

# **Assignment 1**

**Course:** CISC 870 (Cryptography)

**Term:** Fall 2022

## **Submitted by:**

**Kazi Amit Hasan**

**ID: 20341105**

**MSc (research-based) Student, School of Computing  
Queen's University**

**Lab Setup:** For this assignment, I have installed and used SEED Ubuntu 20.4 on VirtualBox on MacBook Pro (2017 Mid). I have followed the instructions given on the website. [1] [2].

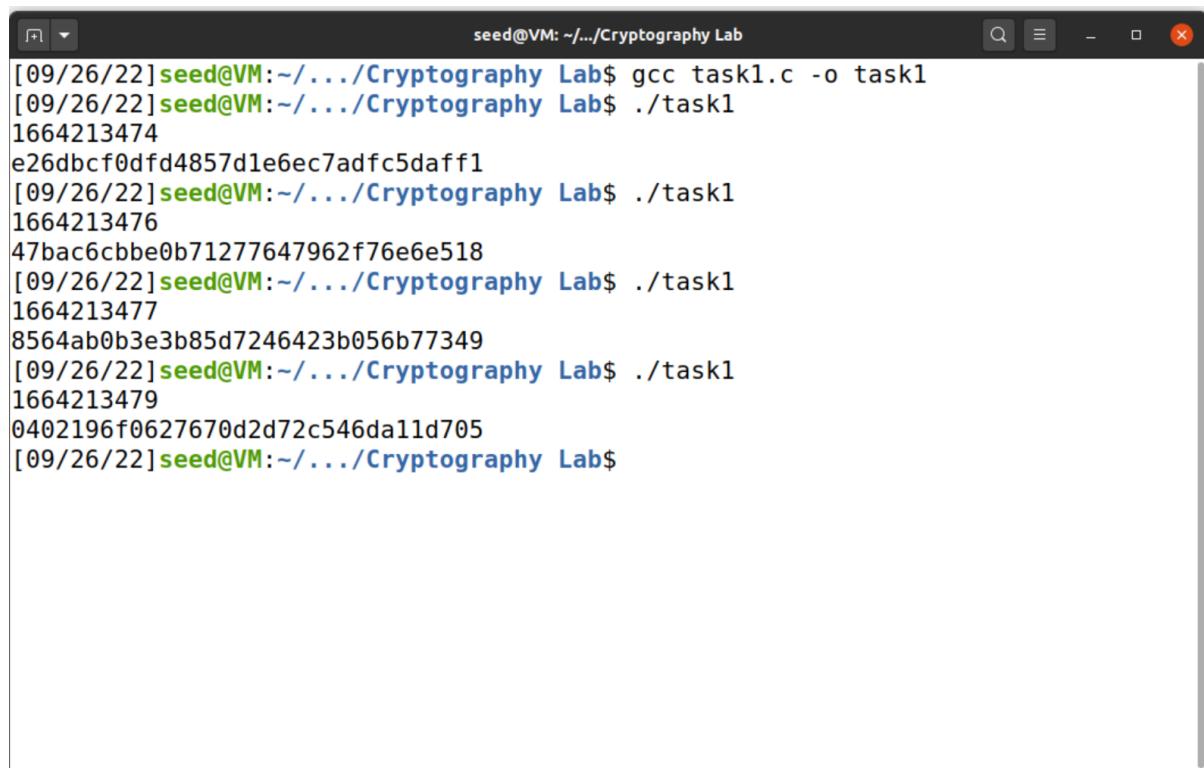
I followed the first approach of using a pre-built SEED VM. Additionally, I have allocated the memory size of 2GB instead of the usual recommendation for better performance of the Virtual machine. Also, we set the video memory to 64MB and increased the number of CPUs to 2 from 1 to increase performance.

### Task 1: Generate Encryption Key in a Wrong Way

**Approach:** To understand the observations in a better way, each code was run multiple times. This approach helped me to understand both of the codes.

**Observations before commenting out Line 1:** Time and the random number change every time when I run the ./task1 command. The output is shown in Figure 1.1.

**Observations after commenting out Line 1:** Time changes on each execution but the random number is not changing. The output is shown in Figure 1.2



```
seed@VM: ~/.../Cryptography Lab$ gcc task1.c -o task1
[09/26/22] seed@VM:~/.../Cryptography Lab$ ./task1
1664213474
e26dbcfd4857d1e6ec7adfc5daff1
[09/26/22] seed@VM:~/.../Cryptography Lab$ ./task1
1664213476
47bac6cbbe0b71277647962f76e6e518
[09/26/22] seed@VM:~/.../Cryptography Lab$ ./task1
1664213477
8564ab0b3e3b85d7246423b056b77349
[09/26/22] seed@VM:~/.../Cryptography Lab$ ./task1
1664213479
0402196f0627670d2d72c546da11d705
[09/26/22] seed@VM:~/.../Cryptography Lab$
```

Figure 1.1: Visual representation of terminal before commenting out

```
[09/26/22] seed@VM:~$ ls
Desktop Downloads output.bin Public Templates
Documents Music Pictures snap Videos
[09/26/22] seed@VM:~$ cd Desktop/
[09/26/22] seed@VM:~/Desktop$ ls
'Cryptography Lab'
[09/26/22] seed@VM:~/Desktop$ cd Cryptography\ Lab/
[09/26/22] seed@VM:~/.../Cryptography Lab$ ls
keys.txt search2.py search.py task1 task1.c task2 task2.c
[09/26/22] seed@VM:~/.../Cryptography Lab$ gcc task1.c -o task1
[09/26/22] seed@VM:~/.../Cryptography Lab$ ./task1
1664213119
67c6697351ff4aec29cdbaabf2fbe346
[09/26/22] seed@VM:~/.../Cryptography Lab$ ./task1
1664213121
67c6697351ff4aec29cdbaabf2fbe346
[09/26/22] seed@VM:~/.../Cryptography Lab$ ./task1
1664213124
67c6697351ff4aec29cdbaabf2fbe346
[09/26/22] seed@VM:~/.../Cryptography Lab$ █
```

Figure 1.2: Visual representation of terminal after commenting out

#### Purpose of srand() & time() functions in both observations:

Like the function name, the rand() function is responsible for generating random numbers. [5] The extra ‘s’ with rand() function added new functionality with it. The main purpose of srand() function is to use the computer’s current time as seed in order to generate random numbers. And the time() represents the time (in seconds) since 00:00:00 on 1/1/1970. [3]

Here in the code, ‘srand(time(NULL));’. It outputs unique random numbers in each execution as it uses the computer’s current time as seed, which is unique in every execution.

