COP514 Coursework Report

Table of Contents

1 - Abstract

This report will cover the detail of the project undertaken by our group, how we achieved the coursework specification, the implementation of the code and descriptions of our implementation of the modes of operations. It will also feature descriptions of the user interface and how the user interacts with the program, furthermore, it will test the speed of encryption and decryption for each mode of operation with a large file, contrast the results and give potential explanations for each time measurement. Towards the end of the report, an analysis of the security aspect of the project will also be produced, which will question some aspects of our implementation, some potential security flaws and how security may be improved.

2 - Description of the program

2.1 – Encryption and decryption algorithms

The encryption algorithm is partially similar to AES algorithm where we have substitution of AES s-box, permutation between bits and XOR operation between block and keys. The decryption algorithm is the inverse of encryption algorithm where all process is reversed. In this section, we will explain the details of the encryption process and briefly about decryption process.

The algorithm received two inputs: a list of 8 ASCII decimals where each decimal represents a byte or 8 bits and a list of keys (each key is a list of 8 ASCII decimals). As we have 8 decimals, we essentially have the information of 8x8 bits. We will refer to this as state. The list of keys is a list of 6 lists where each list contains 8 ASCII decimals which represent the round key for each round of encryption. We will refer to this as key list.

There are 5 rounds in this encryption. At the end of each rounds, we XOR between the state and a different round key in the key list. To use a different round key, we iterate the key list in an ascending order (i.e. round 0 = key 0, round 1 = key 1 etc). Before we start the first round, we XOR between the state and the first round key in the key list (key 0). We iterate each byte in the state and perform the S-box substitution and produce a state with substituted values.

Next, we perform a column-wise permutation. Before the permutation, we first convert each decimal in the state to byte to produce a list of bytes. Imagine the byte is in an 8x8 table where each cell contains a single bit and each row represent a byte. For this permutation, each column of the 8x8 table is shifted downward circularly by 0,1,2,..7.

Table 1

B_{00}	B_{01}	B_{02}	B_{03}	B_{04}	B_{05}	B_{06}	B_{07}
B_{10}	B_{11}	B_{12}	B_{13}	B_{14}	B_{15}	B_{16}	B_{17}
B_{20}	B_{21}	B_{22}	B_{23}	B_{24}	B_{25}	B_{26}	B_{27}
B_{30}	B_{31}	B_{32}	B_{33}	B_{34}	B_{35}	B_{36}	B_{37}
B_{40}	B_{41}	B_{42}	B_{43}	B_{44}	B_{45}	B_{46}	B_{47}
B_{50}	B_{51}	B_{52}	B_{53}	B_{54}	B_{55}	B_{56}	B_{57}
B_{60}	B_{61}	B_{62}	B_{63}	B_{64}	B_{65}	B_{66}	B_{67}
B_{70}	B_{71}	B_{72}	B_{73}	B_{74}	B_{75}	B_{76}	B_{77}

B_{00}	B_{71}	B_{62}	B_{53}	B_{44}	B_{35}	B_{26}	B_{17}
B_{10}	B_{01}	B_{12}	B_{23}	B_{34}	B_{45}	B_{56}	B_{67}
B_{20}	B_{11}	B_{02}	B_{73}	B_{64}	B_{55}	B_{46}	B_{37}
B_{30}	B_{21}	B_{12}	B_{03}	B_{74}	B_{65}	B_{56}	B_{47}
B_{40}	B_{31}	B_{22}	B_{13}	B_{04}	B_{75}	B_{66}	B_{57}
B_{50}	B_{41}	B_{32}	B_{23}	B_{14}	B_{05}	B_{76}	B_{67}
B_{60}	B_{51}	B_{42}	B_{33}	B_{24}	B_{15}	B_{06}	B_{77}
B_{70}	B_{61}	B_{52}	B_{43}	B_{34}	B_{25}	B_{16}	B_{07}

After the permutation, we then XOR the state and the round key and conclude the first round. We repeat until the 4th round. In the last round (5^{th} round), this round excludes the permutation operation so we only need to perform substitution and XOR operation.

For the decryption process, one would only need to reverse all the process of the encryption algorithm. We perform each round in the reverse order (i.e. XOR operation, then permutation and substitution). We will iterate the key lists in descending order for each round and permutation will shift the columns upward circularly.

3 - Implementation

3.1 - Encryption/Decryption function

The encryption function takes a list of 8 bytes and a list of 6 keys (list of lists), so a block and key lists respectively (we will use this term throughout this section). This function ensures that there are five rounds of the substitution and XOR run, but only four rounds of the permutation ran, as per the coursework specification. It also ensures that the keys used for the XOR functionality is different every time a new round is started. This means that the same key is not used, which increases the security of the algorithm greatly (see *Figure 7.1.1 for code*).

The decryption function is very similar to the encryption method, except it calls Substitution and Permutation functions using the inverse S-box instead of the regular S-box, which allows for the decryption to take place (see *Figure 7.1.2 for code*).

