Assignment 2

SYSC 4810

Fall 2020

Valerie Figuracion

100965536

October 11, 2020

Contents

[Table of Figures 3](#_Toc53351532)

[**Symmetric Cryptography** 4](#_Toc53351533)

[Symmetric Encryption using Different Ciphers and Modes 4](#_Toc53351534)

[Different Ciphers, Same Mode (CBC) 4](#_Toc53351535)

[Same Cipher, Different Modes (AES) 5](#_Toc53351536)

[Same Cipher, Same Mode, Different Key Sizes (AES CBC) 5](#_Toc53351537)

[Encryption Mode: ECB vs CBC 6](#_Toc53351538)

[Encryption Method 6](#_Toc53351539)

[Result 6](#_Toc53351540)

[**Asymmetric Cryptography** 7](#_Toc53351541)

[Deriving the Private Key 7](#_Toc53351542)

[Encrypting a Message 8](#_Toc53351543)

[Decrypting a Message 9](#_Toc53351544)

[Signing a Message 10](#_Toc53351545)

[Verifying a Signature 10](#_Toc53351546)

[**Summary of Findings** 12](#_Toc53351547)

[Recommendations 12](#_Toc53351548)

[Cryptographic Solution 12](#_Toc53351549)

[Message Authentication 12](#_Toc53351550)

[**Appendix** 13](#_Toc53351551)

# Table of Figures

[Figure 1: Contents of the files using the command-line hex viewing tool xxd. The files are encrypted with different ciphers but using the same mode. 4](#_Toc53351552)

[Figure 2: Contents of the files using the command-line hex viewing tool xxd. The files are encrypted with the same cipher but using different modes. 5](#_Toc53351553)

[Figure 3: Contents of the files using the command-line hex viewing tool xxd. The files are encrypted using the same cipher and mode but have a different key size. 5](#_Toc53351554)

[Figure 4: CBC encrypted (top left) and ECB encrypted (top right) images of plate1.bmp (bottom). 6](#_Toc53351555)

[Figure 5: CBC encrypted (top left) and ECB encrypted (top right) images of plate2.bmp (bottom). 7](#_Toc53351556)

[Figure 6: Code snippet to derive d from Problem3.c 7](#_Toc53351557)

[Figure 7: Terminal screenshot from Problem3.c. 8](#_Toc53351558)

[Figure 8: Code snippet of the encryption function. 8](#_Toc53351559)

[Figure 9: Terminal screenshot from Problem4.c 8](#_Toc53351560)

[Figure 10: Code snippet of the check() function from Problem4.c. 9](#_Toc53351561)

[Figure 11: Code snippet of the decryption function and conversion into ASCII string. 9](#_Toc53351562)

[Figure 12: Terminal screenshot from Problem5.c 9](#_Toc53351563)

[Figure 13: Code snippet of the signature generation function. 10](#_Toc53351564)

[Figure 14: Terminal screenshot from Problem6a.c. 10](#_Toc53351565)

[Figure 15: Terminal screenshot from Problem6b.c. 10](#_Toc53351566)

[Figure 16: Code snippet of the signature verification function. 11](#_Toc53351567)

[Figure 17: Terminal screenshot from Problem7a.c. 11](#_Toc53351568)

[Figure 18: Terminal screenshot from Problem7b.c. 11](#_Toc53351569)

[Figure 19: Function in Problem3.c to check if decrypting an encrypted message will restore the original message. 13](#_Toc53351570)

[Figure 20: Code snippet to convert a string to a hexadecimal string. 13](#_Toc53351571)

[Figure 21: hex\_to\_int helper function and hex\_to\_ascii function that converts a hex string into an ASCII string. 14](#_Toc53351572)

[Figure 22: Check function from Problem5.c to verify that the decryption was successful. 14](#_Toc53351573)

[Figure 23: Verify function from Problem7a.c to verify that the message came from a valid roadside unit. 14](#_Toc53351574)

# **Symmetric Cryptography**

## Symmetric Encryption using Different Ciphers and Modes

The input text file created has the contents “This is my secret. SHHHH. Don’t tell anyone.” and has the name “P1.txt”.

