

CIS 628 SEED Crypto_Random_Number Lab Report

Introduction to Cryptography

Syracuse University

Fall 2022

Homework 2

Part 2, Question (b)

Saket Kiran Thombre

SU ID: 899913802

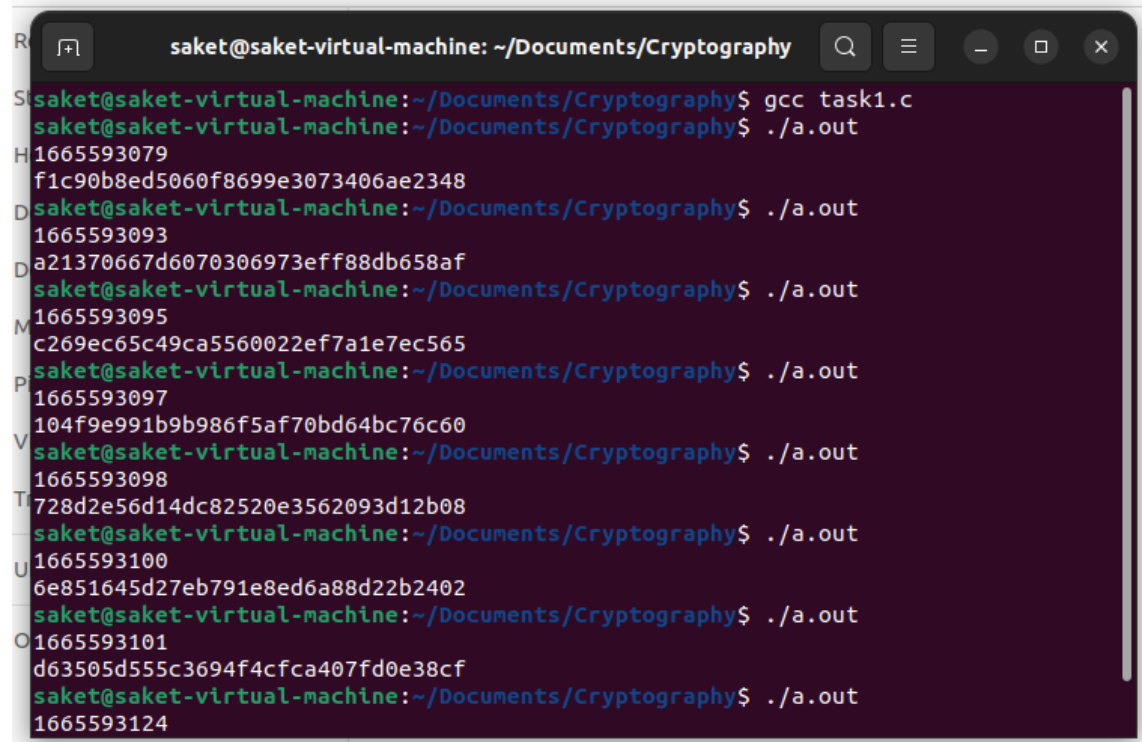
NetID: sthombre

Email: sthombre@syr.edu

Task 1: Generate Encryption Key in a Wrong Way:

Solution:

