# SYSC 4810 ASSIGNMENT 2 FINAL REPORT
Cryptographic Tools

**Shivanshi Sharma**
**101037387**

Professor: Dr. J. Jaskolka
Due: Sunday, October 11, 2020

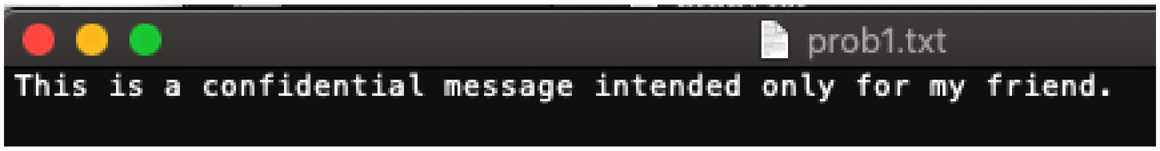
## Introduction:

This assignment is aimed towards comparing and contrasting between various cryptographic mechanisms, to determine which one fits best for the company RoadRunner Systems, Inc. There are multiple problems to test various aspects such as finding the best encryption method for images of license plates, protecting confidentiality, working with digital signatures, and etcetera. These problems will be implemented using a pre-built virtual machine, to help encrypt-decrypt any form of data for testing purposes. Furthermore, the security requirements of the company must be fulfilled while choosing the best cryptographic mechanism.

## Part III Symmetric Cryptography:

### Problem 1:

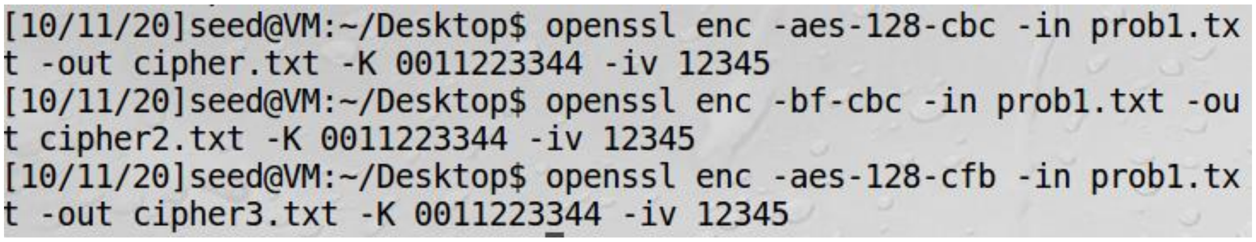
This problem helps deliver an insight of symmetric encryption using different ciphers and modes, to observe the behaviour of a plaintext message. Figure 1 below shows the plaintext file we started with.



```
This is a confidential message intended only for my friend.
```

Figure 1: Starting plaintext file

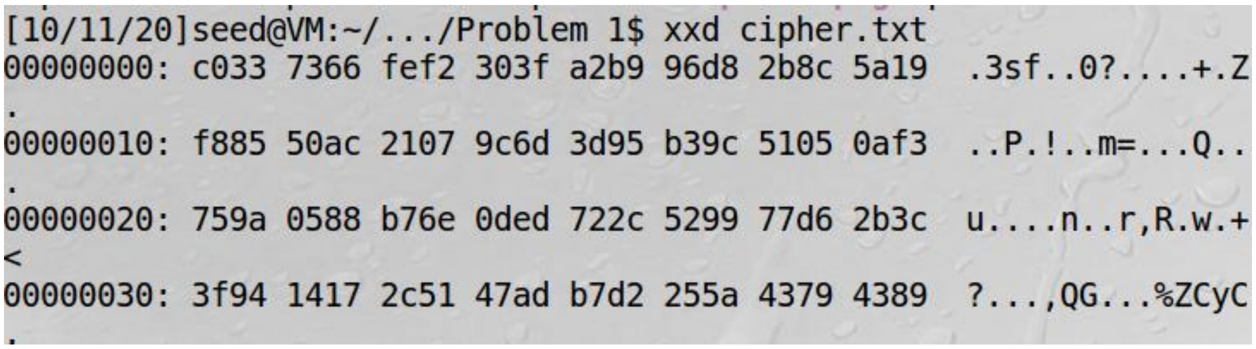
Part b) included encrypting the file using 3 different cipher types which use the same key and initialization vector. The 3 difference cipher types and modes used are: AES-128-CBC, BF-CBC and AES-128-CFB, as shown in Figure 2.



```
[10/11/20]seed@VM:~/Desktop$ openssl enc -aes-128-cbc -in prob1.tx
t -out cipher.txt -K 0011223344 -iv 12345
[10/11/20]seed@VM:~/Desktop$ openssl enc -bf-cbc -in prob1.txt -ou
t cipher2.txt -K 0011223344 -iv 12345
[10/11/20]seed@VM:~/Desktop$ openssl enc -aes-128-cfb -in prob1.tx
t -out cipher3.txt -K 0011223344 -iv 12345
```

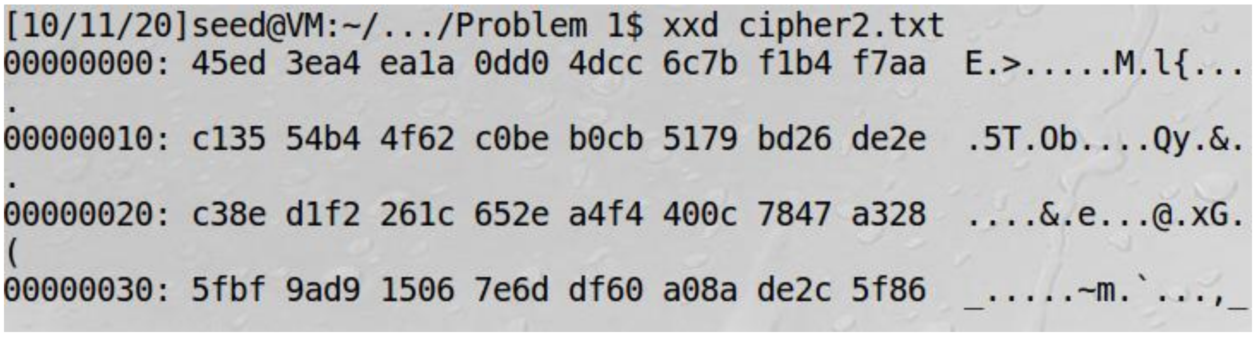
Figure 2: 3 different cipher types and modes used

After verifying the output using the command $ xxd cipher.txt the three cipher texts produced are as follows:



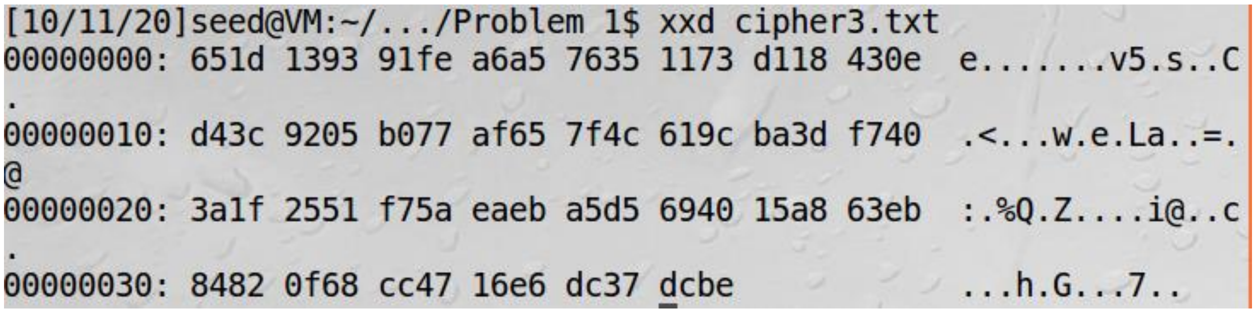
```
[10/11/20]seed@VM:~/.../Problem 1$ xxd cipher.txt
00000000: c033 7366 fef2 303f a2b9 96d8 2b8c 5a19  .3sf..0?....+.Z
.
00000010: f885 50ac 2107 9c6d 3d95 b39c 5105 0af3  ..P.!..m=...Q..
.
00000020: 759a 0588 b76e 0ded 722c 5299 77d6 2b3c  u....n..r,R.w.+
<
00000030: 3f94 1417 2c51 47ad b7d2 255a 4379 4389  ?...,QG...%ZCyC
.
```

Figure 3a: Using AES-128-CBC



```
[10/11/20]seed@VM:~/.../Problem 1$ xxd cipher2.txt
00000000: 45ed 3ea4 ea1a 0dd0 4dcc 6c7b f1b4 f7aa  E.>.....M.l{...
.
00000010: c135 54b4 4f62 c0be b0cb 5179 bd26 de2e  .5T.Ob....Qy.&.
.
00000020: c38e d1f2 261c 652e a4f4 400c 7847 a328  ....&.e...@.xG.
(
00000030: 5fbf 9ad9 1506 7e6d df60 a08a de2c 5f86  _.....~m.`...,_
```