The file was encrypted using three different cipher methods Data Encryption Standard (DES), Triple-DES (3DES), and Advanced Encryption Standard (AES), three different modes, Cipher Block Chaining (CBC), Electronic Code Book (ECB), and Cipher Feedback (CFB), and three different key sizes, 128-bit, 192-bit, 256-bit.

The encryption of the files was done using the openssl enc command. For example, encrypting the file using Advanced Encryption Standard (AES) with a key size of 128 bits using Cipher Block Chaining (CBC) the following command is used.

|  |
| --- |
| $ openssl enc -aes-128-cbc -in P1.txt -out P1cbc.txt -K 0123456789ABCDEF -iv ABCDEF |

Commands for other ciphers, other modes and other key sizes can be found in Problem1b.txt.

### Different Ciphers, Same Mode (CBC)

DES and 3DES provide the same ciphertext. The functionality of 3DES is not properly used since for each encrypt-decrypt-encrypt sequence different keys should be used for each stage however in this scenario there is only one key so 3DES functions as regular DES. Since AES has no relation to DES or 3DES then the ciphertext should be different from the other two which is evident in the figure below.

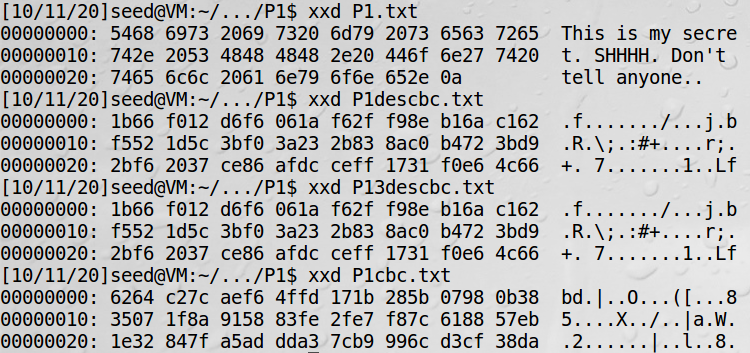


Figure : Contents of the files using the command-line hex viewing tool xxd. The files are encrypted with different ciphers but using the same mode.

### Same Cipher, Different Modes (AES)

Each mode provided a different output since the way that they encrypt is different from each other regardless of the ciphertype. CBC and CFB used the initialization vector while ECB did not. It was also expected that the ciphertext would be different between CBC and CFB despite using the same key and initialization vector since CFB encrypts one bit at a time while CBC encrypts one block at a time.

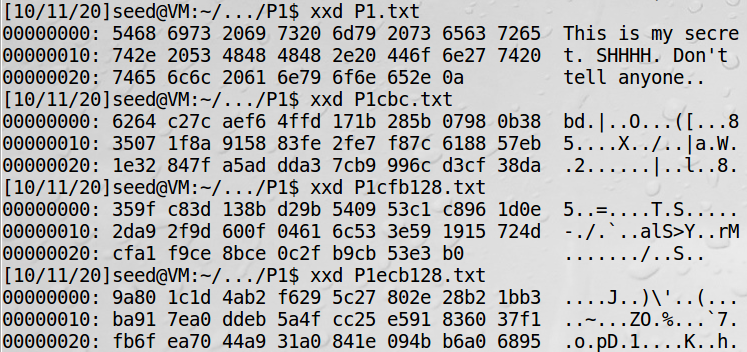


Figure : Contents of the files using the command-line hex viewing tool xxd. The files are encrypted with the same cipher but using different modes.

### Same Cipher, Same Mode, Different Key Sizes (AES CBC)

The different key sizes provided different ciphertexts despite using the same cipher and mode. The total number of keys are increased as each key size increases which allows expanded key to have different permutations for each round.

