HUANG Kunlun 57878689
kl.h@my.cityu.edu.hk

Contents

1 Introduction

This is the solvement for the CS5293 Assignment I. Each Task have a short statment for the result as well as the procedures or the screenshoot or code listing attached.

2 Secret-Key Encryption

2.1 Task 1: Frequency Analysis Against Monoalphabetic Substitution Cipher

For this question, I mainly conducted decryption tests through the statistical frequency of the main letters. After multiple rounds of letter deduction, as shown in Table 1.

Table 1: Frequency Analysis

character	counts	frequency(%)	1st Round Test	2st Round Test	3rd Round Test	Final Result
n	488	12.4	E	E	E	E
y	373	9.5	T	T	T	T
v	348	8.9	A	A	A	A
x	291	7.4	O	O	O	O
u	280	7.1	I	N	N	N
q	276	7	N		S	S
m	264	6.7	S		I	I
h	235	6	H	R	R	R
t	183	4.7	R	H	H	H
i	166	4.2	D		L	L
p	156	4	I	D	D	D
a	116	3	U		C	C
c	104	2.6	C		M	M
z	95	2.4				U
l	90	2.3			W	W
b	83	2.1				F
g	83	2.1			B	B
r	82	2.1				G
e	76	1.9				P
d	59	1.5			Y	Y
f	49	1.2			V	V
s	19	0.5				K
j	5	0.1				Q
k	5	0.1				X
o	4	0.1				J
w	1	0				Z

```
$ tr 'nyvxuqmhtipaczlbgredfsjkow' 'ETAONSIRHLDCMUWFBGPYVKQXJZ' < ciphertext.txt > plaintext.
  txt
```

Listing 1: Decrypt Code

After execute the following command as listing 1, And I finally decrypted the ciphertext to plaintext, which mainly describes the recent controversies and discussions related to the Oscar Awards.

2.2 Task 2: Encryption using Different Ciphers and Modes

```
$ openssl enc -aes-128-cbc -e -in plaintext.txt -out cipher_aes128cbc.bin -K
  00112233445566778889aabbccddeeff -iv 0102030405060708 # aes-128-cbc encryption
$ openssl enc -aes-256-cbc -e -in plaintext.txt -out cipher_aes256cbc.bin -K
  00112233445566778889aabbccddeeff -iv 0102030405060708 # aes-256-cbc encryption
$ openssl enc -bf-cbc -e -in plaintext.txt -out cipher_blowfishcbc.bin -K
  00112233445566778889aabbccddeeff -iv 0102030405060708 # blowfish encryption
$ openssl enc -aes-128-cfb -e -in plaintext.txt -out cipher_aes128cfb.bin -K
  00112233445566778889aabbccddeeff -iv 0102030405060708 # aes-128-cfb encryption
```

Listing 2: TASK 2.2 Command Line

As shown in listing 2, I try to use 4 different encryption ciphers by openssl to encrypt the plaintext generated in Sec 2.1.

2.3 Task 3: Encryption Mode - ECB vs. CBC

I conduct the following command 3 to encrypt the original images to ECB and CBC respectively. It's not hard to seen that the ECB mode encrypted image have a characteristics of the original image, while the CBC doesn't, as shown in Figure 2.

```
$ openssl enc -aes-128-ecb -e -in pic_original.bmp -out encrypted_ecb.bmp -K
    00112233445566778889aabbccddeeff # ECB Mode
$ openssl enc -aes-128-cbc -e -in pic_original.bmp -out encrypted_cbc.bmp -K
    00112233445566778889aabbccddeeff  -iv 0102030405060708 # CBC Mode
```

Listing 3: TASK 2.3 Command Lines

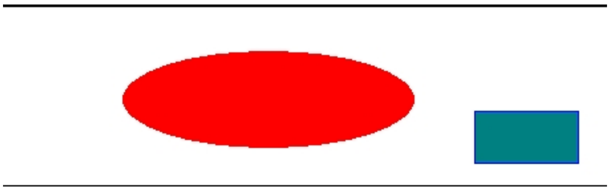
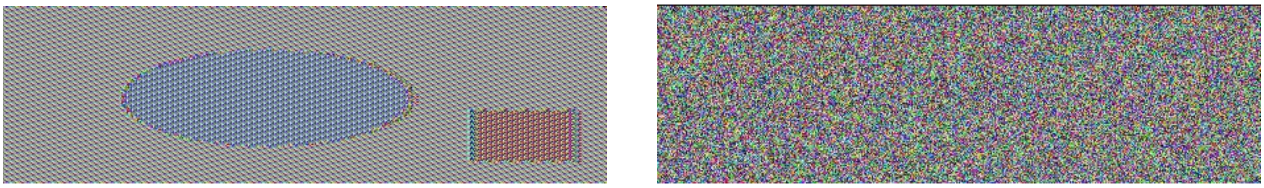


Figure 1: Original image



(a) ECB Mode

(b) CBC Mode

Figure 2: Encrypted images

2.4 Task 4: Padding

1. • **ECB, CBC:** Ciphertext is divided into blocks, so padding is needed.
   • **CFB, OFB:** Operate on smaller units (e.g., 8 bits), so padding isn't strictly required. They can still apply padding for compatibility or security reasons.
2. The following listing shows the results.

```
# generate small files
$ echo -n "12345" > f1.txt
$ echo -n "1234567890" > f2.txt
$ echo -n "1234567890123456" > f3.txt

# encrypt the small files by aes-128-cbc
$ openssl enc -aes-128-cbc -e -in f1.txt -out f1_encrypted_cbc.bin -K
    00112233445566778889aabbccddeeff -iv 0102030405060708
$ openssl enc -aes-128-cbc -e -in f2.txt -out f2_encrypted_cbc.bin -K
    00112233445566778889aabbccddeeff -iv 0102030405060708
$ openssl enc -aes-128-cbc -e -in f3.txt -out f3_encrypted_cbc.bin -K
    00112233445566778889aabbccddeeff -iv 0102030405060708

# dencrypt the small files by aes-128-cbc with -nopad
$ openssl enc -aes-128-cbc -d -nopad -in f1_encrypted_cbc.bin -out f1_decrypted_cbc.bin
     -K 00112233445566778889aabbccddeeff -iv 0102030405060708
$ openssl enc -aes-128-cbc -d -nopad -in f2_encrypted_cbc.bin -out f2_decrypted_cbc.bin
     -K 00112233445566778889aabbccddeeff -iv 0102030405060708
$ openssl enc -aes-128-cbc -d -nopad -in f3_encrypted_cbc.bin -out f3_decrypted_cbc.bin
     -K 00112233445566778889aabbccddeeff -iv 0102030405060708

# observe the results
$ hexdump -C f3_decrypted_cbc.bin
00000000  31 32 33 34 35 36 37 38  39 30 31 32 33 34 35 36  |1234567890123456|
00000010  10 10 10 10 10 10 10 10  10 10 10 10 10 10 10 10  |................|
00000020
$ hexdump -C f3.txt
00000000  31 32 33 34 35 36 37 38  39 30 31 32 33 34 35 36  |1234567890123456|
00000010
```

```
$ hexdump -C f1.txt
00000000  31 32 33 34 35                                   |12345|
00000005
$ hexdump -C f1_decrypted_cbc.bin
00000000  31 32 33 34 35 0b 0b 0b  0b 0b 0b 0b 0b 0b 0b 0b  |12345...........|
```

Listing 4: TASK 2.4 Command Lines

2.5 Task 5: Error Propagation - Corrupted Cipher Text

Expected Behavior:

- In ECB mode, each block is encrypted independently of other blocks. A single bit error in one block should only affect that block during decryption, not propagating errors to subsequent blocks.
- CBC mode XORs each plaintext block with the ciphertext block from the previous iteration. A single bit error in one block will result in unpredictable errors in both the corresponding block and the following blocks during decryption.
- CFB mode operates on the output of the encryption operation, and errors should not propagate to subsequent blocks during decryption.
- Similar to CFB, OFB mode operates on the output of the encryption operation, and errors should not propagate to subsequent blocks during decryption.