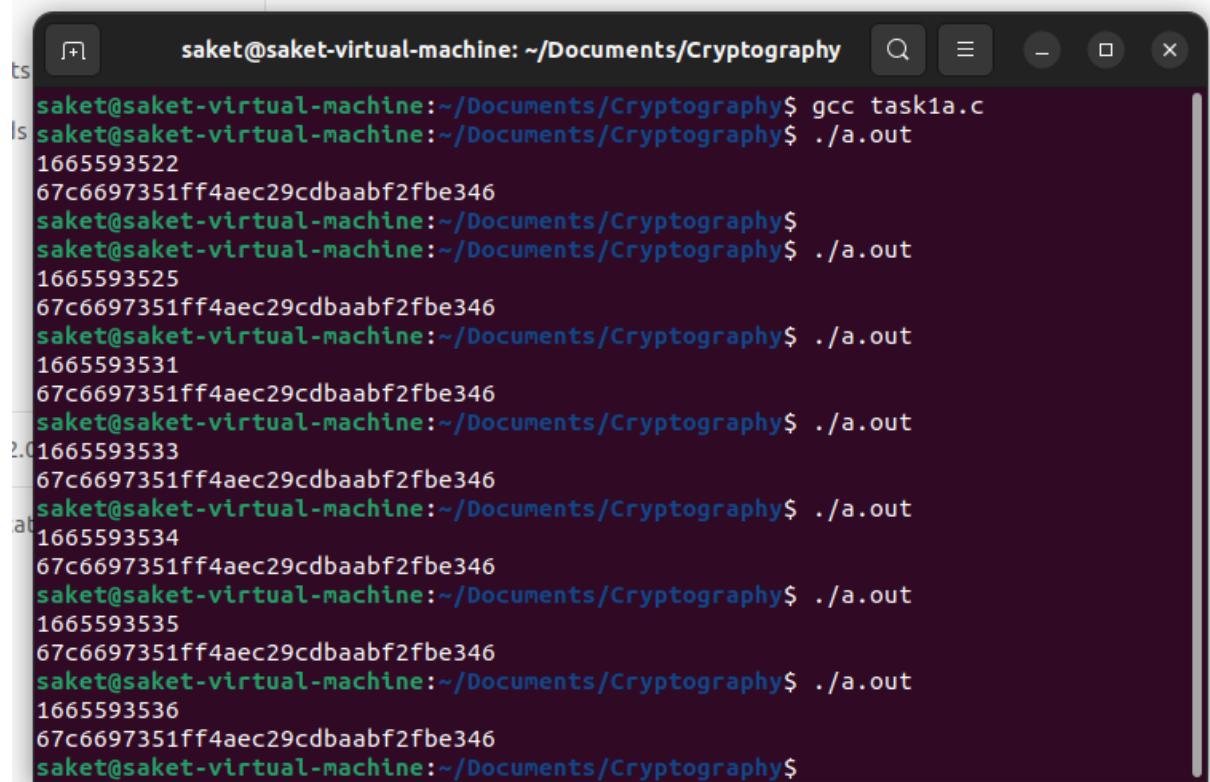
We are using a program that uses the current time as a seed for pseudo-random number generator:



```
saket@saket-virtual-machine: ~/Documents/Cryptography
$ gcc task1.c
$ ./a.out
H1665593079
f1c90b8ed5060f8699e3073406ae2348
D1665593093
a21370667d6070306973eff88db658af
D1665593095
c269ec65c49ca5560022ef7a1e7ec565
M1665593097
104f9e991b9b986f5af70bd64bc76c60
V1665593098
728d2e56d14dc82520e3562093d12b08
T1665593100
6e851645d27eb791e8ed6a88d22b2402
U1665593101
d63505d555c3694f4cfca407fd0e38cf
O1665593124
```

Figure 01: Show the output of program after running it multiple times.

In this output screenshot, we can see that the program task1.c is giving different sets of outputs after running multiple times. This occurs because we are considering the current time every time, we run the program. This will in turn generate a different seed in every iteration.



```
saket@saket-virtual-machine: ~/Documents/Cryptography
saket@saket-virtual-machine:~/Documents/Cryptography$ gcc task1a.c
saket@saket-virtual-machine:~/Documents/Cryptography$ ./a.out
1665593522
67c6697351ff4aec29cdbaabf2fbe346
saket@saket-virtual-machine:~/Documents/Cryptography$
saket@saket-virtual-machine:~/Documents/Cryptography$ ./a.out
1665593525
67c6697351ff4aec29cdbaabf2fbe346
saket@saket-virtual-machine:~/Documents/Cryptography$ ./a.out
1665593531
67c6697351ff4aec29cdbaabf2fbe346
saket@saket-virtual-machine:~/Documents/Cryptography$ ./a.out
1665593533
67c6697351ff4aec29cdbaabf2fbe346
saket@saket-virtual-machine:~/Documents/Cryptography$ ./a.out
1665593534
67c6697351ff4aec29cdbaabf2fbe346
saket@saket-virtual-machine:~/Documents/Cryptography$ ./a.out
1665593535
67c6697351ff4aec29cdbaabf2fbe346
saket@saket-virtual-machine:~/Documents/Cryptography$ ./a.out
1665593536
67c6697351ff4aec29cdbaabf2fbe346
saket@saket-virtual-machine:~/Documents/Cryptography$
```

Figure 02: After commenting the line “*srand(time(NULL));*” We again run the program multiple times.

Now we can see that the numbers generated are the same in multiple iterations.

Role of *srand()* function and *time()*:

This function *srand()* is used to initialize the pseudo-random number generator by passing the argument seed to all subsequent *rand()* calls.

Syntax of the *srand()* function in C language is:

void srand(unsigned int seed);

Where the seed value determines a particular sequence of random numbers when calling the *rand()* function. If the seed value remains the same as in this case, then the *rand()* function will not generate new sequence numbers.