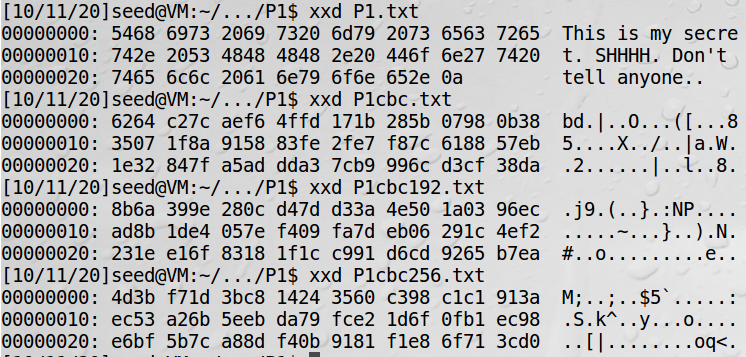


Figure : Contents of the files using the command-line hex viewing tool xxd. The files are encrypted using the same cipher and mode but have a different key size.

## Encryption Mode: ECB vs CBC

### Encryption Method

The encryption of plate1.bmp file was done using AES-128-bit cipher with two modes ECB and CBC. The commands for each method are shown below respectively.

|  |
| --- |
| $ openssl enc -aes-128-ecb -in plate1.bmp -out p1ecb.bmp -K 0123456789  $ openssl enc -aes-128-cbc -in plate1.bmp -out p1cbc.bmp -K 0123456789 -iv ABCDEF |

Each encrypted file must have their header replaced with the header of the original file so that it can be treated as an image and viewed by an image reading software. The command for doing so is below for ECB and CBC respectively.

|  |
| --- |
| $ head -c 54 plate1.bmp > header  $ tail -c +55 p1ecb.bmp > body  $ cat header body > newp1ecb.bmp  $ head -c 54 plate1.bmp > header  $ tail -c +55 p1cbc.bmp > body  $ cat header body > newp1cbc.bmp |

The encryption of plate2.bmp used the same method as encrypting plate1.bmp. The commands for these are found in Problem2.txt

### Result

The original image for plate1.bmp is shown with the encrypted images that used ECB and CBC. The ECB mode did not distort the image as much as the CBC mode. This is because ECB is prone to producing the identical ciphertexts when fed the same plaintext. CBC prevents this by using an initialization vector which scrambles up the plaintext before encryption so that the produced ciphertext is not identical when provided the same plaintext. The CBC encrypted image shows this as there is no resemblance of the numbers and letters that are on the original image.



Figure : CBC encrypted (top left) and ECB encrypted (top right) images of plate1.bmp (bottom).

The same set up as the previous figure is shown for plate2.bmp. Both CBC and ECB encrypted messages are distorted enough that the information in the original image is not decipherable. For ECB, since plate2.bmp had more colour than plate1.bmp it was less likely for the plaintext blocks to be similar to one another therefore it was much less likely to have produced identical ciphertext blocks despite using the same encryption method. Although, there are some information that can be taken away from the ECB encrypted image like how the licence plate has a border since the edges of the image are not as distorted as the rest of the image. The same thing can also be said for the letters and numbers that have straight components such as in “B”, “C”, and “1” in the middle of the image. These issues are taken care of when using CBC mode for the same reason as in the plate1.bmp case where distorting the block before encryption adds another level of security.

