

Image Encryption by RSA Algorithm upon Image Compression by DCT and Huffman Coding

(July - 19 -2022)

Albert Tsang Chun Ching, *MSc Student in INFORMATION TECHNOLOGY*

Abstract— Image Encryption and Image Compression are important for Image Transmission from Senders to Receivers (I will use Verifiers equivalent to Receivers in the context of this paper.) Image Encryption can guarantee the transmission security while Image Compression can reduce the bandwidth required for the transmission. This individual assignment will study the feasibility of Image Encryption by RSA Algorithm upon Image Compression by DCT and Huffman Coding.

Index Terms — # Image Encryption. # Image Compression. #Huffman Coding. # RSA Algorithm

https://github.com/atsangcc/RSA_DCT_HuffmanCoding.git

<https://www.youtube.com/watch?v=VM4CL5XHZqI>

1 INTRODUCTION

THIS individual assignment was inspired by COMP5422 LAB#3 hosted by Mr. Chengdong Dong in 2022 Semister 3. In LAB#3, a simple password key value is applied to a water-marked image for securing a particular image and also validating the image ownership by a embedded watermark. In this individual assignment, I propose to use RSA Algorithm for validating the ownership of a particular 256x256 image and, at the same time, DCT and Huffman Coding will also be applied for Image Compression.

2 PROPOSED WORKFLOW

2.1 RSA Key Pairs Generation

RSA Key Pairs Generation starts from randomly choosing two different prime numbers; and then by the following mathematics, we can have a pair of private key and public key generated from the sender's side.

Key Generation Procedure:	
Choose two distinct large random prime numbers p & q such that $p \neq q$.	Encryption :
Calculate $n = p \times q$	Plaintext Message $< n$
Calculate $\Phi(n) = (p-1)(q-1)$	Ciphertext $C = \text{Message}^e \bmod n$
Choose an integer e such that $\gcd(\Phi(n), e) = 1$; $1 < e < \Phi(n)$	Decryption :
Compute d to satisfy the congruence relation $d = e^{-1} \bmod \Phi(n)$; d is kept as private	Ciphertext C
Public Key $KU = \{e, n\}$	Plaintext Message $= C^d \bmod n$
Private Key $KR = \{d, n\}$	
The public key is (n, e) and the private key is (n, d) . Keep d, p, q and Φ secret.	

Fig. 1. RSA Key Pair Generation Mathematics [1]

2.2 Sender's Side Compression and Encryption

The following flowchart depicts how the Sender

compresses and encrypts a particular 256x256 image.

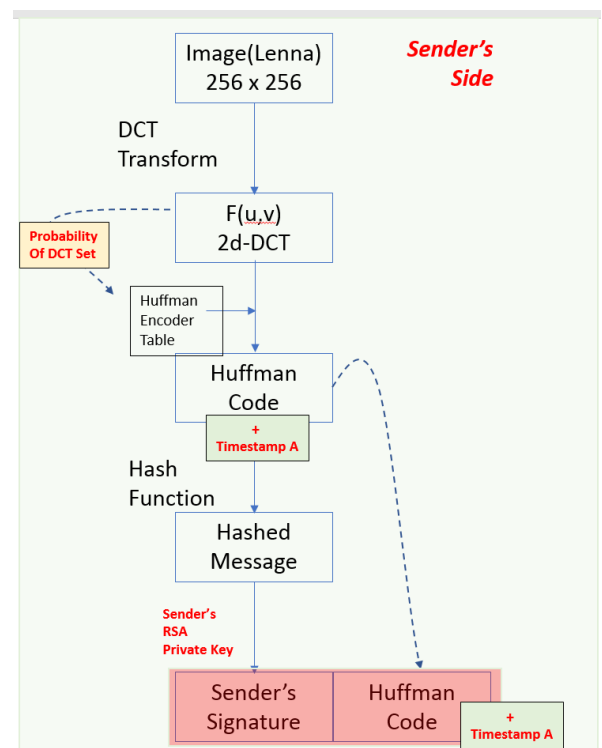


Fig. 2. Sender's Side Compression and Encryption

Assuming the Sender has an 256x256 color image, before he sends out the image, he will first conduct DCT Transform towards the original image and then have an 2d-DCT image. We will then compute a probability set from the DCT image which it will also act as the Huffman Encoder Table for us encoding the DCT matrix into the

Huffman Code. And then the timestamp at that moment will be embedded in the Huffman code and then the whole object will be hashed. After that, the sender will use his RSA Private key to sign on this hashed message (i.e. Huffman code+timestamp) to produce a One-Time Sender Digital Signature. This One-Time Sender Digital Signature will be sent together with the Huffman code to the Receiver's side (i.e. the verifier's side).

