

Assignment 01

60847013S 資工碩一 蘇冠中

1.1 prove Chain rule: $H(X, Y) = H(X) + H(Y|X)$

$$\begin{aligned} H(X, Y) &= -\sum_{x \in X} \sum_{y \in Y} p(x, y) \log_2(p(x, y)) \\ &= -\sum_{x \in X} \sum_{y \in Y} p(x, y) \log_2(p(x)p(y|x)) \\ &= -\sum_{x \in X} \sum_{y \in Y} p(x, y) \log_2(p(x)) - \sum_{x \in X} \sum_{y \in Y} p(x, y) \log_2(p(y|x)) \\ &= -\sum_{x \in X} p(x) \log_2(p(x)) - \sum_{x \in X} \sum_{y \in Y} p(x, y) \log_2(p(y|x)) \\ &= H(X) + H(Y|X) \end{aligned}$$

1.2 $H(X_1, X_2, \dots, X_n) \leq \sum_{i=1}^n H(X_i)$ with equality if and only if the random variables X_i are independent.

呈 1.1 題, $H(X, Y) = H(X) + H(Y|X)$

當 X, Y 是 independent 時, $H(Y|X) = H(Y)$

則 $H(X, Y) = H(X) + H(Y)$

而 Chain rule 可以容易地推廣到更多的變數集合

且可得到 $H(X_1, X_2, \dots, X_n) = \sum_{i=1}^n H(X_i|X_{i-1}, \dots, X_1)$

$$= H(X_1) + H(X_2|X_1) + \dots + H(X_i|X_{i-1}, \dots, X_1)$$

而當 $H(X_1, X_2, \dots, X_n)$ 的 X_i 都是 independent 時

$$\begin{aligned} H(X_1, X_2, \dots, X_n) &= H(X_1) + H(X_2) + \dots + H(X_n) \\ &= \sum_{i=1}^n H(X_i) \quad \text{得證} \end{aligned}$$

2.1 $E'_k(m) = E_k(m) || 0$

Yes,

2.2 $E'_k(m) = E_k(m) || LSB(m)$

yes

2.3 $E'_k(m) = E_k(m) || E_k(m)$

yes

2.4 $E'_k(m) = reverse(E_k(m))$

yes

2.5 $E'_k(m) = E_k(m) || k$

Yes

3. Frequency Analysis Attack

執行 fre.java 檔案，該檔案進行：

1. 將密文 frequency_attack_cipher_example.txt 讀取，並且計算英文字母的出現次數、以及英文詞彙的出現次數，將此兩項存入 temp.txt 檔中。
2. 依照最常出現的英文字母頻率順序、以及最常出現的英文詞彙進行對照(像是 the, is, in, a, of……)，一一轉換過去，[nby=the], [uhx=and], [u=a], [ch=in], [iz=of], 將最多出現的先進行猜測轉換，並且與單一個字母也對照後，慢慢地再將其他字母轉換回來。
3. 將已知的單字先加入自製的轉換表中，並且讀取密文檔，依照已有的轉換表去解回明文，並將名文中某些轉換不完全的詞彙去做猜測，繼續增加轉換表的單字，一個一個解到 26 個單字母都解完為止

```
fre.java  break.txt  temp.txt
1 y 1196
2 n 880
3 u 819
4 h 783
5 i 679
6 c 636
7 l 586
8 m 563
9 b 553
10 x 486
11 f 321
12 w 270
13 o 265
14 g 262
15 a 259
16 q 223
17 z 209
18 s 167
19 v 160
20 j 158
21 p 70
22 e 51
23 r 21
24 k 7
25 t 6
26 d 1
27 ----- 單字母結束 -----
28 uhx 201
29 nby 191
30 iz 90
31 vyaun 41
32 ch 36
33 qum 30
34 u 27
```

```
ArrayList<brewd> bd = new ArrayList<brewd>(); //開始加入轉換字串
brewd b1 = new brewd("y","e"); bd.add(b1); //依照字母出現頻率，以及單字出現頻率去做比對
brewd b2 = new brewd("u","a"); bd.add(b2);
brewd b3 = new brewd("c","i"); bd.add(b3);
brewd b4 = new brewd("h","n"); bd.add(b4);
brewd b5 = new brewd("x","d"); bd.add(b5);
brewd b6 = new brewd("n","t"); bd.add(b6);
brewd b7 = new brewd("b","h"); bd.add(b7);
brewd b8 = new brewd("w","c"); bd.add(b8);
brewd b9 = new brewd("g","m"); bd.add(b9);
brewd b10 = new brewd("v","b"); bd.add(b10);
brewd b11 = new brewd("a","g"); bd.add(b11);
brewd b12 = new brewd("q","w"); bd.add(b12);
brewd b13 = new brewd("m","s"); bd.add(b13);
brewd b14 = new brewd("l","r"); bd.add(b14);
brewd b15 = new brewd("o","u"); bd.add(b15);
brewd b16 = new brewd("z","f"); bd.add(b16);
brewd b17 = new brewd("s","y"); bd.add(b17);
brewd b18 = new brewd("i","o"); bd.add(b18);
```

```
fre.java  break.txt  temp.txt  frequency_attack_cipher_example.txt
1 in the beginning turing created the machine.
2
3 and the machine was cruffy and bogacious, existing in theory only. and von neumann
4
5 and von neumann spoke unto the architecture, and blessed it, saying, "go forth and
6
7 the first systems were mighty giants; many great works of renown did they accomplis
8
9 now the sons of marketing looked upon the children of turing, and saw that they wer
10
11 and the systems and their corporations replicated and grew numerous upon the earth.
12
13 now it came to pass that the spirits of turing and von neumann looked upon the earti
14
15 and that day the spirits of turing and von neumann spake unto moore of intel, grant:
16
17 and the birth of 4004 was the beginning of the third age, the age of microchips. an
18
19 moore begat intel. intel begat mostech, zilog and atari. mostech begat 6502, and zi
20
21 now it came to pass in the age of microchips that ibm, the greatest of the mainfram
22
23 and ibm came unto microsoft, who licensed unto them qdos, the child of cp/m and 808
24
25 in the fullness of time ms-dos begat windows. and this is the lineage of windows: c
26
27 now it came to pass that microsoft had waxed great and mighty among the microchip c
```

