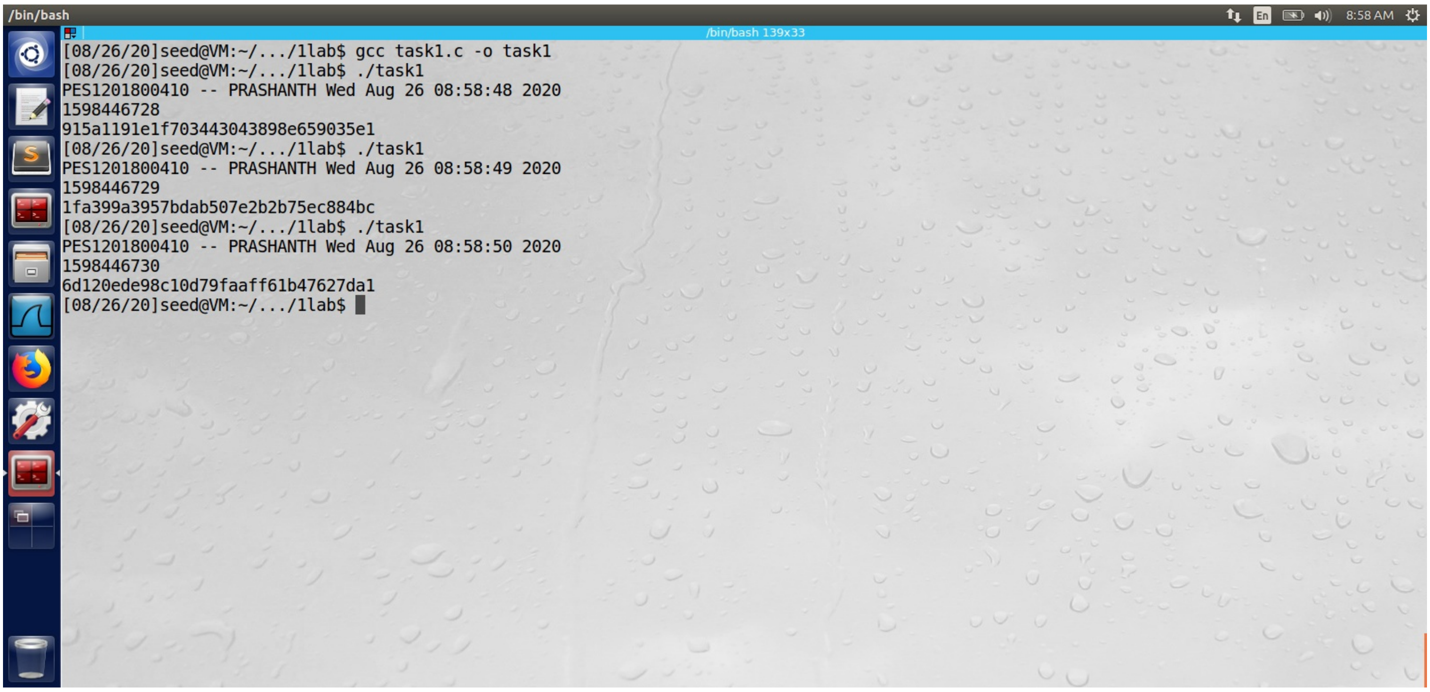
Pseudo Random Number Generation

Task 1: Generate Encryption Key in a Wrong Way

step 1

```
 $gcc task1.c -o task1
$./task1
$./task1
$./task1
```

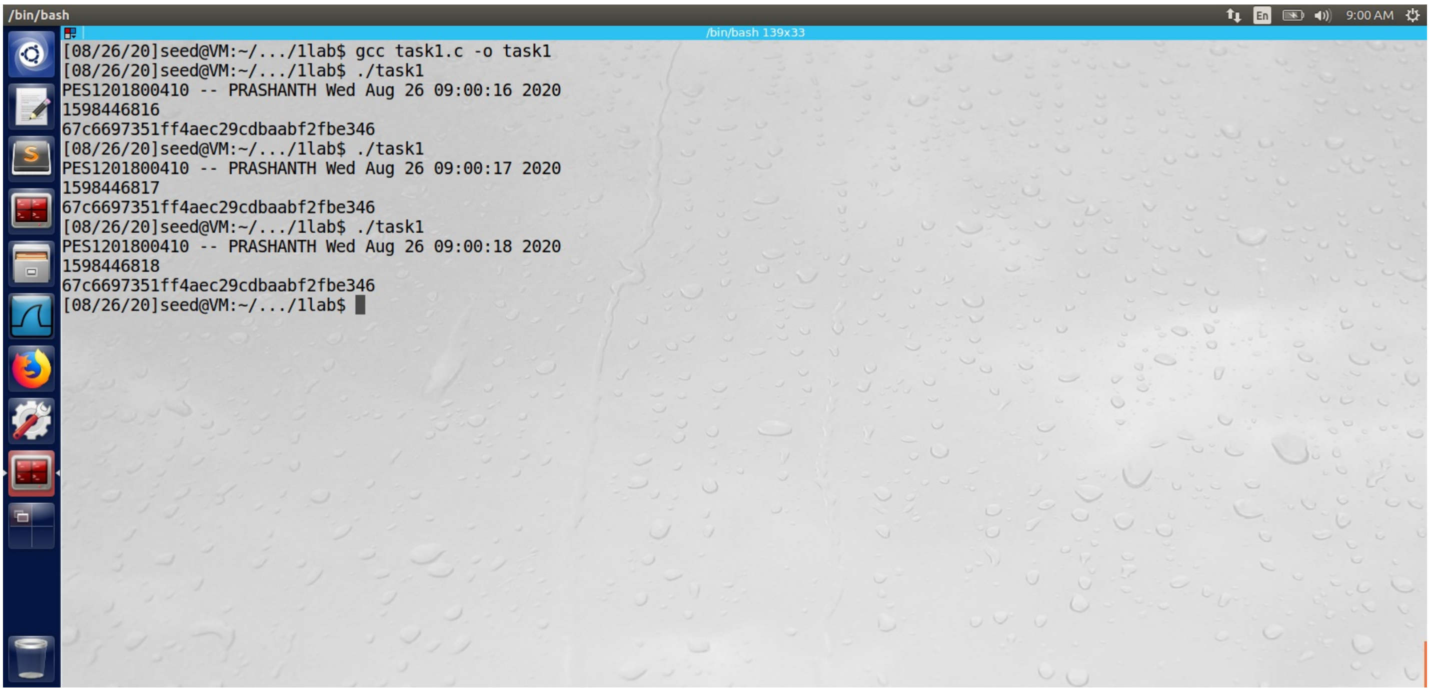


observations

- based on the value of srand(that is seed value) pseudo random number is generated
- 2 random number generated at a same time will generate same number which is a wrong way to generate a truly random number

step 2

> comment the srand(time(NULL)) and excecute the same command of step one



observations

- on commenting the srand(time(NULL)) the generated number will always be same
- there fore its not a truly random number generator but its a pseudo random number generator

Task 2: Guessing the Key

step 1

```
$ date -d "2018-04-17 21:08:49" +%s
1523979529

$ date -d "2018-04-17 23:08:49" +%s
1523986729
```

