Reversible Data Hiding in Encrypted Images with Distributed Source Encoding: Implementation and Experiments

Nagham F.(M.R.) Hamad

Submitted to the
Institute of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Eastern Mediterranean University
January 2018
Gazimağusa, North Cyprus

Approval of the Institute of Graduate Studies and Research

Assoc. Prof. Dr. Ali Hakan Ulusoy
Acting Director

I certify that this thesis satisfies the requirements as a thesis for the degree of Master of Science in Computer Engineering.

Prof. Dr. Işık Aybay
Chair, Department of Computer Engineering

We certify that we have read this thesis and that in our opinion it is fully adequate in scope and quality as a thesis for the degree of Master of Science in Computer Engineering.

Assoc. Prof. Dr. Alexander Chefranov
Co-Supervisor

Assoc. Prof. Dr. Gürcü Öz
Supervisor

Examining Committee

1.Assoc. Prof. Dr. Alexander Chefranov

2.Assoc. Prof. Dr. Gürcü Öz

3.Asst. Prof. Dr. Adnan Acan

4.Asst. Prof. Dr. Mehtap Köse Ulukök

5.Asst. Prof. Dr. Ahmet Ünveren

ABSTRACT

In this thesis, we implemented and investigated Qian-Zhang reversible data hiding scheme proposed in 2016. Qian-Zhang scheme uses Slepian-Wolf encoding based on Low-Density Parity-Check (LDPC) codes to compress selected most significant bits (MSB) from an encrypted image to vacate room for embedding additional data. Compressing process depends on LDPC matrix, H, $r<n$, where r is number of rows and n is number of columns. After extracting embedded data, the original image can be recovered by applying iterative decoding algorithm. We found that the quality of the recovered image depends on the construction method, size, and ratio $R=r/n$. We implemented Qian-Zhang scheme using H matrices constructed by two methods, Gallager and MacKay-Neal, having different sizes and ratios. We evaluated Qian-Zhang scheme with these matrices using decoding time, embedding capacity, and quality of the recovered image, approximate and decoded, by Peak Signal-to-Noise Ratio (PSNR). We get a formula for embedding capacity dependence on the number of bits to be compressed and value of R. In addition, we investigated relation between PSNR of an approximate image and embedding capacity. Changing of the embedding capacity does not affect PSNR of the approximate image. Since we used other H matrices than the one used by Qian-Zhang, we obtained not exactly same PSNR and embedding capacity but close to the values of Qian-Zhang. In addition, we investigated the PSNR of decoded image when decoding fails. The PSNR decreases when the embedding capacity increases.

We found that fixing ratio, R, and increasing size of H leads to the increase of the PSNR of the recovered image. On the other hand, the time of decoding increases with the matrix size growth. These results may be used for choosing suitable H matrix size to meet specified decoding time. We investigated relation between the ratio, R , and embedding capacity. Decreasing of R leads to the increase of the embedding capacity. We investigated relation between R and PSNR of the decoded image. Decreasing of R leads to the decrease of the PSNR. Our results show better embedding capacity than that in the Qian-Zhang's paper due to the use of different size H matrices.

Keywords: Reversible data hiding, Slepian-Wolf encoding, Low-Density Parity-Check (LDPC) code, LDPC matrix , Most Significant Bit (MSB), Distributed Source Decoding (DSD), Selection ratio, Embedding capacity, Host image, Approximate image, Decoded image, Peak Signal-to-Noise Ratio (PSNR).

ÖZ

Bu tezde, Qian-Zhang tarafından 2016 yılında önerilen geri dönüşümlü veri gizleme düzeni uygulanmış ve incenlenmiştir. Qian-Zhang düzeni, Düşük Yoğunluklu Eşlik Kontrolünü (LDPC) baz alan Slepian-Wolf kodlama yöntemini kullanmıştır. Bu yöntemde, LDPC matrisi, $H_{r \times n}$, $r < n$ (r satır sayısı ve n sütun sayısı) kullanılarak ek veri gömme işlemi için yer açmak amacı ile şifrelenmiş görüntüden seçilen en önemli bitler (MSB) sıkıştırılmıştır. Gömülmüş veri çıkartıldıktan sonra, orijinal görüntü yinelemeli kod çözme algoritması uygulayarak kurtarılabilir. Kurtarılan görüntünün kalitesinin yapı yöntemine, H matrisinin büyüklüğüne ve $R = r / n$ oranına bağlı olduğunu tespit ettik. Gallager ve MacKay-Neal yöntemleri kullanılarak oluşturulan farklı boyut ve orandaki H matrislerini kullanarak Qian-Zhang düzenini uyguladık. Bu matrisleri kullanarak, Qian-Zhang düzenini, çözülme süresi, gömme kapasitesi ve kurtarılan görüntünün kalitesini yaklaşık olarak ve çözülmüş olarak, Tepe Sinyal-Gürültü Oranı (PSNR) kullanarak değerlendirdik. Gömme kapasitesinin sıkıştırılacak bit sayısı ve R değerine bağımlılığı ile ilgili bir formül elde ettik. Buna ek olarak yaklaşık görüntü PSNR'si ve gömme kapasitesi arasındaki ilişkiyi araştırdık. Gömme kapasitesinin değiştirilmesi, yaklaşık görüntünün PSNR'sini etkilemediğini gördük. Qian-Zhang tarafından kullanılanlardan daha farklı H matrisleri kullandığımızdan, PSNR ve gömme kapasitesi tam olarak Qian-Zhang sonuçları ile aynı değildi fakat yakındı. Buna ek olarak, kod çözme başarısız olduğunda şifresi çözülmüş görüntünün PSNR'sini araştırdık. Gömme kapasitesi arttıkça PSNR değerinin azaldığını gözlemledik.

Elde edilen sonuçlara göre, *R* oranının sabitlenmesi durumunda *H*'nin boyutunun artırılması kurtarılan görüntünün PSNR'sinin arttığını gördük. Öte yandan, şifre çözme süresinin, matris boyutuna göre büyümesini gözlemledik. Bu sonuçlar, belirtilen kod çözme süresini karşılamak için uygun *H* matris boyutunu seçmek için kullanılabilir. Seçme oranı, *R* ve gömme kapasitesi arasındaki ilişkiyi araştırdık. *R*'nin azalması gömme kapasitesinin artmasına yol açar. Çözülen görüntüdeki *R* ve PSNR arasındaki ilişkiyi araştırdık. *R*'nin azalması PSNR'nin azalmasına neden olur. Sonuçlarımız farklı boyutlardaki *H* matrislerinin kullanılması nedeniyle Qian-Zhang'ın sonuçlarından daha iyi gömme kapasitesi göstermektedir.

Anahtar Kelimeler: Geri döndürülebilir veri gizleme, Slepian-Wolf kodlaması, Düşük yoğunluklu eşlik kontrolü kodu, LDPC matrisi, En önemli bit, Dağıtılmış kaynak kod çözme, Seçme oranı, Gömme kapasitesi, Kaplama resmi, Yaklaşık görüntü, Çözülmüş görüntü, Tepe sinyal-gürültü oranı.

DEDICATION

To my supporting man, Anas, who is always being besides, supporting and encouraging me.

To my son, Maher and my daughter, Sana, your smiles light my way.

To the best woman in the world, Mom. Your words “you can do” was the light in the dark moments.

To my great Dad, your supporting and believing in me pushing me to be successful.

To my sisters and brothers, Deema, Rushdi, Wala’, Hisham and Raghad, thank you for your supporting.

To my friends in Famagusta, Basma and Amani, you were part of my family.

ACKNOWLEDGMENT

I thank Allah so much for giving me strength and patience to finish my master. I dedicate my success to my husband who always besides me. I would like to thank my Assoc. Prof. Dr. Alexander Chefranov, and Assoc. Prof. Dr. Gürcü Öz for their support and advices that leaded me in writing my thesis.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

CB	Collected Bits
DSD	Distributed Source Decoding
EI	Encrypted Image
K_{ENC}	Encryption Key
K_{SF}	Shuffle Key
K_{SL}	Selection Key
KGs	K Groups
LDPC	Low Density Parity Check code
LLR	Log-Likelihood Ratio
LSB	Least Significant Bit
MEI	Marked Encrypted Image
MSB	Most Significant Bit
PSNR	Peak Signal-to-Noise Ratio
RDH	Reversible Data Hiding
SB	Selected Bits
SGs	Syndrome Groups
SHB	Shuffled Bits

Chapter 1

INTRODUCTION

Reversible Data Hiding (RDH) is a technique of fully recovering a host image after extracting embedded secret data. RDH has been emerged in the last few years in many areas such as military and medical reports [2]. Qian and Zhang [1] proposed an RDH scheme that embeds secret data in an encrypted gray scale image using Slepian-Wolf encoding [3] based on LDPC matrix, H. Qian-Zhang scheme [1] is used in [4] [5] [6] [7] [8].

Qian-Zhang scheme [1] encrypts an original gray scale image, O, using a stream cipher encryption with an encryption key, K_{ENC}. Then, with a selection key, K_{SL}, a data hider chooses the Most Significant Bits (MSBs) of the encrypted image depending on a selection ratio, α. These selected bits are scrambled using a shuffle key, K_{SF}. Then, the room for embedding secret data is vacated using Low Density Parity Check Codes (LDPC) matrix [9] , H, $r < n$, where r is number of rows and n is number of columns, by compressing the selected MSBs. The secret data bits are embedded in the vacated room.

On the receiver side, using both selection key K_{SL} and shuffle key K_{SF}, the embedded secret data bits are extracted perfectly. Having an encryption key, K_{ENC}, the approximated original image, O_{approx}, is constructed using the bilinear interpolation.

The original image, O, can be recovered perfectly using O_{approx}, compressed bits, and H matrix via iterative decoding method.

In this thesis, we implemented and investigated Qian-Zhang method [1]. We found by experiments that the time of decoding, and the quality of the recovered image depend on the size of H matrix and the matrix generation method. The matrix H is not exactly specified neither in [1], nor in its reference [10]. Thus, we have generated different H matrices using two different construction methods: Gallager [11] and MacKay-Neal [12], and conducted experiments to evaluate the performance of the method [1]. We evaluated our matrices using decoding time, embedding capacity, and quality of the recovered image (Peak Signal-to-Noise Ratio (PSNR) of the approximate and decoded images).

Our experiments show that matrices generated using Gallager construction method have better decoding time than for MacKay-Neal method. Thus, we use Gallager method for conducting experiments specified in [1] and in extensions of these experiments. By rerun experiments specified in [1], we obtained the following two results which comply with results in [1]:

1. Approximate image construction does not depend on the secret data or embedding capacity.
2. When decoding fails, there is inverse relation between embedding capacity and PSNR of the decoded image.

To extend experiments in [1], we generated different H matrices with different ratios, $R = r / n \in \{0.5, 0.3, 0.25, 0.2\}$. We found inverse relation between embedding capacity and ratio, R. We also conducted and experiment by fixing R=0.5 and generating 9 different matrices with sizes $\{4\times 8, 8\times 16, ..., 1024\times 2048\}$; we found a proportional relation between the matrix size and the PSNR of the decoded image and decoding time. These results may be used for choosing suitable *H* matrix size to meet specified decoding time.

Results obtained on the PSNR and time dependence on the matrix size may be used for making decisions on the Qian-Zhang scheme [1] parameters selection that is not done in [1]. In addition, these results are used in comparison with the other methods to evaluate the performance of the proposed method [1] with our extended experiments.

The remaining part of the thesis is organized as follows. Chapter 2 indicates the recent studies about RDH methods which are used for comparison and the problem definition. Chapter 3 explains the Qian-Zhang scheme implementation. In Chapter 4, the experimental settings and results are discussed. Finally, in the last Chapter 5, the study is concluded. Appendices contains codes implementing the scheme and results of the experiments conducted.

Chapter 2

RELATED WORK AND PROBLEM DEFINITION

In this chapter we explain in details the Qian-Zhang scheme [1] which we implemented and investigated. In addition, we briefly explain RDH schemes [2], [6], [13], [14] that we compare our results with.

2.1 Qian-Zhang RDH Scheme

Qian-Zhang [1] scheme is divided into three stages: encryption, data hiding, and data extraction and image recovery. Figure 1 illustrates Qian-Zhang scheme. Encryption stage occurs at the sender side, data hiding stage occurs at the data hider side, while data extraction and image recovery stage occurs at the receiver side.

2.1.1 Stage 1: Image Encryption Details

This stage occurs at the sender side. Sender is assumed to use an original gray scale image, O, with size $X \times Y$ pixels, and the value of a pixel is between 0 and 255; both X and Y are power of two.

Initially, the original image pixels, $O_{i,j}$, are converted into binary values as follows:

$$b_{i,j,u} = \lfloor O_{i,j}/2^{u-1} \rfloor \, mod \, 2 \quad (1)$$

where $b_{i,j,u}$ is the u-th bit of ij-th pixel binary value, $u = 1,2,\ldots,8$ and $1 \le i \le X,\ 1 \le j \le Y$.

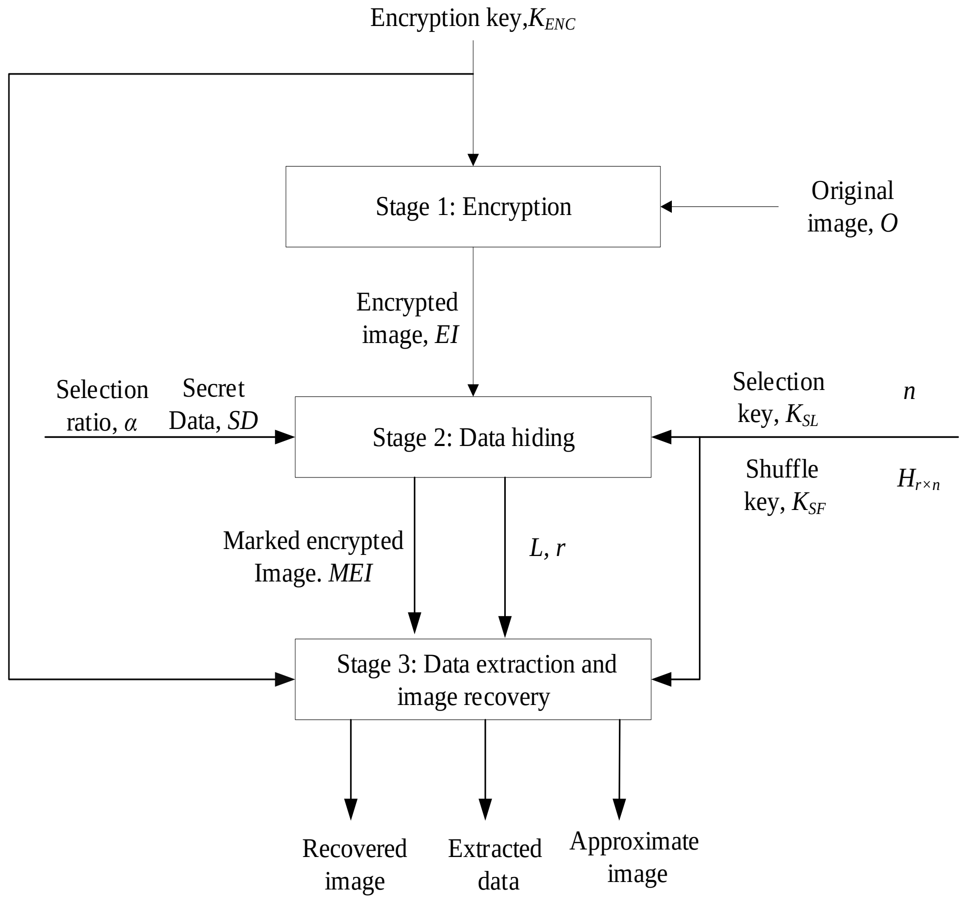


Figure 1: Qian-Zhang Scheme Stages. The Stages are: Encryption, Data Hiding, and Data Extraction and Image Recovery

Sender uses the encryption key, K_{ENC}, of the size X×Y×8 to create encrypted bit stream as follows:

$$e_{i,j,u} = b_{i,j,u} \oplus K_{ENC_{i,j,u}} \tag{2}$$

where $K_{ENC_{i,j,u}}$ is the iju-th bit of the encryption key, K_{ENC} , $e_{i,j,u}$is the iju-th encrypted bit, and $\oplus$ denotes exclusive-or (XOR) operation, $u = 1,2,\ldots,8$, $1 \le i \le X$, $1 \le j \le Y$.

Encryption key, K_{ENC}, construction is not clearly specified in [1], hence, we define it as our implementation problem and generate it as specified in Section 4.2.

Finally, encrypted bits are converted into pixel values to generate the encrypted image, *EI*, with the size of the original image, *O*, using (3):

$$EI_{i,j} = \sum_{u=1}^{8} eI_{i,j,u} \,.\, 2^{u-1} \quad (3)$$

where $EI_{i,j}$ are the pixel values of the encrypted image, EI, $1 \le i \le X$, $1 \le j \le Y$. Figure 2 illustrates image encryption described also by Algorithm 1. Example 1 shows the steps of encryption.