全部解完寫入 break.txt 中，break.txt 即為明文

4. Never Use One Time Pad Twice

執行 one. java 檔案，該檔進行：

1. 讀取各個 cipher1~10，並存入 Array 中，依照題目給的題示，當空白” ” 與英文字進行 XOR 時，結果會是該英文字的大小寫對換，因此假設 $\text{key XOR } m1 = c1, \text{key XOR } m2 = c2, \dots$ ，則若 $m1 \text{ XOR } m2$ 為英文字時，可知 $m1$ 或 $m2$ 該 byte 位置其中一個為空白” ”。
2. 因此將 cipher1 與其他 9 個 cipher 進行 XOR，計算每個 byte 位置出現英文字的次數，該位置出現英文字的次數越高，代表 cipher1 該位置的原文越有可能是” ”空白，則將” ”空白與 cipher 的密文進行 XOR，即可得出該位置加密的 key。
3. 再來將 cipher2 與其他 9 個也做 XOR，直到所有 cipher 都做完 XOR 後，一每個 cipher 解出來的不同位置的 key，寫入 key.txt 檔中。

執行 one2. java 檔案，該檔執行：

1. 讀取 key.txt 檔，以及 challenge.txt 檔，將 challenge 依照可的 key 去進行解密，首先得到[** m1**le**how p*rfecx the cri*e, as *ong*as*pe*p*e*do, t*e*e *s no *ol*ti*n do*’ t apen.]的解密文，可以猜得出的單字文 how perfect the crime, as long as people do, there is no solution don’ t open.。
2. 先依照前面英文的語法判斷，前面兩個字可能為 no matter，依照這兩個字解回去的 key，再與其他 cipher 進行解密後發現所有原文都是解開的，因此再將 challenge 解密完的的 key(使用 test2. java 檔轉成 key)，代回去所有的 cipher，得到所有 cipher 解密的明文，並且都是正確的，則 challenge 解出來的明文即為[no matter how perfect the crime, as long as people do, there is no solution don’ t open.]。

```
24
25 ArrayList<Integer> a1 = new ArrayList<Integer>();
26 read2(a1,"challenge.txt"); //讀取密文
27 for(int i=0;i<a1.size();i++) {
28     if(a[i]!=null) {
29         int c = a1.get(i) ^ Integer.parseInt(a[i], 16);
30         a1.set(i, c); //將密文與key進行XOR，進行解密
31     }
32     else {
33         a1.set(i, 0); //若某byte尚未有key，則不XOR
34     }
35 }
36
37 Console 11
<terminated> one2 [Java Application] C:\Program Files\Java\jre1.8.0_191\bin\java.exe (2020年3月28日 下午10:39:30)
84 6
85 69
86 23
89 a5
90 1
** m1*le**how p*rfecx the cri*e, as *ong*as*pe*p*e*do, t*e*e *s no *ol*ti*n do*’ t apen.
```

```
29 int c = a1.get(i) ^ Integer.parseInt(a[i], 16);
30 a1.set(i, c); //將密文與key進行XOR，進行解密
31 }
32 else {
33     a1.set(i, 0); //若某byte尚未有key，則不XOR
34 }
35 }
36
37 Console 11
<terminated> one2 [Java Application] C:\Program Files\Java\jre1.8.0_191\bin\java.exe (2020年3月28日 下午10:42:37)
82 1d
83 ef
84 6
85 69
86 23
in security sciences, trust is any unauthenticated interactivity between targets
87
88 Console 11
<terminated> one2 [Java Application] C:\Program Files\Java\jre1.8.0_191\bin\java.exe (2020年3月28日 下午10:43:20)
82 1d
83 ef
84 6
85 69
86 23
no matter how perfect the crime, as long as people do, there is no solution don't open.
```

5. Pseudo Random Number Generator

Task1: Generate Encryption Key in a Wrong Way

```
wrong.cpp
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4 #define KEYSIZE 16
5
6 int main()
7 {
8     int i;
9     char key[KEYSIZE];
10    printf("srand(%d), (long long) time(NULL));\n", (long long) time(NULL));
11    // srand(time(NULL));
12
13    for (i = 0; i < KEYSIZE; i++){
14        key[i] = rand()%256;
15        printf("%02x", (unsigned char)key[i]);
16    }
17    printf("\n");
18 }
```

```
Terminal
[03/28/20]seed@VM:~/Desktop$ ./wrong
1585411345
48d8b5c4165a3d0008037741aad5410
[03/28/20]seed@VM:~/Desktop$ ./wrong
1585411347
5abd395c3ea61f2244edf558ad08bac5
[03/28/20]seed@VM:~/Desktop$ ./wrong
1585411348
7766aa6e7e201cb72038e9e4769e0c38
[03/28/20]seed@VM:~/Desktop$ ./wrong
1585411351
6a49d9e54c632960119b251995f36be1
[03/28/20]seed@VM:~/Desktop$
[03/28/20]seed@VM:~/Desktop$
[03/28/20]seed@VM:~/Desktop$ ./wrong
1585411366
67c6697351ff4aec29cdbaabf2fbe346
[03/28/20]seed@VM:~/Desktop$ ./wrong
1585411368
67c6697351ff4aec29cdbaabf2fbe346
[03/28/20]seed@VM:~/Desktop$ ./wrong
1585411371
67c6697351ff4aec29cdbaabf2fbe346
[03/28/20]seed@VM:~/Desktop$
```