In our example, we remove/comment the *srand()* function, and hence all the calls to *rand()* function generates the same sequence.

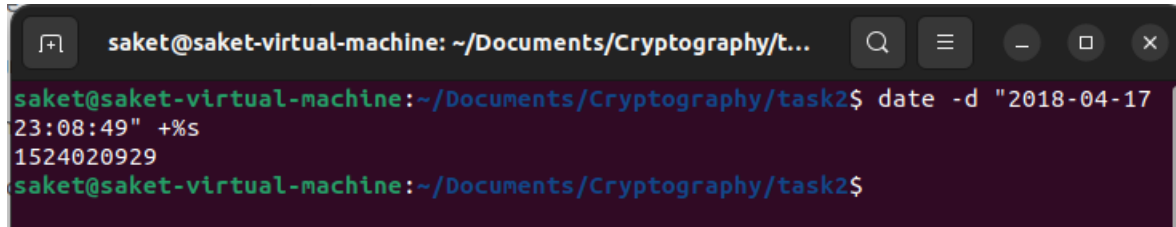
The *time()* function returns the number of seconds passed since Jan 01 1971. It is usually a very long integer whose value changes every second.

This function pair of *srand()* and *rand()* is usually paired with *time* function() and use its return value.

Task 2: Guessing the Key

Solution:

Finding the time stamp of “2018-04-17 23:08:49” we get



```
saket@saket-virtual-machine: ~/Documents/Cryptography/t...
saket@saket-virtual-machine:~/Documents/Cryptography/task2$ date -d "2018-04-17 23:08:49" +%s
1524020929
saket@saket-virtual-machine:~/Documents/Cryptography/task2$
```

We get the value “1524020929”

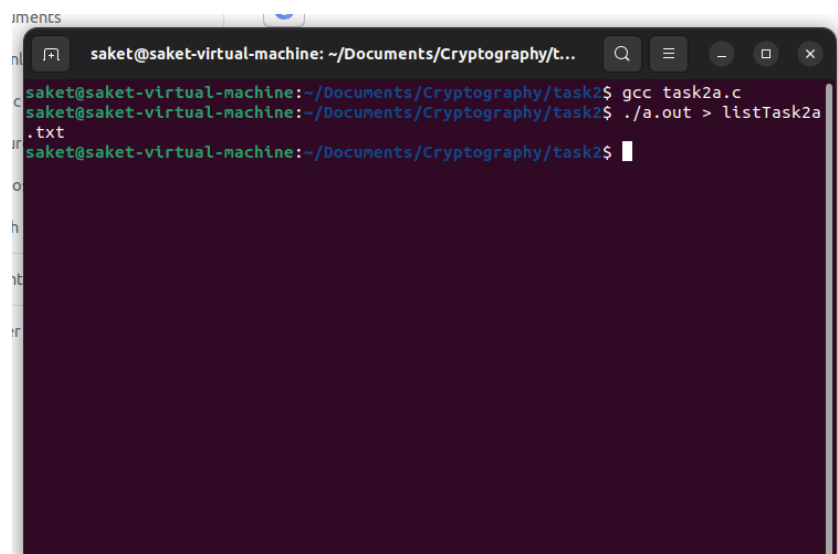
Now we find the value of all possible numbers generated within 2 hours of timestamp generation.

We use task2a.c to generate these values.

Excerpts from the c program to generate all values.

```
11 //Now we create a for loop to generate and traverse through all the timestamps within 2
hour window
12 for (time_t t = 1524020929 - 60 * 60 * 2; t < 1524020929; t++)
13 {
14     //initiating the srand() function
15     srand(t);
16     for (i = 0; i < KEYSIZE; i++)
17     {
18         seed[i] = rand() % 256;
19
20         //Printing all the seed values
21         printf("%.2x", (unsigned char)seed[i]);
22     }
23     printf("\n");
24 }
```

We now store the output in a txt file called listTask2a.txt



```
saket@saket-virtual-machine: ~/Documents/Cryptography/t...
saket@saket-virtual-machine:~/Documents/Cryptography/task2$ gcc task2a.c
saket@saket-virtual-machine:~/Documents/Cryptography/task2$ ./a.out > listTask2a.txt
saket@saket-virtual-machine:~/Documents/Cryptography/task2$
```