Figure 3b: Using BF-CBC



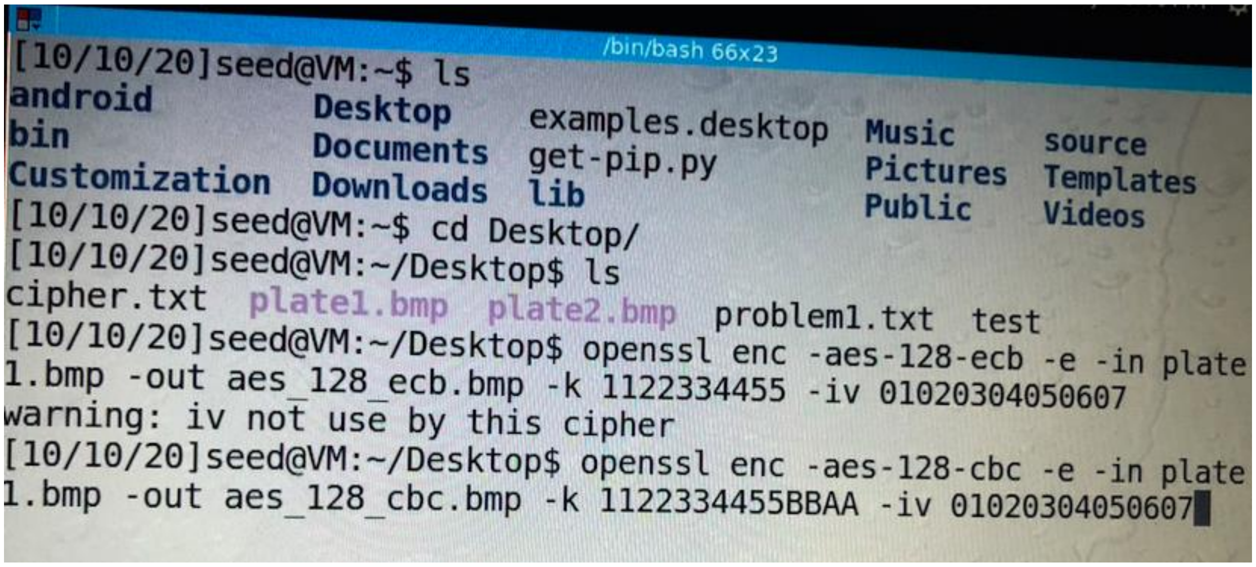
```
[10/11/20]seed@VM:~/.../Problem 1$ xxd cipher3.txt
00000000: 651d 1393 91fe a6a5 7635 1173 d118 430e  e.......v5.s..C
.
00000010: d43c 9205 b077 af65 7f4c 619c ba3d f740  .<...w.e.La..=.
@
00000020: 3a1f 2551 f75a eaeb a5d5 6940 15a8 63eb  :.%Q.Z....i@..c
.
00000030: 8482 0f68 cc47 16e6 dc37 dcbe            ...h.G...7..
```

Figure 3c: Using AES-128-CFB

Figure 3 shows the output of the encrypted files using different encryptions. Each of these methods produce a completely different ciphertext, which are not similar at all. However, they follow a similar structure of hexadecimal patterns and are each successful at hiding the data.

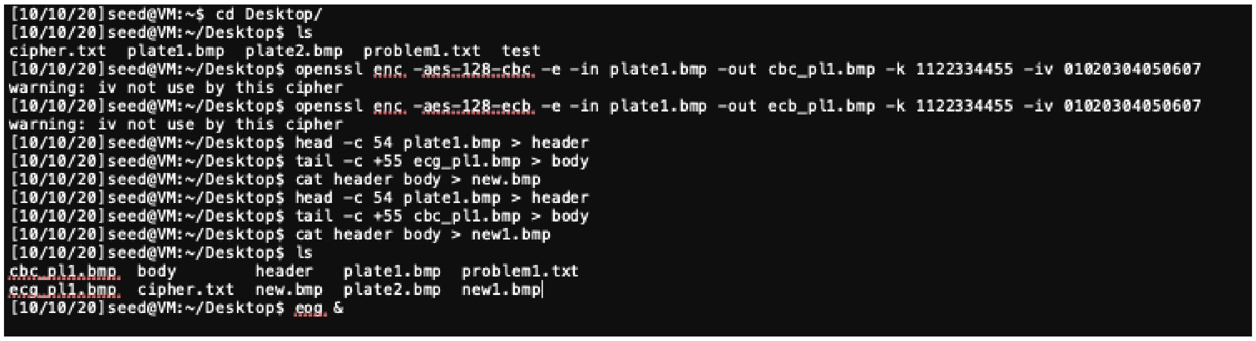
## Problem 2:

The goal of this problem was to help determine the best choice of encryption mode for images. The experiment was done using two difference license plates, to contrast between ECB (Electronic Code Book) and CBC (Cipher Block Chaining). The following figure (Figure 4) is an example of how the plates were encrypted with each mode using *openssl.*



```
[10/10/20]seed@VM:~$ ls
android        Desktop    examples.desktop  Music     source
bin            Documents  get-pip.py        Pictures  Templates
Customization  Downloads  lib               Public    Videos
[10/10/20]seed@VM:~$ cd Desktop/
[10/10/20]seed@VM:~/Desktop$ ls
cipher.txt  plate1.bmp  plate2.bmp  problem1.txt  test
[10/10/20]seed@VM:~/Desktop$ openssl enc -aes-128-ecb -e -in plate
1.bmp -out aes_128_ecb.bmp -k 1122334455 -iv 01020304050607
warning: iv not use by this cipher
[10/10/20]seed@VM:~/Desktop$ openssl enc -aes-128-cbc -e -in plate
1.bmp -out aes_128_cbc.bmp -k 1122334455BBAA -iv 01020304050607
```

Figure 4a: Encrypting Plate 1 using ECB and CBC modes



```
[10/10/20]seed@VM:~$ cd Desktop/
[10/10/20]seed@VM:~/Desktop$ ls
cipher.txt  plate1.bmp  plate2.bmp  problem1.txt  test
[10/10/20]seed@VM:~/Desktop$ openssl enc -aes-128-cbc -e -in plate1.bmp -out cbc_pl1.bmp -k 1122334455 -iv 01020304050607
warning: iv not use by this cipher
[10/10/20]seed@VM:~/Desktop$ openssl enc -aes-128-ecb -e -in plate1.bmp -out ecb_pl1.bmp -k 1122334455 -iv 01020304050607
warning: iv not use by this cipher
[10/10/20]seed@VM:~/Desktop$ head -c 54 plate1.bmp > header
[10/10/20]seed@VM:~/Desktop$ tail -c +55 ecg_pl1.bmp > body
[10/10/20]seed@VM:~/Desktop$ cat header body > new.bmp
[10/10/20]seed@VM:~/Desktop$ head -c 54 plate1.bmp > header
[10/10/20]seed@VM:~/Desktop$ tail -c +55 cbc_pl1.bmp > body
[10/10/20]seed@VM:~/Desktop$ cat header body > new1.bmp
[10/10/20]seed@VM:~/Desktop$ ls
cbc_pl1.bmp  body        header   plate1.bmp  problem1.txt
ecg_pl1.bmp  cipher.txt  new.bmp  plate2.bmp  new1.bmp
[10/10/20]seed@VM:~/Desktop$ eog &
```

Figure 4b: Replacing the encrypted headers

The encrypted images after using the ECB and CBC modes on ‘plate1.bmp’ showed different results for both encryptions, as seen in figures 5a) and 5b) below respectively. ECB displayed the license plate’s outlines and is therefore not a good choice for encrypting this picture. As mentioned in the book, in ECB, each block of plaintext is encrypted using the same key, so it is unreliable for large strings with similar graphics/colors. On the other hand, CBC masked the image and hid all of its components, making it a more reliable encryption method for this problem. This is because CBC uses an initial vector to XOR the first plaintext, and the outputs produced for the following blocks are always random.