先執行 lab 中的程式，可得到在不同時間，time(NULL)執行產生的結果都不同，執行的第一行是 time(NULL)，第二行是產生的 key，執行四次後，將 srand(time(NULL))註解掉，可看到後三次執行雖然 time(NULL)的部分不一樣，但是產生出來的 key 是一樣的。

因此可知，srand()的目的是要設定 seed，且代入的是 time()，因每次代入的 time()會隨時間改變，因此可以產生出不同的亂數 key

。

Task2: Guessing the Key

題目使用的 key 產生器為 Task1 的程式，因此我將 wrong.cpp 改成用 terminator 執行時後面加參數代入，srand(t)即為代入的參數。

依據題目的提示，srand(t)代入的時間點為[2018-04-17 21:08:48]~

[2018-04-17 23:08:49]這兩個小時中間，因此多寫一個 find.sh 的 shell script 檔，該檔從 2018-04-17 的 21:08:48 每次加一秒到 23:08:49，代入 wrong 去產生金鑰。

```
wrong.cpp
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4 #define KEYSIZE 16
5
6 void main(int argc, char** argv)
7 {
8     int i;
9     char key[KEYSIZE], *epr;
10    long t = strtoul(argv[1], &epr, 10);
11    srand(t);
12
13    for (i = 0; i < KEYSIZE; i++){
14        key[i] = rand()%256;
15        printf("%02x", (unsigned char)key[i]);
16    }
17    printf("\n");
18 }
```

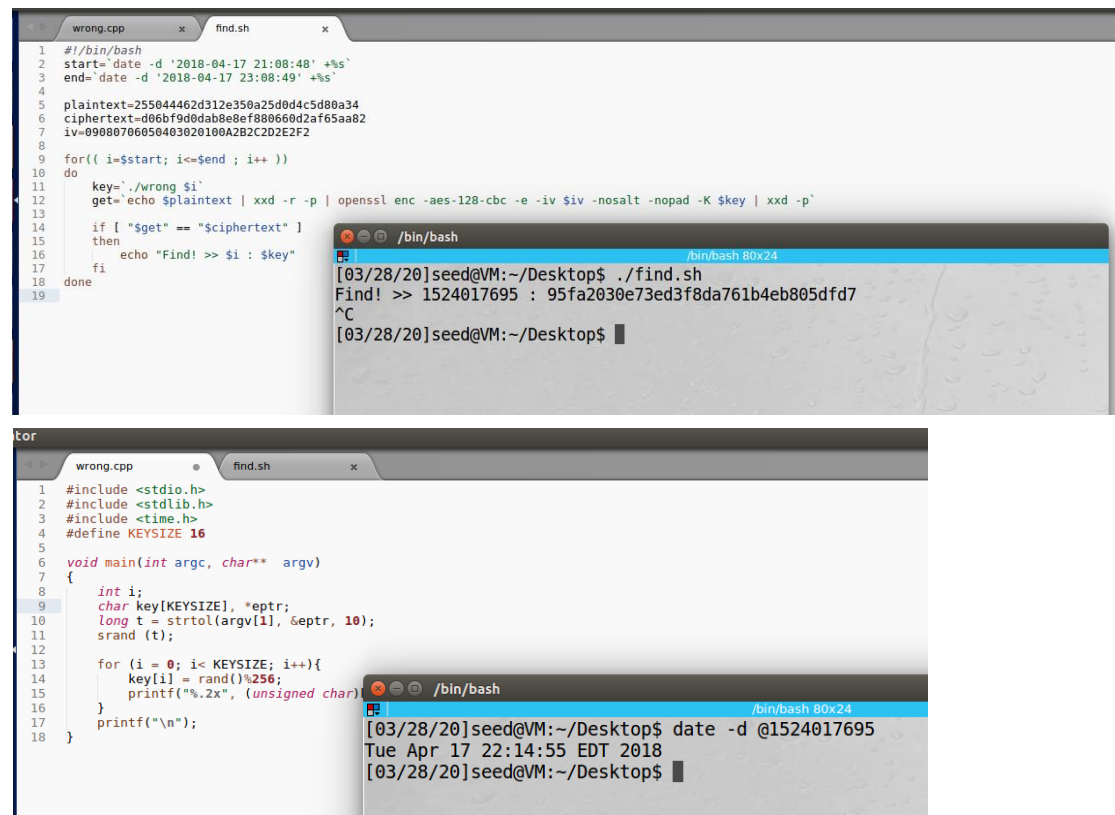
```
find.sh
1 #!/bin/sh
2 for i in $(seq 21:08:48 23:08:49); do
3     ./wrong $i
4 done
```

且加入 Ubuntu 內建的 openssl aes-128-cbc 進行解密，輸入 plaintext、ciphertext、iv，當我的 wrong 產生的金鑰去加密 plaintext 後與 ciphertext 結果相同時，則將該時間點秒數以及加密金鑰印出來。

執行結果顯示金鑰為： 95fa2030e73ed3f8da761b4eb805dfd7

時間點為： 2018-04-17-22:14:55

確實是在 2 小時的範圍內。



```
1 #!/bin/bash
2 start=date -d '2018-04-17 21:00:48' +%s
3 end=date -d '2018-04-17 23:00:49' +%s
4
5 plaintext=25504446d312e350a25d0d4c5d80a34
6 ciphertext=d06bf9d0dab8e8ef880660d2af65aa82
7 iv=09080706050403020100A2B2C2D2E2F2
8
9 for(( i=$start; i<=$end ; i++ ))
10 do
11     key=$(./wrong $i)
12     get=$(echo $plaintext | xxd -r -p | openssl enc -aes-128-cbc -e -iv $iv -nosalt -nopad -K $key | xxd -p)
13
14     if [ "$get" == "$ciphertext" ]
15     then
16         echo "Find! >> $i : $key"
17     fi
18 done
```