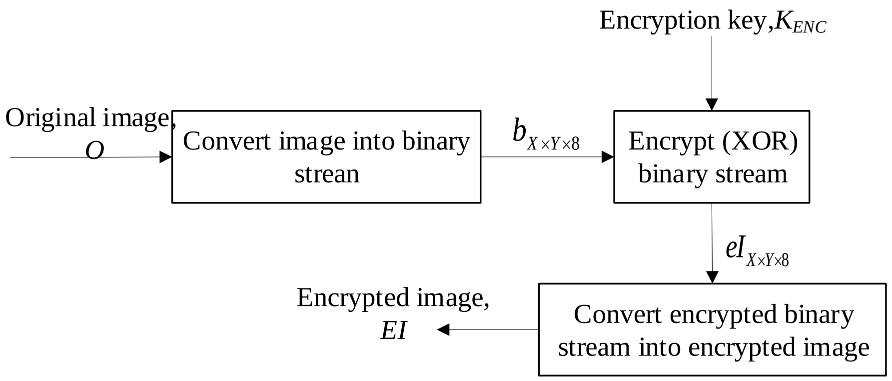


Figure 2: Stage1: Image Encryption Details

Example 1. Stage1: Encryption stage (Figure.1) example.

Let’s consider an original grayscale image O with size 8×8 and encryption key K_{ENC} with size 64×8.

Input:

– Original image O

– Original image O :

$$\begin{bmatrix} 15 & 215 & 5 & 183 & 3 & 12 & 100 & 10 \\ 125 & 7 & 190 & 20 & 10 & 4 & 93 & 102 \\ 62 & 2 & 31 & 85 & 242 & 121 & 10 & 1 \\ 121 & 17 & 8 & 3 & 102 & 18 & 52 & 130 \\ 78 & 108 & 5 & 150 & 27 & 70 & 140 & 175 \\ 201 & 200 & 27 & 5 & 95 & 41 & 89 & 210 \\ 108 & 3 & 28 & 172 & 55 & 100 & 48 & 72 \\ 88 & 130 & 64 & 81 & 117 & 82 & 94 & 1 \end{bmatrix}$$

Algorithm 1. Stage1: Encryption (Figure 2) algorithm.

Input:

- X, Y: powers of 2;
- O: the original image, sized $X \times Y$;
- K_{ENC}: encryption key of the size $X \times Y \times 8$.

Output:

- EI: encrypted image, sized $X \times Y$.

Steps:

1. Convert pixels in O into binary values using equation (1).
2. Encrypt the binary values with encryption key using equation (2).
3. Convert the encrypted binary values into the pixel values using (3) to generate encrypted image EI with size X and Y.

– Encryption key K_{ENC}:

$$\begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ & & & & \vdots & & & \\ & & & & \vdots & & & \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Output**:**

- *EI*: encrypted image, sized $X \times Y$.

Steps:

1. Using (1), *O* is converted into binary values in row major order as follows:

$$\text{Binary stream: } \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ & & & & \vdots & & & \\ & & & & \vdots & & & \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

2. Binary stream is encrypted using encryption key K_{ENC} to obtain encrypted binary stream using (2).

$$\text{Encrypted binary stream} \begin{bmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ & & & \vdots & & & & \\ & & & \vdots & & & & \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{bmatrix}$$

3. Encrypted image *EI* is obtained by converting encrypted binary stream into pixel values with size $X \times Y$.

$$\text{Encrypted image } EI\text{:} \begin{bmatrix} 219 & 92 & 100 & 240 & 3 & 63 & 160 & 218 \\ 200 & 125 & 74 & 229 & 68 & 8 & 193 & 61 \\ 67 & 177 & 142 & 50 & 78 & 247 & 77 & 176 \\ 181 & 2 & 34 & 149 & 233 & 188 & 155 & 207 \\ 121 & 32 & 137 & 171 & 152 & 168 & 57 & 227 \\ 195 & 110 & 209 & 75 & 13 & 162 & 251 & 212 \\ 255 & 189 & 74 & 185 & 254 & 58 & 93 & 153 \\ 90 & 242 & 116 & 138 & 156 & 81 & 229 & 57 \end{bmatrix}$$

2.1.2 Stage 2: Data Hiding Details

Data hider embeds secret data, *SD*, in the encrypted image *EI* by three phases:

– Most Significant Bits (MSBs) selection,

– Encoding and compressing,

– Embedding secret data phase.

Each phase will be described in details in next sections. Figure 3 shows the three phases of the Stage 2: Data hiding (Figure 1). In MSBs selection phase, MSBs in the encrypted image, *EI*, are collected. Using a selection key, K_{SL}, respective number of bits is selected. Then, the selected bits are shuffled using a shuffle key, K_{SF}, to get shuffled bits, *SHB*. In encoding and compressing phase, the shuffled bits are divided into *K* groups, each group contains *n* bits. Then, these groups are encoded using binary *H* matrix with size $r \times n$. In embedding phase, secret data are embedded into syndrome groups, *SGs*, and reverse shuffled using K_{SF} to construct marked encrypted image, *MEI*.

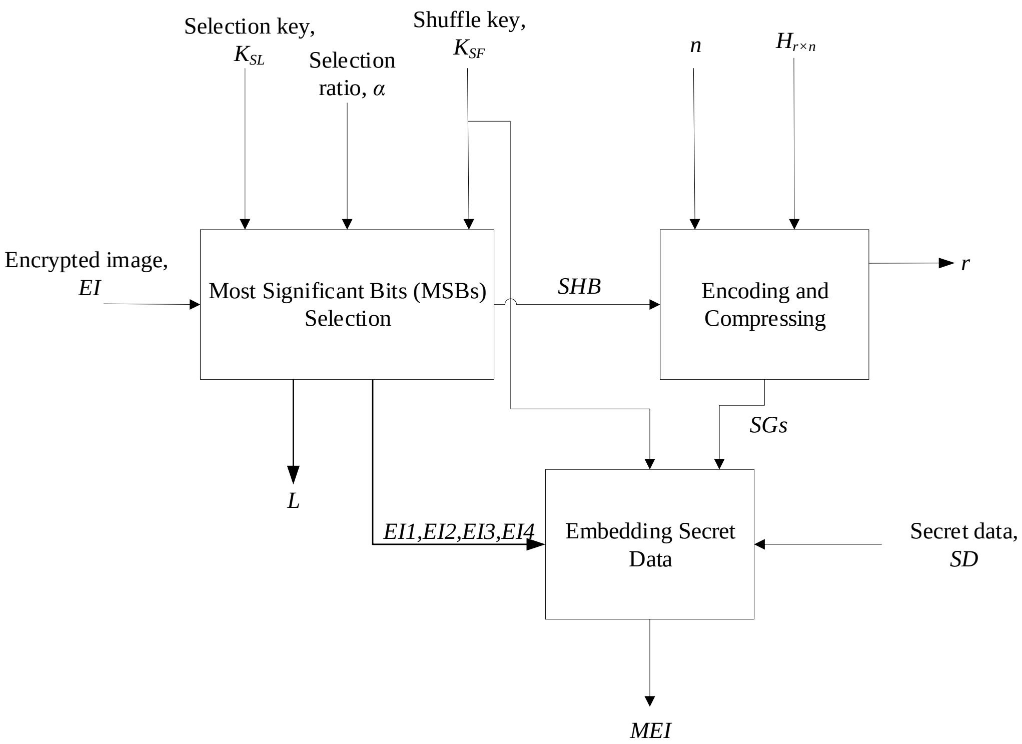


Figure 3: Stage 2: Data Hiding Details

Most Significant Bits (MSBs) Selection Details

In this phase, encrypted image *EI* is decomposed into 4 segments *EI1*, *EI2*, *EI3*, *EI4* defined by (4). Then, MSBs are collected from, *EI2*, *EI3*, *EI4*, forming collected bits, *CB*. After that, a number of bits, *SB*, are selected from collected bits, *CB*, using selection key, K_{SL}, and selection ratio, α. Then, the selected bits, *SB*, are shuffled using K_{SF}. Figure 4 shows diagram of Phase 1.

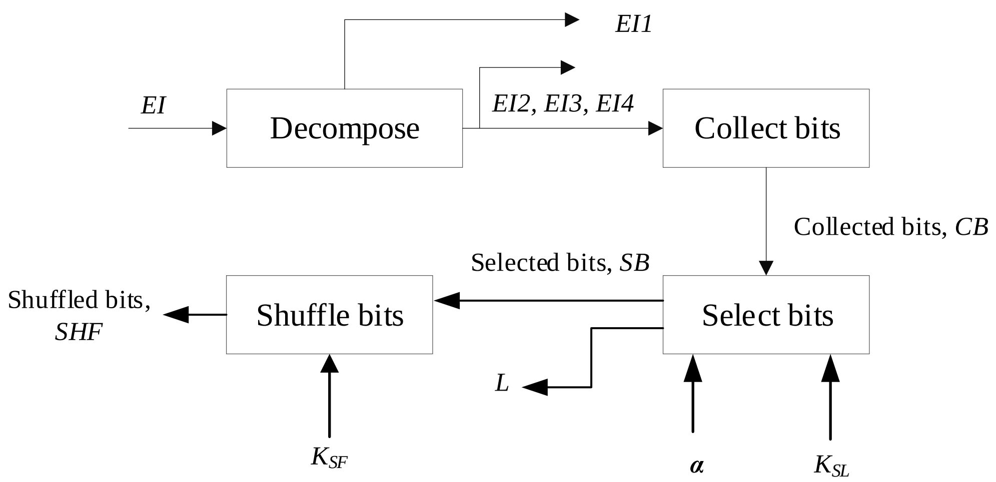


Figure 4: Most Significant Bits (MSBs) Selection Phase Details

Encrypted image *EI* is decomposed into four segments *EI1*, *EI2*, *EI3* and *EI4*, each of them with size $(X / 2) \times (Y / 2)$ as follows:

$$\begin{cases} EI1(i,j) = EI(2i-1, 2j-1) \\ EI2(i,j) = EI(2i-1, 2j) \\ EI3(i,j) = EI(2i, 2j-1) \\ EI4(i,j) = EI(2i, 2j) \end{cases} \qquad \begin{array}{l} i = 1, 2, \ldots, X/2 \\ j = 1, 2, \ldots, Y/2 \end{array}. \tag{4}$$

Decompose (Figure 4) pseudo code is described in Algorithm 2 and Example 2 illustrates the decomposing process (4).

Algorithm 2. Pseudocode of the Decompose (Figure 4) an encrypted image into four segments

Input:

- X, Y: power of 2.
- *EI*: encrypted image, size $X \times Y$.

Output:

- Segments, *EI1*, *EI2*, *EI3*, *EI4*, each of the size $(X / 2) \times (Y / 2)$.

Steps:

// Using MATLAB style pseudocode:

```
EI1(1:X/2,1:Y/2)=EI(1:2:X,1:2:Y);//odd rows and columns

EI2(1:X/2,1:Y/2)=EI(1:2:X,2:2:Y);//odd rows and even columns

EI3(1:X/2,1:Y/2)=EI(2:2:X,1:2:Y);//even rows and odd columns

EI4(1:X/2,1:Y/2)=EI(2:2:X,2:2:Y);//even rows and columns

Where a:b:c means x values such that: a<=x<=c, x=c+i×b.
```

Example 2. Decompose (Figure 4) of an encrypted image into 4 segments.

Let's consider encrypted image from Example 1.

Input:

Encrypted image *EI* with size 8×8

– Encrypted image EI : $\begin{bmatrix} 219 & 92 & 100 & 240 & 3 & 63 & 160 & 218 \\ 200 & 125 & 74 & 229 & 68 & 8 & 193 & 61 \\ 67 & 177 & 142 & 50 & 78 & 247 & 77 & 176 \\ 181 & 2 & 34 & 149 & 233 & 188 & 155 & 207 \\ 121 & 32 & 137 & 171 & 152 & 168 & 57 & 227 \\ 195 & 110 & 209 & 75 & 13 & 162 & 251 & 212 \\ 255 & 189 & 74 & 185 & 254 & 58 & 93 & 153 \\ 90 & 242 & 116 & 138 & 156 & 81 & 229 & 57 \end{bmatrix}$

Output:

– Segments *EI1*, *EI2*, *EI3*, *EI4* each of the size 4×4 $(X/2) \times (Y/2)$

Steps:

1. Using (4), *EI1* is obtained by taking the pixels in odd rows and odd columns of *EI*

   1^{st} row in *EI1* is obtained by 1^{st} row and 1^{st},3^{rd},5^{th},7^{th} columns of *EI*.

   2^{nd} row in *EI1* is obtained by 3^{rd} row and 1^{st},3^{rd},5^{th},7^{th} columns of *EI*.

   3^{rd} row in *EI1* is obtained by 5^{th} row and 1^{st},3^{rd},5^{th},7^{th} columns of *EI*.

   4^{th} row in *EI1* is obtained by 7^{th} row and 1^{st},3^{rd},5^{th},7^{th} columns of *EI*.

2. Using (4), *EI2* is obtained by taking the pixels in odd rows and even columns of *EI*.

1st row in *EI2* is obtained by 1st row and 2nd, 4th, 6th, 8th columns of *EI*.

2nd row in *EI2* is obtained by 3rd row and 2nd ,4th, 6th ,8th columns of *EI*.

3rd row in *EI2* is obtained by 5th row and 2nd ,4th, 6th ,8th columns of *EI*.

4th row in *EI2* is obtained by 7th row and 2nd ,4th, 6th ,8th columns of *EI*.

3. Using (4), *EI3* is obtained by taking the pixels in even rows and odd columns of *EI*.

   1st row in *EI3* is obtained by 2nd row and 1st,3rd,5th,7th columns of *EI*.

   2nd row in *EI3* is obtained by 4th row and 1st,3rd,5th,7th columns of *EI.*

   3rd row in *EI3* is obtained by 6th row and 1st,3rd,5th,7th columns of *EI*.

   4th row in *EI3* is obtained by 8th row and 1st,3rd,5th,7th columns of *EI.*

4. Using (4), *EI4* is obtained by taking the pixels in even rows and even columns of *EI.*

   1st row in *EI4* is obtained by 2nd row and 2nd ,4th, 6th ,8th columns of *EI.*

   2nd row in *EI4* is obtained by 4th row and 2nd ,4th, 6th ,8th columns of *EI.*

   3rd row in *EI4* is obtained by 6th row and 2nd,4th, 6th ,8th columns of *EI.*

   4th row in *EI4* is obtained by 8th row and 2nd,4th, 6th ,8th columns of *EI.*

$$EI1: \begin{bmatrix} 219 & 100 & 3 & 160 \\ 67 & 142 & 78 & 77 \\ 121 & 137 & 152 & 57 \\ 255 & 74 & 254 & 93 \end{bmatrix}, EI2: \begin{bmatrix} 92 & 240 & 63 & 218 \\ 177 & 50 & 247 & 176 \\ 32 & 171 & 168 & 227 \\ 189 & 185 & 58 & 153 \end{bmatrix}$$

$$EI3: \begin{bmatrix} 200 & 74 & 68 & 193 \\ 181 & 34 & 233 & 155 \\ 195 & 209 & 13 & 251 \\ 90 & 116 & 156 & 229 \end{bmatrix}, EI4: \begin{bmatrix} 125 & 229 & 8 & 61 \\ 2 & 149 & 188 & 207 \\ 110 & 75 & 162 & 212 \\ 242 & 138 & 81 & 57 \end{bmatrix}$$

After segmentation, pixels of *EI2*, *EI3* and *EI4* are converted into binary values using (1). Then, Most Significant Bits (MSBs) of *EI2*, *EI3* and *EI4* are collected and

---

concatenated into one row vector. The total number of the collected bits, $|CB|=3XY/4$. Algorithm 3 describes Collect bits (Figure 4) of collecting MSBs of segments *EI2*, *EI3* and *EI4*. Example 3 shows Collect bits (Figure 4) example of MSBs collecting.

Algorithm 3. Collect bits (Figure 4) // collect MSBs from *EI2*…*EI4*

Input:

- X, Y: power of two.
- *EI2*, *EI3*, *EI4*: each of the size $X/2 \times Y/2$

Output:

- CB (1:3XY/4): MSBs from *EI2*,…, *EI4*, $CB_i \in \{1,0\}$

Steps:

1. Convert the pixels values in *EI2*, *EI3*, and *EI4* into binary values.

$z = 1$

$for\ i = 1{:}X$

$for\ j = 1{:}Y$

$for\ u = 1{:}X$

$binaryEI2_{z,u} = \lfloor EI2_{j,i}/2^{u-1} \rfloor mod\ 2$

$binaryEI3_{z,u} = \lfloor EI3_{j,i}/2^{u-1} \rfloor mod\ 2$

$binaryEI4_{z,u} = \lfloor EI4_{j,i}/2^{u-1} \rfloor mod\ 2$

$z = z + 1$

end

end

end

2. Get the MSBs in each *EI2*, *EI3*, *EI4*

$for\ i = 1{:}X \times Y$

$MSBinEI2[i] = binaryEI2[i, 8]$

$MSBinEI3[i] = binaryEI3[i, 8]$

$MSBinEI4[i] = binaryEI4[i, 8]$

end

3. Concatenate the MSB bits from *EI2*, *EI3*, *EI4* into row vector $[c_1, c_2, \ldots, c_{|CB|}]$

$CB = MSBinEI2 || MSBinEI3 || MSBinEI4$

Example 3: Example of Collect bits (Figure 4) collecting MSBs from *EI2*, *EI3*, and *EI4*.

Let's consider *EI2*, *EI3*, *and EI4* from Example 3 output.

