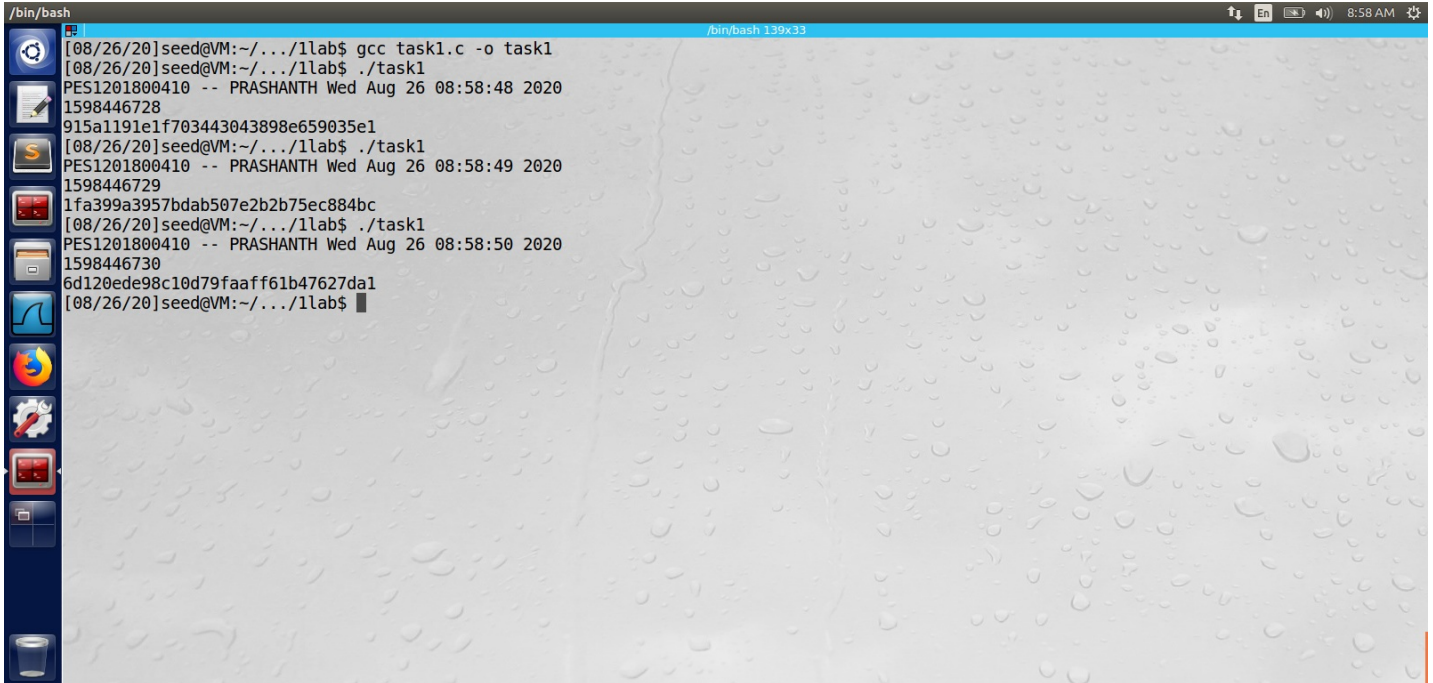


Pseudo Random Number Generation

Task 1: Generate Encryption Key in a Wrong Way

step 1

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



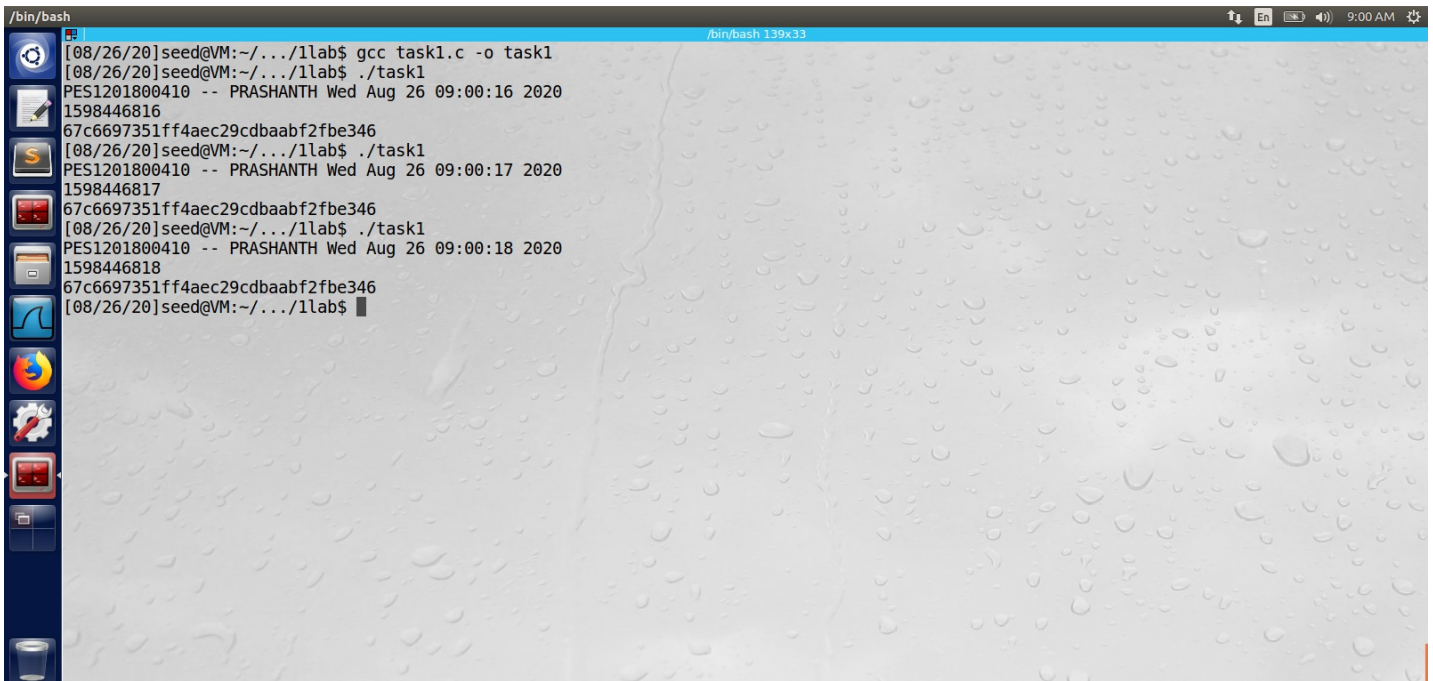
```
/bin/bash
[08/26/20]seed@VM:~/.../1lab$ gcc task1.c -o task1
[08/26/20]seed@VM:~/.../1lab$ ./task1
PES1201800410 -- PRASHANTH Wed Aug 26 08:58:48 2020
1598446728
915a1191e1f703443043898e659035e1
[08/26/20]seed@VM:~/.../1lab$ ./task1
PES1201800410 -- PRASHANTH Wed Aug 26 08:58:49 2020
1598446728
1fa399a3957bdab507e2b2b75ec884bc
[08/26/20]seed@VM:~/.../1lab$ ./task1
PES1201800410 -- PRASHANTH Wed Aug 26 08:58:50 2020
1598446730
6d120ede98c10d79faaff61b47627da1
[08/26/20]seed@VM:~/.../1lab$
```

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 execute the same command of step one



```
/bin/bash
[08/26/20]seed@VM:~/.../1lab$ gcc task1.c -o task1
[08/26/20]seed@VM:~/.../1lab$ ./task1
PES1201800410 -- PRASHANTH Wed Aug 26 09:00:16 2020
1598446816
67c6697351ff4aec29cdbaabf2fbe346
[08/26/20]seed@VM:~/.../1lab$ ./task1
PES1201800410 -- PRASHANTH Wed Aug 26 09:00:17 2020
1598446817
67c6697351ff4aec29cdbaabf2fbe346
[08/26/20]seed@VM:~/.../1lab$ ./task1
PES1201800410 -- PRASHANTH Wed Aug 26 09:00:18 2020
1598446818
67c6697351ff4aec29cdbaabf2fbe346
[08/26/20]seed@VM:~/.../1lab$
```

