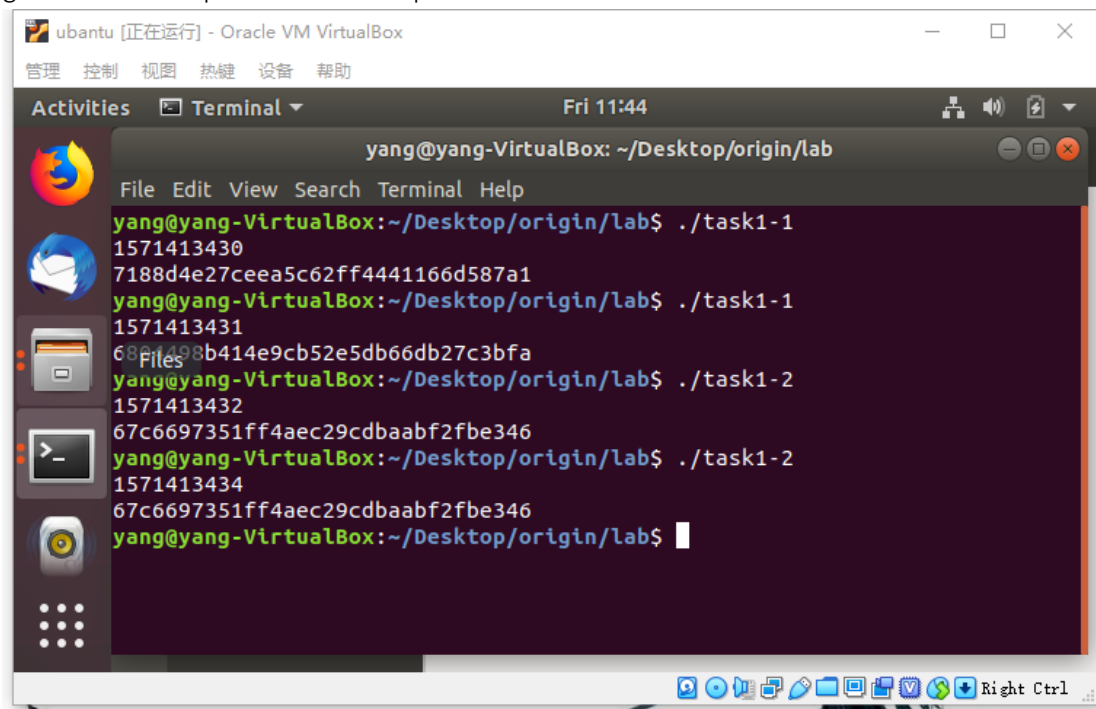


1 Task1: Generate Encryption Key in a Wrong Way

The task1-1 is the original code's output, and the task1-2 is the code that command out the "srand()" line. I found that in task1-1 the key will change with the timestamp but in the task1-2 the key will be the same no matter whether the time change. "time(NULL)" is the function to give a timestamp, and the "srand()" use the current timestamp as seed to generate random number. If command out the line, the seed will be the same, which means the random number generate will output that same sequence of number.



```
yang@yang-VirtualBox: ~/Desktop/origin/lab
yang@yang-VirtualBox:~/Desktop/origin/lab$ ./task1-1
1571413430
7188d4e27ceea5c62ff4441166d587a1
yang@yang-VirtualBox:~/Desktop/origin/lab$ ./task1-1
1571413431
6884498b414e9cb52e5db66db27c3bfa
yang@yang-VirtualBox:~/Desktop/origin/lab$ ./task1-2
1571413432
67c6697351ff4aec29cdbaabf2fbe346
yang@yang-VirtualBox:~/Desktop/origin/lab$ ./task1-2
1571413434
67c6697351ff4aec29cdbaabf2fbe346
yang@yang-VirtualBox:~/Desktop/origin/lab$
```