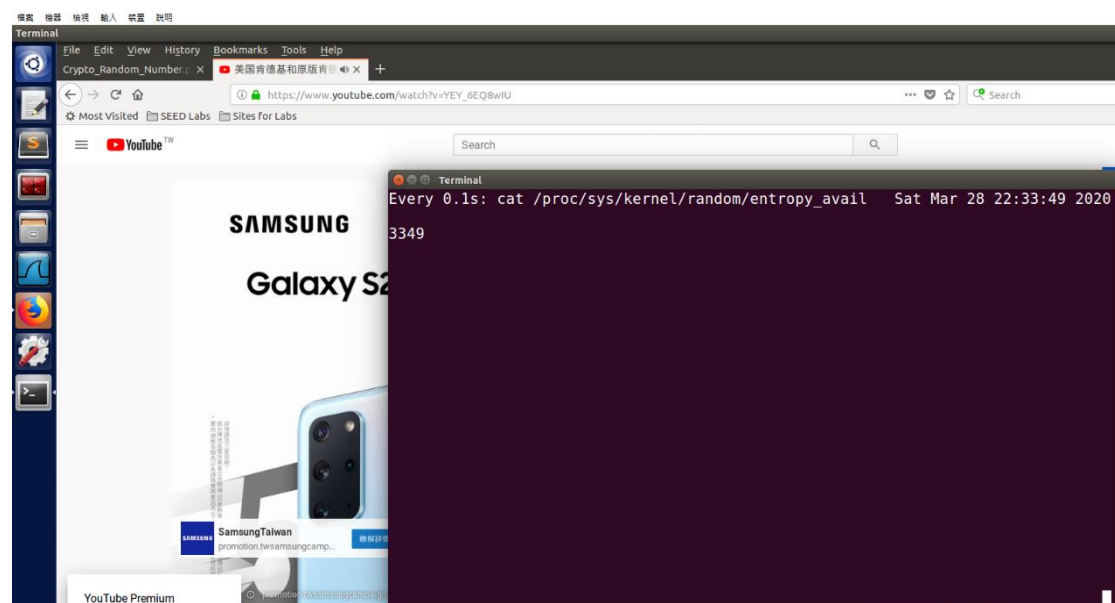
```
[03/28/20]seed@VM:~/Desktop$ ./find.sh
Find! >> 1524017695 : 95fa2030e73ed3f8da761b4eb805dfd7
^C
[03/28/20]seed@VM:~/Desktop$
```

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4 #define KEYSIZE 16
5
6 void main(int argc, char** argv)
7 {
8     int i;
9     char key[KEYSIZE], *eptr;
10    long t = strtoul(argv[1], &eptr, 10);
11    srand (t);
12
13    for (i = 0; i < KEYSIZE; i++){
14        key[i] = rand()%256;
15        printf("%.2x", (unsigned char)key[i]);
16    }
17    printf("\n");
18 }
```

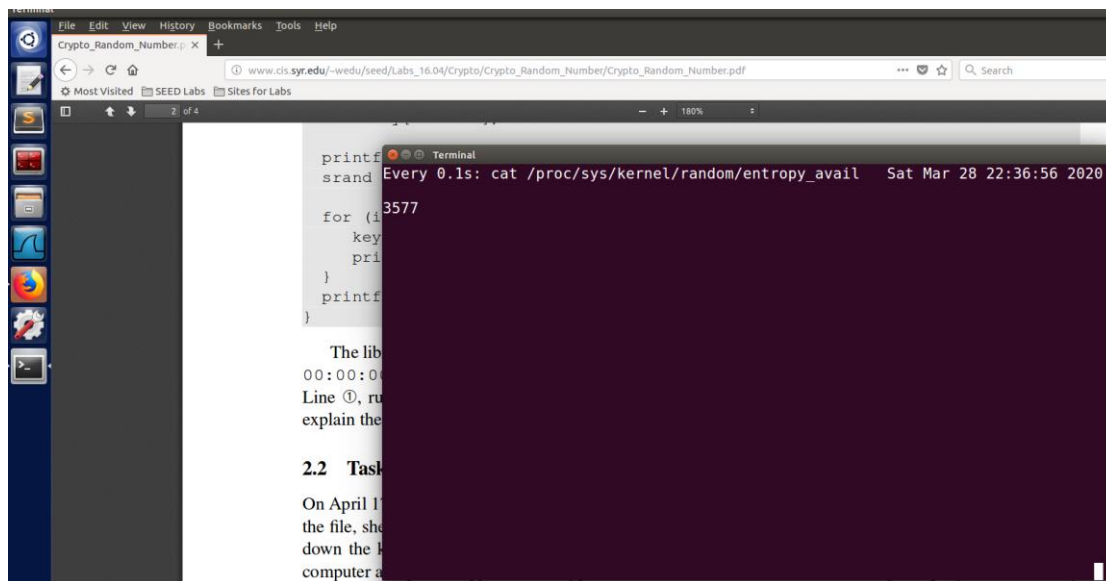
```
[03/28/20]seed@VM:~/Desktop$ date -d @1524017695
Tue Apr 17 22:14:55 EDT 2018
[03/28/20]seed@VM:~/Desktop$
```

Task3:Measure the Entropy of Kernel

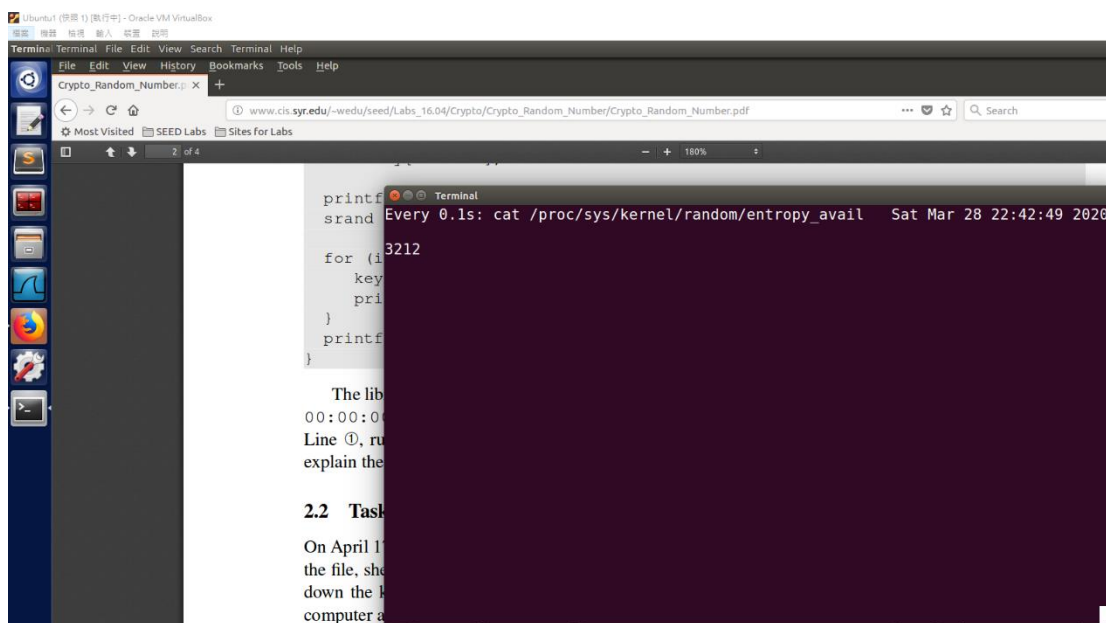
執行 `watch -n .1 cat /proc/sys/kernel/random/entropy_avail` 後，
可以觀測到 `entropy_avail` 每 0.1 秒的變化。



