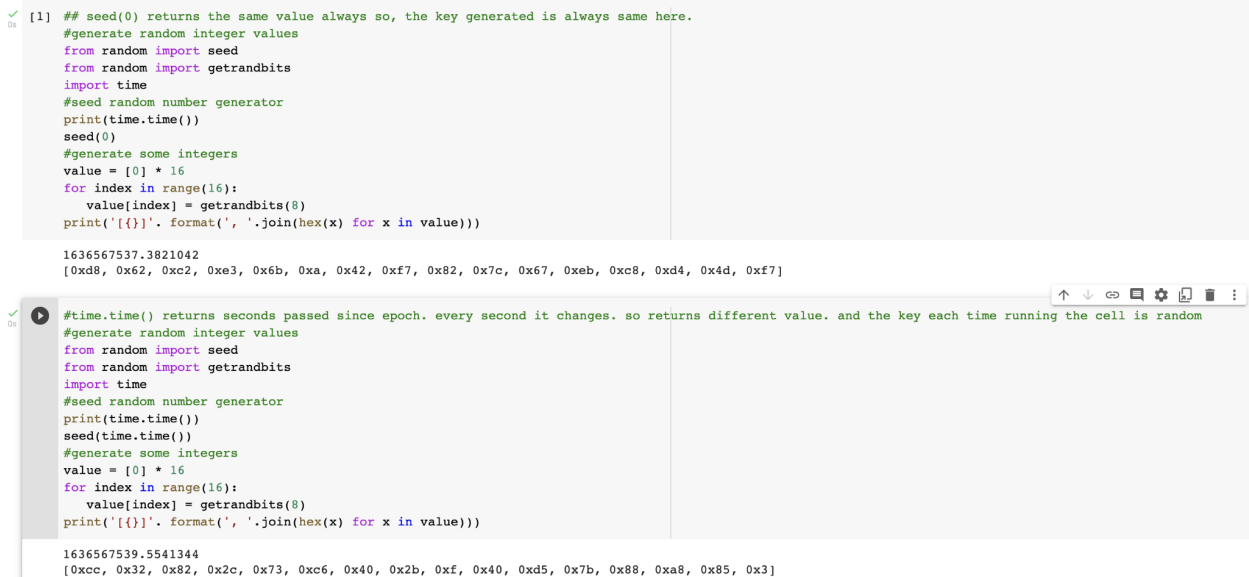


ECE 55560 - HW3 LAB REPORT

Task 1.



```
[1] ## seed(0) returns the same value always so, the key generated is always same here.
#generate random integer values
from random import seed
from random import getrandbits
import time
#seed random number generator
print(time.time())
seed(0)
#generate some integers
value = [0] * 16
for index in range(16):
    value[index] = getrandbits(8)
print('{} {}'.format(' '.join(hex(x) for x in value)))

1636567537.3821042
[0xd8, 0x62, 0xc2, 0xe3, 0x6b, 0xa, 0x42, 0xf7, 0x82, 0x7c, 0x67, 0xeb, 0xc8, 0xd4, 0x4d, 0xf7]
```

```
#time.time() returns seconds passed since epoch. every second it changes. so returns different value. and the key each time running the cell is random
#generate random integer values
from random import seed
from random import getrandbits
import time
#seed random number generator
print(time.time())
seed(time.time())
#generate some integers
value = [0] * 16
for index in range(16):
    value[index] = getrandbits(8)
print('{} {}'.format(' '.join(hex(x) for x in value)))

1636567539.5541344
[0xcc, 0x32, 0x82, 0x2c, 0x73, 0xc6, 0x40, 0x2b, 0xf, 0x40, 0xd5, 0x7b, 0x88, 0xa8, 0x85, 0x3]
```

Figure 1: Giving constant and variable values as seeds

While generating pseudo-random bits with seed is given to the deterministic algorithm, which determines the output completely. In Figure 1, the first code block uses the same seed given to the algorithm. Therefore every time it is run, the algorithm outputs the same 16 byte key. However, in the second block, time() function is used for seed. Time() returns the seconds passed from the Epoch in seconds. Hence it changes every time we run it. Each time it is run, we initialize the algorithm with a different starter(seed) and end up with a different pseudo random 16 byte key. This demonstrates the importance of knowing the seed, since if the attacker knows the seed and the algorithm, he/she can obtain the key.