Conduct Experiments:

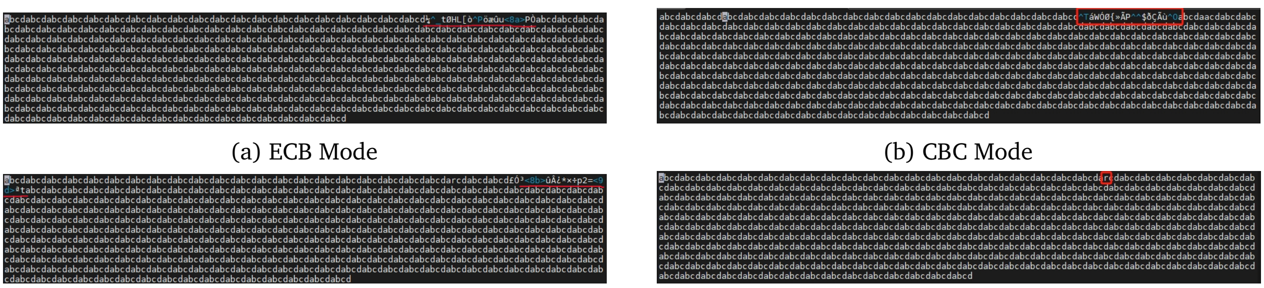
```
# generate a 1.2k bytes abcd repeating textual file
$ echo -n $(printf 'abcd%.0s' {1..300}) > plaintext.txt

# encrypt the file by different mode
$ openssl enc -aes-128-ecb -e -in plaintext.txt -out encrypted_aes_ecb.bin -K
    00112233445566778889aabbccddeeff
$ openssl enc -aes-128-cbc -e -in plaintext.txt -out encrypted_aes_cbc.bin -K
    00112233445566778889aabbccddeeff -iv 0102030405060708
$ openssl enc -aes-128-cfb -e -in plaintext.txt -out encrypted_aes_cfb.bin -K
    00112233445566778889aabbccddeeff -iv 0102030405060708
$ openssl enc -aes-128-ofb -e -in plaintext.txt -out encrypted_aes_ofb.bin -K
    00112233445566778889aabbccddeeff -iv 0102030405060708

# manually edit the bit of the 55th byte
$ bless encrypted_aes_ecb.bin # manually edit the bit of the 55th byte
$ bless encrypted_aes_cbc.bin # manually edit the bit of the 55th byte
$ bless encrypted_aes_cfb.bin # manually edit the bit of the 55th byte
$ bless encrypted_aes_ofb.bin # manually edit the bit of the 55th byte

# decrypt the file by different mode
$ openssl enc -aes-128-ecb -d -in encrypted_aes_ecb.bin  -out decrypted_ecb.txt -K
    00112233445566778889aabbccddeeff
$ openssl enc -aes-128-cbc -d -in encrypted_aes_cbc.bin -out decrypted_cbc.txt -K
    00112233445566778889aabbccddeeff -iv 0102030405060708
$ openssl enc -aes-128-cfb -d -in encrypted_aes_cfb.bin  -out decrypted_cfb.txt -K
    00112233445566778889aabbccddeeff -iv 0102030405060708
$ openssl enc -aes-128-ofb -d -in encrypted_aes_ofb.bin  -out decrypted_ofb.txt -K
    00112233445566778889aabbccddeeff -iv 0102030405060708
```

Listing 5: TASK 2.5 Command Lines



(a) ECB Mode

(b) CBC Mode

(c) CFB Mode

(d) OFB Mode

Figure 3: Error Propagation Observation

Justification:

- In ECB mode, As expected, errors in ECB mode do not propagate to subsequent blocks, and the corrupted block is isolated.
- Errors in CBC mode propagate to subsequent blocks due to the XOR operation with the previous ciphertext block.
- CFB mode operates on the output of the encryption operation, and errors should not propagate to subsequent blocks during decryption.
- CFB and OFB: Errors in both CFB and OFB modes do not propagate to subsequent blocks, as they operate on the output of the encryption operation.

2.6 Task 6: Initial Vector (IV)

- **Task 6.1**

```
# generate a plain text
$ echo -n "12345678901234561231 23" > plaintext.txt
# Encrypting with Two Different IVs:
$ openssl enc -aes-128-cbc -e -in plaintext.txt -out encrypted_iv1.bin -K
    00112233445566778889aabbccddeeff -iv 0102030405060708
$ openssl enc -aes-128-cbc -e -in plaintext.txt -out encrypted_iv2.bin -K
    00112233445566778889aabbccddeeff -iv 1122334455667788
# Encrypting with the Same IV:
$ openssl enc -aes-128-cbc -e -in plaintext.txt -out encrypted_same_iv.bin -K
    00112233445566778889aabbccddeeff -iv 0102030405060708
# Observations:
$ hexdump -C encrypted_iv1.bin
00000000  ec fb 05 81 98 77 71 c9  46 02 98 ba 32 85 a2 c2  |.....wq.F...2...|
00000010  3a 5a 5c 7f ac 22 10 62  b6 a8 6b 86 52 da 3a 24  |:Z\..".b..k.R.:$|
00000020
$ hexdump -C encrypted_iv2.bin
00000000  bb cf 5b 59 ae 45 65 1d  00 d5 9c ba ec 4e 13 9e  |..[Y.Ee......N..|
00000010  ef 6d 53 d9 63 8d 0b aa  48 ae 20 28 b9 2c 45 f8  |.mS.c...H. (.,E.|
00000020
$ hexdump -C encrypted_same_iv.bin
00000000  ec fb 05 81 98 77 71 c9  46 02 98 ba 32 85 a2 c2  |.....wq.F...2...|
00000010  3a 5a 5c 7f ac 22 10 62  b6 a8 6b 86 52 da 3a 24  |:Z\..".b..k.R.:$|
```

Listing 6: TASK 2.6.1 Command Lines

In the first case, where different IVs are used: encrypted_iv1.bin and encrypted_iv2.bin will be different. In the second case, where the same IV is used: encrypted_same_iv.bin will be the same as encrypted_iv1.bin because the same IV was used.

When different IVs are used, the resulting ciphertexts are different, even for the same plaintext and key. This is crucial for security because it ensures that patterns in the plaintext are not easily observable in the ciphertext.

If the same IV is reused, an observer might notice patterns or repetitions in the ciphertext, which can lead to vulnerabilities and compromise the security of the encryption. Using unique IVs helps in preventing such patterns and adds an additional layer of security.

- **Task 6.2**

(1) Yes, they can potentially decrypt all subsequent messages encrypted with the same key and IV.