Input:

– *EI2*, *EI3*, *EI4*, each of the size 4×4 (*X* / 2 × *Y* / 2)

$$EI2: \begin{bmatrix} 92 & 240 & 63 & 218 \\ 177 & 50 & 247 & 176 \\ 32 & 171 & 168 & 227 \\ 189 & 185 & 58 & 153 \end{bmatrix}$$

$$EI3: \begin{bmatrix} 200 & 74 & 68 & 193 \\ 181 & 34 & 233 & 155 \\ 195 & 209 & 13 & 251 \\ 90 & 116 & 156 & 229 \end{bmatrix}, \; EI4: \begin{bmatrix} 125 & 229 & 8 & 61 \\ 2 & 149 & 188 & 207 \\ 110 & 75 & 162 & 212 \\ 242 & 138 & 81 & 57 \end{bmatrix}$$

Output:

– Collected bits *CB* (1:3*XY*/4) = *CB* (1:48)

Steps:

1. *EI2* pixels values are converted into binary values

MSB LSB

$$binaryEI2 = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \end{bmatrix}$$

2. MSBs are collected from binary *EI2*

MSBs of *EI2*: $[0\ 1\ 0\ 1\ 1\ 0\ 1\ 1\ 0\ 1\ 1\ 0\ 1\ 1\ 1\ 1]$

3. *EI3* pixel values are converted into binary values

MSB LSB

$$binaryEI3 = \begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \end{bmatrix}$$

4. MSBs are collected from binary *EI3*

MSBs of *EI3*: $[1\ 1\ 1\ 0\ 0\ 0\ 1\ 0\ 0\ 1\ 0\ 1\ 1\ 1\ 1\ 1]$

5. *EI4* pixel values are converted into binary values

$$
\begin{array}{c}
\text{MSB} \qquad\qquad\qquad\qquad\qquad \text{LSB} \\
binaryEI4 = \begin{bmatrix}
0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\
1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\
1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\
1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\
0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\
1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 \\
1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\
1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 0 & 0 & 1
\end{bmatrix}
\end{array}
$$

6. MSBs are collected from binary *EI4*

MSBs of binary *EI4*: $[0\ 0\ 0\ 1\ 1\ 1\ 0\ 1\ 0\ 1\ 1\ 0\ 0\ 1\ 1\ 0]$

7. MSBs from *EI2*, *EI3*, *EI4* are concatenated into one row vector:

Collected bits *CB*

$[0\ 1\ 0\ 1\ 1\ 0\ 1\ 1\ 0\ 1\ 1\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 0\ 0\ 1\ 0\ 0\ 1\ 0\ 1\ 1\ 1\ 1\ 1\ 0\ 0\ 0\ 1\ 1\ 1\ 0\ 1\ 0\ 1\ 1\ 0\ 0\ 1\ 1\ 0]$

Consider now Select bits (Figure 4). Data hider fixes selection ratio α, where α is the selection ratio of the number of selected bits (*SB*) to the total collected bits (*CB*), where selection process of *SB* is done using selection key K_{SL} according to selection ratio α. Construction of K_{SL} is not clarified in [1], hence construction of K_{SL} is our problem and it is described in Section 3.1.2. Since the number of collected bits $|CB| = 3XY/4$ and if the number of selected bits *SB* is L, where L is $1 \leq L \leq 3XY/4$, then $\alpha = L / (3XY/4)$. We can see that by fixing α, L is computed as follows:

$$L = \alpha \times \left(\frac{3XY}{4}\right) \quad (5)$$

The selected bits are chosen according to K_{SL} which contains the indices of L bits selected from CB.

The Algorithm 4 of Select bits (Figure 4) describes the selecting bits steps and Example 4 shows an example of selecting L from MSBs.

Algorithm 4. Select bits (Figure 4) // selects L bits from collected bits, CB, according to K_{SL}

Input:

- X, Y: power of 2.
- CB (1:3XY/4): collected bits.
- α: Selection Ratio,
- K_{SL} (1: L): Selection Key = [K_{SL1}, K_{SL2}, K_{SLL}], L= α (3XY/4).

Output:

- SB (1: L): the Selected Bits.
- L = α (3XY/4)

Steps:

1. Calculate L using (5).
2. Select L bits from CB :

$for\ i = 1{:}L$

$\quad index = K_{SL}(i);$

$\quad SB(i) = CB(index)$

end

Example 4. Selecting bits (Figure 4) example of L bits selection from collected bits, CB obtained in Example 3.

Input:

– α=1

– Collected bits *CB* are

[0 1 0 1 1 0 1 1 0 1 1 0 1 1 1 1 1 1 1 0 0 0 1 0 0 1 0 1 1 1 1 1 0 0 0 1 1 1 0 1 0 1 1 0 0 1 1 0]

– K_{SL}

[47 27 35 34 11 1 13 21 38 10 42 48 8 29 19 3 46 9 4 36 20 26 6
31 32 30 37 25 33 43 18 24 45 40 44 16 28 41 5 12 7 39 17 23 2 22 14 15]

Output:

- Selected bits *SB* (1: *L*) =*SB* (1:48).

Steps:

1. L=48 using (5).

$for\ i = 1{:}L$

$index = K_{SL}(i);$

$SB(i) = CB(index)$

end

According to indices in K_{SL} bits are selected from *CB*. First bit in *SB* will be the bit that has index equal to K_{SL} (1). For example, when i=1 then, K_{SL} (1) = 47, the selected bit, *SB* (1) will be *CB* (47), which is 1. The second value in K_{SL} is 27, so, the *SB* (2) = *CB* (27), and so on.

Consider now Shuffle bits (Figure 4). After getting, *SB*, it is shuffled using K_{SF} to produce shuffled bits (*SHB*). Shuffling key K_{SF} is not specified in [1]. Hence, construction of K_{SF} is our problem and will be described in Section 3.1.2.1. Steps of shuffling selected bits, *SB*, are described in Algorithm 5 and Example 5.

Algorithm 5. Shuffle bits (Figure 4) // shuffles selected bits, *SB*, according K_{SF}

Input:

- L: number of the bits in *SB* , $L = \alpha \times \left(\frac{3XY}{4}\right)$
- $SB(1: L)$: Selected bits
- K_{SF}: Shuffle key (positive integer such that gcd(K_{SF}, L)=1, where gcd is the greatest common divisor

Output:

- *SHB* (1:L): Shuffled bits

Steps:

1. Create Shuffle row (*SR*) containing indices from 1 to L

   $SR = [1, 2, 3,\ldots,L]$
2. Get *SR1* indices by shuffling *SR* using K_{SF}.

   $$\begin{aligned}&for\ \ i = 1:L\\ &\quad SR1(i) = ((K_{SF} \times SR(i)) \bmod L) + 1\\ &end\end{aligned}$$
3. *SR*=*SR1*. Put each bit in the *SB* to the corresponding index in the *SR*.

   $$\begin{aligned}&for\ \ i = 1:L\\ &\quad index = SR(i);\\ &\quad SHB(i) = SB(index);\\ &end\end{aligned}$$

Example 5: Example of Shuffle bits (Figure 4) that shuffles L selected bits, *SB*, getting shuffled bits, *SHB*.

Let's consider selected bits *SB* from Example 4 output.

Input:

- Number of bits in *SB*, L=48;
- Shuffle key K_{SF} =13, gcd(13,48)=1
- Selected bits, *SB*:

$$[1\ 0\ 0\ 0\ 1\ 0\ 1\ 0\ 1\ 1\ 1\ 0\ 1\ 1\ 1\ 0\ 1\ 0\ 1\ 1\ 0\ 1\ 0\ 1\ 1\ 1\ 1\ 0\ 0\ 1\ 1\ 0\ 0\ 1\ 0\ 1\ 1\ 0\ 1\ 0\ 1\ 0\ 1\ 1\ 1\ 0\ 1\ 1].$$

Output:

Shuffled bits *SHB* (1: *L*) = *SHB* (1:48).

Steps:

1. Create shuffle row, *SR*, vector containing values between 1 and *L* *SR* is

[1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48]

2. Update shuffle row *SR* using

$SR1(i)=(K_{SL}\times SR(i)) \bmod L+1$ for $i=1,\ldots,L$

$SR=SR1$;

SR

[14 27 40 5 18 31 44 9 22 35 48 13 26 39 4 17 30 43 8 21 34 47 12 25
38 3 16 29 42 7 20 33 46 11 24 37 2 15 28 41 6 19 32 45 10 23 36 1]

3. According to *SR*, bits are shuffled to obtain *SHB*. The first bit in *SHB* will be the bit in index 14 in *SB*: *SHB* (1) = *SB* (14) =1. The second bit in *SHB* will be the bit in index 27 in *SB*, *SB* (2) =*SB* (27) =1and so on to produce *SHB* as follows:

[1 1 0 1 0 1 1 1 1 0 1 1 1 1 0 1 1 1 0 0 1 1 0 1 0 0 0 0 0 1 1 0 0 1 1 1 0 1 0 1 0 1 0 1 0 1 1 0 1 1]

Encoding and Compressing Phase Details

In this phase, Shuffled bits, *SHB*(1: *L*) obtained by Most Significant Bits (MSBs) Selection (Figure 4) phase divided into *K* groups, *KGs* , each containing *n* bits. After that, *KGs* are compressed using binary *H* matrix with size $r \times n$ to produce syndrome groups, *SGs*. Figure 5 shows diagram of Phase 2.

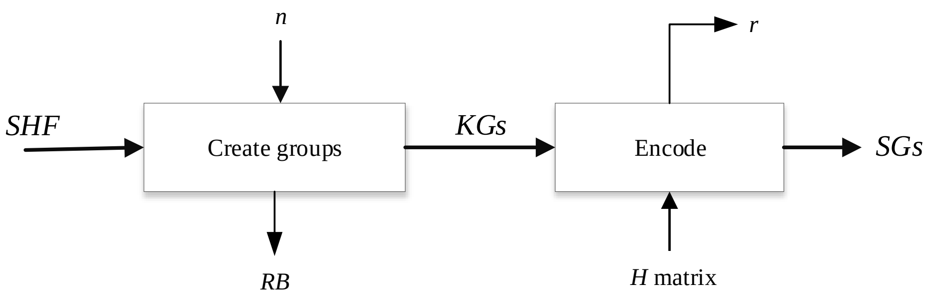


Figure 5: Encoding and Compressing Phase Details

The shuffled bits *SHB* with size *L* are divided into *K* groups, *KGs*, each group contains *n* bits, the number of groups *KGs*, *K*, is calculated as follows:

$$K = \lfloor L/n \rfloor \tag{6}$$

Creation groups (Figure 5) pseudo code is described in Algorithm 6 and Example 6 illustrates the creation process.

Algorithm 6. Create groups (Figure 5) // divide *SHB*(1: *L*) into *KGs*

Input:

- X, Y: power of 2.
- L: number of the bits in *SB*, $L=\alpha\times\left(\frac{3XY}{4}\right)$.
- *SHB* (1: *L*): Shuffled bits,
- $n \in [1{:}L]$, number of bits in each group

Output:

- *KGs*: *K* groups, size $K \times n$, *K* is the number of the groups, *n* is the number of bits in each group
- $RB[K \times n + 1 \ldots L]$: the remaining bits

Steps:

- Calculate the number of the groups using (6).
- Create matrix *KGs* with size $K\times n$.

$$
\begin{aligned}
& x = 1; \\
& for\ i = 1:K \\
& \quad for\ j = 1:n \\
& \quad\quad KGs(i, j) = SHB(x) \\
& \quad\quad x = x + 1 \\
& \quad end \\
& end
\end{aligned}
$$

- Calculate number of remaining bits |*RB*|=*L* mod *n*.

$$
\begin{aligned}
& T = L \bmod n \\
& for\ \ z = 1:T \\
& \quad RB(z) = SHB(x) \\
& \quad x = x + 1 \\
& end
\end{aligned}
$$

Example 6: Example of create *K* groups (Figure 5) from shuffled bits *SHB*

Let's consider *SHB* from Example 5. *n*=8;

Inputs:

- Shuffled bits *SHB*

$$[1\ 1\ 0\ 1\ \ 0\ 1\ 1\ 1\ 1\ 0\ 1\ 1\ 1\ 1\ 0\ 1\ 1\ 1\ 0\ 0\ 1\ 1\ 0\ 1\ \ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 0\ 0\ 1\ 1\ 1\ 0\ 1\ 0\ 1\ 0\ 1\ 0\ 1\ 1\ 0\ 1\ 1]$$

- *n* =8

Outputs:

- *KGs*

- $RB[(K \times n) + 1 \ldots L]$

Steps:

1. Number of shuffled bits *SHB*: L = 48.

2. Using (6): *K*= $\left\lfloor \frac{48}{8} \right\rfloor$ = 6. Number of group is 6

3. Divide the *SHB* into 6 groups, each group contains 8 bits to get *KGs*

$$KGs=\begin{bmatrix}1&1&0&1&0&1&1&1\\1&0&1&1&1&1&0&1\\1&1&0&0&1&1&0&1\\0&0&0&0&0&1&1&0\\0&1&1&1&0&1&0&1\\0&1&0&1&1&0&1&1\end{bmatrix}$$

Remainder bits *RB* = 48 mod 8 =0. No remainder bits.

RB= []. There is no remainder bits.

Consider now *K* groups, *KGs* (Figure 5) phase. After create *KGs*, the bits in each group are encoded using Slepian-Wolf encoding [12]. In [2], using LDPC *H* matrix [10] with size $r \times n$, where $0 < r < n$, compress each group into syndrome or encoded groups, SGs, as follows:

$$\begin{bmatrix} SGs(1,1)\ldots\ldots SGs(1,r) \\ SGs(2,1)\ldots\ldots SGs(2,r) \\ \vdots \\ SGs(k,1)\ldots\ldots SGs(k,r) \end{bmatrix} = \begin{bmatrix} KGs(1,1)\ldots\ldots KGs(1,n) \\ KGs(2,1)\ldots\ldots KGs(2,n) \\ \vdots \\ KGs(k,1)\ldots\ldots KGs(k,n) \end{bmatrix} \bullet H^T \quad ,k=1,2,\ldots,K \qquad (7)$$

Where *k* the group number and *K* is the total number of *KGs* groups. *SGs* (*k*, *r*) is the syndrome or encoded bit. LDPC *H* matrix in [2] is used as in [6] with size *n*=6336, *r*=3840. However, this *H* matrix can't be obtained exactly neither in [2] nor in it reference [10]. Thus, construction of *H* is our problem and it is described in Section 3.4.

Algorithm 7 of encoding and compressing process (Figure 5) describes compressing procedure and Example 7 shows an example of encoding *KGs* into *SGs*.

Algorithm 7. Encoding (Figure 5) // encodes and compresses *KGs* into *SGs* groups

Inputs:

- *KGs*: *K* groups, size $K \times n$.
- *H*: binary matrix, size *r* rows and *n* columns.

Output:

- *SGs*: encoded groups, size *K*: rows (groups), *r*: columns (number of bits in each group).
- $r : 0 < r < n$.

Steps:

1. Convert *H* matrix into transpose $H = H^T$.
2. Logical multiplication: $SGs = KGs.\ H^T$.

Example 7. Encoding and compressing (Figure 5) of *K* groups.

Let's *K* group from Example 6. *H* is constructed using Gallager method described in Section 3.3 with size 4×8.

Inputs:

$$- \ KGs = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \end{bmatrix}$$

$$- \ H = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

Outputs:

- *SGs*, size $K \times r$.

Steps:

1. *H* is transposed.

$$H^T = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

2. Using (7), logical multiplying *KGs* withH^T, *SGs* is obtained

$$\underbrace{\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}_{6\times4}}_{r} = \underbrace{\begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \end{bmatrix}_{6\times8}}_{n} \bullet \begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}_{8\times4}$$

Embedding Secret Data (Figure 3) Phase Details

In this phase, secret data, *SD*, are embedded into syndrome group *SGs* obtained by encoding and compressing phase (Figure 5) to produce embedded groups, *EMBG*, with size $K \times n$. Then, embedded groups with remainder bits *RB* after create groups (Figure 5) are reverse shuffled using K_{SF} to obtain inversed shuffle bits, *ISHB*. After that, MSBs in segments *EI2*, *EI3*, *EI4* which are obtained after decomposing (Figure 4) are replaced with inversed shuffle bits *ISHB* to get *EI`2*, *EI`3*, *EI`4*. Marked encrypted image, *MEI*, are obtained from composing *EI1*, *EI`2*, *EI`3*, *EI`4*. Figure 6 shows diagram of Phase 3.