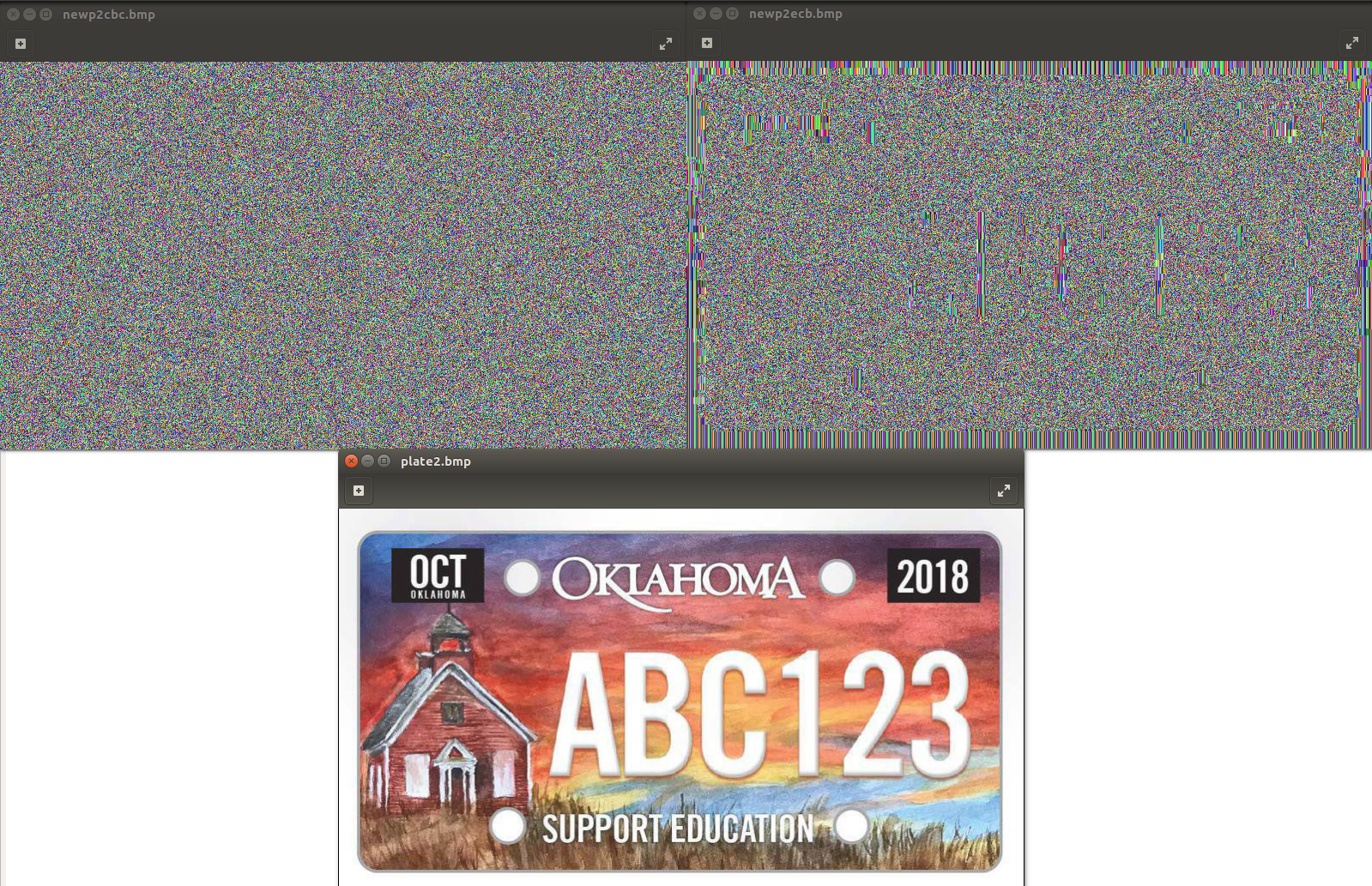


Figure : CBC encrypted (top left) and ECB encrypted (top right) images of plate2.bmp (bottom).

# **Asymmetric Cryptography**

## Deriving the Private Key

The program used to derive the private key is “Problem3.c”. Below is a snippet of how d in the private key is found. Since prime numbers p and q are given it is easy to find the totient of n by multiplying (p-1) and (q-1). Using the totient and e from the public key d is found using the multiplicative inverse.

