Network security: Assignment 2

Prepared by

Emran Altamimi

201510662

Ea1510662@qu.edu.qa

A hands on assignment on encryption mechanisms

11/3/2023

Task 1: Modify the files DES_Demo.py and AES_Demo.py to change the text input, and run both files to test both ciphers with different input. Report the output of the text/images you test with.

DES_Demo.py

=========DES(small Msg, ECB)=========

PlainText : b'This is a different test input for DES_Demo.py for task 1 in the assignment' ==In Binary>>

01010100 01101000 01101001 01110011 00100000 01101001 01110011 00100000 01100001 00100000 01100100 01101001 01100110 01100110 01100101 01110010 01100101 01101110
01110100 00100000 01110100 01100101 01110011 01110100 00100000 01101001 01101110 01110000 01110101 01110100 00100000 01100110 01101111 01110010 00100000 01000100
01000101 01010011 01011111 01000100 01100101 01101101 01101111 00101110 01110000 01111001 00100000 01100110 01101111 01110010 00100000 01110100 01100001 01110011
01101011 00100000 00110001 00100000 01101001 01101110 00100000 01110100 01101000 01100101 00100000 01100001 01110011 01110011 01101001 01100111 01101110 01101101
01100101 01101110 01110100

64-bit Key : b'U\x04a\xf5Y\x19m\xb2'

DES cipherText :
b'\x1b\xde\x13\xe0\xc9\xa8\xd5\x9d6\x0b\xc4>\xf3\xea^K\x87\xd3\x88vCR\x99tI\xb4\x10\x7fg\xc9\xd2*\xc1\xa0P\xa4M,\x80V\xda\x11\xdacCI\xa7<\xfe\x08\xf3Lz\xfa%\x13\xae\xa3\xb67\xc8\x17M\x17\xb2\xc1\x1d:\x1d\xba\xc1\xd6\x15\xc6 \xf2\x97\xb1I"'

Dycrpted cipher Text : b'This is a different test input for DES_Demo.py for task 1 in the assignment'

=========DES (Long Msg, CBC)=========

64-bit Key : b'U\x04a\xf5Y\x19m\xb2'

DES cipherText :
b'\xbc\x17\x1d=\x18g\xfe*\xc2o\xc6\x03\x9b3\xf7\xf2F\x17\xfd\x1e\xe9X\t\x8aR\xd2&\xdf\xa3\x19\t\xf2\xa3\xd0\x8b\xce2\xf86B\xc5\x90-\xc1\x9b@Spv=\xe3Q\xde\xf8Q\x8c\x80[\\=)\xe9\xed\xee\xad\x82\xed\x98\xd9/~>\x84\xba\xf5T\xea\xc4\xcf'

Decrypted cipher Text : b'This is a different test input for DES_Demo.py for task 1 in the assignment'

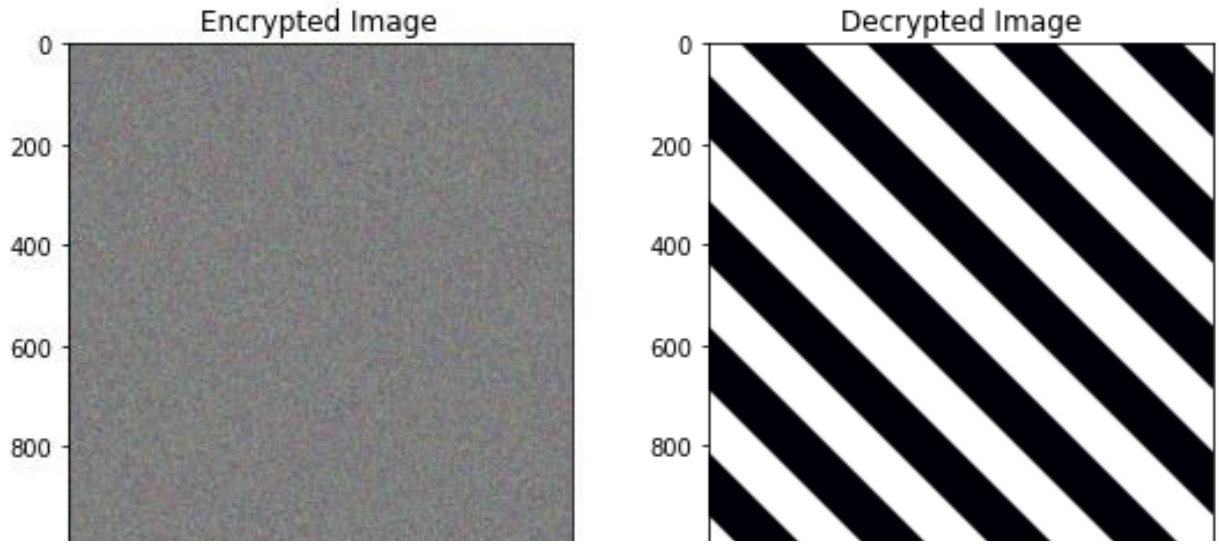

Figure 1 DES image encryption example

AES_Demo.py

=========AES-128 (small Msg, ECB)=========

PlainText : b'This is a different test input for AES_Demo.py for task 1 in the assignment' ==In Binary>>

01010100 01101000 01101001 01110011 00100000 01101001 01110011 00100000 01100001 00100000 01100100 01101001 01100110 01100110 01100101 01110010 01100101 01101110 01110100 00100000 01110100 01100101 01110011 01110100 00100000 01101001 01101110 01110000 01110101 01110100 00100000 01100110 01101111 01110010 00100000 01000001 01000101 01010011 01011111 01000100 01100101 01101101 01101111 00101110 01110000 01111001 00100000 01100110 01101111 01110010 00100000 01110100 01100001 01110011 01101011 00100000 00110001 00100000 01101001 01101110 00100000 01110100 01101000 01100101 00100000 01100001 01110011 01110011 01101001 01100111 01101110 01101101 01100101 01101110 01110100

128-bit Key : b'\x99\xc0\xb3\x1d\xbb\xa4\xd5\xb7\xa2\xeb\xdd3\xe5\xb5\\\xc4'

AES-128 cipherText :
b'\x0cXK\x059\x8b~\x82\xfc\x8a\x06h+\xef\xc6\xa0Z\xf0/\xd5y\x9c\xb5nO\x92\xc5T\xfe\xf2\x01\xa0[\xdf\x8e\xce\xba\xbe%\x8a\xbc\xbc\xcd\xe7~?cfT\xd7\xa3\xd2\xad\x07\x80\x0e\xd3&0\xea\xff\xb6PC\r#\xc8\x80\x14`\xc6\xe9.\x81/\xa9\x87&\xe4\xf4'

Decrypted cipher Text : b'This is a different test input for AES_Demo.py for task 1 in the assignment'

=========AES-192 (Long Msg, CBC)=========

192-bit Key : b'P\xc1\x14\xe46%\x10\xc9z\xd7X\xa3\xc5\x00b\xc5\x84Vo\xbd\xbd\x95\xd2\xf8'

AES-192 cipherText :
b'\xb70\xd6\xda\x12\x96\x993k\xc25P\x85\xc0\xbe\xa8\xf3\n\xf61\xd1\xaa\t\x10\xe8\xfd\x07\xea7u\x8f[7\xd67)\xaeW\xb9\x9b82\xce\xad\xc6\x18\x854\xc6\xe7\xb3\xbf\xd3&\xdc\xf78\xd1}o\xda;\xe9\xba\xbe\xd1\xc1\xca](5\xa0NkWY\xf1(\xb2\x06'

Decrypted cipher Text : b'This is a different test input for AES_Demo.py for task 1 in the assignment'

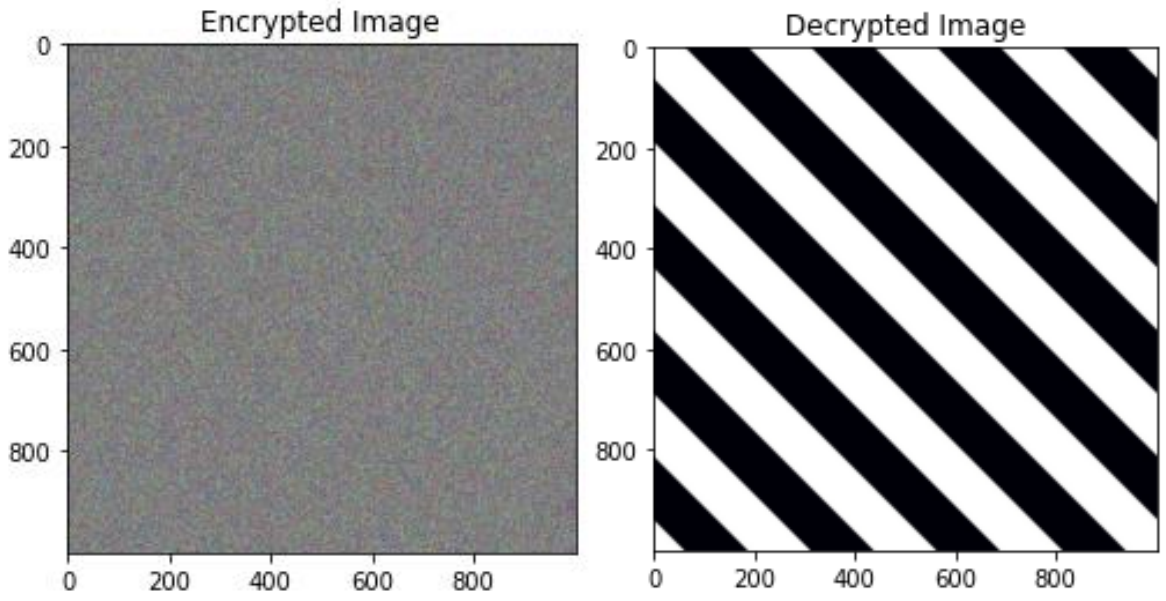

Figure 2 AES image encryption example

Task 2: Using the DES,py and DES_Demo.py files, create two files DES3.py and DES3_Demo.py to create the triple DES cipher/decipher, according to figure 2.3 in the textbook. Test the implemented algorithm with arbitrary text similar to what you did for DES and AES. Report your results.

