T.C.

MARMARA UNIVERSITY

FACULTY OF ENGINEERING

COMPUTER SCIENCE AND ENGINEERING

CSE 4057 Project 1

Group Members

150117009 - Eymen Topçuoğlu

150117013 - Mehmet Ali Yüksel

150117018 - Ahmet Önkol

Submitted to

Assoc. Prof. Ömer Korçak

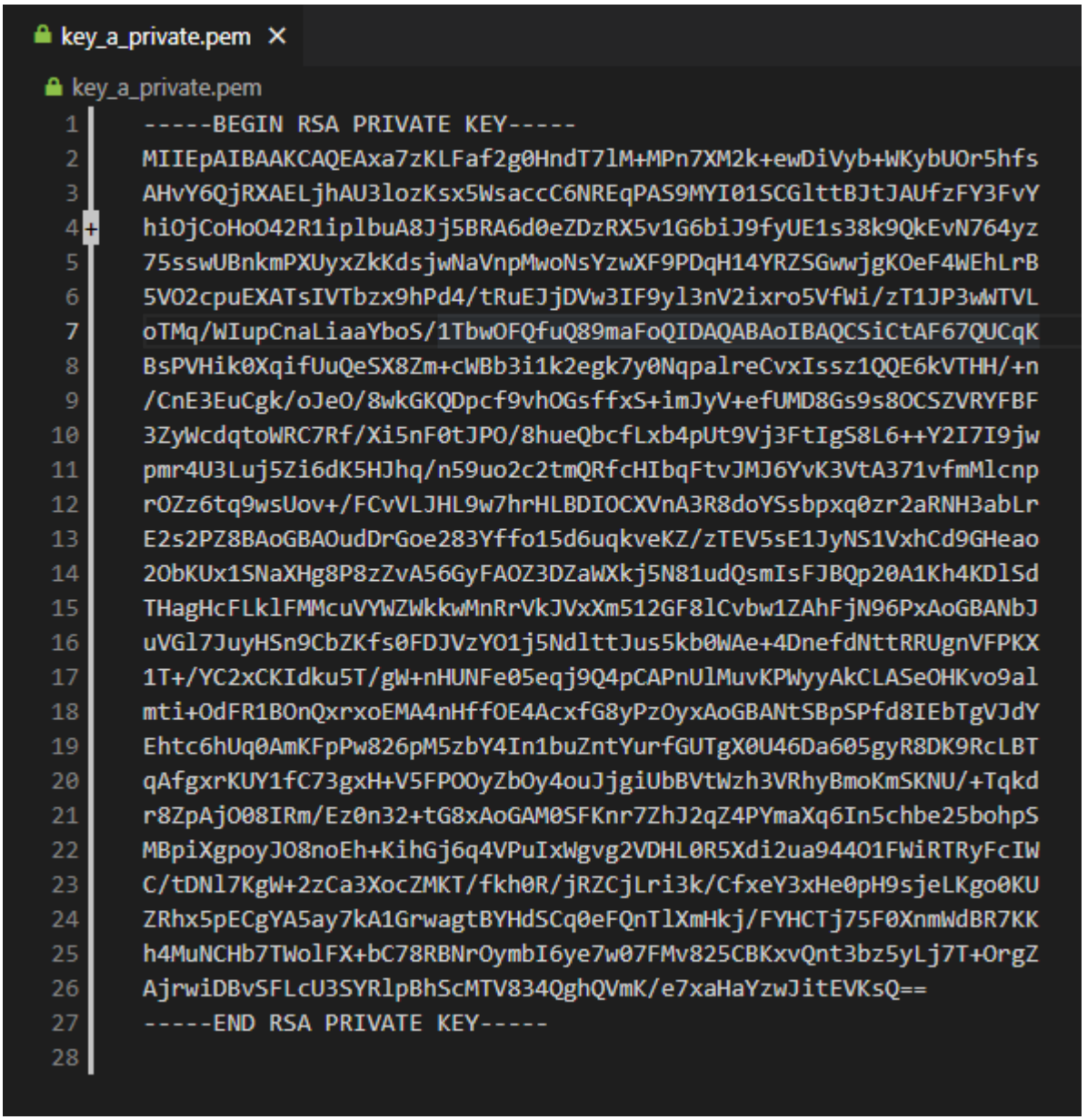
Introduction

For the development of related methods we have used the following library and its functionalities. -> https://cryptography.io/en/2.4.2/