step 2

```
$ gcc task2.c -o task2
$ ./task2
```
```


 - keys.txt file is created based on the time it value 1 and value 2
 - all the possible keys that could be generated between the time is in keys.txt

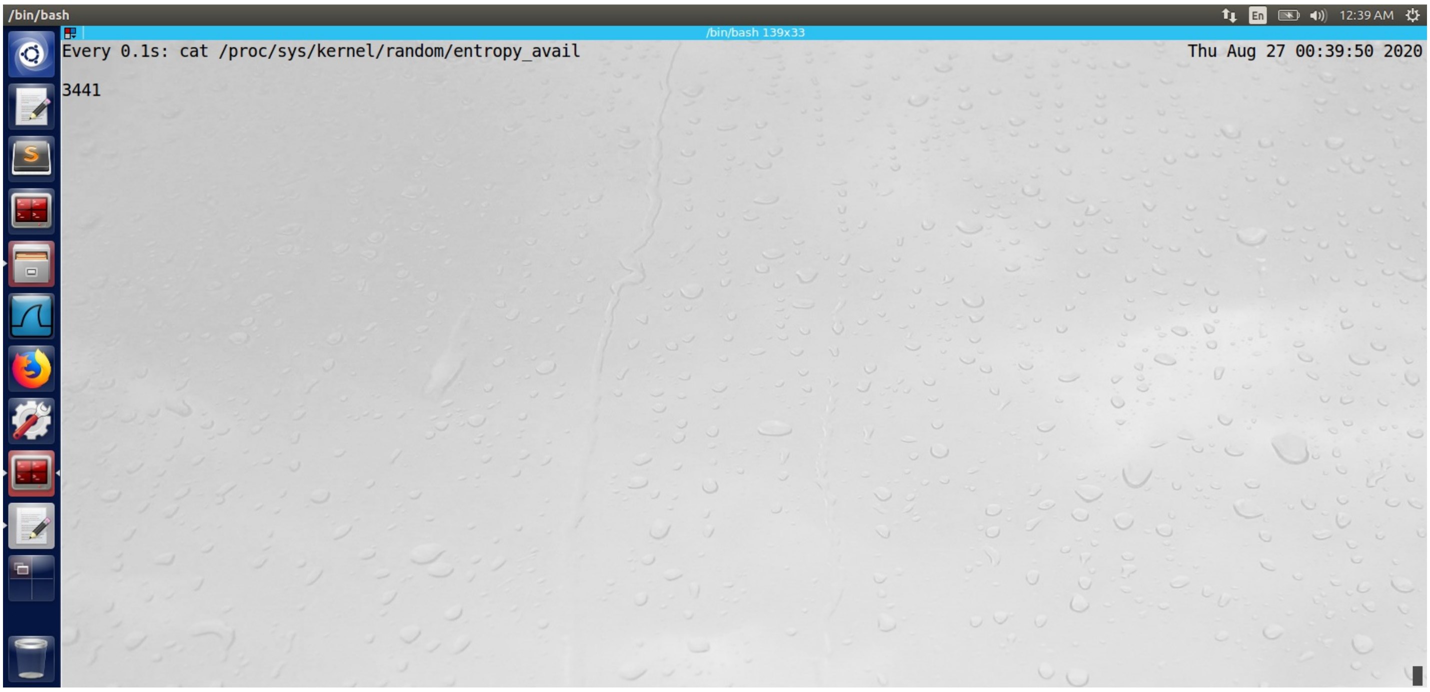
###### step 3
```sh
$ python decrypt.py
```

observations

- we see that giving time as the seed value is not a truly random value
- therefore it is not wise to use time as a seed value to generate truly random number
- as in the above case we see that we are able to crack the cipher text by knowing the time gap

Task 3: Measure the Entropy of Kernel

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

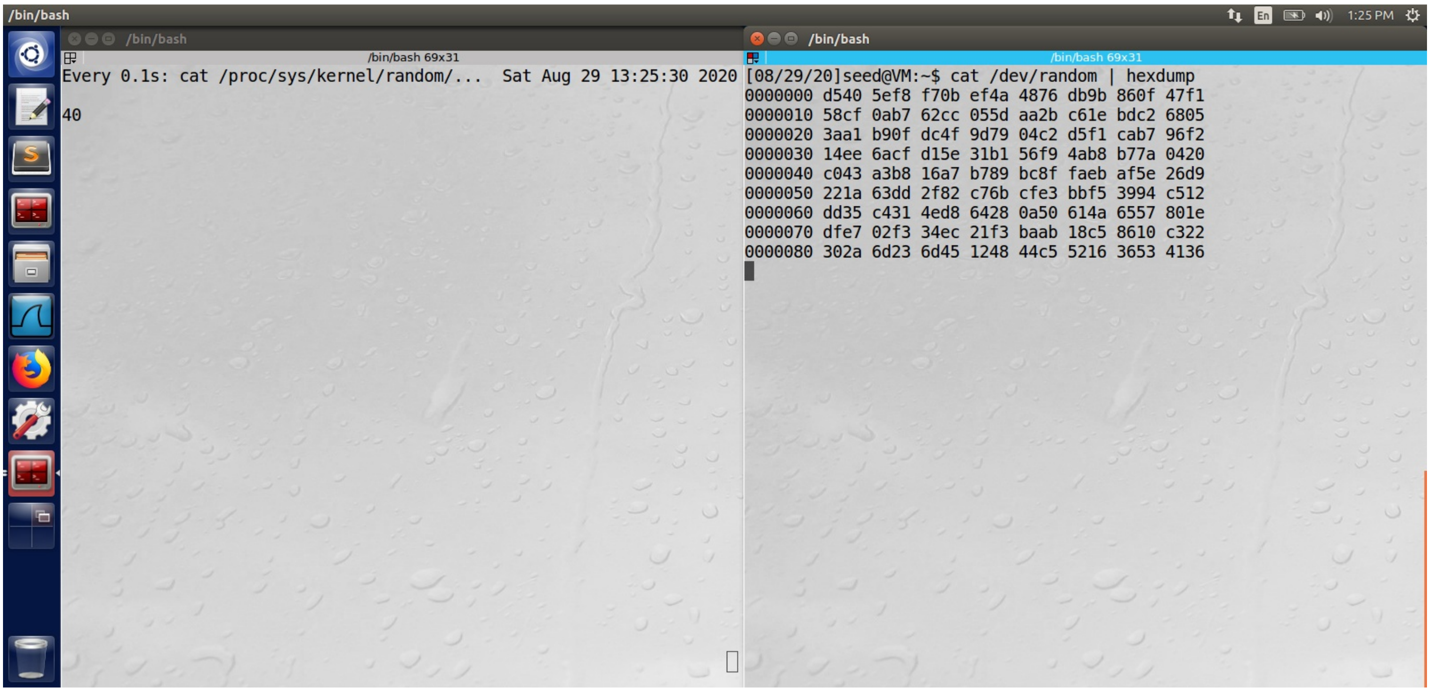


- the value keeps increasing on pressing any key or moving mouse

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