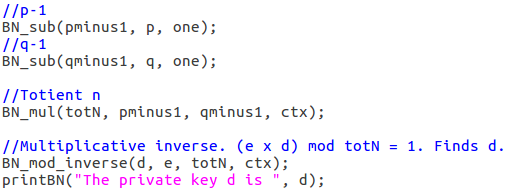


Figure : Code snippet to derive d from Problem3.c

Compiling the code requires the following command that uses -lcrypto since the BIGNUM library is used. BIGNUM is used since it makes it easier to compute integers of any size.

|  |
| --- |
| $ gcc Problem3.c -lcrypto -o p3 |

The private key d is 02BC85A25966C61A41AB80BF3614905F129A6BF1DE2170A8DE4F44C90449ADF5 shown in the screenshot below using Problem3.c.

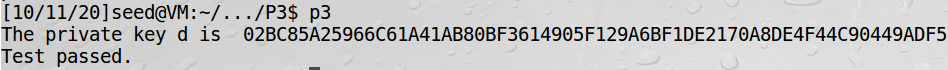


Figure : Terminal screenshot from Problem3.c.

There is also a function within the program that feeds a random message, encrypts it using the public keys and then decrypts it using the private keys to compare if the decrypted ciphertext is equal to the original plain text. A “Test passed” result shows that the private key d is indeed the correct one. A snippet of the code is figure 19 that is shown in the appendix.

The private key d simple to find since the prime numbers given are small. Typically, the p and q should be larger as it is much more difficult to calculate d using brute force or mathematical attacks. With the increase of computing power, it is easier to find the necessary prime numbers from a given shared number n, which is computed as p x q, if n is small.

## Encrypting a Message

The program used to encrypt the message is “Problem4.c” The function string2hexString() converts an ASCII string into a hexadecimal string. The code snippet is provided as figure 20 in the appendix. The hexadecimal string message was then converted into BIGNUM using the hex-to-bn function. The message was then encrypted using the public keys (e, n) given and using the encryption function which is shown in the BN\_mod\_exp() function.



Figure : Code snippet of the encryption function.

Compiling the code uses the following command that uses -lcrypto since the BIGNUM library is used again.

|  |
| --- |
| $ gcc Problem4.c -lcrypto -o p4 |

The encrypted message is 0D24F7257C5BFC302B75494C312E74511CE4D0D802C729D23D68FBFA1489778A as shown below.

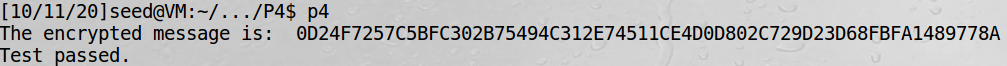


Figure : Terminal screenshot from Problem4.c

To verify that the message was properly encrypted the check() function takes the ciphertext C and decrypts it with the private keys (d, n) given and the decryption function . The decrypted message is compared to the original message M and will print “Test passed” if both messages are identical which is the case in the screenshot above.

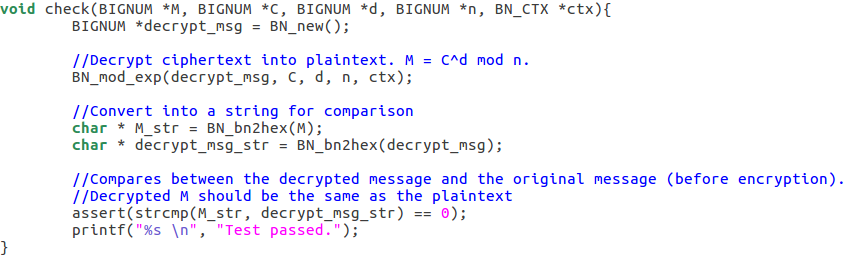


Figure : Code snippet of the check() function from Problem4.c.

## Decrypting a Message

The program used to decrypt the message is “Problem5.c” The ciphertext given was decrypted using the same private keys (d, n) that was in Encrypting a Message and the decryption function . The result was then converted into an ASCII string by first converting it into a hexadecimal string from a BIGNUM using BN\_bn2hex() and then using hex\_to\_ascii(). The hex\_to\_ascii() function code snippet is provided as figure 21 in the appendix.