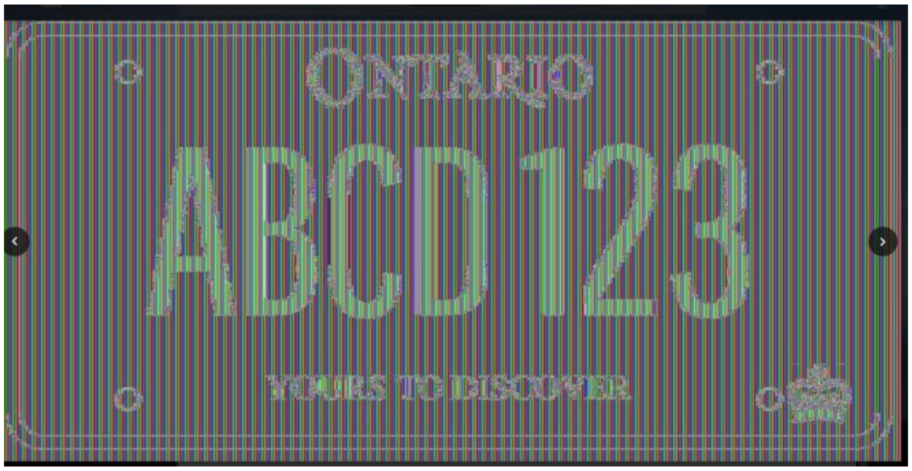


Figure 5a) Output of ECB for Plate 1

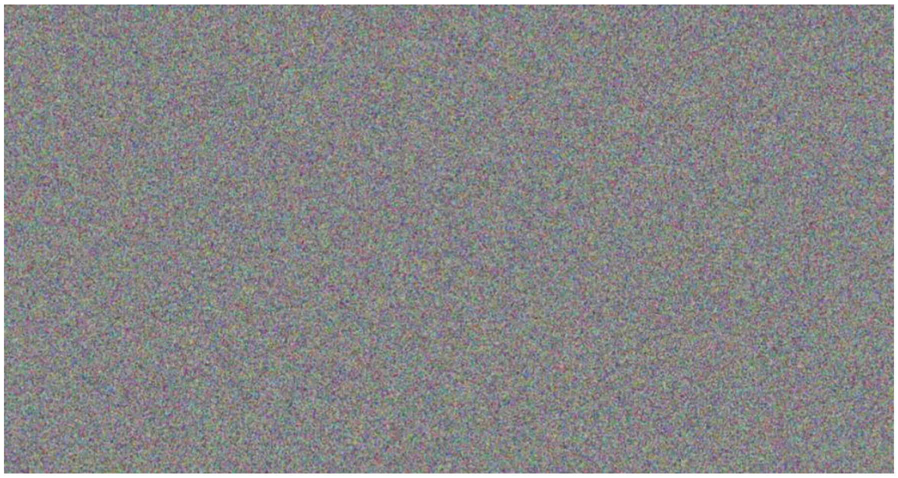


Figure 5b): Output of CBC for plate 1

However, when the same procedure was done using ‘plate2.bmp’, both ECB and CBC mode were able to successfully hide data after encryption. The reason why ECB mode worked on this picture better than the last one, is that the image did not have many consecutive large areas where the colors were the same, so there were less chances of two similar cipher text blocks being placed next to each other. The results can be seen in figures 6a) and 6b) below.

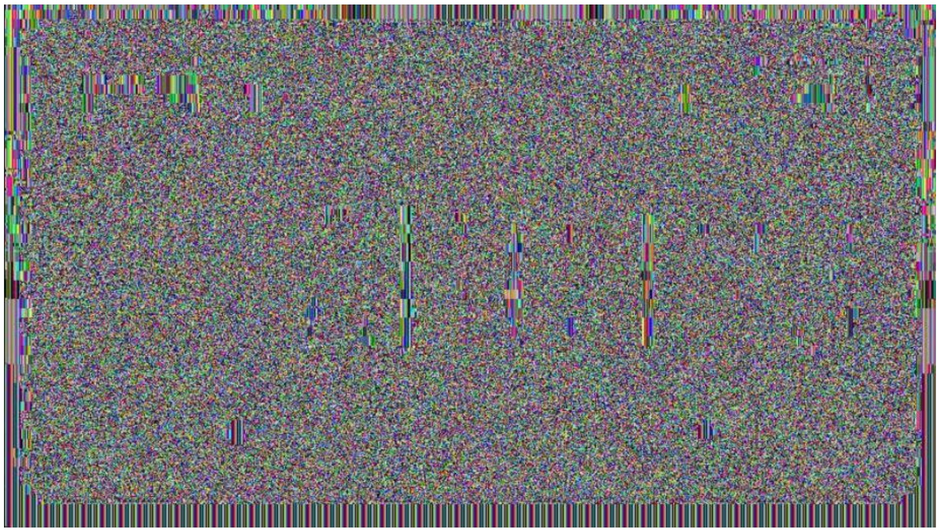


Figure 6a): Output of ECB for Plate 2

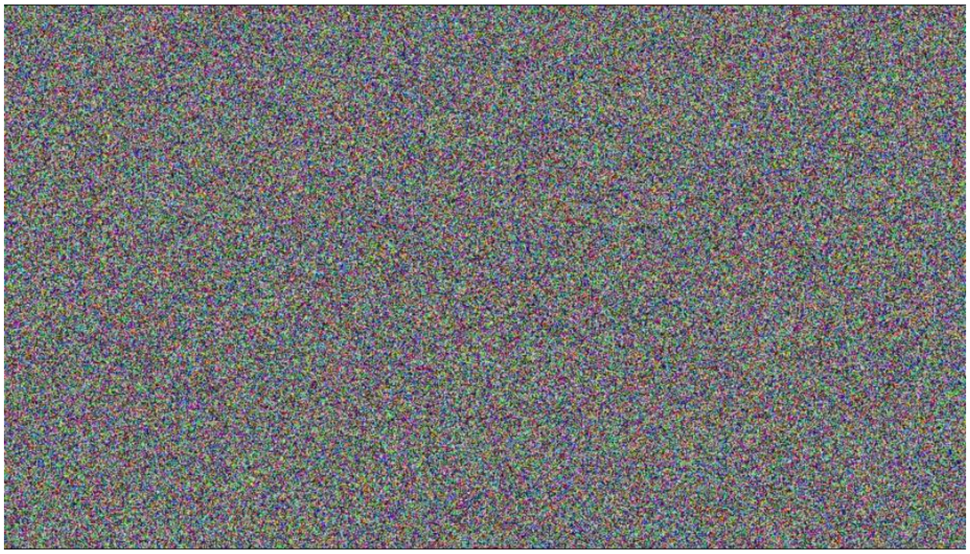
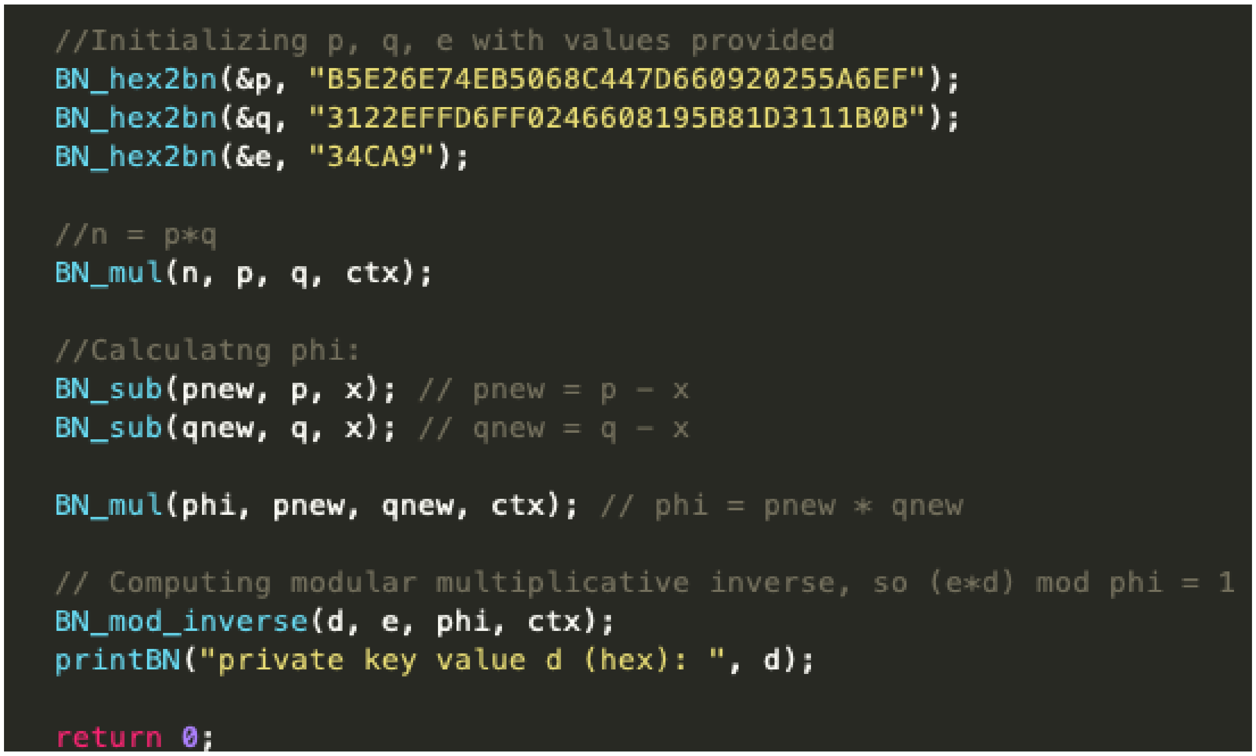