```
$ watch -n .1 cat /proc/sys/kernel/random/entropy_avail
$ cat /dev/random | hexdump
```

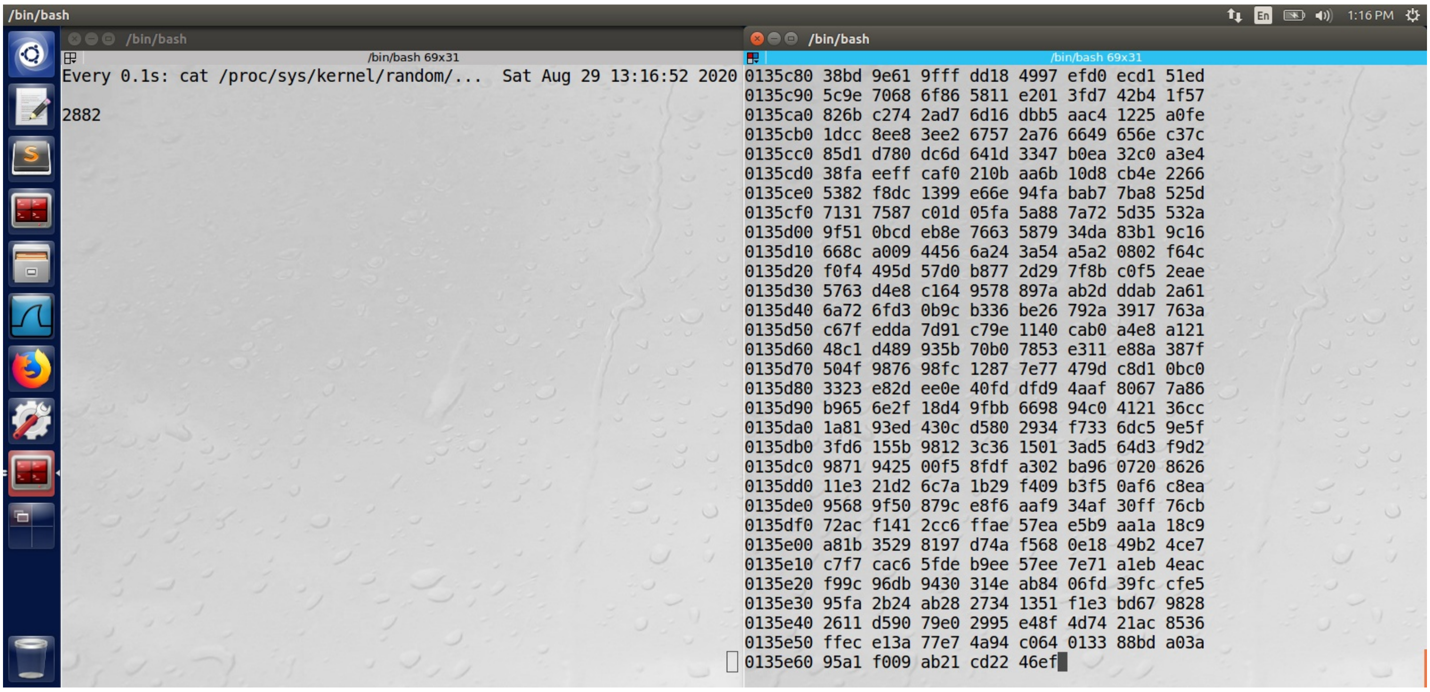
- entropy_avail value changes continoulsy from 0 to 60-70 (as fas as i could see)
- cat /dev/random prints random numbers only when there is enough entropy



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

step 1

