Hacettepe University

Department of Computer Engineering

BBM 465
Information Security Laboratory Report
Assignment I

Mustafa Can Genç - 21827426
Murat Şehzade - 21827828

I. Introduction

Goal of the Project

In this assignment, we are asked to create a new file after encryption or decryption of a file given as input is done in the desired encryption mode and encryption algorithm. The program to be developed must be a CLI program and must produce an output file each time it runs and terminates.

Block Ciphers, DES and Encryption Modes

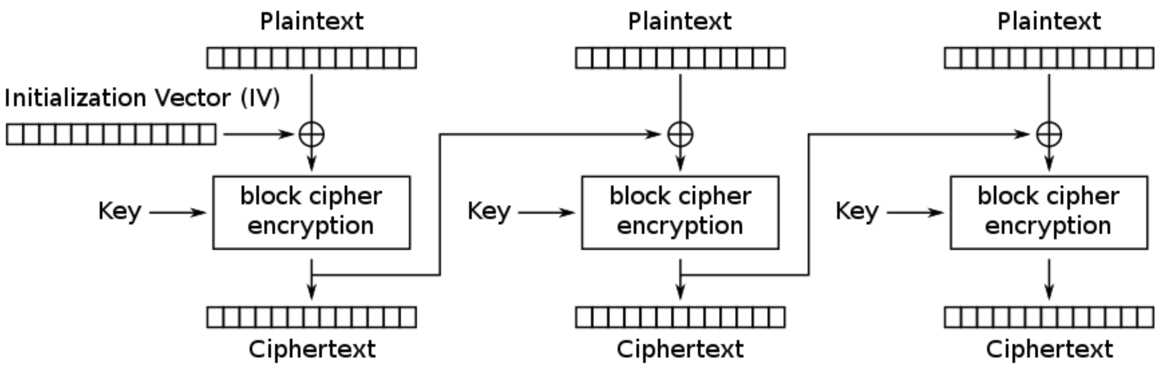
An algorithm that uses a block cipher to offer information security, such as confidentiality or authenticity, is known as a block cipher mode of operation in cryptography. Only one fixed-length collection of bits, known as a block, can be securely transformed (encrypted or decrypted) using a block cipher alone. A mode of operation explains how to consistently use a cipher's single-block operation to safely transform data volumes greater than a block.

The classic block cipher is known as DES (Data Encryption Standard), which converts a predetermined length string of plaintext bits into a second bitstring of the same length through a series of intricate procedures. The block size in the case of DES is 64 bits. Decryption is apparently only possible by individuals who are aware of the specific key that was used to encrypt using DES, which also employs a key to personalize the transformation. The Des algorithm encrypts the 64-bit long A block and turns it into a 64-bit B block. This kind of encryption is known as the Electronic Code Book if each 64-bit encryption is carried out separately.

As we mentioned, we used 4 different encryption modes for this assignment. Let's briefly explain the processes of these modes.

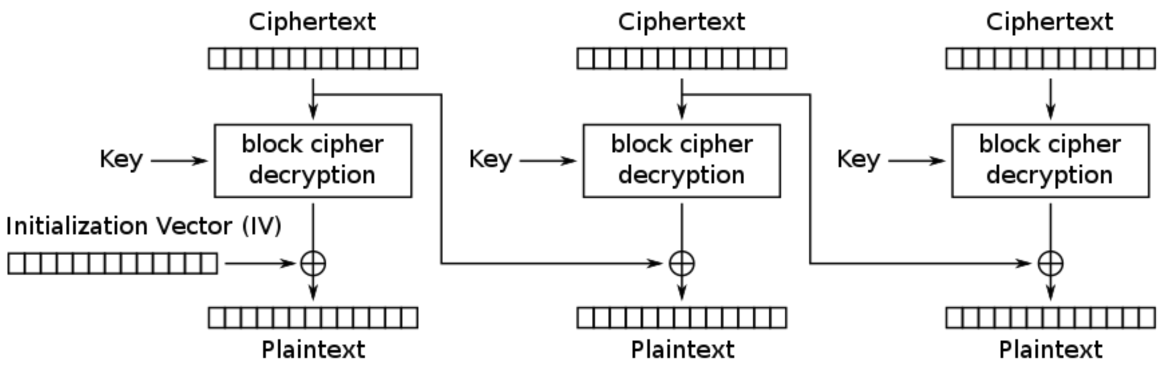
- ### CBC (Cipher Block Chaining)

Each block of plaintext in CBC mode is XORed with the block of ciphertext that came before it before being encrypted. In this manner, each plaintext block processed up until that moment is dependent on every ciphertext block. An initialization vector must be used in the first block to make each message distinct.



Cipher Block Chaining (CBC) mode encryption

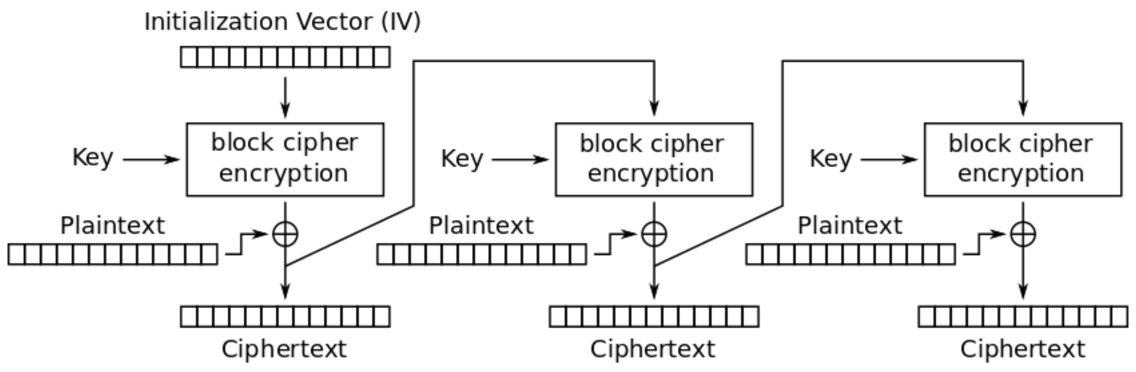
Similar to that, use a key and ciphertext to decrypt a block cipher, XORed with the previous block but only in the first block XORed with Initialization Vector .