Figure 6b): Output of CBC for plate 2

## Part IV Asymmetric Cryptography

### Problem 3:

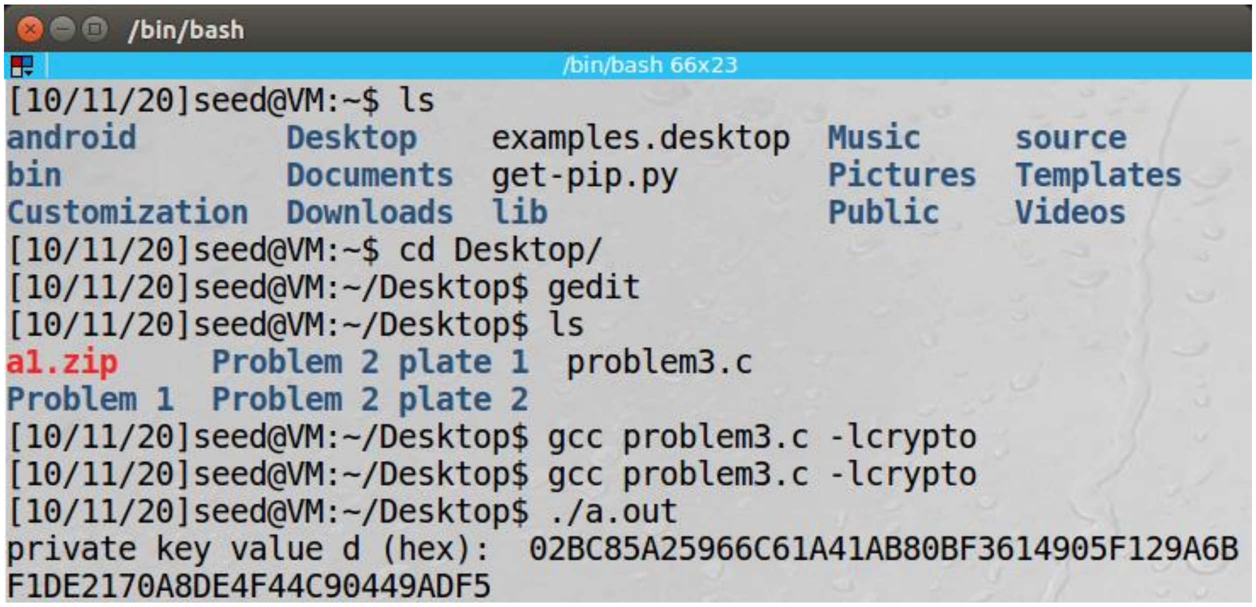
This problem consisted of deriving the private key using a public key (*e*,*n*) with given parameters for p, q, and e. The following snippet of the code (Figure 7) shows the main logic used to derive the answer, after initializing variables with the provided values.



```
//Initializing p, q, e with values provided
BN_hex2bn(&p, "B5E26E74EB5068C447D660920255A6EF");
BN_hex2bn(&q, "3122EFFD6FF0246608195B81D3111B0B");
BN_hex2bn(&e, "34CA9");

//n = p*q
BN_mul(n, p, q, ctx);

//Calculatng phi:
BN_sub(pnew, p, x); // pnew = p - x
BN_sub(qnew, q, x); // qnew = q - x

BN_mul(phi, pnew, qnew, ctx); // phi = pnew * qnew

// Computing modular multiplicative inverse, so (e*d) mod phi = 1
BN_mod_inverse(d, e, phi, ctx);
printBN("private key value d (hex): ", d);

return 0;
```

Figure 7: Problem 3 code snippet

The code shows that first, the n value was calculated using the formula: *n = p x q*. Then, phi was calculated using additional temporary variables, after using the multiplication formula: *res = a x b*. Lastly, the modular multiplicative inverse was calculated to find the private key value *d* in hexadecimal. The output of this problem is show in the next figure.



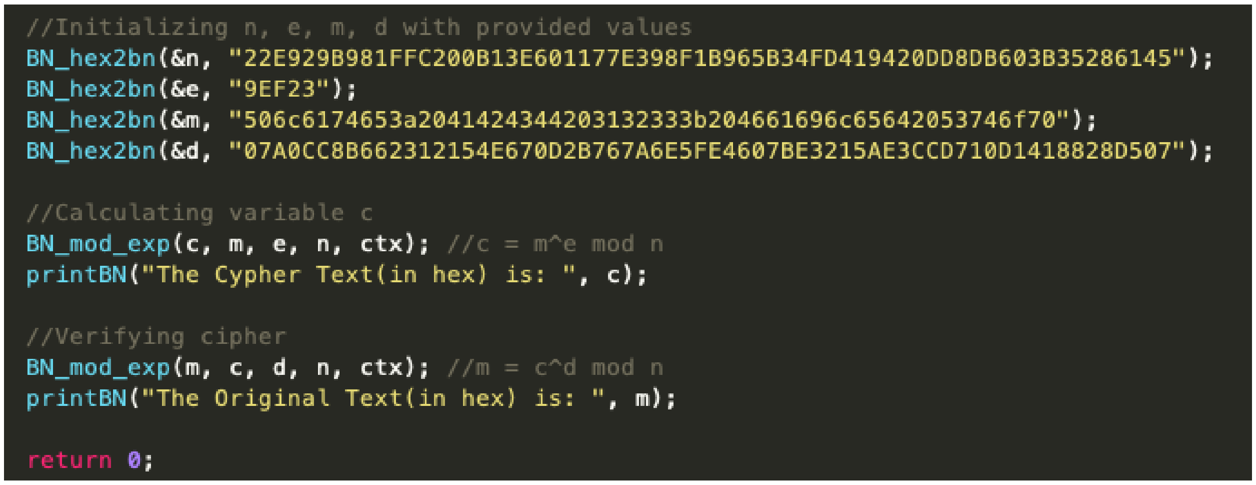
```
[10/11/20]seed@VM:~$ ls
android        Desktop    examples.desktop  Music     source
bin            Documents  get-pip.py        Pictures  Templates
Customization  Downloads  lib               Public    Videos
[10/11/20]seed@VM:~$ cd Desktop/
[10/11/20]seed@VM:~/Desktop$ gedit
[10/11/20]seed@VM:~/Desktop$ ls
a1.zip     Problem 2 plate 1  problem3.c
Problem 1  Problem 2 plate 2
[10/11/20]seed@VM:~/Desktop$ gcc problem3.c -lcrypto
[10/11/20]seed@VM:~/Desktop$ gcc problem3.c -lcrypto
[10/11/20]seed@VM:~/Desktop$ ./a.out
private key value d (hex):  02BC85A25966C61A41AB80BF3614905F129A6B
F1DE2170A8DE4F44C90449ADF5
```

Figure 8: Problem 3 output

The prime numbers (*p* and *q)* used for this problem are relatively small, compared to actual numbers used in practice. Numbers with a bigger size are generally considered more secure, since they require more time to break the cipher (as mentioned in slides). However, 128 bits were used in this question for simplicity, as lower bit sizes have a greater performance.

## Problem 4:

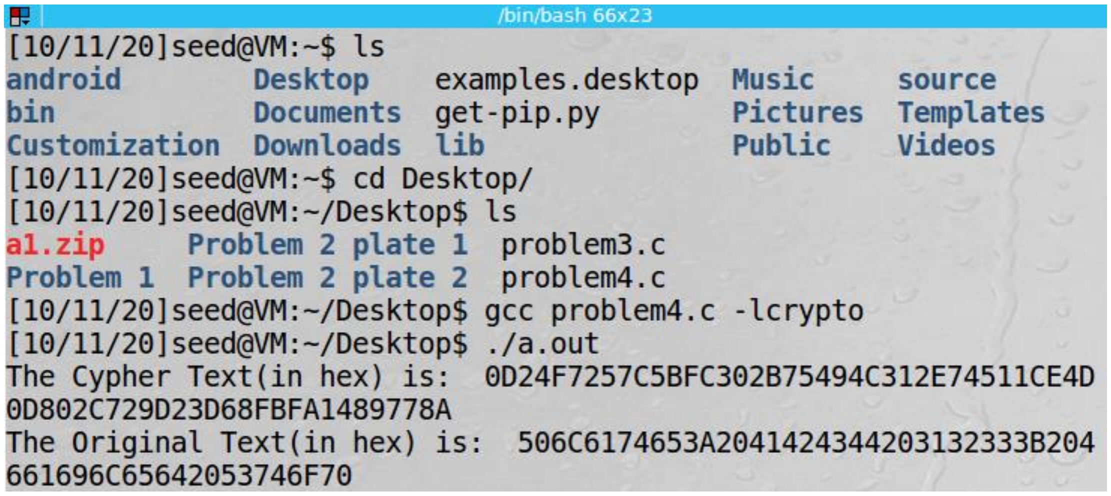
This problem consisted of encrypting a message using various calculations, for asymmetric cryptography. The figure below (Figure 9) shows a snippet of code used to find the result for this problem. Firstly, the necessary variables (n, e, M and d) were initialized using the values provided. Then, variable c, the encrypted message, was calculated using the formula: *res = a^c* mod *n.*



```
//Initializing n, e, m, d with provided values
BN_hex2bn(&n, "22E929B981FFC200B13E601177E398F1B965B34FD419420DD8DB603B35286145");
BN_hex2bn(&e, "9EF23");
BN_hex2bn(&m, "506c6174653a20414243442031323333b204661696c65642053746f70");
BN_hex2bn(&d, "07A0CC8B662312154E670D2B767A6E5FE4607BE3215AE3CCD710D1418828D507");

//Calculating variable c
BN_mod_exp(c, m, e, n, ctx); //c = m^e mod n
printBN("The Cypher Text(in hex) is: ", c);

//Verifying cipher
BN_mod_exp(m, c, d, n, ctx); //m = c^d mod n
printBN("The Original Text(in hex) is: ", m);

return 0;
```

Figure 9: Code for problem 4

In order to verify the result, the original message *M* was calculated to verify that the encrypted message belonged to *M*, using the given private key *d*. The following figure shows the output of the cypher text, and the original message itself.



