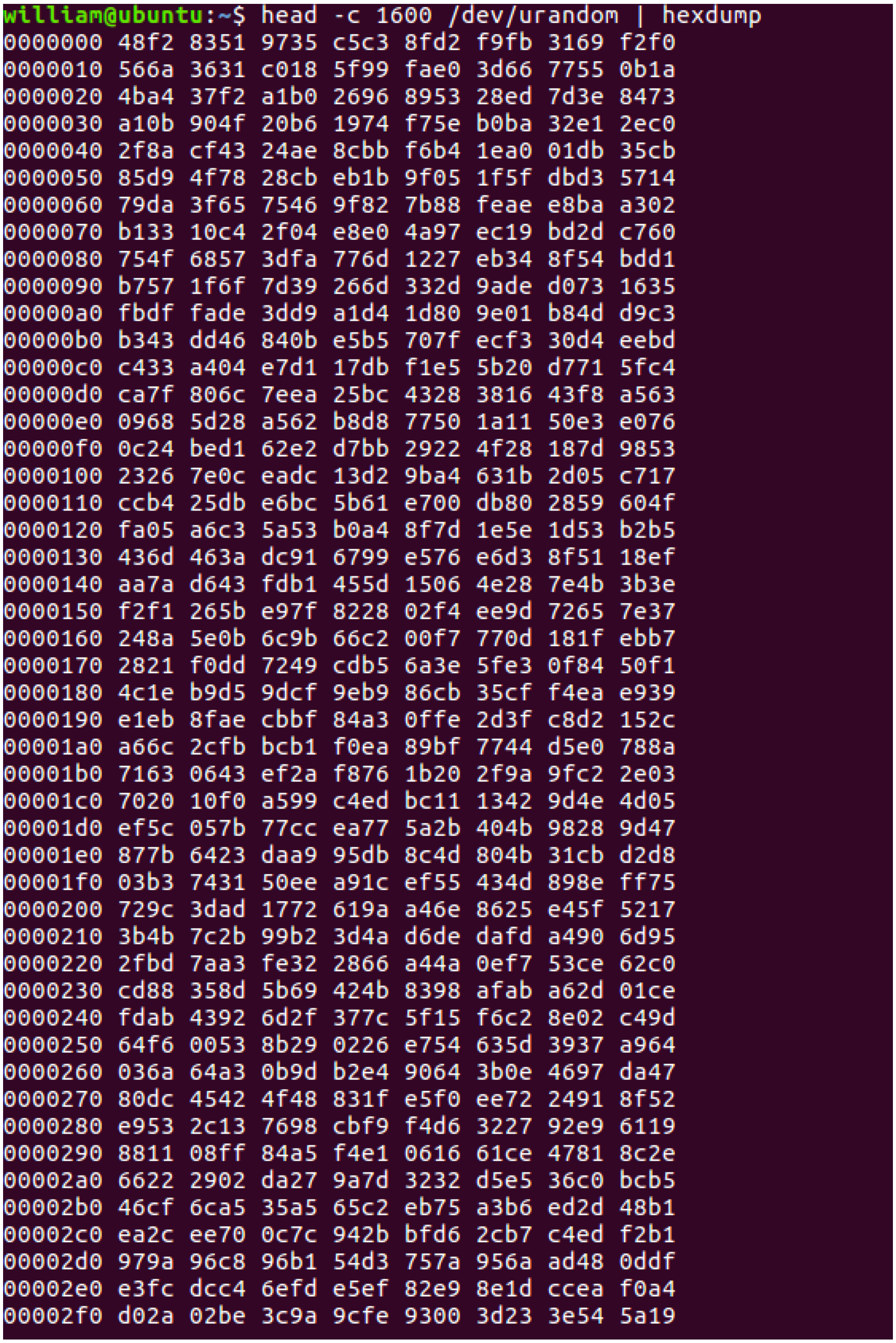
```
william@ubuntu: ~/CryptoLabs/LAB3
0000010
william@ubuntu:~/CryptoLabs/LAB3$ head -c 16 /dev/random | hexdump
0000000 487d 0489 54a3 bff5 6e87 4b6d 43b5 f069
0000010
william@ubuntu:~/CryptoLabs/LAB3$ head -c 16 /dev/random | hexdump
0000000 3f81 825a 81c7 6ae6 e001 421c 4bf8 a86a
0000010
william@ubuntu:~/CryptoLabs/LAB3$ head -c 16 /dev/random | hexdump
0000000 7d1d a7be ebbd c275 dfe9 4853 f1b4 9fed
0000010
william@ubuntu:~/CryptoLabs/LAB3$ head -c 16 /dev/random | hexdump
0000000 cddb d339 df3d bd5a 4130 e137 df48 3cd5
0000010
william@ubuntu:~/CryptoLabs/LAB3$ head -c 16 /dev/random | hexdump
0000000 b462 88c2 952a 5ded 9258 925a 4d39 cf0d
0000010
william@ubuntu:~/CryptoLabs/LAB3$ head -c 16 /dev/random | hexdump
0000000 4278 2937 fd8a 6887 dc9c a755 733b dc78
0000010
william@ubuntu:~/CryptoLabs/LAB3$ head -c 16 /dev/random | hexdump
0000000 e62e aff5 1a51 ef0d 085a a223 b25a ad8d
0000010
william@ubuntu:~/CryptoLabs/LAB3$ head -c 16 /dev/random | hexdump
```



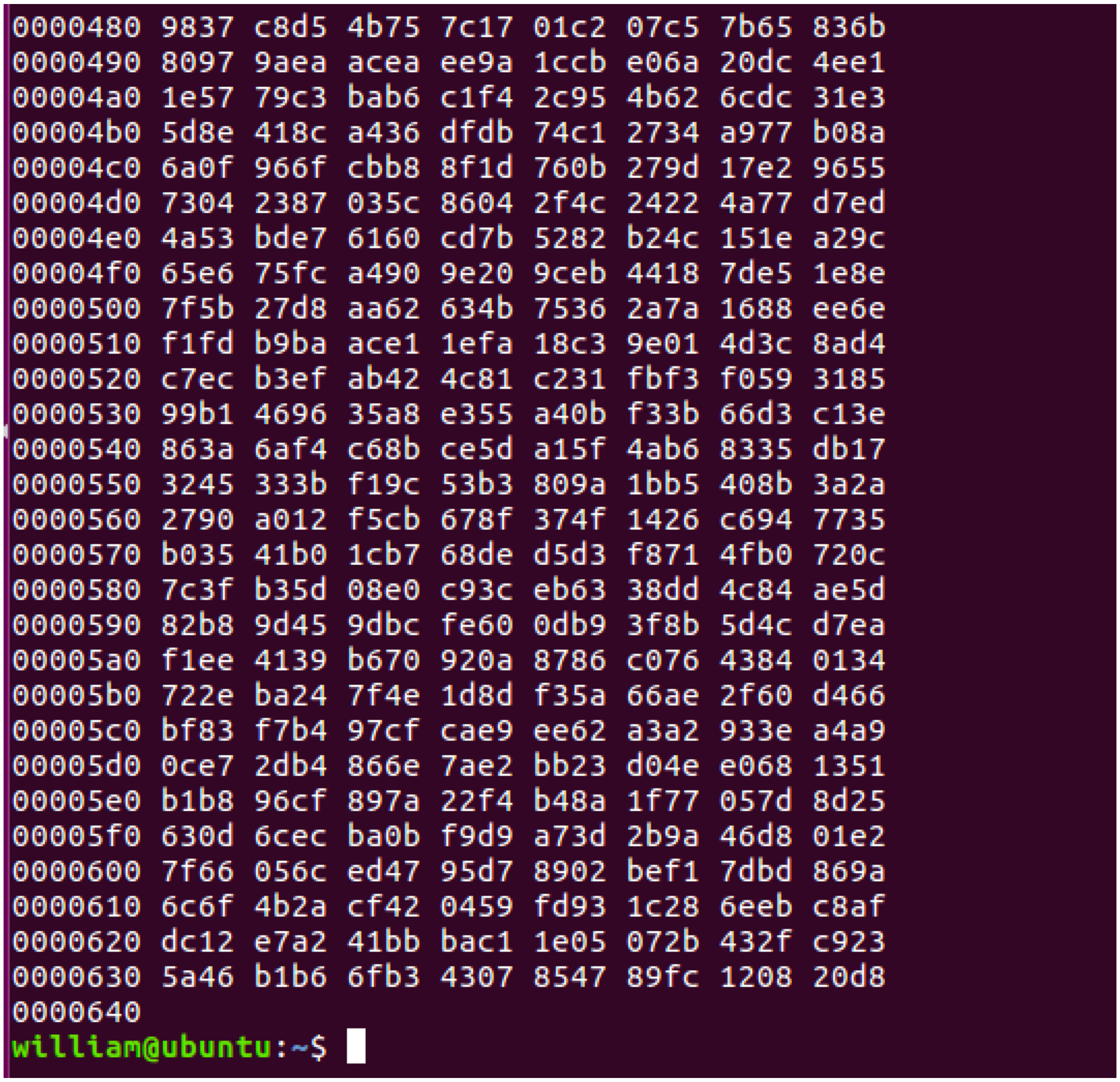
```
william@ubuntu:~/CryptoLabs/LAB3$ head -c 16 /dev/random | hexdump
0000000 4c64 da2a 18b7 2034 e004 9f39 5a54 feb4
0000010
william@ubuntu:~/CryptoLabs/LAB3$ cat /proc/sys/kernel/random/entropy_avail
101
william@ubuntu:~/CryptoLabs/LAB3$ head -c 16 /dev/random | hexdump
0000000 3fdc 0184 c1a4 2a16 b58b dc9f 7a28 8f67