3.1.1 – Substitution function

The substitution function takes a block and string which will instruct the function to use a regular s-box or an inverse s-box. The two s-boxes are hardcoded into the program as dictionaries. To use the s-boxes, we need two keys (the hexadecimal) and insert them into the dictionary to find the substitution. *Figure 7.1.3* below shows the process of performing the substitution.

```
def substitution(block,mode):
    if mode == 'inv':
        for i in range(len(block)):
            hx = hex(block[i]).lstrip('-0x').zfill(2).upper()
            sub = sboxinv[hx[0]][hx[1]]
            block[i] = int(sub,16)
    else:
        for i in range(len(block)):
            hx = hex(block[i]).lstrip('-0x').zfill(2).upper()
            sub = sbox[hx[0]][hx[1]]
            block[i] = int(sub,16)
```

Figure 7.1.3 – Substitution

In the function, we first determine whether we use the regular s-box or inverse s-box from the input *'mode'.* We use inverse s-box for decryption. Next, it will loop through the block. For each byte (decimal) in the block, we first convert it into hexadecimal (i.e. 42 to 0x2a). Then strip away the unnecessary characters '-0x', pads string on the left with zeros to fill width to 2 and convert all of the characters to upper-case (i.e. lstrip, zfill, upper). This will ensure that only the important data is captured from the hexadecimal block which makes it much easier to deal with. We get the two number of hexadecimal and use it as a key to find the substitution in the s-box. Lastly, we convert the substituted value into decimal and store it back into the block.

3.1.2 – Permutation function

The permutation function takes a block and string which determine whether to perform an encryption permutation or decryption permutation. The permutation function starts off by iterating each byte (decimal) in the block and converting the byte to binary. Then strips unnecessary characters (just as the substitution function did, except this time it removes '-0b' and pads string on the left with zeros to fill width to 8) and adds the converted number to a list. We have a list of binary numbers at the end of the iteration named 'bitelist' (*see Figure 7.1.4*)

```
def permutation(block,mode):
    bitelist = []
    length = len(block)
    for i in range(length):
        bite = bin(block[i]).lstrip('-0b').zfill(8)
        bitelist.append(str(bite))
    if mode == 'inv':
        for i in range(length):
            pbite = ''
            for j in range(length):
                num = (i+j) % length
                pbite = pbite + str(bitelist[num][j])
            block[i] = int(pbite,2)
    else:
        for i in range(length):
            pbite = ''
            for j in range(length):
                num = (i-j) % length
                pbite = pbite + str(bitelist[num][j])
            block[i] = int(pbite,2)
```

Figure 7.1.4 – Permutation

After the conversion, we will perform the permutation of the bits. For normal permutation, we have two for loop. The first for loop 'i', will iterate through binary numbers in the list (row x) and the second for loop 'j', will iterate through the bits in the binary number (column y). The row and column refer to *Table 1* above, where a bit is represented by $B_{x,y}$.

Notice that on each row we have $B_{(x-y \bmod 8),y}$.

x/y	0	1	2
0	B_{00} -> $B_{(0-0 \bmod 8),0}$ -> B_{00}	B_{01} -> $B_{(0-1 \bmod 8),1}$ -> B_{71}	B_{02} -> $B_{(0-2 \bmod 8),2}$ -> B_{62}
1	B_{10} -> $B_{(1-0 \bmod 8),0}$ -> B_{10}	B_{11} -> $B_{(1-1 \bmod 8),1}$ -> B_{01}	B_{12} -> $B_{(1-2 \bmod 8),2}$ -> B_{72}

By subtracting row x by column y, we will obtain the column wise permutation where bits are shifted downward circularly. We store the new binary number into the block which will complete the permutation operation. To reverse the operation, we use $B_{(x+y \bmod 8),y}$.

3.1.3 – XOR function

The XOR function is very simple. It takes a block and a round key, loop through the block and key, getting the current byte (decimal) of the block and byte of the key and do an XOR operation using the operator '^'. The operation is the same for both encryption and decryption. This offers more flexibility in the system as there Is less redundancy. For this function (*see Figure 7.1.5*).

3.1.4 - Key creation function

The key creation takes place at the initialisation of the program. It uses the hardcoded key in the program which is consists of 8 bytes (decimals) and use the 'createkeys' and 'roundkey' functions to create the next 5 round keys for the encryption/decryption. In the end, we have a list of keys which consists of the hardcoded key and the 5 round keys created using the two functions. The two figures below show the function of creating round keys.

```
def createkeys(keylist,origin):
    key = copy.deepcopy(origin)
    keylist.append(copy.deepcopy(key))
    for i in range(5):
        roundkey(key)
        keylist.append(copy.deepcopy(key))
```

Figure 7.2.1 - Create keys

```
def roundkey(key):
    for i in range(len(key)):
        if i%2 == 0:
            key[i] = key[i] ^ key[i+1]
        else:
            hx = hex(key[i]).lstrip('-0x').zfill(2).upper()
            sub = sbox[hx[0]][hx[1]]
            key[i] = int(sub,16)
```

Figure 7.2.2 - Round keys

The 'createkeys' function takes an empty list (key list) and the initial key (hardcoded in the program). The initial key is copied into a variable and appended into the key list. Then, the program takes the copied key into a loop. The loop will call the 'roundkey' function with the copied key where it will modify the copied key and append the modified key into the key list. The loop will occur five times and we will have a list of 6 keys in the key list.