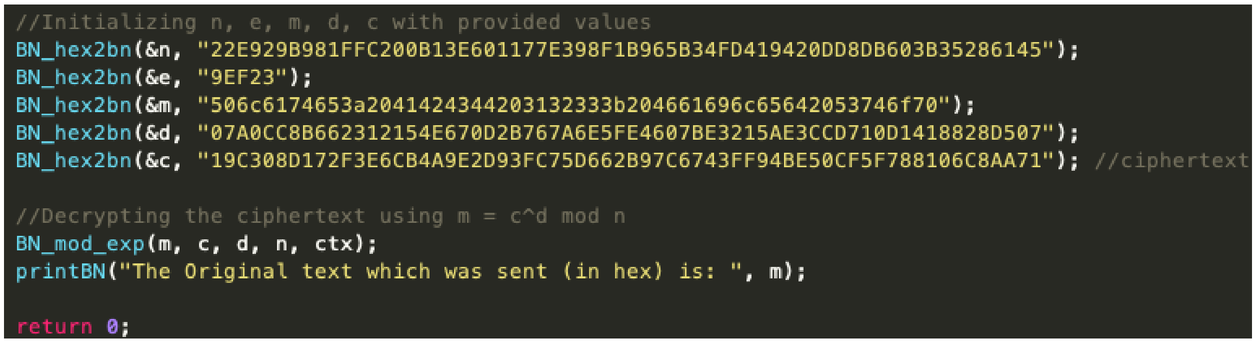
```
/bin/bash 66x23
[10/11/20]seed@VM:~$ ls
android         Desktop     examples.desktop  Music     source
bin             Documents   get-pip.py        Pictures  Templates
Customization   Downloads   lib               Public    Videos
[10/11/20]seed@VM:~$ cd Desktop/
[10/11/20]seed@VM:~/Desktop$ ls
a1.zip     Problem 2 plate 1  problem3.c
Problem 1  Problem 2 plate 2  problem4.c
[10/11/20]seed@VM:~/Desktop$ gcc problem4.c -lcrypto
[10/11/20]seed@VM:~/Desktop$ ./a.out
The Cypher Text(in hex) is:  0D24F7257C5BFC302B75494C312E74511CE4D
0D802C729D23D68FBFA1489778A
The Original Text(in hex) is:  506C6174653A20414243442031323333B204
661696C65642053746F70
```

Figure 10: The output of code showing the cypher text and original message

To conclude, since the original text matches our m value we used to initialize in the code, the resulting encrypted message is therefore correct.

## Problem 5:

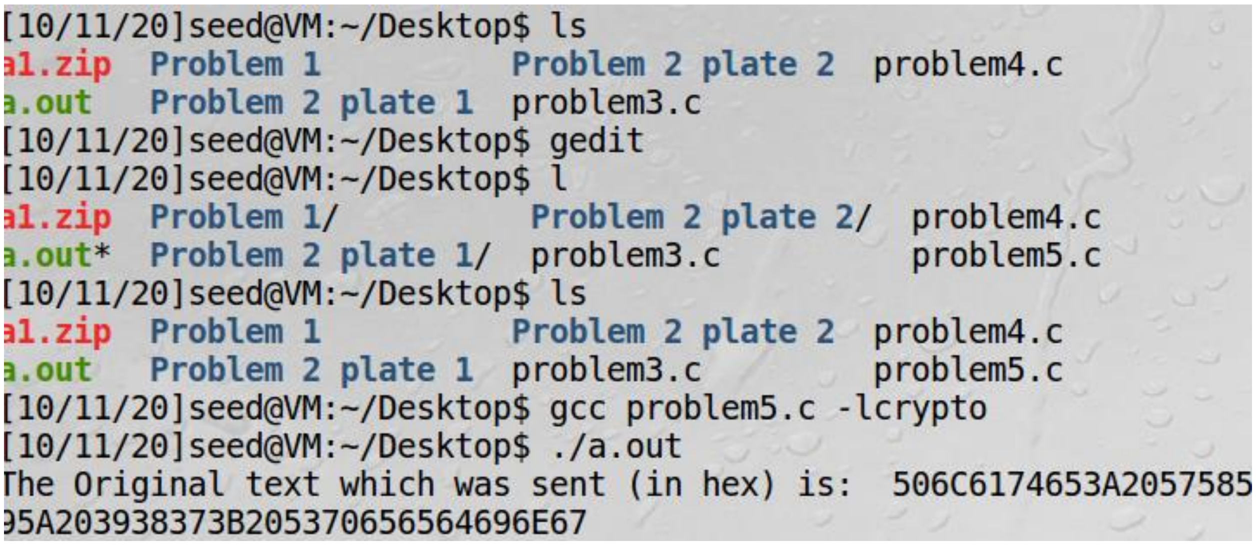
This problem involved decrypting a given ciphertext C using public/private keys. The figure 11 below shows the values used to initialize given variables. The public/private keys were reused from Problem 4. The problem only needed one formula to decrypt the ciphertext, using *res* = *a*^*c* mod *n*.



```
//Initializing n, e, m, d, c with provided values
BN_hex2bn(&n, "22E929B981FFC200B13E601177E398F1B965B34FD419420DD8DB603B35286145");
BN_hex2bn(&e, "9EF23");
BN_hex2bn(&m, "506c6174653a20414243442031323333b204661696c65642053746f70");
BN_hex2bn(&d, "07A0CC8B662312154E670D2B767A6E5FE4607BE3215AE3CCD710D1418828D507");
BN_hex2bn(&c, "19C308D172F3E6CB4A9E2D93FC75D662B97C6743FF94BE50CF5F788106C8AA71"); //ciphertext

//Decrypting the ciphertext using m = c^d mod n
BN_mod_exp(m, c, d, n, ctx);
printBN("The Original text which was sent (in hex) is: ", m);

return 0;
```

Figure 11: Code used for problem 5

The following screenshot shows the output of the code. The decrypted value found matched the given message M value from the question. This verifies that the result is thereby correct, and the ciphertext has been successfully decrypted using this formula.



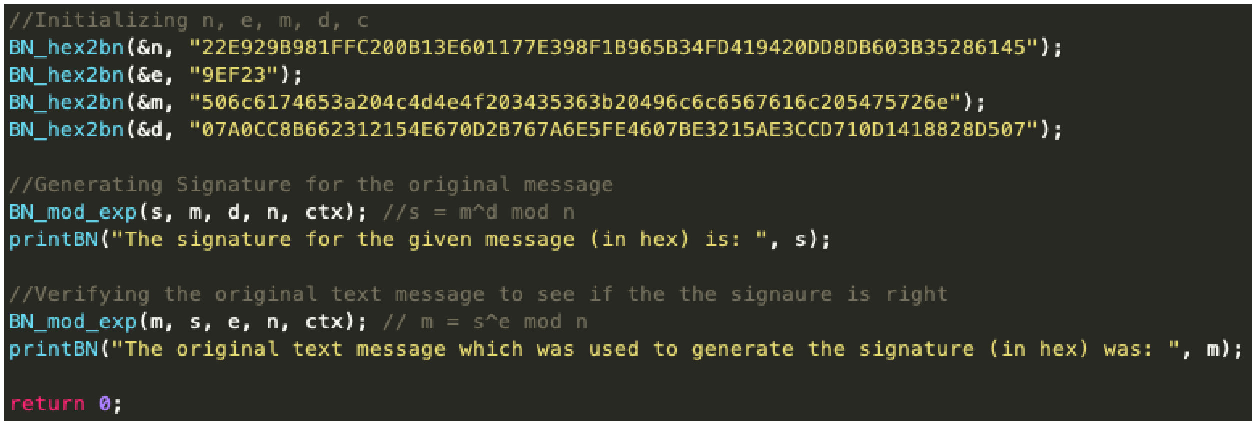
```
[10/11/20]seed@VM:~/Desktop$ ls
a1.zip   Problem 1          Problem 2 plate 2   problem4.c
a.out    Problem 2 plate 1  problem3.c
[10/11/20]seed@VM:~/Desktop$ gedit
[10/11/20]seed@VM:~/Desktop$ l
a1.zip   Problem 1/          Problem 2 plate 2/   problem4.c
a.out*   Problem 2 plate 1/  problem3.c           problem5.c
[10/11/20]seed@VM:~/Desktop$ ls
a1.zip   Problem 1          Problem 2 plate 2   problem4.c
a.out    Problem 2 plate 1  problem3.c          problem5.c
[10/11/20]seed@VM:~/Desktop$ gcc problem5.c -lcrypto
[10/11/20]seed@VM:~/Desktop$ ./a.out
The Original text which was sent (in hex) is:  506C6174653A2057585
95A203938373B20537065656564696E67
```

Figure 12: Resulting message value after decryption

## Problem 6:

This problem involved generating a signature for a given message, using the same public/private keys as problem 4.