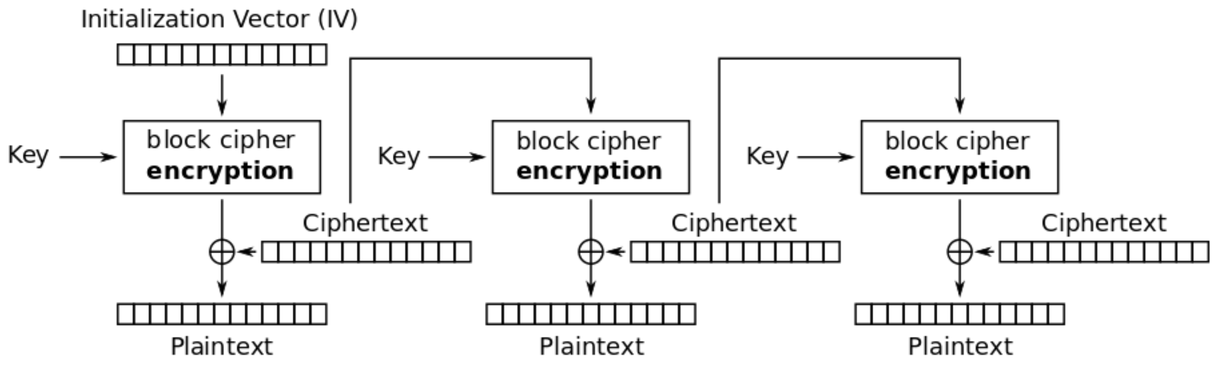
Cipher Block Chaining (CBC) mode decryption

- **CFB (Cipher FeedBack)**

The block cipher's whole output is used in the cipher feedback (CFB) mode's most basic configuration. This version converts a block cipher into a self-synchronizing stream cipher, which is quite similar to CBC. This variant's CFB decryption is nearly identical to CBC encryption carried out backwards.



Cipher Feedback (CFB) mode encryption

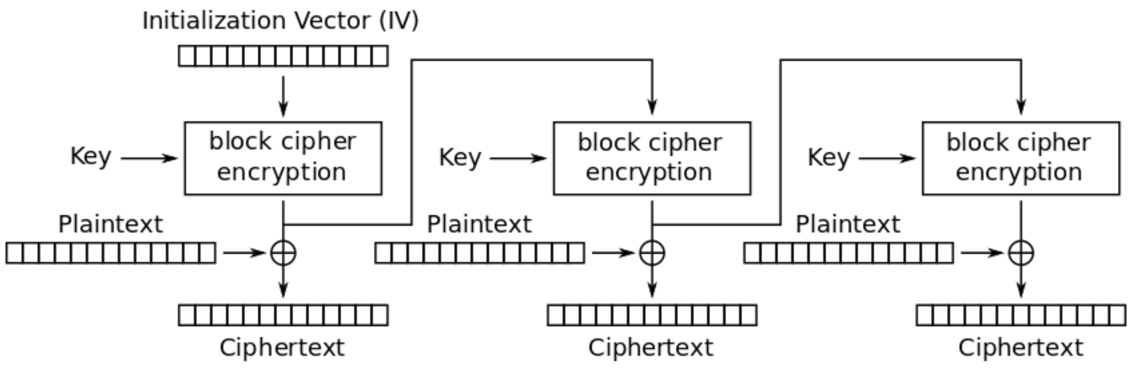


Cipher Feedback (CFB) mode decryption

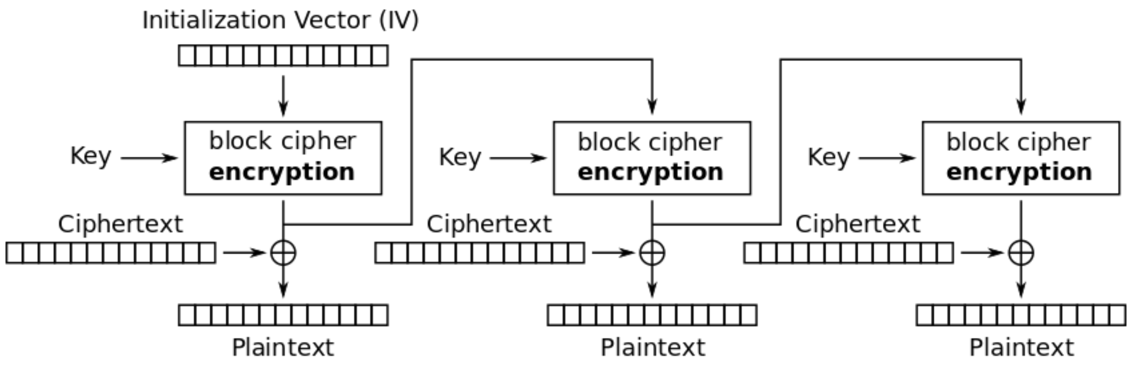
But there is a point to be noted in this mode: block cyper's encryption is used in both encryption and decryption.

- ## OFB (Output FeedBack)

An asynchronous stream cipher is created by a block cipher using the output feedback (OFB) mode. Keystream blocks are created, and the ciphertext is obtained by XORing the keystream blocks with the plaintext blocks. Because of the symmetry of the XOR operation, encryption and decryption are exactly the same, also both of them use block cipher encryption like CFB and CTR on the decryption process.



Output Feedback (OFB) mode encryption


Initialization Vector (IV)
Key
block cipher **encryption**
Key
block cipher **encryption**
Key
block cipher **encryption**
Ciphertext
Ciphertext
Ciphertext
Plaintext
Plaintext
Plaintext

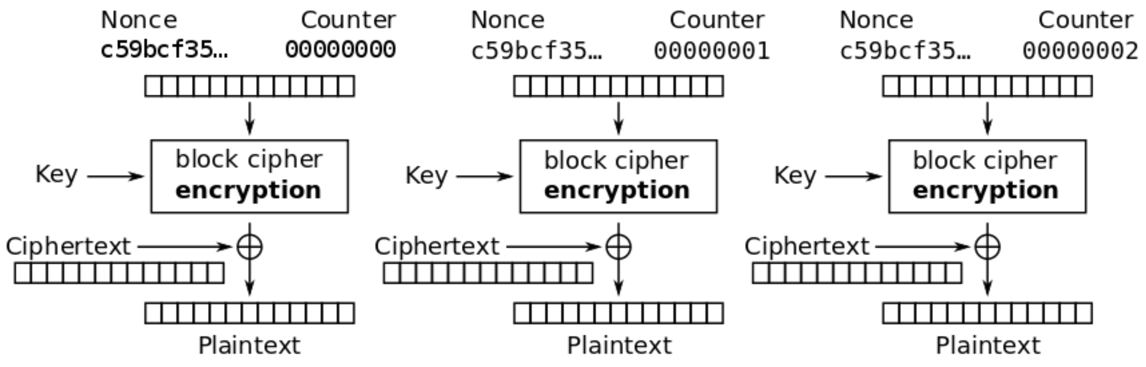
Output Feedback (OFB) mode decryption

- ## CTR (Counter)