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

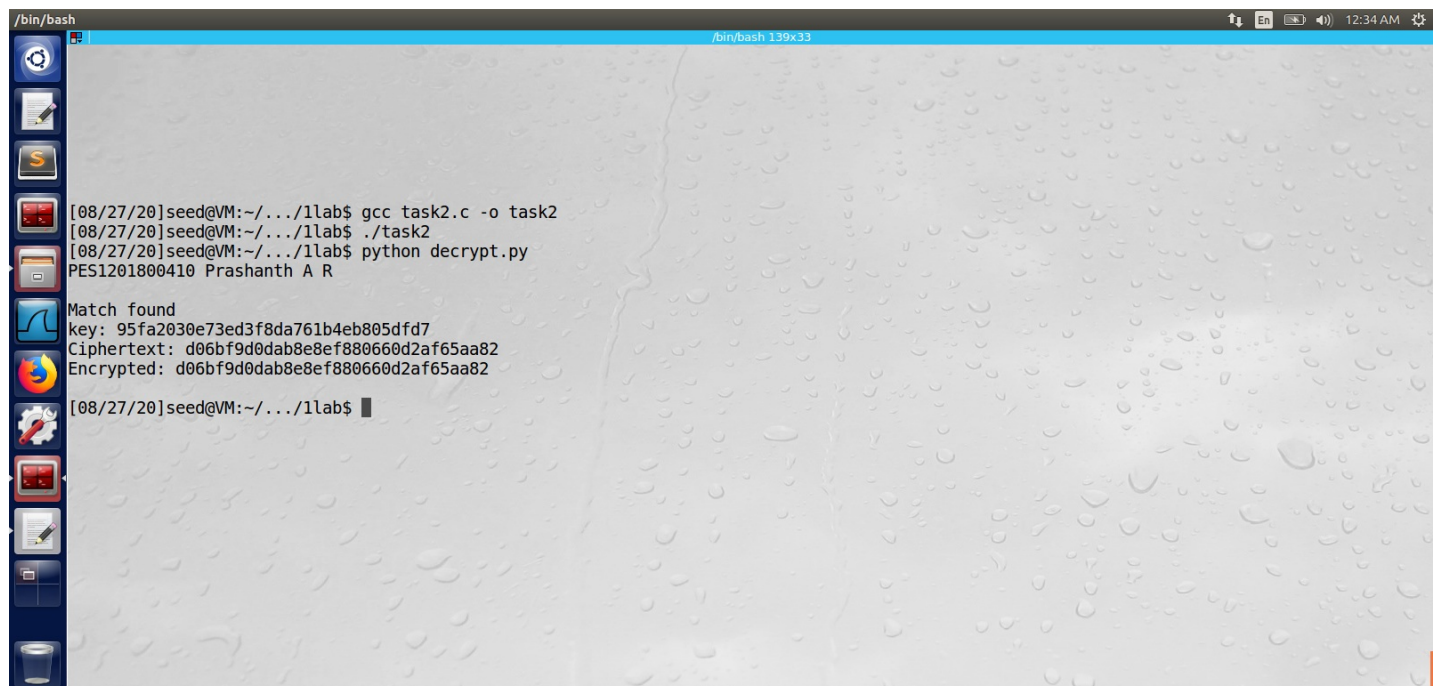
$ date -d "2018-04-17 23:08:49" +%s
1523986729
```

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
```



```
/bin/bash
/bin/bash 139x33

[08/27/20]seed@VM:~/.../1lab$ gcc task2.c -o task2
[08/27/20]seed@VM:~/.../1lab$ ./task2