```
Every 0.1s: cat /proc/sys/kernel/random/entropy_avail Sat Mar 28 22:33:49 2020
3349
```

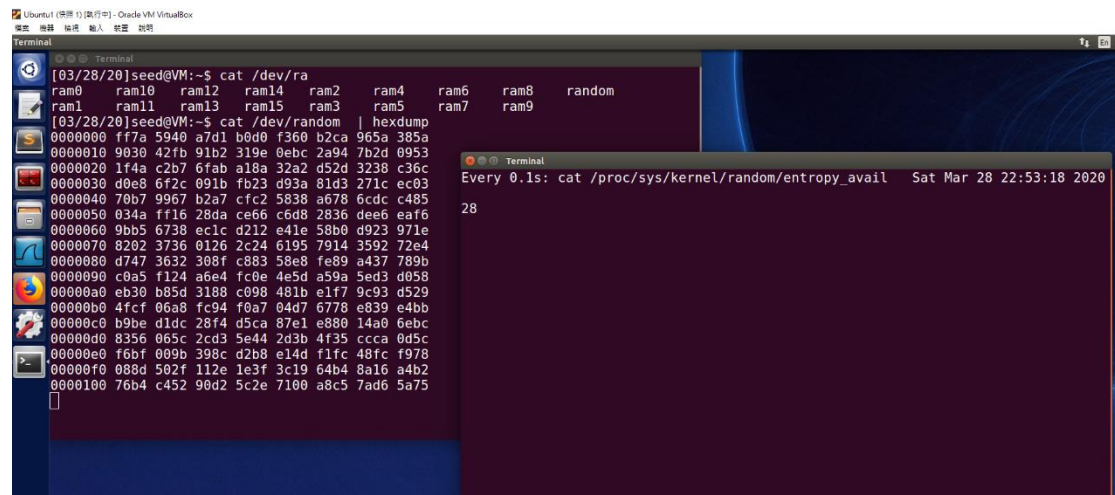


通過觀察可發現:移動滑鼠、開啟檔案、開啟網頁等動作都會增加數值，而移動滑鼠增加的數值較少，當開啟網頁瀏覽影片時增加的幅度較大，但數值也不是無限增長的，一段時間沒用後，數值也會自動往下掉。



Task4: Get Pseudo Random Numbers from /dev/random

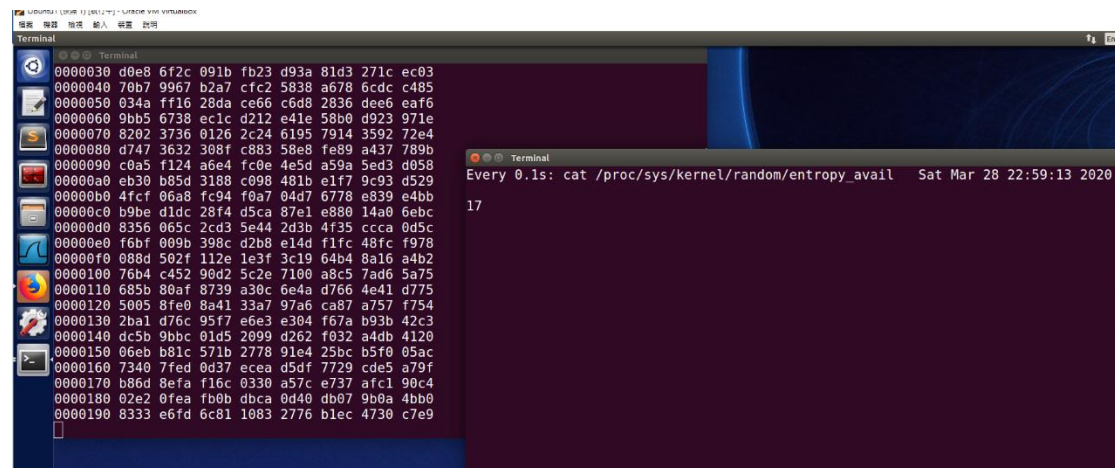
通過執行 `cat /dev/random | hexdump` 後可發現 `entropy_avail` 瞬間從 3000 多下降到 20、30，並且 `/dev/random` 會獲得一串 16 進位的數值。