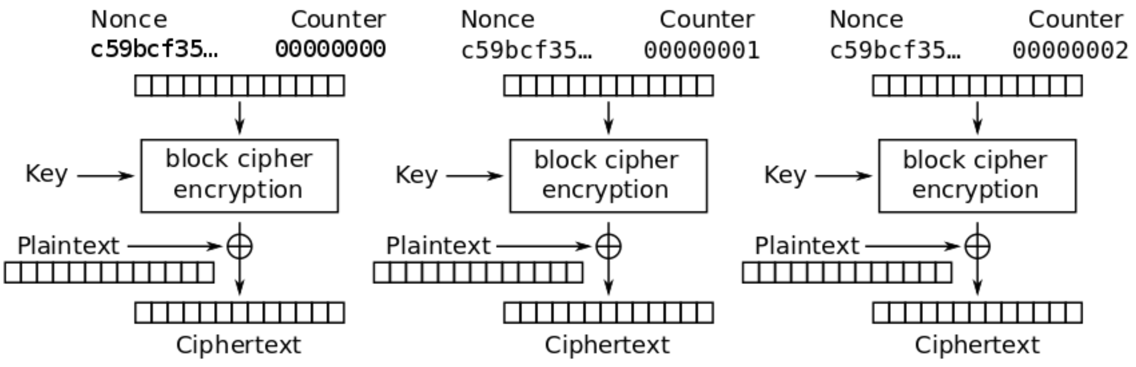
Counter mode converts a block cipher into a stream cipher similarly to OFB. It creates the following keystream block by encrypting a "counter's" increasing values. The counter can be any function that generates a sequence that is guaranteed not to repeat for a very long period, though the most common and straightforward is an actual increment-by-one counter. We are using that in that assignment. Our counter starts from 0 and increments by one .

Substring has 8 characters with each cycle.In this manner, the created combined value and the nonce bit combine to form the counter bit, which then enters the encryption block. The outcome and the ciphertext are entered into the Xor operation to produce the plaintext. The counter value is raised by one for our code in decryption mode.
(So long as there are fewer than 8 bits.)

The terms ciphertext and plaintext are interchanged during encryption and decryption procedures. Keep in mind that the initialization vector (IV) in the other diagrams and the nonce in this picture are the same. Due to the reliance on byte offset, it will be impossible to partially retrieve such data if the offset/location information is corrupt.



Counter (CTR) mode decryption



Counter (CTR) mode encryption

Approach

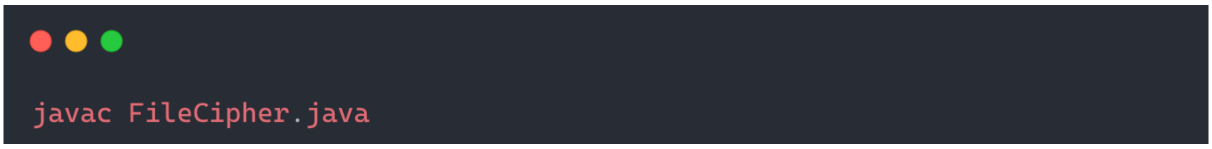
Since we will do all our operations in the program over bytes, we have generally built our FileIO operations on byte reading and writing. We have gathered some functions that will be used in many places in the program under one class.

We used the Java Crypto API for simplicity while developing the program. We implemented separate block cipher algorithms for DES and 3DES. Since the purpose of the assignment is to teach the logic of block ciphering, we always gave the ECB mode in the block mode section when creating the Cipher object, and we determined the values entering and leaving the cipher blackbox separately for each mode.

II. Software Usage

Compile

You can compile the project by using the following java compile code while in the main directory where all the files of the program are located. At the same time, you can use the program by opening the main directory of the project as a project in any IDE and setting the *FileCipher.java* class as the program's entry point.



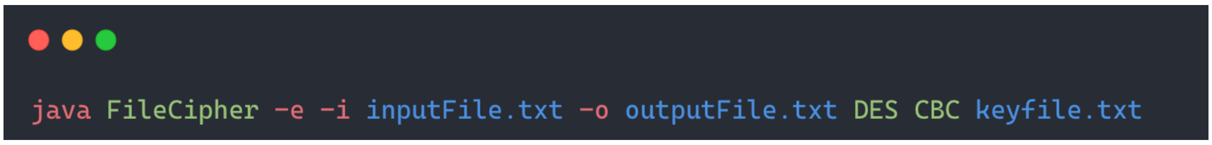
```
javac FileCipher.java
```

Figure 1. Terminal code for compiling project

Command Line Arguments

After compiling the program, we can run it as FileCipher. The FileCipher program needs a set of arguments when executing. The required argument format for the program is:

FileCipher [-e/-d] -i inputFile.txt -o outputFile.txt [Algorithm] [Mode] keyfile.txt



```
java FileCipher -e -i inputFile.txt -o outputFile.txt DES CBC keyfile.txt
```

Figure 2. Example arguments to run the program

The first argument **-e** or **-d** is used to determine whether the program should run in encryption mode (-e) or decryption mode (-d).

Then the **-i** flag indicates that the next argument will be the path of the input file.

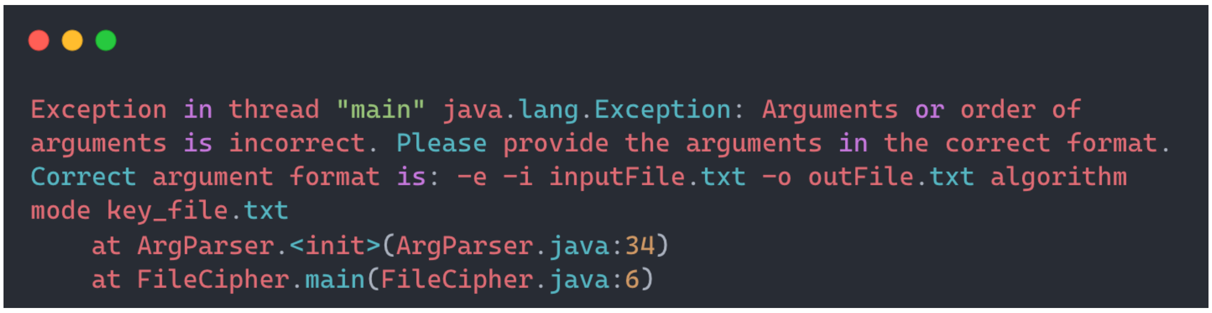
With the same logic, the **-o** flag indicates that the next argument will be the path of the file to be output.