The 'roundkey' function takes the key (list of 8 decimals) and modify the list in order to produce a new key. We iterate through each byte (decimals) in the key. When the index (row) of key is even number, the byte is XOR'd with the next byte in the key. Otherwise, the byte is substituted with the s-box. We use modulo 2 to determine if the row is even or odd.

3.2 - Modes of operation

All modes of operation required in the coursework specification are implemented. The implemented modes are ECB (electronic codebook), CBC (cipher block chaining), CFB (cipher feedback), OFB (output feedback) and CTR (counter). All modes of operations take a list of blocks, a key list and a string which determine whether it is an encryption or decryption. If the passed string is 'deqt' then the decryption function is used, otherwise the encryption function is used. All initialisation vectors (IV) is hardcoded in the program and it will initialise when the function is called in the program.

3.2.1 – ECB (Electronic Codebook)

ECB is the simplest of all the modes of operation. The ECB function will iterate through the list of blocks and run the encryption or decryption function with each block and the key list. The downside of ECB is that it does not hide patterns well, meaning it is quite vulnerable to a number of attacks. (See *Figure 7.3.1* for the code).

3.2.2 – CBC (Cipher Block Chaining)

In CBC encryption, we first perform an XOR operation between the plaintext and the ciphertext (which is obtained from the previous block), run the encryption function with the key list and repeat this process until we reach the last block.

This process can be achieved by a for loop with the number of blocks as the limit. For the first block in the list, the initialisation vector (IV) is XOR'd with the first block (plaintext) since the previous block (ciphertext) doesn't exist. The block is then encrypted with the key list by the encryption function. For the remaining blocks, the current block will be XOR'd with the previous block and encrypted. (See *Figure 7.3.2* for the code).

In CBC decryption, the process is reversed where we iterate the list of blocks in a descending order starting from the last block, run the decryption function and XOR'd with the next block until we reach the first block where we XOR'd with the IV. In CBC, if a block is altered, then the rest of the blocks following it will be changed resulting a different ciphertext starting from the altered block. This is much more secure than ECB.

3.2.3 – OFB (Output Feedback)

For OFB encryption, it will run the same function regardless whether it is for encryption or decryption. In this encryption, we perform an encryption operation between the key and the initialisation vector to produces the keystream which is then used in an XOR operation with the plaintext to produces the block of ciphertext. IV is replaced by the keystream and a new keystream is produced by repeatedly encrypting the old keystream using the same key. (See *Figure 7.3.3* for the code) In result, every subsequent encryption operation after the first one is not dependent on the ciphertext of the first one, but the keystream. This eliminates the possibility of one block of ciphertext potentially damaging the security of another.

3.2.4 – CFB (Cipher Feedback)

CFB operation is similar to OFB but instead of encrypting the old keystream with the key to produce a new keystream, the ciphertext produced by the XOR'd operation is used.
For CFB encryption, we use the IV and the key list for encryption to produce a keystream. Once we produce a ciphertext from the XOR'd operation with the keystream and first block, we use the encrypted block (ciphertext) and the key list to produce a new keystream. We continue using the previous ciphertext with the key list to produce a new keystream until the last block. (See *Figure 7.3.4* for the code).

For CFB decryption, we reverse the process, iterating through the list of block in descending order. Producing a keystream using the previous block and the key and XOR the keystream with the current block. The last keystream will be produced by the IV and key.
Therefore, this meets the CFB requirements.

3.2.5 – CTR (Counter Mode)

CTR has a unique implementation. For every block, the counter increment by 1 and the counter value is added to the initialisation vector results a different initialisation vector. The IV is encrypted with the key list and XOR'd with the block to produce the ciphertext.
As a CTR operation is totally separated from any subsequent operations, the first ciphertext block is in no way a security risk to any subsequent blocks. (*see Figure 7.3.5* for code)

3.3 – Input, Output and Conversion

In the program, the user can either input a message into the terminal or insert a .txt file (see section 3.5 for details). To input an message, 'raw_input()' is used to ask the user to input a message. The input is then stored as a string and inserted into the "asciitodec" function for conversion. The "asciitodec" function is used to convert string to list of bytes (decimals). It converts the string into an individual list of characters and convert each character into their corresponding ASCII decimals by iterating through the list and using the build-in function 'ord(*char*)'. (see *Figure 7.4.1*)

To read a file, the program will run the function 'readfile(*mode*)', where '*mode*' is a string which help determine the text format in the file (*see Figure 7.4.7*). In the function, it will prompt the user to input the name of the file to open the file in the program. First, the function will read the first two character in the file, to determine whether the text is in hexadecimal or binary. If the character is '0x', then it is in hexadecimal format. If the character is '0b', then it is in binary format. If it is neither, we check is the string '*mode*' == '*deqt*', which means it is a ciphertext and it is in decimal format. If all fails, then the program will determine the text format as normal text in ASCII characters. Once the function recognises the format, it will split the character codes by ';' and convert them into ASCII decimal. The function will return a list of ASCII decimals.

Once we obtain a list of ASCII decimals, we use 'createblocks' function to split into a list of lists (list of blocks), where each of individual list will contains 8 ASCII decimals. This function will be explained in section 3.4 (each will act as a 'block', see *Figure 7.4.2*). The outputs in this program consists either printing ciphertext to the terminal and output both plain and cipher text into .txt file.