```
$ watch -n .1 cat /proc/sys/kernel/random/entropy_avail
$ cat /dev/urandom | hexdump
```



- Both /dev/random and /dev/urandom use the random data from the pool to generate pseudo random numbers.
- When the entropy is not sufficient, /dev/random will pause, while /dev/urandom will keep generating new numbers.
- as stated in the lab manual
- it genarates continuous stream of random numbers which intern changes the entropy

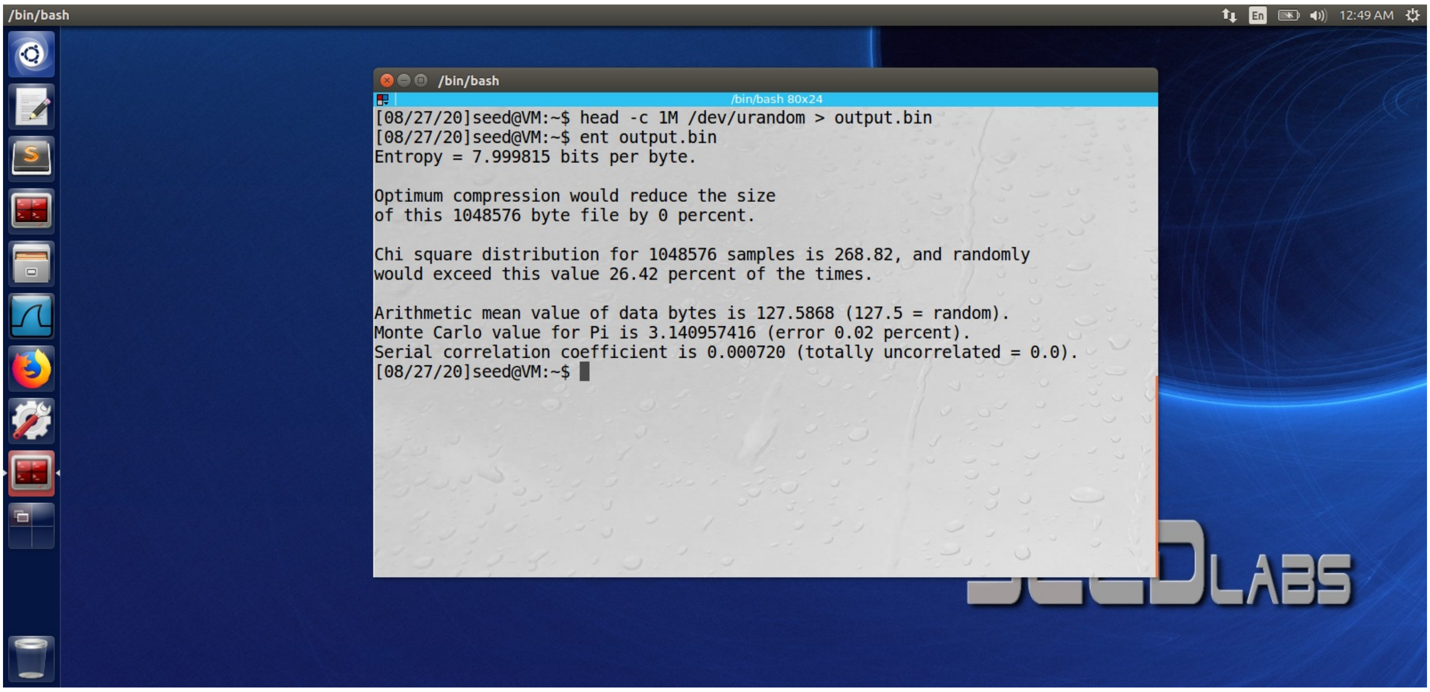
Step 2: Measure the quality of the random number using a tool called ent.

```
$ head -c 1M /dev/urandom > output.bin
$ ent output.bin
Entropy = 7.999815 bits per byte.

Optimum compression would reduce the size
of this 1048576 byte file by 0 percent.

Chi square distribution for 1048576 samples is 268.82, and randomly
would exceed this value 26.42 percent of the times.