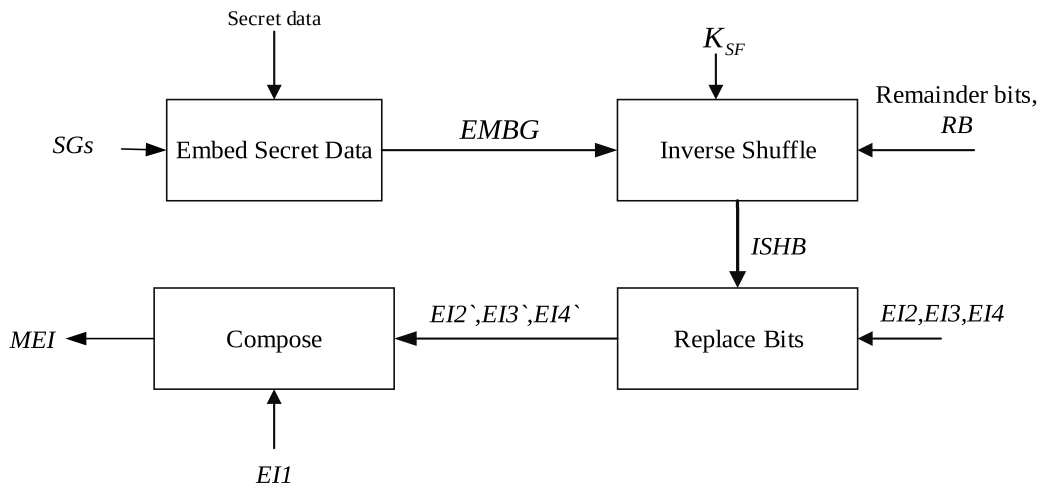


Figure 6: Embedding Secret Data Phase Details

After encoding and compression (Figure 5) phase, the total vacated room for embedding both Secret data *SD* and syndrome groups *SGs* will be divided into *K* groups, each of size *r* bits. Each of these *K* groups will hold *SD* in size $n - r$ and the left *r* bits hold *SGs* as follows:

$$EMBG = SGs \,||\, SD \tag{8}$$

Where *EMBG* is a matrix holds both *SGs* and *SD* of size $K \times n$. Algorithm 8 describes embedding secret data *SD* (Figure 6) and syndrome groups *SGs* into *EMBG*. Example 8 shows embedding secret data *SD* (Figure 6) example of embedding *SD*.

Algorithm 8. Embed Secret Data (Figure 6) into *EMBG*

Input:

- *SGs*: syndrome groups $K \times r$, *K*: number of the groups, *r*: number of groups after encoding.
- *SD*: Secret data, $K \times (n - r)$.
- $r \in [1,\ldots,n\text{-}1]$

Output:

– *EMBG*: Embedded groups, size $K \times n$, *K*: number of the groups, *n*: number of groups after embedding.

Steps:

1. Declare a matrix *EMBG* with size $K \times n$: *EMBG* [*K*, *n*] = {0}.
2. Divide *SD* into groups with size $K \times (n - r)$ to concatenate *SGs* with *SD*

$x = 1$

$for\ i = 1{:}K$

$for\ j = 1{:}n - r$

$SDG(i, j) = SD(x)$

$x = x + 1$

end

end

3. Embed the syndrome *SGs* in *EMBG* in $K \times r$ space

$for\ y = 1{:}K$

$for\ z = 1{:}r$

$EMBG(y, z) = SGs(y, z)$

end

end

4. Embed the secret data with $K \times (n - r)$

$for\ a = 1{:}K$

$for\ b = n - r + 1{:}n$

$EMBG(a, b) = SDG(a, b)$

end

end

Example 8. Example of embedding secret data *SD* (Figure 6) with syndrome groups *SGs* into embedded group *EMBG*.

Let's consider *SD* are generated randomly with size $K \times (n - r) = 6\times (8\text{-}4) = 24$ bits.

Input:

- $SD = \begin{bmatrix}1\,0\,1\,0\;\,0\,1\,0\,1\,1\,0\,0\,0\,1\,0\,0\,0\;\,0\,0\,0\,1\,0\,1\,1\,0\end{bmatrix}$

$$- \; SGs = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

Outputs

- *EMBG* size $K \times n$

Steps

1. *SD* are divided into $K =$ 6 groups, each group contains 4 bits

$$\begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

2. Declare *EMBG* matrix with size 6×8 contains zeros values

3. Assign SD groups to *EMBG* matrix in space i = 1…6 and j = 4…8

$$EMBG = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

4. Assign *SGs* to *EMBG* matrix in space i =1...6 and $j =$ 1...4

$$EMBG = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \end{bmatrix}$$

After embedding secret data (Figure 6), *EMBG* are converted into row vector then concatenated with remainder bits *RB* obtained by create groups Algorithm 6 (Figure 5). The resultant vector will be shuffled in reverse way using K_{SF} (Figure 6) to obtain Inversed Shuffled Bits (*ISHB*). Algorithm 9 describes inverse shuffle bits (Figure 6) of embedded group *EMBG*. Example 9 shows inverse shuffle bits (Figure 6) of embedded group *EMBG*.

Algorithm 9. Inverse shuffled bits (Figure 6) of embedded groups *EMBG*

Input:

- *EMBG*: Embedded groups, size $K \times n$, K: number of the groups, n: number of groups after embedding.
- K_{SF}: Shuffle key (positive integer such that gcd (K_{SF}, L)=1, where gcd is the greatest common divisor
- $RB[(K \times n) + 1 \dots L]$: the remaining bits

Output:

- *ISHB* (1 : L): Inversed Shuffled Bits , $L = \alpha \times \left(\frac{3XY}{4} \right)$

Steps:

1. Reshape the *EMBG* from matrix into row vector *B*

$x = 1$

$for\ i = 1{:}\,K$

$for\ j = 1{:}\,n$

$B(x) = EMBG(i, j)$

$x = x + 1$

end

end

2. Concatenate the row vector of *EMBG* with *RB*

   $C=B||RB$

3. Declare shuffle row vector *SR* contains indices between 1 and L.
4. Get *SR1* indices by shuffling *SR* using K_{SF}.

$for\ j = 1{:}\,L$

$$SR1(i) = \left(\left(K_{SF}\ \times SR(i)\right) mod\ L\right) + 1$$

end

5. Shuffle the selected bits depend on the row vector *SR1*. Each bit in the selected bit vector to the corresponding index in the shuffle vector.

$for\ i = 1{:}\,L$

$index = SR1(i)$

$ISHB(i) = C(index)$

end

Example 9. Example of Inverse Shuffle bits *ISHB* (Figure 6) in embedding groups *EMBG*.

Let's consider embedded groups *EMBG* from Example 8 and remainder bits from Example 6.

Input:

$$- EMBG = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \end{bmatrix}$$

- K_{SF} =13

- *RB* = []. There is no remainder bits which is obtained from Example 6.

Outputs:

- *ISHB* (1: *L*) = *ISHB* (1:*48*)

Steps:

1. Reshape *EMBG* into row vector

$$[1\ 1\ 1\ 1\ 1\ 0\ 1\ 0\ 1\ 1\ 1\ 1\ 0\ 1\ 0\ 1\ 0\ 1\ 1\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 1\ 0\ 1\ 0\ 0\ 0\ 0\ 1\ 0\ 1\ 0\ 1\ 0\ 1\ 1\ 0]$$

2. Concatenate *EMBG* of row vector with *RB* = *EMBG* || *RB*

3. Declare shuffle row *SR* containing values from 1 …. *L*, *L*= *48*

SR =

$$[1\ \ 2\ \ 3\ \ 4\ \ 5\ \ 6\ \ 7\ \ 8\ \ 9\ \ 10\ \cdots\cdots\ 37\ \ 38\ \ 39\ \ 40\ \ 41\ \ 42\ \ 43\ \ 44\ \ 45\ \ 46\ \ 47\ \ 48]$$

4. Get *SR*1 using *SR*1= (K_{SF} × *SR*) mod *L*.

[14 27 40 5 18 31 44 9 22 35 48 13 26 39 4 17 30 43 8 21 34 47 12 25
38 3 16 29 42 7 20 33 46 11 24 37 2 15 28 41 6 19 32 45 10 23 36 1]

5. Using *SR*1, bits in *EMBG* in row vector are shuffled in reverse order to produce *ISHB*

$$[0\ 0\ 0\ 0\ 1\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 1\ 1\ 0\ 0\ 1\ 1\ 1\ 0\ 0\ 1\ 1\ 1\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1\ 1\ 0\ 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 1]$$

Consider now inverse shuffle bits (Figure 6). After getting, *ISHB*, the MSB of the segments *EI2*, *EI3*, and *EI4* (Figure 4) are replaced with the *ISHB* to get *EI2`*, *EI3`*, and *EI4*`. By converting the *EI2*, *EI3*, and *EI4* into binary values using (1), the MSBs are collected and selected using K_{SL} using same steps in Algorithm 3 and 4. Then, the selected bits *SB* are replaced with *ISHB*. Steps of replacing MSBs (Figure 6) are described in Algorithm 10 and Example 10.

Algorithm 10. Replace MSB bits (Figure 6) in *EI2*,…, *EI4* with *ISHB*

Inputs:

- X,Y: power of 2;
- *ISHB*: Inversed shuffle bits vector [1….*L*], $L = \alpha\,(3XY/4)$
- *EI2*, *EI3*, *EI4*, each of the size $X/2 \times Y/2$
- K_{SL}: Selection Key = $[K_{SL1}, K_{SL2}, \ldots K_{SLL}]$, $K_{SLi} \in [1,\ldots,L]$, $L = \alpha\,(3XY/4)$

Output:

- Segments, *EI2*`, *EI3*`, *EI4*`, each of the size $X/2 \times Y/2$

Steps:

- Collect MSBs from *EI2*, *EI3*, *EI4* using Algorithm 3.
- K_{SL} defines indices of collected bits that will be replaced by *ISHB*
- Then, the replaced modified collected bits are return into segments *EI2*, *EI3*, *EI4*
- Convert the binary values into pixels values using (3) to obtain *EI2*`, *EI3*`, *EI4*`.

Example 10. Replace MSBs (Figure 6) in *EI2*, *EI3* and *EI4* to obtain *EI2*`, *EI3*` and *EI4*`.

Let's consider *EI2*,…, *EI4* obtained from Example 2 and inversed shuffle bits obtained from Example 9.

Inputs:

- *ISHB*

[0 0 0 0 1 0 0 10 0 0 0 1 1 0 0 1 1 1 0 0 1 1 1 0 0 1 0 0 0 0 0 0 1 1 1 0 0 1 1 1 1 1 1 1 1 0 1]

$$EI2=\begin{bmatrix} 92 & 240 & 63 & 218 \\ 177 & 50 & 247 & 176 \\ 32 & 171 & 168 & 227 \\ 189 & 185 & 58 & 153 \end{bmatrix}, \mathrm{EI3}=\begin{bmatrix} 200 & 74 & 68 & 193 \\ 181 & 34 & 233 & 155 \\ 195 & 209 & 13 & 251 \\ 90 & 116 & 156 & 229 \end{bmatrix}$$

$$EI4=\begin{bmatrix} 125 & 229 & 8 & 61 \\ 2 & 149 & 188 & 207 \\ 110 & 75 & 162 & 212 \\ 242 & 138 & 81 & 57 \end{bmatrix}$$

-K_{SL}

[47 27 35 34 11 1 13 21 38 10 42 48 8 29 19 3 46 9 4 36 20 26 6
31 32 30 37 25 33 43 18 24 45 40 44 16 28 41 5 12 7 39 17 23 2 22 14 15]

Outputs:

- Segments *EI2`*, *EI3`*, *EI4`* each of the size 4×4 (X/2 × Y/2)

Steps:

Collected bits *CB*

1. $[0\,1\,0\,1\,1\,0\,1\,1\,0\,1\,1\,0\,1\,1\,1\,1\,1\,1\,1\,0\,0\,0\,1\,0\,0\,1\,0\,1\,1\,1\,1\,1\,0\,0\,0\,1\,1\,1\,0\,1\,0\,1\,1\,0\,0\,1\,1\,0]$

2. Using K_{SL}, the collected bits are replaced with *ISHB*

K_{SL}

[47 27 35 34 11 1 13 21 38 10 42 48 8 29 19 3 46 9 4 36 20 26 6
31 32 30 37 25 33 43 18 24 45 40 44 16 28 41 5 12 7 39 17 23 2 22 14 15]

ISHB

[0 0 0 0 1 0 0 10 0 0 0 1 1 0 0 1 1 1 0 0 1 1 1 0 0 1 0 0 0 0 0 0 1 1 1 0 0 1 1 1 1 1 1 1 1 0 1]

To obtain modified collected bits, *CB’*

[0 1 0 1 1 1 1 1 1 0 1 1 0 0 1 1 1 0 0 0 1 1 1 0 0 1 0 0 1 0 1 0 0 0 0 0 1 0 1 1 0 0 0 1 0 1 0 0]

1. MSBs in *EI2* is replaced with *CB’* from (1 : 16)

$$\begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \end{bmatrix}$$

2. MSBs in *EI3* is replaced with *CB*’ from (16 +1: 16×2)

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \end{bmatrix}$$

3. MSBs in *EI4* is replaced with *CB’* from (16×2+1 : 16×3)

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{bmatrix}$$

4. Convert binary *EI2*, *EI3*, *EI4* after replacing bits to produce *EI2`*, *EI3`*, *EI4`* using (3).

$$EI2` = \begin{bmatrix} 92 & 240 & 191 & 90 \\ 177 & 178 & 119 & 48 \\ 32 & 171 & 168 & 227 \\ 189 & 185 & 186 & 153 \end{bmatrix}$$

$$EI3` = \begin{bmatrix} 200 & 202 & 68 & 193 \\ 53 & 162 & 233 & 27 \\ 67 & 209 & 13 & 251 \\ 90 & 116 & 28 & 101 \end{bmatrix}, EI4` = \begin{bmatrix} 125 & 229 & 8 & 61 \\ 2 & 21 & 60 & 207 \\ 110 & 203 & 34 & 84 \\ 114 & 138 & 209 & 57 \end{bmatrix}$$

After MSBs in *EI2… EI4* are replaced with *ISHB* to obtain *EI2`… EI4`* (Figure 6) are converted into pixels values. Marked encrypted image (*MEI*) is constructed from *EI1*, *EI2`*, *EI3`* and *EI4`* which are composed (Figure 6) to construct *MEI* using (4). Algorithm 11 describes the composition (Figure 6) of *EI1*, *EI2`*, *EI3`* and *EI4`* to get

MEI. Example 11 shows compose (Figure 6) example of *EI1*, *EI2`*, *EI3`* and *EI4`* to obtain *MEI*.

Algorithm 11. Compose *EI1*, *EI2`*, *EI3`* and *EI4`* (Figure 6) to get marked encrypted image *MEI*

Input:

- *X*, *Y*: power of 2.
- Segments *EI1*, *EI2`*, *EI3`*, *EI4`*, size *X* /2, *Y/2*.

Output:

- *MEI*: Marked encrypted image *MEI* , size *X* , *Y*

Steps:

- Combine the *EI1*, *EI2`*, *EI3`*, *EI4`* using (4) to construct *MEI*.

```
// Using MATLAB style pseudocode:
MEI(1:2:X,1:2:Y)=EI1(1:X/2,1:Y/2);//odd rows and columns
MEI(1:2:X,2:2:Y)=EI'2(1:X/2,1:Y/2)=//odd rows and even columns
MEI(2:2:X,1:2:Y)= EI'3(1:X/2,1:Y/2);//even rows and odd columns
MEI(2:2:X,2:2:Y)= EI'4(1:X/2,1:Y/2);//even rows and columns
Where a:b:c means x values such that: a<=x<=c, x=c+i×b.
```

Example 11. Compose (Figure 6) *EI1*, *EI2`*, *EI3`*, *EI4`* to obtain Marked Encrypted Image *MEI*.

Let's consider *EI1* obtained from Example 2 and *EI2`*… *EI4`* obtained from Example 10.

Inputs:

$$EI1:\begin{bmatrix}219 & 100 & 3 & 160\\ 67 & 142 & 78 & 77\\ 121 & 137 & 152 & 57\\ 255 & 74 & 254 & 93\end{bmatrix}, EI2`:\begin{bmatrix}92 & 240 & 191 & 90\\ 177 & 178 & 119 & 48\\ 32 & 171 & 168 & 227\\ 189 & 185 & 186 & 153\end{bmatrix}$$

$$EI3`:\begin{bmatrix}200 & 202 & 68 & 193\\ 53 & 162 & 233 & 27\\ 67 & 209 & 13 & 251\\ 90 & 116 & 28 & 101\end{bmatrix}, EI4`:\begin{bmatrix}125 & 229 & 8 & 61\\ 2 & 21 & 60 & 207\\ 110 & 203 & 34 & 84\\ 114 & 138 & 209 & 57\end{bmatrix}$$

Outputs:

- Marked encrypted image *MEI* with size 8×8

Steps:

1. Using Step 1 in Algorithm 11, *MEI* is constructed.

$$MEI:\begin{bmatrix}219 & 92 & 100 & 240 & 3 & 191 & 160 & 90\\ 200 & 125 & 202 & 229 & 68 & 8 & 193 & 61\\ 67 & 177 & 142 & 178 & 78 & 119 & 77 & 48\\ 53 & 2 & 162 & 21 & 233 & 60 & 27 & 207\\ 121 & 32 & 137 & 171 & 152 & 168 & 57 & 227\\ 67 & 110 & 209 & 203 & 13 & 34 & 251 & 84\\ 255 & 189 & 74 & 185 & 254 & 186 & 93 & 153\\ 90 & 114 & 116 & 138 & 28 & 209 & 101 & 57\end{bmatrix}$$

The data hider constructs K_{SL}, K_{SF} and K_{ENC} to be used in receiver side. Also, the parameters L, n and r are used to extract the secret data. These data are transmitted through trusted channel.