Implementation Details

1) Generation of public-private key pairs:

a) ***generate_rsa_key_pair()*** is responsible for this part. We create a private key then obtain the public pair of it. Corresponding key pairs are written into pem files at the end. Generated keys for KA can be accessed in the format of key_a_private.pem and key_a_public.pem. Key size is given as 2048.



b) ***generate_elliptic_curve_diffie_helman_key_pair()*** is responsible for this part. We are using the elliptic curve functionality of the corresponding library. Process of generating key pairs is the same with 1.a part. We also generated public and private keys for KB and KC but didn't include in the report since it isn't asked.

For both parts in the first section we returned both key pairs from methods for later usage.

2) Generation of Symmetric keys:

a) ***generate_symmetric_key()*** and ***key_encrypt_decrypt()*** is responsible for generating the symmetric keys. Method takes length information(128/8 = 16 bytes for K1 and 256/8 = 32 bytes for K2). Deriving a key suitable for use as input to an encryption algorithm means taking a password and running it through an algorithm such as PBKDF2_HMAC or HKDF. We use PBKDF2_HMAC for securing the key. Values of the keys are as follows in the below screenshot after encrypting with KA+, and decrypting with KA- . Since we are assigning corresponding keys from function calls, we managed to do our encryption and decryption with them. Additionally we compare encrypted and original keys with the following line to assure they are the same.

```
print("Encrypted key:" + str(ciphertext))
print("Decrypted key:" + str(plaintext))
print("Are decrypted key and original key same: " + str(plaintext == message))
```

```
----------------------2-a-----------------------------------
***Key_1***
Key length:
16
b'\x81\xfc\xd4\x06\xca\x01yA\xa6\x0e\x86\xd3\xc5\x98\xec\x0f'
Encrypted key:b'\x10\xe7\x1f,V\x15\xdbj\x81\x96\xfd\xe7\x83\xb0tL3\x96\x820&\x
xe7\xc1\x14\xc7\x8b*\xea\x9d\xfb\x827\xa2\x97\xb9\xe5_Pg\xa9g$\xcf\x90n\xca\x0
d7\x87x\xaf\x93+\x95%\x83\x92\x08\x0e\xa0ZLYg"[\x18\x1b|\x17\xb0\xf3\xb6\x8fUg
64V\x98\xe5I\x97\x89\xeem\x9f\x93\xe6J\xf0P\x0f\xc6\x14,\xce6\xe4[\x08\x00\xfa
-\x8f\x93;\xe5\xe1so\xad\xcdy\xc4\xd5\xec\x15\x8a.hd\x7f/\xaae\xdf\xf46r%\x82\
f\xd4\xf1\xc5\x0f\x80\x91x\x1c\xdck\xac\xcd\x18\xb0hV\xe5d\xdbS%\x14\x82\xca\x
9\xcb^-\x1a\xd1R;ev\xbb\xaf\xe8\xe6\xa7\x02)\x7f'
Decrypted key:b'\x81\xfc\xd4\x06\xca\x01yA\xa6\x0e\x86\xd3\xc5\x98\xec\x0f'
Are decrypted key and original key same: True
***Key_2***
Key length:
32
b'S\x16\x0b]\xf8\x88C\x8d\x12X\xa4"\xb1\xc1]+\xca0\xd5k\xe8H\xbckw\x8e\xd3u\xb
Encrypted key:b"\x16\xfa\xa5(\xd1\xb2_h\xf8w8\xf5.\xf1a\x8f\xd3\x17)\xba\x87\\
xe0\xdb\xe1\x83w\xfe\x9fl\x7fJ\xf6\xee\x1dc\xcbX\xaeU\x0eRl\x81\xa2\xdbp\xde\x
5\xf8\x8d\xe9<\xa6\x15\xff\xc4\x8b\xce\x9cw3MC\xb7\x06nL\x9b\xa4\xabY)\x13\x80
\x9b\xedP\x94^\x1f\x0f\xdc7\xa3\xc7Rw\xbf1\xfa3\xa9\xcb\xbcu\xfc\xe1\x15\x9b\x
7\xd4\xefhk\x086\xaf\x88\xd0u\xae\xbb\xb3~I\xe4(TI\xca!\x90\xd1Z\x91]\xf7\x94\
e10pQ@\x93\xf8\xe4\x89\xb2\\\x81\xa5\x1b^5\x0e\xb0\xf5|\x81\x0f\xb6+\x1fK\xa6F
I\xb8\x17-\x94\xea3>\xac\xa8\xfe\x1c\x85b\x10\x15v\xea\xd2\x0f\xa1x\t"
Decrypted key:b'S\x16\x0b]\xf8\x88C\x8d\x12X\xa4"\xb1\xc1]+\xca0\xd5k\xe8H\xbc
Are decrypted key and original key same: True
```