(2) Yes, I can likely decrypt P2 using a known-plaintext attack.

$$\begin{aligned} output\_stream &= C1 \oplus P1 \\ P2 &= C2 \oplus output\_stream \end{aligned} \tag{1}$$

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[]) {
    // Check arguments
    if (argc != 4) {
        printf("Usage: %s <plaintext> <ciphertext1> <ciphertext2>\n", argv[0]);
        return 1;
    }
    // Convert arguments to byte arrays
    size_t len1 = strlen(argv[1]);
```

```c
    unsigned char plaintext[len1 + 1];
    memcpy(plaintext, argv[1], len1);
    plaintext[len1] = '\0';
    size_t len2 = strlen(argv[2]);
    if (len2 % 2 != 0) {
        printf("Error: ciphertext1 must be in hexadecimal format\n");
        return 1;
    }
    size_t ciphertext1_len = len2 / 2;
    unsigned char ciphertext1[ciphertext1_len];
    for (int i = 0; i < ciphertext1_len; i++) {
        sscanf(argv[2] + 2 * i, "%2hhx", &ciphertext1[i]);
    }
    size_t len3 = strlen(argv[3]);
    if (len3 % 2 != 0) {
        printf("Error: ciphertext2 must be in hexadecimal format\n");
        return 1;
    }
    size_t ciphertext2_len = len3 / 2;
    if (ciphertext1_len != ciphertext2_len) {
        printf("Error: ciphertexts must be of equal length\n");
        return 1;
    }
    unsigned char ciphertext2[ciphertext2_len];
    for (int i = 0; i < ciphertext2_len; i++) {
        sscanf(argv[3] + 2 * i, "%2hhx", &ciphertext2[i]);
    }
    // Perform XOR operation and decode
    unsigned char recovered_text[ciphertext1_len];
    for (int i = 0; i < ciphertext1_len; i++) {
        recovered_text[i] = plaintext[i] ^ ciphertext1[i] ^ ciphertext2[i];
    }
    // Print recovered plaintext
    printf("Recovered plaintext: %s\n", recovered_text);
    return 0;
}
```

Listing 7: known-plaintext-attack in C

After execute the program by `./attack P1 C1 C2`, the result is Recovered plaintext: "Order: Launch a missile!"

(3) Only the first block of P2 can be confidently revealed in CFB mode.

- **Task 6.3** Here's how we can exploit predictable IVs to determine the content of P1:

  Construct P2:

  Eve creates a message P2 identical to P1 except for the first character: If she suspects P1 is "Yes", P2 will be "Xes". If she suspects P1 is "No", P2 will be "Wo".

  Request Encryption:

  Eve sends P2 to Bob for encryption. Bob, unaware of the attack, encrypts P2 using AES-128 in CBC mode with the predictable IV "1234567890123457".

  Analyze Ciphertext C2:

  Bob returns the ciphertext C2 to Eve.

  Eve examines the first block of C2:

  If the first block is "bef65565572ccee2", it strongly suggests P1 was "Yes". If the first block is anything else, it indicates P1 was likely "No".

2.7 Task 7: Programming using the Crypto Library

```python
#!/usr/bin/python3
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad

plaintext = "This is a top secret."
ciphertext = "764aa26b55a4da654df6b19e4bce00f4ed05e09346fb0e762583cb7da2ac93a2"
iv = "aabbccddeeff00998877665544332211"
plaindatabits = bytearray(plaintext, encoding='utf-8')
cipherdatabits = bytearray.fromhex(ciphertext)
ivdatabits = bytearray.fromhex(iv)

```

```
with open('./words.txt') as f:
  keys = f.readlines()

for k in keys:
  k = k.rstrip('\n')
  if len(k) <= 16:
    key = k + '#'*(16-len(k))
    cipher = AES.new(key=bytearray(key,encoding='utf-8'), mode=AES.MODE_CBC, iv=ivdatabits)
    testbits = cipher.encrypt(pad(plaindatabits, 16))
    if cipherdatabits == testbits:
      print("find the key:",key)
      exit(0)

print("cannot find the key!")
```

Listing 8: find key script in Python

The script use the Crypto to call the AES API in Python to encrypt the text, and simple loop to find a matched key in word list from a file.
After execute the script, the matched key is **"Syracuse"**.

3 MD5 Collision Attack

3.1 Task 8: Generating Two Different Files with the Same MD5 Hash

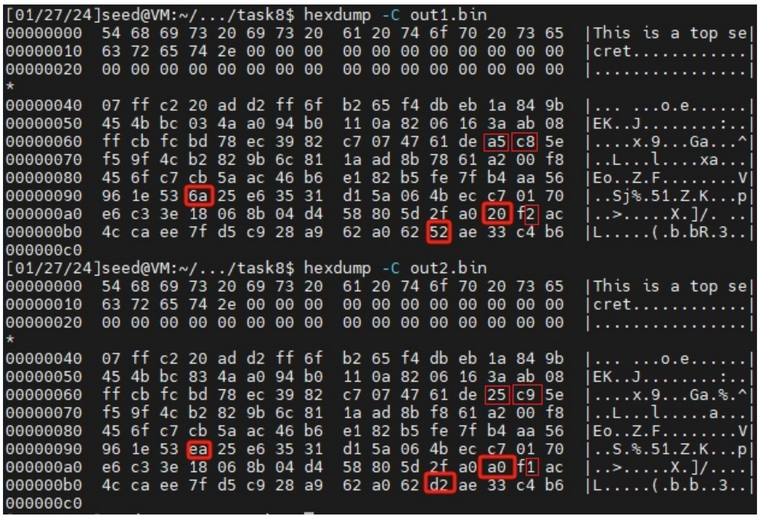

```
[01/27/24]seed@VM:~/.../task8$ hexdump -C out1.bin
00000000  54 68 69 73 20 69 73 20  61 20 74 6f 70 20 73 65  |This is a top se|
00000010  63 72 65 74 2e 00 00 00  00 00 00 00 00 00 00 00  |cret............|
00000020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000040  07 ff c2 20 ad d2 ff 6f  b2 65 f4 db eb 1a 84 9b  |... ...o.e......|
00000050  45 4b bc 03 4a a0 94 b0  11 0a 82 06 16 3a ab 08  |EK..J........:..|
00000060  ff cb fc bd 78 ec 39 82  c7 07 47 61 de a5 c8 5e  |....x.9...Ga...^|
00000070  f5 9f 4c b2 82 9b 6c 81  1a ad 8b 78 61 a2 00 f8  |..L...l....xa...|
00000080  45 6f c7 cb 5a ac 46 b6  e1 82 b5 fe 7f b4 aa 56  |Eo..Z.F........V|
00000090  96 1e 53 6a 25 e6 35 31  d1 5a 06 4b ec c7 01 70  |..Sj%.51.Z.K...p|
000000a0  e6 c3 3e 18 06 8b 04 d4  58 80 5d 2f a0 20 f2 ac  |..>.....X.]/. ..|
000000b0  4c ca ee 7f d5 c9 28 a9  62 a0 62 52 ae 33 c4 b6  |L.....(.b.bR.3..|
000000c0
[01/27/24]seed@VM:~/.../task8$ hexdump -C out2.bin
00000000  54 68 69 73 20 69 73 20  61 20 74 6f 70 20 73 65  |This is a top se|
00000010  63 72 65 74 2e 00 00 00  00 00 00 00 00 00 00 00  |cret............|
00000020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000040  07 ff c2 20 ad d2 ff 6f  b2 65 f4 db eb 1a 84 9b  |... ...o.e......|
00000050  45 4b bc 83 4a a0 94 b0  11 0a 82 06 16 3a ab 08  |EK..J........:..|
00000060  ff cb fc bd 78 ec 39 82  c7 07 47 61 de 25 c9 5e  |....x.9...Ga.%.^|
00000070  f5 9f 4c b2 82 9b 6c 81  1a ad 8b f8 61 a2 00 f8  |..L...l.....a...|
00000080  45 6f c7 cb 5a ac 46 b6  e1 82 b5 fe 7f b4 aa 56  |Eo..Z.F........V|
00000090  96 1e 53 ea 25 e6 35 31  d1 5a 06 4b ec c7 01 70  |..S.%.51.Z.K...p|
000000a0  e6 c3 3e 18 06 8b 04 d4  58 80 5d 2f a0 a0 f1 ac  |..>.....X.]/....|
000000b0  4c ca ee 7f d5 c9 28 a9  62 a0 62 d2 ae 33 c4 b6  |L.....(.b.b..3..|
000000c0
```

Figure 4: MD5 Collision

Question:

1. Length Not Multiple of 64: md5collgen might pad the prefix with additional bytes to reach a multiple of 64, as MD5 operates on 64-byte blocks.

2. Prefix of Exactly 64 Bytes: No padding should be necessary, and the collision blocks will directly follow the prefix in both output files.

3. Extent of Differences: The number and distribution of differing bytes can vary depending on the specific collision generated.

3.2 Task 9: Understanding MD5's Property