0000010
william@ubuntu:~/CryptoLabs/LAB3$ cat /proc/sys/kernel/random/entropy_avail
58
william@ubuntu:~/CryptoLabs/LAB3$ head -c 16 /dev/random | hexdump
0000000 89fb 3e60 be87 974e 4763 9640 3850 af4a
0000010
william@ubuntu:~/CryptoLabs/LAB3$ cat /proc/sys/kernel/random/entropy_avail
34
```

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



```
william@ubuntu:~$ head -c 1600 /dev/urandom | hexdump
0000000 48f2 8351 9735 c5c3 8fd2 f9fb 3169 f2f0
0000010 566a 3631 c018 5f99 fae0 3d66 7755 0b1a
0000020 4ba4 37f2 a1b0 2696 8953 28ed 7d3e 8473
0000030 a10b 904f 20b6 1974 f75e b0ba 32e1 2ec0
0000040 2f8a cf43 24ae 8cbb f6b4 1ea0 01db 35cb
0000050 85d9 4f78 28cb eb1b 9f05 1f5f dbd3 5714
0000060 79da 3f65 7546 9f82 7b88 feae e8ba a302
0000070 b133 10c4 2f04 e8e0 4a97 ec19 bd2d c760
0000080 754f 6857 3dfa 776d 1227 eb34 8f54 bdd1
0000090 b757 1f6f 7d39 266d 332d 9ade d073 1635
00000a0 fbdf fade 3dd9 a1d4 1d80 9e01 b84d d9c3
00000b0 b343 dd46 840b e5b5 707f ecf3 30d4 eebd
00000c0 c433 a404 e7d1 17db f1e5 5b20 d771 5fc4
00000d0 ca7f 806c 7eea 25bc 4328 3816 43f8 a563
00000e0 0968 5d28 a562 b8d8 7750 1a11 50e3 e076
00000f0 0c24 bed1 62e2 d7bb 2922 4f28 187d 9853
0000100 2326 7e0c eadc 13d2 9ba4 631b 2d05 c717
0000110 ccb4 25db e6bc 5b61 e700 db80 2859 604f
0000120 fa05 a6c3 5a53 b0a4 8f7d 1e5e 1d53 b2b5
0000130 436d 463a dc91 6799 e576 e6d3 8f51 18ef