Arithmetic mean value of data bytes is 127.5868 (127.5 = random).
Monte Carlo value for Pi is 3.140957416 (error 0.02 percent).
Serial correlation coefficient is 0.000720 (totally uncorrelated = 0.0).
```

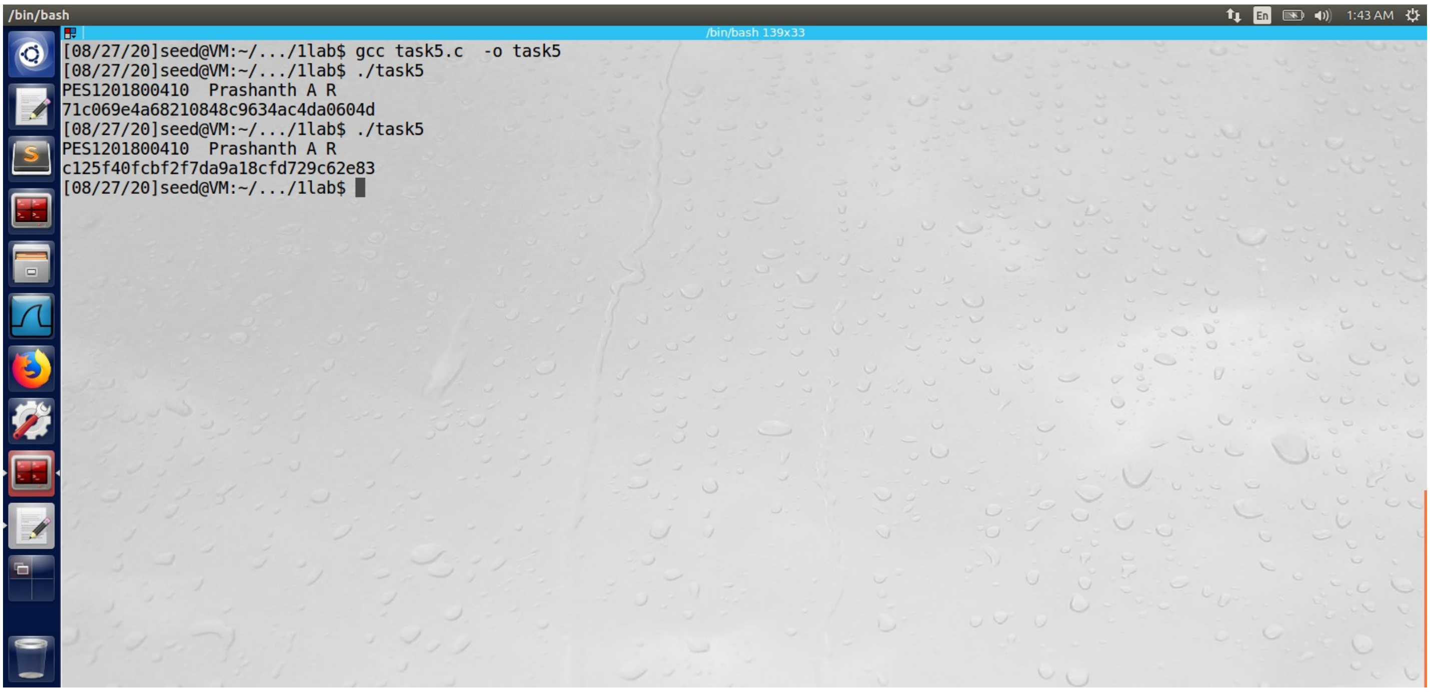


- entropy = 7.999.... bits per bytes indicate that the file is extremely dense in information—essentially random. Hence, compression of the file is unlikely to reduce its size.
- as it is random compression is not possible ie 0% compression on the file
- The chi-square test is the most commonly used test for the randomness of data

- 26.42% indicates how frequently a truly random sequence would exceed the value calculated
- Its a simple arithmetic mean of all bytes . Value above 127.5 indicates its random
- Monte Carlo value for Pi will be close to Pi value if it is random .
- In our case its just 0.02% therefore we can consider it as random
- Serial correlation coefficient -- This quantity measures the extent to which each byte in the file depends upon the previous byte
- for a random number this value will be close to 0

Step 3: The program from Task 1 can be modified to write a new 128 bit key using /dev/urandom

```
 $ gcc task5.c -o task5
$ ./task5
$ ./task5
```



- the values are truly random numbers
- values are read from /dev/urandom therefore they are truly random numbers