BLOCK CIPHERS

STUDENTS: BASTIDA PRADO JAIME ARMANDO

SOLORIO PAREDES DANIEL

PROFESSOR: DÍAZ SANTIAGO SANDRA

SUBJECT: CRYPTOGRAPHY

GROUP: 3CM6

March 20th 2019

1. LIST OF FUNCTIONS USED

This program is implemented on C++ using the library of functions Crypto++.

To store the key the program uses an object of the class 'SecByteBlock' to ensure the sensitive material is zeroized.

- void Assign (const T *ptr, size_type len)

  This function receives a pointer so the 'SecByteBlock' uses it to store the sensitive data, if '0x00' is passed to it, it creates a new array of length 'lenás specified at call. We call this function to store our key. It is called in the program with one of these two values 'DES_EDE2::DEFAULT_KEYLENGTH' or 'DES_EDE3::DEFAULT_KEYLENGTH' depending if using EDE or EEE.

```
139     SecByteBlock key;
140     byte *iv;
141     if(variant.compare("EDE") == 0)
142     {
143         key.Assign(0x00, DES_EDE2::DEFAULT_KEYLENGTH);
144         iv = (byte *) malloc(DES_EDE2::BLOCKSIZE);
145     }
```

Figura 1: Assign function

- virtual void GenerateBlock (byte *output, size_t size)

  This function generates a random array of bytes, stores them into óutput' of size 'size'. It is used when we generate the random key and IV.

```
156     auto_srp.GenerateBlock(key, key.size());
157     auto_srp.GenerateBlock(iv, sizeof(iv));
```

Figura 2: GenerateBlock function

- void SetKeyWithIV (const byte *key, size_t length, const byte *iv)

  This function sets the key and IV to the object who calls it. The object who calls this function can appear in our program as one of the following examples (there are more):

  - CBC_Mode<DES_EDE2>::Encryption e;
  - CTR_Mode<DES_EDE3>::Encryption e;
  - OFB_Mode<DES_EDE3>::Encryption e;
  - CFB_Mode<DES_EDE2>::Encryption e;

  Varying on the mode of operation we want and the variant of 3DES.

```
184         try
185         {
186             CBC_Mode<DES_EDE2>::Encryption e;
187             e.SetKeyWithIV(key, key.size(), iv);
```

Figura 3: SetKeyWithIV function

- StringSource (const std::string string, bool pumpAll, BufferedTransformation *attachment=NULL)

  The call of this constructor creates a 'StringSource' . A 'StringSource', is an abstraction used by Crypto++ to handle better the manipulation of the data across the different functions of the library. The 'string' parameter can be thought as a source from which the data will flow to the 'BufferedTransformation' , the destination. This 'BufferedTransformation' can be a file or a string (among others), this destination/end point is called 'Sink' in the abstraction of the library.

```
    StringSource ss_plaintext(plaintext, true, new
        StreamTransformationFilter(e, new StringSink(ciphertext), BlockPaddingSchemeDef::PKCS_PADDING));
}
```

Figura 4: StringSource constructor

- StreamTransformationFilter (StreamTransformation c, BufferedTransformation *attachment=NULL, BlockPaddingScheme padding=DEFAULT_PADDING)

  The call of this constructor creates another abstraction of Crypto++. This time a 'StreamTransformationFilter' takes a 'BufferedTransformation' basically an object of the class 'CBC_Mode<DES_ED
  e;' or any other similar as mentioned above, a 'Sinkóbject (i.e. a string or file) and a 'BlockPaddingScheme' better known as only Padding. With all those parameters it enciphers and adds padding according to the class of é' and the type of padding specified to finally store the result into the 'Sink' object. The type of padding used by CBC and CTR in our program is 'PCKS_PADDING' , OFB and CFB don't use padding.

```
StreamTransformationFilter(e, new StringSink(ciphertext), BlockPaddingSchemeDef::PKCS_PADDING));
```

Figura 5: StreamTransformationFilter constructor

- Base64Encoder(BufferedTransformation *attachment = NULL, bool insertLineBreaks = true, int maxLineLength = 72)

  The call of this constructor takes a source 'Sink"BufferedTransformation' (i.e. a file or string) and encodes it using base 64. It adds by default a line break every 72 chars, if you want you can avoid it by passing false to the second argument.

```
321     StringSource ss_key(key_string, true, new Base64Encoder(new FileSink(key_fn)));
```

Figura 6: Base64Encoder constructor

2. GRAPHS

We proved the program with different file sizes: 500kb, 1MB, 5MB and 10MB; variants: EEE, EDE and modes of operation: CBC, CTR, OFB and CFB.