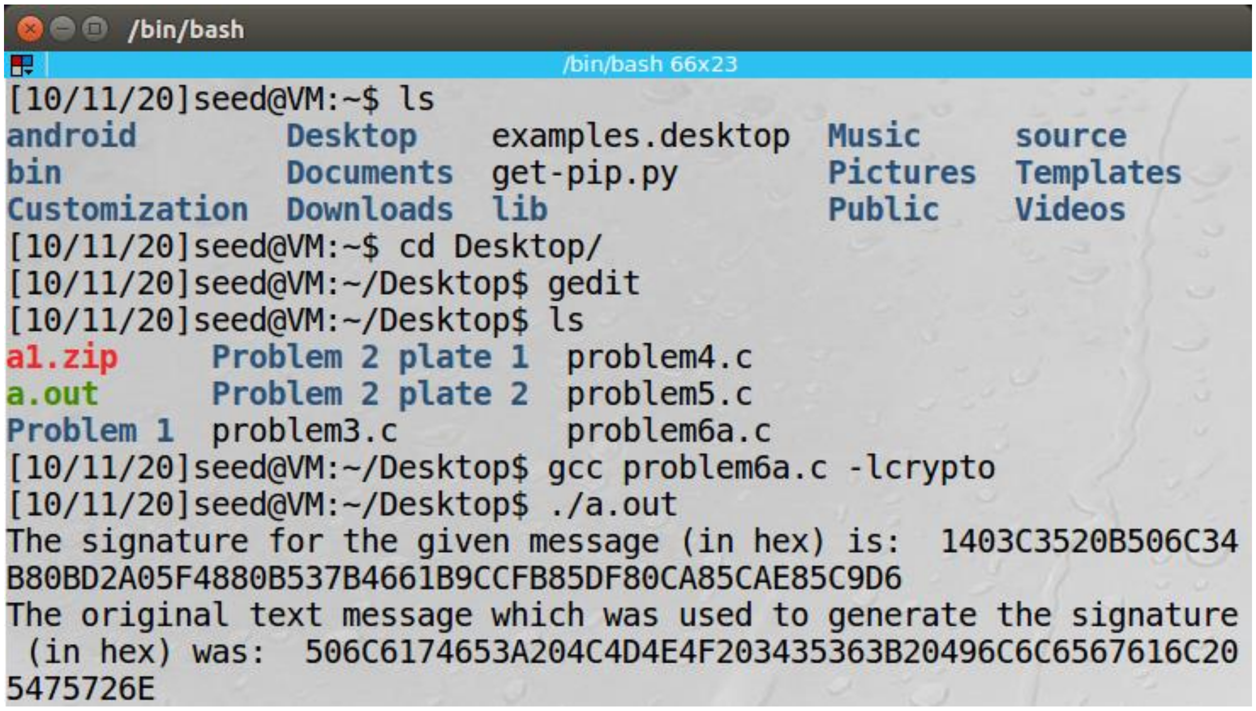
For part a), the given message was M = Plate: LMNO 456; Illegal Turn. This message was converted to hexadecimal (using a free online software). The question asked to directly sign this message, rather than signing its hash value. After initializing all necessary variables with the given values, the formula *res* = *a*^*c* mod *n* was used twice. Firstly, it was used to generate signature for the original message using the private key *d*. And secondly, it was used to check if the original text message was produced using that signature, as seen in figure 13 below.



```
//Initializing n, e, m, d, c
BN_hex2bn(&n, "22E929B981FFC200B13E601177E398F1B965B34FD419420DD8DB603B35286145");
BN_hex2bn(&e, "9EF23");
BN_hex2bn(&m, "506c6174653a204c4d4e4f203435363b20496c6c6567616c205475726e");
BN_hex2bn(&d, "07A0CC8B662312154E670D2B767A6E5FE4607BE3215AE3CCD710D1418828D507");

//Generating Signature for the original message
BN_mod_exp(s, m, d, n, ctx); //s = m^d mod n
printBN("The signature for the given message (in hex) is: ", s);

//Verifying the original text message to see if the the signaure is right
BN_mod_exp(m, s, e, n, ctx); // m = s^e mod n
printBN("The original text message which was used to generate the signature (in hex) was: ", m);

return 0;
```

Figure 13: Code for problem 6a

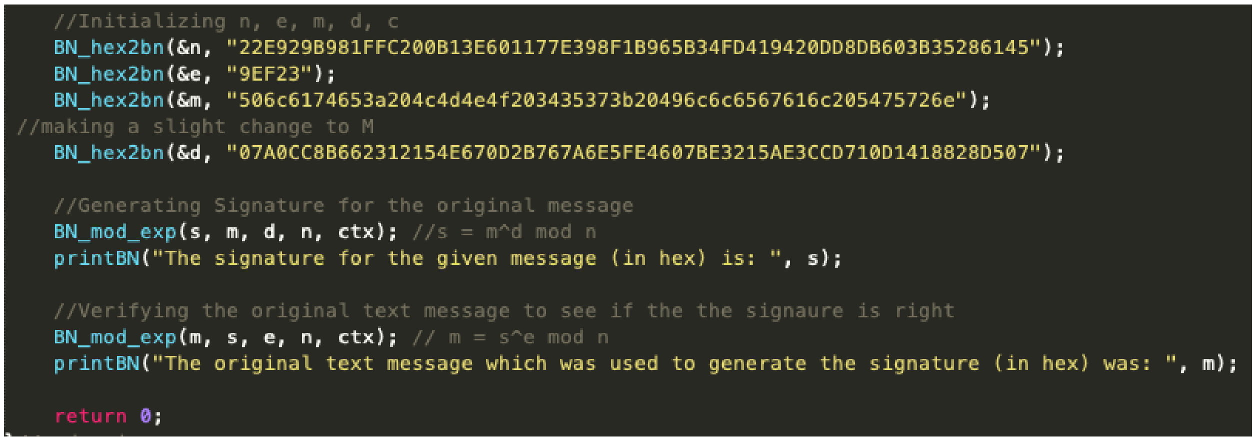
The final output of problem 6a can be seen in figure 14 below. As the original text message (in hexadecimal) matches the input value, which was set before doing calculations, the result is correct. Therefore, the problem helps successfully depict the signature of a message.



```
[10/11/20]seed@VM:~$ ls
android          Desktop     examples.desktop  Music      source
bin              Documents   get-pip.py        Pictures   Templates
Customization    Downloads   lib               Public     Videos
[10/11/20]seed@VM:~$ cd Desktop/
[10/11/20]seed@VM:~/Desktop$ gedit
[10/11/20]seed@VM:~/Desktop$ ls
a1.zip     Problem 2 plate 1  problem4.c
a.out      Problem 2 plate 2  problem5.c
Problem 1  problem3.c         problem6a.c
[10/11/20]seed@VM:~/Desktop$ gcc problem6a.c -lcrypto
[10/11/20]seed@VM:~/Desktop$ ./a.out
The signature for the given message (in hex) is:  1403C3520B506C34
B80BD2A05F4880B537B4661B9CCFB85DF80CA85CAE85C9D6
The original text message which was used to generate the signature
 (in hex) was:  506C6174653A204C4D4E4F203435363B20496C6C6567616C20
5475726E
```

Figure 14: Output of problem 6a

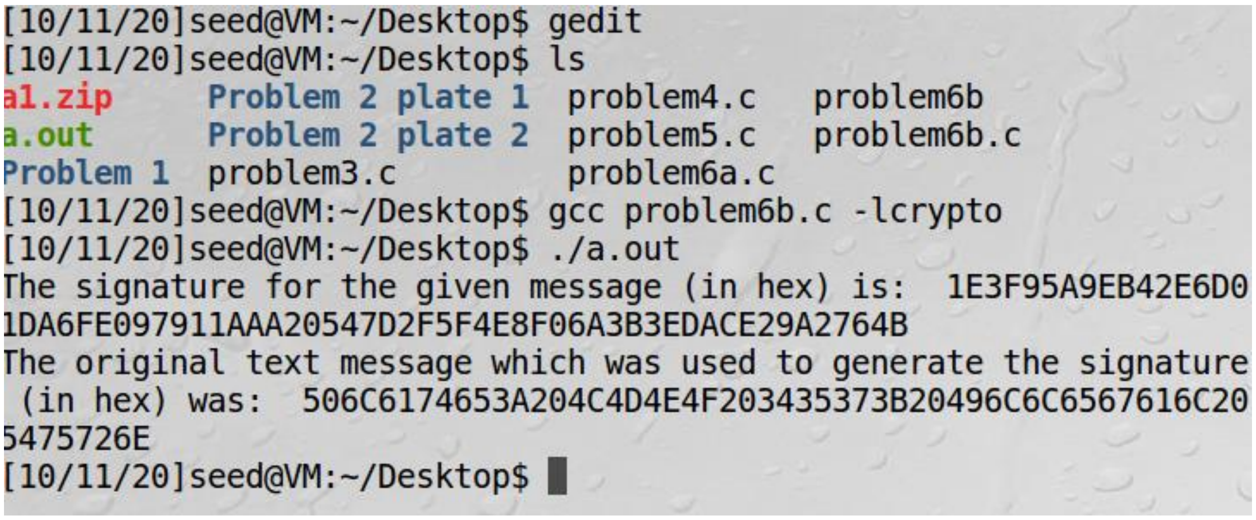
For part b), there was a slight change made to the message, which changed the LMNO 456 value to LMNO 457, changing the values from *34 35 3**__6__*** to *34 35 3**__7__**,* as shown in figure 15 below.



```
    //Initializing n, e, m, d, c
    BN_hex2bn(&n, "22E929B981FFC200B13E601177E398F1B965B34FD419420DD8DB603B35286145");
    BN_hex2bn(&e, "9EF23");
    BN_hex2bn(&m, "506c6174653a204c4d4e4f203435373b20496c6c6567616c205475726e");
 //making a slight change to M
    BN_hex2bn(&d, "07A0CC8B662312154E670D2B767A6E5FE4607BE3215AE3CCD710D1418828D507");

    //Generating Signature for the original message
    BN_mod_exp(s, m, d, n, ctx); //s = m^d mod n
    printBN("The signature for the given message (in hex) is: ", s);

    //Verifying the original text message to see if the the signaure is right
    BN_mod_exp(m, s, e, n, ctx); // m = s^e mod n
    printBN("The original text message which was used to generate the signature (in hex) was: ", m);

    return 0;
```