```
[03/28/20]seed@VM:~$ cat /dev/random | hexdump
00000000 f77a 5940 a7d1 b0d0 f360 b2ca 965a 385a
00000010 9030 42fb 91b2 319e 0ebc 2a94 7b2d 0953
00000020 1f4a c2b7 6fab a18a 32a2 d52d 3238 c36c
00000030 d0e8 6f2c 091b fb23 d93a 81d3 271c ec03
00000040 70b7 9967 b2a7 cfc2 5838 a678 6cdc c485
00000050 034a ff16 28da ce66 c6d8 2836 dee6 eaf6
00000060 9bb5 6738 ec1c d212 e41e 58b0 d923 971e
00000070 8202 3736 0126 2c24 6195 7914 3592 72e4
00000080 d747 3632 308f c883 58e8 fe89 a437 789b
00000090 c0a5 f124 a6e4 fc0e 4e5d a59a 5ed3 d058
000000a0 eb30 b85d 3188 c098 481b e1f7 9c93 d529
000000b0 4fcf 06a8 fc94 f0a7 04d7 6778 e839 e4bb
000000c0 b9be d1dc 28f4 d5ca 87e1 e880 14a0 6ebc
000000d0 8356 065c 2cd3 5e44 2d3b 4f35 ccca 0d5c
000000e0 f6bf 009b 398c d2b8 e14d f1fc 48fc f978
000000f0 088d 502f 112e 1e3f 3c19 64b4 8a16 a4b2
00001000 76b4 c452 90d2 5c2e 7100 a8c5 7ad6 5a75
```

```
Every 0.1s: cat /proc/sys/kernel/random/entropy_avail Sat Mar 28 22:53:18 2020
28
```

且此時 `entropy_avail` 增加到 64 後會清空為 0，清空 2~3 次後，`/dev/random` 會再獲得一串 16 進位的新數值。



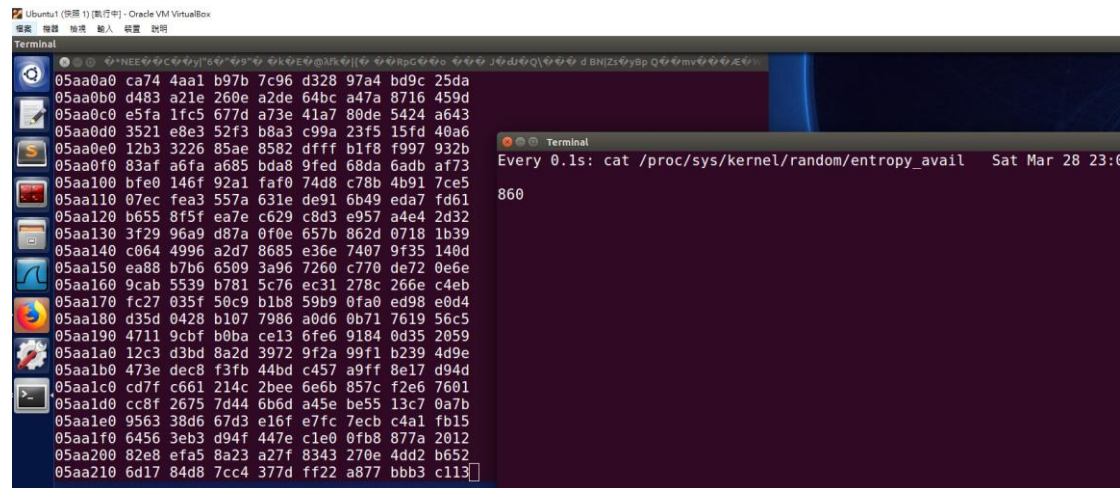
```
00000030 d0e8 6f2c 091b fb23 d93a 81d3 271c ec03
00000040 70b7 9967 b2a7 cfc2 5838 a678 6cdc c485
00000050 034a ff16 28da ce66 c6d8 2836 dee6 eaf6
00000060 9bb5 6738 ec1c d212 e41e 58b0 d923 971e
00000070 8202 3736 0126 2c24 6195 7914 3592 72e4
00000080 d747 3632 308f c883 58e8 fe89 a437 789b
00000090 c0a5 f124 a6e4 fc0e 4e5d a59a 5ed3 d058
000000a0 eb30 b85d 3188 c098 481b e1f7 9c93 d529
000000b0 4fcf 06a8 fc94 f0a7 04d7 6778 e839 e4bb
000000c0 b9be d1dc 28f4 d5ca 87e1 e880 14a0 6ebc
000000d0 8356 065c 2cd3 5e44 2d3b 4f35 ccca 0d5c
000000e0 f6bf 009b 398c d2b8 e14d f1fc 48fc f978
000000f0 088d 502f 112e 1e3f 3c19 64b4 8a16 a4b2
00001000 76b4 c452 90d2 5c2e 7100 a8c5 7ad6 5a75
00001100 685b 80af 8739 a30c 6e4a d766 4e41 d775
00001200 5005 8fe0 8a41 33a7 97a6 ca87 a757 f754
00001300 2ba1 d76c 95f7 e6e3 e304 f67a b93b 42c3
00001400 dc5b 9bbc 01d5 2099 d262 f032 a4db 4120
00001500 06eb b81c 571b 2778 91e4 25bc b5f0 05ac
00001600 7340 7fed 0d37 ecea d5df 7729 cde5 a79f
00001700 b86d 8efa f16c 0330 a57c e737 afc1 90c4
00001800 02e2 0fea fb0b dbca 0d40 db07 9b0a 4bb0
00001900 8333 e6fd 6c81 1083 2776 b1ec 4730 c7e9
```