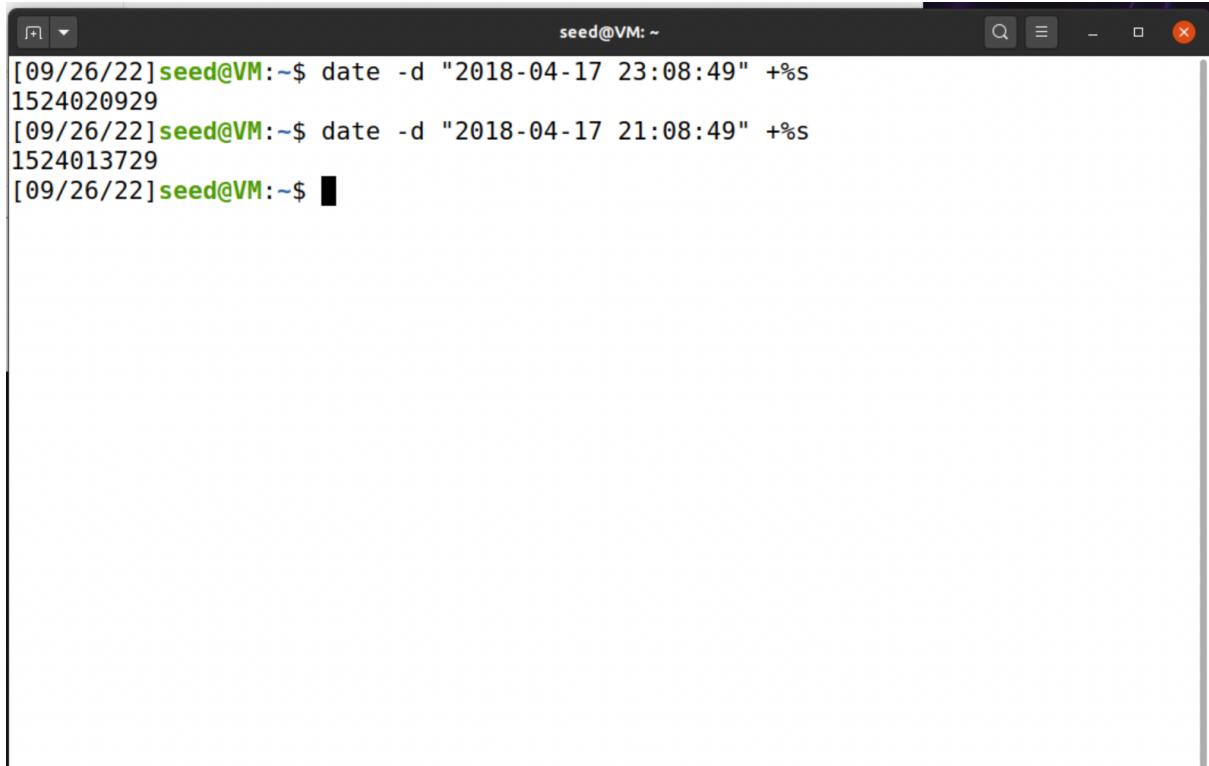
For example, The ‘1664213479’ represents the time in seconds. This shows the time since 00:00:00 on 1/1/1970. According to the figure, it is clear that the generated numbers are random and unique.

If I comment out the code, the generated random number is the same each time. I tried this with multiple executions, but the result is always the same. The reason behind this ‘same’ random number is that there is no seed set in the program.

## Task 2: Guessing the key

**Approach & Observations:** I have divided the task into several steps. The steps are described below:

1. Firstly, I generated the seed range. As per the question, the timestamp of the encrypted file was "2018-04-17 23:08:49" and Bob guessed that it might be generated within a 2 hours time frame. So, using "date -d "2018-04-17 23:08:49" +%s" and "date -d "2018-04-17 21:08:49" +%s" commands, I generated the range. The figure is shown in Fig 2.1
2. Secondly, I used those seeds and generated keys.txt file with the help of the program of task 1. Figures 2.2 and 2.3 are representing programs and keys respectively.
3. For this part, I went with the modularized approach. In the utils folder, I kept the ciphertext, plaintext, and IV in a credential file. Also, I kept the read\_keys function aside. Figure 2.4 is representing the program.
4. The output of 'search.py' is shown in Figure 2.5.
5. The structure of the utils folder is shown in Figure 2.6. Also, Figure 2.7, figure 2.8 and figure 2.9 are representing the functions inside the utils folder.
6. In this assignment, the `__init__.py` function is called as an empty constructor. Screenshot added in figure 2.9.



The screenshot shows a terminal window titled "seed@VM: ~". It displays two command-line entries and their outputs:

```
[09/26/22] seed@VM:~$ date -d "2018-04-17 23:08:49" +%s  
1524020929  
[09/26/22] seed@VM:~$ date -d "2018-04-17 21:08:49" +%s  
1524013729  
[09/26/22] seed@VM:~$ █
```

Figure 2.1: Range of seeds.

```
task2.c
~/Desktop/Cryptography Lab
Open Save ⌘ + ⌘ - ⌘ x
search.py credential.py read_keys.py task2.c
1 // Task2: Guessing the key
2
3 // Modified the first code to generate the keys
4
5 #include <stdio.h>
6 #include <stdlib.h>
7 #include <time.h>
8
9 #define KEYSIZE 16
10
11 void main()
12 {
13     char key[KEYSIZE];
14     FILE *f;
15     f = fopen("keys.txt", "w");
16     // used the seed range that I got from executing the 'date' command
17     for (long long i = 1524013729; i<=1524020929; i++) {
18         srand(i);
19         for (int j = 0; j < KEYSIZE; j++) {
20             key[j] = rand()%256;
21             fprintf(f, "%2x", (unsigned char) key [j]);
22         }
23         fprintf(f, "\n");
24     }
25 }
26
27
```