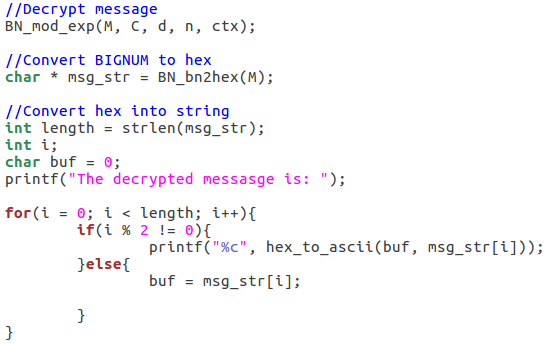


Figure : Code snippet of the decryption function and conversion into ASCII string.

Compiling the code uses the following command that uses -lcrypto since the BIGNUM library is used again.

|  |
| --- |
| $ gcc Problem5.c -lcrypto -o p5 |

The decrypted message is “Plate: WXYZ 987; Speeding” as shown below.

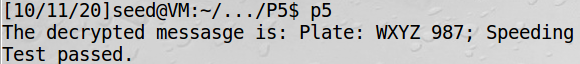


Figure : Terminal screenshot from Problem5.c

The check() function within the Problem5.c program verifies that the decryption was successful by encrypting the decrypted message using the pubic keys (e, n) from Encrypting a Message and comparing the encrypted message with the original ciphertext. A “Test passed” result shows that the encrypted message and the ciphertext are the same as shown in the screenshot above. A code snippet of the check() function is provided as figure 22 in the appendix.

## Signing a Message

The program used to sign the message is “Problem6a.c”. The process and code for signing the message is the same as in Encrypting a Message except the keys used for the signing are the private keys (d, n) instead of the public keys (e, n). The equation to generate the signature is also reflected as such as as shown in the code snippet below.



Figure : Code snippet of the signature generation function.

Compiling the code uses the following command that uses -lcrypto since the BIGNUM library is used again.

|  |
| --- |
| $ gcc Problem6a.c -lcrypto -o p6a |

The signature is 1403C3520B506C34B80BD2A05F4880B537B4661B9CCFB85DF80CA85CAE85C9D6 as shown below.



Figure : Terminal screenshot from Problem6a.c.

The check() function within Problem6a.c verifies the signature by using the public keys to compute the message and then compare that to the original plaintext given. A “Test passed” result shows that the signature is valid which is the case above. A code snippet of the check() function is provided as figure 10 above.

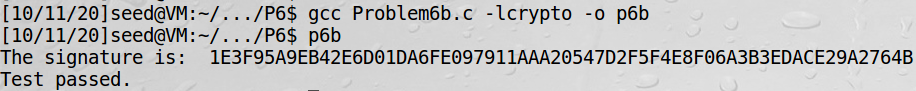
A slight change in the message, in this case changing “Plate: LMNO 456; Illegal Turn” to “Plate: LMNO 457; Illegal Turn”, provides a different signature. The program used for this is “Problem6b.c”. Compilation and running of the program are as follows. 

Figure : Terminal screenshot from Problem6b.c.

The signature generated is now completely different from before despite one small change in the message.

## Verifying a Signature

The program used to verify the signature is “Problem7a.c”. The process and code for verifying the signature is the same as that in Decrypting the Message except the keys used for verification are the public keys (e, n) rather than the private keys (d, n). The equation to generate the signature is also reflected as such as as shown in the code snippet below.



Figure : Code snippet of the signature verification function.

Compiling the code uses the following command that uses -lcrypto since the BIGNUM library is used again.

|  |
| --- |
| $ gcc Problem7a.c -lcrypto -o p7a |

The message decrypted from the provided signature was “Plate: GOOD 100; Clear Record” which was to be expected.