0000140 aa7a d643 fdb1 455d 1506 4e28 7e4b 3b3e
0000150 f2f1 265b e97f 8228 02f4 ee9d 7265 7e37
0000160 248a 5e0b 6c9b 66c2 00f7 770d 181f ebb7
0000170 2821 f0dd 7249 cdb5 6a3e 5fe3 0f84 50f1
0000180 4c1e b9d5 9dcf 9eb9 86cb 35cf f4ea e939
0000190 e1eb 8fae cbbf 84a3 0ffe 2d3f c8d2 152c
00001a0 a66c 2cfb bcb1 f0ea 89bf 7744 d5e0 788a
00001b0 7163 0643 ef2a f876 1b20 2f9a 9fc2 2e03
00001c0 7020 10f0 a599 c4ed bc11 1342 9d4e 4d05
00001d0 ef5c 057b 77cc ea77 5a2b 404b 9828 9d47
00001e0 877b 6423 daa9 95db 8c4d 804b 31cb d2d8
00001f0 03b3 7431 50ee a91c ef55 434d 898e ff75
0000200 729c 3dad 1772 619a a46e 8625 e45f 5217
0000210 3b4b 7c2b 99b2 3d4a d6de dafd a490 6d95
0000220 2fbd 7aa3 fe32 2866 a44a 0ef7 53ce 62c0
0000230 cd88 358d 5b69 424b 8398 afab a62d 01ce
0000240 fdab 4392 6d2f 377c 5f15 f6c2 8e02 c49d
0000250 64f6 0053 8b29 0226 e754 635d 3937 a964
0000260 036a 64a3 0b9d b2e4 9064 3b0e 4697 da47
0000270 80dc 4542 4f48 831f e5f0 ee72 2491 8f52
0000280 e953 2c13 7698 cbf9 f4d6 3227 92e9 6119
0000290 8811 08ff 84a5 f4e1 0616 61ce 4781 8c2e
00002a0 6622 2902 da27 9a7d 3232 d5e5 36c0 bcb5
00002b0 46cf 6ca5 35a5 65c2 eb75 a3b6 ed2d 48b1
00002c0 ea2c ee70 0c7c 942b bfd6 2cb7 c4ed f2b1
00002d0 979a 96c8 96b1 54d3 757a 956a ad48 0ddf
00002e0 e3fc dcc4 6efd e5ef 82e9 8e1d ccea f0a4
00002f0 d02a 02be 3c9a 9cfe 9300 3d23 3e54 5a19
```



```
0000480 9837 c8d5 4b75 7c17 01c2 07c5 7b65 836b