C Tab Width: 8 Ln 5, Col 19 INS

Figure 2.2: Program to generate the keys.

```
keys.txt
~/Desktop/Cryptography Lab
Open Save ⌘ + ⌘ - ⌘ x
search.py credential.py read_keys.py task2.c keys.txt
1 0a6226fc01a201b82b7d42caa7de3e05
2 12d494f3e5506c3fc152d68ae5d35bc8
3 64b838761768baa431e899b84dc5bbcd0
4 fd9b1b3ae04452506a7f269b77d95e8e
5 9a45a8c0eea61d185e2e896ea1e96167
6 bd858bd80cb81b68981fc6ab1f6b6ff0
7 405350cf2bf03e912e03bba28a2a3cc6
8 66b32d34c8315750343b34fbf329b8b5
9 794885b8757f4791ee06970ed2c1f92b
10 c270b9219acd47d50997e8404ef066f3
11 6203e205ae67a8ae76871c3061f1a8a7
12 018f4c7bd0b7ba866f27560b599540e5
13 da0178625826d50ecbcc10361d351a8a
14 4e1c67274a2434aa9d2461d9db86266d
15 7ecf3134dd870a2397da6b469802d0dd
16 68c87b6f9a4df9ae0af7d99d2458935
17 6e7b43cf01ab2d03bb37db5796b4ce72
18 19640ceb1984f19c9fc8978aea81ea8b
19 59b7a75473ea9cb18cc76a85209633a6
20 164838987cba367c53fb418b5e18f9f1
21 dde0a306a3a564f4cb8ccefbef11017
22 691a489da8eb95542c9a32cd177d6398
23 8c8e9425e5b9799cd9c8c8e76e86fce6
24 2dc1ff950d5ade1698647d99f94eabc
25 c6426944a1a230163c50a4ea2c37eba7
26 81b28317cab6d081ed228f8522102177
27 5cb1eea488990c612c46558a802e00c9
28 0671eb7a30349893c7375cfc917e3110
29 52e34e56fa4845f2a129d2a75fcfa150f
30 62458880c6c7008f33e0ed381dc8e55c
```

Plain Text Tab Width: 8 Ln 11, Col 33 INS

Figure 2.3: Visual representation of keys.txt.

The screenshot shows a code editor window with three tabs open: `search.py`, `credential.py`, and `read_keys.py`. The `search.py` tab is active and contains the following Python code:

```
1# importing package
2# I have modularized the codes so that, the keys and credential (plain, ciphertexts) kept untouched
   during the execution.
3
4from Crypto.Cipher import AES
5from utils.credential import *
6from utils.read_keys import *
7
8for k in key:
9    k = k.rstrip("\n") # this function removes any trailing characters
10   tkey= bytes(bytarray.fromhex(k)) # converting to bytes object
11   ci = AES.new(tkey, AES.MODE_CBC, iv) # as per question, bob knows the file is encrypted using
      aes-128-cbc
12   tCipher = ci.encrypt(plain)
13   # finding matching
14   if tCipher == cipher:
15       print ("FOUND")
16       print (k)
17       break
```

The code implements a暴力破解 algorithm to find the correct key by encrypting plain text with different keys and comparing the results. It uses the `AES` module from the `Crypto.Cipher` package and modularizes the credential and key reading logic.

Figure 2.4: Program to find the correct key.

```
[09/27/22]seed@VM:~/.../Cryptography Lab$ python3 search.py
FOUND
95fa2030e73ed3f8da761b4eb805dfd7
[09/27/22]seed@VM:~/.../Cryptography Lab$
```

The terminal output shows the program running and successfully finding the correct key, which is printed as `95fa2030e73ed3f8da761b4eb805dfd7`.

Figure 2.5: Output of search.py

## Utils function:

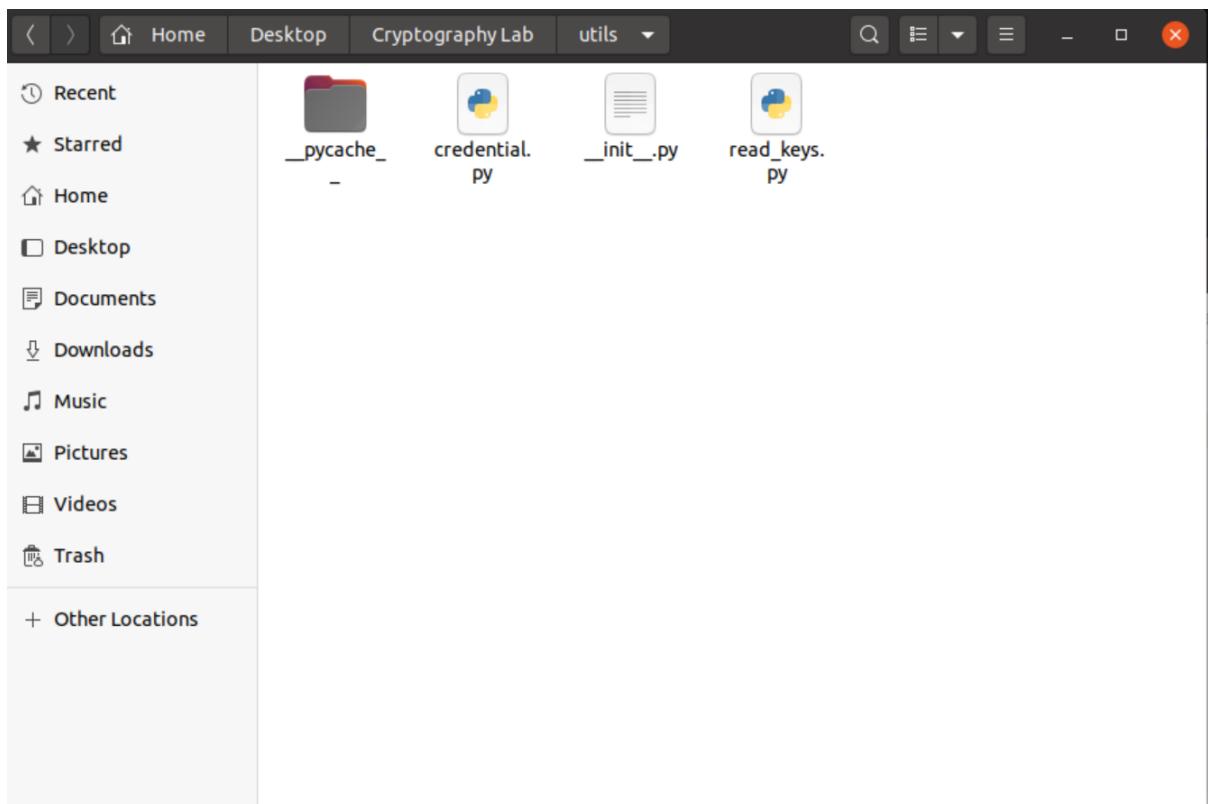


Figure 2.6: Structure of utils folder

The screenshot shows a code editor with the following tabs:

- Open
- search.py
- credential.py
- read\_keys.py
- task2.c

The credential.py tab is active, showing the following Python code:

```
1 #Given plain, cipher and initial vectors
2
3 plain = bytes.fromhex("255044462d312e350a25d0d4c5d80a34")
4 cipher = bytes.fromhex("d06bf9d0dab8e8ef880660d2af65aa82")
5 iv = bytes.fromhex("09080706050403020100A2B2C2D2E2F2")
```

At the bottom of the editor, the status bar displays:

Python Tab Width: 8 Ln 5, Col 66 INS

Figure 2.7: Given credentials in the utils folder

A screenshot of a code editor window titled "read\_keys.py". The window shows several tabs at the top: "search.py", "credential.py", "read\_keys.py", and "task2.c". The "read\_keys.py" tab is active. The code in the editor is:

```
1 # this read_keys.py file will read the keys that were generated in task 2.1
2
3 f = open("keys.txt")
4 key = f.readlines()
5
6#print(key)
```

The status bar at the bottom right shows "Python", "Tab Width: 8", "Ln 2, Col 1", and "INS".