[08/27/20]seed@VM:~/.../1lab$ python decrypt.py
PES1201800410 Prashanth A R

Match found
key: 95fa2030e73ed3f8da761b4eb805dfd7
Ciphertext: d06bf9d0dab8e8ef880660d2af65aa82
Encrypted: d06bf9d0dab8e8ef880660d2af65aa82

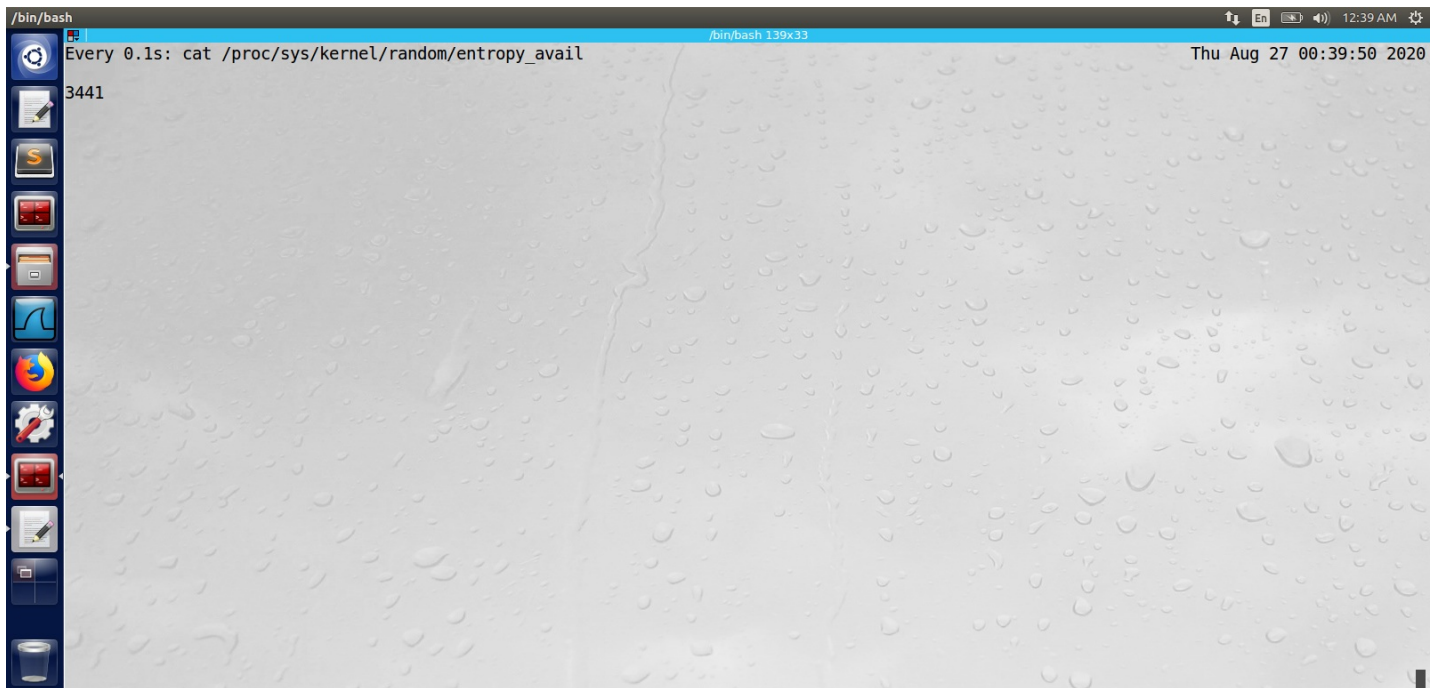
[08/27/20]seed@VM:~/.../1lab$
```