The arguments that can be used in the Algorithm part are **DES** and **3DES**.

The parameter taken as Mode, on the other hand, specifies which encryption/decryption mode the encryption process will be made in. Arguments that can be used are **CBC**, **CFB**, **OFB** and **CTR**.

Error Messages

If any argument other than the requested format is given to the program, an error message is shown to the user stating the error and showing the correct format.

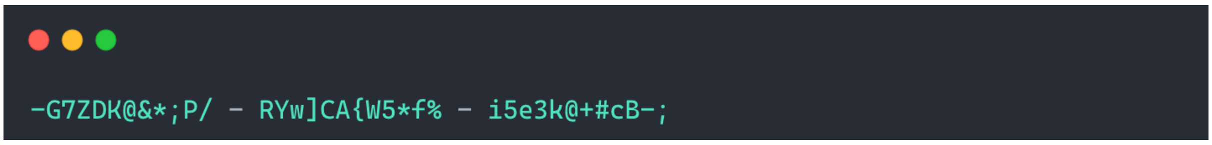


```
Exception in thread "main" java.lang.Exception: Arguments or order of
arguments is incorrect. Please provide the arguments in the correct format.
Correct argument format is: -e -i inputFile.txt -o outFile.txt algorithm
mode key_file.txt
	at ArgParser.<init>(ArgParser.java:34)
	at FileCipher.main(FileCipher.java:6)
```

Figure 3. Example error message of the program when wrong arguments passed

Key File Format

It is mandatory to have 3 different strings, each of which is at least 8 characters long, in the key file. Keys can contain any special character except " - " character pattern. Example key format is shown in Figure 4.



```
-G7ZDK@&*;P/ - RYw]CA{W5*f% - i5e3k@+#cB-;
```

Figure 4. Example of a key file content

III. Algorithms

Parsing Command Line Arguments

First, it is checked whether the command line arguments given during the execution of the program are the correct number. Then, it is checked whether the command given for each flag is in the correct format and the correct ones are saved in the feature fields in the ArgParser class to be used in the later stages of the program. In case of any errors in these controls, an error is thrown and the user is informed (Figure 3.).

File Input/Output

File reading and writing operations are done through the functions of a class we call FileIO. This class has 5 basic functions.

The first of these is the r*eadKeyFile()* function, which allows the key file to be read. In this function, the key file read as a string is split with the "-" separator and all three values in it are returned in a three-element string array.

Read and write functions do simple file read and write operations. The read function returns a byte array. The write function takes the content data of the byte array type as a parameter to write to the file.

Apart from these, the *writeDecryptedFile()* function actually performs the same operation as the write function. The reason why this function is different is due to the fact that some decrypted files have been padded. When the block is not enough for you while the file is being encrypted, padding can be done. Naturally, we need to delete the padding part that we added while decrypting the padded file so that the original file and the decrypted version are identical. Details about the padding process mentioned here are examined in detail in the "Helper Functions in Crypto Class" section.

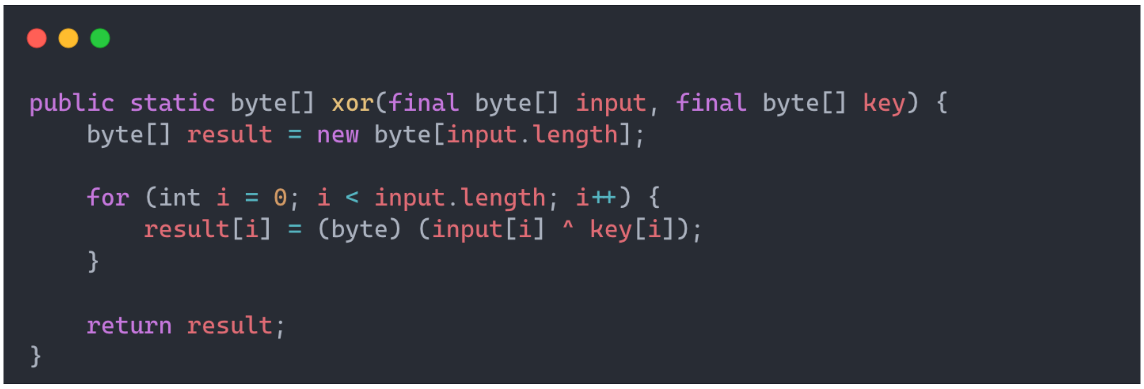
Finally, there is the *writeToLogFile()* function, where logging is done to the log file. After combining the parameters it contains as strings in the desired format, this function calls the write function of FileIO, which I mentioned before, and ensures that the logs are written to the "run.log" file.

Helper Functions in Crypto Class

There is a set of helper functions placed in *Crypto* class. Since these functions are used in common in many algorithms such as mode algorithms and file input/output operations, they are designed as modular as possible and gathered under a single class.

xor()

This function simply XORs each element of the 64-bit byte array given as input with each element of the 64-bit key and returns the calculated value as a byte array.



