Project Execution

For the first part we would be applying 2 hash functions on the data file to generate data digest on SHA-256 and Blake2s. The purpose of this is to ensure data integrity at the end of the encryption excise by comparing the results of the decrypted data and draw a conclusion.

Fig 1 Hash functions application

```
# Applying SHA-256 hash function on data file to generate data digests
from Crypto.Hash import SHA256

input_file = b'R11503611_data.docx'

with open(input_file, 'rb') as f:
    data = f.read()

h = SHA256.new(data)
h.hexdigest()
```

```
'a0699bb21389615336dfdd240d1b252413d30a96a6cdd5425cdc9183c87e5f15'
```

```
# Applying BLAKE2s hash function on data file to generate data digests

from Crypto.Hash import BLAKE2s

input_file = b'R11503611_data.docx'

with open(input_file, 'rb') as f:
    data = f.read()

Hblk = BLAKE2s.new(digest_bits=256)
Hblk.update(data)
Hblk.hexdigest()
```

```
'56a80abb788d816464f79891e4ef3ad2e01ff69c3789fe45c8fccc7e11f7f0af'
```

Fig 1 shows hash functions applied to data file - R11503611_data.docx (News report file). A record of the hexadecimal data digest value of each hash function was also recorded which will be used later for comparison with the encrypted data. To generate data digest value we use hexdigest() function.

SHA-256 value –
'a0699bb21389615336dfdd240d1b252413d30a96a6cdd5425cdc9183c87e5f15'

Blake2s value –
‘56a80abb788d816464f79891e4ef3ad2e01ff69c3789fe45c8fccc7e11f7f0af’

Data Encryption.

We encrypt the news report file with the symmetric ciphers – AES in CBC mode and CAST128. Before encryption we generate symmetric key that will be used for both encryption and decryption.

Several methods can be used for key choice to generate random bytes for password or you could use a passphrase. For this report we used random generator functions

Fig3, and fig4 below shows python script of encryption using AES in CBC mode and CAST128 ciphers. The choice key and news report document were read from their respective documents and encrypted into a file document R11503611_aes.bin as the encrypted file. Other libraries used in the process was padding to pad the document and crypto.random as a random generator.

Fig2 – AES encryption in CBC mode using random key genetrator

```
# Aes encryption with random key selection

from Crypto.Cipher import AES
from Crypto.Util.Padding import pad
from Crypto.Random import get_random_bytes
import os

output_file = 'R11503611_aes.bin' # Output file
input_file = b'R11503611_data.docx'

with open(input_file, 'rb') as f:
    data = f.read()

key = get_random_bytes(16)
file = open('key2', 'wb')
file.write(key) # The key is type bytes still
file.close()
print(key)

# Create cipher object and encrypt the data
cipher = AES.new(key, AES.MODE_CBC) # Create a AES cipher object with the key using the mode CBC
ciphered_data = cipher.encrypt(pad(data, AES.block_size)) # Pad the input data and then encrypt

file_out = open(output_file, "wb") # Open file to write bytes
file_out.write(cipher.iv) # Write the iv to the output file (will be required for decryption)
file_out.write(ciphered_data) # Write the varying length ciphertext to the file (this is the encrypted data)
file_out.close()
```

```
b'C\xc2\xd2\x9a?q_v\x93\xbd\xcb\xc8V\x9c\xf2='
```

Fig 3 – Cast128 encryption using OPENPGP mode

```
]: from Crypto.Cipher import CAST
   from Crypto import Random

   output_file = 'R11503611_cast.bin' # Output file
   input_file = 'R11503611_data.docx'

   with open(input_file, 'rb') as f:
       data = f.read()

   key = os.urandom(16)

   file = open('castkey', 'wb')
   file.write(key) # The key is type bytes still
   file.close()
   print(key)

   iv = Random.new().read(CAST.block_size)
   cipher = CAST.new(key, CAST.MODE_OPENPGP, iv)
   plaintext = data
   msg = cipher.encrypt(plaintext)


   file_out = open(output_file, "wb") # Open file to write bytes
   #file_out.write(iv) # Write the iv to the output file (will be required for decryption)
   file_out.write(msg) # Write the varying length ciphertext to the file (this is the encrypted data)
   file_out.close()
```

For cast128 encryption, key choice was derived using os.urandom(16) generator and the news report document was encrypted in OPENPGP mode and document was written to a file R11503611_cast.bin.

Data Decryption

After encryption we then decrypt the (ciphertext) encrypted bin file to its original word document format and save the decrypted file which will be used for integrity check. For the decryption we use the same choice key as no other key will work. You will get an error message shown below.