In [1], the embedding capacity formula are not clearly specified. Thus, we analysis to find formula for embedding capacity. By definition, embedding capacity is the number of bits to be embedded in each pixel in the encrypted image,

$$E_{emb} = \frac{number\ of\ embedded\ bits}{image\ size} = \frac{number\ of\ selected\ bits \times embedding\ ratio}{image\ size} \quad (9)$$

where number of selected bits is L, and embedding ratio is $(1 - r/n)$. Thus, embedding capacity E_{emb} is

$$E_{emb} = \frac{L \times \left(1 - \frac{r}{n}\right)}{XY} \tag{10}$$

From (5) we can write (10) as follows:

$$E_{emb} = \frac{\alpha \times \left(\frac{3XY}{4}\right) \times \left(1 - \frac{r}{n}\right)}{XY} = \frac{3\alpha(n-r)}{4n} \tag{11}$$

2.1.3 Stage 3: Data Extraction and Image Recovery Details

Receiver may use three options depending on his authority and privileges:

- Option1: data extraction,
- Option 2:approximate image construction,
- Option 3: lossless recovery.

Option 1 is used when the receiver has only K_{SL}, K_{SF}, L, n and r. The secret data, *SD*, is extracted perfectly without distortion. However, the image can't be constructed. Option 2 is used when the receiver has only the encryption key K_{ENC}, an approximate image is constructed with high quality. Option 3 is used when the receiver has K_{SL}, K_{SF}, L, n, r and K_{ENC}. In that case, the secret data is extracted perfectly, and recovered image is constructed perfectly in some conditions. Figure 7 shows the diagram of Stage 3: Data Extraction and Image Recovery (Figure 3) details. Each option will be explained in details further.

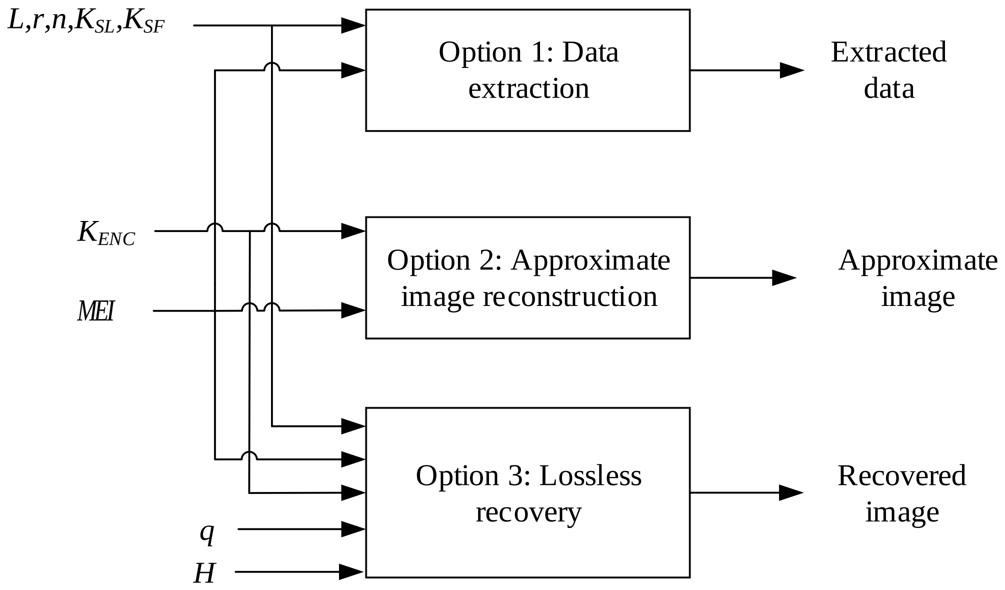


Figure 7: Stage 3: Data Extraction and Image Recovery Stage Details

Option 1: Data Extraction Details

In this option, the receiver has only K_{SL}, K_{SF}, L, n and r. In this option, the secret data will be extracted perfectly. Marked encrypted image *MEI* is decomposed into 4 segments *V1*, *V2*, *V3*, *V4* defined by (4). Then, MSBs are collected from *V2*, *V3*, *V4*, forming collected bits, *CB*`. After that, *L* bits *SB*` are selected from collected bits *CB*` using $K_{SL}$. Then, the selected bits *SB*` are shuffled using K_{SF}. Shuffled bits *SHB*` are divided into *K* groups, each with size *r* forming *KGs* groups. After that, created groups *KGs*, the secret data will be *n* - *r* bits in each group. Figure 8 shows diagram of Option 1.

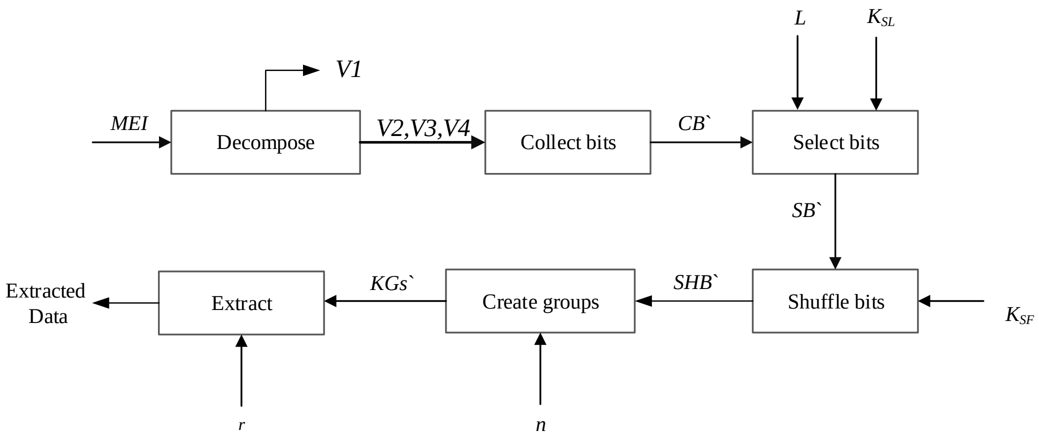


Figure 8: Data Extraction Details

Receiver divides the marked encrypted image, *MEI*, (Figure 8) into four segments, *V1*, *V2*, *V3* and *V4*, using (4) and steps in Algorithm 2 of image decomposition. Then, the Most Significant Bits (MSBs) are collected from *V2*, *V3* and *V4* segments using steps Collect MSBs (Figure 8) Algorithm 3 to obtain *CB*`. After that, *L* bits, *SB*`, are selected from *CB*` using $K_{SL}$. Selecting bits, *SB*`, (Figure 8) in Algorithm 4 is used except the first step.

After *SB*` are selected, they are shuffled using the $K_{SF}$ to obtain shuffled bits *SHB*` (Figure 8) using same steps as in Algorithm 5. Then, the shuffled bits *SHB*` are divided into *K* groups (*KGs*`) using steps of create groups Algorithm 6. Each of these groups contains *r* bits. Up to here, the same steps are used as in the data hiding phase.

As we mentioned in embedding secret data phase in Section 2.1.2, the groups consist of syndrome groups *SGs* with size ($K \times r$) and secret data with size $K \times (n - r)$. Thus, each group contains ($n - r$) secret data. The secret bits can be extracted from the last ($n - r$) bits in each group, that is, [ED (k, r+1),… ,ED (k, n)] are the extracted bits. Data extraction (Figure 8) is described in Algorithm 12 and Example 12.

Algorithm 12. Data extraction (Figure 8)

Input:

- *KGs*: *K* groups, size $K \times n$, *K*: number of groups, r : number of bits in each group
- $r \in [1,\dots,n]$ where $1 < r < n$

Output:

- $ED(1: K \times (n - r))$: Extracted data

Steps:

Get the [ED (k, 1), …, ED (k, n-r)] in each group [KGs (k, r+1),…, KGs (k, n)].

1. Store the *ED* as row vector.

$y = 1$

$for\ i = 1: K$

$for\ j = n - r: n$

$ED(y) = KGs(i, j)$

$y = y + 1$

end

end

Example 12. Data extraction (Figure 8) from marked encrypted image *MEI*

Let's consider the *MEI* from Example 11. The same procedure will apply to have *KGs* groups after decomposing, collecting, selecting, shuffling and division procedure. *n* and *r* are received with *MEI*.

Input:

$$-KGs = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \end{bmatrix}$$

- n=8

- r =4

Output:

- $ED(1: K\times(n - r)) = ED(1: 24)$

Steps:

1. $x = K\times (n - r) = 6\times (8 - 4) = 24$ bits

$$\overbrace{\begin{matrix} & & & \end{matrix}}^{\text{Syndrom groups}} \overbrace{\begin{matrix} & & & \end{matrix}}^{\text{Secret data}}$$

$$\left[\begin{array}{cccc|cccc} 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \end{array}\right]$$

The right side of *KGs* are the extracted data.

2. Extract the secret data *SD* groups and put them in a row vector as follows:

$$[\underbrace{1\,0\,1\,0}_{K=1}\;\underbrace{0\,1\,0\,1}_{K=2}\;\underbrace{1\,0\,0\,0}_{K=3}\;\underbrace{1\,0\,0\,0}_{K=4}\;\underbrace{0\,0\,0\,1}_{K=5}\;\underbrace{0\,1\,1\,0}_{K=6}]$$

If we compare the extracted data with secret data *SD* in Example 8. We will see that they are same. Thus, the secret data is extracted perfectly.

In this option, receiver granted authority and privilege doesn't allow him to know the encryption key, the image can't be decrypted.

Option 2: Approximate Image Construction Details

In this option, receiver granted authority and privilege allow him to possesses K_{ENC} only without K_{SL}, K_{SF}, L, r and n. Hence, receiver can construct approximate image without extracting embedded secret data causing approximate image to be with quality satisfactory to the human eye.

Marked encrypted image is decrypted using encryption key K_{ENC}. Then, decrypted image, *AI* is decomposed into 4 segments *AI1, AI2, AI3, AI4* defined by (4). The procedure of decomposing *AI* (Figure 8) is followed by constructing reference image *BI* with size $X \times Y$ using bilinear interpolation (Figure 8) depending on the segment *AI1* to construct reference image *BI*. The reference image *BI* is decomposed into 4 segments *BI1*, *BI2*, *BI3*, *BI4* defined by (4). Segments *AI2*, *AI3*, *AI4* and *BI2*, *BI3*, *BI4* are used in estimation the MSBs to form *AI2`, AI3`, AI4`*. Then, approximate image *AI* is constructed by compose *AI1, AI2`, AI3`, AI4`*. Figure 8 shows diagram of Option 2.

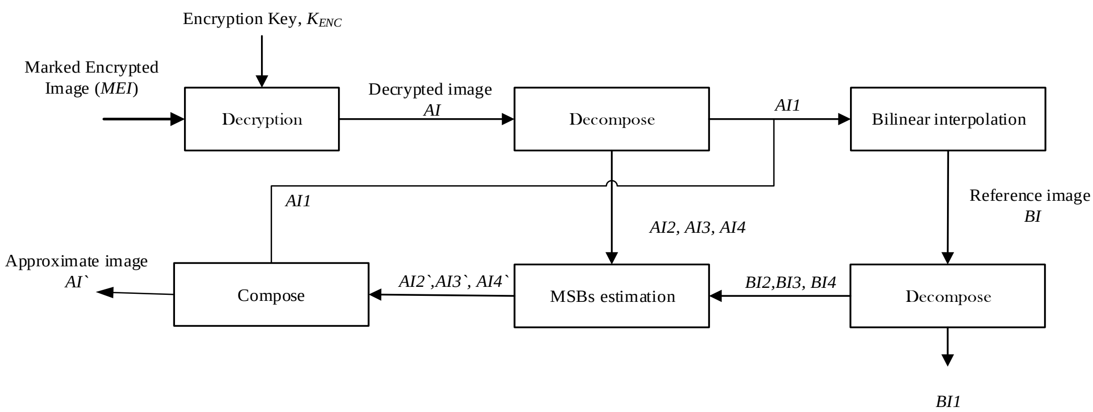


Figure 9: Approximate Image Construction Details

At the beginning, *MEI* is processed using (1), pixel values of *MEI* are converted into binary values. Then, these bits are decrypted using K_{ENC}, which contains embedded data as follows:

$$b`_{i,j,k} = v`_{i,j,u} \oplus K_{ENC_{i,j,u}} \tag{12}$$

$K_{ENC_{i,j,u}}$ is the iju-th bit of the encryption key, K_{ENC}, $e_{i,j,u}$ is the iju-th encrypted bit, and $\oplus$ denotes exclusive-or (XOR) operation, u = 0,1,2,…,7. Then, the decrypted binary values are converted into pixel values to construct decrypted image, *AI* (Figure 9) using (3). The decrypted image is the same as the original one, but with modified MSBs due to embedding secret data (Section 2.1.2). Since the MSBs are modified, the decrypted image *AI* will not be identified to human eye. Then, *AI* is decomposed (Figure 9) into 4 segments *AI1*, *AI2, AI3, AI4* (Figure 8) according to (4).Example 13 illustrates the image decryption of *MEI* and decomposing process (4).

Example 13. Image decryption (Figure 9) from marked encrypted image *MEI* and decompose (Figure 9) into segments *AI1,…, AI4*

Let's consider *MEI* from Example 11.

$$MEI: \begin{bmatrix} 219 & 92 & 100 & 240 & 3 & 191 & 160 & 90 \\ 200 & 125 & 202 & 229 & 68 & 8 & 193 & 61 \\ 67 & 177 & 142 & 178 & 78 & 119 & 77 & 48 \\ 53 & 2 & 162 & 21 & 233 & 60 & 27 & 207 \\ 121 & 32 & 137 & 171 & 152 & 168 & 57 & 227 \\ 67 & 110 & 209 & 203 & 13 & 34 & 251 & 84 \\ 255 & 189 & 74 & 185 & 254 & 186 & 93 & 153 \\ 90 & 114 & 116 & 138 & 28 & 209 & 101 & 57 \end{bmatrix}$$

1. *MEI* is decrypted by encryption key in Example 1 using (2). Decrypted image *AI* as follows

$$AI = \begin{bmatrix} 15 & 215 & 5 & 183 & 3 & 140 & 100 & 138 \\ 125 & 7 & 62 & 20 & 10 & 4 & 93 & 102 \\ 62 & 2 & 31 & 213 & 242 & 249 & 10 & 129 \\ 249 & 17 & 136 & 131 & 102 & 146 & 180 & 130 \\ 78 & 108 & 5 & 150 & 27 & 70 & 140 & 175 \\ 73 & 200 & 27 & 133 & 95 & 169 & 89 & 82 \\ 108 & 3 & 28 & 172 & 55 & 228 & 48 & 72 \\ 88 & 2 & 64 & 81 & 245 & 210 & 222 & 1 \end{bmatrix}$$

2. *AI* is decomposed into $AI1, AI2, AI3, AI4$ using (4)

$$AI1= \begin{bmatrix} 15 & 5 & 3 & 100 \\ 62 & 31 & 242 & 10 \\ 78 & 5 & 27 & 140 \\ 108 & 28 & 55 & 48 \end{bmatrix}, AI2= \begin{bmatrix} 215 & 183 & 140 & 138 \\ 2 & 213 & 249 & 129 \\ 108 & 150 & 70 & 175 \\ 3 & 172 & 228 & 72 \end{bmatrix}$$

$$AI3=\begin{bmatrix}125 & 62 & 10 & 93\\ 249 & 136 & 102 & 180\\ 73 & 27 & 95 & 89\\ 88 & 64 & 245 & 222\end{bmatrix},\ AI4=\begin{bmatrix}7 & 20 & 4 & 102\\ 17 & 131 & 146 & 130\\ 200 & 133 & 169 & 82\\ 2 & 81 & 210 & 1\end{bmatrix}$$

After segmentation, *AI1* is used to construct reference image *BI* using bilinear interpolation algorithm (Figure 9). To construct reference image *BI*, *AI1* is used to be interpolated. Bilinear interpolation (Figure 9) is construct new points from known points. We know that the size of reference image is *X*×*Y* and size of one segment after decomposing is *X/2* × *Y/2*.

Example 14 shows constructing reference image using bilinear interpolation (Figure 9).

Example 14. Construct reference image using bilinear interpolation (Figure 9).

Lets' consider *AI1* from example 13. Size of segment *AI1* is 4×4, size of reference image is 8×8.

$$\begin{array}{c} \\ AI1=\begin{array}{c}1\\2\\3\\4\end{array}\end{array}\begin{array}{c}\begin{array}{cccc}1 & 2 & 3 & 4\end{array}\\ \begin{bmatrix}15 & 5 & 3 & 100\\ 62 & 31 & 242 & 10\\ 78 & 5 & 27 & 140\\ 108 & 28 & 55 & 48\end{bmatrix}\end{array}$$

1. Initially, the matrix *AI1* is expanded into 8×8 matrix as follows

$$\begin{bmatrix} 15 & p1 & 5 & p2 & 3 & p3 & 100 & p4 \\ p5 & p6 & p7 & p8 & p9 & p10 & p11 & p12 \\ 62 & p13 & 31 & p14 & 242 & p15 & 10 & p16 \\ p17 & p18 & p19 & p20 & p21 & p22 & p23 & p24 \\ 78 & p25 & 5 & p26 & 27 & p27 & 140 & p28 \\ p29 & p30 & p31 & p32 & p33 & p34 & p35 & p36 \\ 108 & p37 & 28 & p38 & 55 & p39 & 48 & p40 \\ p41 & p42 & p43 & p44 & p45 & p46 & p47 & p48 \end{bmatrix} \quad (13)$$

We want to find the unknown values in (13) as follows:

First, we traverse rows and we calculate unknown values as average of the two known neighboring in the row values.