DES3_Demo.py

=========DES(small Msg, ECB)=========

PlainText : b'This is a test input for DES3 similar to task 1' ==In Binary>> 01010100 01101000 01101001 01110011 00100000 01101001 01110011 00100000 01100001 00100000 01110100 01100101 01110011 01110100 00100000 01101001 01101110 01110000 01110101 01110100 00100000 01100110 01101111 01110010 00100000 01000100 01000101 01010011 00110011 00100000 01110011 01101001 01101101 01101001 01101100 01100001 01110010 00100000 01110100 01101111 00100000 01110100 01100001 01110011 01101011 00100000 00110001

64-bit Key : [b'\xc6\xa9\xf0Y\x9d\xb2\xb1\xf4',

b'B\x8f\xc4\x81\xbb\xca\xb2F',

b'\xb7\xbf\xa4\xae\x81b\x8e\r']

DES cipherText : b'\xe4@\xa5X\xf5\xc2\x8e\xd5&\xe42\xf5\x81 i\x83\xa0\x83\x92\xca!\xea\x01c;X\x11\xa0\x13~\x8c#~\x87\x11\xed\x8f\x95\xfc\xd0\xfa\xf0;L\xf2\x1 b=\x0f'

Decrypted cipher Text : b'This is a test input for DES3 similar to task 1'

=========DES (Long Msg, CBC)=========

64-bit Key : [b'\xc6\xa9\xf0Y\x9d\xb2\xb1\xf4',

b'B\x8f\xc4\x81\xbb\xca\xb2F',

b'\xb7\xbf\xa4\xae\x81b\x8e\r']

DES cipherText : b"\xe1\x10\xef\x91#\x82p_\x8ai\x05A\n\xe9\xd2b~Q\x84\xa9\xb2\xb6\x98[\x07'\x15nj\xfb\x81\xa3I\ xd4^8\x8e\x87!\x99\xac\xe4\xd4\xab\t\xac\xd4J"

Decrypted cipher Text : b'This is a test input for DES3 similar to task 1'

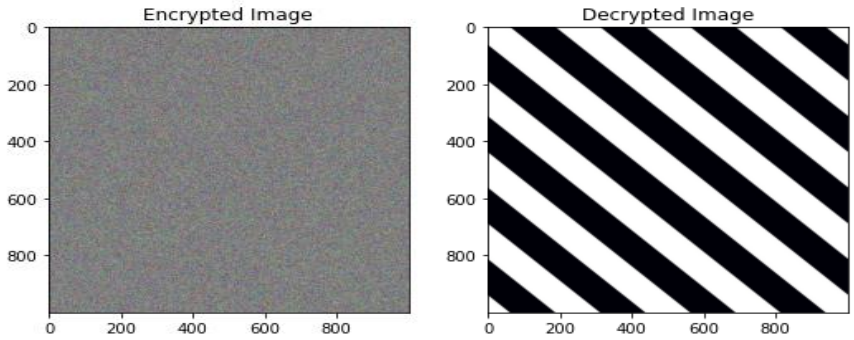

Figure 3 DES3 image encryption example

Code for DES3.py :

The DES3 class in the DES3 module simply inherits the DES class from the DES module then implements it's own encrypt_block and decrypt_methods.

The Methods are shown below:

```
def encrypt_block(self, plaintext):
    cipherText1 = self.d1.encrypt_block( plaintext)
    decryptedCipherText1WdifferentKey = self.d2.decrypt_block(cipherText1)
    cipherTextFinal = self.d3.encrypt_block(decryptedCipherText1WdifferentKey)
    return cipherTextFinal
def decrypt_block(self, ciphertext):
    plainText1 = self.d3.decrypt_block(ciphertext)
    encryptedPlainTextWithDifferentKey = self.d2.encrypt_block(plainText1)
    plainTextFinal = self.d1.decrypt_block( encryptedPlainTextWithDifferentKey)
    return plainTextFinal
```

d1, d2, and d3 are simply three DES objects from the DES class that are initialized with three different keys.

Task 3: Modify the RC4.py file to fill the encrypt function as explained in the function. Create a file RC4_Demo.py file and test the cipher with different input. Describe how the decrypt function can be done? Report your output.

RC4_Demo.py

========= RC4=========

PlainText : b'This is a test input for RC4 similar to task 1' ==In Binary>> 01010100 01101000 01101001 01110011 00100000 01101001 01110011 00100000 01100001 00100000 01110100 01100101 01110011 01110100 00100000 01101001 01101110 01110000 01110101 01110100 00100000 01100110 01101111 01110010 00100000 01010010 01000011 00110100 00100000 01110011 01101001 01101101 01101001 01101100 01100001 01110010 00100000 01110100 01101111 00100000 01110100 01100001 01110011 01101011 00100000 00110001

64-bit Key : b'\xd8\x89T]\xf6Hv\xc1'

DES cipherText : b'
y\x05IE$\xad\n\x95\x9bP\x94\xe2c:e\x9br\xfb\xfc\x8dF\xb5/\xd9\xf3\xa2\t*32\xf8?\xf9\x86g\xb9\xf7
W\x99\x96\x8c;\x07\xd7\xcf'

Decrypted cipher Text : b'This is a test input for RC4 similar to task 1'

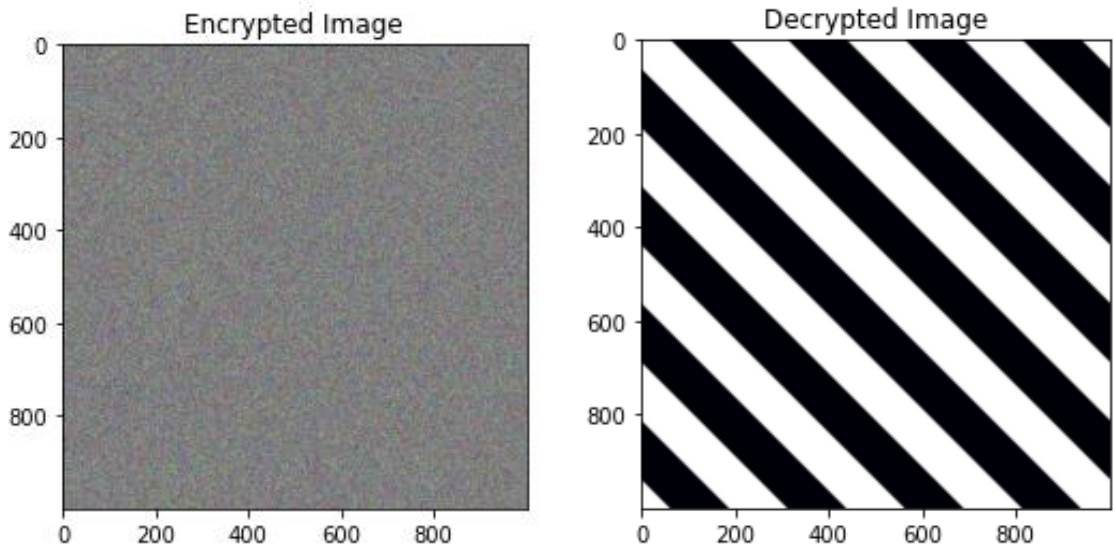

Figure 4 RC4 image encryption example

Code for RC4.py

```
def encrypt(self, plaintext):

    plaintext = strToBytes(plaintext)

    return b''.join([bytes([a ^ b]) for a, b in zip(plaintext, self.PRGA(len(plaintext)))])

def decrypt(self, ciphertext):

    return b''.join([bytes([a ^ b]) for a, b in zip(ciphertext, self.PRGA(len(ciphertext)))])
```

The above code simply XOR's a plain text of arbitrary length with the keystream produced by the given PRGA method.

Task 4: Modify the utils.py file to fill the function channelError, to generate a random noise with the same length as the input, then add the noise to the input to simulate an error according to the probability of error prb parameter. Test your code for an arbitrary input text. Report your results.

From utils.py import channelError

From utils.py import task4 #to test the channel error on text

Input: task4(b'this is a sample long text to test the channel error function effect on texts', 0.01)

Original text: b'this is a sample long text to test the channel error function effect on texts'

Noisy text: b't(is is a sample long uext to test the$ch!nnel evror function effect /n texts'

Code channelError

The function places specific amount of 1's in a noise array randomly and XOR's it with the input.

```
def channelError(input, prb,blocksize = 16):

   #write the code here to add noise the input such that the noise array will have 1's with a probability
prb

   # the input is an array of bits

   inputLenght = len(input)

   numberOfOnes = int( inputLenght * prb)

   noise = [0] * inputLenght

   indexesOfNoise = sample(range(inputLenght), numberOfOnes)

   for i in indexesOfNoise:

      noise[i] = 1

   return [x ^ y for x, y in zip(noise, input)]
```

Task5: Modify the main.py file by filling the compareCHERR_DES_AESx() function, to simulate the transfer of the monalisa200.jpg image over the error channel using the DES, AES-128, AES-192, and AES-256 cipher. Calculate the pixel error rate (see the PixelErrorRate function in the utils file) of the recovered image for channel error rates of 10e-4, 10e-5, and 10e-6. Also, calculate the time complexity for the encryption in each case. Report the recovered images for each case, a figure showing the pixel error rate for all four ciphers vs the channel error rates, and a figure showing the time complexity vs the channel error rates.