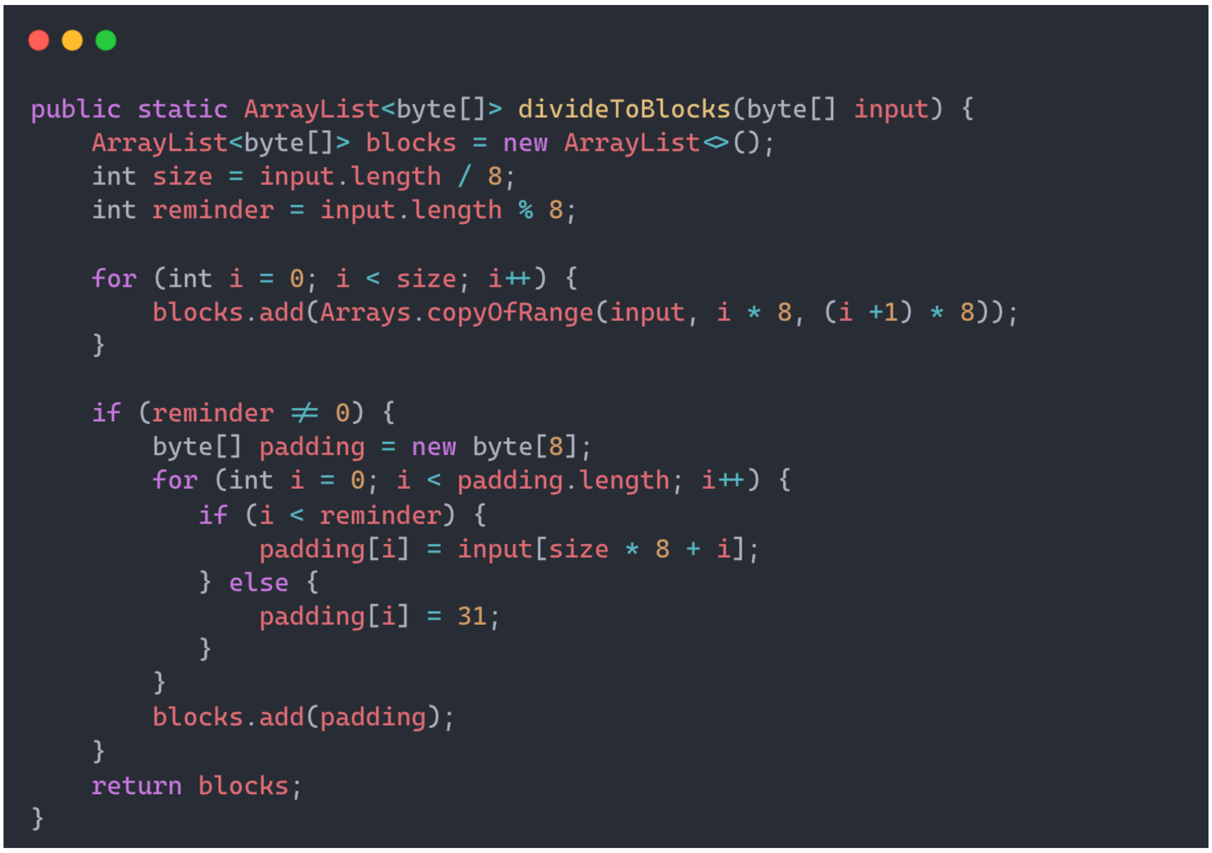
```java
public static byte[] xor(final byte[] input, final byte[] key) {
    byte[] result = new byte[input.length];

    for (int i = 0; i < input.length; i++) {
        result[i] = (byte) (input[i] ^ key[i]);
    }

    return result;
}
```

Figure 5. xor function

divideToBlocks()

In this function, we divide the byte array given as input into blocks of 8 bytes and return the blocks we have divided into a list. While doing the division, we first divide the length of the input array by 8. This value shows us how many blocks there are. At the same time, we look at the remainder of the input array length divided by 8. This value shows us how many bytes are left in the last block.

When we subtract the reminder value from the block size we have determined, we can find how many more bytes we need to add to the end of the last block. We fill the empty parts of the last block with the value corresponding to 31 in the ASCII code that we have determined. We chose this value because it is the least used ASCII character according to this answer. We can change the value we use for padding as we want.



```java
public static ArrayList<byte[]> divideToBlocks(byte[] input) {
    ArrayList<byte[]> blocks = new ArrayList<>();
    int size = input.length / 8;
    int reminder = input.length % 8;

    for (int i = 0; i < size; i++) {
        blocks.add(Arrays.copyOfRange(input, i * 8, (i +1) * 8));
    }

    if (reminder ≠ 0) {
        byte[] padding = new byte[8];
        for (int i = 0; i < padding.length; i++) {
            if (i < reminder) {
                padding[i] = input[size * 8 + i];
            } else {
                padding[i] = 31;
            }
        }
        blocks.add(padding);
    }
    return blocks;
}
```

Figure 6. Algorithm to divide blocks

byteArrayContainsPadding()

This function checks whether the value we specified for padding (ASCII 31) exists in the byte array given as input, and returns true if there is a value, otherwise it returns false.

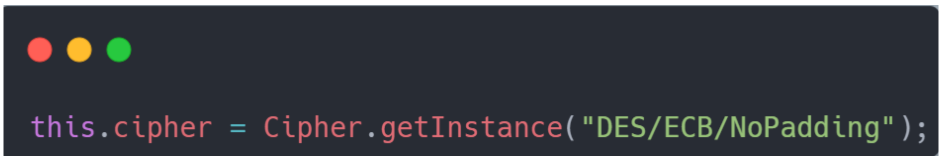
combineNonceAndCounter()

This function takes a byte array containing a 7-byte nonce and a one-byte counter value as input in its parameters. It then adds the counter value to the end of the nonce value and returns the new byte array.

DES Class

The popularity of the Data Encryption Standard (DES) has been discovered to be somewhat declining as a result of the discovery that DES is susceptible to very strong assaults. Since DES is a block cipher, it encrypts data in blocks of 64 bits each. As a result, DES receives 64 bits of plain text as input and outputs 64 bits of ciphertext.