2.1. EDE

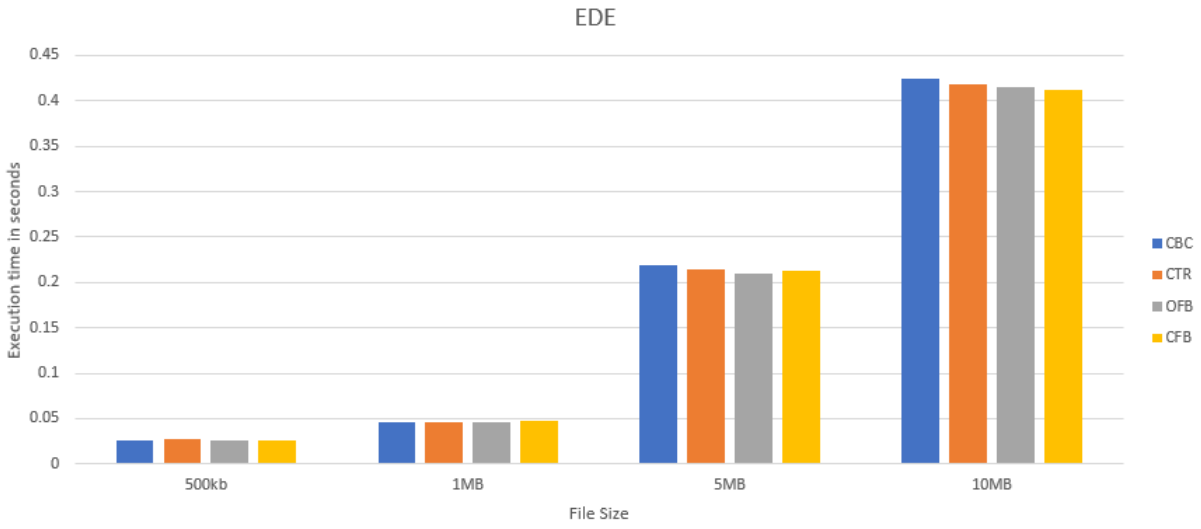

Figura 7: EDE graph comparison

2.2. EEE

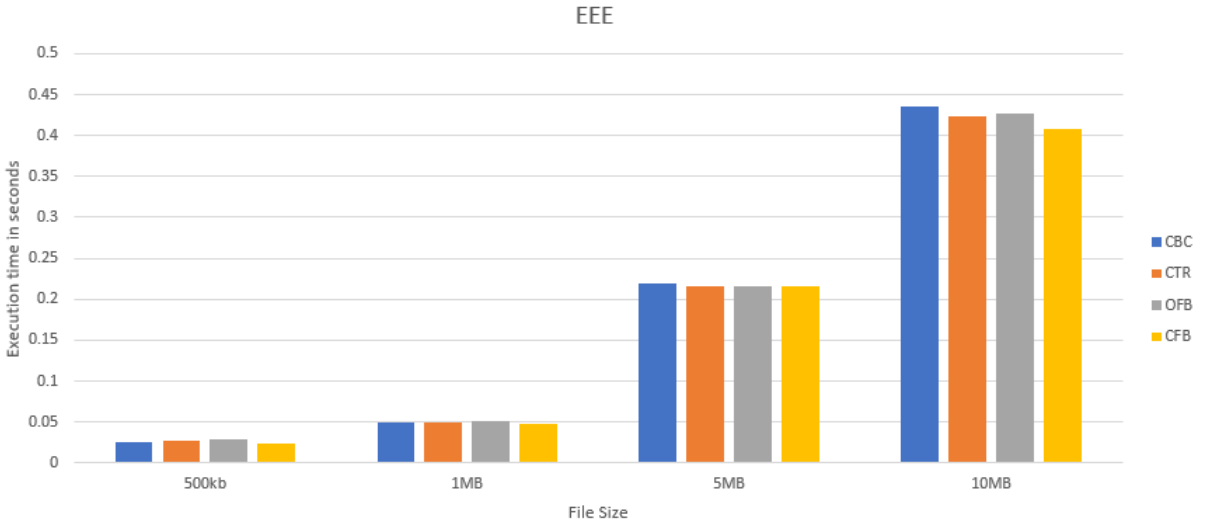

Figura 8: EEE graph comparison

As we can see there is no significant difference between any of the variant of 3DES nor any mode of operation. However, analyzing the chart minutely we can that the overall performance of CBC stands out as the most slowly while the rest of the modes behave quite similarly, standing out for just a little the CFB mode as the fastest.

3. CONCLUSIONS

For a very little difference the mode of operation CFB with EDE is the one who has the better performance according to the data we gathered from our program. Although is worth mention that every time the process is run the times of execution change a little bit, this could lead to another choice in which is the best at performance since every mode of operation behaves very closely to the other.