From main.py import compareCHERR_DES_AESx

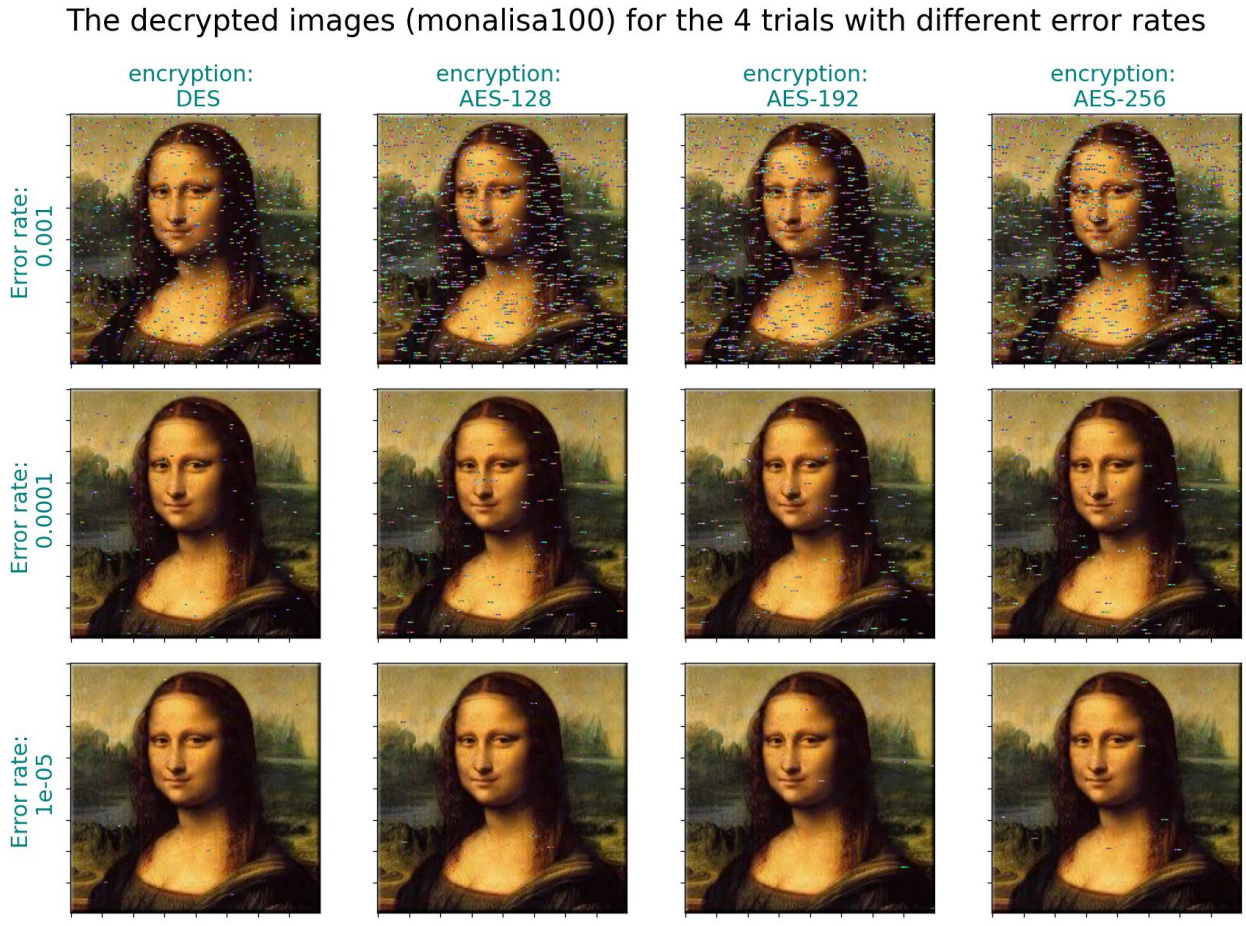

Figure 5 The effect of noise on images quality is depicted

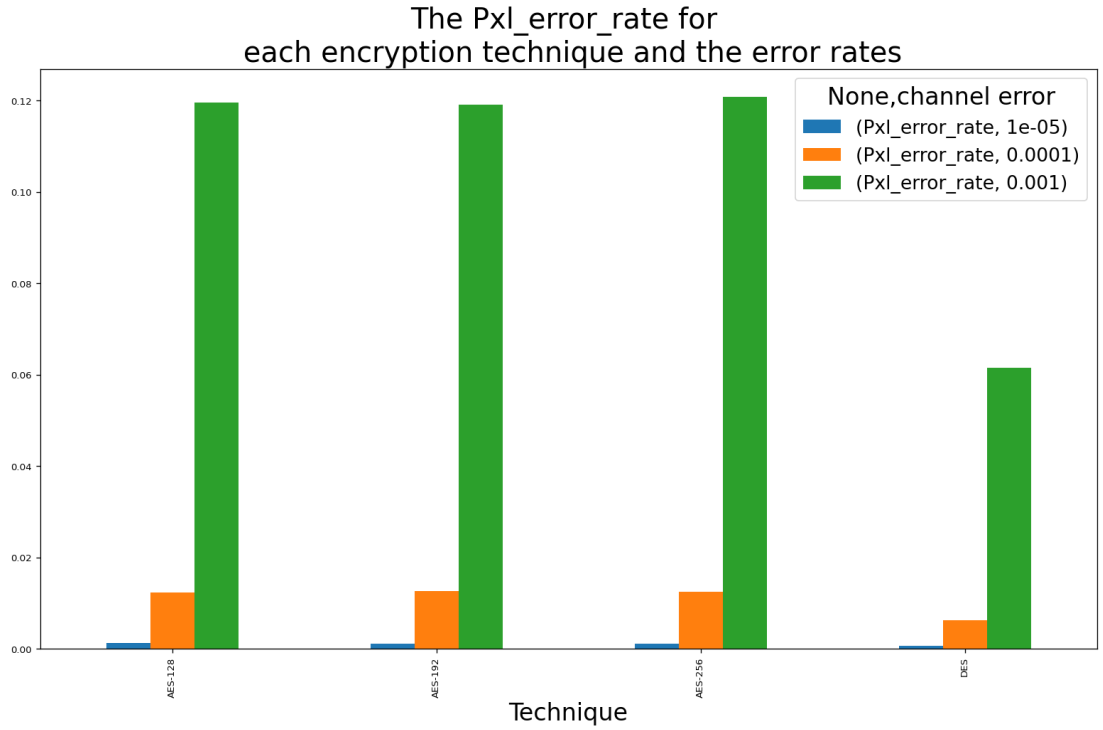

Figure 5 The pixel error rate using the utils.pixelErrorRate function

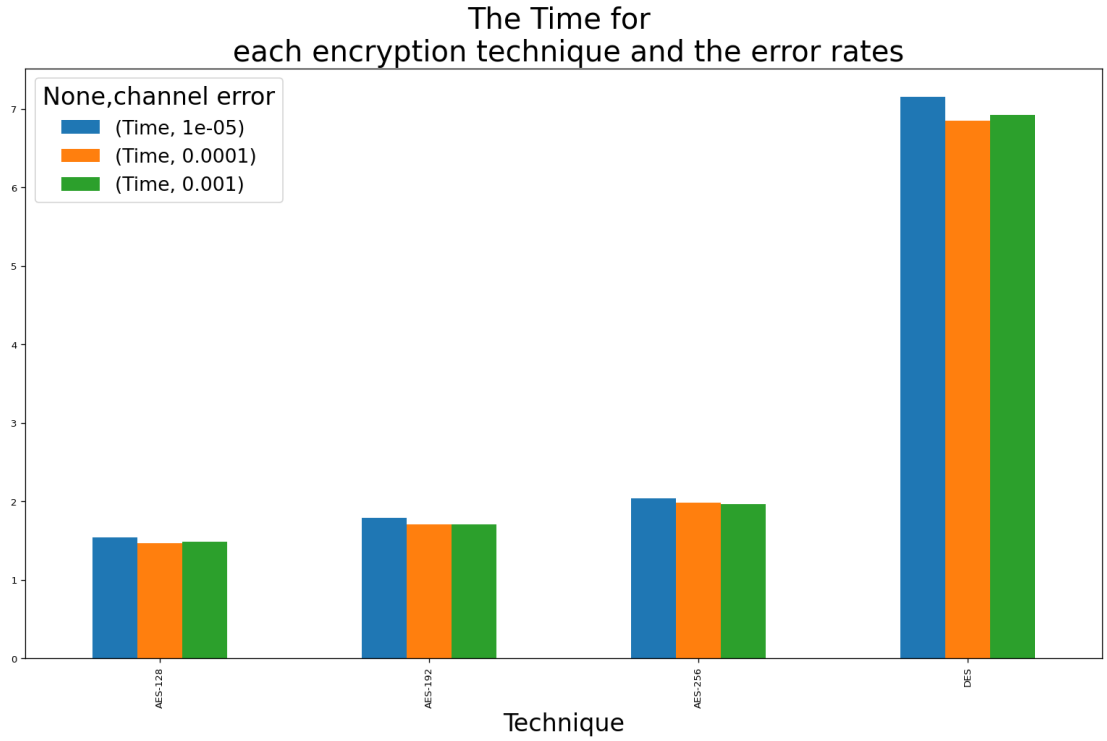

Figure 6 The total time to encrypt, decrypt and communicate the image

Comments on results for task 5:

The fact that the pixel error rate is almost similar for all AES encryption techniques can be attributed to the fact that the block size and the encryption algorithm remain the same regardless of the key length used. Therefore, increasing the key length does not propagate the noise further, as the encryption process still operates on the same blocks of data. Additionally, AES is designed to be resistant to attacks that take advantage of statistical patterns in the plaintext, which means that the noise introduced by the channel is less likely to affect the overall encryption process.

DES had a significantly better pixel error rate than AES, because DES is a block cipher with a smaller block size (64 bits) than AES (128 bits). This means that errors in a single block of AES encryption can affect twice as many pixels as errors in a block of DES encryption. Additionally, DES has a simpler structure than AES, which may make it more resistant to errors introduced by the noisy channel.

It is expected that there will be a slight increase in the encryption and decryption time with increasing key length for AES. This is because the longer the key length, the more rounds the AES algorithm has to perform to achieve the same level of security as a shorter key length. More rounds mean more computations, which can lead to a longer encryption and decryption time.

DES had a significantly higher time complexity. This could be due to the fact that DES has a more complex key generation process compared to AES. DES also uses a larger number of rounds (16 rounds) compared to AES-128, which only uses 10 rounds. The larger number of rounds means that DES requires more computation for each encryption and decryption operation, which could result in higher time complexity.

Task 6: Modify the main.py file by filling the compareIMGQLTY_DES_AES_RC4() function, to simulate the transfer of all four monalisa images with different resolutions (i.e. 100 x 100, 200 x 200, 300 x 300, and 400 x 400) images using the DES, DES3, AES-128, and RC4 cipher. Calculate the time complexity for each case. Report the recovered images for each case, and a figure showing the time complexity vs the channel error rates.

From main import compareIMGQLTY_DES_AES_RC4

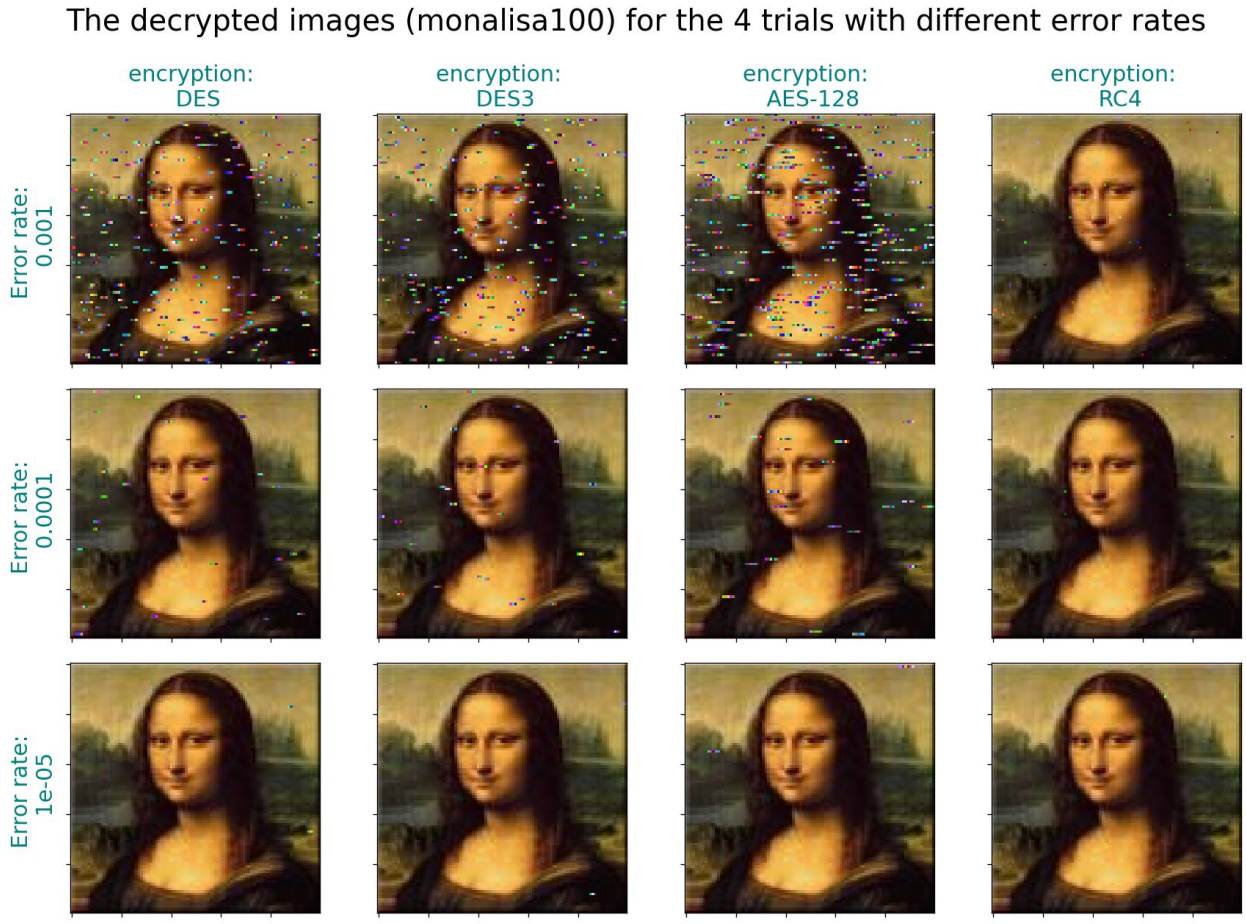

Figure 6: The monalisa image of size 4Kbytes with different encryption mechanisms and error rates