b) ***generate_symmetric_with_ECDH()*** is responsible for this part. It takes private_key_1, public_key_1, private_key_2, public_key_2 as arguments which are *key_b_private*, *key_b_public*, *key_c_private*, *key_c_public* in our case. Initially we create $K3$ with ECDH and hold it in a variable called *shared_key*, then we derive it into *key_3* using HKDF. In the meantime of creation of *key_3* we used *key_b_private* with *key_c_public*, and for *key_3_test* we used *key_c_private* with *key_b_public.* We also compared our results and output is as follows:

```
----------------------2-b-----------------------------------
First key:
b'x%\xdc\xca\x8f\xd5\x0cLu1\xa7\x8e\xf9ch\xf6\x9e\xb8\xd9\xc0\xd4\x08\xe3>\xe2|\x18A\xa1\r;\xa3'
Second key:
b'x%\xdc\xca\x8f\xd5\x0cLu1\xa7\x8e\xf9ch\xf6\x9e\xb8\xd9\xc0\xd4\x08\xe3>\xe2|\x18A\xa1\r;\xa3'
Are the ECDH keys same: True
```

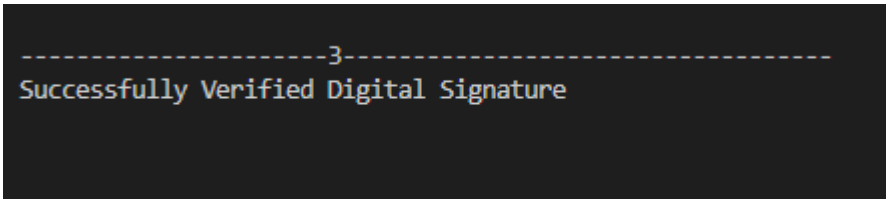
3) Generation and Verification of Digital Signature:

We have constructed 2 methods for this purpose:

generate_digital_signature(): This method takes a private key to be used in creating digital signature and a message to be encrypted. Firstly, we create a digest of the message (H(m)) by simply hashing it with SHA256 hashing algorithm and then we use the sign method of the cryptography library to create the signature from the given private key. Sign method takes the digest and the hash function used as parameters along with the parameter specifying padding features. Finally, we print the signature we have in our hand.

verify_digital_signature(): As its name suggests this method simply verifies the given digital signature. This method uses cryptography library's verify method which takes the digest and the signature as parameters and checks if the signature has the digest matching with the given (original) digest. This method raises an error if there is a non-match situation which states that the signature process has failed. In the documentation of the library it is stated as "*If the signature does not match, verify() will raise an InvalidSignature exception*". By checking if there's an error or not we understand whether this signaturing process succeeded or not.