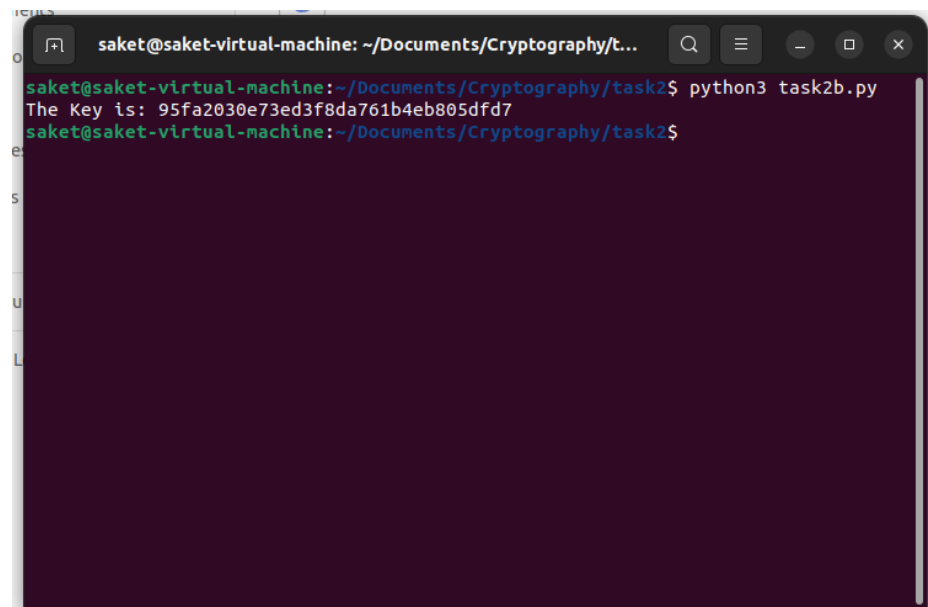
Further on, we now use brute force to crack the key from the generated txt file.

For this, we write a python script since we can directly use inbuilt “pycryptodome” library in the code “task2b.py”.

Excerpts from the python code:

```
8 with open('listTask2a.txt') as f:
9     possibleKeys = f.readlines()
10
11 for seed in keys:
12     #removing all trailing characters at the end of the srring
13     seed = seed.rstrip('\n')
14
15     #Now we return a new bytearray object initialized from a seed string
16     possibleKeys = bytearray.fromhex(seed)
17
18     #using AES cipher library to encrypt
19     cipher = AES.new(possibleKeys=possibleKeys, mode=AES.MODE_CBC, iv=iv)
20
21     guess = cipher.encrypt(data)
22
23     if guess == ciphertext:
24         print("The Key is:", seed)
25         exit(0)
26
```

The python code traverses through all the values present in the txt file and encrypts them one by one. Subsequently it compares the values with the given key and provides the matching output.

A screenshot of a terminal window titled "sakat@saket-virtual-machine: ~/Documents/Cryptography/t...". The terminal shows the command "python3 task2b.py" being executed. The output is "The Key is: 95fa2030e73ed3f8da761b4eb805dfd7". The prompt then changes to "sakat@saket-virtual-machine:~/Documents/Cryptography/task2\$".

```
sakat@saket-virtual-machine:~/Documents/Cryptography/task2$ python3 task2b.py
The Key is: 95fa2030e73ed3f8da761b4eb805dfd7
sakat@saket-virtual-machine:~/Documents/Cryptography/task2$
```

Thus, we find the key which is “95fa2030e73ed3f8da761b4eb805dfd7”

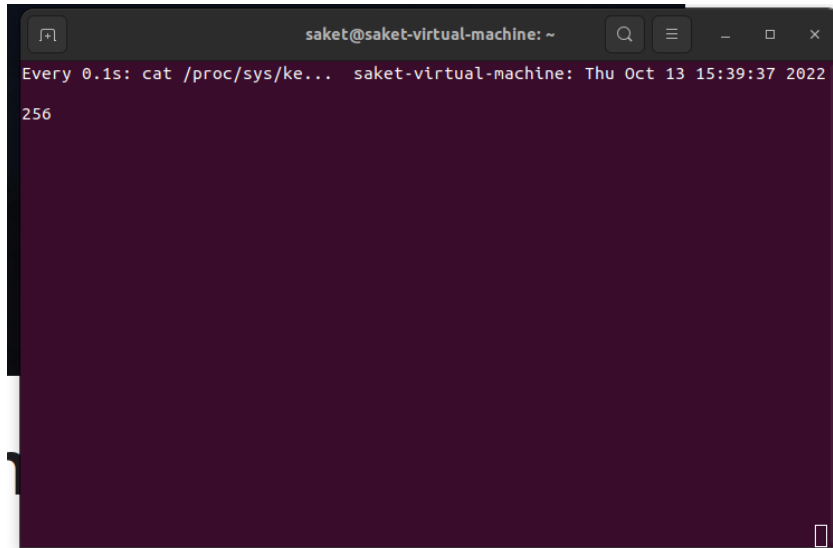
Hence, we can say that Bob has successfully generated the correct key using Alice’s file timestamp for her tax return PDF file.