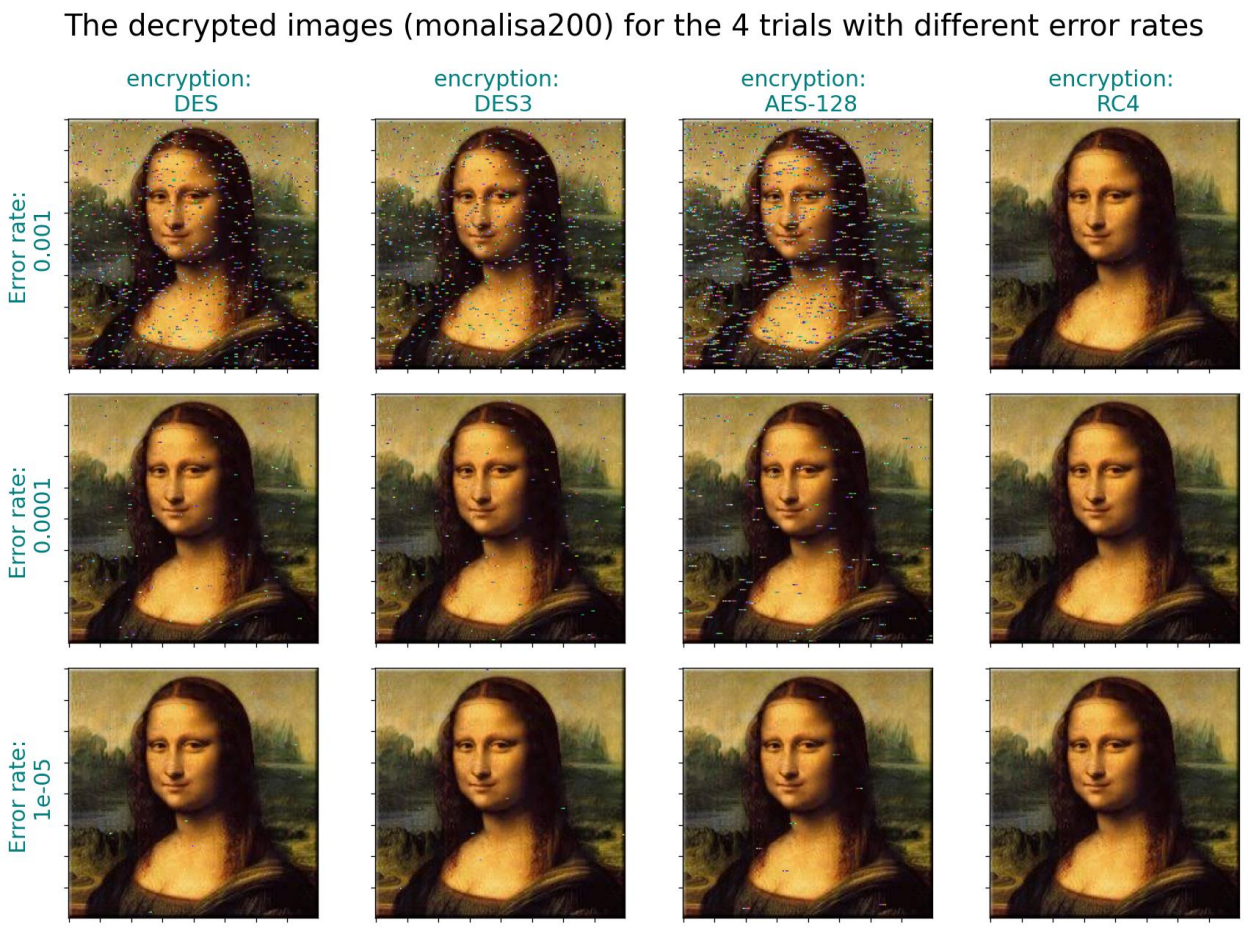

Figure 7: The monalisa image of size 8Kbytes with different encryption mechanisms and error rates

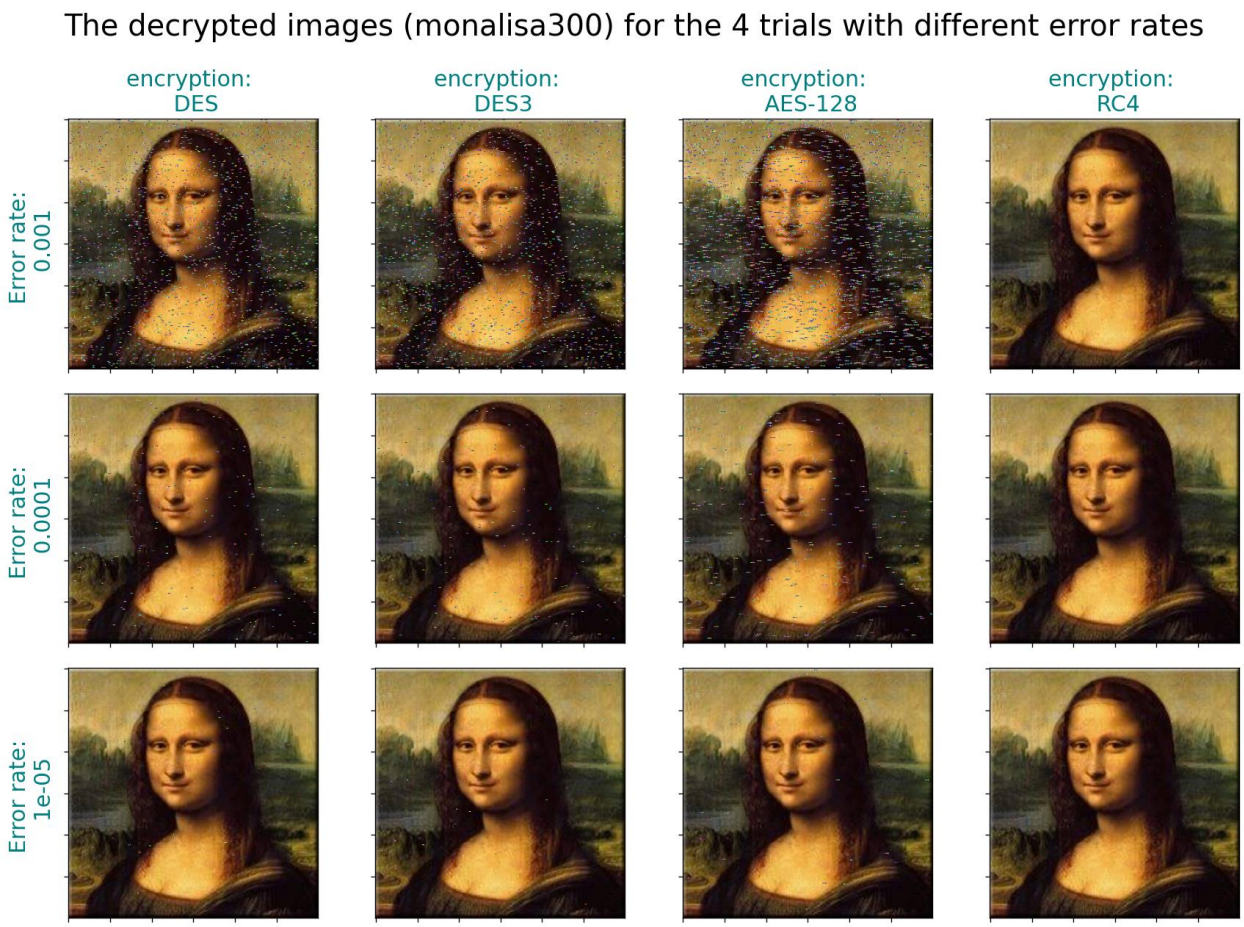

Figure 8: The monalisa image of size 16Kbytes with different encryption mechanisms and error rates

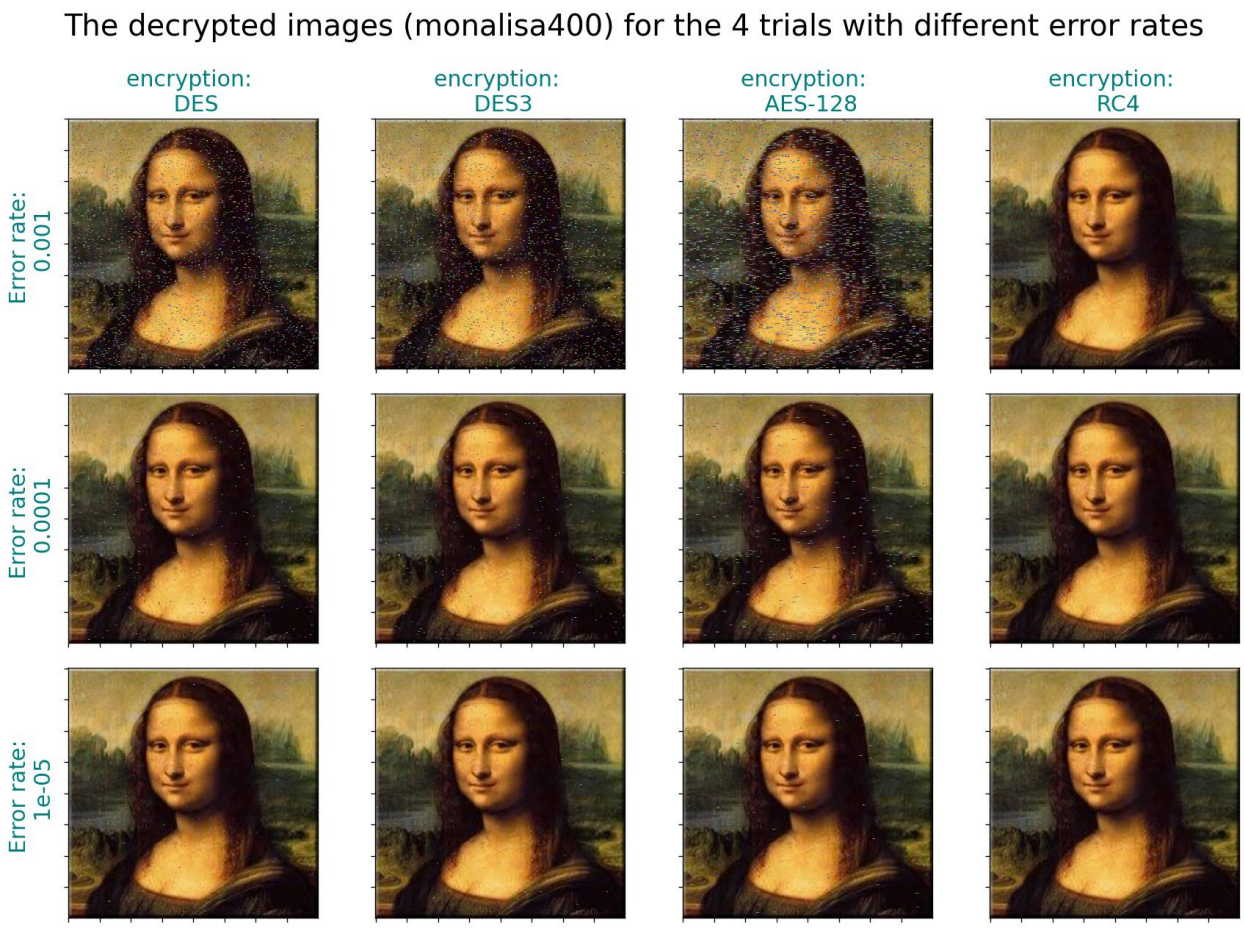

Figure 9: The monalisa image of size 20Kbytes with different encryption mechanisms and error rates