Since the library methods handle the matching of the digests in the background, it was not possible for us to observe manually whether the matching is correct or not. We simply checked whether there has been an error occurred or not and by that we understood that the process succeeded



4) AES Encryption:

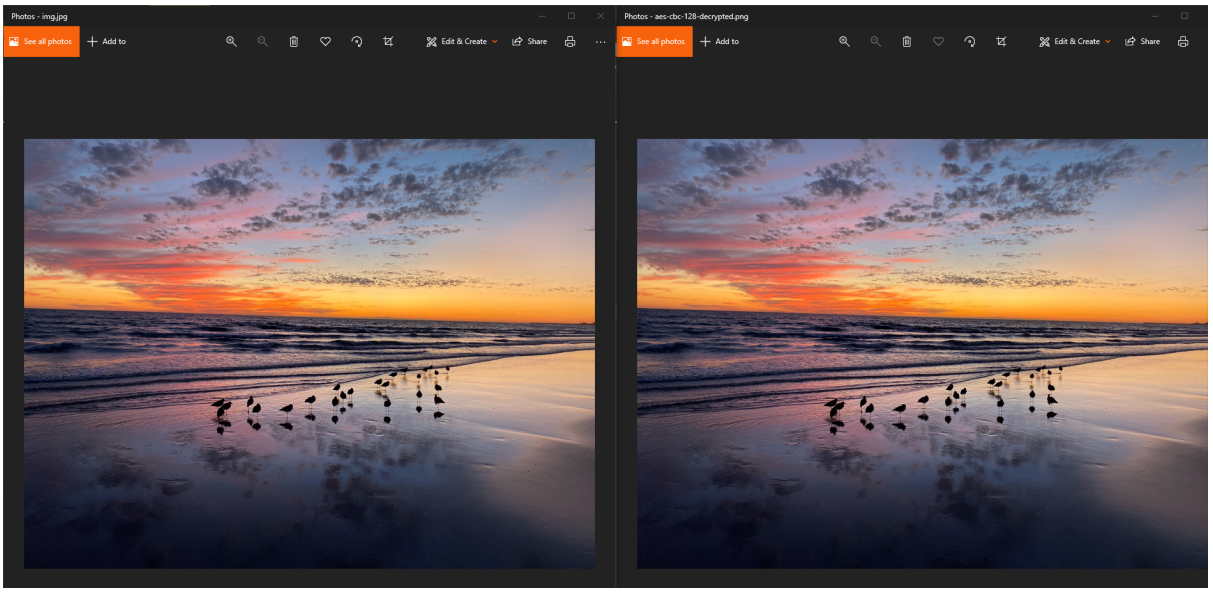
encrypt_with_AES(): We first apply padding to the given data. Then we calculate a random initialization vector / nonce (based on the mode parameter). Then we encrypt the padded data and measure the time. At the end, we write the encrypted data to file

decrypt_with_AES(): We first decrypt the given ciphertext and measure the time. And at the end we write the decrypted data to a file.

1) ***AES (128 bit key) in CBC mode***

   a) *See aes-cbc-128-encrypted.png for encrypted file*

   b) *See aes-cbc-128-decrypted.png for decrypted file*
   They are the same files:



   c) See *below figure* for comparison of the elapsed times.