Task 3: Measure the Entropy of Kernel

Solution:

We ran the following command Ubuntu, and we get the following output

```
"$ watch -n .1 cat /proc/sys/kernel/random/entropy_avail"
```

A terminal window titled 'saket@saket-virtual-machine: ~' with search, menu, and window control icons. The terminal shows the command 'Every 0.1s: cat /proc/sys/kernel/random/entropy_avail' and the output '256'. The terminal background is dark purple with white text. The window title bar is dark grey with standard Linux window controls (minimize, maximize, close) and a search icon.

Entropy is a way to measure the difficulty it needs to guess the strength of passwords using a secret value drawn from a pool.

A Linux machine pulls entropy from the last few timestamp digits fired by mouse, keyboards, disk drive or other hardware events. These new data points are then mixed up with the old points in the entropy pool resulting in even higher entropy. A keystroke or a mouse movement eventually adds up a couple of bits to the entropy in the pool, however it also tracks the number of bytes you pull from the function. In addition, as the network traffic rises, many network drivers may turn ON or OFF which in turn reduces the entropy of the pool.

There are many untapped entropy sources like temperature sensors, fan speed, audio inputs from surrounding etc. that can increase the entropy of the pool significantly. However, this can be only implemented in normal machines but not in virtual machines. For virtual machines, we need to use a pseudo-random number generator which will have a lesser entropy.

Task 4: Get Pseudo Random Numbers from /dev/random

Solution:

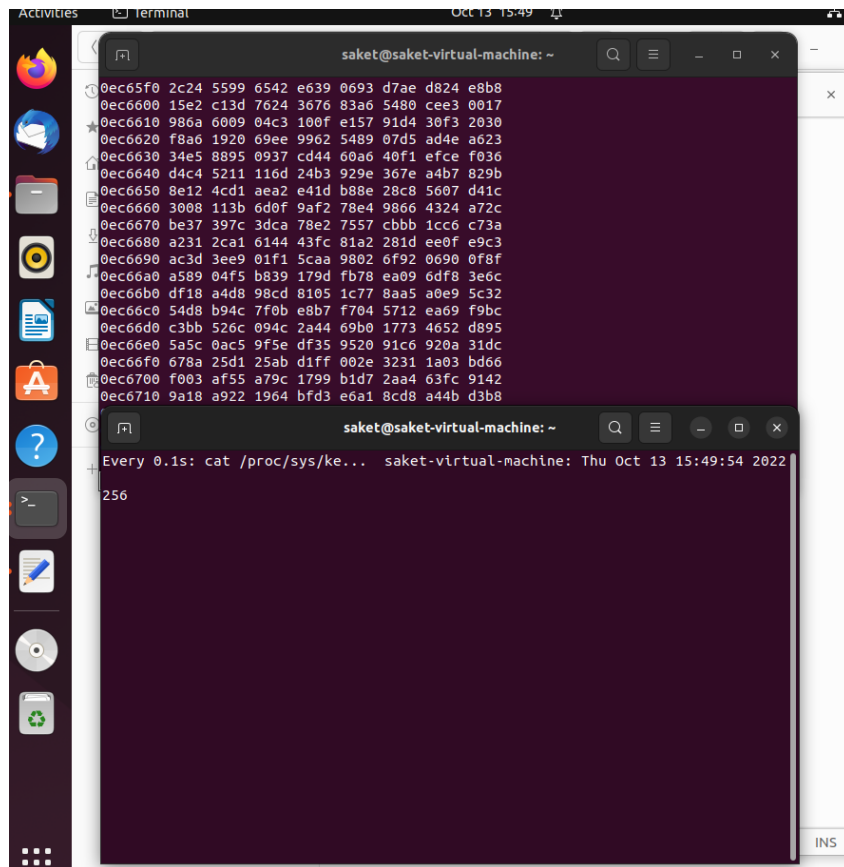
First, we run the command

```
"$ cat /dev/random | hexdump"
```

And then we run

“\$ watch -n .1 cat /proc/sys/kernel/random/entropy_avail”