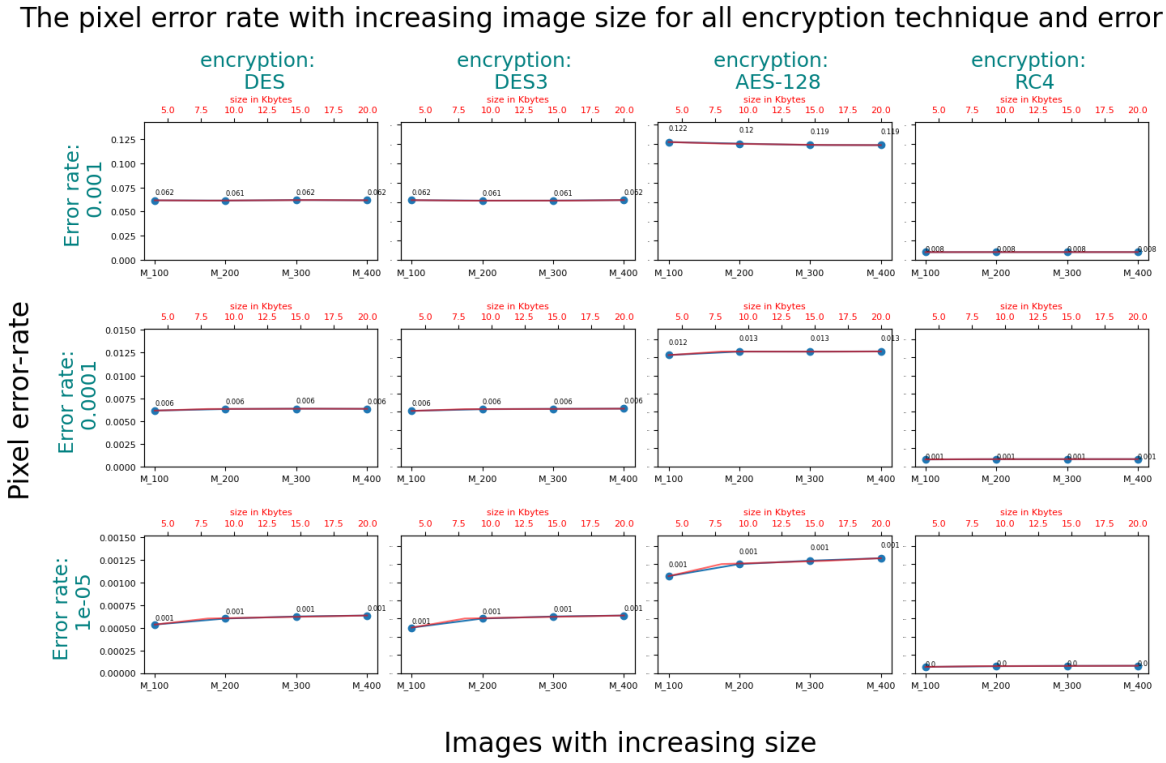

Figure 10 The pixel error rate vs the monalisa images with different encryption mechanisms and error rates

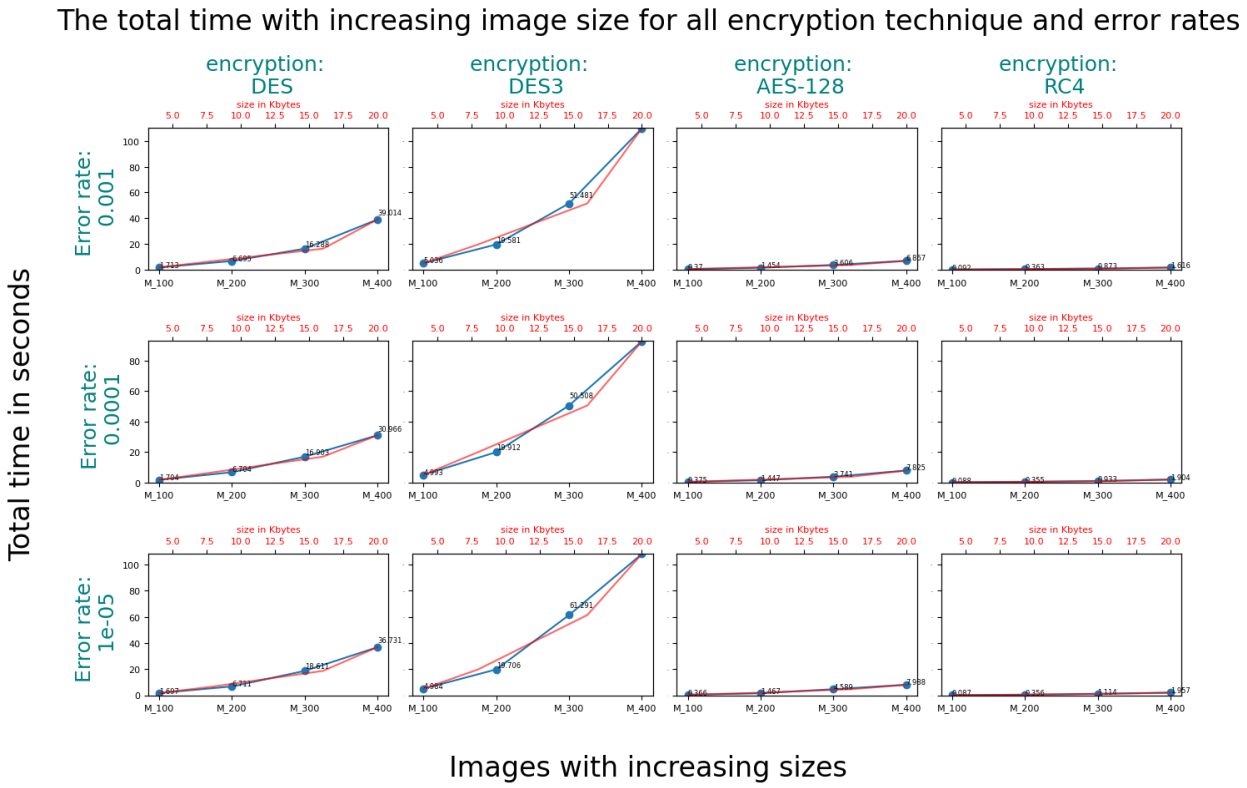

Figure 11The time complexity vs the monalisa images with different encryption mechanisms and error rates

Comments on results

The fact that the pixel error rate didn't change with increasing image size suggests that the error rate of the noisy channel is the main factor impacting the error rate of the encrypted data. This makes sense since the noisy channel is introducing errors at a fixed rate, regardless of the size of the image being transmitted.

One possible reason why 3DES did not propagate errors more than DES is that errors introduced in one block of the encrypted data may not necessarily affect the decryption of other blocks. In ECB mode, each block is encrypted and decrypted independently of the others, so errors in one block should not impact the decryption of other block/s.

Another factor that may contribute to the similar pixel error rate of DES and 3DES is the way in which errors are introduced by the noisy channel. Since the errors are randomly distributed across the encrypted data, then it's possible that they affect both DES and 3DES equally.

The fact that AES had a higher pixel error rate than DES and triple DES is somewhat expected, as AES is generally considered to be a more complex and computationally intensive encryption algorithm. This may lead to a higher likelihood of errors in the encrypted data, particularly in the presence of a noisy channel/s.

It is not surprising that RC4 had the lowest pixel error rate of all the encryption techniques tested. RC4 is a stream cipher that encrypts data byte-by-byte, rather than in blocks like the other algorithms. This

means that errors in the encrypted data will only affect individual bytes, rather than whole blocks, which can result in fewer errors overall.

The fact that the noisy channel error rate did not affect the encryption and decryption time is expected. The error rate only affects the accuracy of the transmitted data, but it should not impact the speed of the encryption and decryption process.

It is not surprising that Triple DES is slower than DES since it involves encrypting the data three times instead of just once.

AES is much faster than DES and scales better and this is attributed to different reasons including

-Improved algorithm design: AES was designed specifically to be more efficient than DES. The designers of AES used a variety of techniques to optimize the algorithm, including table lookups, sub-byte transformations, and mix-columns operations. These optimizations make AES faster and more efficient than DES.

- Block size: AES uses a larger block size of 128 bits, while DES uses a smaller block size of 64 bits. This means that AES can encrypt more data in a single operation, reducing the number of operations required for large inputs.

The fact that RC4 had the lowest time in all of them is also expected. RC4 is a relatively simple encryption algorithm that has a low computational overhead, making it faster than other encryption techniques. However, it is important to note that RC4 has some known weaknesses and is no longer recommended for use in modern encryption systems.

Task 7: (Only for the AES cipher) Modify the AES.py file, to implement the encrypt and decrypt functions code for the two extra modes of operations CFB, and CTR modes of operation. Test the code using the AES_Demo.py to trigger the CFB and CTR modes of operation. Report the results.

Import AES_Demo.py

=========AES-192 (Long Msg, CFB)=========

192-bit Key : b'a\x01\x0b\x8a\x01[l\x0c\t\xf10\x17\xa0\xaf\x00u\xd2\x9f\xd4\x90\xfe\xc0\xc2\xd7'

AES-192 cipher Text :
b'\nf\xb4i\xc4$\xb9|\xeaC\x8a\xd9H\xa3\x05F\xfe\nSD\xea@,\x044\xb4\xec\xb3\xdc\xa4\x9e\xb0V\xb8\xebsV\xec\x8e\xe1\xaa\xe9\xa6x!\xd8!\xa2\x80{\xbd5L\x94+\xb6e\x89j%\x18\xa7\t\x1bh\x8eG\x19\xea0\x908\xd4\x0b\x02\xbf\r\xbf\x83T_\xc15\ryZ\xd4t\xf5\x9f\xdc\xf8\xf7y:\xc3'

Decrypted cipher Text : b'This is a long text as a test input for AES_Demo.py for task 7 in the assignment'

=========AES-192 (Long Msg, CTR)=========

192-bit Key : b'a\x01\x0b\x8a\x01[l\x0c\t\xf10\x17\xa0\xaf\x00u\xd2\x9f\xd4\x90\xfe\xc0\xc2\xd7'

AES-192 cipherText : b'\xcb\xac>_TCVp\xd9\xffuC\x89\xce\xac\xcfY46\x807\xed\xe9\xec8
\x1a\x90\xc1\xab\x8fM\xe8\x16g\x89\xa2\x0bz\xec\xea\x96r\xba\x13}\xb1j[\x89\xfb\xd1\xc2\xd0\x1
7r\xd5\xd5\xb8\x05\xddi\xd3\x10@\x10\x0b\xba\xa6\xa1HAl\xf34y_[\x8ami\x0e\xd2=\xfb\xc2uk\xe5
GB9\x12#\xab9'

Decrypted cipher Text : b'This is a long text as a test input for AES_Demo.py for task 7 in the assignment'

Code for AES modes of operations:

CFB: See comments for easier explanation

For cipher feed back mode encryption:

```
    elif (mode == utils.BC_mode.CFB and iv != None):
        previous = iv
        s = 8
        #The plain text will be encrypted depending on the number of bits s
        for plaintext_block in tqdm(utils.split_blocks(plaintext, block_size=s ), desc="CFB encryption"):
            #encrypt block previous and choose s bytes
            block = self.encrypt_block(previous)[:s]
            #XOR block with the plain text to produce first cipher
            ciphertext_block = self.xor_bytes(block, plaintext_block)
            blocks.append(ciphertext_block)
            #now we must update the blocks for the next loop
            #The cipher block we just encrypted will take the LSB
            prev_ciphertext_block = ciphertext_block
            previous = previous[:s] + prev_ciphertext_block
        ciphertext = b''.join(blocks)
# now for decryption
    elif (mode == utils.BC_mode.CFB and iv != None):
        previous = iv
        s = 8
        for ciphertext_block in tqdm(utils.split_blocks(ciphertext, block_size=s), desc="CFB decryption"):
```

```
            #we encrypt the IV and shift using the cipher text to get the same values that were used to XOR
#the plain text.
            temp = self.encrypt_block(previous)[:s]
            plaintext_block = self.xor_bytes(ciphertext_block, temp)
            blocks.append(plaintext_block)
            previous = previous[:s] + ciphertext_block
        plaintext =  utils.unpad( b''.join(blocks) )
```

For the CTR (short for counter):

The initial value of the counter (number_int) was initialized using the Initialization vector.