The function "iotext" handle most of the output to terminal messages (see *Figure 7.4.6*, for code). This function prompts the user choose whether to print encrypted/decrypted text to terminal and to save the text into a .txt file. The user will also be given a choice to print the encrypted/decrypted text into decimal, hexadecimal or binary for encrypted text (ciphertext) and normal text for decrypted text (plaintext). To output as normal text, 'decryptascii' function is used. (See section 3.4 for details). The function 'newfile' is called when the user wants to write encrypted/decrypted text to .txt file.

The function 'newfile' will create a .txt file with the encrypted/decrypted text (see *Figure 7.4.4*, for code). It will take two inputs, a list of blocks and string '*mode*' which specify whether the list is encrypted or decrypted. For encrypted text, the user will have a choice the write the text as a hexadecimal, decimal or binary into the file. For these format, each byte is separated with ';', which will make it easier for the program the read the ciphertext later. For decrypted text, the function will automatically write to file as normal ASCII text using the 'decryptascii' function.

3.4 – Creating Blocks and Padding

Once the program obtains a list of decimals, we convert the list into a list of lists where each list contains 8 decimals. In other words, a list of blocks and each block has 8 bytes, so 64 bits in total for a block. The function 'createblocks' will perform this operation and also apply any padding when necessary (*see Figure 7.4.2* for code). The padding method used for this program is ANSI X7.23. It set the last byte in the padded block as the number of padding bytes required to complete the block and set all padding bytes as zeros.

The function 'createblock' first check the inputted list of decimals requires any padding. To check this, we work out if (*length of list) modulo 8 = 0*. If this calculation is true, then we can divide the decimals into blocks of 8 decimals (byte), else we work out the number of padding needed by using this equation *8 – [(length of list) modulo 8]*. Next, we iterate through the list of decimals, append decimals into a temporary list (which is the block) and for every 8th round, we append that block into a list called 'listblocks' which is the list of blocks the function will be returning and clear the temporary list for the next set of decimals. When the iterate reaches the last decimals, if the padding needed is greater than 0, a for loop will add the padding bytes needed to the last block and set the last byte in the block as the number of padding bytes used.

3.5 - User Interface

The program has 6 selections. The selections are:

1. To enter a message to encrypt (direct user input)
2. Reading from a file to encrypt
3. Reading from a file to decrypt
4. Testing performance of encrypt and decrypt functionality
5. Test mode
6. Termination

Before the user is able to select the options, a list of keys is created and this list of keys will be used throughout the program. (see *Figure 7.2.1 and section 3.1.4 for details*)

3.5.1 - Direct User Input

This selection allows the user to directly input a message (which will act as the plaintext to become ciphertext later on) into the program without needing to store it in a file first. This has the capability to allow for quick encryption of a message that is needed to be kept secret.

The message is then converted into an individual list of characters, and then utilising the "asciitodec" function (see *Figure 7.4.1*) the characters are converted into their corresponding ASCII decimals. Then using the 'createblocks' function, the list of ASCII decimals is split into a 'list of lists', each of which contains 8 ASCII decimals. (each will act as a 'block', see *Figure 7.4.2*).

After splitting the decimals into blocks, the function 'modeofop' will be called. This function allows the user to choose which mode of operations they wish to use to encrypt/decrypt the message (see *Figure 7.4.3*, for code). The function takes three inputs, a list of blocks, a

list of keys and a string which specify whether we use the mode of operation for encryption or decryption.

There are five available operations: ECB, CBC, OFB, CFB and CTR. Each of them utilise the encryption function in different ways (see section 3.2 for details). In the function, aside from calling the chosen operation, it will also call the function 'iotext' which will handle all outputs (i.e. to terminal or to file) available for the user (see section 3.3 for details). After all the output to terminal or creating .txt file to store the encrypted text, the user will also be given the choice to decrypt the message immediately. Once selected, it will run the 'modeofop' function again but for decryption. At the end, the program will direct the user back to the main menu.

3.5.2 – Reading from a file for encryption

This selection is almost identical to the selection 1 except for inputting and reading the plaintext into the program. When is selection is chosen, we run the function 'readfile'. This function allows the user to read a .txt file and return a list of ASCII decimals. (see Section 3.2 for details). Once we have a list of ASCII decimals, the rest of the operation for this selection is exactly the same as described in selection 1. The choice to decrypt the generated ciphertext immediately is also available to the user. At the end, the program will direct the user back to the main menu.

3.5.3 – Reading from a file for decryption

This selection is again almost identical to selection 2 but instead of reading a file to encrypt, we are reading a file to decrypt. Using the function 'readfile', we pass the string 'deqt' to specify for reading an encrypted file (see section 3.2 for details). Next, we convert the list of decimals into a list of blocks and run the 'modeofop' function for decryption. Once the decryption is finished ('modeofop' function ended), the program will direct the user back to the main menu.