Figure 15: Code for problem 6b, with a slightly changed *m*

The output of problem 6b can be seen in the figure 16 below. This reveals the new signature found after the code was slightly changed. As the original text message matches the value, we used to initialize *m*, we can conclude that the signature found is correct.



```
[10/11/20]seed@VM:~/Desktop$ gedit
[10/11/20]seed@VM:~/Desktop$ ls
a1.zip      Problem 2 plate 1  problem4.c   problem6b
a.out       Problem 2 plate 2  problem5.c   problem6b.c
Problem 1   problem3.c         problem6a.c
[10/11/20]seed@VM:~/Desktop$ gcc problem6b.c -lcrypto
[10/11/20]seed@VM:~/Desktop$ ./a.out
The signature for the given message (in hex) is:  1E3F95A9EB42E6D0
1DA6FE097911AAA20547D2F5F4E8F06A3B3EDACE29A2764B
The original text message which was used to generate the signature
 (in hex) was:  506C6174653A204C4D4E4F203435373B20496C6C6567616C20
5475726E
[10/11/20]seed@VM:~/Desktop$ 
```

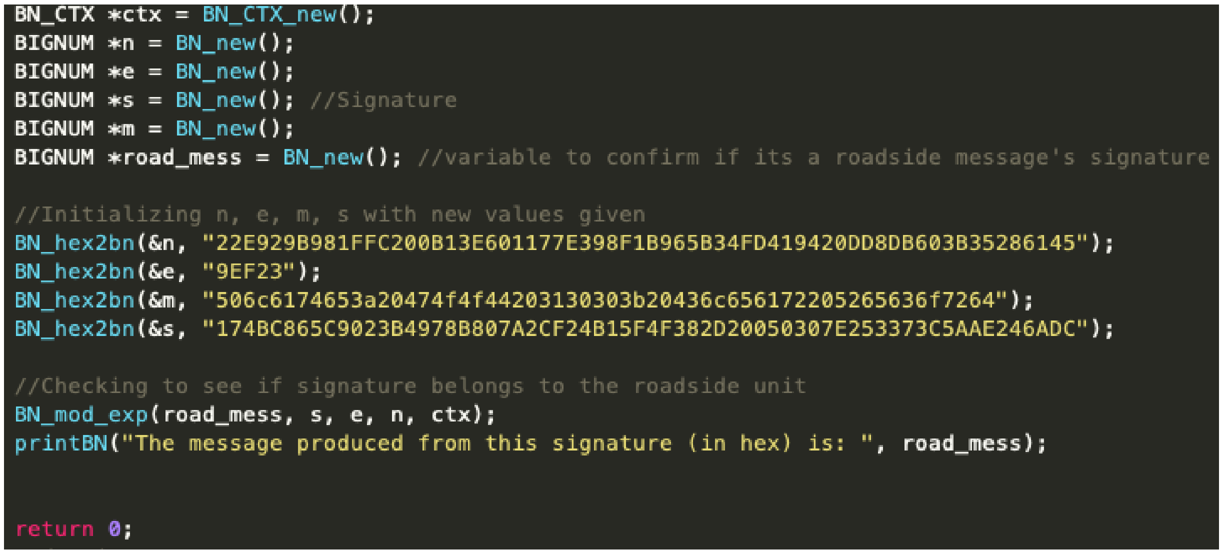
Figure 16: Output of problem 6b

To compare both signatures found in part a) and b), even a slight change in the message can cause a significant difference in both signatures. As observed from both figures, the resulting signatures and not identical at all, even though just two characters in the original message had been changed. The person must be careful with signing a message and check to see if the message they are signing is the one they are intending to sign.

## Problem 7:

This question helps verify a signature using roadside unit's public key (*e*,*n*).

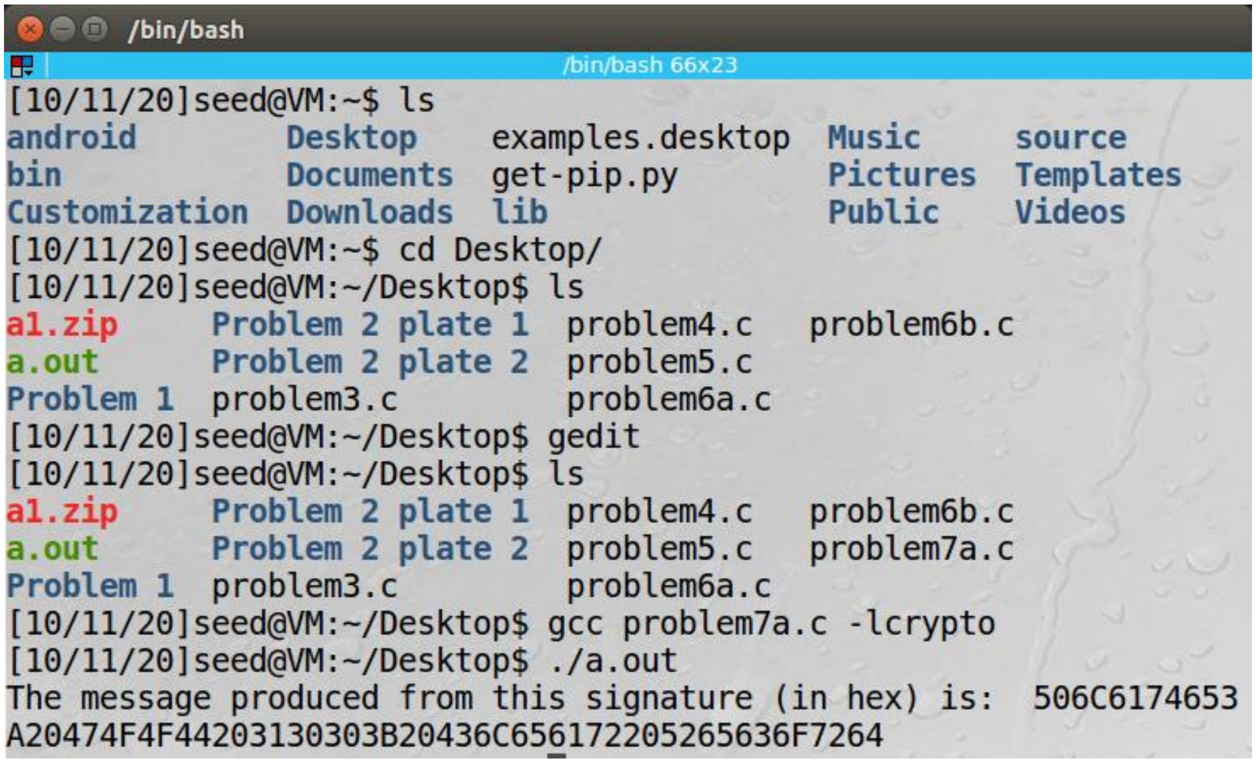
Part a) consisted of writing a program to verify if the signature belonged to the roadside unit. The figure below shows how the code was implemented for this problem. After assigning values to necessary variables, the formula *res* = *a*^*c* mod *n* was used with the variables s, e and n, to determine the roadside message.



```
BN_CTX *ctx = BN_CTX_new();
BIGNUM *n = BN_new();
BIGNUM *e = BN_new();
BIGNUM *s = BN_new(); //Signature
BIGNUM *m = BN_new();
BIGNUM *road_mess = BN_new(); //variable to confirm if its a roadside message's signature

//Initializing n, e, m, s with new values given
BN_hex2bn(&n, "22E929B981FFC200B13E601177E398F1B965B34FD419420DD8DB603B35286145");
BN_hex2bn(&e, "9EF23");
BN_hex2bn(&m, "506c6174653a20474f4f44203130303b20436c656172205265636f7264");
BN_hex2bn(&s, "174BC865C9023B4978B807A2CF24B15F4F382D20050307E253373C5AAE246ADC");

//Checking to see if signature belongs to the roadside unit
BN_mod_exp(road_mess, s, e, n, ctx);
printBN("The message produced from this signature (in hex) is: ", road_mess);


return 0;
```