Figure 2.8: read keys in the utils folder

A screenshot of a code editor window titled "\_\_init\_\_.py". The window shows several tabs at the top: "search.py", "credential.py", "read\_keys.py", "task2.c", "keys.txt", and "\_\_init\_\_.py". The "\_\_init\_\_.py" tab is active. The code in the editor is:

```
1 |
```

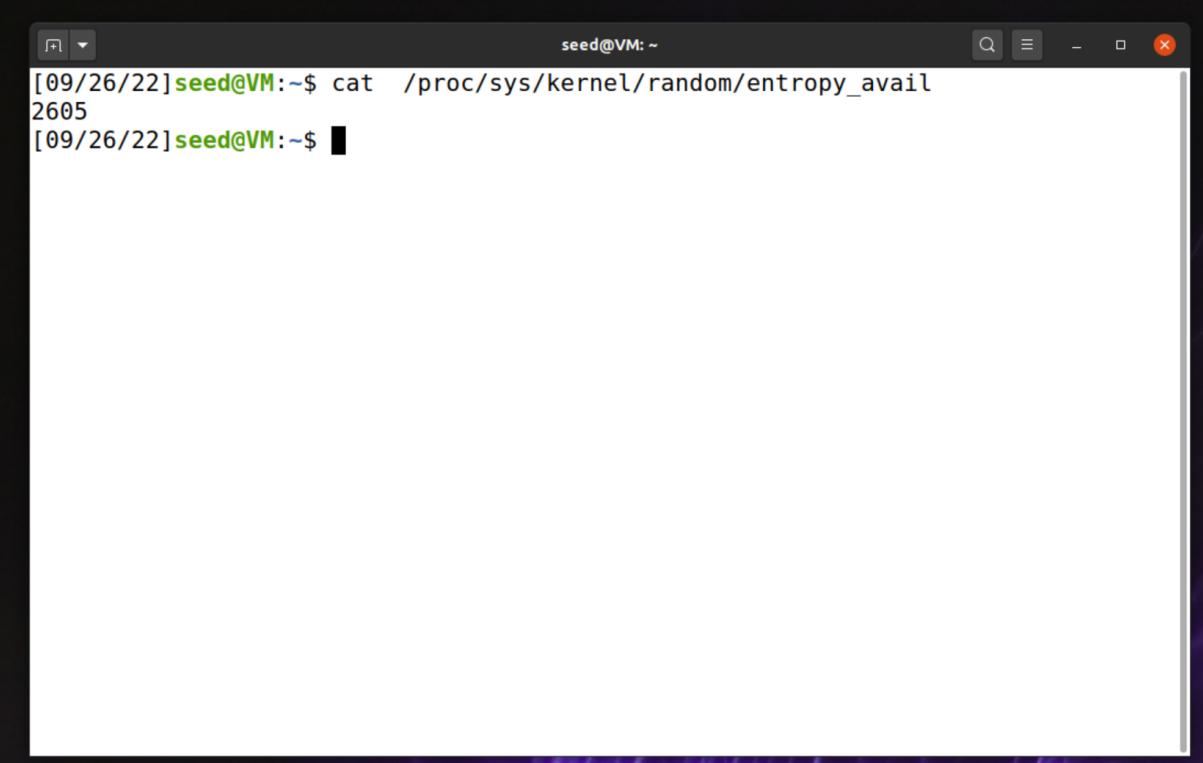
The status bar at the bottom right shows "Python", "Tab Width: 8", "Ln 1, Col 1", and "INS".

Figure 2.9: Representation of \_\_init\_\_.py file

### Task 3: Measure the Entropy of Kernel

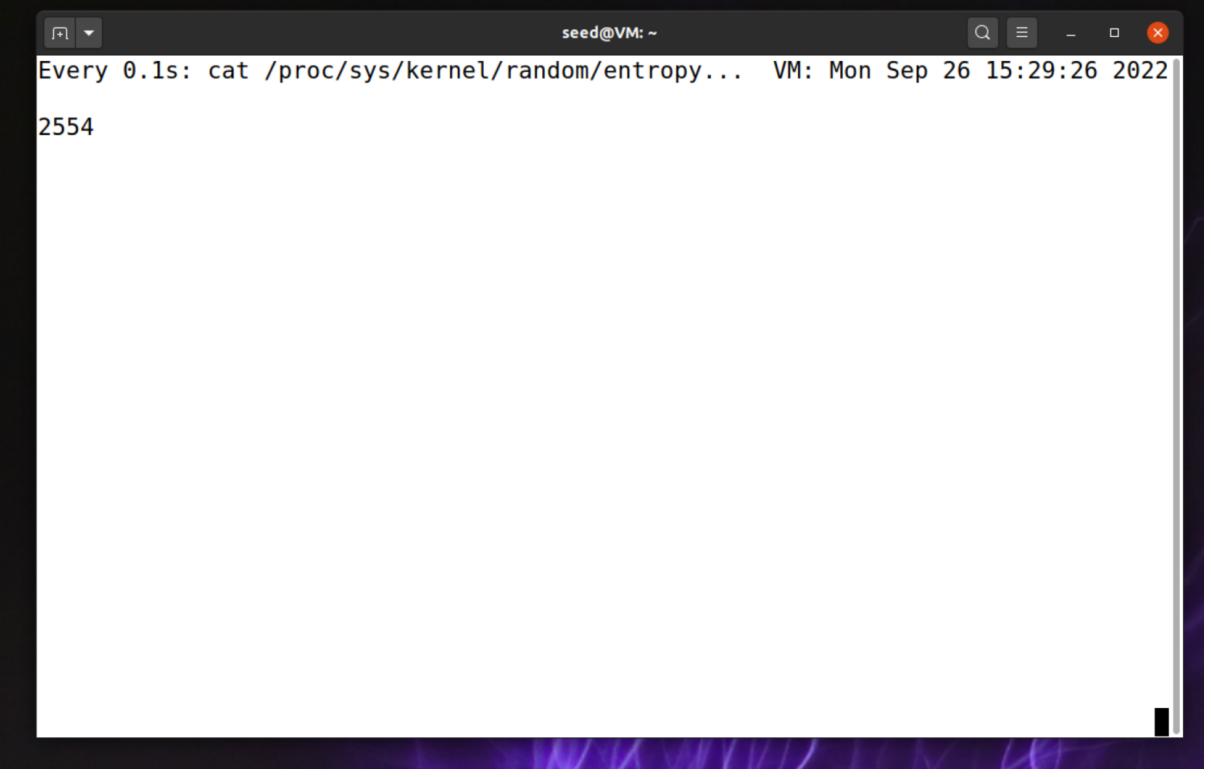
**Approach & Observations:** For this task, firstly, I run the '\$ cat /proc/sys/kernel/random/entropy\_avail' command to check the current entropy. Then, I added the 'watch' command to it and observed the changes in entropy. The final command was 'watch -n .1 cat /proc/sys/kernel/random/entropy\_avail'.