2 Task2: Guessing the Key

The following is my Python code to break the key. The code use the timestamp as seed to generate key, and then traverse all key to encrypt the plaintext message. Comparing the output and the ciphertext, if they are the same, it means the key generator get the correct key that Alice used. As shown below, the key is 2129322710651590.

```
import sys
import time
import random
from Crypto.Cipher import AES
from binascii import b2a_hex, a2b_hex
plaintext="255044462d312e350a25d0d4c5d80a34"
ciphertext="d06bf9d0dab8e8ef880660d2af65aa82"
iv = "09080706050403020100A2B2C2D2E2F2"

def getST(string):
    timeArray = time.strptime(string, "%Y-%m-%d %H:%M:%S")
    timestamp = time.mktime(timeArray)
    return timestamp

dt1 = getST("2018-04-17 21:08:49")
dt2 = getST("2018-04-17 23:08:49")
```

```

class prpcrypt():
    def __init__(self, key, iv):
        self.key = key
        self.iv = iv
        self.mode = AES.MODE_CBC

    def encrypt(self, text):
        cryptor = AES.new(self.key, self.mode, self.iv)
        length = 16
        count = len(text)
        if (count % length != 0) :
            add = length - (count % length)
        else:
            #add = 0
            #text = text + ('\0' * add)
            self.ciphertext = cryptor.encrypt(text)
        return b2a_hex(self.ciphertext)

    def decrypt(self, text):
        cryptor = AES.new(self.key, self.mode, self.iv)
        plain_text = cryptor.decrypt(a2b_hex(text))
        return plain_text.rstrip('\0')

if __name__ == '__main__':
    for i in range(int(dt1), int(dt2)):
        try:
            #print(i)
            random.seed(int(i))
            key = int(random.random()*1000000000000000)
            pc = prpcrypt(str.encode(str(key)), str.encode(iv[:16]))
            e = pc.encrypt(str.encode(plaintext))
            if e == ciphertext:
                break
        except Exception as e:
            print(e)
    print(key)

```

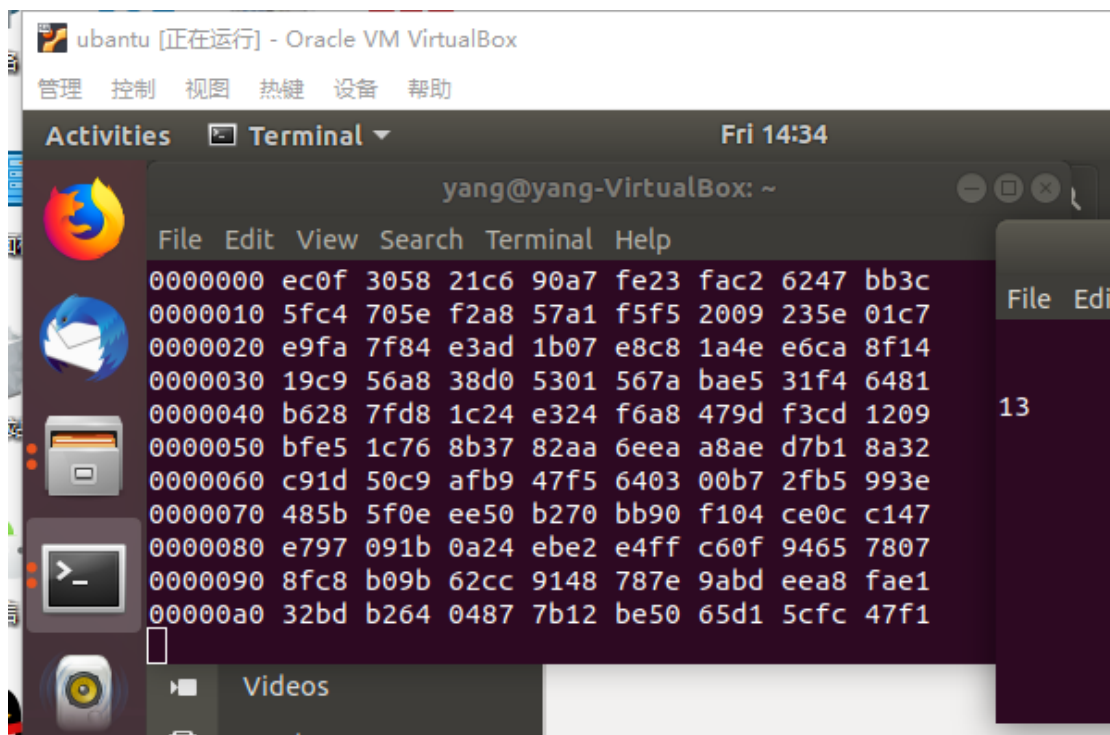
Incorrect AES key length (14 bytes)
2129322710651590