Task 2.

```
denizaytemiz@Denizs-MacBook-Air Desktop % date -j -f "%Y %m %d %H %M %S" "2018 04 17 23 08 49" "+%s"
1524020929
denizaytemiz@Denizs-MacBook-Air Desktop % date -j -f "%Y %m %d %H %M %S" "2018 04 17 21 08 49" "+%s"
1524013729
denizaytemiz@Denizs-MacBook-Air Desktop %
```

Figure 2: Finding the inputs/ seed values for every possible date. These values will be used to create all the potential keys used for encryption.

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define KEYSIZE 16
int main(void)
{
    int i;
    FILE *kfile;
    kfile = fopen("key.txt", "w");
    char key[KEYSIZE];
    int k = 1524013729;
    while(k<=1524020929)
    {
        srand(k);
        for (i = 0; i< KEYSIZE; i++){
            key[i] = rand()%256;
            fprintf(kfile, "%.2x", (unsigned char)key[i]);
        }
        fprintf(kfile, "\n");
        k++;
    }
    return 0;
}

```

Figure 3: The modified c code for creating all possible keys depending on the date and time of file creation.

In Figure 3, we scan every value in between the 2 seeds. Each can be the seed used for encryption during the 2 hours window. We create 16 byte keys for each possible seed value and save them line by line to a text file. Later on this file will be used in the python code that detects the correct key.

```

6474 5024236b51f5170cea9f1a362640a48f
6475 682aae52b64a81afa8c5de8a9a683c14
6476 0e2cc2dc46b6e9c6d28ab51b853597c5
6477 84380b71086ccfd5223d9949ef37ebd1
6478 89301d27e5f709888650a00a2f62a7ab
6479 9e0d5d89c7a31408b13b18fc4a6ed8c8
6480 89a3293652a3c4b38a534787074255dd
6481 913d7d500dce2c40e9322ab8c2135c7e
6482 132831cf54bea0fe438c3caf6219df52
6483 7291c45dfd7985127ea69204719f06b2
6484 7c89c0bd15140638946c6b9a0c439e38
6485 640538f559a22a99c660715ed5cd68d0
6486 ce1d340c376fb45d34c3a24ae06efbef
6487 cb5a9bb3ae39b8139e3b6597389045c7
6488 1645318024fca78b8d948394b3465779
6489 3975a895b3489a7b24299d6393f7dcac
6490 0193d52973507e4443cf925eed71b01d
6491 524aac9b32b891b273c2cf72b4479eb9
6492 55576fd422290b994339d34fa9dc4575
6493 1efddea8033cbdbf92ead95cb0399174
6494 61e9dd3c31a1b2e7e7002e71377ad49f
6495 b5c6dc8b5aa60b28a41951b71440ef66
6496 fe363d1569c9305c6a62df1df918643b
6497 e37b103067cd851f5e78857df848a990
6498 2c33d2cef3077e2de51ebf8ba6e16a07
6499 058f2a58c57cbc402b9a81e1f969635d
6500 46d163e0c572971f2a3686a0f90398e5
6501 b72a91b61aa611c20eb2c6954dd4e995
6502 28b76ed87f7186253e99b4225b75c291
6503 70fd23b73274dea215cb03a6d3ee2a6b
6504 04dbbe4ccb1e83eb7ce4a3d09bccd48b
6505 da4772ccf94fc3722184d5bdSafa24fb
6506 644e03af5552ee346935fdf832054129
6507 10e0b5308c97099bdd424ebe33a4cb25

```

Figure 4: The respective key.txt file with all possible keys written

```

!pip uninstall crypto
!pip uninstall pycryptodome
!pip install pycryptodome

WARNING: Skipping crypto as it is not installed.
WARNING: Skipping pycryptodome as it is not installed.
Collecting pycryptodome
  Downloading pycryptodome-3.11.0-cp35-abi3-manylinux2010_x86_64.whl (1.9 MB)
    |████████████████████| 1.9 MB 4.3 MB/s
Installing collected packages: pycryptodome
Successfully installed pycryptodome-3.11.0

[2] import binascii
    from Crypto.Cipher import AES

[4] file1 = open('/content/key.txt', 'r')
    Lines = file1.readlines()

[9] for line in Lines:
    print(format(line.strip()))