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
```

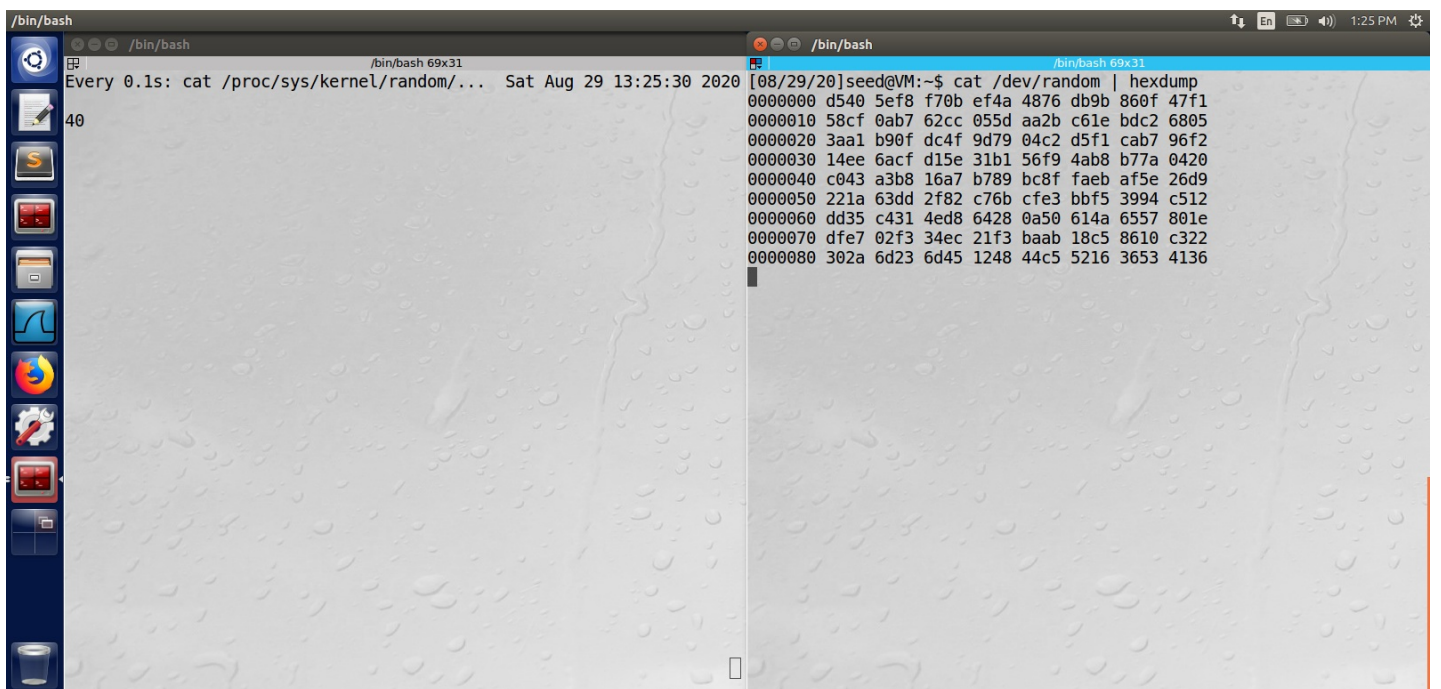


- the value keeps increasing on pressing any key or moving mouse

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 continuously from 0 to 60-70 (as far 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
```



```
/bin/bash
Every 0.1s: cat /proc/sys/kernel/random/... Sat Aug 29 13:16:52 2020
2882

0135c80 38bd 9e61 9fff dd18 4997 efd0 ecd1 51ed
0135c90 5c9e 7068 6f86 5811 e201 3fd7 42b4 1f57
0135ca0 826b c274 2ad7 6d16 dbb5 aac4 1225 a0fe
0135cb0 1dcc 8ee8 3ee2 6757 2a76 6649 656e c37c
0135cc0 85d1 d780 dc6d 641d 3347 b0ea 32c0 a3e4
0135cd0 38fa eeff caf0 210b aa6b 10d8 cb4e 2266
0135ce0 5382 f8dc 1399 e66e 94fa bab7 7ba8 525d
0135cf0 7131 7587 c01d 05fa 5a88 7a72 5d35 532a
0135d00 9f51 0bcd eb8e 7663 5879 34da 83b1 9c16
0135d10 668c a009 4456 6a24 3a54 a5a2 0802 f64c
0135d20 f0f4 495d 57d0 b877 2d29 7f8b c0f5 2eae
0135d30 5763 d4e8 c164 9578 897a ab2d ddab 2a61
0135d40 6a72 6fd3 0b9c b336 be26 792a 3917 763a
0135d50 c67f edda 7d91 c79e 1140 cab0 a4e8 a121
0135d60 48c1 d489 935b 70b0 7853 e311 e88a 387f
0135d70 504f 9876 98fc 1287 7e77 479d c8d1 0bc0
0135d80 3323 e82d ee0e 40fd dfd9 4aaf 8067 7a86
0135d90 b965 6e2f 18d4 9fbb 6698 94c0 4121 36cc
0135da0 1a81 93ed 430c d580 2934 f733 6dc5 9e5f
0135db0 3fd6 155b 9812 3c36 1501 3ad5 64d3 f9d2
0135dc0 9871 9425 00f5 8fdf a302 ba96 0720 8626
0135dd0 11e3 21d2 6c7a 1b29 f409 b3f5 0af6 c8ea
0135de0 9568 9f50 879c e8f6 aaf9 34af 30ff 76cb
0135df0 72ac f141 2cc6 ffae 57ea e5b9 aala 18c9
0135e00 a81b 3529 8197 d74a f568 0e18 49b2 4ce7
0135e10 c7f7 cac6 5fde b9ee 57ee 7e71 a1eb 4eac
0135e20 f99c 96db 9430 314e ab84 06fd 39fc cfe5
0135e30 95fa 2b24 ab28 2734 1351 f1e3 bd67 9828
0135e40 2611 d590 79e0 2995 e48f 4d74 21ac 8536
0135e50 ffec e13a 77e7 4a94 c064 0133 88bd a03a
0135e60 95a1 f009 ab21 cd22 46ef
```