The encryption is simply encrypting then XORing the counter block with the plain text.

And the reverse operation is done for decrypting. The decryption code is not included in the report for space.

```
        for counter, plaintext_block in tqdm(enumerate(utils.split_blocks(plaintext)), desc="CTR
encryption"):
            counter_block = number_int + counter
            counter_block = counter_block % 2**16

            counter_block_to_encrypt = int_to_bytes(counter_block)

            block = self.encrypt_block(counter_block_to_encrypt)
            ciphertext_block = self.xor_bytes(block, plaintext_block)

            blocks.append(ciphertext_block)



            # CTR mode encrypt: plaintext_block XOR encrypt(counter)
            # write the code for the counter mode here
        ciphertext = b''.join(blocks)
```

Task 8: Repeat 5, fill the compareCHERR_AESmodes_RC4() function using the ciphers AES, AES with CBC, AES with OFB, AES with CFB, AES with CTR modes, and RC4.

From main import compareCHERR_AESmodes_RC4

The decrypted images (monalisa200) for the 6 trials with different error rates

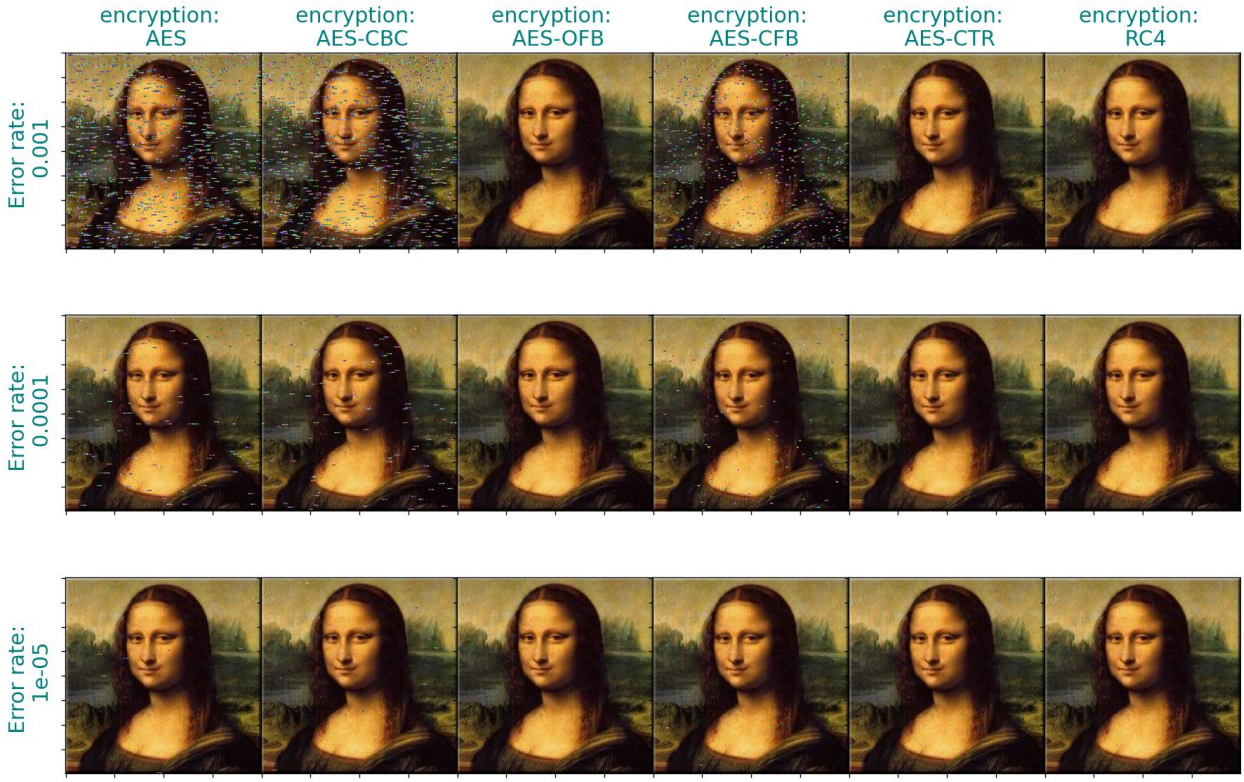

Figure 12 Comparing Monalisa200 with the different modes of operations of AES and RC4

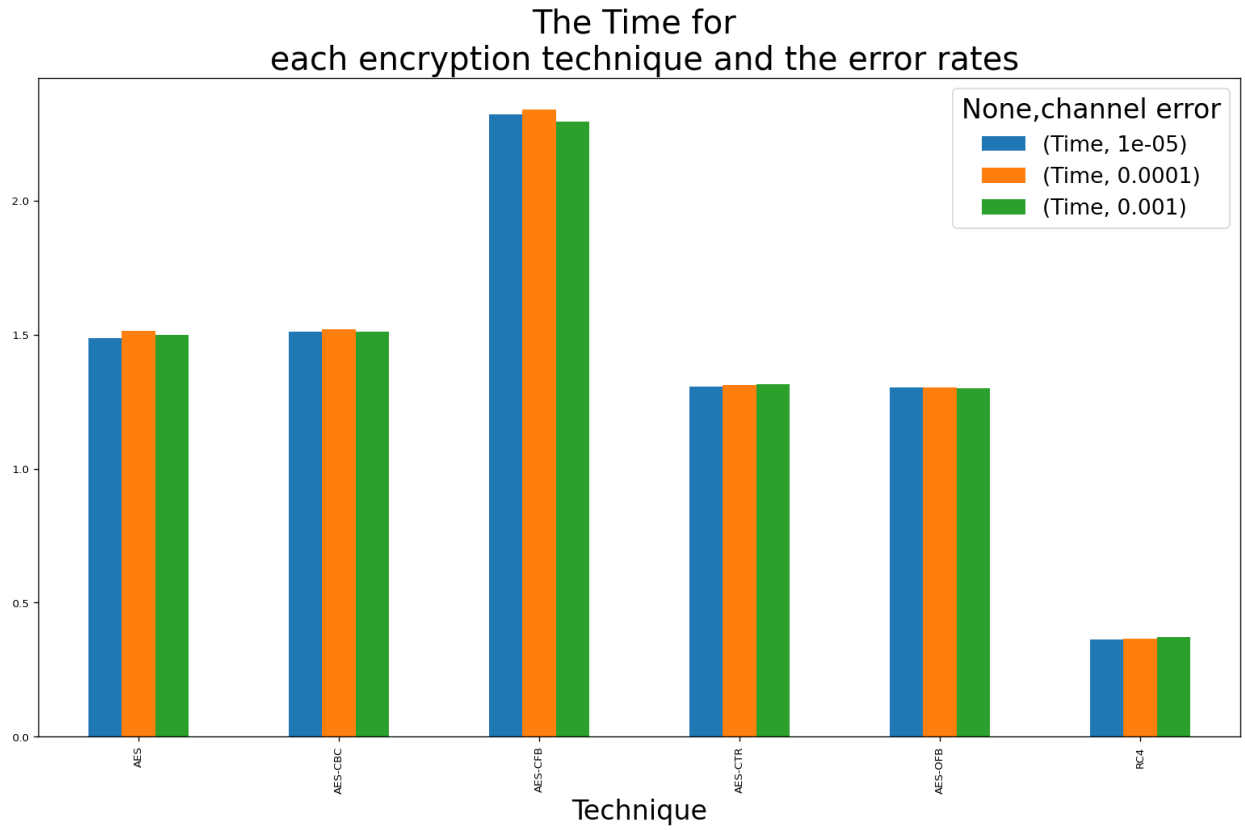

Figure 13 The time complexity of the different encryption mechanisms

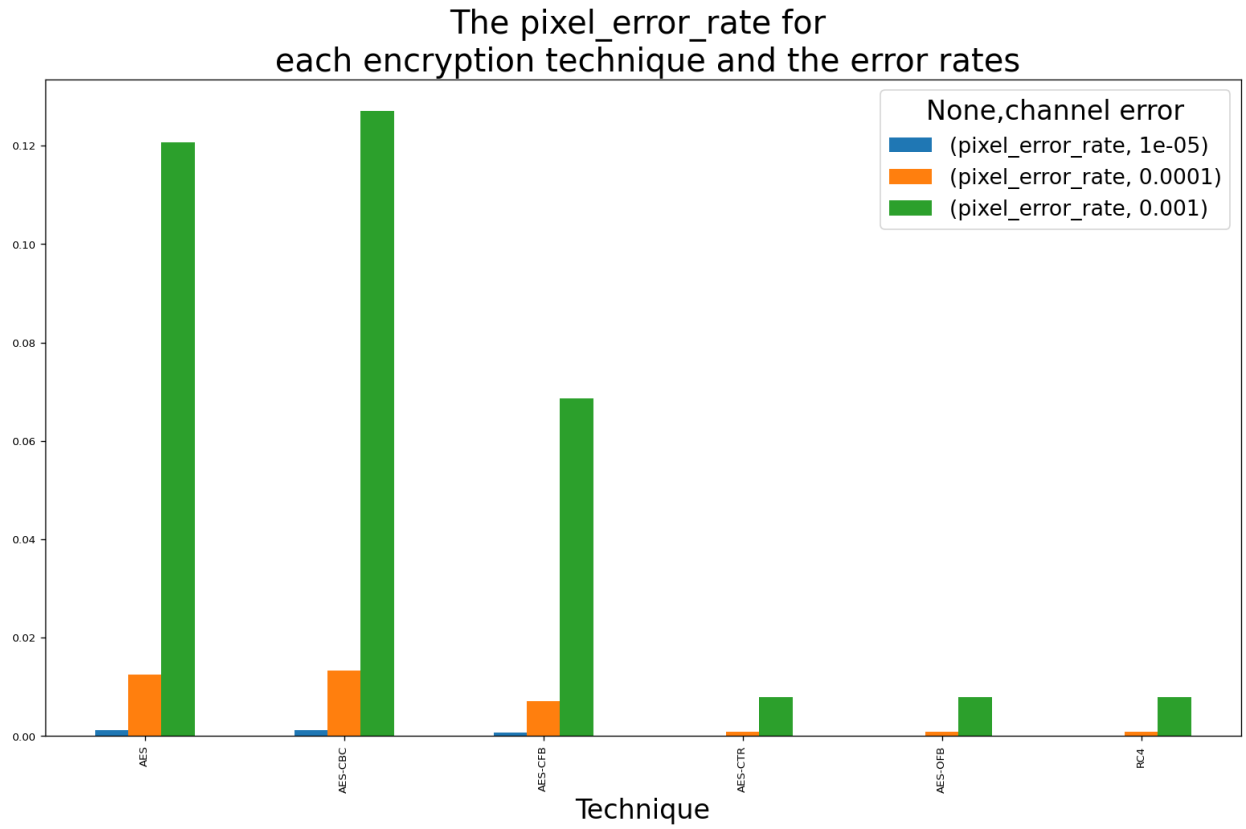

Figure 14 The pixel error rate with the different encryption mechanisms

Comments on results

ECB and CBC modes are both block ciphers, meaning that they process the plaintext in fixed-sized blocks and apply the encryption algorithm to each block independently. In ECB mode, the same key is used to encrypt every block of plaintext, which can result in patterns being preserved in the ciphertext that can be exploited by an attacker. In CBC mode, each block is XORed with the previous block of ciphertext before encryption, which adds some level of randomness to the ciphertext. However, if an error occurs in one block, it can propagate to the next block and affect the decryption of subsequent blocks, resulting in a higher overall pixel error rate.