```
Every 0.1s: cat /proc/sys/kernel/random/entropy_avail Sat Mar 28 22:59:13 2020
17
```

觀察結果得知:執行 `/dev/random` 會限制 `entropy_avail`，並且將數值限於 64，當超過會清空為 0，且 `/dev/random` 收集到一定 `entropy_avail` 數值後會產生新的一串 16 進位數值。

Task5: Get Random Numbers from /dev/urandom

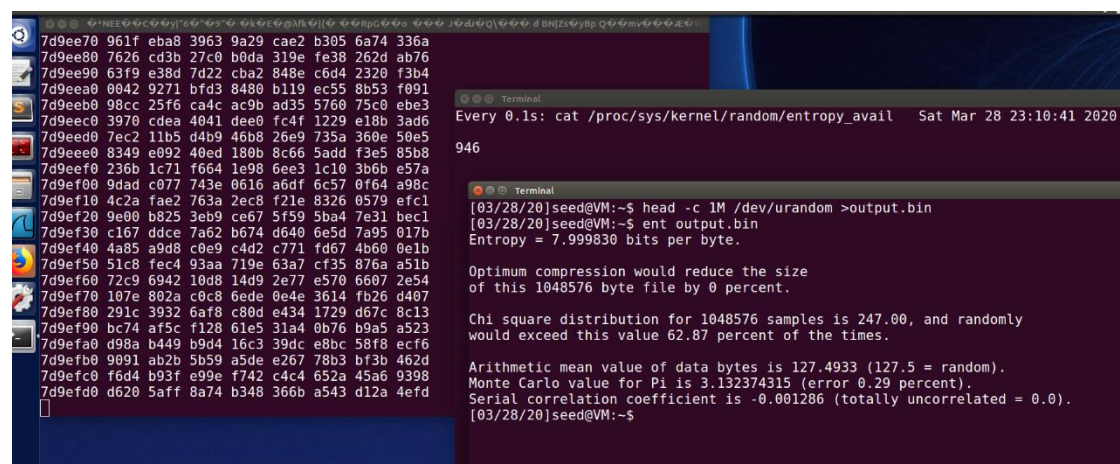
執行 `cat /dev/urandom | hexdump` 後，發現 `/dev/urandom` 會不斷輸出數值，而 `entropy_avail` 會有小幅度的增減，移動滑鼠會增加 `entropy_avail` 的數值，但不會影響到 `/dev/urandom`。
與 `/dev/random` 不同，`/dev/urandom` 不會限制 `entropy_avail` 的數值。



```
05aa0a0 ca74 4aa1 b97b 7c96 d328 97a4 bd9c 25da
05aa0b0 d483 a21e 260e a2de 64bc a47a 8716 459d
05aa0c0 e5fa 1fc5 677d a73e 41a7 80de 5424 a643
05aa0d0 3521 e8e3 52f3 b8a3 c99a 23f5 15fd 40a6
05aa0e0 12b3 3226 85ae 8582 dfff b1f8 f997 932b
05aa0f0 83af a6fa a685 bda8 9fed 68da 6adb af73
05aa100 bfe0 146f 92a1 faf0 74d8 c78b 4b91 7ce5
05aa110 07ec fea3 557a 631e de91 6b49 eda7 fd61
05aa120 b655 8f5f ea7e c629 c8d3 e957 a4e4 2d32
05aa130 3f29 96a9 d87a 0f0e 657b 862d 0718 1b39
05aa140 c064 4996 a2d7 8685 e36e 7407 9f35 140d
05aa150 ea88 b7b6 6509 3a9e 7260 c770 de72 0e6e
05aa160 9cab 5539 b781 5c76 ec31 278c 266e c4eb
05aa170 fc27 035f 50c9 b1b8 59b9 0fa0 ed98 e0d4
05aa180 d35d 0428 b107 7986 a0d6 0b71 7619 56c5
05aa190 4711 9cbf b0ba ce13 6fe6 9184 0d35 2059
05aa1a0 12c3 d3bd 8a2d 3972 9f2a 99f1 b239 4d9e
05aa1b0 473e dec8 f3fb 44bd c457 a9ff 8e17 d94d
05aa1c0 cd7f c661 214c 2bee 6e6b 857c f2e6 7601
05aa1d0 cc8f 2675 7d44 6b6d a45e be55 13c7 0a7b
05aa1e0 9563 38d6 67d3 e16f e7fc 7ecb c4a1 fb15
05aa1f0 6456 3eb3 d94f 447e c1e0 0fb8 877a 2012
05aa200 82e8 efa5 8a23 a27f 8343 270e 4dd2 b652
05aa210 6d17 84d8 7cc4 377d ff22 a877 bbb3 c113
```

```
Every 0.1s: cat /proc/sys/kernel/random/entropy_avail Sat Mar 28 23:06:00
860
```

依照題目要求，用 `/dev/urandom` 產生 1M 的 `output.bin` 檔案，並以 `ent` 這個內建程式分析 `output` 產生的是不是亂數，`ent` 跑了 6 個測試：