2.3 Verifier's Side Decryption and Decompression

The following flowchart depicts how the Receiver's side (i.e. the verifier's side) decrypts and decompresses the received package.

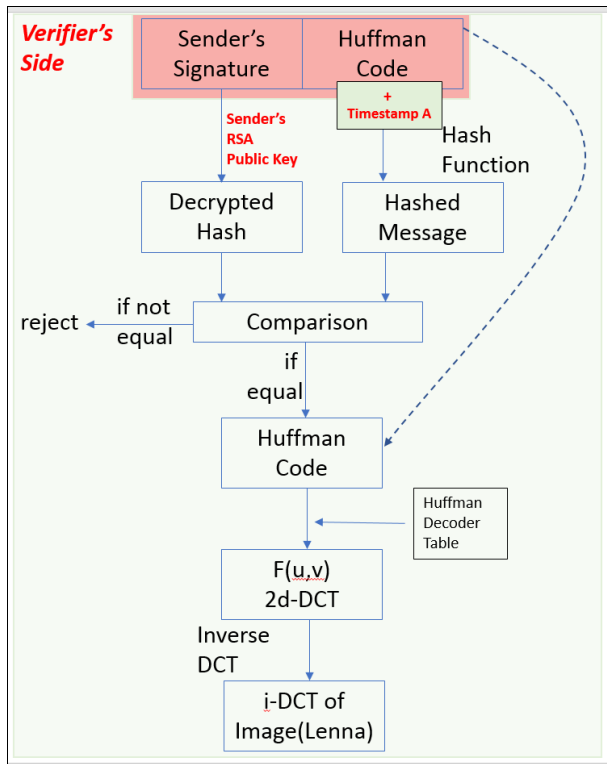


Fig. 3. Verifier Side Decryption and Decompression

When the verifier gets the sender's one time digital signature with the Huffman code. The verifier will use the sender's public key to decrypt sender's one time digital signature and get back the hash value. Then, comparison will be conducted between the hash value of decrypted signature and the hash value of the "Huffman code+timestamp". If they are the same, Huffman decoding and inverse DCT will be conducted accordingly to recover the image in gray level. If such verification fails, decoding process will stop (i.e. rejected).

3 IMPLEMENTATION

In this individual assignment, PYTHON will be used for implementing the ideas proposed in Section#2.

3.1 Folder Structure of my Individual assignment

There will be two python files simulating the Sender Application and the Verifier Application.

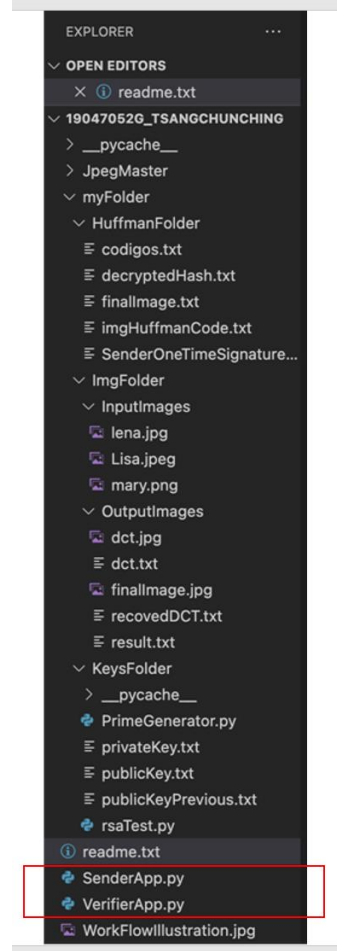


Fig. 4. Folder Structure containing SenderApp.py and VerifierApp.py

SenderApp.py will execute the workflow proposed in Fig2 while VerifierApp.py will execute the workflow proposed in Fig3. Commands for running the applications will be:

- python SenderApp.py ; and
- python VerifierApp.py .

Results from these two executions will be discussed in Section#4

3.2 RSA Key Pairs Generation in Library

The rsaTest.py is modified from a github library created by ErbaAitbayev[3]. Instead of two fixed primary number values, I advance the ErbaAitbayev's library to rsaTest.py where two random prime numbers will be used. So that the key pairs alternate every time for a new image.