Streaming output truncated to the last 5000 lines.
96b2560b2f1b73b71818a92b0ff9cec8
ec810cfd2bac6b7892a414194724db4b
006fb86453d7f0c80631aa9349f854ea
224a57d812ad75deb81b33aaee4e5774
0b84aa1a5200747035ae2731e303a55b
16487eabe060c81c1b26a26121ea647c
0e744856e13b0b3bab63b7f29cea9efc
5f3df66b7325bc197b162636cc82e163
3943d234d5e97a34c47e925917502d2b
b3fcbccc7a5a9b425557dce0e53eb670
a5b5e80eef2e1c8ec7b6f6928e6ae79e
d04fa9c789e96279e60829a7c1453528
00546cf6a006a05a7f1a27b6076a02da

[9] e7f3a06aeb5ecd0109a0a847616de7c
    c5176e19fc7327434e7b83fdc41374e7
    8792aa901df1376d2b67edb77f9808e0
    912450e66eabc7cacbe28ab61b144625
    d3aad689bfe73773a88519f56bdd3c37
    9ed928e1fd4c31542c256d220a72360c
    b1aea397238ad6a1a47394b228f0a1c1
    326e58c95a5b352e3d1e6589da8f6347
    89045d2431714599035c4431e4686b53
    ef1b8096c5113d25bab71ef68e0ee5fd

    for line in Lines:
    e = AES.new(key, AES.MODE_CBC, iv)
    key = (line.lower())
    key = binascii.unhexlify(key[0:32])
    #print(len(key))
    iv = binascii.unhexlify('09080706050403020100A2B2C2D2E2F2'.lower())
    plaintext = binascii.unhexlify('255044462d312e350a25d0d4c5d80a34')
    ciphertext = e.encrypt(plaintext)
    #print(plaintext)
    if(ciphertext==binascii.unhexlify('d06bf9d0dab8e8ef880660d2af65aa82')):
    print ('Key that gives the matching plaintext-ciphertext is: ')
    print ('key: ', binascii.hexlify(key))
    print ('ciphertext: ', binascii.hexlify(ciphertext))