Figure 3.1 shows the current entropy of the system which is 2605. Then I executed the command with 'watch' and observed the changes. The figure 3.2 represents the entropy while running the 'watch -n .1 cat /proc/sys/kernel/random/entropy\_avail' command only. Figure 3.3 shows the output of entropy while moving the mouse, or typing something. It was observed that the entropy increased while typing something really fast. Figure 3.4 shows that the entropy also increased while reading a file. Also, I saw a sudden increase in entropy while visiting a website, figure 3.5 represents the increase. After executing all these tasks, I encountered that reading large files and visiting websites increase the entropy of the system significantly.



The screenshot shows a terminal window with a black background and white text. The title bar says 'seed@VM: ~'. The command entered is '[09/26/22] seed@VM:~\$ cat /proc/sys/kernel/random/entropy\_avail'. The output is '2605'. The prompt '[09/26/22] seed@VM:~\$' is visible at the bottom. The window has standard Linux-style window controls (minimize, maximize, close) in the top right corner.

Figure 3.1: Current entropy



```
seed@VM: ~
Every 0.1s: cat /proc/sys/kernel/random/entropy...  VM: Mon Sep 26 15:29:26 2022
2554
```

Figure 3.2: Change in entropy while keeping everything as it is.



```
seed@VM: ~
Every 0.1s: cat /proc/sys/kernel/random/entropy...  VM: Mon Sep 26 15:31:05 2022
2772
```

Figure 3.3: Change in entropy while moving mouse, typing something

The screenshot shows a terminal window and a code editor side-by-side. The terminal window at the top displays the command "cat /proc/sys/kernel/random/entropy..." followed by the number "2837". The code editor below contains a C program named "task1.c" which generates a key of size 16 bytes by reading the current time and generating random bytes.

```
seed@VM: ~
Every 0.1s: cat /proc/sys/kernel/random/entropy...  VM: Mon Sep 26 15:34:36 2022
2837

task1.c
~/Desktop/Cryptography Lab
Open Save
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4 #define KEYSIZE 16
5 void main()
6 {
7     int i;
8     char key[KEYSIZE];
9     printf("%lld\n", (long long) time(NULL));
10    srand(time(NULL));
11    for (i = 0; i < KEYSIZE; i++)
12    {
13        key[i] = rand()%256;
14        printf("%.2x", (unsigned char)key[i]);
15    }
16    printf("\n");
17
18
19 }
```

Figure 3.4: Change in entropy while reading a file

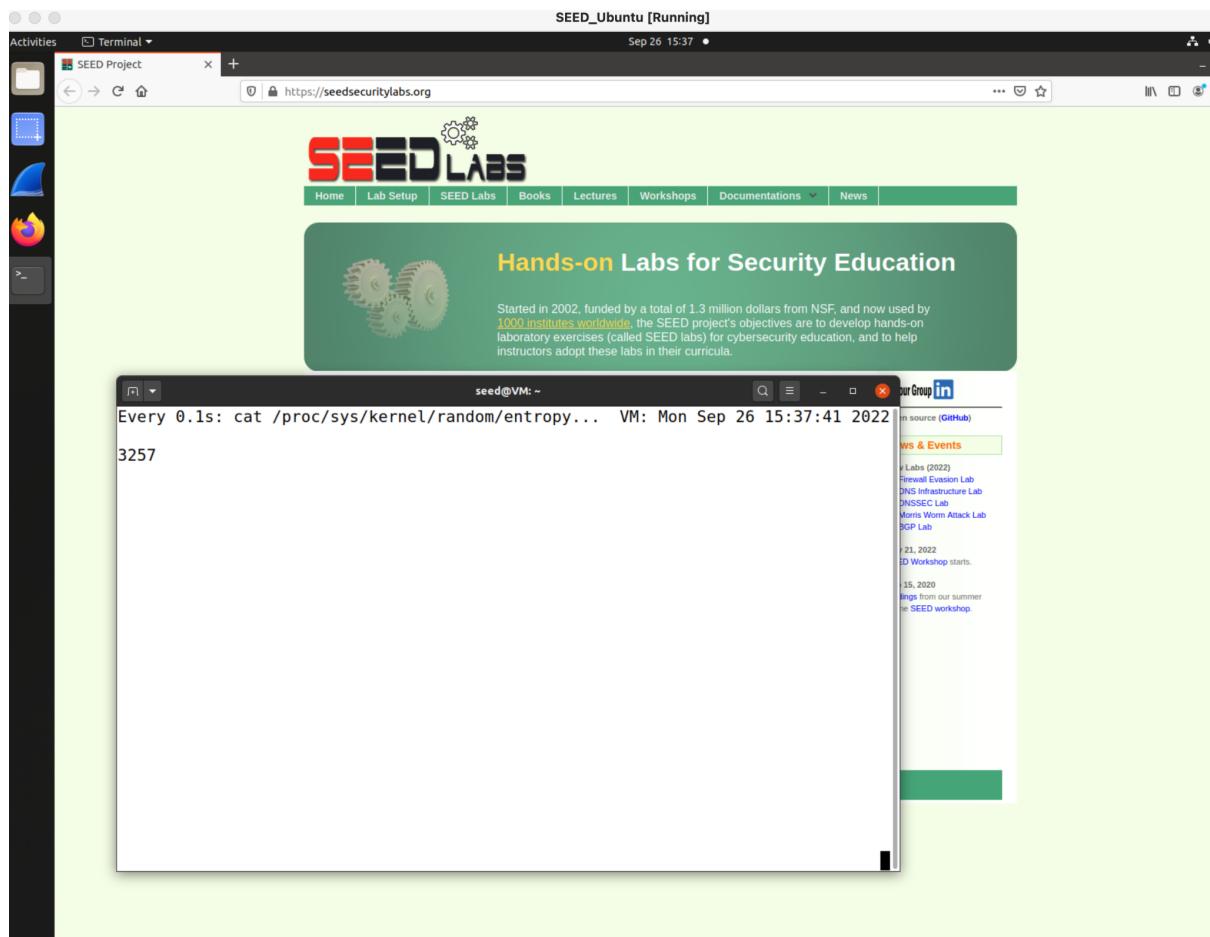


Figure: 3.5: Change in entropy while visiting a website

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

**Approach:** For this task, I executed the command ‘cat /dev/random | hexdump’ and observed the entropy at the same time by using the “watch -n .1 cat /proc/sys/kernel/random/entropy\_avail” command. There are some interesting observations while doing this task.