```
# out1.bin and out2.bin are  two files  generated in Task8 with the same MD5 hash:
$ md5sum out1.bin
189b839e730f57d79a8b4a98aadb36ea  out1.bin
$ md5sum out2.bin
189b839e730f57d79a8b4a98aadb36ea  out2.bin
# generate a suffix
$ echo -n "Lanch a missile." > suffix.txt
# concate two files with the suffix
$ cat out1.bin suffix.txt > out3.bin
```

```
$ cat out2.bin suffix.txt > out4.bin
# observe the concated files
$ md5sum out3.bin
30c758787b6e481692efd404ccb5c1a4  out3.bin
$ md5sum out4.bin
30c758787b6e481692efd404ccb5c1a4  out4.bin
```

Listing 9: MD5 Property test script

Result:
The two concated files that with the same prefix MD5 and same suffix will result in the same result MD5 HASH value.

3.3 Task 10: Generating Two Executable Files with the Same MD5 Hash

```
# After Compliering the source code to Executable Program
# Cut the Executable Program to two parts
$ head -c 4224 out1 > prefix
$ tail -c +4287 out1 > suffix
# generated the same MD5 part of the program and join them to suffix
$ md5collgen -p prefix -o out1.bin out2.bin
$ cat out1.bin suffix > program1
$ cat out2.bin suffix > program2
# observe the MD5 of two Different program
$ md5sum program1
60824a7b99425740d90c0cfb41a34a1e  program1
$ md5sum program2
60824a7b99425740d90c0cfb41a34a1e  program2
$ chmod +x program1
$ chmod +x program2
$ ./program1
$ ./program2
# observe the execute result
```

Listing 10: MD5 Executable Program

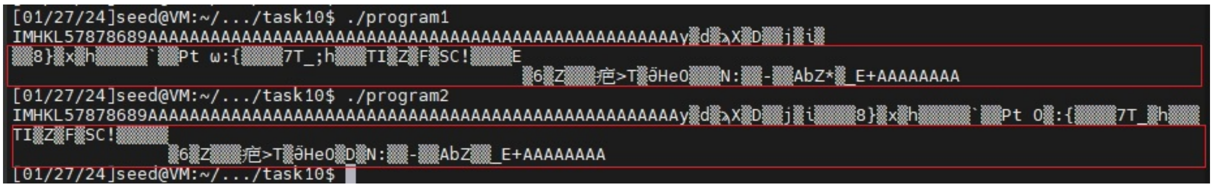


Figure 5: Execute Result

Obviously, they are not the same program according to the execute result but they have the same MD5 HASH value.

3.4 Task 11: Making the Two Programs Behave Differently

```
# The same operation as TASK10
$ head -c 4160 program > prefix
$ md5collgen -p prefix -o out1.bin out2.bin
$ tail -c +4288 program > suffix
$ cat out1.bin suffix > test1
$ cat out2.bin suffix > test2
# Use bless modify both Y Array to be same as the X Array of test1
$ bless test1
$ bless test2
$ ./test1 # the X array and Y array are the same in test1
This is a good program
$ ./test2 # the X array and Y array are different in test2
This is a bad program
# Observe the MD5 HASH, they are the same
$ md5sum test1
8022e63d3dba85eb0ba3278ff78485c6  test1
$ md5sum test2
8022e63d3dba85eb0ba3278ff78485c6  test2
```

Listing 11: Different Behavior Program

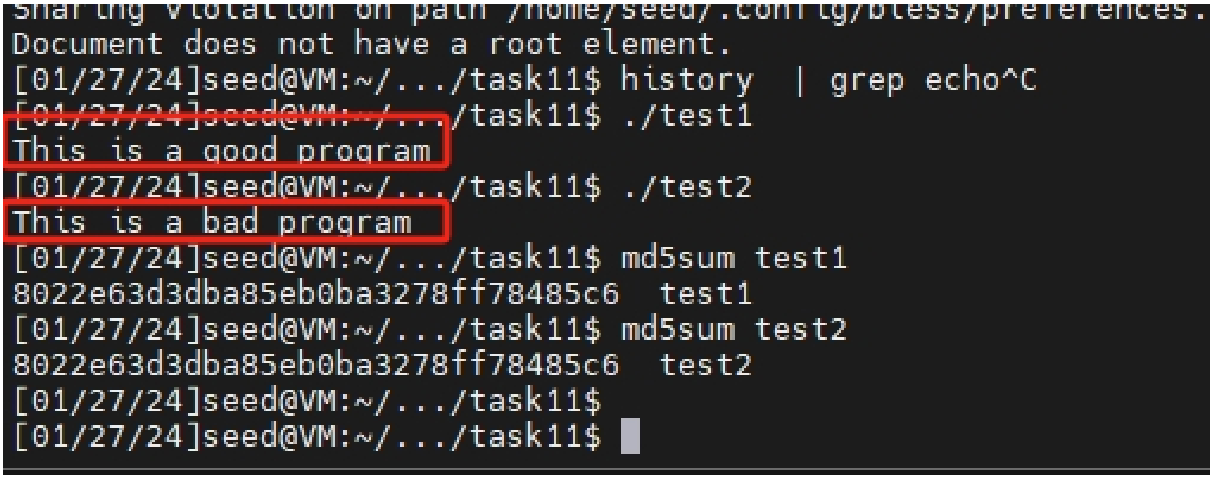


Figure 6: Different Behavior

(a) same arrays in test1

(b) different arrays in test2

Figure 7: test program binary

4 RSA Public-Key Encryption and Signature

4.1 BIGNUM APIs

4.2 A Complete Example

The running example result is:

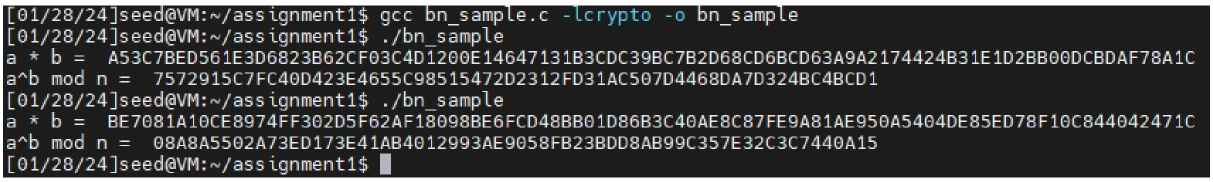


Figure 8: BIGNUM API Example

4.3 Task 12: Deriving the Private Key