Key that gives the matching plaintext-ciphertext is:
key: b'98ca4699fe3bba5ead0c7b478b013754'
ciphertext: b'd06bf9d0dab8e8ef880660d2af65aa82'

```

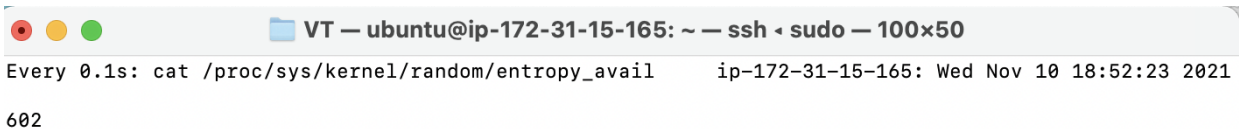
Figure 5: Python code

In Figure 5, key.txt file is opened and read line by line and passed to the variable key. In the key.txt file, for some reason the lines were taken as the key and an additional space character. In

order to eliminate the space character and just get the key I used key[0:32]. Hence each line is a 16 byte session key. In the for loop all the possible keys are used to AES 128 CBC encrypt the plaintext and the one that matches with the known ciphertext is printed out.

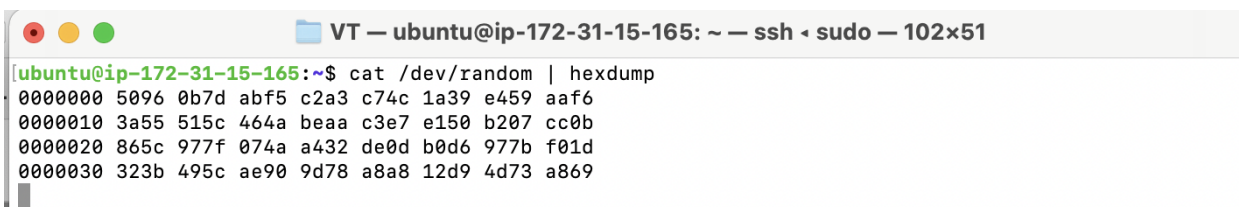
Task 3.

The entropy of the kernel at the moment is 602. But when we generate random numbers as in Figure 7, it drops dramatically which can be seen in Figure 8. When the entropy pool is empty, one can increase the entropy again with moving the mouse and pressing the keyboard. The result of this can be seen in Figure 9.



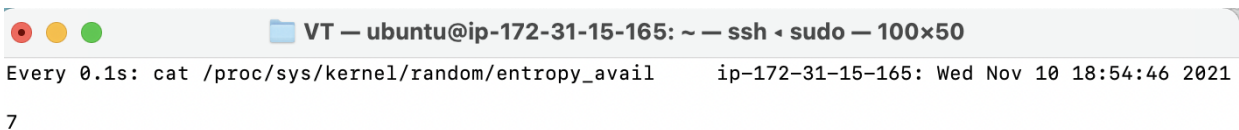
```
VT — ubuntu@ip-172-31-15-165: ~ — ssh ◀ sudo — 100x50
Every 0.1s: cat /proc/sys/kernel/random/entropy_avail    ip-172-31-15-165: Wed Nov 10 18:52:23 2021
602
```

Figure 6 : Entropy of the kernel



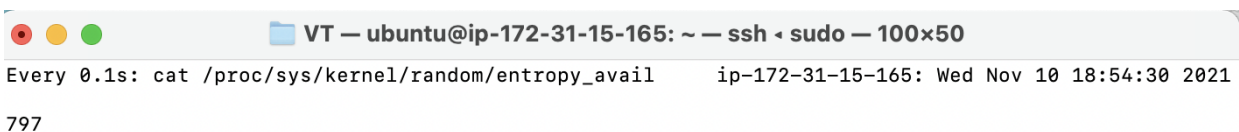
```
VT — ubuntu@ip-172-31-15-165: ~ — ssh ◀ sudo — 102x51
[ubuntu@ip-172-31-15-165:~$ cat /dev/random | hexdump
00000000 5096 0b7d abf5 c2a3 c74c 1a39 e459 aaf6
00000010 3a55 515c 464a beaa c3e7 e150 b207 cc0b
00000020 865c 977f 074a a432 de0d b0d6 977b f01d
00000030 323b 495c ae90 9d78 a8a8 12d9 4d73 a869
```

Figure 7 : Random numbers are generated



```
VT — ubuntu@ip-172-31-15-165: ~ — ssh ◀ sudo — 100x50
Every 0.1s: cat /proc/sys/kernel/random/entropy_avail    ip-172-31-15-165: Wed Nov 10 18:54:46 2021
7
```

Figure 8: Entropy drop after random number generation



```
VT — ubuntu@ip-172-31-15-165: ~ — ssh ◀ sudo — 100x50
Every 0.1s: cat /proc/sys/kernel/random/entropy_avail    ip-172-31-15-165: Wed Nov 10 18:54:30 2021
797
```

Figure 9: Entropy pool filled up after mouse and keyboard action

Task 4.

```

Every 0.1s: cat /proc/sys/kernel/random/entropy... VM: Wed Nov 10 15:31:11/10/21 seed@vm:~$ cat /dev/random | hexdump
61
00000000 ec80 8855 bf14 0470 df96 504e 2cfd b2f4
00000010 8c3e c90e b2e0 d5fb 9634 2579 7139 382c
00000020 3f50 ea53 41cd ecad 0246 18e7 8d90 60c5
00000030 c2e3 75cd 80cd 619b 1512 6776 8286 ba4c
00000040 1faf 9ed5 fb87 4599 83c5 f219 bf51 58a3
00000050 d9eb 3bc9 3587 0087 07ea 822c cbf8 5836
00000060 91db e173 3c11 93f4 e819 2c88 8396 0193
00000070 37b6 681a 061d 7745 3109 f0ed 5010 4d47
00000080 8b7e 1978 6085 0e9e 021b c39c 9409 ff8c
00000090 1c90 6802 419e 8eee 3e43 c449 0b62 7a60
000000a0 9aed 1d5f 7fe2 7f25 d00a 9f61 6028 21bc
000000b0 2ddc 5ae1 7331 b37b aacb 6739 faa1 5bfb
000000c0 a6c9 279f 6452 98a8 e1d8 0481 a434 229a
000000d0 1333 930e f292 696d 5d26 a126 d618 90f5
000000e0 afae 0f65 9629 34af c942 c311 9bf5 a335
000000f0 39ab 4e2d a832 f2de ec47 b26c 2615 dede
0000100 696f 62be 16fe 5aa9 fb21 ce63 37f1 8b84
0000110 15d2 d943 3a96 33dd 1464 6617 1964 5257
0000120 9c56 a1cb 3a75 ac29 9e72 57d2 0e0a b66f