Second, we traverse columns and we calculate unknown values as average of the two known neighboring in the column values. Figure 10 shows the grid representation of 4 known values from the left top corner of matrix shown in (13) specified by dash one in solid boxes, and where values (p1, p6, p13) are calculated and displayed in dashed boxes.

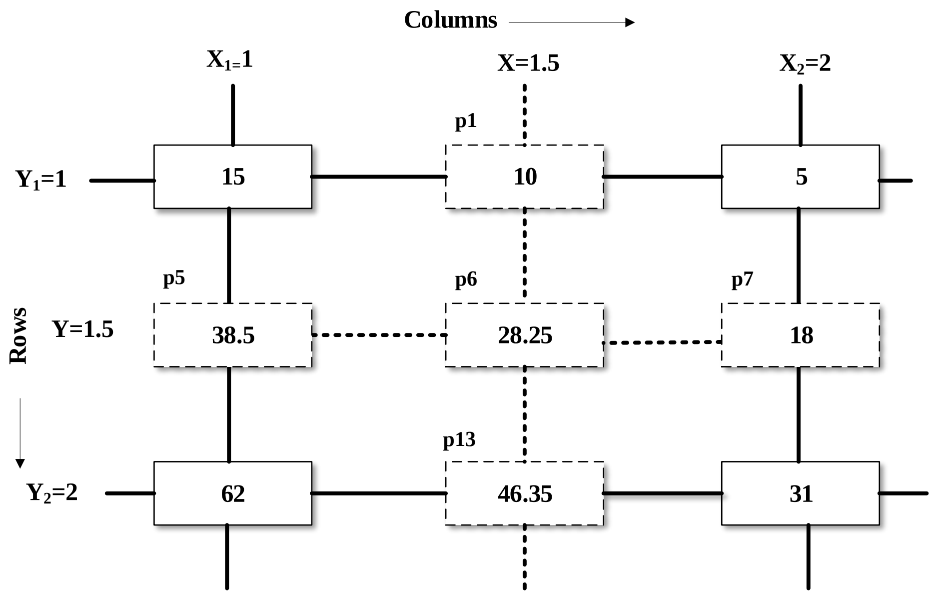


Figure 10: Example of Bilinear Interpolation of Grayscale Values. Dashed Boxes Refer to Unknown Values are Interpolated.

For calculating p1, we use neighboring values 15 and 5 by taking the average as follows:

$$p1 = \frac{15+5}{2}$$

For calculating p13, we use neighboring values 15 and 5 by taking the average as follows:

$$p13 = \frac{62+31}{2} = 46.5$$

For p5, we use the neighboring values 15 and 62 as follows:

$$p6 = \frac{15+62}{2} = 38.5$$

For p7, we use neighboring values 5 and 31 as follows:

$$p6 = \frac{31+5}{2} = 18$$

Then, the middle point p6 are calculated using p1 and p13 as follows:

$$p6 = \frac{10+46.5}{2} = 28.25$$

Other points will be calculated in a same way. The right border and bottom border cannot be calculated using bilinear interpolation, since there is only one neighboring known value beside each value in the border. In order to calculate the unknown values in right and bottom border, we use extrapolation. Figure 11 shows example of extrapolation points.

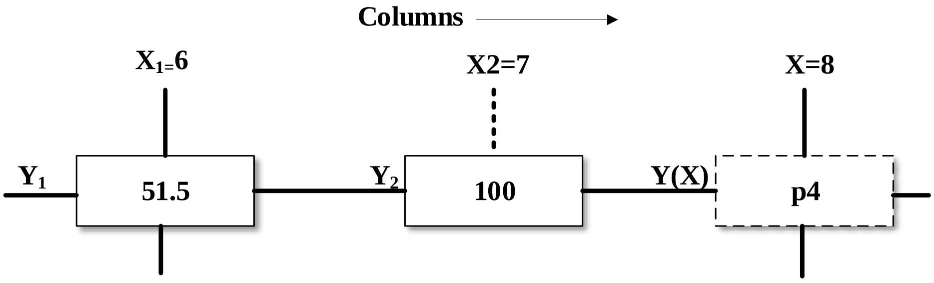


Figure 11: Example of Extrapolation in Grayscale Values. Dashed Boxes Refer to Unknown Values are Extrapolated

We calculate all unknown values in right and bottom borders using MATLAB function in Appendix A.3.4.1, as follows:

B(512,1:511)=interp1(1:512,B(1:511,1:511),512,'linear','extrap');

B(1:512,512)=interp1(1:512,B(1:512,1:512),512,'linear','extrap');

We obtained unknown values as shown below:

$$BI=\begin{bmatrix} 15 & 10 & 5 & 4 & 3 & 52 & 100 & 123 \\ 39 & 28 & 18 & 70 & 123 & 89 & 55 & 81 \\ 62 & 47 & 31 & 137 & 242 & 126 & 10 & 40 \\ 70 & 44 & 18 & 76 & 135 & 71 & 7 & 54 \\ 78 & 42 & 5 & 16 & 27 & 16 & 4 & 69 \\ 93 & 55 & 17 & 29 & 41 & 34 & 26 & 70 \\ 108 & 68 & 28 & 42 & 55 & 52 & 48 & 70 \\ 123 & 81 & 40 & 54 & 69 & 70 & 70 & 58 \end{bmatrix}$$

After constructing image *BI*, it is divided into *BI1*, *BI2*, *BI3*, *BI4* segments using (4). Consider now MSBs estimation (Figure 9) from *AI2*,*AI3*,*AI4* and *BI2*, *BI3*, *BI4* Since, *AI1* was not modified in the embedding stage (Section 2.1.2), the pixels values in *AI1* are the same as in the original, *EI1*, while, the MSBs in *AI2*,*AI3*,*AI4* are modified in embedding stage. Since *BI1* and *AI1* are same, no don't need to estimate their MSBs. The MSBs are estimated to get *AI2*`, *AI3*`, *AI4*` using Algorithm13.

Since, LSBs in each segment are not modified while inserting secret data, the interpolated values of *BI* are used to estimate MSBs in *AI*. For each bit in segment *AI2*, *AI3*, *AI4*, if $|128 + AI2(i,j) mod\ 128 - BI2|$ is greater than the interpolated value mod(*AI*(i, j),128), the MSB of the pixel in (i, j) is 1, otherwise is 0. Algorithm 13 describes the mechanism of MSBs estimation. Example 13 shows an example of estimating MSBs.

Algorithm 13. MSBs estimation.

Input:

- *AI2*, *AI3*, *AI4*: segments with size $X / 2 \times Y / 2$, after decomposing *AI*.
- *BI2*, *BI3*, *BI4*: segments with size $X / 2 \times Y / 2$, after decomposing *BI*.

Output:

- *AI2`*, *AI3`*, *AI4`* : segment with size $X / 2 \times Y / 2$.

Steps:

1. Calculating the MSBs in *AI2`* using *AI2* , *BI2* as follows

$for\ i = 1{:}X/2$

$for\ j = 1{:}Y/2$

$if\ |128 + AI2(i,j) mod\ 128 - BI2| < |AI2(i,j) mod\ 128 - BI2(i,j)|\ then$

$AI2`(i,j) = 128 + AI2(i,j)\ mod\ 128$

$else$

$AI2`(i,j) = AI2(i,j)\ mod\ 128$

end

end

2. Calculating the MSBs in *AI3`* using *AI3*, *BI3*. Using step 1
3. Calculating the MSBs in *AI4`* using *AI4*, *BI4*. using step 1

Example 13. MSBs estimation (Figure 9) from *AI2*, …, *AI4* and *BI2*,…, *BI4*

Let's consider *AI2* and *BI2* after decomposing

$$AI2=\begin{bmatrix} 215 & 183 & 140 & 138 \\ 2 & 213 & 249 & 129 \\ 108 & 150 & 70 & 175 \\ 3 & 172 & 228 & 72 \end{bmatrix}, \text{ and } BI2=\begin{bmatrix} 10 & 4 & 52 & 100 \\ 47 & 137 & 126 & 55 \\ 42 & 16 & 84 & 10 \\ 68 & 42 & 52 & 75 \end{bmatrix}$$

1. *AI2* (1, 1) =215, *BI2* (1, 1) =10, using (13) and step 1 in Algorithm 12

   Is (|128+ (215 mod 128) -10 |) < (| 215 mod 128 – 10 |)?

   (205 < 77) it's false

Then, the estimated MSB is

AI2` = 215 mod 128= 87. This will be the approximate pixel value. All other pixels calculating in same procedure.

After calculating all pixels, the resultant *AI`2* as follows

$$AI2`=\begin{bmatrix} 87 & 55 & 12 & 138 \\ 2 & 85 & 121 & 1 \\ 108 & 22 & 70 & 47 \\ 131 & 44 & 100 & 72 \end{bmatrix}, AI3`=\begin{bmatrix} 125 & 62 & 138 & 93 \\ 121 & 8 & 102 & 52 \\ 73 & 27 & 95 & 89 \\ 88 & 64 & 117 & 94 \end{bmatrix}$$

$$AI4`=\begin{bmatrix} 7 & 20 & 132 & 102 \\ 17 & 131 & 146 & 2 \\ 72 & 5 & 41 & 82 \\ 130 & 81 & 82 & 129 \end{bmatrix}$$

The final step, the receiver can construct the approximate image by composing (Figure 9) the estimated 4 segments *AI1`*,*AI2`*,*AI3`*,*AI4`* using same steps in Algorithm 11.

Option 3: Lossless Recovery Details

In the third option, receiver possesses keys: K_{ENC}, K_{SL}, K_{SF}, *L*, *r* and *n* to extract secret data *SD* without any distortion and the image is recovered lossless.

At the beginning, secret data *SD* is extracted from *MEI*. *L* bits are selected from MSBs, and divided into $K \times n$ groups. Thus, *SD* resultant as $K \times (n - r)$. The remaining groups in $K \times n$ are the syndrome groups *SGs* with size $K \times r$. The syndrome groups *SGs* are extracted to be used in decoding process. After that, an approximate image *AI* is constructed using steps in Section 2.1.3. The approximate image *AI* is encrypted using encryption key K_{ENC}. Then, decrypted approximate image is decomposed into 4 segments *AI1*, *AI2*, *AI3*, *AI4* defined by (4). After that, MSBs are collected from *AI2*, *AI3*, *AI4*. *L* bits are selected from collected bits using selection key K_{SL} and shuffled using shuffle key K_{SF}. Then, the shuffled bits are divided into *K* groups, each with size *n* to form *UG* groups. These groups *UG* are using with *H* matrix and syndrome groups *SGs* in sum-product decoding (Figure 11). The *UG* groups are decoded to get *R* groups. Then, bits in *UG* groups are replaced (Figure 11) with decoded group *R* groups to obtain *R*\` groups. The replaced groups *R*\` groups are reversed shuffle (Figure 11) using K_{SF}. After that, MSBs in *AI2*, *AI3*, *AI4* which obtained from decomposing approximate image are replaced (Figure 11) with inversed shuffle bits to get *AI2*\`, *AI3*\`, *AI4*\`. Segments *AI1*, *AI2*\`, *AI3*\`, *AI4*\` are composed (Figure 11) into encrypted decoded image. Decoded image is obtained by decrypting (Figure 11) encrypted decoded image using K_{ENC}. Figure 12 shows the diagram of decoding and obtaining decoded image.

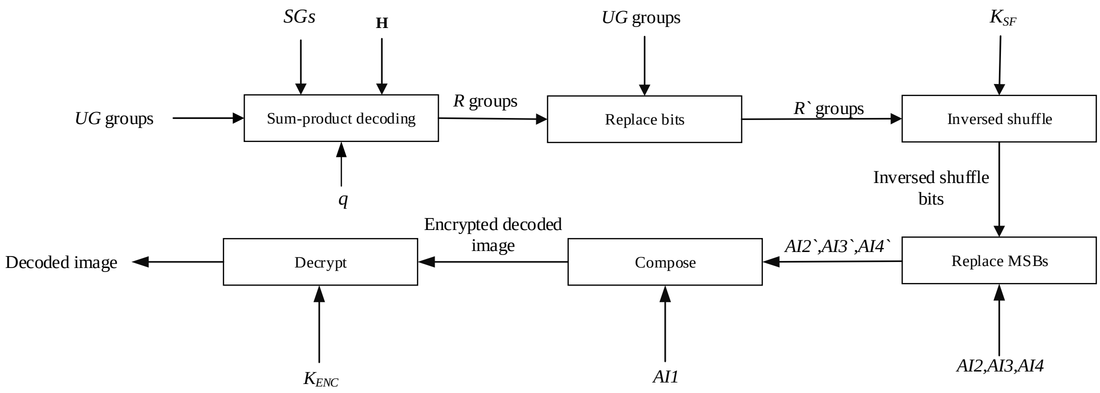


Figure 12: Lossless Recovery Details

Using same steps in Algorithm 12 the secret data is extracted. Marked encrypted image is decomposed into 4 segments *V1*, *V2*, *V3*, and *V4* using Algorithm 2. MSBs are collected from *V2*, *V3*, and *V4* using Algorithm 3. *L* bits are selected from collected bits using selection key K_{SL}, see Algorithm 4 except Step 1. Selected bits are shuffled using shuffle key K_{SF}. The, shuffled bits are divided into *K* groups, each group contains *n* bits. After that, (*n* - *r*) from each group are extracted, as a result, *K* (*n* - *r*) are the extracted data.

Consider now get syndrome (Figure 13). Syndrome groups *SGs* are extracted from *K* groups. The extracted data is in *n* – *r* space in each group. Thus, the total secret data is in *K* (*n* - *r*) space in *K* groups. Syndrome extraction (Figure 13) is described in Algorithm 14. Example 14 shows an example of syndrome extraction.

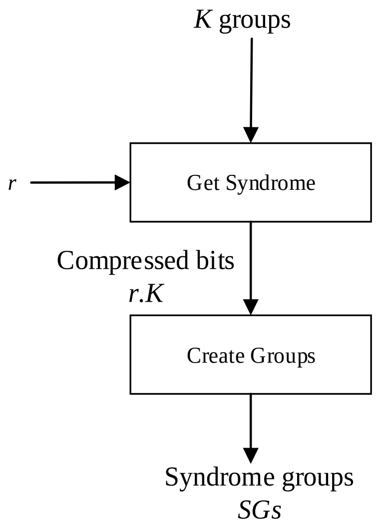


Figure 13: Syndrome Extraction Details. Using *K* Groups and *r*, *r.K* Compressed Bits are Extracted, then Form Compressed Bits into Syndrome Groups *SGs*

Algorithm 14. Syndrome extraction (Figure 13) from *K* groups.

Input:

- *KGs*: *K* groups, size $K \times n$, *K*: number of groups, *n*: number of bits in each group.
- $r \in [1, \ldots, n]$ where $1 < r < n$.

Output:

- *SGs*: syndrome groups, size $K\ (n - r)$, *K*: number of groups, *r:* number of bits in each syndrome group.

Steps:

1. Get the $[SGs(k,1), \ldots, SGs(k\ , r)]$ in each group

$for\ i = 1{:}K$

$\quad for\ j = 1{:}r$

$\quad SGs(i,j) = KGs(i,j)$

$\quad end$

end

Example 14. Syndrome extraction (Figure 13) from *K* groups.

Let's consider the *MEI* from Example 11. The same procedure will apply to have *KGs* groups after decomposing, collecting, selecting, and shuffling and division procedure. *n* and *r* are received with *MEI*

Input:

- $r = 4$

$$- KGs = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \end{bmatrix}$$

Output:

- *SGs*: syndrome groups, size $K\ (n - r)$

Steps:

1. $x = K \times r = 6 \times 4 = 24$ bits

$$\left[\begin{array}{cccc|cccc} \multicolumn{4}{c|}{\overbrace{\hphantom{1\ 1\ 1\ 1}}^{Syndrom\ groups}} & \multicolumn{4}{c}{\overbrace{\hphantom{1\ 0\ 1\ 0}}^{Secret\ data}} \\ 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \end{array}\right]$$

2. The left side of *KGs* are the syndrome groups.

$$SGs = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

With K_{ENC}, approximate image *AI* is constructed using the estimating algorithm as in Section 2.1.3. Then, *AI* is encrypted when the *SD* are embedded, the original image was encrypted. Then, encrypted approximate image is decomposed into 4 segments *AI1, AI2, AI3, AI4* using (4). After that, the *L* bits are selected from MSBs *AI2, AI3, AI4* segments using selection key K_{SL}. Then, the selected bits are shuffled using shuffle key K_{SF}. The shuffled bits are denoted as *UG*.

Shuffled bits *UG* are divided into *K* groups [*UG* (*k*, 1),…, *UG* (*k*, 2),…, *UG* (*k*, *n*)] using create groups Algorithm 6, each group has *n* bits. İn other words, we follow the same procedure in MSBs selection phase. As a result *UG* groups are produced with size $K \times n$ that will be used in decoding algorithm.

In decoding algorithm, the log-likelihood ratios (*LLR*) are calculated using *q*, where *q* is the crossover probability of the channel.

$$\begin{aligned} LLR(k,i) &= \log \frac{\Pr\left[R(k,i)=0 \mid UG(k,i)\right]}{\Pr\left[R(k,i)=1 \mid UG(k,i)\right]} \\ &= \left[1-2UG(k,i)\right] \log \frac{1-q}{q} \end{aligned} \qquad i=1,2,\ldots,n \qquad (14)$$

Using *LLR* defined by (14), *SGs* groups, *H* matrix and *UG* groups, the original bits [*R* (*k*, 1), *R* (*k*, 2), *R* (*k*, *n*)] are restored. Decoding algorithm in [2] is not specify clearly and not in its reference [15], so we use Sum-Product decoding algorithm [16].