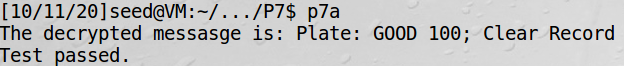


Figure : Terminal screenshot from Problem7a.c.

A verify function in the Problem7a.c program ensures that the message received from the roadside unit has not been tampered with by comparing the M that was computed using the provided signature and the expected message. A “Test passed” result shows that the message has not been tampered. The verify() function is provided in the appendix as figure 23.

Program7b.c is an example of what would happen if the message has been tampered or corrupted. The last byte of the signature has been altered. Compilation and running of the program are as follows.

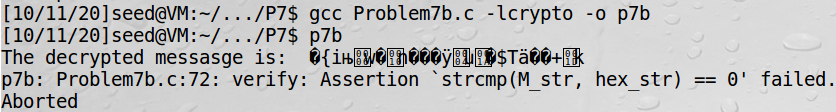


Figure : Terminal screenshot from Problem7b.c.

The decrypted message is now unrecognizable and very different from the message that was to be expected. The verify() function, which is the same as in Program7a.c, shows that the expected message and the decrypted message are not identical and does not provide a “Test passed” result.

# **Summary of Findings**

## Recommendations

### Cryptographic Solution

The best option for maintaining confidentiality is using the Advanced Encryption Standard (AES) cipher using either Cipher Block Chaining (CBC) or Cipher Feedback (CFB). As shown in the Encryption Mode: ECB vs CBC it is beneficial to have a system that uses an initialization vector (IV) as an extra step to ensure that no patters are recognizable. Both CBC and CFB uses an IV. AES is more secure than Data Encryption Standard (DES) or Triple-DES (3DES) since it uses larger key sizes when one key is used. The smallest key size in AES is large enough that successful decryption will take many years to decode therefore either 128, 192, or 256 bit will suffice.

### Message Authentication

A digital signature is a valid choice to ensure that the message has not been altered. If it was tampered or corrupted the signature, when processed through the signature verification function, will not provide the message as expected since the key that the message has been signed with in the first place is only known by one entity. Any alteration with the message would require a re-sign and if it was from a third party then that third party must sign it with their own private key as the keys used for generating the signature are not public. Digital signatures use asymmetric cryptography which is much slower than symmetric cryptography. If performance is an issue and numerous images and messages must be sent and received another alternative could be a better solution especially when the messages become large.

A digital envelope is a valid alternative as it has both aspects of asymmetric and symmetric encryption. Asymmetric encryption is more secure than symmetric encryption as it requires both public and private keys rather than one symmetric key however it does take longer to encrypt and decrypt. For a digital envelope, a symmetric key will be used to encrypt the message while the asymmetric keys will be used to encrypt the symmetric key. Encryption and decryption are faster since the encrypted symmetric key can be much shorter than the message it encrypts even if the message is long. It also keeps the confidentiality of the message even if the message itself is intercepted it cannot be read without the symmetric key that only the receiver’s private key can decrypt.

# **Appendix**

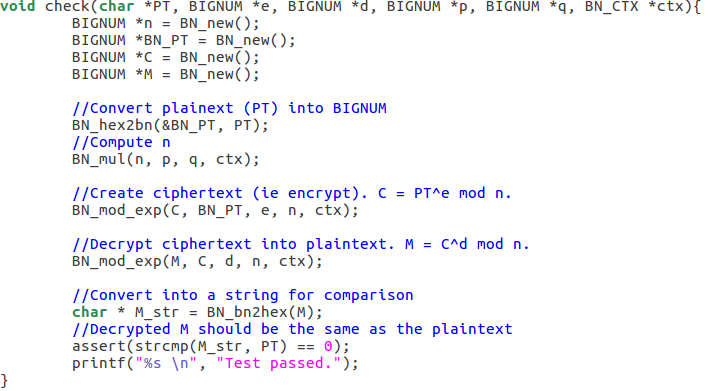


Figure : Function in Problem3.c to check if decrypting an encrypted message will restore the original message.

