

Information Security HW01

40747019s 莊博傑

P1

(1)

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

(2)

當 $n = 2$ 時

$$\begin{aligned} H(X_1, X_2) &= H(X_1) + H(X_2|X_1) \\ &= H(X_1) + H(X_2) \\ &= \sum_{i=1}^n H(X_i) \end{aligned}$$

成立

假設當 $n = k$ 時題敘成立，則當 $n = k + 1$ 時

$$\begin{aligned} H(X_1, X_2, \dots, X_{k+1}) &= H(X_1, X_2, \dots, X_k) + H(X_{k+1} | X_1, X_2, \dots, X_k) \\ &= \sum_{i=1}^k H(X_i) + H(X_{k+1}) \\ &= \sum_{i=1}^{k+1} H(X_i) \end{aligned}$$

依然成立，故題敘為真

P2

1. Yes

2. No

密文的最後一個 bit 會是 $LSB(m)$ ，可以藉此分辨被加密的 m ，所以此加密方式並不滿足 semantic security

3. Yes

4. Yes

5. No

他直接把 key 接在後面，可以用來解密

P3

首先，嘗試分析密文內的各字元的頻率，先將最高幾個填上去諸如 e, a, t 這種高頻率的字母，此時應該會有部份單字的缺字較少了，將這些單字拿去 dict.org (<http://dict.org>) 做查詢 (此網站支援 *regex* 搜尋)，將查到的單字逐漸填進表內。source code 及解出的文章請見 p3 這個資料夾

P4

一開始我是使用 p4/crack.py 的方式，詳細描述如下：

首先建一個表，以 xor 得出來的值作為 key，可能的明文字元集合作為 value

然後將 challenge ciphertext 拿去跟其餘明文做 xor 得到一些 byte 的序列，這時再對每個位置去查表得到可能的字元集，取這 10 個集合的交集，作為這位置可能的候選

然而在這時我發現有些位置的結果會是空集合，雖然可以復原出部份明文 (p4/part-plaintext)，但是似乎難以直接看出完整的訊息

所以我轉而向彭建霖同學求助，得到另一種解法(p4/crack-v2.py)，步驟如下：

對於任兩組密文，取得他們 xor 過後的序列，然後取出大寫字母的部份，稱它為 M ，此結果應為一個小寫字母與空格 xor 的結果，所以將 M 轉成小寫，再與那個位置的密文 xor 得到可能的 key 並且將它存起來

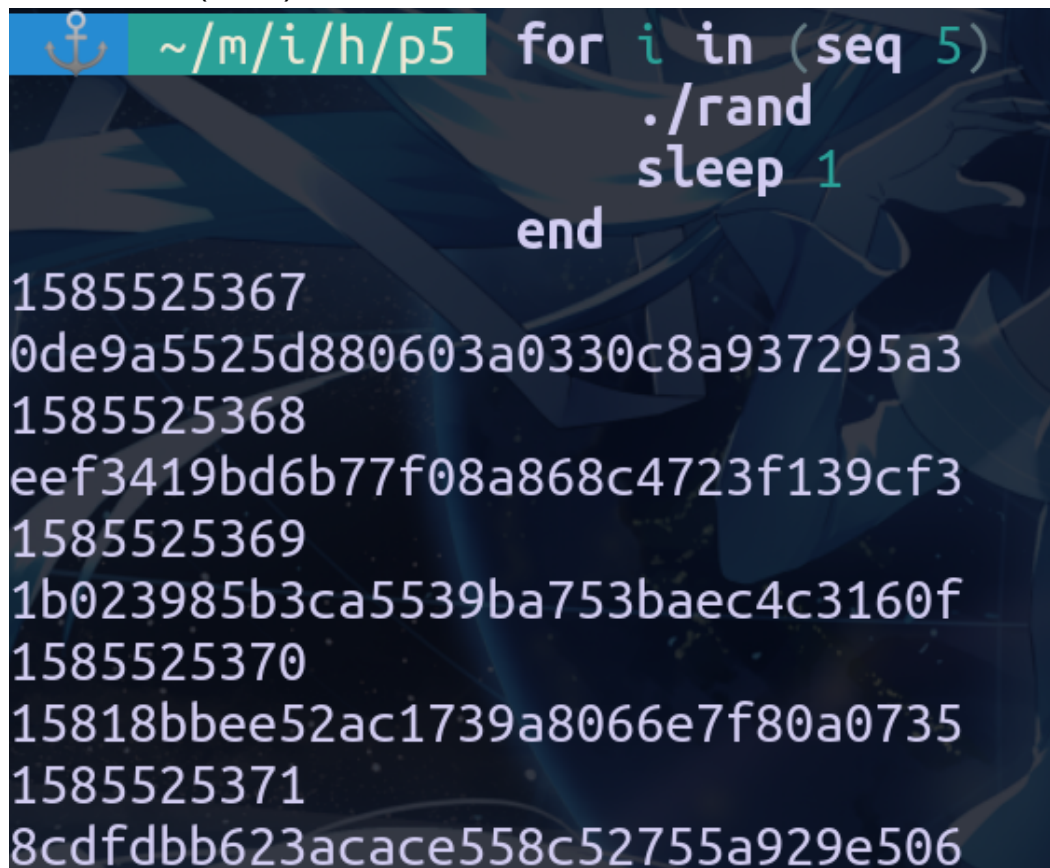
重複上述步驟後，就可以得到一串二維的 list，每個元素代表的是這位置可能的 key，所以取每個元素裡次數最多的作為這位置的 key，將它跟 challenge ciphertext 做 xor 後得到部份明文(p4/plaintext 的第一行)