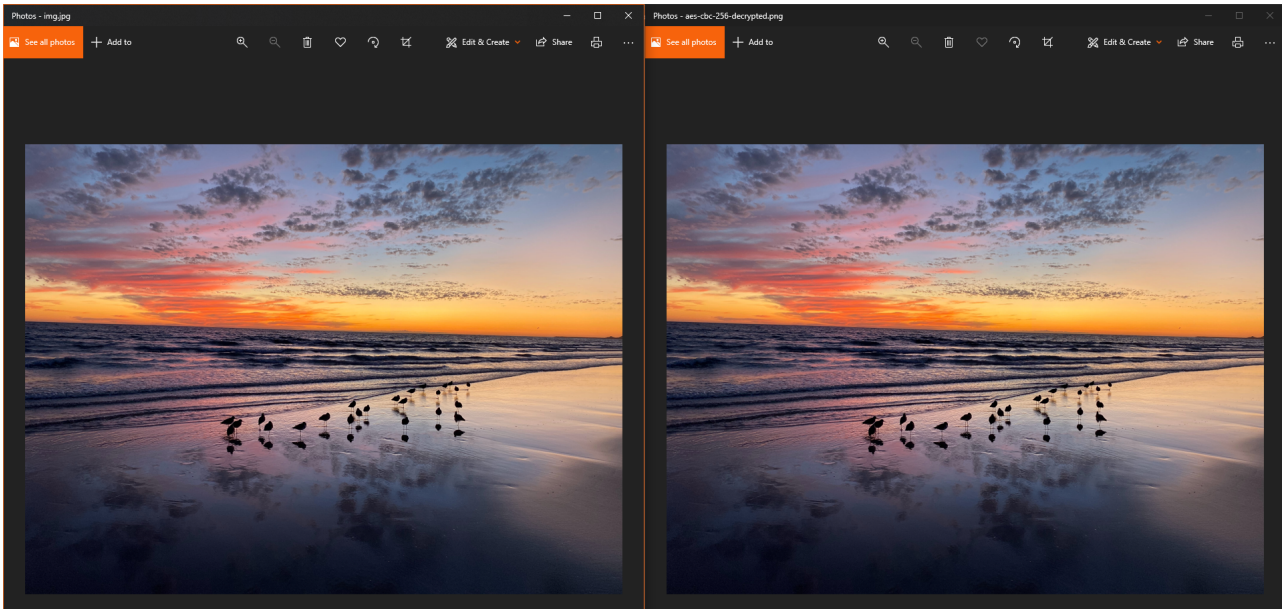
   d) See *below figure* for the result which is indicated with the result of "For 128 bit key1 in CBC mode, Are ciphertexts same with different IVs: " + str(cipher_text == cipher_text_2)

2) *AES (256 bit key) in CBC mode*

a) *See aes-cbc-256-encrypted.png for encrypted file*

b) *See aes-cbc-256-decrypted.png for decrypted file*

They are the same files:



c) See *below figure* for comparison of the elapsed times.

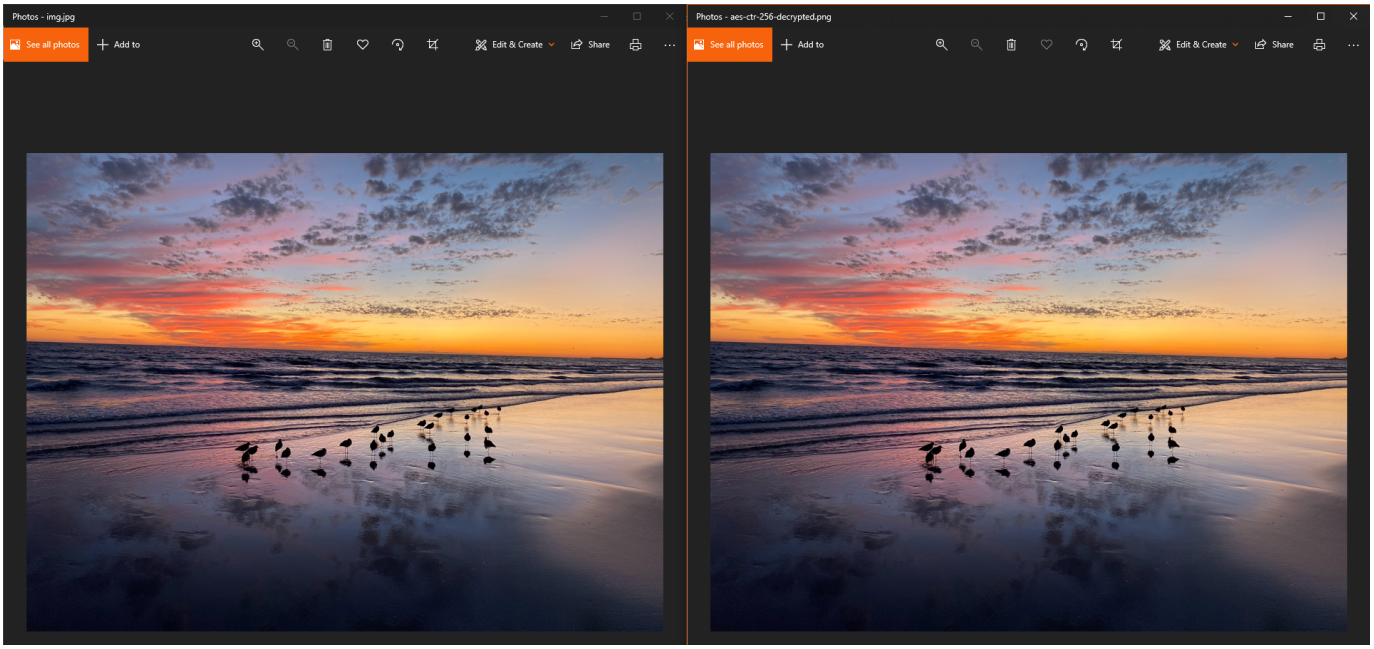
d) See *below figure* for the result which is indicated with the result of *"For 128 bit key1 in CBC mode, Are ciphertexts same with different IVs: " + str(cipher_text == cipher_text_2)*