3.5.4 – Testing performance selection

This selection allows the user to measure the speed of encryption and decryption of a particular plaintext with a specific mode of operation and display its statistical information in the terminal. For this selection, the user can either input a message, insert a file or use the default block (hardcoded in the program) for testing. The inputted data will be converted into a list of blocks using methods mentioned in section 3.3.1 and section 3.3.2. Next, the user will be prompt to choose a mode of operation for the test. Once chosen, we use the 'wrap' function (see *Figure 7.4.5* for code) to wrap the mode of operation function and it associated inputs into a variable so we can call refer the function without arguments in order to pass it into 'timeit.timeit' function. There are two 'wrapped' function, one for encryption and one for decryption. The 'timeit.timeit' function is a way to time small bits of python code. Using the 'timeit.timeit' function, we can strictly measure the execution time for the encryption and decryption of the mode of operation with the given data. This will remove any redundant time measurement needed for the program.

At the end of the selection, it will output to the terminal, the number of blocks created from the plaintext, the number of bytes and bits in total from the plaintext, the time needed for

encryption in seconds, the time need for decryption in seconds and the encryption and decryption speed per block in seconds. Then the program will direct the user back to the main menu.

3.5.5 – Test mode

The selection is similar to selection 4. This selection shows all the format the plaintext will take in the program before and after the encryption process. First the user will input data either by terminal, insert file or using default block and it will be converted in to list of blocks. Similar to selection 4, the user will then choose a mode of operation and it will be wrapped by the 'wrap' function. Now, we print out the initial list of blocks to the terminal and label this as the original to show the user the initial data the program obtained from the plaintext. The list of blocks is in decimal format.

Next, it will call the 'wrapped' encryption function and print out the list of blocks into the terminal for the user to see. Lastly, we call the 'wrapped' decryption function and print out the block. This selection exists to let the user see the padding, the difference of encrypted text between different mode of operation and to check the encryption and decryption is working in the program.

4 - Demonstration

The plaintext message that is to be used is “Hello :) this is a demonstration”. This is 32 characters, which is the required 256-bit message that requested in the coursework specification. To demonstrate the encryption and decryption for different mode of operation, each section shows one mode of operation and display the ciphertext in ASCII decimal form, and then the plaintext in ASCII decimal form, as well as printing the decrypted ciphertext in text form. The decrypted ciphertext message should be the same the plaintext message and each mode should generate a different ciphertext. This serve as a proof that the program is generating different ciphertext for different mode of operations.

4.1 - ECB

The screenshot below shows the original plaintext in decimal form in blocks. We can see that the decrypted blocks and the original blocks are the same which imply successful decryption.

```
Please enter an message: Hello :) this is a demonstration
original
[72, 101, 108, 108, 111, 32, 58, 41]
[32, 116, 104, 105, 115, 32, 105, 115]
[32, 97, 32, 100, 101, 109, 111, 110]
[115, 116, 114, 97, 116, 105, 111, 110]
Choose mode of encryptions:
1.ECB  2.CBC  3.OFB  4.CFB  5.CTR
1
Encrypt:
[28, 22, 58, 184, 106, 41, 48, 51]
[194, 190, 117, 76, 83, 38, 125, 4]
[165, 68, 161, 74, 81, 53, 44, 190]
[146, 174, 95, 109, 105, 87, 88, 177]
Decrypt:
[72, 101, 108, 108, 111, 32, 58, 41]
[32, 116, 104, 105, 115, 32, 105, 115]
[32, 97, 32, 100, 101, 109, 111, 110]
[115, 116, 114, 97, 116, 105, 111, 110]
Plaintext:
Hello :) this is a demonstration
```

4.2 – CBC

We can see the screenshot below is has a different encryption but the same decryption as the original blocks which you can see from ECB and decrypted ciphertext message is the same as the plaintext message.

```
Choose mode of encryptions:
1.ECB  2.CBC  3.OFB  4.CFB  5.CTR
2
Encrypt:
[15, 27, 89, 75, 193, 222, 41, 217]
[55, 140, 62, 224, 219, 101, 165, 119]
[55, 200, 248, 219, 116, 131, 51, 42]
[188, 219, 143, 56, 161, 19, 131, 175]
Decrypt:
[72, 101, 108, 108, 111, 32, 58, 41]
[32, 116, 104, 105, 115, 32, 105, 115]
[32, 97, 32, 100, 101, 109, 111, 110]
[115, 116, 114, 97, 116, 105, 111, 110]
Plaintext:
Hello :) this is a demonstration
```

4.3 – OFB

```
Choose mode of encryptions:
1.ECB  2.CBC  3.OFB  4.CFB  5.CTR
3
Encrypt:
[77, 253, 100, 42, 16, 230, 11, 201]
[72, 196, 157, 87, 156, 157, 86, 81]
[125, 207, 81, 12, 167, 245, 168, 187]
[74, 247, 146, 124, 244, 63, 16, 39]
Decrypt:
[72, 101, 108, 108, 111, 32, 58, 41]
[32, 116, 104, 105, 115, 32, 105, 115]
[32, 97, 32, 100, 101, 109, 111, 110]
[115, 116, 114, 97, 116, 105, 111, 110]
Plaintext:
Hello :) this is a demonstration
```