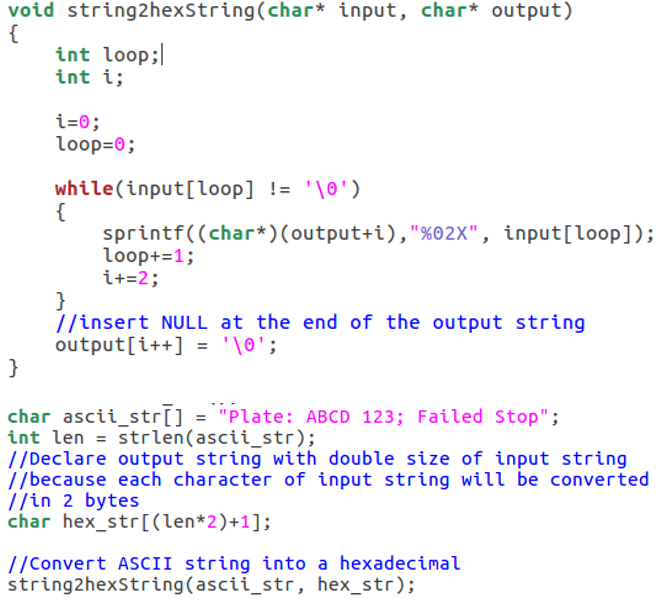


Figure : Code snippet to convert a string to a hexadecimal string.

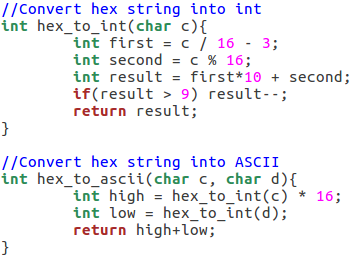


Figure : hex\_to\_int helper function and hex\_to\_ascii function that converts a hex string into an ASCII string.

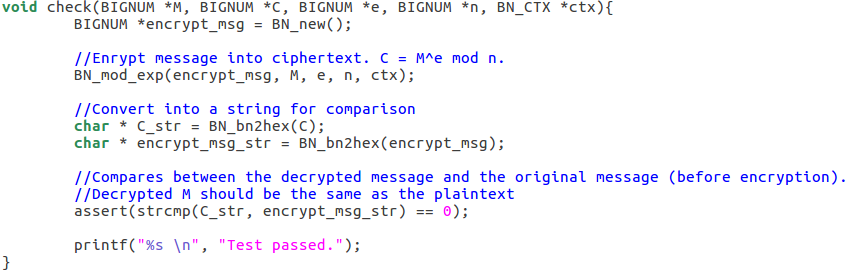


Figure : Check function from Problem5.c to verify that the decryption was successful.

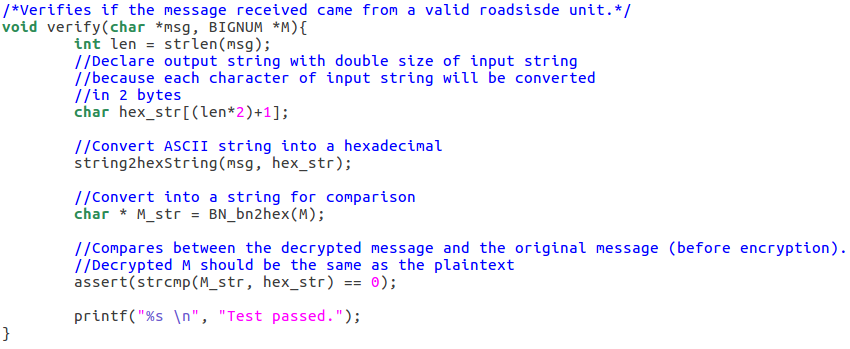


Figure : Verify function from Problem7a.c to verify that the message came from a valid roadside unit.