3) *AES (256 bit key) in CTR mode*

a) *See aes-ctr-256-encrypted.png for encrypted file*

b) *See aes-ctr-256-decrypted.png for decrypted file*

They are the same files:



c) See *below figure* for comparison of the elapsed times.

d) See *below figure* for the result which is indicated with the result of *"For 128 bit key1 in CBC mode, Are ciphertexts same with different IVs: " + str(cipher_text == cipher_text_2)*

```
----------------------4-i-----------------------------------
Elapsed time while encrypting with AES: 0.001993417739868164 secs
Elapsed time while encrypting with AES: 0.0019931793212890625 secs
For 128 bit key1 in CBC mode, Are ciphertexts same with different IVs: False
Elapsed time while decrypting with AES: 0.001995086669921875 secs


----------------------4-ii----------------------------------
Elapsed time while encrypting with AES: 0.004988670349121094 secs
Elapsed time while encrypting with AES: 0.003952741622924805 secs
For 256 bit key2 in CBC mode, Are ciphertexts same with different IVs: False
Elapsed time while decrypting with AES: 0.0029866695404052734 secs


----------------------4-iii----------------------------------
Elapsed time while encrypting with AES: 0.005197763442993164 secs
Elapsed time while encrypting with AES: 0.004294872283935547 secs
For 256 bit key3 in CTR mode, Are ciphertexts same with different nonces: False
Elapsed time while decrypting with AES: 0.0025053024291992188 secs
```

As can be seen, elapsed time increases as the key size increases. As for the CTR and CBC, results are varying and there is no certain observation depending on the change of the mode. But in general, we observe that CTR is slightly slower while encrypting, and faster while decrypting.

5) Message Authentication Codes

For the both a and b parts we have implemented a single method ***message_auth():***

This method makes use of the library's HMAC algorithm. Simply it takes a symmetric key, message and the specified hashing function then returns the generated authentication code. For part b, we have given the *key_2* as the key and the message and obtained the new 256 bit key.

```
----------------------5a----------------------------------
Message auth code using key2: b'\xd9n:]\x84o\x03W\xd0}zqhfc\xbf\xf5T\x1c\xaclT\x
93\x9d\xb1\xa0\xc7h]\x0c\xd0\xce'

----------------------5b----------------------------------
New key2 with message auth code: b'Dj\xdc\xecH_t\xdb\xc1\xc9\x7f\xad\xe1\xca\xf0
S\xad[\x80\xd4\x03\xe3N\xe2\xbeI\xfe+\x86\t\x02\xee'
```