- 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 generates 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).

```
/bin/bash
[08/27/20]seed@VM:~$ head -c 1M /dev/urandom > output.bin
[08/27/20]seed@VM:~$ 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.

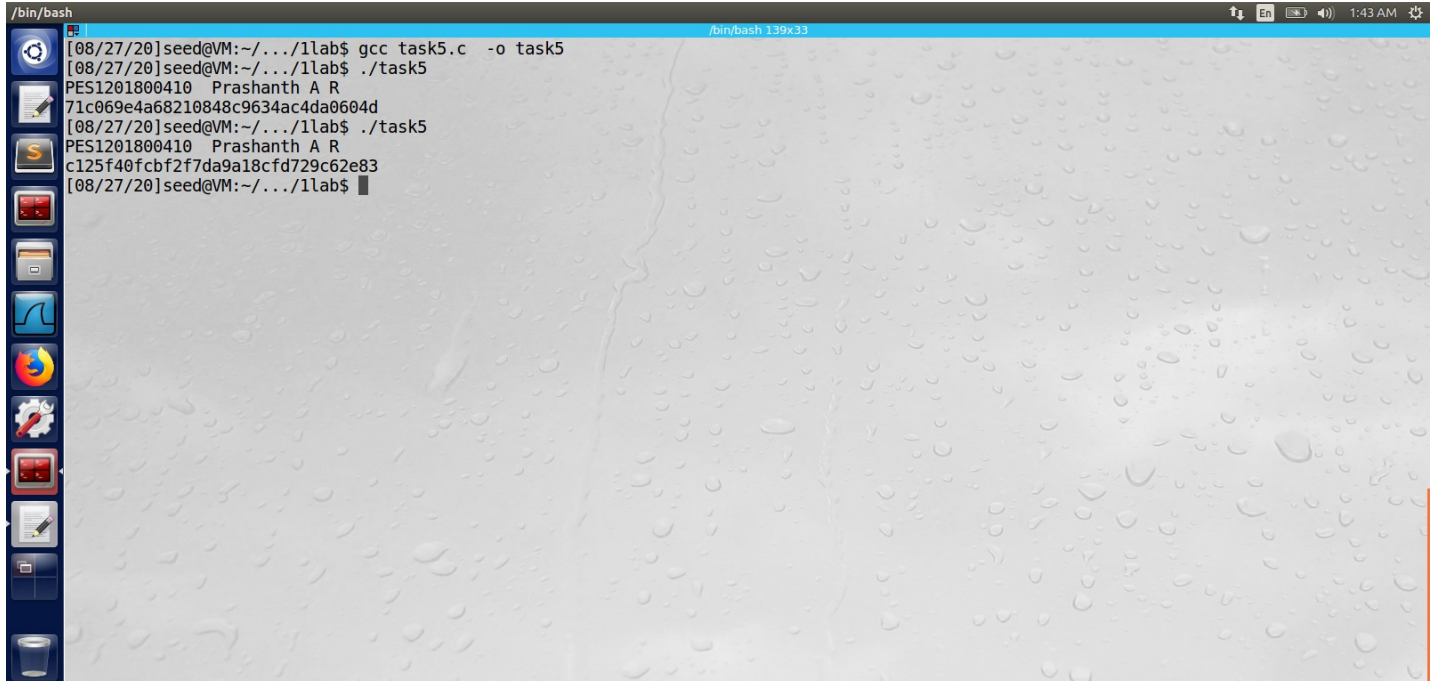
Arithmetic mean value of data bytes is 127.5868 (127.5 = random).
Monte Carlo value for Pi is 3.140957416 (error 0.02 percent).
Serial correlation coefficient is 0.000720 (totally uncorrelated = 0.0).
[08/27/20]seed@VM:~$
```

- 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