```
#include <stdio.h>
#include <openssl/bn.h>
int main() {
    // Given values
    char p_hex[] = "F7E75FDC469067FFDC4E847C51F452DF";
    char q_hex[] = "E85CED54AF57E53E092113E62F436F4F";
    char e_hex[] = "0D88C3";
```

```
    // Convert hexadecimal strings to BIGNUM
    BIGNUM *p = BN_new();
    BIGNUM *q = BN_new();
    BIGNUM *e = BN_new();
    BN_hex2bn(&p, p_hex);
    BN_hex2bn(&q, q_hex);
    BN_hex2bn(&e, e_hex);
    // Calculate phi(n) = (p-1)*(q-1)
    BIGNUM *phi_n = BN_new();
    BN_sub(phi_n, p, BN_value_one());
    BN_sub_word(phi_n, 1);
    BN_mul(phi_n, phi_n, q, BN_CTX_new());
    BN_sub_word(phi_n, 1);
    // Calculate d = e^-1 mod phi(n)
    BIGNUM *d = BN_new();
    BN_mod_inverse(d, e, phi_n, BN_CTX_new());
    // Print the private key d
    char *d_hex = BN_bn2hex(d);
    printf("Private Key (d): %s\n", d_hex);
    // Free allocated memory
    return 0;
}
```

Listing 12: C Program Code for Deriving the Private Key

```
$ gcc main.c -lcrypto -o main
$ ./main
Private Key (d): 9B740D556F9080815E14B6633E9BCC3C87EAA0F0AD699E0A7E0719A725A94AA7
```

Listing 13: Result of the Task 12

4.4 Task 13: Encrypting a Message

```
#include <stdio.h>
#include <string.h>
#include <openssl/bn.h>
int main() {
    // Given public key values
    char n_hex[] = "DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5";
    char e_hex[] = "010001";
    // Given message
    char message_hex[] = "4120746f702073656372657421";  // Hex representation of "A top
    secret!"
    // Convert public key values to BIGNUM
    BIGNUM *n = BN_new();
    BIGNUM *e_value = BN_new();
    BN_hex2bn(&n, n_hex);
    BN_hex2bn(&e_value, e_hex);
    // Convert the message from hex to BIGNUM
    BIGNUM *message_bn = BN_new();
    BN_hex2bn(&message_bn, message_hex);
    // Allocate memory for the result
    BIGNUM *result = BN_new();
    // Perform encryption: ciphertext = message^e mod n
    BN_mod_exp(result, message_bn, e_value, n, BN_CTX_new());
    // Print the encrypted result
    char *result_hex = BN_bn2hex(result);
    printf("Encrypted Result: %s\n", result_hex);
    // Free allocated memory
    return 0;
}
```

Listing 14: C Program Code for Encrypt

```
#include <stdio.h>
#include <openssl/bn.h>
int main() {
    // Given private key values
    char n_hex[] = "DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5";
    char e_hex[] = "010001";
    char d_hex[] = "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D";
    // Given ciphertext
    char ciphertext_hex[] = "6
    FB078DA550B2650832661E14F4F8D2CFAEF475A0DF3A75CACDC5DE5CFC5FADC";
    // Convert private key values to BIGNUM
```

```c
    BIGNUM *n = BN_new();
    BIGNUM *e_value = BN_new();
    BIGNUM *d = BN_new();
    BN_hex2bn(&n, n_hex);
    BN_hex2bn(&e_value, e_hex);
    BN_hex2bn(&d, d_hex);
    // Convert ciphertext from hex to BIGNUM
    BIGNUM *ciphertext_bn = BN_new();
    BN_hex2bn(&ciphertext_bn, ciphertext_hex);
    // Allocate memory for the result
    BIGNUM *result = BN_new();
    // Perform decryption: message = ciphertext^d mod n
    BN_mod_exp(result, ciphertext_bn, d, n, BN_CTX_new());
    // Print the decrypted result
    char *result_hex = BN_bn2hex(result);
    printf("Decrypted Result: %s\n", result_hex);
    // Free allocated memory
    return 0;
}
```

Listing 15: C Program Code for Evaluation

```
$ ./main
Encrypted Result: 6FB078DA550B2650832661E14F4F8D2CFAEF475A0DF3A75CACDC5DE5CFC5FADC
$ ./decrypt
Decrypted Result: 4120746F702073656372657421
$ python -c 'print("4120746F702073656372657421".decode("hex"))'
A top secret!
```

Listing 16: Result of the Task 13

4.5 Task 14: Decrypting a Message

This Task is a lot of same with the Task 13. Just change the ciphertext value.

```c
//same with the Task 13 evaluation
#include <stdio.h>
#include <openssl/bn.h>
int main() {
    // Given private key values
    // Given ciphertext
    char ciphertext_hex[] = "8
    C0F971DF2F3672B28811407E2DABBE1DA0FEBBBDFC7DCB67396567EA1E2493F";
    // Convert private key values to BIGNUM
    //...
    // Free allocated memory
    return 0;
}
```

Listing 17: C Program Code for Decryption

```
$ gcc decrypt.c -lcrypto -o decrypt
$ ./decrypt
Decrypted Result: 50617373776F72642069732064656573
$ python -c 'print("50617373776F72642069732064656573".decode("hex"))'
Password is dees
```

Listing 18: Result of the Task 14

4.6 Task 15: Signing a Message

Signature the message is using $S = m^d \; mod \; n$.

```
$ python -c 'print("I owe you $2000.".encode("hex"))'
49206f776520796f752024323030302e # covert the text to hex
$ python -c 'print("I owe you $3000.".encode("hex"))'
49206f776520796f752024333030302e # covert the text to hex
$ gcc sign.c -lcrypto -o sign # sign.c is listing as Listing 20
$ ./sign
Signing: I owe you $2000.
Signature Result: 55A4E7F17F04CCFE2766E1EB32ADDBA890BBE92A6FBE2D785ED6E73CCB35E4CB
Signing: I owe you $3000.
Signature Result: BCC20FB7568E5D48E434C387C06A6025E90D29D848AF9C3EBAC0135D99305822
```

Listing 19: Result of the Task 15

```c
#include <stdio.h>
#include <openssl/bn.h>
#include <string.h>
int main() {
    //signing the first message
    printf("Signing: I owe you $2000.\n");
    BIGNUM *m = BN_new();
    BN_hex2bn(&m, "49206f776520796f752024323030302e");
    BIGNUM *d = BN_new();
    BIGNUM *n = BN_new();
    BN_hex2bn(&d, "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
    BN_hex2bn(&n, "DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
    BIGNUM *signature = BN_new();
    //s = m ^ d mod n
    BN_mod_exp(signature, m, d, n, BN_CTX_new());
    char *signature_hex = BN_bn2hex(signature);
    printf("Signature Result: %s\n", signature_hex);
    //signing another message
    printf("Signing: I owe you $3000.\n");
    BIGNUM *m2 = BN_new();
    BN_hex2bn(&m2, "49206f776520796f752024333030302e");
    BN_mod_exp(signature, m2, d, n, BN_CTX_new());
    //...
    // Free BIGNUMs
    return 0;
}
```

Listing 20: C Program Code for Signature

Compare both signatures, obviously the message has only 1 bit change while the signature is totally different.

4.7 Task 16: Verifying a Message

Signature the message is using $m' = s^e \ mod \ n$. And then compare the whether $m' = m$?