Every 0.1s: cat /proc/sys/kernel/random/entropy... VM: Wed Nov 10 15:41:11/10/21 seed@vm:~$ cat /dev/random | hexdump
2
00000000 ec80 8855 bf14 0470 df96 504e 2cfd b2f4
00000010 8c3e c90e b2e0 d5fb 9634 2579 7139 382c
00000020 3f50 ea53 41cd ecad 0246 18e7 8d90 60c5
00000030 c2e3 75cd 80cd 619b 1512 6776 8286 ba4c
00000040 1faf 9ed5 fb87 4599 83c5 f219 bf51 58a3
00000050 d9eb 3bc9 3587 0087 07ea 822c cbf8 5836
00000060 91db e173 3c11 93f4 e819 2c88 8396 0193
00000070 37b6 681a 061d 7745 3109 f0ed 5010 4d47
00000080 8b7e 1978 6085 0e9e 021b c39c 9409 ff8c
00000090 1c90 6802 419e 8eee 3e43 c449 0b62 7a60
000000a0 9aed 1d5f 7fe2 7f25 d00a 9f61 6028 21bc
000000b0 2ddc 5ae1 7331 b37b aacb 6739 faa1 5bfb
000000c0 a6c9 279f 6452 98a8 e1d8 0481 a434 229a
000000d0 1333 930e f292 696d 5d26 a126 d618 90f5
000000e0 afae 0f65 9629 34af c942 c311 9bf5 a335
000000f0 39ab 4e2d a832 f2de ec47 b26c 2615 dede
0000100 696f 62be 16fe 5aa9 fb21 ce63 37f1 8b84
0000110 15d2 d943 3a96 33dd 1464 6617 1964 5257
0000120 9c56 a1cb 3a75 ac29 9e72 57d2 0e0a b66f
0000130 c316 b850 1bc3 f44c 842c c7e0 903c 7bb7
0000140 ae43 79a0 4145 967d 5e84 0f47 6fcf f8c1
0000150 f99f 4b8d 66e9 9f2f e92d 73e8 5da6 465e
0000160 c19e 9cad 564e 1be0 d529 5949 4202 2a41
0000170 5b09 61a0 fc39 628a 0637 4818 dffa 656d
0000180 c072 1503 d5c6 9e81 06b1 d01b bbed 523c

```

Figure 10: /dev/random demonstration

/dev/random waits until it has received sufficient entropy or blocks the system until it gains entropy and the entropy pool fills up to some point. The entropy of a computer is finite. Along with other ways using the keyboard and the mouse is the simplest way to increase the entropy of the computer.

In Figure 10, one can observe how the entropy of the system decreases when random numbers are being generated.

If we use the /dev/random as the server to generate random session keys to the client, after some point it is very possible that the entropy pool will be emptied. Meaning there is a limited amount of clients the system can serve in a specific time period. Hence if one can continuously request a new truly random number as a client, they can overwhelm the system, empty the entropy pool and cause DoS so the main clients/users cannot access it anymore.

Task 5.