Figure 17: Code for problem 7a

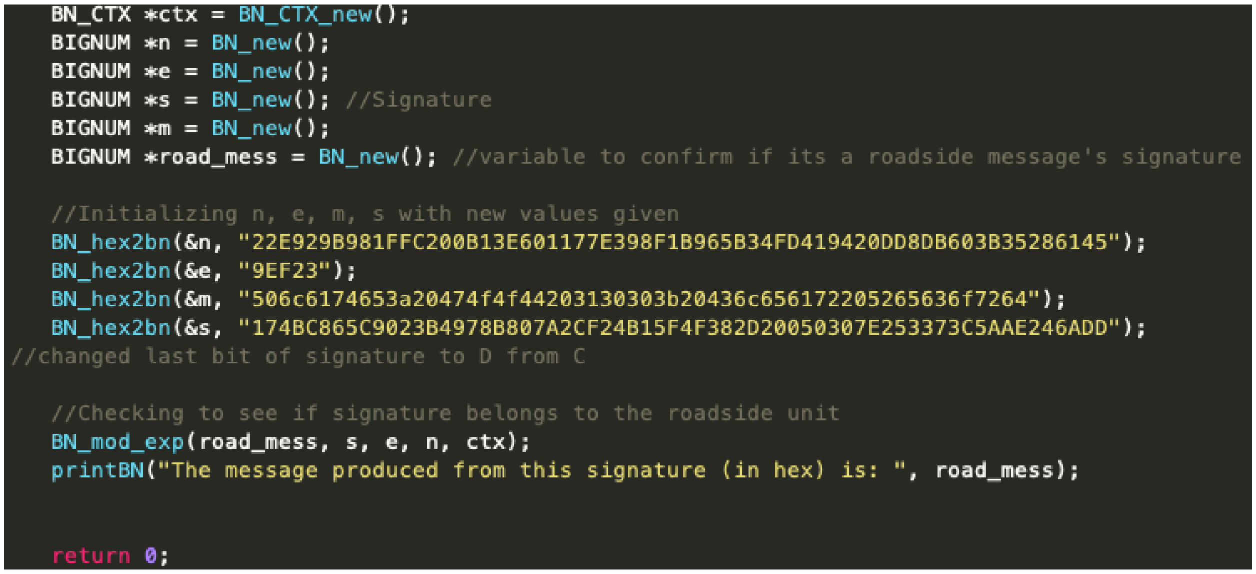
The result of this code is shows in figure 18 below. As expected, the message produced from the signature matched the original message *m* value, therefore verifying the signature.



```
[10/11/20]seed@VM:~$ ls
android        Desktop    examples.desktop  Music     source
bin            Documents  get-pip.py        Pictures  Templates
Customization  Downloads  lib               Public    Videos
[10/11/20]seed@VM:~$ cd Desktop/
[10/11/20]seed@VM:~/Desktop$ ls
a1.zip     Problem 2 plate 1  problem4.c   problem6b.c
a.out      Problem 2 plate 2  problem5.c
Problem 1  problem3.c         problem6a.c
[10/11/20]seed@VM:~/Desktop$ gedit
[10/11/20]seed@VM:~/Desktop$ ls
a1.zip     Problem 2 plate 1  problem4.c   problem6b.c
a.out      Problem 2 plate 2  problem5.c   problem7a.c
Problem 1  problem3.c         problem6a.c
[10/11/20]seed@VM:~/Desktop$ gcc problem7a.c -lcrypto
[10/11/20]seed@VM:~/Desktop$ ./a.out
The message produced from this signature (in hex) is:  506C6174653
A20474F4F44203130303B20436C656172205265636F7264
```

Figure 18: output of problem 7a

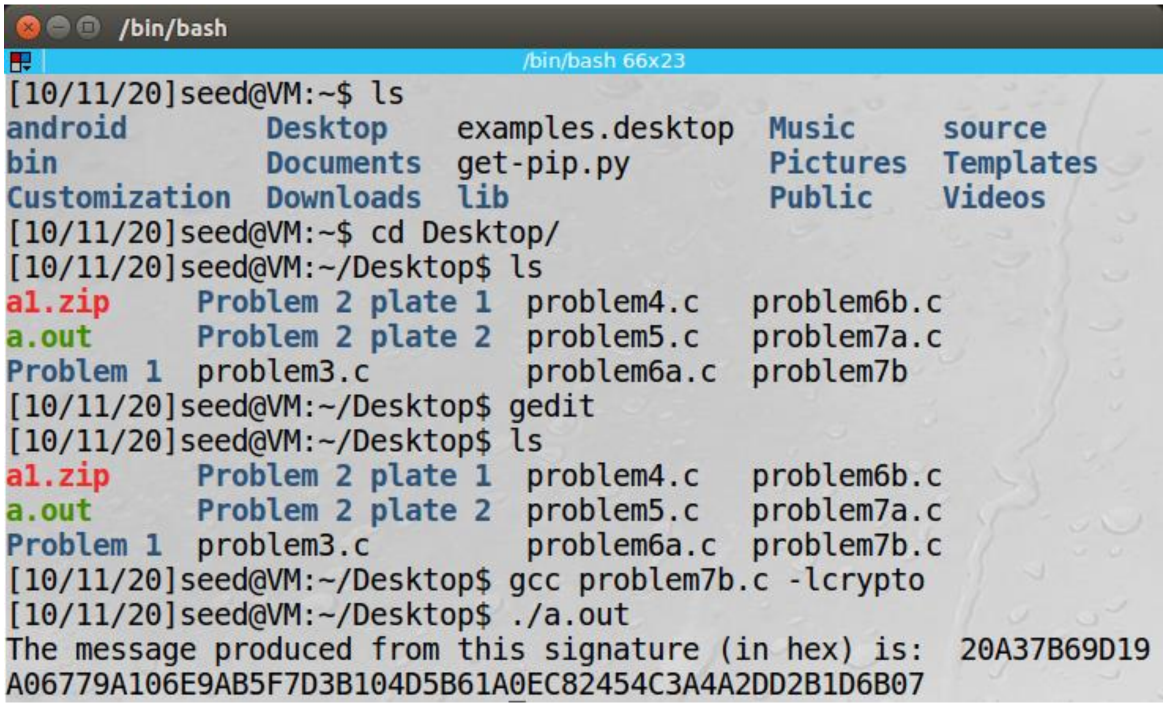
In 7b), the last byte of the signature was changed from DC to DD, resulting in a value change, as seen in figure below. The aim of this problem was to check if the verification process still worked, and to compare the new result.



```
    BN_CTX *ctx = BN_CTX_new();
    BIGNUM *n = BN_new();
    BIGNUM *e = BN_new();
    BIGNUM *s = BN_new(); //Signature
    BIGNUM *m = BN_new();
    BIGNUM *road_mess = BN_new(); //variable to confirm if its a roadside message's signature

    //Initializing n, e, m, s with new values given
    BN_hex2bn(&n, "22E929B981FFC200B13E601177E398F1B965B34FD419420DD8DB603B35286145");
    BN_hex2bn(&e, "9EF23");
    BN_hex2bn(&m, "506c6174653a20474f4f44203130303b20436c656172205265636f7264");
    BN_hex2bn(&s, "174BC865C9023B4978B807A2CF24B15F4F382D20050307E253373C5AAE246ADD");
//changed last bit of signature to D from C

    //Checking to see if signature belongs to the roadside unit
    BN_mod_exp(road_mess, s, e, n, ctx);
    printBN("The message produced from this signature (in hex) is: ", road_mess);


    return 0;
```

Figure 19: Code for problem 7b, with a changed signature value

The output of problem 7b, as shown in figure 20, reveals that changing even a small change of one byte in the signature can produce a completely different message. The result found in problem 7b did not match the original sent message. As the signature verification failed, it is evident that either the message or the signature had been corrupted (signature in this case). The sender must be careful with the message they want to deliver, as well as the signature. If the signature is corrupted, the message will be modified, resulting in a faulty communication.



```
[10/11/20]seed@VM:~$ ls
android          Desktop    examples.desktop  Music     source
bin              Documents  get-pip.py        Pictures  Templates
Customization    Downloads  lib               Public    Videos
[10/11/20]seed@VM:~$ cd Desktop/
[10/11/20]seed@VM:~/Desktop$ ls
a1.zip      Problem 2 plate 1   problem4.c   problem6b.c
a.out       Problem 2 plate 2   problem5.c   problem7a.c
Problem 1   problem3.c          problem6a.c  problem7b
[10/11/20]seed@VM:~/Desktop$ gedit
[10/11/20]seed@VM:~/Desktop$ ls
a1.zip      Problem 2 plate 1   problem4.c   problem6b.c
a.out       Problem 2 plate 2   problem5.c   problem7a.c
Problem 1   problem3.c          problem6a.c  problem7b.c
[10/11/20]seed@VM:~/Desktop$ gcc problem7b.c -lcrypto
[10/11/20]seed@VM:~/Desktop$ ./a.out
The message produced from this signature (in hex) is:  20A37B69D19
A06779A106E9AB5F7D3B104D5B61A0EC82454C3A4A2DD2B1D6B07
```

Figure 20: Output of problem 7 b), with a corrupted signature value

## Summary of Findings

### Problem 8:

1) Based on the experiment done in Problem 2, AES is the best choice of encryption for license plates. Based on results from problem 2 with license plate #1, aes-128-cbc should be used to encrypt the license plates. 128 bits of key size is preferred as it does not hinder the performance of a busy system and is secure for the means of encrypting license plates. AES will not only help improve the system's performance but will provide security from various attacks to reveal the images. As the goal of AES is to provide non-linearity between blocks, there will be less correlation among neighboring blocks, which will eventually make it harder for any possible attack to occur. Lastly, AES system relies on random inputs and an initializing vector, which helps add more confidentiality to the system, additionally helping to avoid several kinds of attacks.

2) Digital signatures are a way to provide authentication to users and for the purpose of non-repudiation. Because the main purpose of RoadRunner Systems is to ensure that only a guilty vehicle owner is issued a fine, and the messages sent from the roadside units have not been altered, digital signatures is the best option for the company. Firstly, they would require digital signature with hashing, and keep the private key a secret to help avoid any message alteration. Since digital signatures use a private key to encrypt a message, when the message is decrypted, it is clear to know who the author is. Secondly, if a message is altered by chance, the entire signature will change, stating that information has been altered.

However, a drawback of using digital signatures is that it does not guarantee confidentiality. Due to the usage of public keys to decrypt a digital signature, there is no certainty that no one else can read that message.