#### Observations:

1. **Do not move mouse/type anything:** If I do not move mouse or type anything, the change in entropy is comparatively slower than other tasks. Eventually, it starts coming to lower side gradually. I observed close to ‘1’ a few times. Also, the generation of numbers is really slow. Even this could be compared to no output. The visuals are added in figure 4.1. The figure might seem confusing. But the screenshot was taken at the initial time, the entropy started dropping after some moment.
2. **Randomly moving mouse:** When I randomly moved the mouse, I observed a good speed in changing the entropy in the terminal. Also, the generation of numbers is

really faster than the previously ‘not moving’ mouse. The visuals are added in figure 4.2

**Explanations and learnings:**

As it is described in the task, /dev/random device is a blocking device. As the entropy drops, the speed of random number generation also drops. Thus, when the entropy becomes zero, the /dev/random command will stop/pause some functionalities until it achieves some randomness. [6]

The image shows two terminal windows side-by-side. Both windows have a dark background and a light-colored text area. The top window has a title bar that reads "seed@VM: ~" and a status bar at the bottom that says "Every 0.1s: cat /proc/sys/kernel/random/entropy... VM: Mon Sep 26 16:08:52 2022". It contains a single line of text: "56". The bottom window also has a title bar reading "seed@VM: ~" and a status bar at the bottom. It contains a large amount of hexagonal entropy data, starting with "00000090 4a44 1674 335d f2a9 92ff c4e2 7f22 4b12" and ending with "00001f0 653c 4a8b d3f6 8358 36f1 64e5 4ec8 8719".

```
00000090 4a44 1674 335d f2a9 92ff c4e2 7f22 4b12
00000a0 298c 0db6 8f59 792e 3f14 a467 05aa af8c
00000b0 e94c 2def 5806 01b9 3eeb f10b 4353 4099
00000c0 4203 bce0 ba17 211c e776 650f b030 d4da
00000d0 2c42 ba0b b50e d377 5a3d 78c5 7c95 be62
00000e0 6588 df2f 57d3 a26c 39f8 5a10 7894 ed7c
00000f0 e1cb 2d7c 9a14 5a7c ed9c 6cd8 5ecd 1283
0000100 5614 b28f 6a61 634e c304 a378 7045 ba0d
0000110 9ee2 9dd8 3424 27c7 3d7f 3885 3a34 e8dc
0000120 2bee eb3a d5d4 b1cf 06dd 9b48 b798 694b
0000130 ab5b 49d6 e6c2 9e56 f122 5b43 2cda 7e4d
0000140 bda9 e291 e285 87f2 40d0 5d5f 133e 350d
0000150 b2c5 68e0 dab0 99e0 96c8 d4ac 01b8 ee2f
0000160 57a9 44c0 0ec9 5613 aa4d d21c 2912 c5ec
0000170 fc90 c7c0 6c70 020c 1ae7 d554 e561 9d06
0000180 8b5d 56d0 6055 4242 44aa f033 9df5 985b
0000190 bae1 eabd 7623 e1d6 252d 0cb2 60d6 5b55
00001a0 ca4b b07c d41c 3c1e 54c6 bc34 75a8 3bf1
00001b0 8bd2 2f2c f38f e6e1 11ec f437 fd21 1f09
00001c0 474c a440 dd36 621b 4653 f082 b5e1 1a92
00001d0 1ce9 5866 c49c 3b5d ed08 d6dc fe87 c946
00001e0 e70b 6adb b0a8 ee14 d0b6 95e1 47c5 a558
00001f0 653c 4a8b d3f6 8358 36f1 64e5 4ec8 8719
```

Figure 4.1: Do not moving mouse/type anything

The image shows two terminal windows side-by-side. Both windows have a dark background and light-colored text. The top window displays a single line of text: "Every 0.1s: cat /proc/sys/kernel/random/entropy... VM: Mon Sep 26 16:13:25 2022". Below this, the number "52" is printed. The bottom window displays a large amount of entropy data in hex format, starting with "00000f0 e1cb 2d7c 9a14 5a7c ed9c 6cd8 5ecd 1283" and continuing for many lines. The terminal windows have standard Linux-style interfaces with tabs, a search bar, and a close button.

```
Every 0.1s: cat /proc/sys/kernel/random/entropy... VM: Mon Sep 26 16:13:25 2022
52

00000f0 e1cb 2d7c 9a14 5a7c ed9c 6cd8 5ecd 1283
0000100 5614 b28f 6a61 634e c304 a378 7045 ba0d
0000110 9ee2 9dd8 3424 27c7 3d7f 3885 3a34 e8dc
0000120 2bee eb3a d5d4 b1cf 06dd 9b48 b798 694b
0000130 ab5b 49d6 e6c2 9e56 f122 5b43 2cda 7e4d
0000140 bda9 e291 e285 87f2 40d0 5d5f 133e 350d
0000150 b2c5 68e0 dab0 99e0 96c8 d4ac 01b8 ee2f
0000160 57a9 44c0 0ec9 5613 aa4d d21c 2912 c5ec
0000170 fc90 c7c0 6c70 020c 1ae7 d554 e561 9d06
0000180 8b5d 56d0 6055 4242 44aa f033 9df5 985b
0000190 bae1 eabd 7623 e1d6 252d 0cb2 60d6 5b55
00001a0 ca4b b07c d41c 3c1e 54c6 bc34 75a8 3bf1
00001b0 8bd2 2f2c f38f e6e1 11ec f437 fd21 1f09
00001c0 474c a440 dd36 621b 4653 f082 b5e1 1a92
00001d0 1ce9 5866 c49c 3b5d ed08 d6dc fe87 c946
00001e0 e70b 6adb b0a8 ee14 d0b6 95e1 47c5 a558
00001f0 653c 4a8b d3f6 8358 36f1 64e5 4ec8 8719
0000200 565a febc dfb4 772d b040 9677 0005 2906
0000210 620a f824 c358 de03 7df1 fa31 fab8 42dc
0000220 2214 77f2 b3bd bc6c bffd 6a87 efaf 7c8f
0000230 0957 107e 7354 d4b4 d2be a065 39d0 d769
0000240 d522 366f 5e0f 1bf9 e0cd f0a9 7c0d a345
0000250 1a98 4fb7 9d4c 16bf 4539 ef7c adcc 039e
```

Figure 4.2: Randomly moving mouse:

**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.

**Answer:** As we know, /dev/random device is a blocking device, the entropy will drop gradually and will pause functionalities if there's not enough entropy found. At the time of such situations, a Denial of Service (DOS) attack could launch by sending request. [8]

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

### Approach:

1. The first task was to observe the behavior of “cat /dev/urandom | hexdump”. For this, I executed this command on the terminal. The observation is described in a later section.
2. The next task was to evaluate the quality of random numbers. For this, I executed two commands “head -c 1M /dev/urandom > output.bin” and “ent output.bin” respectively. After running these, I got some interesting parameters to explore. The observations are described in a later section.
3. The last task was to generate a 256-bit encryption key. For this, I added and modified the code of task 2 and added it in figure 5.3 and the output is also shown in figure 5.4. The code snipped was already provided in the instructions.