對於其中有缺漏的部份就靠上下文與上網查字典補齊

P5

Task1

1. 當使用 `time(NULL)` 作為種子時



```
~/m/i/h/p5 for i in (seq 5)
./rand
sleep 1
end

1585525367
0de9a5525d880603a0330c8a937295a3
1585525368
eef3419bd6b77f08a868c4723f139cf3
1585525369
1b023985b3ca5539ba753baec4c3160f
1585525370
15818bb5e52ac1739a8066e7f80a0735
1585525371
8cdfdbb623acace558c52755a929e506
```

2. 沒有呼叫 srand 時

```
~/m/i/h/p5 for i in (seq 5)
              ./rand
              sleep 1
            end

1585525442
67c6697351ff4aec29cdbaabf2fbe346
1585525443
67c6697351ff4aec29cdbaabf2fbe346
1585525444
67c6697351ff4aec29cdbaabf2fbe346
1585525445
67c6697351ff4aec29cdbaabf2fbe346
1585525446
67c6697351ff4aec29cdbaabf2fbe346
```

可以發現，如果沒有呼叫 `srand` 的話，每次隨機出來的字串都會是一樣的。在原本的 code 當中，使用了 `time` 作為種子，以確保每次呼叫產生出來的 `key` 都會不一樣。

Task2

解出得到 `key` 為 `95fa2030e73ed3f8da761b4eb805dfd7`

產生時間應為 `2018/4/17 22:14:55 (UTC-4)`

```
~/m/i/h/p5 python3 crack.py
timestamp: 1524017695
key: 95fa2030e73ed3f8da761b4eb805dfd7
```

實做方式是先使用 `c` 寫出產生 `key` 的 function，然後再由 `python` 這邊呼叫，具體來說就是算出可能的時間區間後嘗試每個 `key` 的可能性。