Fig4 – wrong key choice usage (padding is incorrect

```
cipher = AES.new(key,AES.MODE_CBC, iv=iv)
original_data = unpad(cipher.decrypt(ciphered_data), AES.block_size)
file_out = open(output_file, "wb") # Open file to write bytes
file_out.write(original_data) # Write original data to the file (this is the original data)
file_out.close()
print(key)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-81-782d91f3b123> in <module>
     16
     17 cipher = AES.new(key,AES.MODE_CBC, iv=iv)
---> 18 original_data = unpad(cipher.decrypt(ciphered_data), AES.block_size)
     19 file_out = open(output_file, "wb") # Open file to write bytes
     20 file_out.write(original_data) # Write original data to the file (this is the original data)

~\Anaconda3\lib\site-packages\Crypto\Util\Padding.py in unpad(padded_data, block_size, style)
     88         padding_len = bord(padded_data[-1])
     89         if padding_len<1 or padding_len>min(block_size, pdata_len):
---> 90             raise ValueError("Padding is incorrect.")
     91         if style == 'pkcs7':
     92             if padded_data[-padding_len:]!=bchr(padding_len)*padding_len:

ValueError: Padding is incorrect.
```

The decrypted files will be written into R11503611_aes_decrypted_docx and R11503611_cast_decrypted.docx files when running the script functions respectively. Fig7 and Fig8 show the process of decryption. All libraries used in encryption were used in the decryption process as well.

Fig5 -AES in CBC mode decryption

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import unpad

output_file = 'R11503611_aes_decrypted.docx'
input_file = 'R11503611_aes.bin' # Input file

file = open('key2', 'rb')
key = file.read() # The key will be type bytes
file.close()

# Read the data from the file
file_in = open(input_file, 'rb') # Open the file to read bytes
iv = file_in.read(16) # Read the iv out - this is 16 bytes long
ciphered_data = file_in.read() # Read the rest of the data
file_in.close()

cipher = AES.new(key,AES.MODE_CBC, iv=iv)
original_data = unpad(cipher.decrypt(ciphered_data), AES.block_size)
file_out = open(output_file, "wb") # Open file to write bytes
file_out.write(original_data) # Write original data to the file (this is the original data)
file_out.close()
print(key)
```

```
b'C\xc2\xd2\x9a?q_v\x93\xbd\xcb\xc8V\x9c\xf2='
```

Fig6 – CAST in decryption in OPENPGP mode

```
from Crypto.Cipher import CAST
from Crypto import Random

output_file = 'R11503611_cast_decrypted.docx' # Output file
input_file = 'R11503611_cast.bin'

with open(input_file, 'rb') as f:
    data = f.read()

file = open('castkey', 'rb')
key = file.read() # The key will be type bytes
file.close()

#key = b'\xda\xe6\xc0\xcc\x83\xdaC\x9a4H\x87W\xc9\xe1\xacN'
eiv = data[:CAST.block_size+2]
ciphertext = data[CAST.block_size+2:]
cipher = CAST.new(key, CAST.MODE_OPENPGP, eiv)
document = cipher.decrypt(ciphertext)

file_out = open(output_file, "wb") # Open file to write bytes
#file_out.write(iv) # Write the iv to the output file (will be required for decryption)
file_out.write(document) # Write the varying length ciphertext to the file (this is the encrypted data)
file_out.close()
print(key)
```

Data Integrity Verification

The final step of this report is for data integrity. Encryption provides confidentiality by preventing unauthorized disclosure. Integrity helps in detecting unauthorized writing and modification ensuring no intruder has tampered with the original plaintext. It is of the assumptions that the hash functions used in part one of the execution process will stand as a benchmark to confirm this. The hexadecimal digest value we generated from the original file will be used for this purpose.

The same process of hash function will be applied on one of the decrypted data files for comparison which is shown below in fig9

Fig7 – Integrity verification with decrypted Aes word document

```
from Crypto.Hash import SHA256

input_file = b'R11503611_data.docx'

with open(input_file, 'rb') as f:
    data = f.read()

h = SHA256.new(data)
h.hexdigest()
```

```
'a0699bb21389615336dfdd240d1b252413d30a96a6cdd5425cdc9183c87e5f15'
```

```
from Crypto.Hash import SHA256

input_file = b'R11503611_aes_decrypted.docx'

with open(input_file, 'rb') as f:
    data = f.read()

h2 = SHA256.new(data)
h2.hexdigest()
```

```
'a0699bb21389615336dfdd240d1b252413d30a96a6cdd5425cdc9183c87e5f15'
```

From fig7 it shows that after applying same SHA-256 hash function on decrypted aes file, the hexadecimal value is the same. The hash functions take input analysis of any size plaintext and produce output of a fixed sized. If the same document is encrypted with different a cipher, when it is decrypted, they and hash function applied they produce same hexadecimal value as well.

R11503611_aes_decrypted.docx Hexadecimal value –
'a0699bb21389615336dfdd240d1b252413d30a96a6cdd5425cdc9183c87e5f15'

Hashed original R11503611 SHA-256 value –
'a0699bb21389615336dfdd240d1b252413d30a96a6cdd5425cdc9183c87e5f15'

Conclusion

With this the receiver of the message that has a pair of symmetric keys, can ensure the integrity of the plaintext as nothing was tampered with or modified in the process of transmission.