### Observations:

1. **After ruining cat/dev/urandom**, it is shown that the terminal was generating random numbers constantly at an excellent increased speed. I moved the mouse and typed, but it has no impact on number generation. The numbers were generating rapidly.
2. **Measuring quality of random number**: Figure 5.2 represents the outcome of 1 MB of pseudo-random number from /dev/urandom. Here, each outcome has its own significant meaning. [7]
  - a. **Entropy**: 7.999842 bit per byte. This represents the density of information. The achieved score of 7.999842 is considered a very better score. [7]
  - b. **Optimum Compression**: As per Figure 5.2, the optimum compression is 0%. This reveals that the generated data is random. [7]
  - c. **Chi-square distribution**: According to Figure 5.2, 87.59% was achieved. [7]
  - d. **Arithmetic means**: I got the arithmetic mean of 127.65 which is close to 127.5. [7]
  - e. **Monte Carlo value of pi**: I got the value of 3.122848422 with a 0.1% error. This refers to a good outcome. [7]
  - f. **Serial correlation coefficient**: 0.000318 and it's close to 0.  
After observing these parameters, we can say that the quality of random number is good.
3. **Generating 256-bit encryption key**: As the code snipped was already given, it was given 128 bits, so I had to generate 256 bits. So I changed the length to 32 from 16.

The image shows two terminal windows side-by-side. Both windows have a dark theme with white text and a light gray background. The top window has a title bar that reads "seed@VM: ~" and a status bar at the bottom that says "Every 0.1s: cat /proc/sys/kernel/random/entropy... VM: Mon Sep 26 17:14:56 2022". The main content of this window is the number "2204". The bottom window also has a title bar reading "seed@VM: ~" and a status bar at the bottom. The main content of this window is a long list of hexadecimal values, each on a new line, starting with "9f3b960 c080 4ed6 5ffc e797 2e54 ecdb 65b8 b3dd" and ending with "9f3bac0 b7cb 7e59 8995 d9ef 027f 2c34 5caa 390d".

```
Every 0.1s: cat /proc/sys/kernel/random/entropy... VM: Mon Sep 26 17:14:56 2022
2204

9f3b960 c080 4ed6 5ffc e797 2e54 ecdb 65b8 b3dd
9f3b970 0b54 8b56 bd0e a051 987f ed63 60cf 8e3f
9f3b980 f1f7 ee87 dc80 3802 9182 f487 c039 cf9d
9f3b990 dda4 c97f 9d03 3eef 8e1e 11eb decc ef62
9f3b9a0 bcac c977 d58b 6f06 2666 952e 5e88 61d5
9f3b9b0 d9c8 e814 8715 2110 8cc9 1701 76c3 dbe2
9f3b9c0 54ed 78ee 3a3d 14d5 cd04 1663 c805 f160
9f3b9d0 8521 ba99 c905 7b4f 3381 5277 92c7 ff34
9f3b9e0 d671 9d73 0c31 e215 7ab6 f14d d497 315b
9f3b9f0 deba 1286 3cb8 1c7a 8177 4c56 a2db aa52
9f3ba00 2058 7d77 0fad c386 b75a 6b93 2b41 733f
9f3ba10 d828 bade 4782 5724 d4bb 987b a37a 3171
9f3ba20 9cf5 12a3 a2bb 942f dd87 1a6c 6928 ddd9
9f3ba30 ba73 194e 674c 6058 6d0e f8ff 4578 4643
9f3ba40 5172 3fee 01b1 36ee 5ce9 9084 a394 7d16
9f3ba50 073f 5f69 9058 6c9b 605d 7701 ee02 ed13
9f3ba60 c8b9 4b1e d9f7 5293 33ee 6813 dc8e 7891
9f3ba70 0fd9 dc7b bc2e e60f 854f b057 aef9 852f
9f3ba80 93be 5fa5 72c7 50d1 b0a0 4fed 96e7 0905
9f3ba90 0d9d dc72 a77d 1f45 19e8 9754 7002 4ce7
9f3baa0 f3c7 0601 20ce cbc0 39e0 5651 8760 50f7
9f3bab0 58c9 4e0c a01a 2cbb 542d 84cc b310 581b
9f3bac0 b7cb 7e59 8995 d9ef 027f 2c34 5caa 390d
```

Figure 5.1: After running cat/dev/urandom.

The screenshot shows a terminal window titled "seed@VM: ~/Desktop". The output of the command "head -c 1M /dev/urandom > output.bin" is displayed, followed by the results of the "ent" command on the file "output.bin". The entropy value is 7.999842 bits per byte, which is near the maximum of 8 bits per byte. The terminal also displays statistical analysis results for the 1048576 samples, including the arithmetic mean, Monte Carlo value for Pi, and serial correlation coefficient.

```
[09/26/22] seed@VM:~/Desktop$ head -c 1M /dev/urandom > output.bin
[09/26/22] seed@VM:~/Desktop$ ent output.bin
Entropy = 7.999842 bits per byte.

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

Chi square distribution for 1048576 samples is 229.17, and randomly
would exceed this value 87.59 percent of the times.

Arithmetic mean value of data bytes is 127.6525 (127.5 = random).
Monte Carlo value for Pi is 3.144848422 (error 0.10 percent).
Serial correlation coefficient is 0.000318 (totally uncorrelated = 0.0).
[09/26/22] seed@VM:~/Desktop$
```