To get the recovered image, *UG* groups are replaced with *R* groups (Figure 12) to get *R*\` groups. Then, the replaced bits *R*\` groups are inversed shuffled (Figure 12) using K_{SF}, same steps in Algorithm 9. MSBs in approximate image are replaced (Figure 12) with inversed shuffle bits to produce modified segments *AI2*\`, *AI3*\`, *AI4*\`. Four segments *AI1*, *AI2*\`, *AI3*\`, *AI4*\` are composed (Figure 12) to produce encrypted decoded image.

Using encryption key K_{ENC}, the encrypted decoded image are decrypted (Figure 12) to get decoded image.

Pseudo code of sum-product algorithm (Figure 12) is detailed in Algorithm 13. Example 15 shows sum-product decoding (Figure 12) of *UG* groups.

Algorithm 13. Sum-Product Decoding (Figure 12) of *UG* groups to obtain *R* groups.

Inputs:

- *S*: Syndrome, size $1 \times r$.
- *H*: with size $r \times n$.
- *UG*: size $1 \times n$.

Outputs:

- *R*: size $1 \times n$

Steps:

1. Initiation z

for $i = 1:n$

$LL = \log((1-q)/q)$

$z(i) = (1-(2 \times UG(i))) \times LL$

end

2. $Iter = 1;$

$Iter_{\max} = 10;$

$I = 0$

for $x = 1:n$

for $y = 1:r$

$N_{y,x} = z_i$

end for

end for

---

3. Check

$while(Iter < Iter_{\max})$

for $y = 1:r$ do

for x $\in C_y$ do

$$By, x = \log\left(\frac{1+\prod_{x'\in C_y, x'\neq x}\tanh(N_{y,x'}/2)}{1-\prod_{x'\in C_y, x'\neq x}\tanh(N_{y,x'}/2)}\right)$$

end for

end for

4. Test

for $x = 1:n$ do

$L_x = \sum_{y\in Ax} B_{y,x} + z_x$

$$J_x = \begin{cases} 1, & L_x \le 0 \\ 0, & L_x > 0. \end{cases}$$

end for

if $Iter = Iter_{\max}$ or $J.H^{\mathrm{T}} = S$ then

Finished

else

5. Bit messages

for x $= 1:n$ do

for y $\in Ax$ do

$N_{y,x} = \sum_{y'\in A_x, y'\neq y} B_{y',x} + r_x$

end for

end for

$Iter = Iter + 1$

end if

until Finished

Example 15. Sum-Product decoding (Figure 12) of *UG* groups.

Let's consider $K_{1\times 12}$ as original group with n =12.

$$K = \begin{bmatrix} 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 1 \ 0 \end{bmatrix}$$

H matrix is constructed using Gallager method with size 9×12 showed as follow

$$H = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

K is compressed using (6) to obtain the syndrome *S*

$$S=K.\, H^T = [0\ 0\ 1\ 0\ 0\ 1\ 0\ 1\ 0].$$

Let's consider received group as

$$UG = [0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 1\ 1\ 1\ 0]$$

To recover *K*: *S*, *UG* and *H* are used in decoding.

Initially, *p* values are calculated using (15)

$$p_i = \begin{cases} \log \dfrac{p}{1-p}, & \text{if } UG_i = 1, \\ \log \dfrac{1-p}{p}, & \text{if } UG_i = 0. \end{cases} \tag{15}$$

We obtain *p* as follows

$$\begin{bmatrix} 2.197 & 2.197 & 2.197 & -2.197 & 2.197 & 2.197 & 2.197 & 2.197 & -2.197 & -2.197 & -2.197 & 2.197 \end{bmatrix}$$

Negative values refer to 1's and positive values refer to 0's. Then, a matrix *N* with size $r\times n$ are defined contains zeros. Each element in each row of *M* matrix multiplied to the corresponding element in *p*. For example, the values in the first row in *N* matrix will be $N_{1,1} = 2.197, N_{1,2} = 2.197, N_{1,3} = 2.197, N_{1,4} = -2.197$. The other values will be zeros since the first 4 values in the first row in *H* matrix are 1's and the other are zeros. *N* matrix shows as follows:

$$\begin{bmatrix} 2.197 & 2.197 & 2.197 & -2.197 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2.197 & 2.197 & 2.197 & 2.197 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -2.197 & -2.197 & -2.197 & 2.197 \\ 2.197 & 0 & 2.197 & 0 & 0 & 2.197 & 0 & 0 & 0 & -2.197 & 0 & 0 \\ 0 & 2.197 & 0 & 0 & 0 & 0 & 2.197 & 2.197 & 0 & 0 & 0 & 2.197 \\ 0 & 0 & 0 & -2.197 & 2.197 & 0 & 0 & 0 & -2.197 & 0 & -2.197 & 0 \\ 2.197 & 0 & 0 & -2.197 & 0 & 0 & 2.197 & 0 & 0 & -2.197 & 0 & 0 \\ 0 & 2.197 & 0 & 0 & 0 & 2.197 & 0 & 2.197 & 0 & 0 & -2.197 & 0 \\ 0 & 0 & 2.197 & 0 & 2.197 & 0 & 0 & 0 & -2.197 & 0 & 0 & 2.197 \end{bmatrix}$$

Graphical representation of *H* matrix are represented in Figure 14. Using graphical representation of *H* matrix is used in decoding process. *UG* values are assigned into variable nodes in left side and *S* values are assigned into check nodes in right side.

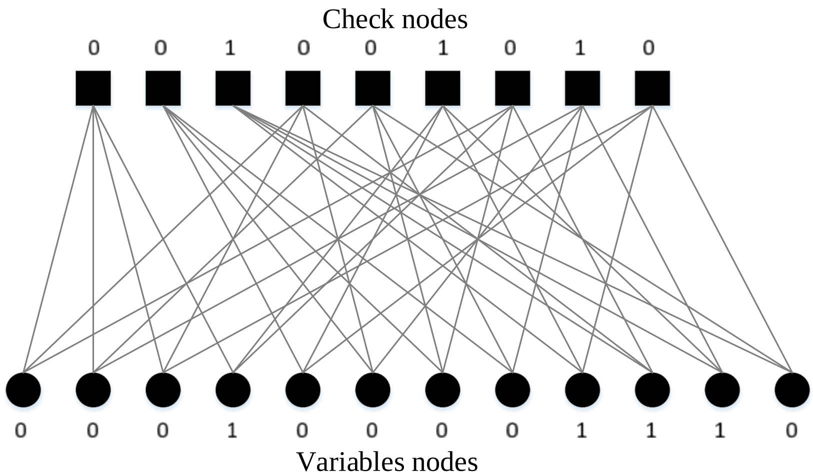


Figure 14: *H* Matrix Graphical Representation. Left Side are Variable Nods Contain *UG* Values. Right Side are Check Nods Contain *S* Values. 1's in *H* are Represented as a Connection between Variable Nods and Check Nods

Number of iteration are initialized, such as Iteration No = 7. The next step is to calculate the outer probabilities at the check nodes using (16)

$$B_{j,i} = \log\left(\frac{1+\prod_{i'\in C_j, i'\neq i} \tanh(M_{j,i'}/2)}{1-\prod_{i'\in C_j, i'\neq i} \tanh(M_{j,i'}/2)}\right) \tag{16}$$

C vector containing the indices of bits variable nodes. For first check node, C= {1, 2, 3, 4}. Thus, outer probability of first bit depends on probability of second, third and fourth bits

$$B_{1,1} = \log\left(\frac{1+\tanh(N_{1,2}/2)\tanh(N_{1,3}/2)\tanh(N_{1,4}/2)}{1-\tanh(N_{1,2}/2)\tanh(N_{1,3}/2)\tanh(N_{1,4}/2)}\right) = -1.131$$

Similarly, the outer probability from first check to second bit depends on first, third and fourth bits

$$B_{1,2} = \log\left(\frac{1+\tanh(N_{1,1}/2)\tanh(N_{1,3}/2)\tanh(N_{1,4}/2)}{1-\tanh(N_{1,1}/2)\tanh(N_{1,3}/2)\tanh(N_{1,4}/2)}\right) = -1.131$$

$$B_{1,3} = \log\left(\frac{1+\tanh(N_{1,1}/2)\tanh(N_{1,2}/2)\tanh(N_{1,4}/2)}{1-\tanh(N_{1,1}/2)\tanh(N_{1,2}/2)\tanh(N_{1,4}/2)}\right) = -1.131$$

$$B_{1,4} = \log\left(\frac{1+\tanh(N_{1,1}/2)\tanh(N_{1,2}/2)\tanh(N_{1,3}/2)}{1-\tanh(N_{1,1}/2)\tanh(N_{1,2}/2)\tanh(N_{1,3}/2)}\right) = 1.131$$

Repeating for all checks gives the outer LLR: B is represented as follows:

$$\begin{bmatrix}
-1.131 & -1.131 & -1.131 & 1.131 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1.131 & 1.131 & 1.131 & 1.131 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1.131 & -1.131 & -1.131 & 1.131 \\
-1.131 & 0 & -1.131 & 0 & 0 & -1.131 & 0 & 0 & 0 & 1.131 & 0 & 0 \\
0 & 1.131 & 0 & 0 & 0 & 0 & 1.131 & 1.131 & 0 & 0 & 0 & 1.131 \\
0 & 0 & 0 & -1.131 & 1.131 & 0 & 0 & 0 & -1.131 & 0 & -1.131 & 0 \\
1.131 & 0 & 0 & -1.131 & 0 & 0 & 1.131 & 0 & 0 & -1.131 & 0 & 0 \\
0 & 1.131 & 0 & 0 & 0 & 1.131 & 0 & 1.131 & 0 & 0 & -1.131 & 0 \\
0 & 0 & -1.131 & 0 & -1.131 & 0 & 0 & 0 & 1.131 & 0 & 0 & -1.131
\end{bmatrix}$$

Outer values from check nodes are inner values for variable nodes

$$L_i = p_i + \sum B_{i,j}$$

The 1-st bit has outer $LLRs$ from the 1st, 4th and 7th checks and an inner to first variable nodes as follows:

$$L_1 = p_1 + B_{1,1} + B_{1,4} + B_{1,7} = 1.066$$

$$L_2 = p_2 + B_{1,1} + B_{1,5} + B_{1,8} = 3.328$$

Repeating for variable nodes. *L* represented as follows:

$$\begin{bmatrix} 1.066 & 3.328 & -1.195 & -3.328 & 3.328 & 3.328 & 5.590 & 5.590 & -3.328 & -3.328 & -5.590 & 3.328 \end{bmatrix}$$

Values of *L* are converted into binary. If $L_i < 0$ then $J_i = 1$, otherwise $J_i = 0$. *J* showed as follows:

$$\begin{bmatrix} 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

Next, *S`* are calculated using (7).

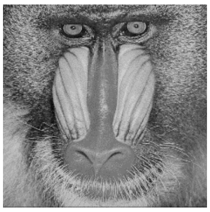
$$S` = J \,.\, H^T$$

If *S*= =*S`* then, *J* is obtained to the recovered one from *UG*, otherwise, *N* is recalculated using obtained new values

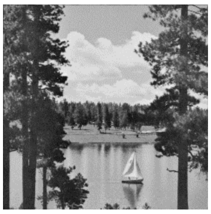
2.2 Qian-Zhang Experimental Settings and Results

Experiments in [1] are conducted using one *H* matrix with size r = 3840 and n = 6336, and selection ratio α =1. Using Lena, Baboon, Lake, and Man images of size 512 × 512 grayscale as shown in Figure 15.

a) Lena



b) Baboon



c) Lake

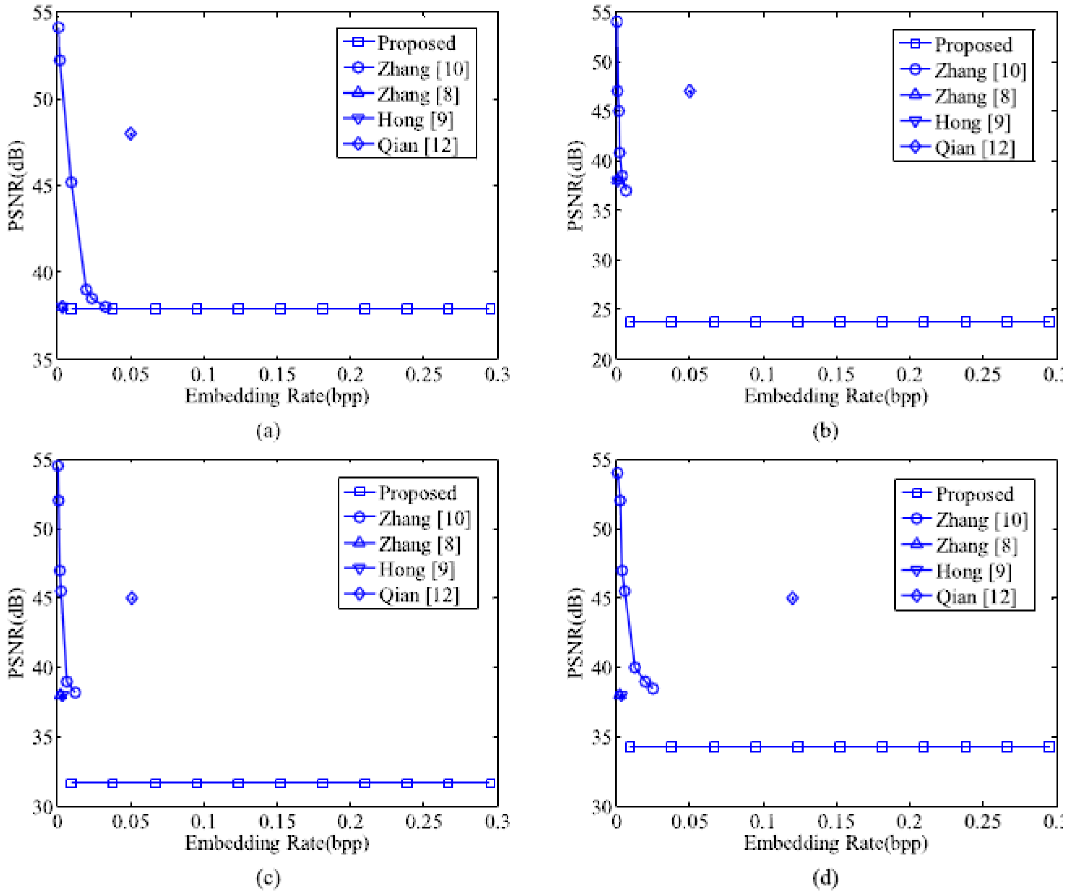
d) Man

Figure 15: Gray Scale Images are Used in Qian-Zhang

The total number of collected bits from *EI2*, *EI3*, *EI4* is $(3 \times 512 \times 512)/4$ = 196608 bits. L bits are selected from collected bits where selection ratio α =1, using (5) L=1.0×196608=196608. These selected bits are shuffled then divided into 31 groups using (7) where n = 6336, K=$\lfloor 196608/6336 \rfloor = 31$. Each group is encoded with the $H_{3840\times6336}$matrix using (8). The resultant syndrome group *SGs* with size 31×3840 . Number of bits to be embedded in each group is $n - r$ = 6336 - 3840=2496. The total number of bits is K (n - r) =31 × 2496 = 77376. Embedding capacity is obtained from [1] defining using (11)

$E_{emb=} \frac{3\times1.0\times(6336-3840)}{4\times6336}$ = 0.2952 bpp.

The PSNR of approximate image keeps constant for all embedding capacity. Figure 16 results of PSNR of approximate image in [1].



Comparisons of four different approaches using the images. (a) *Lena*. (b) *Baboon*. (c) *Lake*. (d) *Man*.

Figure 16: Results for PSNR of Approximate Image in Qian-Zhang Scheme. PSNR of Approximate Image is Constant When Embedding Capacity Changes [1]

When decoding fails, PSNR of decoded image decreases when embedding capacity increases. Figure 17 shows results of PSNR of decoded image in [1] when decoding fails.

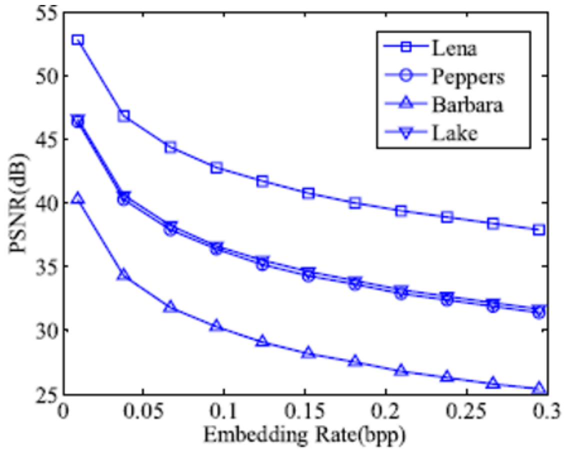


Figure 17: PSNR of Decoded Image When Decoding Fails [1]

2.3 Review of RDH schemes

In this section, we review RDH data embedding techniques. In [2], a RDH data embedding scheme is proposed that embedded bits into an encrypted image by flipping a small number of LSBs (less than 5) of pre-defined pixels in the encrypted image after dividing the encrypted image into blocks. Conducted experiments of [2] embedded 256 bits to gray scale Lena image with size 512×512 pixels, results PSNR value 37.9 dB after decrypting encrypted image holds embedded data, with error extraction rate 1.21 % from recovered image. In [13], a n new algorithm is proposed to better estimate the smoothness of image blocks. This algorithm improved data extraction and image recovery strategies in [2]. Using this algorithm, with error extraction rate from recovered image drops from 1.21% in [2] to 0.34% in [13].