```c
#include <stdio.h>
#include <openssl/bn.h>
#include <string.h>
int main() {
    // Convert signature to BIGNUM
    BIGNUM *signature = BN_new();
    BN_hex2bn(&signature, "643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6802F"
    );
    // Encrypt the message using Alice's public key
    BIGNUM *e = BN_new();
    BIGNUM *n = BN_new();
    BIGNUM *message = BN_new();
    BN_hex2bn(&e, "010001");
    BN_hex2bn(&n, "AE1CD4DC432798D933779FBD46C6E1247F0CF1233595113AA51B450F18116115");
    BN_hex2bn(&message, "4c61756e63682061206d697373696c652e");
    BIGNUM *test_message = BN_new();
    //m' = s ^ e mod n
    BN_mod_exp(test_message, signature, e, n, BN_CTX_new());
    // Compare the m with m'
    if (BN_cmp(message, test_message) == 0) {
        printf("Signature is valid.\n");
    } else {
        printf("Signature is invalid.\n");
    }
    // Free BIGNUMs
}
```

Listing 21: C Program Code for Verifying

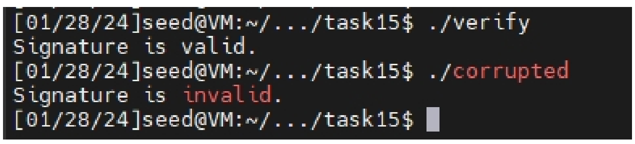


Figure 9: Verification Result

Figure 9 show the result of the right signature and the corrupted one, respectively.
The modified signature (with 3F) won't match the encrypted message generated using Alice's public key.
And RSA signature verification is highly sensitive to even minor changes in the signature or message.

4.8 Task 17: Manually Verifying an X.509 Certificate

According to Step 1 to Step 4, the X.509 certificate detailed is:

```
[01/28/24]seed@VM:~/.../task17$ openssl x509 -in c1.pem -noout -modulus
Modulus=C14BB3654770BCDD4F58DBEC9CEDC366E51F311354AD4A66461F2C0AEC6407E52EDCDCB90A20EDDFE3C4D09E9AA97A1D8288E51156
DB1E9F58C251E72C340D2ED292E156CBF1795FB3BB87CA25037B9A52416610604F571349F0E8376783DFE7D34B674C2251A6DF0E9910ED5751
7426E27DC7CA622E131B7F238825536FC13458008B84FFF8BEA75849227B96ADA2889B15BCA07CDFE951A8D5B0ED37E236B4824B62B5499AEC
C767D6E33EF5E3D6125E44F1BF71427D58840380B18101FAF9CA32BBB48E278727C52B74D4A8D697DEC364F9CACE53A256BC78178E490329AE
FB494FA415B9CEF25C19576D6B79A72BA2272013B5D03D40D321300793EA99F5
```

(a) Modulus

```
                bb:b4:8e:27:87:27:c5:2b:74:d4:a8:d6:97:de:c3:
                64:f9:ca:ce:53:a2:56:bc:78:17:8e:49:03:29:ae:
                fb:49:4f:a4:15:b9:ce:f2:5c:19:57:6d:6b:79:a7:
                2b:a2:27:20:13:b5:d0:3d:40:d3:21:30:07:93:ea:
                99:f5
            Exponent: 65537 (0x10001)
    X509v3 extensions:
        X509v3 Basic Constraints: critical
            CA:TRUE, pathlen:0
        X509v3 Subject Key Identifier:
            B7:6B:A2:EA:A8:AA:84:8C:79:EA:B4:DA:0F:98:B2:C5:95:76:B9:F4
        X509v3 Authority Key Identifier:
            keyid:03:DE:50:35:56:D1:4C:BB:66:F0:A3:E2:1B:1B:C3:97:B2:3D:D1:55
```

(b) Exponent

```
[01/28/24]seed@VM:~/.../task17$ cat signature | tr -d '[:space:]:'
59e44ad8a982ba9a4af1630c6d762675b33c74bec5f73da79192f8cf062d5810edf3b8d6fc6cff139632cd4fe98724850b74a2c2f60ff5a7d8
7d768aaee9c9582b6e006fb9cd24eec442c54c16859d34613923bfc68e95c984a9b2e5410f4478d795b9cfd974bf584fe716ff7c4030c46c4e
224dcb83673a93bf2bc5c59c1af243a1253b84f6f7536ea885aede14749130060df207d4c408ba4364c5e23fdaacc541afa437e8427674f713
bb4a7d3659819bc744df8973b93342e860c24d615d125a10f6efff33891450e8d69fc6b95c2b35dbadeddd36b625f2958aac693f9afe1af815
286dea185ac2d26218af4078b5fa5e098f53f9ccf823a1833123f4c6[01/28/24]seed@VM:~/.../task17$
```

(c) Signature

```
[01/28/24]seed@VM:~/.../task17$ sha256sum c0_body.bin
bbc2a75949c896bd66db4e636aab8b2cbaa970bc8302d8d02c99104ab04f4dd6  c0_body.bin
```

(d) Body HASH

Figure 10: Task 17 Information

And then, a program to verity the information as follows:

```c
#include <stdio.h>
#include <openssl/bn.h>
#include <string.h>
int main()
{
    char bodyhash[] = "BBC2A75949C896BD66DB4E...302D8D02C99104AB04F4DD6"; # the Body HASH
    BIGNUM *n = BN_new();
    BIGNUM *e = BN_new();
    BIGNUM *s = BN_new();
    BIGNUM *m = BN_new();
    BN_hex2bn(&n, "C14BB3654770BC...F5"); # the Modulus
    BN_hex2bn(&e, "010001"); # the Exponent
    BN_hex2bn(&s, "59e44ad8a982ba9a4af163...23f4c6"); # the Signature
    BN_mod_exp(m, s, e, n, BN_CTX_new());
    char *message_hex = BN_bn2hex(m);
    size_t length = strlen(message_hex);
    //compare the last 64 bytes
    if (length > 64) {
        // Move the pointer to the start of the last 64 bytes
        char *substring = message_hex + (length - 64);
        // Print or manipulate the substring as needed
        int result = strcmp(substring, bodyhash);
        if (result == 0) {
            printf("Verification PASS.\n");
        } else {
            printf("Verification Failed\n");
        }
    } else {
        // If the string is shorter than 64 bytes, handle it accordingly
        printf("Something Wrong");
    }
    return 0;
}
```

Listing 22: C Program Code for Manually Verifying