0000490 8097 9aea acea ee9a 1ccb e06a 20dc 4ee1
00004a0 1e57 79c3 bab6 c1f4 2c95 4b62 6cdc 31e3
00004b0 5d8e 418c a436 dfdb 74c1 2734 a977 b08a
00004c0 6a0f 966f cbb8 8f1d 760b 279d 17e2 9655
00004d0 7304 2387 035c 8604 2f4c 2422 4a77 d7ed
00004e0 4a53 bde7 6160 cd7b 5282 b24c 151e a29c
00004f0 65e6 75fc a490 9e20 9ceb 4418 7de5 1e8e
0000500 7f5b 27d8 aa62 634b 7536 2a7a 1688 ee6e
0000510 f1fd b9ba ace1 1efa 18c3 9e01 4d3c 8ad4
0000520 c7ec b3ef ab42 4c81 c231 fbf3 f059 3185
0000530 99b1 4696 35a8 e355 a40b f33b 66d3 c13e
0000540 863a 6af4 c68b ce5d a15f 4ab6 8335 db17
0000550 3245 333b f19c 53b3 809a 1bb5 408b 3a2a
0000560 2790 a012 f5cb 678f 374f 1426 c694 7735
0000570 b035 41b0 1cb7 68de d5d3 f871 4fb0 720c
0000580 7c3f b35d 08e0 c93c eb63 38dd 4c84 ae5d
0000590 82b8 9d45 9dbc fe60 0db9 3f8b 5d4c d7ea
00005a0 f1ee 4139 b670 920a 8786 c076 4384 0134
00005b0 722e ba24 7f4e 1d8d f35a 66ae 2f60 d466
00005c0 bf83 f7b4 97cf cae9 ee62 a3a2 933e a4a9
00005d0 0ce7 2db4 866e 7ae2 bb23 d04e e068 1351
00005e0 b1b8 96cf 897a 22f4 b48a 1f77 057d 8d25
00005f0 630d 6cec ba0b f9d9 a73d 2b9a 46d8 01e2
0000600 7f66 056c ed47 95d7 8902 bef1 7dbd 869a
0000610 6c6f 4b2a cf42 0459 fd93 1c28 6eeb c8af
0000620 dc12 e7a2 41bb bac1 1e05 072b 432f c923
0000630 5a46 b1b6 6fb3 4307 8547 89fc 1208 20d8
0000640
william@ubuntu:~$
```

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