We get the following output



```
saket@saket-virtual-machine: ~  
0ec65f0 2c24 5599 6542 e639 0693 d7ae d824 e8b8  
0ec6600 15e2 c13d 7624 3676 83a6 5480 cee3 0017  
0ec6610 986a 6009 04c3 100f e157 91d4 30f3 2030  
0ec6620 f8a6 1920 69ee 9962 5489 07d5 ad4e a623  
0ec6630 34e5 8895 0937 cd44 60a6 40f1 efce f036  
0ec6640 d4c4 5211 116d 24b3 929e 367e a4b7 829b  
0ec6650 8e12 4cd1 aea2 e41d b88e 28c8 5607 d41c  
0ec6660 3008 113b 6d0f 9af2 78e4 9866 4324 a72c  
0ec6670 be37 397c 3dca 78e2 7557 cbbb 1cc6 c73a  
0ec6680 a231 2ca1 6144 43fc 81a2 281d ee0f e9c3  
0ec6690 ac3d 3ee9 01f1 5caa 9802 6f92 0690 0f8f  
0ec66a0 a589 04f5 b839 179d fb78 ea09 6df8 3e0c  
0ec66b0 df18 a4d8 98cd 8105 1c77 8aa5 a0e9 5c32  
0ec66c0 54d8 b94c 7f0b e8b7 f704 5712 ea69 f9bc  
0ec66d0 c3bb 526c 094c 2a44 69b0 1773 4652 d895  
0ec66e0 5a5c 0ac5 9f5e df35 9520 91c6 920a 31dc  
0ec66f0 678a 25d1 25ab diff 002e 3231 1a03 bd66  
0ec6700 f003 af55 a79c 1799 b1d7 2aa4 63fc 9142  
0ec6710 9a18 a922 1964 bfd3 e6a1 8cd8 a44b d3b8  
  
Every 0.1s: cat /proc/sys/ke... saket-virtual-machine: Thu Oct 13 15:49:54 2022  
256
```

When we run the hexdump command, we can see the entropy slowly decrease to zero. When we do a mouse movement, we can see temporarily that the entropy pool stops decreasing and start again.

When we stop the hexdump command, we can see that the entropy pool starts building up again and slowly increases as you do mouse movement and keystrokes.

Question: If a server uses /dev/random to generate the random session key with a client. Please describe how you can launch a Denial-Of-Service (DOS) attack on such a server.

Solution:

As we know that /dev/random is used to generate random numbers with highest possible entropy, it also gets blocked when the pool gets exhausted.

The attack relies on this property of /dev/random. The attacker will request a huge amount of session IDs which will require private keys to be generated. These private keys are generated using the random numbers. This by effect reduces the pool entropy and may also stop the service.

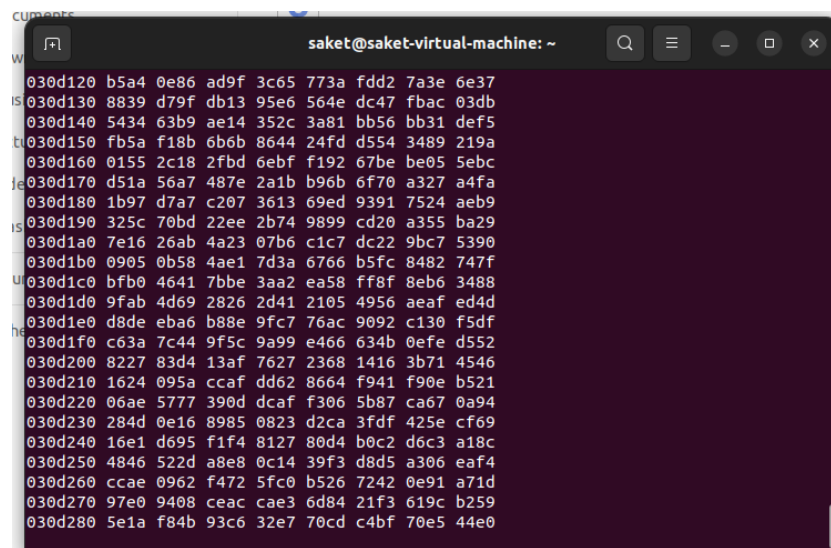
`/dev/random` will block all request till the entropy pool is filled again with the intention of providing random numbers with the highest possible entropy.