CFB mode is a bit different from ECB and CBC modes in that it turns the block cipher into a stream cipher by encrypting the previous ciphertext block and XORing the result with the plaintext block. This creates a feedback loop that makes it more difficult for errors to propagate, but it still relies on a fixed key for encryption.

OFB and CTR modes are both stream ciphers that use a counter to generate a key stream that is XORed with the plaintext to produce the ciphertext. This makes them more resistant to errors because they do not rely on the previous ciphertext block to encrypt the next block of plaintext. Instead, errors are isolated to the block in which they occur and do not affect subsequent blocks.

RC4 is a stream cipher and encrypts byte by byte, that’s why errors don’t propagate.\

Based on the results of time complexity provided in figure 13, it makes sense that AES-CFB mode took the most time among all the modes of operation because it requires more processing and computation compared to other modes. (Each block of plaintext or ciphertext depends on the previous block.)

It's also expected that AES-ECB and AES-CBC took slightly more time than AES-CTR and AES-OFB because they are generally more computationally expensive due to their block-based nature and the need for padding.

It's interesting to note that RC4 was the fastest by a huge margin. RC4 is a stream cipher and therefore doesn't require any padding or block processing. This can make it faster than block ciphers like AES.

Overall, these results seem to make sense and are consistent with what we would expect based on the characteristics of the encryption mechanisms used.

Task 9: Repeat 6, fill the compareIMGQLTY_AESmodes_RC4() using the ciphers AES, AES with CBC, AES with OFB, AES with CFB, AES with CTR modes, and RC4.