```
[01/28/24]seed@VM:~/.../task17$ ./verify
Verification PASS.
[01/28/24]seed@VM:~/.../task17$
```

Figure 11: Verification Result

5 Pseudo Random Number Generation

5.1 Task 18: Generate Encryption Key in a Wrong Way

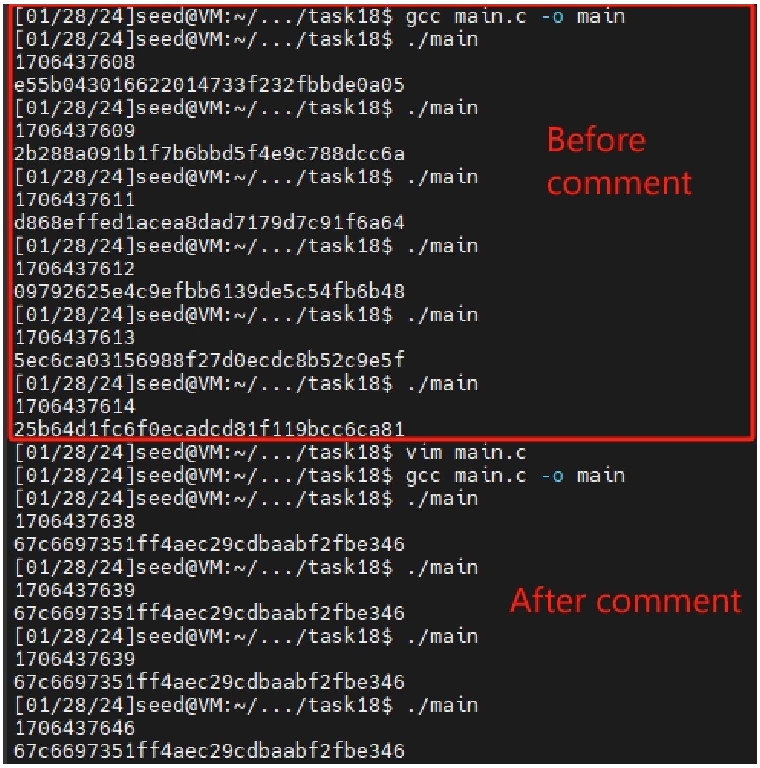


Figure 12: Result Observation

The `time(NULL)` function returns the current time as the number of seconds since the Epoch (1970-01-01 00:00:00 +0000 UTC). It prints this value to the console.
`srand(time(NULL))` seeds the pseudo-random number generator (`rand()`) with the current time. The purpose of seeding is to initialize the random number generator with a somewhat unpredictable value, ensuring that subsequent calls to rand() produce different sequences of pseudo-random numbers.
When comment out the line `srand(time(NULL))` (Line 13) and rerun the program, the pseudo-random numbers generated by `rand()` will be the same every time when running the program because the generator is not reseeded.
In summary, the `srand(time(NULL))` line is responsible for seeding the pseudo-random number generator based on the current time, introducing unpredictability and ensuring a different sequence of pseudo-random numbers each time the program is executed.

5.2 Task 19: Guessing the Key

Firstly, I program a C program to generate all potential key from time to time accordingly.

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define KEYSIZE 16
void main() {
    long long int timestamp;
    for(timestamp = 1523920129; timestamp <= 1523920129+2*24*60*60; timestamp++) {
        char key[KEYSIZE];
        srand(timestamp);
        for (int i = 0; i< KEYSIZE; i++){
            key[i] = rand()%256;
            printf("%.2x", (unsigned char)key[i]);
        }
        printf("\n");
    }
}
```

Listing 23: C Program Code for Generating Keys

And using genkey > key.txt to generate a key files, after generating, then I use a Python script to find the key.

```python
import binascii
from Crypto.Cipher import AES
# Read keys from the file
with open('./key.txt') as fp:
    keys = fp.readlines()
# Iterate through each key in the file
for keyhex in keys:
    # Remove trailing newline characters
    keyhex = keyhex.rstrip()
    # Convert IV, key, and plaintext from hex to bytes
    iv = binascii.unhexlify('09080706050403020100A2B2C2D2E2F2'.lower())
    key = binascii.unhexlify(keyhex.lower())
    plaintext = binascii.unhexlify('255044462d312e350a25d0d4c5d80a34'.lower())
    # Create AES encryptor object
    encryptor = AES.new(key, AES.MODE_CBC, iv)
    # Encrypt the plaintext
    ciphertext = encryptor.encrypt(plaintext)
    # Check if the ciphertext matches the known value
    if ciphertext == binascii.unhexlify('d06bf9d0dab8e8ef880660d2af65aa82'.lower()):
        print("Key Found: " + binascii.hexlify(key))
```

Listing 24: Python Script for Finding the Key

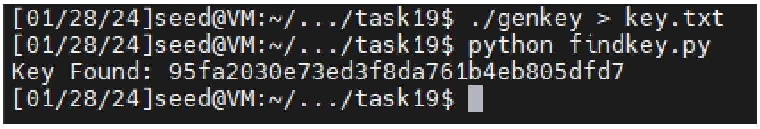


Figure 13: Result

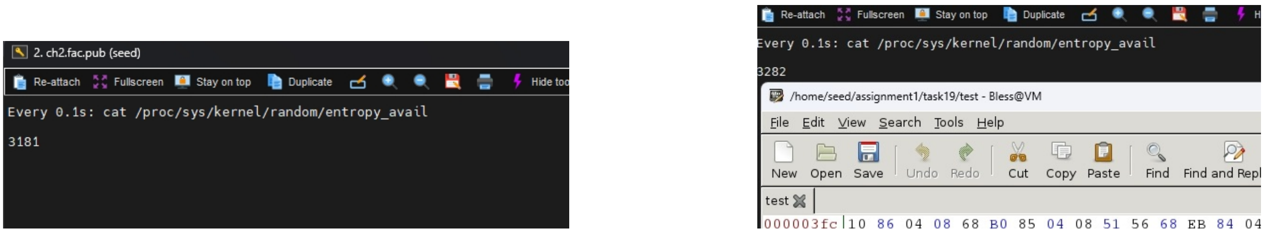
And finally find the key: `95fa2030e73ed3f8da761b4eb805dfd7`

5.3 Task 20: Measure the Entropy of Kernel

Since my Seed lab is running in a remote VM, I try to measure the entropy by such:

- User input: Typing rapidly and randomly on the keyboard introduces unpredictable timing between key presses, boosting entropy.
- Moving the mouse remotely in X11 erratically and clicking randomly generates unpredictable mouse events, adding to entropy.
- Disk activity: Reading large files from a hard drive or SSD creates variable disk access times due to seek times and read speeds, contributing to entropy.
- Network activity: Visiting the server by ssh involves network interactions that introduce variability in packet arrival times and server responses, increasing entropy. Downloading large files also involves unpredictable network delays and data transfer patterns.

These activities significantly increase entropy.