Task 5: Get Random Numbers from `/dev/urandom`

Solution:

First, we run the command for `/dev/urandom`

We get the following output



```
saket@saket-virtual-machine: ~  
030d120 b5a4 0e86 ad9f 3c65 773a fdd2 7a3e 6e37  
030d130 8839 d79f db13 95e6 564e dc47 fbac 03db  
030d140 5434 63b9 ae14 352c 3a81 bb56 bb31 def5  
030d150 fb5a f18b 6b6b 8644 24fd d554 3489 219a  
030d160 0155 2c18 2fbd 6ebf f192 67be be05 5ebc  
030d170 d51a 56a7 487e 2a1b b96b 6f70 a327 a4fa  
030d180 1b97 d7a7 c207 3613 69ed 9391 7524 aeb9  
030d190 325c 70bd 22ee 2b74 9899 cd20 a355 ba29  
030d1a0 7e16 26ab 4a23 07b6 c1c7 dc22 9bc7 5390  
030d1b0 0905 0b58 4ae1 7d3a 6766 b5fc 8482 747f  
030d1c0 bfb0 4641 7bbe 3aa2 ea58 ff8f 8eb6 3488  
030d1d0 9fab 4d69 2826 2d41 2105 4956 aeaf ed4d  
030d1e0 d8de eba6 b88e 9fc7 76ac 9092 c130 f5df  
030d1f0 c63a 7c44 9f5c 9a99 e466 634b 0efe d552  
030d200 8227 83d4 13af 7627 2368 1416 3b71 4546  
030d210 1624 095a ccaf dd62 8664 f941 f90e b521  
030d220 06ae 5777 390d dcaf f306 5b87 ca67 0a94  
030d230 284d 0e16 8985 0823 d2ca 3fdf 425e cf69  
030d240 16e1 d695 f1f4 8127 80d4 b0c2 d6c3 a18c  
030d250 4846 522d a8e8 0c14 39f3 d8d5 a306 eaf4  
030d260 ccae 0962 f472 5fc0 b526 7242 0e91 a71d  
030d270 97e0 9408 ceac cae3 6d84 21f3 619c b259  
030d280 5e1a f84b 93c6 32e7 70cd c4bf 70e5 44e0
```

LINUX/UNIX systems have 2 main types of random number generators namely, `/dev/random` and `/dev/urandom`.

These generators gather external noise from hardware drivers and collect it in a entropy pool. This entropy pool is further used to create random numbers.

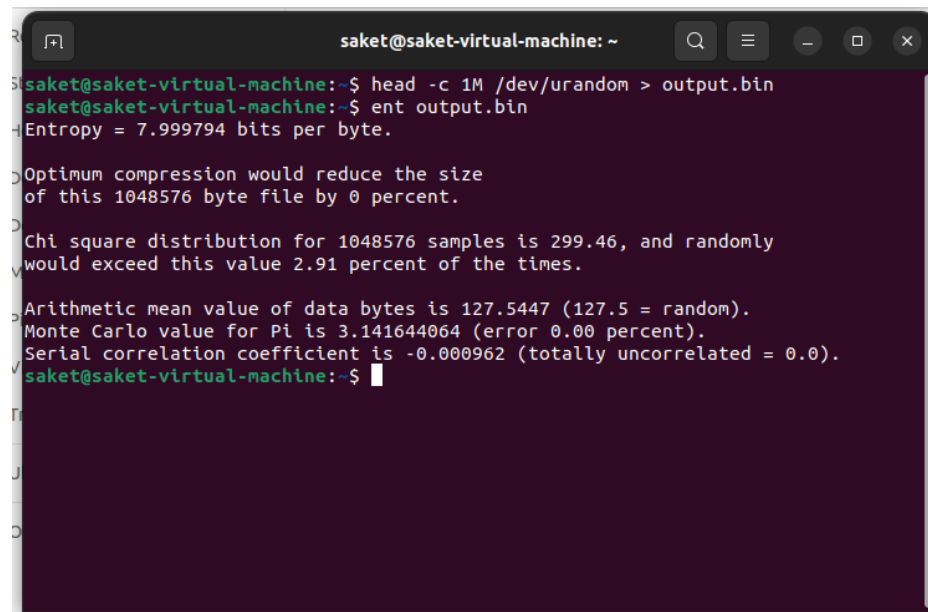
However, when there is a need to generate high number of random numbers, the pool might get exhausted and, in that case, `/dev/random` function will stop but `/dev/urandom` will reuse the pool and keep generating random numbers. This in turn reduces the entropy for `/dev/urandom`.

When we run a hexdump for `/dev/urandom`, the entropy starts reducing slowly, when we generate some mouse movements, the entropy seems to increase and become stable momentarily and once the mouse movements stop, its starts decreasing again.