4.4 – CFB

```
Choose mode of encryptions:
1.ECB  2.CBC  3.OFB  4.CFB  5.CTR
4
Encrypt:
[77, 253, 100, 42, 16, 230, 11, 201]
[210, 215, 122, 78, 163, 186, 155, 192]
[203, 171, 160, 237, 183, 204, 242, 97]
[123, 125, 170, 15, 55, 13, 173, 170]
Decrypt:
[72, 101, 108, 108, 111, 32, 58, 41]
[32, 116, 104, 105, 115, 32, 105, 115]
[32, 97, 32, 100, 101, 109, 111, 110]
[115, 116, 114, 97, 116, 105, 111, 110]
Plaintext:
Hello :) this is a demonstration
```

4.5 – CTR

```
Choose mode of encryptions:
1.ECB  2.CBC  3.OFB  4.CFB  5.CTR
5
Encrypt:
[77, 253, 100, 42, 16, 230, 11, 201]
[7, 87, 242, 103, 63, 230, 253, 114]
[61, 198, 56, 150, 129, 81, 20, 169]
[84, 185, 209, 176, 152, 91, 226, 150]
Decrypt:
[72, 101, 108, 108, 111, 32, 58, 41]
[32, 116, 104, 105, 115, 32, 105, 115]
[32, 97, 32, 100, 101, 109, 111, 110]
[115, 116, 114, 97, 116, 105, 111, 110]
Plaintext:
Hello :) this is a demonstration
```

5 - Performance

5.1 - Analysis of time performance measurements

The time performance of our program is measured accurately within the program itself, and the details presented to the user in a clear and constructive manner. When encrypting and decrypting a file of 98,440 bytes (which has 787,520 bits and 12,305 blocks), the results for utilising the ECB mode of operation encrypted the file in 2.559 seconds, but decrypted it in 2.552 seconds, it therefore had an encryption speed per block of 0.208 milliseconds and a decryption rate per block of 0.207 milliseconds.

In general, the encryption and decryption rate per block do not change too drastically depending on the file size, but they do change quite a lot when using different modes of operations. For example, when encrypting and decrypting the very same file using CBC rather than ECB (as the initial test was done in), the program took 2.363 seconds to encrypt the file and 2.407 seconds to decrypt, meaning it had a per block throughput of 0.192 milliseconds and 0.196 milliseconds respectively, which is a marginal increase in speed on a small scale, but for very large file encryption and decryption the change in speed would be noticeable.

For OFB, the encryption and decryption speeds of the same file were 2.425 seconds and 2.390 seconds respectively. This means the per block throughput was 0.197 milliseconds and 0.194 milliseconds, which is not a large enough difference with CBC to be noticeable, and within a decent margin of error.

CFB on the other hand was slower than both OFB and CBC, and more comparable with ECB. With an absolute speed for encryption and decryption of 2.582 seconds and 2.534 seconds respectively (giving a per block throughput of 0.210 milliseconds and 0.206 milliseconds for encryption and decryption), it was significantly slower than both OFB and CBC. This might be because instead of using the keystream for the next operation, the program must wait for the ciphertext to be produced before it is able to continue onto the next block.

The same may be said for CTR, except with this mode of operation the program must wait for the entire block to be completely encrypted (or decrypted) before it moves on to the next block, as a result, it had speeds similar to that of ECB and CFB. The absolute speeds recorded were 2.596 seconds for encryption and 2.602 seconds for decryption, with a per block throughput of 0.211 milliseconds for encryption and 0.212 milliseconds for decryption.

Based on these findings, it is relatively safe to assume that on the current build of the group's program, CBC is the fastest mode of operation for this specific file in use. This may be because the XOR operation is done on the plaintext and initialisation vector, rather than on the keystream and message (after the encryption) as in CFB or CTR (which were the slowest two operations by some margin).

5.2 - Possible performance improvements

There are a few potential performance increases that could be made by some alterations to the project. For example, since each block encryptions are independent for ECB and CTR operation, it might be possible to run them in a multithreading format, meaning that encrypting blocks could happen concurrently. This could potentially increase the performance of the program by a very large amount, specifically when encrypting and decrypting a large number of bits.

Another improvement could be made by having two separate programs, one for encryption, and one for decryption. This would make the program file much smaller in general, and would increase the runtime of the program on low-end computers in particular. It also assists in saving memory (i.e. the inverse S-box would not be saved and initialised in the memory until the decryption program was run). The increase in performance for that aspect may be marginal, but it would increase the performance slightly.

6 - Analysis of security

There are a few potential security issues that would affect the program that our group has developed, for example utilising a 64-bit key nowadays is not a particularly large key given modern computing and processing power, it is not beyond the realm of possibility for an attacker to attempt to brute force a 64-bit key. The security of this section could be increased the number of bits in a key, but the size of the block would then also need to be increased, which was not allowed under the coursework specification.

Another security weakness is that the keys, counters and the incremental initialisation vector (IV) are hardcoded in the program, which means the same keys and IV are used for all encryption. This implies that the keystream for OFB will the same, the counters in CTR will be in a fixed pattern making it vulnerable to attacks. Also, since the same keys and IV is used for all encryption, it is possible to identify any pattern and similarity between ciphertexts and to uncover the key and IV. If a ciphertext is brute forced and broken, then all ciphertext created by this program is compromised since they all uses the same key and IV. The fix to this issue would be to create random initialisation vectors, counters and keys, which would then be much harder to guess.