3.3 DCT and Huffman Coding

“Jpeg-master” with four python files is also a github library ; it was created by Edgard Diaz[2]. I only choose to use some of the methods from “Jpeg-master” which are in my concerns of this project. Overall, “Jpeg-master” can well address the the solution for Huffman encoding and decoding ; while it also provides methods towards DCT Transform and inverse DCT Transform. However, thereare two limitations that “Jpeg-master” is only able to handle 256x256 images and the recoved images are in gray level after decoding and decompressing. My individual assignemt is also inherited with the limitations of the “Jpeg-master” library. But it is enough to verify my hypothesis proposed in Section#2.

4 RESULTS

4.1 Overall Results

TABLE 1
LENA VS LISA VS MARY



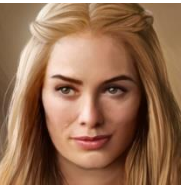

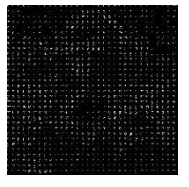




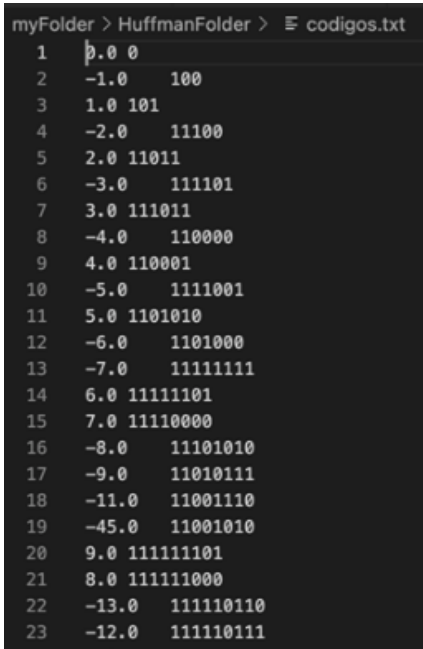
		
Original Lena	Original Liza	Original Mary
		
Lena's DCT	Liza's DCT	Mary's DCT
		
Recovered Img	Recovered Img	Recovered Img

Table 1 shows the results of three 256x256 rgb images being converted to their corresponding DCT images and recovered back to gray-level images if verifications are successful.

4.2 Huffman Encoder Table

The following screencapture shows a selected portion of the Huffman Encoder Table of the Lena’s DCT which is used for Huffman coding:



1	0
2	-1.0 100
3	1.0 101
4	-2.0 11100
5	2.0 11011
6	-3.0 111101
7	3.0 111011
8	-4.0 110000
9	4.0 110001
10	-5.0 1111001
11	5.0 1101010
12	-6.0 1101000
13	-7.0 1111111
14	6.0 11111101
15	7.0 11110000
16	-8.0 11101010
17	-9.0 11010111
18	-11.0 11001110
19	-45.0 11001010
20	9.0 111111101
21	8.0 111111000
22	-13.0 111110110
23	-12.0 111110111

Fig. 5. Huffman Encoder Table of the Lena’s DCT

4.3 Signature Verification

The following screencapture shows successful verifications if the hash value of decrypted signature are equal the hash value of the “Huffman code+timestamp”:

```
(base) Alberts-MBP:19047052g_TsangChunChing alberttsang$ python SenderApp.py
Hash Value of Image Huffman Code + Timestamp
988acb0a1bf94499133852e5cb9b2e6ff66eb6741c558f34852b4a7cd2884a2
(base) Alberts-MBP:19047052g_TsangChunChing alberttsang$ python VerifierApp.py
The decrypted HASH message is:
988acb0a1bf94499133852e5cb9b2e6ff66eb6741c558f34852b4a7cd2884a2
Verification successful:
988acb0a1bf94499133852e5cb9b2e6ff66eb6741c558f34852b4a7cd2884a2 = 988acb0a1bf94499133852e5cb9b2e6ff66eb6741c558f34852b4a7cd2884a2
```

Fig. 6. Successful Verifications

If the verifier mistakenly uses an outdated public-key file(e.g. publicKeyPrevious.txt), he will fail to verify the identity and the decoding process will also stop. The following picture illustrates such a situation.



```
(base) Alberts-MBP:19047052g_TsangChunChing alberttsang$ python VerifierApp.py
Verification failed
Invalid Authentication !!!
(base) Alberts-MBP:19047052g_TsangChunChing alberttsang$
```

Fig. 7. Verifications fails due to Outdated Public Key File

For bigger screeptature of Fig.7 , please refer to Fig.8 at the end of this report.