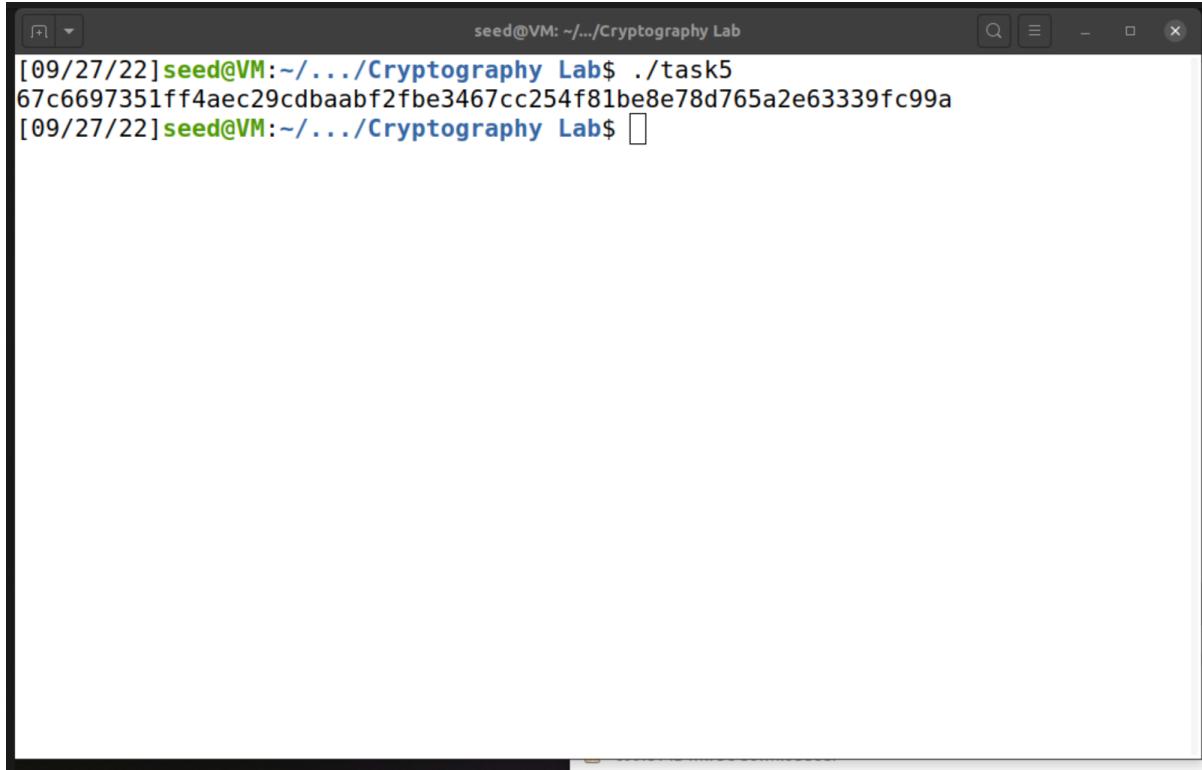
Figure 5.2: Measuring entropy of 1MB of pseudo-random number

The screenshot shows a code editor window with two tabs: "task2.c" and "\*task5.c". The code in "task5.c" is a C program designed to generate a 256-bit encryption key. It includes headers for stdio.h, stdlib.h, and time.h, defines a length of 32 bytes (256 bits), and uses malloc to allocate memory for the key. The program reads from "/dev/urandom" and writes to "keys.txt". It includes sections of code from task2.c and adds a snippet for generating the key. The code uses standard C syntax with comments explaining its purpose.

```
task2.c
*task5.c
-/Desktop/Cryptography Lab
Save

1 // Task5: Generate 256 bit encryption key
2
3 // Modified the task2.c program to gennerate the keys
4 // Commented out the sections of task2 and added the given code snippet.
5
6 #include <stdio.h>
7 #include <stdlib.h>
8 #include <time.h>
9 #define LEN 32 // 256 bit
10
11 // #define KEYSIZE 16
12
13 void main()
14 {
15     unsigned char *key = (unsigned char *) malloc(sizeof(unsigned char)*LEN);
16
17     //char key[KEYSIZE];
18     //FILE *f;
19
20     FILE* random = fopen("/dev/urandom", "r");
21     fread(key, sizeof(unsigned char)*LEN, 1, random);
22
23     //f = fopen("keys.txt", "w");
24     // used the seed range that I got from executing the 'date' command
25
26     fclose(random);
27     for (long long i = 0; i<LEN; i++) {
28         //srand(i);
29         //for (int j = 0; j < KEYSIZE; j++) {
30         key[i] = rand()%256;
31         printf("%2x", (unsigned char) key [i]);
32     }
33     printf("\n");
34
35     //fprintf(f, "\n");
36 }
37 }
```

Figure 5.3: Generating 256 bit encryption key



The screenshot shows a terminal window titled "seed@VM: ~/.../Cryptography Lab". The command entered is ". ./task5". The output is a long hexadecimal string: "67c6697351ff4aec29cdbaabf2fbe3467cc254f81be8e78d765a2e63339fc99a". Below the output, there is a small square icon.

Figure 5.4: The output of 256-bit generated key.

### Conclusion:

In this assignment, I learned about pseudo-random number generation. Generating encryption keys random/urandom devices. Some outputs were really interesting and were fun to explore and observe. I had spent a good amount of time installing VM and Ubuntu 20.4 on my personal machine. In the first 3 attempts, I got errors and faced a lot of problems. Then for the 4th time I reinstalled Ubuntu 20.04 on VM with some extra cores and allocated some more space. Still, the machine felt a bit slow while doing the assignments. However, this assignment was really informative and interesting.

### References:

1. (Available online) <https://seedsecuritylabs.org/labsetup.html>
2. (Available online) <https://github.com/seed-labs/seed-labs/blob/master/manuals/vm/seedvm-manual.md>
3. (Available online) [https://www.geeksforgeeks.org/time-function-in-c/#:~:text=The%20time\(\)%20function%20is,object%20pointed%20to%20by%20second.](https://www.geeksforgeeks.org/time-function-in-c/#:~:text=The%20time()%20function%20is,object%20pointed%20to%20by%20second.)
4. (Available online) <https://cplusplus.com/reference/cstdlib/srand/>
5. (Available online) <https://www.geeksforgeeks.org/rand-and-srand-in-ccpp/?ref=gcse>

6. (Available online)  
[https://seedsecuritylabs.org/Labs\\_20.04/Files/Crypto\\_Random\\_Number/Crypto\\_Random\\_Number.pdf](https://seedsecuritylabs.org/Labs_20.04/Files/Crypto_Random_Number/Crypto_Random_Number.pdf)
7. (Available online)  
[https://programmersought.com/article/21636379515/#22\\_Task\\_2\\_Guessing\\_the\\_Key\\_57](https://programmersought.com/article/21636379515/#22_Task_2_Guessing_the_Key_57)
8. ((Available online) )  
<https://security.stackexchange.com/questions/194958/dev-random-generate-session-key>