Task3

整體來說在數字小的時候增長較快，到接近 4000 時則幾乎不會有大的變動。根據題本總共實驗以下五種動作：

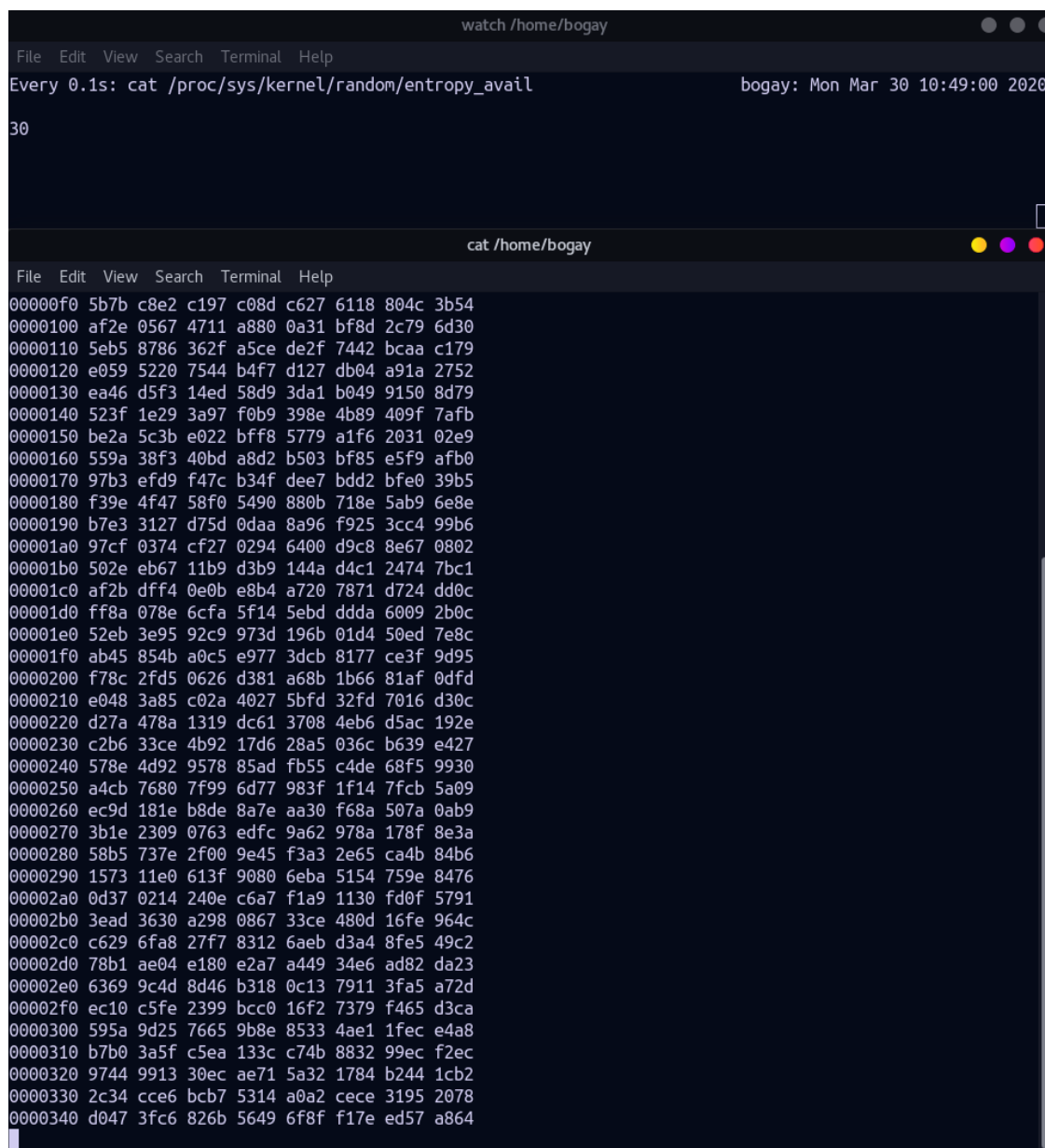
1. 移動滑鼠：看起來似乎是增長最快的方式，雖然說在數字大的時候感覺不

大，但當 entropy 低落時，動動滑鼠就能使它快速上升

2. 點擊滑鼠：看起來影響不算很大，推測是因為點擊的動作較零散，無法提供足夠多的亂度
3. 打字：感覺與點擊滑鼠相差不大
4. 讀檔：在讀非常大的檔案的時候整體 entropy 會上升挺多，然而上升的速率似乎不會比移動滑鼠快很多
5. 瀏覽網頁：感覺與點擊滑鼠的差異也不大，雖然說在瀏覽網頁時也能看到 entropy 增長的較迅速，但我想應該是滑鼠移動 + 點擊的影響