In the second part, we generate a 1MB file consisting of pseudo-random numbers from `/dev/urandom` and we save it in `output.bin` file.

Then we run “ent” on this file to check its entropy.

We get the following output



```
saket@saket-virtual-machine: ~  
saket@saket-virtual-machine:~$ head -c 1M /dev/urandom > output.bin  
saket@saket-virtual-machine:~$ ent output.bin  
Entropy = 7.999794 bits per byte.  
  
Optimum compression would reduce the size  
of this 1048576 byte file by 0 percent.  
  
Chi square distribution for 1048576 samples is 299.46, and randomly  
would exceed this value 2.91 percent of the times.  
  
Arithmetic mean value of data bytes is 127.5447 (127.5 = random).  
Monte Carlo value for Pi is 3.141644064 (error 0.00 percent).  
Serial correlation coefficient is -0.000962 (totally uncorrelated = 0.0).  
saket@saket-virtual-machine:~$
```

“ent” performs produces standard output by performing various tests on a stream of bytes.

In our case, we look for the entropy, it is the information density of the contents of a file which are expressed as a number of bits per character.

In our case, we get the Entropy = 7.999794. this value of entropy is supposed to be very high. This means that the random numbers to be generated are going to be of a good value.

Along with entropy, “ent” function also provides optimum compression, chi square, arithmetic mean, Monte Carlo value and serial correlation coefficient.

In part 3

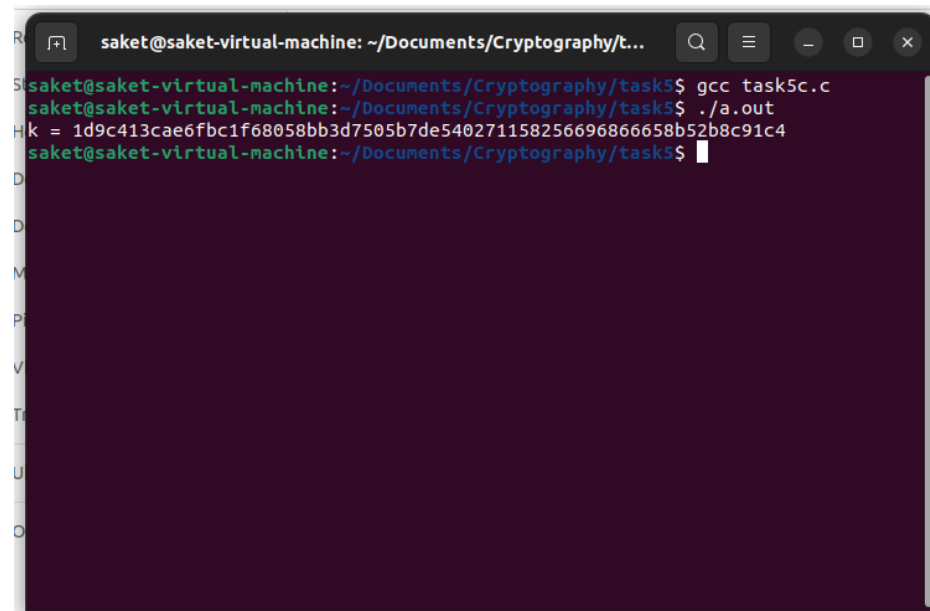
We are given a code snippet for 128 bits, and we need to convert it to 256 bits and generate random number.

So, we modify the code to get 256-bit number



```
task5c.c  
~/Documents/Cryptography/task5  
Save  
1 #include <stdio.h>  
2 #include <stdlib.h>  
3 #define LEN 32 // 256 bits  
4  
5 int main()  
6 {  
7     unsigned char *key = (unsigned char *)malloc(sizeof(unsigned char) * LEN);  
8     FILE *random = fopen("/dev/urandom", "r");  
9     fread(key, sizeof(unsigned char) * LEN, 1, random);  
10    fclose(random);  
}
```

After running the updated code, we get the following output

A terminal window titled 'saket@saket-virtual-machine: ~/Documents/Cryptography/t...' with standard window controls. The terminal shows the following commands and output:

```
saket@saket-virtual-machine:~/Documents/Cryptography/task5$ gcc task5c.c
saket@saket-virtual-machine:~/Documents/Cryptography/task5$ ./a.out
Hk = 1d9c413cae6fbc1f68058bb3d7505b7de540271158256696866658b52b8c91c4
saket@saket-virtual-machine:~/Documents/Cryptography/task5$
```