The encryption algorithm for this project has good confusion and diffusion. It has a good confusion because for each block in the algorithm, we XOR'd the block with several round keys which is created from the original key hardcoded in the program. The resulting ciphertext would depend several parts of the key due to the round key operation. If one bit of the ciphertext is changed, one column of the block is changed due to permutation, resulting into a different byte in the blocks.

The algorithm has good diffusion because if we change one bit of plaintext, the whole block will be affected by the change. If one bit of plaintext was changed, the substitution operation would affect one byte and the bitwise permutation would affect more the half of the block because all columns are shifted downward circularly. Since one of the row has been affected, the shifting will cause all columns to be affected, so all bytes in the block will be compromised.

7 - Appendices

7.1 - Encryption and Decryption

Figure 7.1.1 – Encryption function

```
def encryption(state,keys):
    bitxorstate(state,keys[0])
    for i in range(5):
        if i == 4:
            substitution(state,'reg')
            bitxorstate(state,keys[i+1])
        else:
            substitution(state,'reg')
            permutation(state,'reg')
            bitxorstate(state,keys[i+1])
```

Figure 7.1.2 – Decryption function

```
def decryption(state,keys):
    for i in range(5, 0, -1):
        if i == 5:
            bitxorstate(state,keys[i])
            substitution(state,'inv')
        else:
            bitxorstate(state,keys[i])
            permutation(state,'inv')
            substitution(state,'inv')
    bitxorstate(state,keys[0])
```

Figure 7.1.3 – Substitution

```
def substitution(block,mode):
    if mode == 'inv':
        for i in range(len(block)):
            hx = list(hex(block[i]).lstrip('-0x').zfill(2).upper())
            sub = sboxinv[hx[0]][hx[1]]
            block[i] = int(sub,16)
    else:
        for i in range(len(block)):
            hx = list(hex(block[i]).lstrip('-0x').zfill(2).upper())
            sub = sbox[hx[0]][hx[1]]
            block[i] = int(sub,16)
```

Figure 7.1.4 – Permutation

```
def permutation(block,mode):
    bitelist = []
    length = len(block)
    for i in range(length):
        bite = bin(block[i]).lstrip('-0b').zfill(8)
        bitelist.append(str(bite))
    if mode == 'inv':
        for i in range(length):
            pbite = ' '
            for j in range(length):
                num = (i+j) % length
                pbite = pbite + str(bitelist[num][j])
            block[i] = int(pbite,2)
    else:
        for i in range(length):
            pbite = ''
            for j in range(length):
                num = (i-j) % length
                pbite = pbite + str(bitelist[num][j])
            block[i] = int(pbite,2)
```

Figure 7.1.5 – XOR function

```
def bitxorstate(state,key):
    for i in range(len(state)):
        state[i] = state[i] ^ key[i]
```

7.2 - Key Creation

Figure 7.2.1 - Create keys

```
def createkeys(keylist,origin):
    key = copy.deepcopy(origin)
    keylist.append(copy.deepcopy(key))
    for i in range(5):
        roundkey(key)
        keylist.append(copy.deepcopy(key))
```

Figure 7.2.2 - Round keys

```
def roundkey(key):
    for i in range(len(key)):
        if i%2 == 0:
            key[i] = key[i] ^ key[i+1]
        else:
            hx = list(hex(key[i]).lstrip('-0x').zfill(2).upper())
            sub = sbox[hx[0]][hx[1]]
            key[i] = int(sub,16)
```

7.3 - Modes of Operation

Figure 7.3.1 - ECB

```
def modeECB(tx,key,mode):
    if mode == 'deqt':
        for block in tx:
            decryption(block,key)
    else:
        for block in tx:
            encryption(block,key)
```

Figure 7.3.2 – CBC

```
def modeCBC(tx,key,mode):
    iv = [1,2,3,4,5,6,7,8]
    if mode == 'deqt':
        for i in range(len(tx)-1,-1,-1):
            decryption(tx[i],key)
            if i == 0:
                bitxorstate(tx[i],iv)
            else:
                bitxorstate(tx[i],tx[i-1])
    else:
        for i in range(len(tx)):
            if i == 0:
                bitxorstate(tx[i],iv)
            else:
                bitxorstate(tx[i],tx[i-1])
            encryption(tx[i],key)
```

Figure 7.3.3 - OFB

```
def modeOFB(tx,key,mode):
    iv = [1,2,3,4,5,6,7,8]
    for i in range(len(tx)):
        encryption(iv,key)
        bitxorstate(tx[i],iv)
```

Figure 7.3.4 - CFB

```
def modeCFB(tx,key,mode):
    iv = [1,2,3,4,5,6,7,8]
    if mode == 'deqt':
        for i in range(len(tx)-1,-1,-1):
            if i == 0:
                encryption(iv,key)
                bitxorstate(tx[i],iv)
            else:
                tempc = copy.deepcopy(tx[i-1])
                encryption(tempc,key)
                bitxorstate(tx[i],tempc)
    else:
        for i in range(len(tx)):
            if i == 0:
                encryption(iv,key)
                bitxorstate(tx[i],iv)
            else:
                tempc = copy.deepcopy(tx[i-1])
                encryption(tempc,key)
                bitxorstate(tx[i],tempc)
```