compareIMGQLTY_AESmodes_RC4

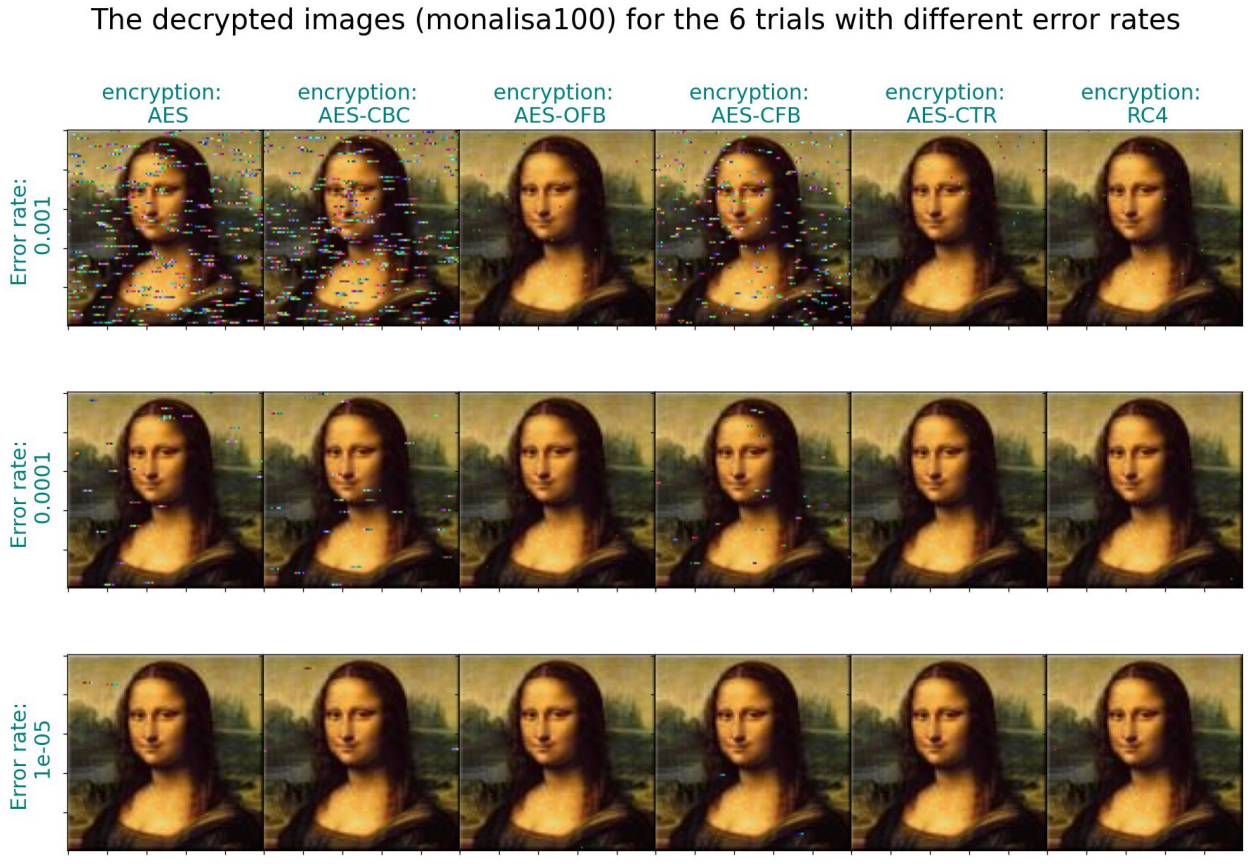

Figure 15 The recovered images from different modes of operation on AES and RC4

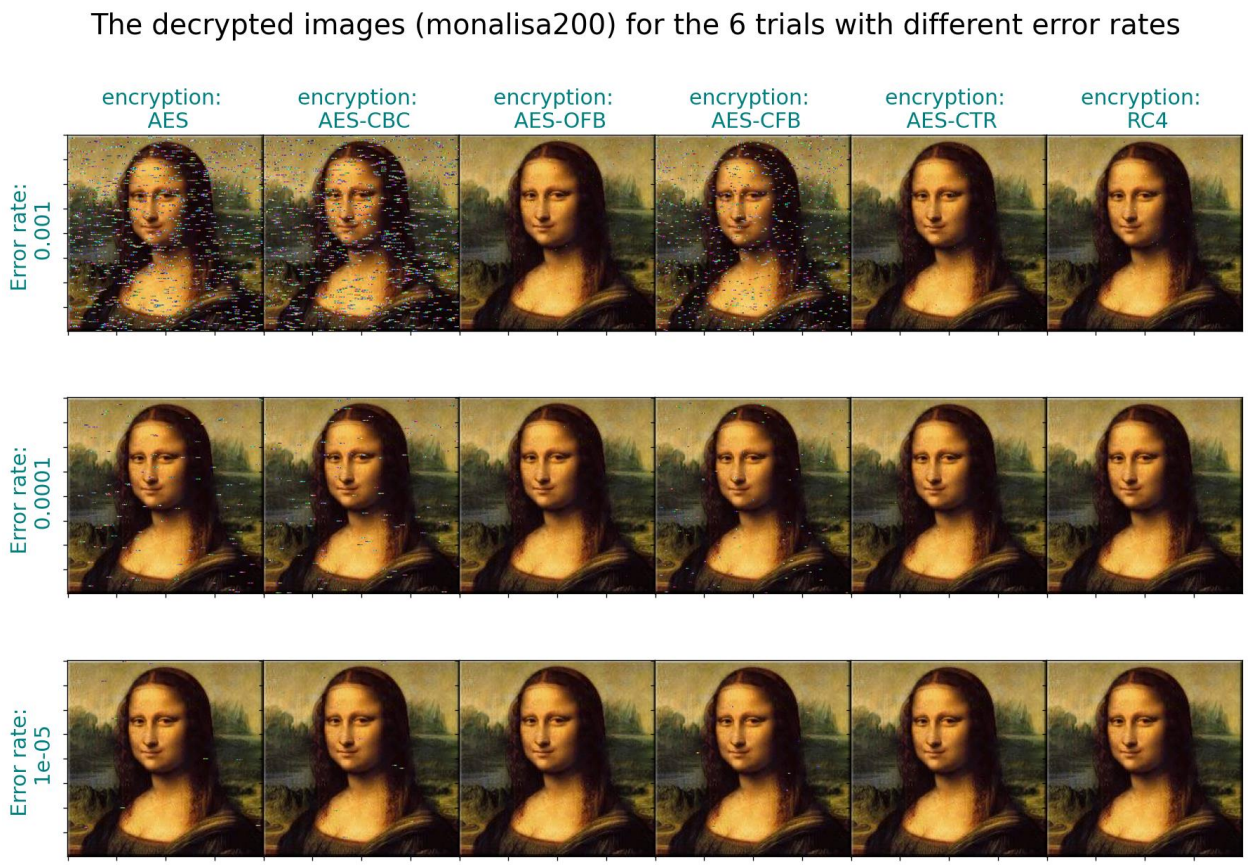

Figure 16 The recovered images from different modes of operation on AES and RC4

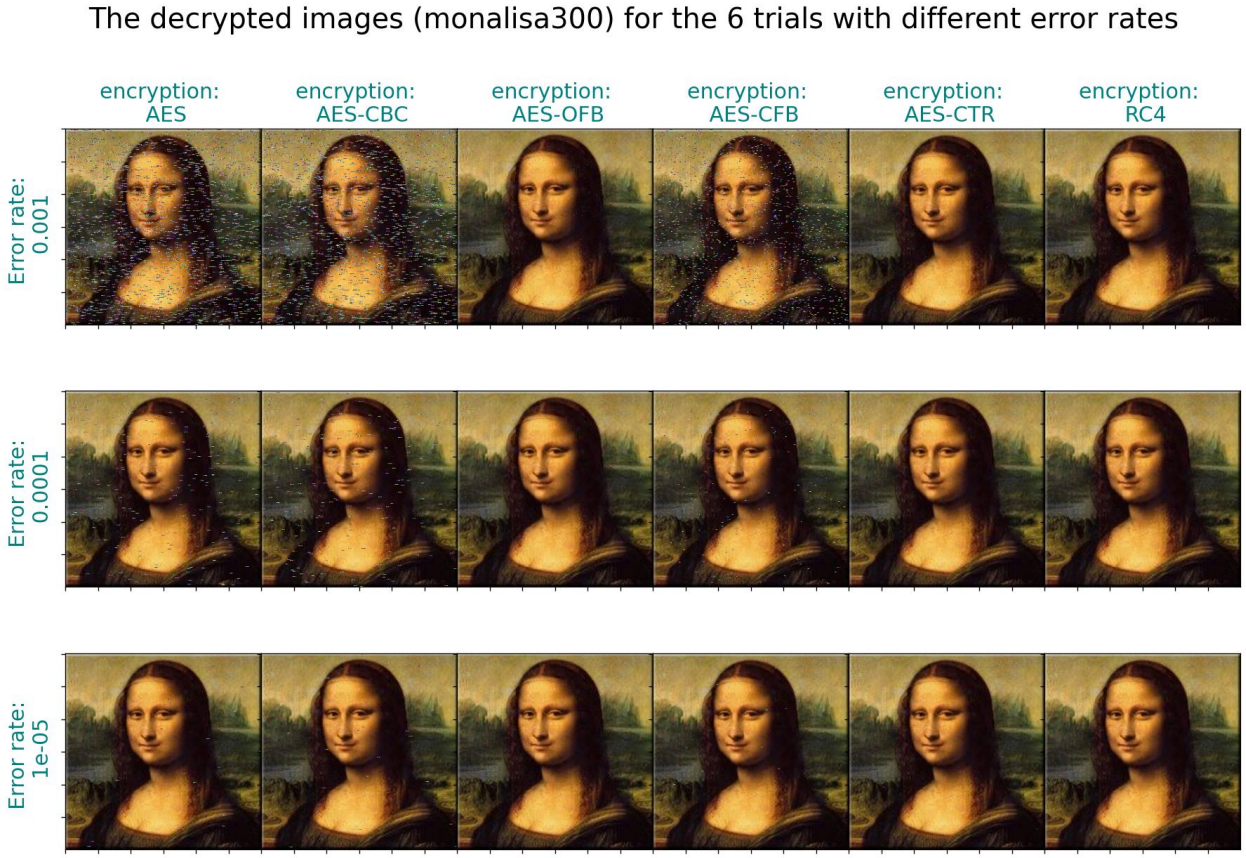

Figure 17 The recovered images from different modes of operation on AES and RC4

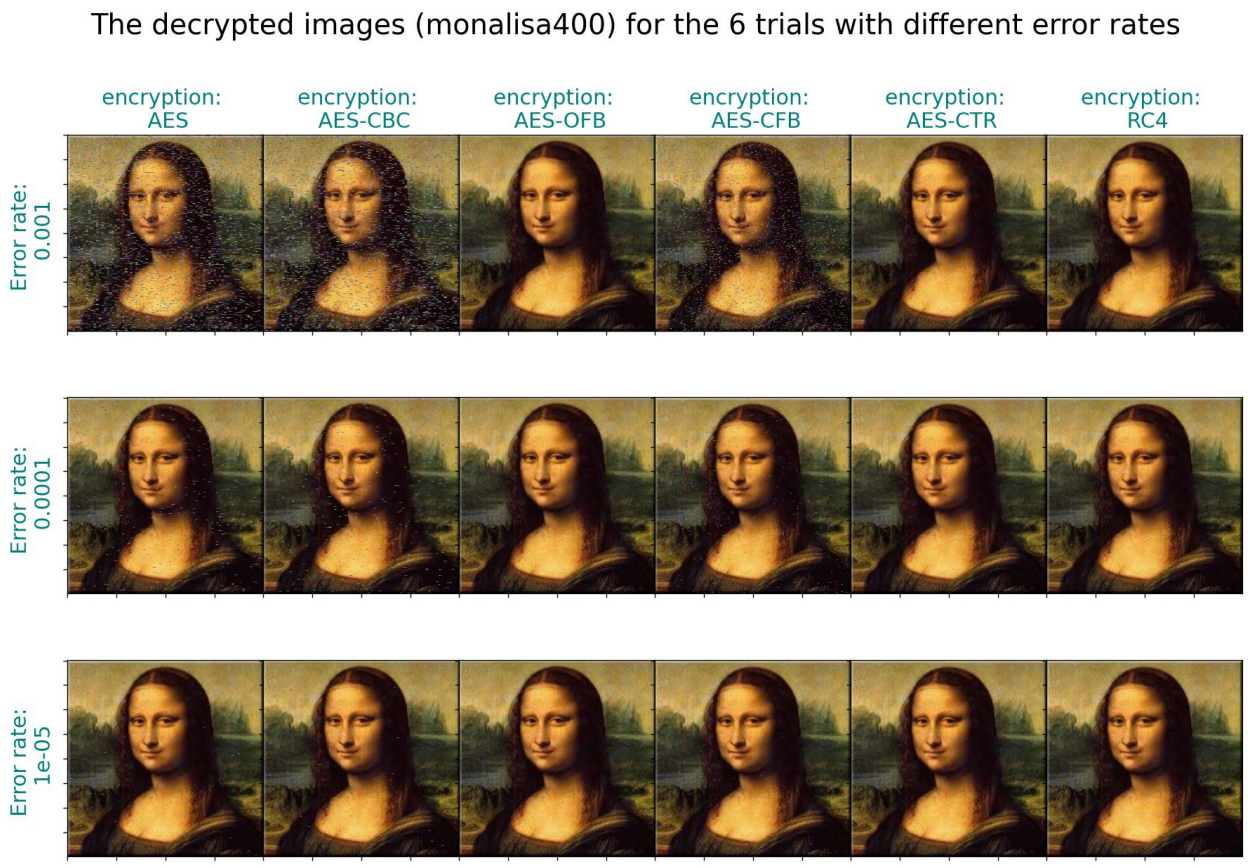

Figure 18 The recovered images from different modes of operation on AES and RC4

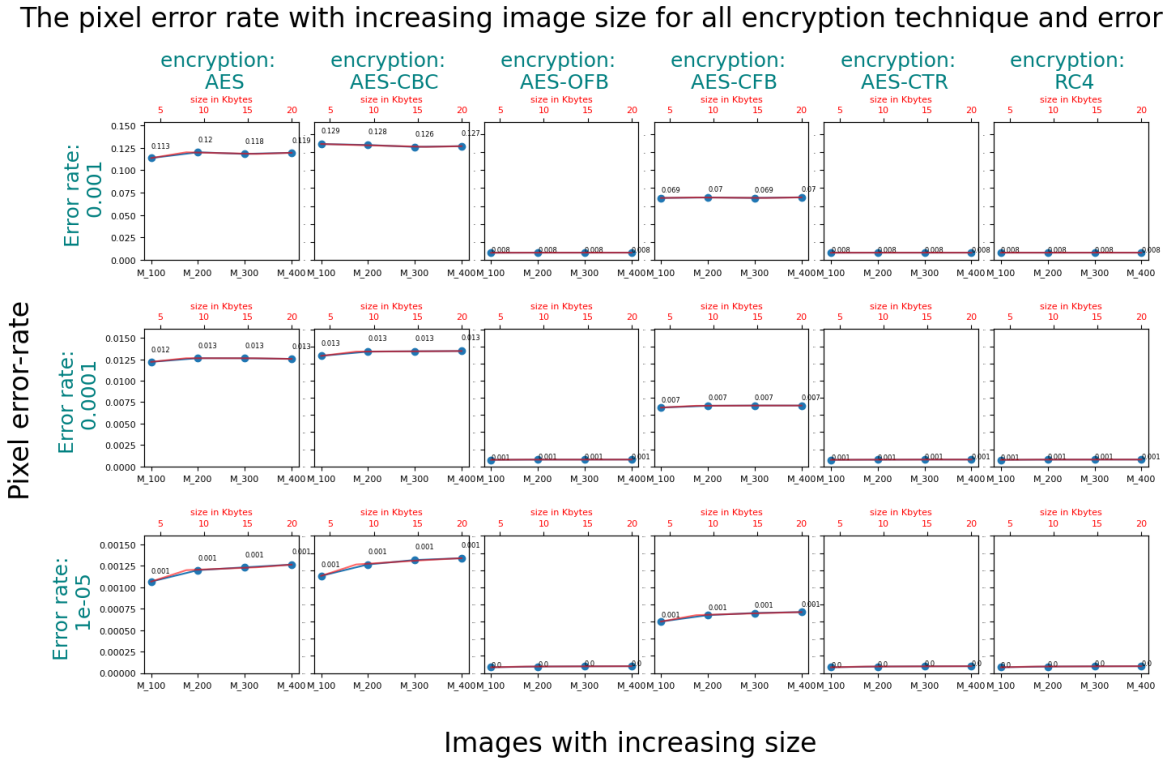

Figure 19 The recovered images pixel error rate from different modes of operation on AES and RC4

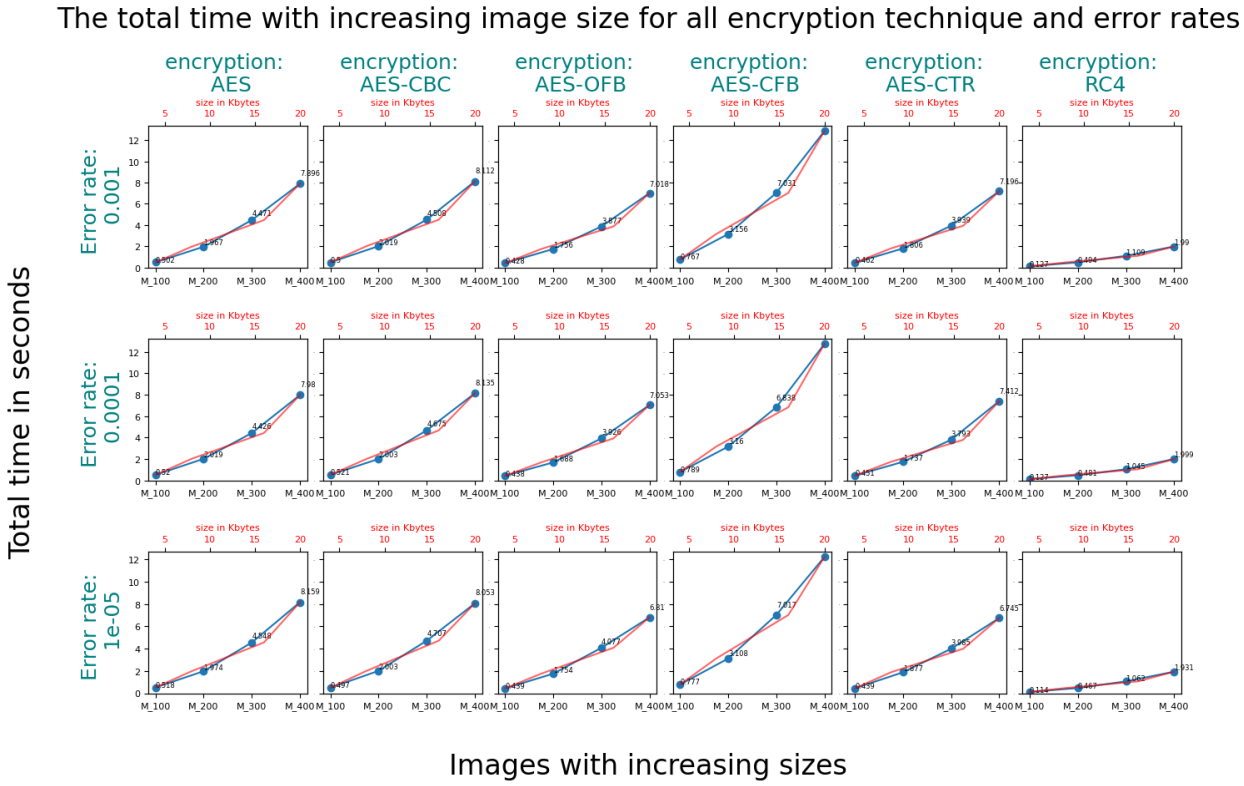

Figure 20 The recovered images time complexity from different modes of operation on AES and RC4

Comments on results

The fact that the pixel error rate didn't change with increasing image size suggests that the error rate of the noisy channel is the main factor impacting the error rate of the encrypted data. This makes sense since the noisy channel is introducing errors at a fixed rate, regardless of the size of the image being transmitted.

All ciphers scaled with the same slope across several image sizes, it suggests that the encryption and decryption time is proportional to the size of the image being processed. This makes sense since the amount of data being processed increases with the size of the image, and encryption and decryption algorithms are designed to work on fixed-size blocks of data.

However, it's possible that this observation could change for very large image sizes or very small image sizes. For example, very large image sizes may exceed the amount of memory available on the computer, leading to slower processing times due to disk swapping. Conversely, very small image sizes may not provide enough data to effectively utilize modern hardware optimizations, leading to suboptimal performance.