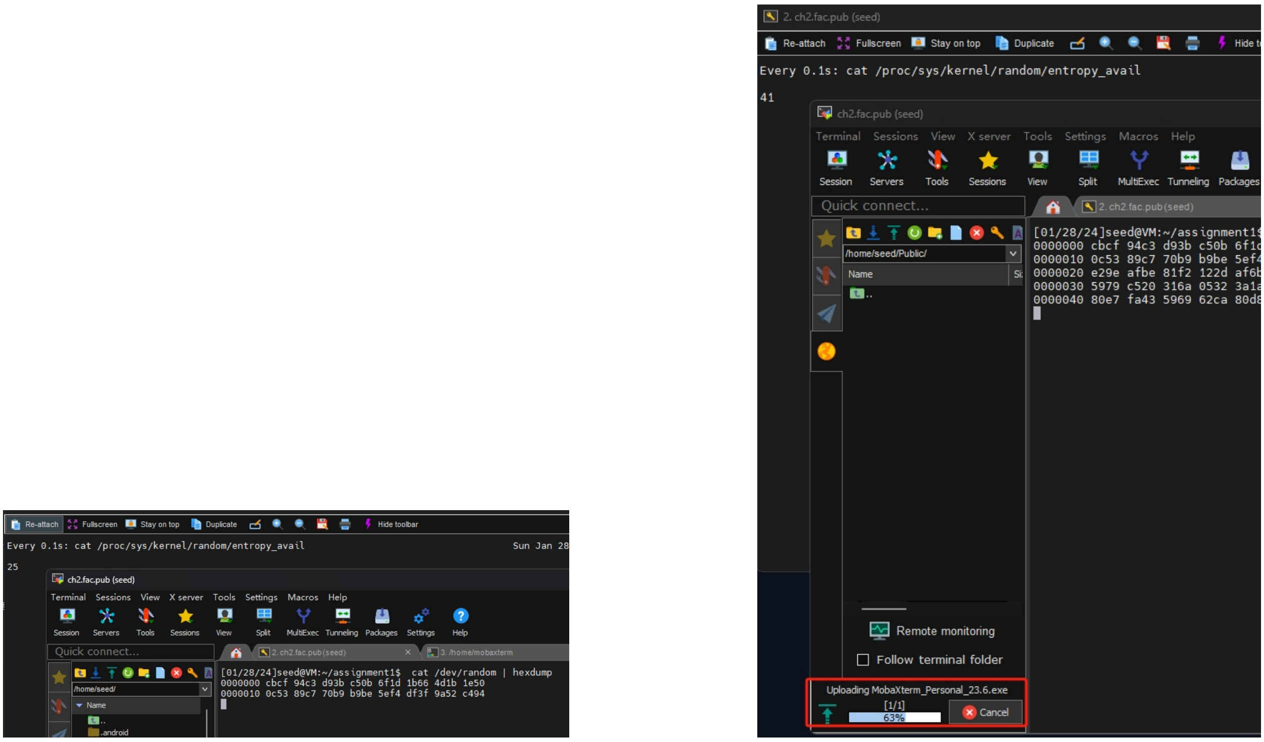
(a) Background Entropy

(b) Activity Entropy

Figure 14: Task 20 Observation

5.4 Task 21: Get Pseudo Random Numbers from /dev/random

In the task, when execute the command, `cat /dev/random | hexdump`, the entropy will decrease significantly, when the entropy approaching 0, the command will stop outputing any information. And I try to increase the entropy by upload a file to the server, shown as 15(b),



(a) Background Entropy

(b) Upload a file for increasign the Entropy

Figure 15: Task 21 Observation

Q: Launching a DoS Attack:
Exhaust Entropy Pool: Flood the server with requests that require random numbers from `/dev/random`. This depletes the entropy pool faster than it can be replenished, causing `/dev/random` to block.
Prevent Entropy Refilling: Avoid activities that generate entropy, like mouse movements, keyboard input, or disk I/O. This prevents the server from replenishing the entropy pool and resuming normal operations.
Impact: The server will become unresponsive to new requests that require random numbers, effectively denying service.
Prevention: Use `/dev/urandom` for non-critical randomness: `/dev/urandom` doesn't block and is suitable for most applications. Consider hardware RNGs: Hardware random number generators provide a more resilient source of entropy. Implement rate limiting: Restrict the rate of requests that use `/dev/random` to mitigate depletion attacks. Monitor entropy levels: Set up alerts for low entropy conditions to allow for proactive measures.

5.5 Task 22: Get Random Numbers from /dev/urandom

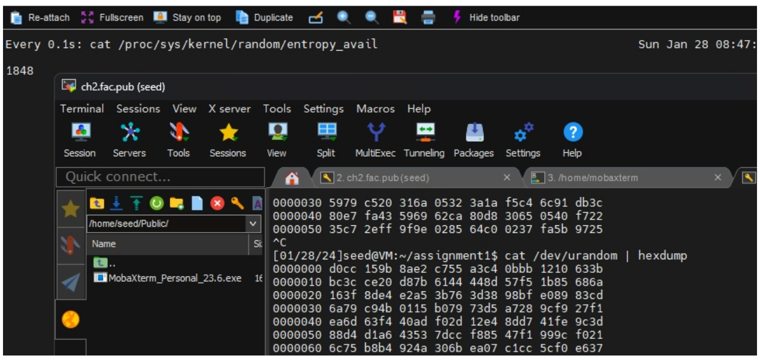


Figure 16: Not block random number generating by /dev/urandom

```
[01/28/24]seed@VM:~/assignment1$ head -c 1M /dev/urandom > output.bin
[01/28/24]seed@VM:~/assignment1$ ent output.bin
Entropy = 7.999847 bits per byte.

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

Chi square distribution for 1048576 samples is 222.59, and randomly
would exceed this value 92.94 percent of the times.

Arithmetic mean value of data bytes is 127.5297 (127.5 = random).
Monte Carlo value for Pi is 3.143108914 (error 0.05 percent).
Serial correlation coefficient is -0.000978 (totally uncorrelated = 0.0).
```

Figure 17: Measurement of the random number generating by /dev/urandom

Behavior of `/dev/urandom`:
No blocking: Unlike `/dev/random`, `/dev/urandom` won't block, even when the entropy pool is low. It will continue generating pseudorandom numbers using the available seed.
Network workload: won't have a noticeable effect on the output of `cat /dev/urandom | hexdump`. This is because `/dev/urandom` doesn't directly rely on real-time entropy input.
Quality of Random Numbers: Testing with ent: The ent tool evaluates randomness based on statistical tests. Good results typically indicate: Entropy estimates close to 8 bits per byte.
Passing most or all statistical tests. No significant patterns or correlations in the data.

```c
#include <stdio.h>
#include <stdlib.h>
#define KEY_LEN 32  // 256 bits
int main() {
    unsigned char *key = (unsigned char *)malloc(sizeof(unsigned char) * KEY_LEN);
    FILE *random = fopen("/dev/urandom", "r");
    if (random == NULL) {
        perror("Error opening /dev/urandom");
        return 1;
    }
    fread(key, sizeof(unsigned char) * KEY_LEN, 1, random);
    fclose(random);
    // Print the generated key (replace with secure handling for actual use):
    printf("Generated 256-bit encryption key: ");
    for (int i = 0; i < KEY_LEN; i++) {
        printf("%02x", key[i]);
    }
    printf("\n");
    free(key);
    return 0;
}
```

Listing 25: C Program Code for Generating Keys by /dev/urandom

```
[01/28/24]seed@VM:~/.../task22$ gcc main.c -o main
[01/28/24]seed@VM:~/.../task22$ ./main
Generated 256-bit encryption key: 0e21b839ec74024298fe257960a275c330135a32b84d77b7dceadd16f117e4b8
[01/28/24]seed@VM:~/.../task22$ ./main
Generated 256-bit encryption key: 1f5f073ac04245b6ca2d7a60d8bd6cec8fe588baebba7f1c2125b29efaf3df51
[01/28/24]seed@VM:~/.../task22$ ./main
Generated 256-bit encryption key: 8804308c13342292a5b33da41167018078217d3034fc031e09aa5aabb6d41b7d
[01/28/24]seed@VM:~/.../task22$
```

Figure 18: Generating Key by /dev/urandom

This code generates a 256-bit encryption key using `/dev/urandom`. It allocates memory for the key, opens `/dev/urandom`, reads random bytes into the key, prints the key in hexadecimal format, and then frees the allocated memory. Compile and run this code to obtain your 256-bit encryption key.