5 CONCLUSION

Overall, the python works in this individual assignment can primarily verify my ideas of Image Encryption by RSA Algorithm upon Image Compression by DCT and Huffman Coding. In the future, I may spend time on modifying the "Jpeg-master" library such that it can also handle images in a wide range of resolutions not only limited to 256x256. And this individual assignment may also base my personal interests in NFT by which we can manage the artwork ownership by using decentralized cryptographic technologies.

ACKNOWLEDGMENT

This individual assignment was inspired by COMP5422 LAB#3 hosted by Mr. Chengdong Dong.

REFERENCES

- [1] O. F. A. Wahab, A. A. M. Khalaf, A. I. Hussein and H. F. A. Hamed, "Hiding Data Using Efficient Combination of RSA Cryptography, and Compression Steganography Techniques," in IEEE Access, vol. 9, pp. 31805-31815, 2021, doi: 10.1109/ACCESS.2021.3060317.
- [2] Compression of images with JPEG algorithm using python and Huffman codes <https://github.com/josgard94/JPEG>
- [3] Simple Python RSA for digital signature with hashing implementation <https://gist.github.com/ErbaAitbyev/8f491c04af5fc1874e2b0744965a732b>
- [4] COMP5422 Lecturer's Notes : [Fundamentals of Data Compression] + [Image Compression Part 1] by Prof. Ajay Kumar
- [5] COMP5422 LAB#3 : [Watermarks on Image] by Mr. Chengdong Dong
- [6] THE HONG KONG POLYTECHNIC UNIVERSITY COMP 5521 Lecturer's Notes : [Digital Signatures] by Prof. Bin Xiao

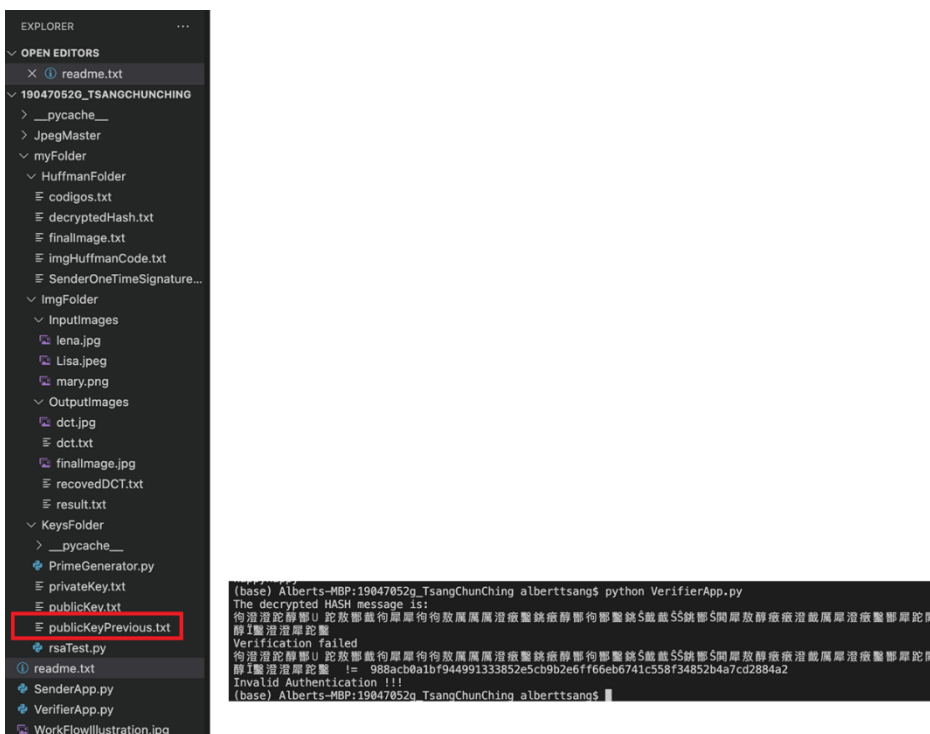


Fig. 8. Bigger Screen Capture of Fig. 7. - Verifications fails due to Outdated Public Key File