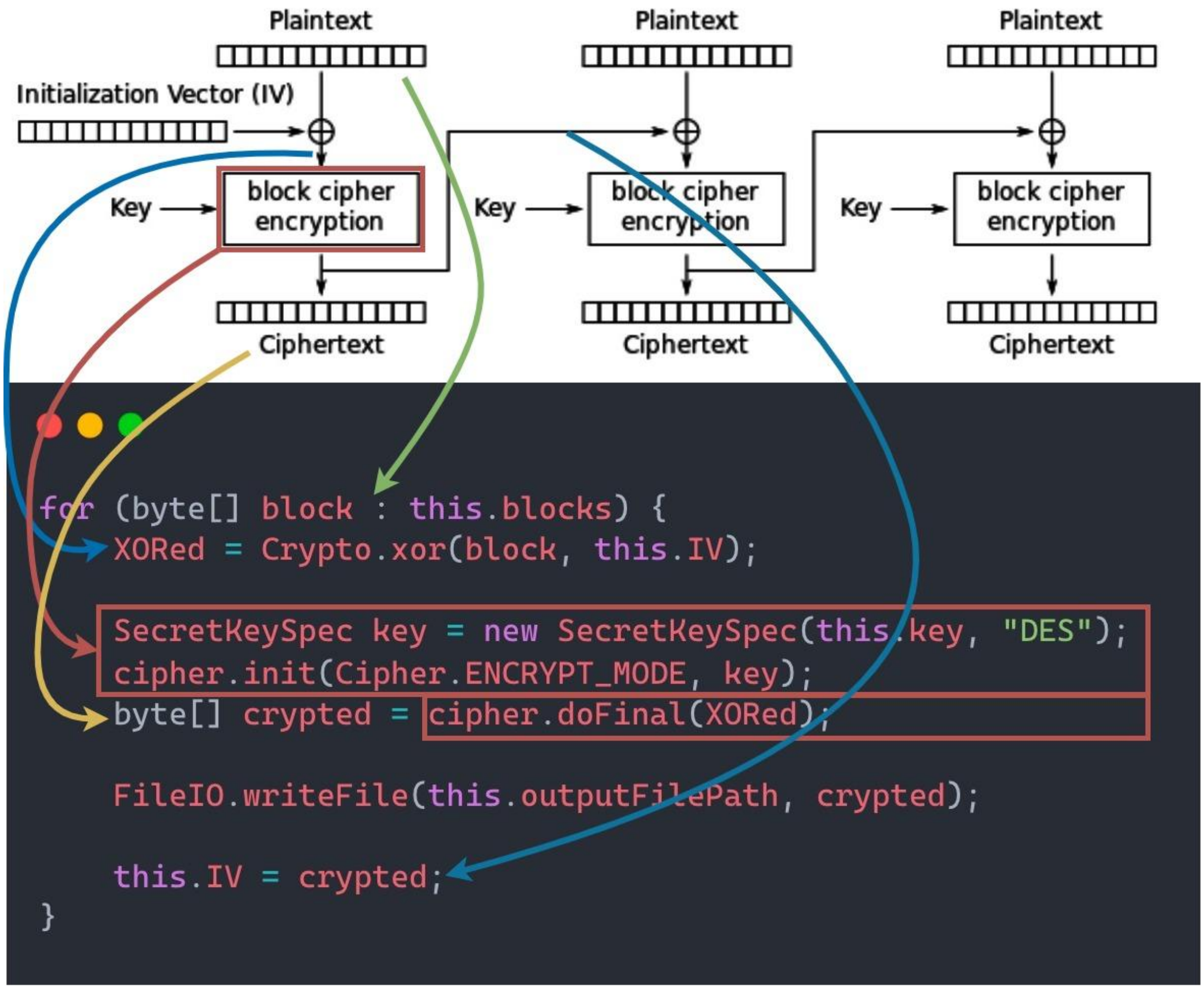
In our encryption implementation, the message that needs to be encrypted first is split into blocks of 64 bits each. Afterwards, as you can see the details below, encryption processing is applied for each block according to the characteristics of the relevant mode.



```java
this.cipher = Cipher.getInstance("DES/ECB/NoPadding");
```

Figure 7. Cipher instance which is used by all modes

Encryption and decryption operations in all modes are done with the cipher defined in the constructor, which you can see above. In our DES implementation, we first determine the last eight bytes of the key value and IV value given in the key file. You can see an example below, using the DES process which is encryption by CBC mode with explanations.Firstly each block xored with the IV output of this operation will be input of the cipher. After that output of the cipher will be a new IV for the next block and this process goes on until it's done for all blocks.



```java
for (byte[] block : this.blocks) {
    XORed = Crypto.xor(block, this.IV);

    SecretKeySpec key = new SecretKeySpec(this.key, "DES");
    cipher.init(Cipher.ENCRYPT_MODE, key);
    byte[] crypted = cipher.doFinal(XORed);

    FileIO.writeFile(this.outputFilePath, crypted);

    this.IV = crypted;
}
```

Figure 8. DES scheme and corresponding code block

THREE_DES Class

Triple DES (3DES) is built on top of the DES algorithm, which has difficulty resisting brute force attacks. 3DES is simply based on the logic of using the DES algorithm three times in a row. The fact that the consecutive DES algorithms are encryption - decryption - encryption, respectively, significantly reduces the probability of a man in the middle attack. At the same time, the key given as input in each DES stage is different, which increases the security level of the algorithm.

We decided to implement 3DES with two keys for simplicity as shown in Figure . In our 3DES implementation, unlike our DES implementation, we first determine the last eight bytes of the key value given in the key file as K1, and the first eight bytes as K2. Next, we create our cipher object as DES.

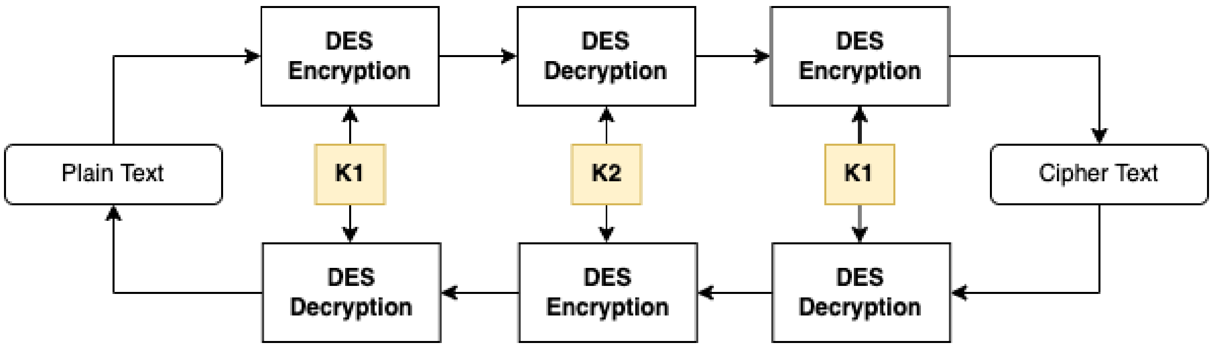
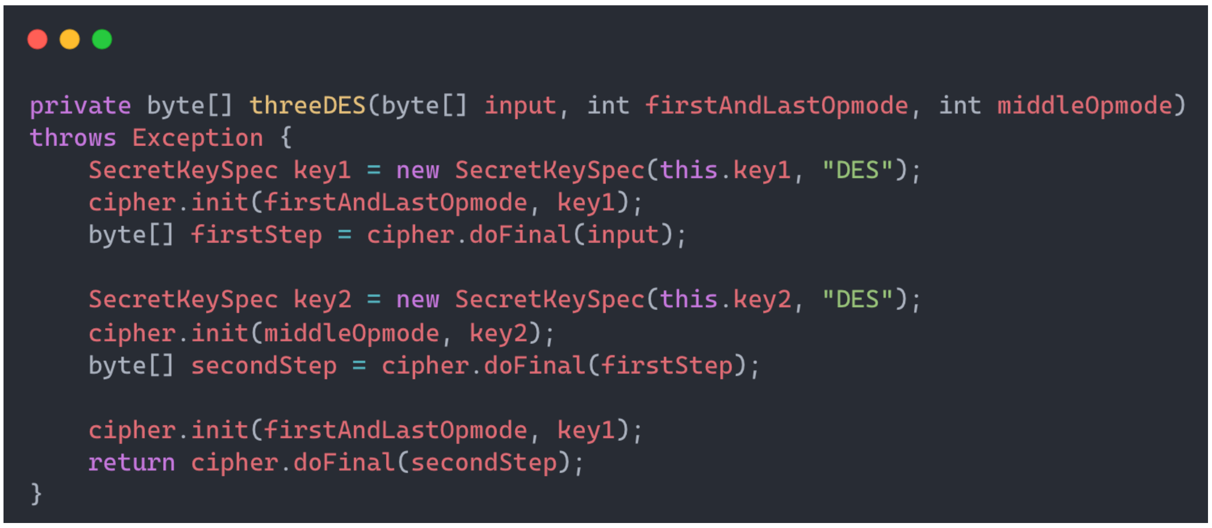
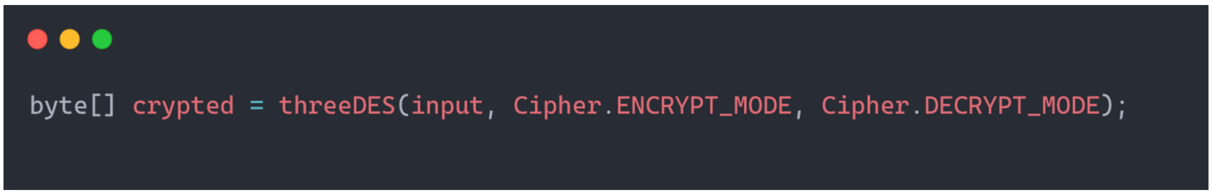