2.3 Task3: Measure the Entropy of Kernel

When moving the mouse, the number increased by 2 or 3 each time. The click action will make number increase by 3 or 4. The typing will make number increase 5. The reading would not make entropy increase unless using mouse or keyboard to turn page. And visiting a website will make number keep going by 1 without other actions. If we only consider the step of the number, typing increased the entropy significantly, but if we consider the all action we need to do, visiting a website will be the champion.

2.4 Task4: Get Pseudo Random Numbers from /dev/random

If we do not do any action, the number will only increase 1 each time, and if we randomly move mouse the number will increase much faster. No matter how faster the number increases, it will go back to 1 once it reach 64, with the increase in first 6 bit of output of hexdump. The output of hexdump is hex. The reason that number increase faster is that the increase is the sum of default number and the number triggered by actions.



2.5 Task5: Get Random Numbers from /dev/urandom

In this case, the number generate very fast, and it seems that there is no affect by the moving the mouse on the outcome. The following is the outcome of random number analysis.

```
yang@yang-VirtualBox:~$ ent output.bin
Entropy = 7.999836 bits per byte.

Optimum compression would reduce the size
of this 1048576 byte file by 0 percent.
Terminal
Chi square distribution for 1048576 samples is 239.16, and randomly
would exceed this value 75.39 percent of the times.

Arithmetic mean value of data bytes is 127.5553 (127.5 = random).
Monte Carlo value for Pi is 3.141918724 (error 0.01 percent).
Serial correlation coefficient is -0.000706 (totally uncorrelated = 0.0).
yang@yang-VirtualBox:~$
```

It shows that the quality of random number is good, because the number it generate is close to the truly random number.

The following is the code to generate a 256-bit encryption key.

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #define LEN 32 // 256 bits
4  void main(){
5      unsigned char *key = (unsigned char *)malloc(sizeof(unsigned char)*LEN);
6      FILE* random = fopen("/dev/urandom", "r");
7      fread(key, sizeof(unsigned char)*LEN, 1, random);
8      fclose(random);
9      for( int i = 0; i < LEN; i++ )
10     {
11         printf( "%02X ", key[i] );
12     }
13 }
```

This screenshot shows the key that generate by the code.

```
yang@yang-VirtualBox:~/Desktop/origin/lab$ ./task5
4F 29 2D 34 1A B9 BB DE D8 84 AC 5D 25 CA 52 01 51 FD 84 FB 8D BE 29 89 4D 85 59 99
71 45 41 49 yang@yang-VirtualBox:~/Desktop/origin/lab$ ./task5
86 61 16 B1 7F FF 4A D9 B0 30 F3 2B B3 8D 5B 08 CB 0A 60 9D FF 4C A6 AD AE 59 28 8A
FF AD 02 C6 yang@yang-VirtualBox:~/Desktop/origin/lab$ ./task5
A7 Show Applications 12 58 9A 61 FB 25 1A 86 BF A2 68 E2 27 CB 7A C2 26 77 00 54 78 0F
41 1E 13 F0 yang@yang-VirtualBox:~/Desktop/origin/lab$
```