```

61898a0 ff 5c 0a bc d2 1b fd 26 da 64 38 9a 5c 84 f4 8f
61898b0 bf f4 b9 b4 47 1d fb a2 58 ef 9f e5 34 20 c4 0d
61898c0 0b 13 c4 63 9c 44 de 99 79 db 3b b1 e2 e4 a7 4b
61898d0 ed ce 24 4d 16 6d 39 b3 c5 dd a8 6d 5e ec c2 38
61898e0 e6 c0 e1 8a 25 d7 12 3b 17 ce 93 1d 9b 5e 88 e6
61898f0 f9 88 16 16 fa b0 c7 af fb 53 5b 1c ce 3f 7e 4e
6189900 64 b2 f9 4a c4 97 7e 8e 04 b0 cc 1f e4 38 14 33
6189910 57 4c 39 28 11 d7 84 e7 e8 a0 b4 ec 5e 98 9b 2f
6189920 51 d5 ca ae 8f 5b c9 39 a6 0f 1b d6 e5 f5 38 e3
6189930 3b ee f7 0d 36 91 74 9b ce 8a 12 d5 05 c9 82 47
6189940 14 6f ea 92 2d 22 cb df 71 1d a7 b7 be 26 1c 9f
6189950 6f 99 6e 58 8c e6 c4 d9 d0 25 1c 28 f3 b3 50 a4
6189960 38 99 64 bf 4f 9e af ea 36 7d e5 e9 10 ef b5 22
6189970 ea f5 b9 41 94 57 da ed f7 38 20 5b 34 d9 25 06
6189980 75 92 02 9c c8 05 18 7b 7a 91 26 bf 3e 10 2d 46
6189990 56 74 8e 74 8e 19 54 1e 58 6e 2a fe c7 b4 aa 86
61899a0 fd 36 f9 66 61 61 ea 93 4b 64 b5 d1 a5 2e 2b d8
61899b0 0c f1 bb cd 59 5f 7f 6a a6 36 fb a9 55 3b f5 9e
61899c0 15 2e a9 9e 1b dd 1e 82 54 a1 5b b4 22 6e 79 02
61899d0 ab fd 0d 91 8a 0e 0b 8f 9c b9 59 45 0c cb 37 73
61899e0 01 f2 bb 62 bd ab 8e ae f6 ba ff a3 11 f0 39 d1
61899f0 40 fa b1 bb 31 6f cd 6a 90 3c 4e 32 92 fa 31 17
6189a00 41 aa 74 e3 1b 07 4c fa 47 e5 7e 02 6d 98 2b 74
6189a10 00 0a 5f f0 93 0e 0a 26 93 d5 88 0c 42 c1 3e 61
6189a20 b0 ec 45 31 07 76 5f 38 e0 4b b1 67 63 07 23 a2
6189a30 99 f7 8b f1 20 db 2c 3d 2a 80 de 96 de 71 96 e8
6189a40 c5 3f 80 4d ab e9 87 7a 29 06 64 77 82 61 db 4a
6189a50 3b 4e 24 cf ff cb 1b ff c6 dc da 65 c4 02 25 9c
6189a60 a1 2f 94 07 c0 63 0d ee 7c 75 be 9c 34 43 cf 55^C
denizaytemiz@Denizs-MacBook-Air ~ % head -c 1M /dev/urandom > output.bin
head: illegal byte count -- 1M
denizaytemiz@Denizs-MacBook-Air ~ % head -c 1000000 /dev/urandom > output.bin
denizaytemiz@Denizs-MacBook-Air ~ %

```

Figure 11: /dev/urandom for random number generation

I observed no significant change with the mouse movement. This is expected since urandom does not depend on system entropy but rather creates pseudorandom numbers.

```

denizaytemiz@Denizs-MacBook-Air ~ % ent output.bin
Entropy = 7.999816 bits per byte.

Optimum compression would reduce the size
of this 1000000 byte file by 0 percent.

Chi square distribution for 1000000 samples is 254.49, and randomly
would exceed this value 49.72 percent of the times.

Arithmetic mean value of data bytes is 127.5742 (127.5 = random).
Monte Carlo value for Pi is 3.139260557 (error 0.07 percent).
Serial correlation coefficient is -0.001646 (totally uncorrelated = 0.0).
denizaytemiz@Denizs-MacBook-Air ~ %

```

Figure 12: Entropy test results of the output file.

In Figure 12 one can observe that the pseudorandom numbers pass the entropy test, they look pretty random.

```
[2] import os
```

```
[3] size= 32
    result = os.urandom(size)
```

```
▶ print(result.hex())
```

```
80bf0c2548d82956ba76d49b180ab0097bf88957dbfdd9d427827d7c197ede84
```

Figure 13: The code is modified for 256 bit encryption key

```
[43] size = 10^7
      result = os.urandom(size)
      result.hex()
```

```
'71d80ce429463e31f190df9713'
```

```
▶ size = 10^8
   result = os.urandom(size)
   result.hex()
```

```
'd9fc'
```

Figure 14: The limit of bits allocated

In Figure 14 one can observe the limit for bit size is 10^7 since it still gives a valid value. However when it is increased to 10^8 it returns an error.