Task4

觀察 entropy



```
watch /home/bogay
File Edit View Search Terminal Help
Every 0.1s: cat /proc/sys/kernel/random/entropy_avail bogay: Mon Mar 30 10:49:00 2020
30

cat /home/bogay
File Edit View Search Terminal Help
00000f0 5b7b c8e2 c197 c08d c627 6118 804c 3b54
0000100 af2e 0567 4711 a880 0a31 bf8d 2c79 6d30
0000110 5eb5 8786 362f a5ce de2f 7442 bcaa c179
0000120 e059 5220 7544 b4f7 d127 db04 a91a 2752
0000130 ea46 d5f3 14ed 58d9 3da1 b049 9150 8d79
0000140 523f 1e29 3a97 f0b9 398e 4b89 409f 7afb
0000150 be2a 5c3b e022 bff8 5779 a1f6 2031 02e9
0000160 559a 38f3 40bd a8d2 b503 bf85 e5f9 afb0
0000170 97b3 efd9 f47c b34f dee7 bdd2 bfe0 39b5
0000180 f39e 4f47 58f0 5490 880b 718e 5ab9 6e8e
0000190 b7e3 3127 d75d 0daa 8a96 f925 3cc4 99b6
00001a0 97cf 0374 cf27 0294 6400 d9c8 8e67 0802
00001b0 502e eb67 11b9 d3b9 144a d4c1 2474 7bc1
00001c0 af2b dff4 0e0b e8b4 a720 7871 d724 dd0c
00001d0 ff8a 078e 6cfa 5f14 5ebd ddda 6009 2b0c
00001e0 52eb 3e95 92c9 973d 196b 01d4 50ed 7e8c
00001f0 ab45 854b a0c5 e977 3dc6 8177 ce3f 9d95
0000200 f78c 2fd5 0626 d381 a68b 1b66 81af 0dfd
0000210 e048 3a85 c02a 4027 5bfd 32fd 7016 d30c
0000220 d27a 478a 1319 dc61 3708 4eb6 d5ac 192e
0000230 c2b6 33ce 4b92 17d6 28a5 036c b639 e427
0000240 578e 4d92 9578 85ad fb55 c4de 68f5 9930
0000250 a4cb 7680 7f99 6d77 983f 1f14 7fcb 5a09
0000260 ec9d 181e b8de 8a7e aa30 f68a 507a 0ab9
0000270 3b1e 2309 0763 edfc 9a62 978a 178f 8e3a
0000280 58b5 737e 2f00 9e45 f3a3 2e65 ca4b 84b6
0000290 1573 11e0 613f 9080 6eba 5154 759e 8476
00002a0 0d37 0214 240e c6a7 f1a9 1130 fd0f 5791
00002b0 3ead 3630 a298 0867 33ce 480d 16fe 964c
00002c0 c629 6fa8 27f7 8312 6aeb d3a4 8fe5 49c2
00002d0 78b1 ae04 e180 e2a7 a449 34e6 ad82 da23
00002e0 6369 9c4d 8d46 b318 0c13 7911 3fa5 a72d
00002f0 ec10 c5fe 2399 bcc0 16f2 7379 f465 d3ca
0000300 595a 9d25 7665 9b8e 8533 4ae1 1fec e4a8
0000310 b7b0 3a5f c5ea 133c c74b 8832 99ec f2ec
0000320 9744 9913 30ec ae71 5a32 1784 b244 1cb2
0000330 2c34 cce6 bcb7 5314 a0a2 cece 3195 2078
0000340 d047 3fc6 826b 5649 6f8f f17e ed57 a864
```

在實驗過程中，可以發現當 entropy 降到大約 60 以下，的時候 hexdump 的輸出就會卡住，然後滑鼠隨機移動時 entropy 會快速上升到約莫 60 左右時再降至接近 0 的值，並且看到新的一行隨機的 16 個 bytes 被產生出來

如何執行 DOS 攻擊

假設服務需要使用 `/dev/random` 產生 session key 的話，只要大量發出請求，entropy 自然會降至無法生出新的 key，因此其他人的請求就會被阻塞

Task5

Ent 測試

運行以下指令

```
head -c 1M /dev/urandom > output.bin
ent output.bin
```

得到輸出

```
Entropy = 7.999834 bits per byte.
```

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

```
Chi square distribution for 1048576 samples is 241.16, and random
would exceed this value 72.40 percent of the times.
```

```
Arithmetic mean value of data bytes is 127.6711 (127.5 = random)
Monte Carlo value for Pi is 3.141392294 (error 0.01 percent).
Serial correlation coefficient is -0.000620 (totally uncorrelated)
```

嘗試與 C 的 `rand` 來做比較，先產生 1M 的隨機資料，使用 `ent` 分析後得到以下輸出

Entropy = 7.999832 bits per byte.

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

Chi square distribution for 1048576 samples is 243.93, and random
would exceed this value 68.00 percent of the times.

Arithmetic mean value of data bytes is 127.5626 (127.5 = random)
Monte Carlo value for Pi is 3.138439707 (error 0.10 percent).
Serial correlation coefficient is 0.000544 (totally uncorrelated)

將兩份輸出的比較做成表格

indicator	/dev/urandom	C rand
Entropy (bits/byte)	7.999834	7.999832
Chi square	241.16	243.93
Mean	127.6711	127.5626
Monte Carlo error	0.01%	0.1%
Serial correlation	-0.000620	0.000544

以這次比較來看，使用 C 的 rand 似乎表現上是稍微差了一些，以蒙地卡羅法求 π 值的誤差大約 10 倍之多，但後來自己再手動執行幾次，有時候 C 的 rand 表現也是不錯的

使用 /dev/urandom 產生 key

source code

```
#include <stdio.h>
#include <stdlib.h>
#define KEYSIZE 32 // 256 bits

int main()
{
    unsigned char key[KEYSIZE];
    // read random bytes
    FILE *random = fopen("/dev/urandom", "r");
    fread(key, sizeof(unsigned char) * KEYSIZE, 1, random);
    fclose(random);
    // print key
    for (int i = 0; i < KEYSIZE; i++)
        printf("%.2x", key[i]);
    puts("");
    return 0;
}
```

結果



```
~/m/i/h/p5 ./rand
63e0371e75937f3717683e830987dfefa8bbcd3c5b2516eeae862154b4858241
```