Figure 9. Triple DES scheme that we implemented

As a matter of fact, there is not much difference between using 3DES and using DES, except for the use of keys in encryption and decryption logic. Since most of the 3DES algorithm remains the same in each block mode, we created a *threeDES()* method in the *THREE_DES* class to avoid code duplication. This function is the byte array to be encrypted as input, the opcode specifying which function (enc/dec) the first DES block will perform (this opcode is also used in the last DES block), and the opcode specifying which function (enc/dec) the second DES block will perform as a parameter. gets. Specified opcodes are static integer values specified in the *Cipher* class provided by the Crypto API.

As it can be understood from the code of the method in Figure 8, it first encrypts the input byte array with K1, then gives the *firstStep* byte array from the first stage as input to the second block and decrypts it with K2. Finally, *secondStep* returns the byte array with K1 to the cipher and returns its output as a byte array. Since the cipher object returns to the default mode every time the *doFinal()* method is called, there is no harm in using the same object over and over.


```java
private byte[] threeDES(byte[] input, int firstAndLastOpmode, int middleOpmode)
throws Exception {
    SecretKeySpec key1 = new SecretKeySpec(this.key1, "DES");
    cipher.init(firstAndLastOpmode, key1);
    byte[] firstStep = cipher.doFinal(input);

    SecretKeySpec key2 = new SecretKeySpec(this.key2, "DES");
    cipher.init(middleOpmode, key2);
    byte[] secondStep = cipher.doFinal(firstStep);

    cipher.init(firstAndLastOpmode, key1);
    return cipher.doFinal(secondStep);
}
```
Figure 10. Generic 3DES algorithm to be used in all modes.



```
byte[] crypted = threeDES(input, Cipher.ENCRYPT_MODE, Cipher.DECRYPT_MODE);
```

Figure 11. Example usage of *threeDES()* method.

UML Diagram

The UML diagram of the program is as given in the figure.

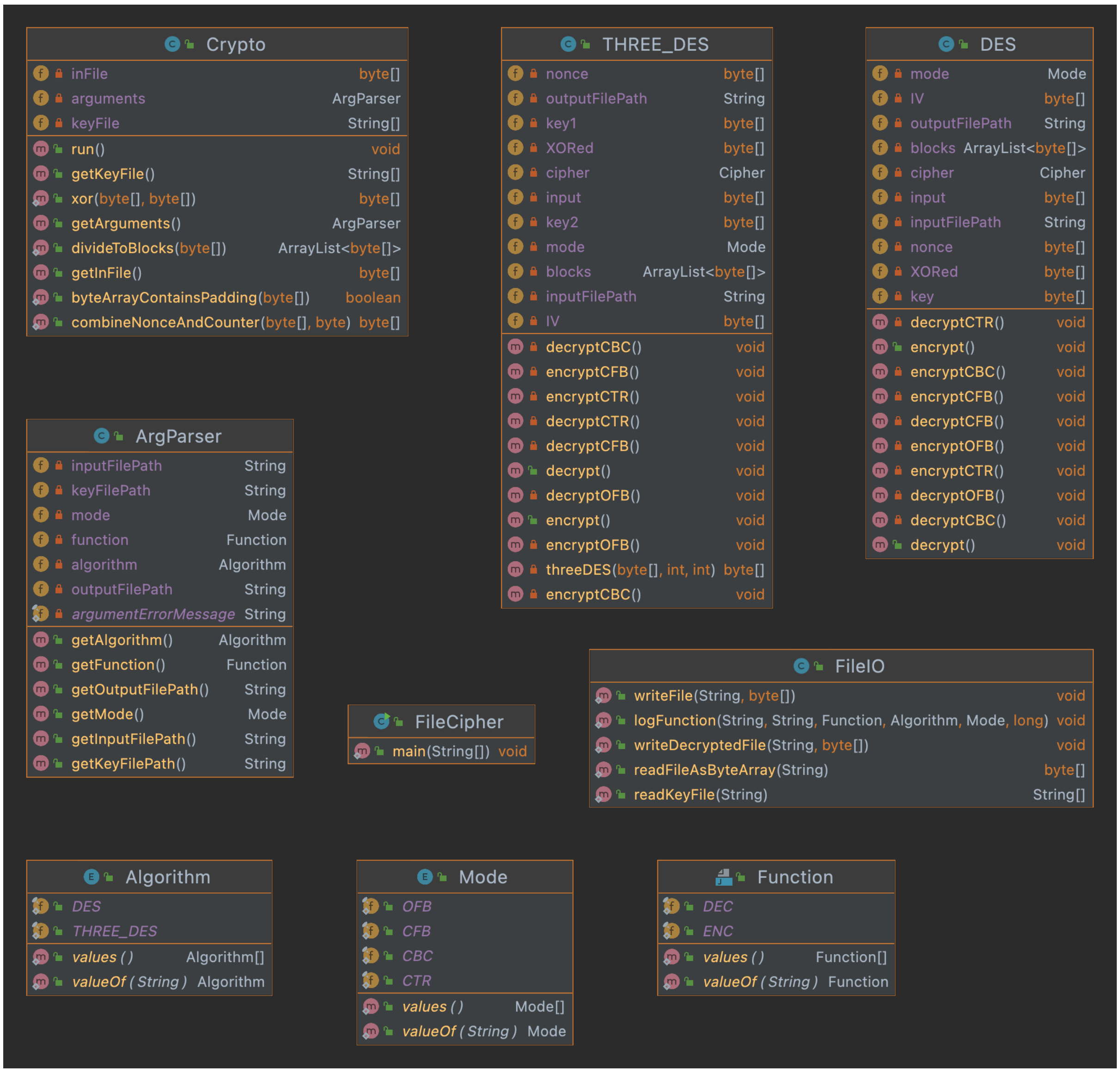


Figure 12. UML diagram of FileCipher.