```
7d9ee70 961f eba8 3963 9a29 cae2 b305 6a74 336a
7d9ee80 7626 cd3b 27c0 b0da 319e fe38 262d ab76
7d9ee90 63f9 e38d 7d22 cba2 848e c6d4 2320 f3b4
7d9eea0 0042 9271 bfd3 8480 b119 ec55 8b53 f091
7d9eeb0 98cc 25f6 ca4c ac9b ad35 5760 75c0 ebe3
7d9eec0 3970 cdea 4041 de00 fc4f 1229 e18b 3ad6
7d9eed0 7ec2 11b5 d4b9 46b8 26e9 735a 360e 50e5
7d9eee0 8349 e092 40ed 180b 8c66 5add f3e5 85b8
7d9eef0 236b 1c71 f664 1e98 6ee3 1c10 3b6b e57a
7d9ef00 9dad c077 743e 0616 a6df 6c57 0f64 a98c
7d9ef10 4c2a fae2 763a 2ec8 f21e 8326 0579 efc1
7d9ef20 9e00 b825 3eb9 ce67 5f59 5ba4 7e31 bec1
7d9ef30 c167 ddce 7a62 b674 d640 6e5d 7a95 017b
7d9ef40 4a85 a9d8 c0e9 c4d2 c771 fd67 4b60 0e1b
7d9ef50 51c8 fec4 93aa 719e 63a7 cf35 876a a51b
7d9ef60 72c9 6942 10d8 14d9 2e77 e570 6607 2e54
7d9ef70 107e 802a c0c8 6ede 0e4e 3614 fb26 d407
7d9ef80 291c 3932 6af8 c80d e434 1729 d67c 8c13
7d9ef90 bc74 af5c f128 61e5 31a4 0b76 b9a5 a523
7d9efa0 d98a b449 b9d4 16c3 39dc e8bc 58f8 ecf6
7d9efb0 9091 ab2b 5b59 a5de e267 78b3 bf3b 462d
7d9efc0 fd64 b93f e99e f742 c4c4 652a 45a6 9398
7d9efd0 d620 5aff 8a74 b348 366b a543 d12a 4efd
```

```
Every 0.1s: cat /proc/sys/kernel/random/entropy_avail Sat Mar 28 23:10:41 2020
946

[03/28/20]seed@VM:~$ head -c 1M /dev/urandom >output.bin
[03/28/20]seed@VM:~$ ent output.bin
Entropy = 7.999830 bits per byte.

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

Chi square distribution for 1048576 samples is 247.00, and randomly
would exceed this value 62.87 percent of the times.

Arithmetic mean value of data bytes is 127.4933 (127.5 = random).
Monte Carlo value for Pi is 3.132374315 (error 0.29 percent).
Serial correlation coefficient is -0.001286 (totally uncorrelated = 0.0).
[03/28/20]seed@VM:~$
```

1. Entropy: 資訊密度測試，與 8 bits per byte 越相近則越 random，我的結果為 7.999830 bits per byte
2. Optimum compression: 資料重複程度，越接近 0% 越好，我的結果為 0%
3. Chi square distribution: 卡方分布，介於 10%~90% 間比較好，我的結果為 62.87%
4. Arithmetic mean: 越接近 127.5 越好，我的結果為 127.4933
5. Monte Carlo value for Pi: 越接近 3.14159265 越好，我的結果為 3.132374315
6. Serial correlation coefficient: 每個 byte 與前一個 byte 的相依程度，越接近 0 越好，我的結果為 -0.001286。

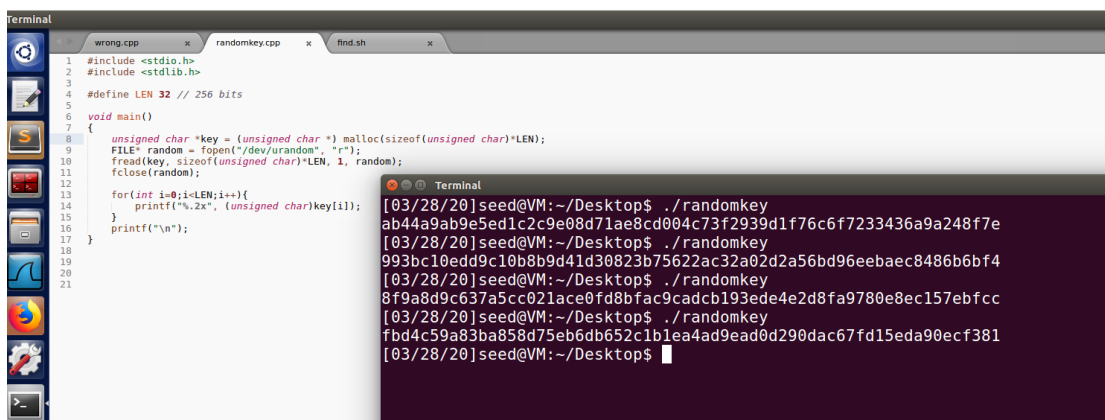
由此結果來看，用/dev/urandom 產生的亂數品質是良好的。

最後，題目要求我們將範例程式修改，改成產出 256-bit 的金鑰

結合 wrong 與題目給的範例，寫成 randomkey.cpp。

首先將 define LEN 改成 32(256bits)

並且讀取/dev/urandom 中 sizeof(unsigned char)*LEN 長度的值作為 key，最後印出 key 來，執行四次後可看到每次的 random key 都不同。



```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define LEN 32 // 256 bits
5
6 void main()
7 {
8     unsigned char *key = (unsigned char *) malloc(sizeof(unsigned char)*LEN);
9     FILE* random = fopen("/dev/urandom", "r");
10    fread(key, sizeof(unsigned char)*LEN, 1, random);
11    fclose(random);
12
13    for(int i=0; i<LEN; i++){
14        printf("%.2x", (unsigned char)key[i]);
15    }
16    printf("\n");
17 }
18
19
20
21
```

```
[03/28/20]seed@VM:~/Desktop$ ./randomkey
ab44a9ab9e5ed1c2c9e08d71ae8cd004c73f2939d1f76c6f7233436a9a248f7e
[03/28/20]seed@VM:~/Desktop$ ./randomkey
993bc10edd9c10b8b9d41d30823b75622ac32a02d2a56bd96eebaec8486b6bf4
[03/28/20]seed@VM:~/Desktop$ ./randomkey
8f9a8d9c637a5cc021ace0fd8bfac9cadcb193ede4e2d8fa9780e8ec157ebfcc
[03/28/20]seed@VM:~/Desktop$ ./randomkey
fbd4c59a83ba858d75eb6db652c1b1ea4ad9ead0d290dac67fd15eda90ecf381
[03/28/20]seed@VM:~/Desktop$
```

-----結束-----