Figure 7.3.5 - CTR

```
def modeCTR(tx,key,mode):
    iv = [1,2,3,4,5,6,7,8]
    counter = 0
    for i in range(len(tx)):
        ivcp = copy.deepcopy(iv)
        for j in range(len(ivcp)):
            ivcp[j] = int(ivcp[j])+counter
        encryption(ivcp,key)
        bitxorstate(tx[i],ivcp)
        counter += 1
```

7.4 - Miscellaneous Functions

Figure 7.4.1 – asciitodec()

```
def asciitodec(string):
    letters = list(string)
    for i in range(len(letters)):
        letters[i] = ord(str(letters[i]))
    return letters
```

Figure 7.4.2 – createblocks()

```
def createblocks(listletters):
    listblocks = []
    length = len(listletters)
    #print 'length',length
    if length % 8 != 0:
        padding = 8 - (length % 8)
    else:
        padding = 0
    block = []
    for i in range(length):
        if (i != 0) and (i % 8 == 0):
            listblocks.append(copy.deepcopy(block))
            block = []
        if i == length-1:
            block.append(listletters[i])
            if padding > 0:
                for j in range(padding):
                    if j == padding-1:
                        block.append(padding)
                        listblocks.append(copy.deepcopy(block))
                    else:
                        block.append(0)
            else:
                listblocks.append(copy.deepcopy(block))
        else:
            block.append(listletters[i])
    return listblocks
```

Figure 7.4.3 – modeofop()

```
def modeofop(crypt,listblocks,keyslist):
    print 'Choose mode of encryptions:'
    mode = input('1.ECB  2.CBC  3.OFB  4.CFB  5.CTR \n')
    if mode == 1:
        modeECB(listblocks,keyslist,crypt)
        iotext(crypt,listblocks)
    elif mode == 2:
        modeCBC(listblocks, keyslist, crypt)
        iotext(crypt,listblocks)
    elif mode == 3:
        modeOFB(listblocks, keyslist, crypt)
        iotext(crypt,listblocks)
    elif mode == 4:
        modeCFB(listblocks, keyslist, crypt)
        iotext(crypt,listblocks)
    elif mode == 5:
        modeCTR(listblocks, keyslist, crypt)
        iotext(crypt,listblocks)
```

Figure 7.4.4 – newfile()

```
def newfile(mode,ptx):
    filename = raw_input('Please enter a file name: ')
    fname = filename+'.txt'
    f = open(fname, 'w')
    if mode == 'enqt':
        mode = raw_input('Choose your choice of output: dec, hex or bin \n')
    else:
        mode = 'text'

    if mode == 'hex':
        for i in range(len(ptx)):
            for j in range(8):
                f.write(hex(ptx[i][j]))
                f.write(';')
    elif mode == 'bin':
        for i in range(len(ptx)):
            for j in range(8):
                f.write(bin(ptx[i][j]))
                f.write(';')
    elif mode == 'dec':
        for i in range(len(ptx)):
            for j in range(8):
                f.write(str(ptx[i][j]))
                f.write(';')
    elif mode == 'text':
        output = decryptascii(ptx)
        f.write(output)

    f.close()
```

Figure 7.4.5 - wrap()

```
def wrap(func, *args, **kwargs):
    def wrapped():
        return func(*args, **kwargs)
    return wrapped
```

Figure 7.4.6 - iotext()

```
def iotext(mode,listblock):
    if mode == 'enqt':
        text = 'cipher'
    else:
        text = 'plain'
    listblocks = copy.deepcopy(listblock)
    sel1 = input('Print '+text+'text to screen? Yes = 1, No = 0 \n')
    if sel1 == 1:
        if text == 'cipher':
            sel3 = raw_input('Choose your choice of output: dec, hex, bin\n')
            if sel3 == 'hex':
                for list in listblocks:
                    print ([hex(n) for n in list])
            elif sel3 == 'bin':
                for list in listblocks:
                    print ([bin(n) for n in list])
            elif sel3 == 'dec':
                printblocks(listblocks)
        else:
            output = decryptascii(listblocks)
            print output

    sel2 = input('Store '+text+'text in .txt file? Yes = 1, No = 0 \n')
    if sel2 == 1:
        newfile(mode,listblocks)
        print 'File created, please check directory\n'
```

Figure 7.4.7 - readfile()

```
def readfile(mode):
    filename = raw_input('Please enter a file name: ')
    fname = filename+'.txt'
    f = open(fname, 'r')
    data = []
    formatting = f.read(2)
    f.seek(0,0)
    if formatting == '0x':
        for line in f.readlines():
            temp = line.split(';')
            del temp[-1]

            for i in range(len(temp)):
                temp[i] = int(temp[i],0)
            data.extend(temp)
    elif formatting == '0b':
        for line in f.readlines():
            temp = line.split(';')
            del temp[-1]

            for i in range(len(temp)):
                temp[i] = int(temp[i],0)
            data.extend(temp)
    elif mode == 'deqt':
        for line in f.readlines():
            temp = line.split(';')
            del temp[-1]

            for i in range(len(temp)):
                temp[i] = int(temp[i],0)
            data.extend(temp)
    else:
        data = f.read()
        data = asciitodec(data)
    f.close()
    return data
```