IV. Results and Outcome

Algorithmic Comparison

We tested the DES and 3DES algorithms in all modes using the same input file for both encryption and decryption. To automate this test, we created a function called *runTests()*. If you wish, you can see the outputs by calling these functions in your local. In order to minimize the possibility of test results being affected by the test environment, we added the elapsed times after running the same process 100 times for each process, divided it by 100 and got the average.

The results are as seen in the graphs below. As we can see from the graphics, we can see that 3DES is a little slower at every stage, even if there is not a big time difference between DES and 3DES. After performing the tests, we found it strange that there was not much difference between the times of DES and 3DES in our first observations because 3DES was doing the same thing as DES 3 times in a row, it should have been much slower. But when we thought about it in more detail, we realized that the part of the time complexity of both algorithms is related to how many blocks are in the input. For example, if the encryption step in DES takes x ns of time, it will take 3x ns of time in 3DES. Since this encryption process will be done for each block, let's call the number of blocks N. Since the other variables are constant for both algorithms, we can say that in this case, it takes xN for DES and 3xN for 3DES. So DES O(xN), 3DES O(3xN) has big-o complexity. Since constant number values do not matter in Big-O notation, both algorithms actually have O(xN) algorithmic complexity.

In order to show the slight variation between 3DES and DES, the vertical axis values in the graph are set between 145 and 160 ms. The elapsed time values are very close to each other.

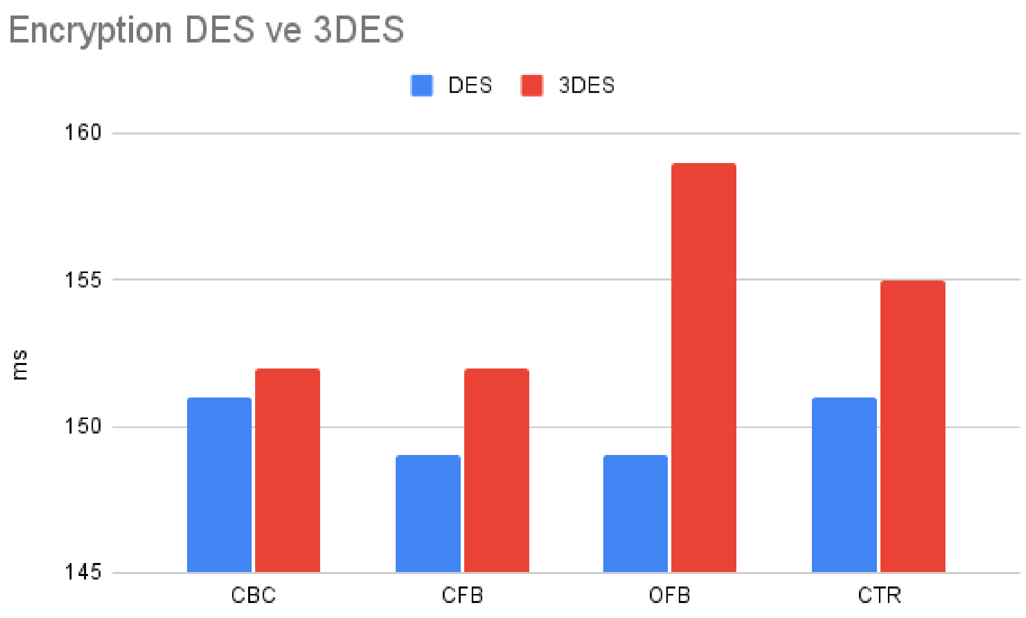


Fig 13. Time comparison of encryption process on each mode and algorithm.

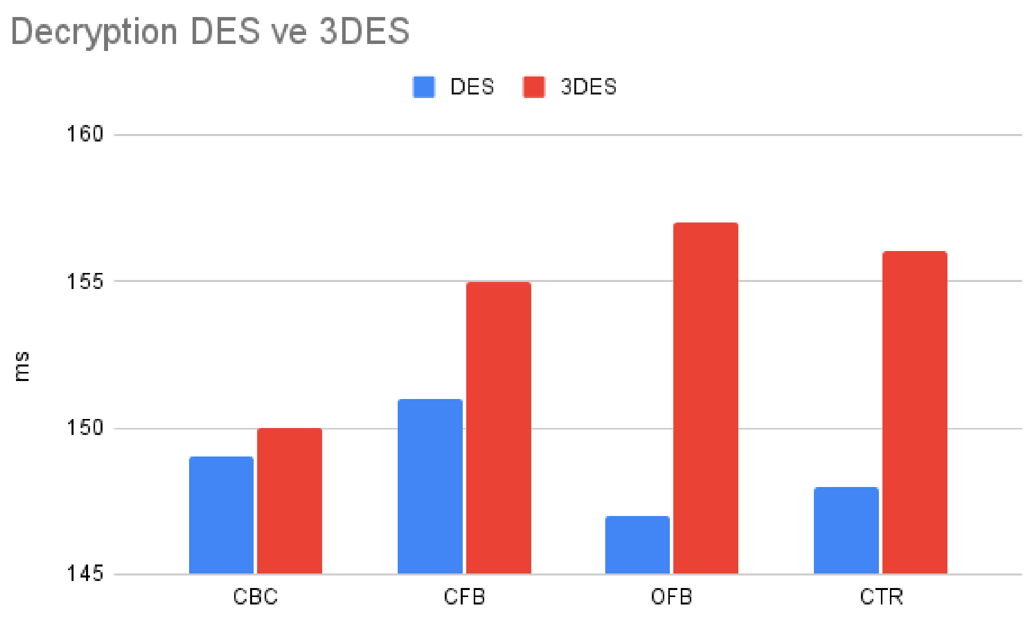


Fig 14. Time comparison of decryption process on each mode and algorithm

Learning Outcome

With the help of this assignment, we gained knowledge of the design of block ciphers, the (DES, 3DES) encryption algorithms, and finally the application of encryption mode (CBC, CFB, OFB, CTR). We were able to apply the theoretical information from the BBM463 course to programming and increase our understanding of the topics thanks to this assignment.