In [14], an RDH method proposed embeds secret data after compressing bits in encrypted image. The test images are sized 512×512. The embedding capacity achieves 0.033 bpp for Lena image with PSNR 37.96 dB after decryption while for Man and Lake 0.0250 bpp and 0.0130 bpp respectively with PSNR 37.95 dB for both after decryption.

In [6], an RDH method proposed embeds secret data after encoding encrypted image. The original images are used sized 512×512. Embedding capacity for Lena is 0.043 bpp and for man 0.035 bpp with PSNR 38.1 dB.

In [1] , the embedding capacity achieves 0.2952 bpp for all images with different PSNR of approximate image.

2.4 Problem Definition

The following problems related to Qian-Zhang scheme are solved in the thesis:

1. Implement the Qian-Zhang scheme and get the same experimental results as in [1].
2. The selection key K_{SL} that is used is not specified clearly in [1]. Thus, we need proposing an algorithm to generate different selection keys depending on the selection ratio.
3. The shuffle key K_{SF} generation is not defined in [1]. Hence, we have to propose an algorithm to generate shuffle key.
4. The encryption key K_{ENC} used is not defined in [1]. Thus we have to propose an algorithm to generate an encryption key.
5. The decoding process done in [1] is not clearly explained. We used sum-product decoding algorithm [20].
6. In [1], only one H matrix is used in (7) of size n = 3840 and r = 6336 with ratio $R = r/n \approx 0.62$ without specifying the construction method of this matrix. Hence, we need to find a construction method provides higher embedding capacity with higher PSNR of decoded image. We conducted experiments on two construction methods: Gallager and MacKay-Neal by generating different H matrices for construction method and applying these matrices scheme [1].

7. Qian-Zhang scheme studies relation between the embedding capacity and the quality of approximate image using only one matrix. We have to confirm this relation by conducting experiments in [1] using several constructed *H* matrices by Gallager method each of with different sizes.

8. We need to extend the experiments by generating different *H* matrices with different sizes with the same ratio *R* to find the relation between the *H* matrix size and the quality of the recovered image. Moreover, these experiments are used to find the relation between the *H* matrix size and decoding time.

9. The extension of the experiments settings of [1] also shall be done by generating different *H* matrices with different *R* and sizes to find the relation between *H* matrix ratio and the embedding capacity and to be confirmed by conducting experiments.

10. When the decoding fails, the relation between embedding capacity and the quality of recovered image is investigated.

2.5 Summary of Chapter 2

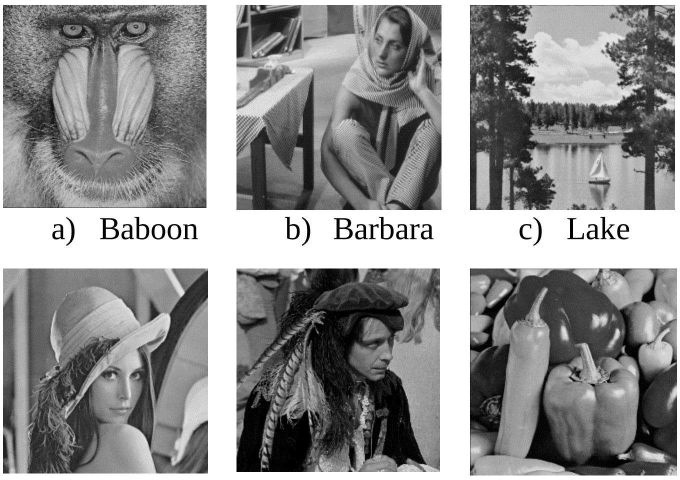
In this chapter we have:

1. Presented Qian-Zhang scheme [2] experimental results and settings
2. Presented the related work on RDH and provided known experimental results on PSNR and embedding capacity.
3. Problem definition for the thesis is given.

Chapter 3

QIAN-ZHANG SCHEME IMPLEMENTATION

In this chapter, we present implementation of Qian-Zhang scheme [1]. We have chosen 6 images as shown in Figure 18 with size 512 × 512 from [21] for implementation. These images are “pgm” format, in this format we couldn’t measure the quality of approximate and decoded images. Hence, these images are converted into “bmp” in order to identify each one. In Appendix A.2 the implementation of image conversion is given. In Appendix A.2, line 3, images with “pgm” format are read. In lines 6-13, each image is converted into matrix contains pixel values, then saved as “bmp” format.



a) Baboon b) Barbara c) Lake

d) Lena e) Man f) Peppers

Figure 18: Images used in our implementation for Qian-Zhang scheme

We implemented Qian-Zhang scheme as follows:

1. Selection ratio α is fixed equal to 1, Appendix A.3, to find:

1.1. Optimal H matrix construction method.

1.2. The relation between H matrix size and decoding time and PSNR of decoded image.

1.3. Relation between H matrix ratio and embedding capacity.

2. Selection ratio $\alpha \in [0.1, 1.0]$ with step 0.1, Appendix A.4 , to find:

2.1 Relation between embedding capacity and PSNR of approximate image.

2.2 Relation between embedding capacity and PSNR of decoded image.

3.1 Qian-Zhang Implementation

Appendix A.3 present implementation of Qian-Zhang scheme using 9 H matrices constructed by Gallager and MacKay-Neal method with selection ratio α equal to 1.0, line 9. In line 3, the images loaded from directory. In line 5, we declare variable of for storing PSNR for each decoded image for each H matrix. In line 6, we declare variable for storing decoding time for each image of each H matrix. Line 7, define the column size of H matrix which is equal to n. In lines 8-54, each image goes through three stages: image encryption in line 20, data hiding in line 28 and data extraction and image recovery in line 46. The resultant PSNR of decode image is assigned in line 48 for each H matrix of each image and decoding time is assigned in line 49 for each H matrix of each image. Code 1 shows MATLAB code for Qian-Zhang implementation steps. In Code 1 line 1, the original image is encrypted. Then, the secret data is embedded in line 2. The data is extracted and the image is recovered in line 3.

Code 1: Qian-Zhang implementation

```
1. [EncryptedImage]=encrypt(OriginalImage,EncryptionKey );
2.Marked_encrypted_image,selectionkey,Shufflekey,L,r,H,syndorm,kgroups,secret
Data]=HideData(EncryptedImage,selectionRatio,seed,numberofbits,secretData,im
Name,j);
```

```
3.[extractedData,ApproximateImage,PSNR,RecievedImage,
DecodedPSNR]=Reciever(Marked_encrypted_image,selectionkey,Shufflekey,L,r,
numberofbits,EncryptionKey,imName,OriginalImage,H);
```

The implementation for three stages is described in details in the next sections.

3.1.1 Stage 1: Image Encryption Implementation

We use 512×512 grayscale images in Appendix A.1 Figure A.1. In Appendix A.3.1, we implemented Section 2.1.1. In line 4, the image is converted into binary values using de2bi MATLAB function to obtain 262144×8 binary image according to (1).

Then, the binary image is encrypted in line 5 by MATLAB xor function using encryption key, which is obtained from Appendix A.3.1.1, according to (2). After that, encrypted binary image is converted into decimal values using bi2de MATLAB function to obtain 262144 ×1 decimal image in line 6. In lines 7 and 8, the decimal image is converted into 512×512 grayscale encrypted image using reshape MATLAB function according to (3).

In Appendix A.3.1.1 line 2, encryption key K_{ENC} is generated randomly with size (512×512) ×8 binary values. Then in line 6, K_{ENC} are stored in '.mat' file in order not to be generated each time during running. Code 2 shows the MATLAB implementation of image encryption stage. In line 1, the original image is converted into binary values. Then, in line 2, the binary values are encrypted using encryption key. After that, the encrypted binary are converted into pixel values in line 3.

Code 2: Image Encryption Implementation Stage

```
1.binary=de2bi(OriginalImage,8,2,'left-msb');
2.binaryImage=xor(binary,EncryptionKey);
3.EncryptedImage=bi2de(binaryImage,'left-msb');
```

3.1.2 Stage 2: Data Hiding Implementation

In this section we present implementation of, Section 2.1.2, of three phases.

In Appendix A.3.2 lines 3-12, presents MSBs selection phase steps. The encrypted image is decomposed into 4 segments *EI1*, *EI2*, *EI3*, *EI4* in line 3. Then, in line 4, the MSBs are collected from *EI2*, *EI3*, *EI4*. After that, in line 5, the selection key K_{SL} is constructed with selection ratio α equal to 1.0. After constructing K_{SL}, number of bits are selected from collected bits in line 9. After selecting bits, shuffle key K_{SF} is generated based on selected bits in line 10. Then, the selected bits are shuffled using K_{SF} in line 12.

Lines 13-23 present encoding and compressing phase steps. In line 14, K groups are created. In lines 18-21, *H* matrix is loaded for encoding. In line 23, syndrome groups *SGs* is obtained.

Lines 25-31 present embedding secret data phase steps. In line 5, secret data *SD* is embedded into syndrome groups *SGs*. In line 27, the embedded groups is reversed shuffle. In line 29, MSBs are replaced with inversed shuffle bits. Marked encrypted image *MEI* is constructed in line 31.

Code 3 shows the steps of data hiding stage implementation. Lines 1-5 presents MSBs selection phase steps. In line 1, the encrypted image is decomposed into 4 segments. Then, the MSBs are collected from the last three segments in line 2. After that, in line 3, number of bits are selected using selection key. In line 4, the shuffle key is generated. In line 5, the selected bits are shuffled using shuffle key.

Lines 6-7 presents encoding and compressing phase steps. In line 6, K groups are created. In line 7, syndrome groups *SGs* is obtained.

Lines 8-11 present embedding secret data phase steps. In line 8, secret data *SD* is embedded into syndrome groups *SGs*. In line 9, the embedded groups is reversed shuffle. In line 10, MSBs are replaced with inversed shuffle bits. Marked encrypted image *MEI* is constructed in line 11.

Code 3: Data Hiding Stage Implementation

```
1. [E1,E2,E3,E4]=decompose(A);
2. [collectedbits]=collectBits(E2,E3,E4);
3. [selectedBits]=selectbits(collectedbits,selectionkey);
4. [shufflekey]=generateshuffelkey(selectedBits);
5. [shuffledbits]=shufflebits(selectedBits,shufflekey);
6. [kgroups,reminderBits]=createGroups(shuffledbits,numberofbits);
7. [syndorm]=GetSyndorm(kgroups,H);
8. [image_after_embedding,r,secretData]=embedData(kgroups,syndorm,additional
   Bits)
```

9. [inverseSuhffledBits]=inverseshuffle(image_after_embedding,reminderBits,shufflekey);
10. [EE2,EE3,EE4]=returnBits(inverseSuhffledBits,E2,E3,E4,selectionkey);
11. [MarkedEncryptedImage]=compose(A,E1,EE2,EE3,EE4);

Most Significant Bits (MSBs) Selection Phase

This section presents implementation of MSBs selection phase.

Appendix A.3.2.1 present implementation of decomposing encrypted image into *EI1*, *EI2*, *EI3*, *EI4*. In lines, 2-14, the size encrypted image is checked if it is power of 2. In lines 15-18, the encrypted image is decomposed.

Appendix A.3.2.2 presents implementation collecting of MSBs from *EI2*, *EI3*, *EI4*. In lines 5-7, MSBs from *EI2* are collected. In lines 8-11, MSBs from *EI3* are collected. In lines 12-15, MSBs from *EI4* are collected. In line 16, the collected bits *CB* from are *EI2*, *EI3*, *EI4* concatenated into row vector.

For selecting bits, we have to construct selection key K_{SL} that is used to select number of bits, L, pseudo randomly from the collected MSB bits. The construction of K_{SL} depends on the selection ratio (α) and selection seed (Seed). Selection ratio, α, is in range [0.1, 1.0] and we define seed as a positive integer number. We fix selection ratio equal to 1.0, hence, L will be all the MSBs are selected according to (5).

The size of K_{SL} will be same as L containing the indices between 1 and L. Algorithm 15 describes the algorithm of constructing selection key K_{SL} and Example 15 shows an example of K_{SL}.

Algorithm 15. Selection key construction

Input:

- Seed $\in Z^+$, Z is positive integer numbers set
- α: selection Ratio, $\alpha = L/(3XY/4)$.
- *CB*: MSB's , [c_1, c_2, …, $c_{|CB|}$]; $c_i \in \{1,0\}$, |*CB*| = $3XY/4$

Output:

- K_{SL}: Selection Key = [$K_{SL1}, K_{SL2,\ldots,} K_{SLL}$], $L = \alpha \times \left(\frac{3XY}{4}\right)$.

Steps:

1. Take the length of the *CB*: T .
2. Determine the number of the bits to be selected based on α:$L = \lfloor \alpha \times T \rfloor$.
3. Seeds the random number generator using the seed
4. Select randomly number between 1 and *T*.
5. Checks whether in the K_{SL} or not, if not stored in the K_{SL}, if yes, select again another number. Until we generate a key with length *L*.

The next pseudo-code, Code 4, implements Algorithm 15:

Code 4 :

$T = |CollectedBits|$

$L = \lfloor \alpha \times T \rfloor$

$randomNumberGenerator(Seed)$

$K_{SL}[1, L] = \{\}$

$index = 1$

$while \;\; index <= L$

$\quad y = select \; Random \; Number \; between \; 1 \; and \; L$

$\quad if \, (y \text{ in } K_{SL})$

$\qquad repeat$

$\quad else$

$\quad K_{SL}(index) = y$

$\quad index = index + 1$

$\quad end \; if$

end

Example 15. Selection key K_{SL} construction

Let's consider

Input:

- α=1.0,
- T= 48. Total number of collected bits *CB*
- Seed = 4.

Output:

- K_{SL}

  [47 27 35 34 11 1 13 21 38 10 42 48 8 29 19 3 46 9 4 36 20 26 6 31 32 30 37 25 33 43 18 24 45 40 44 16 28 41 5 12 7 39 17 23 2 22 14 15]

Steps:

1. L = 1.0×48=48. Number of bits to be selected

2. K_{SL} [1, L] = {}

3. index=1

4. While

5. y =47. Based on seed

6. if (y in K_{SL}) → false

7. K_{SL} (*index*) = [47]. *index* = *index* +1. Go to step 3.

Appendix A.3.2.3 presents the implementation of the K_{SL} construction. We fixed selection seed equal to 4 in line 2. In line 3, total number, T, of collected bits CB are calculated. According to (5), L bits is determined in line 4 which is length of K_{SL} using total number of collected (T) and α equal 1.0. In order to selected bits pseudo randomly, in line 5, we use rng MATLAB function which is control the generation of random number between 1 and L based on selection seed. In line 6, K_{SL} is initialized with size L containing zeros. Lines 7-14 describes assigning random numbers between 1 and L to K_{SL} 'randi' function which is controlled by rng function . If the random number is exist in K_{SL}, another random number is generated, otherwise, generated random number is added into K_{SL}. At the end of this function, K_{SL} is created containing all the indices between 1 and L to select bits from the collected bits CB.

Appendix A.3.2.4 presents selecting bits SB from collected bits CB using constructed selection key K_{SL}. In line 2, the bits are selected using selection key K_{SL} from collected bits CB.

For shuffling bits, shuffle key K_{SF} is constructed in Appendix A.3.2.5. For K_{SF} construction, length of the selected bits L is used. Then, we select all the prime numbers in L. After that, a number from selected prime number are chosen that is not equal to 1, not selected before and GCD between the number and L equal to 1. Algorithm 16 below describes the construction of K_{SF}. An example of K_{SF} construction is given in Example 16.

Algorithm 16. Shuffle key construction

Input:

– SB: the Selected Bits = [1….L],$L = \alpha(3XY/4)$.

Output:

– K_{SF}: the Shuffle Key $\in Z^{+}$

Steps:

1. Find the length of the SB : L

L: length of selected bits

2.declare empty “selected primes” array

2. Find all prime numbers form L: Ps.

Ps[]= all prime numbers in L

3. Select randomly prime number P from Ps such that $P \neq 1$, P not in “selected primes” array, and gcd (P, L) =1, where gcd is greatest common divisor.

$$K_{SF} = select\ random\ prime\ number\ in\ Ps$$
$$if\,(K_{SF} = 1)$$
$$continue$$
$$end$$
$$if\ \ GCD(K_{SF} and\ \ L)$$
$$done = 1$$
$$end\ \ if$$

The next pseudo-code, Code 5, implements Algorithm 16:

Code 5:

$K_{SF} = construct\ Shuffle\ key(SB)$

{

$L = length\ of\ selected\ bits$

$selected\ primes = []$

$while\ \ done \neq 1$

$\quad Ps\ [] = all\ prime\ numbers\ \text{in}\ L$

$\quad K_{SF} = select\ random\ prime\ number\ in\ Ps$

$\quad if\ (K_{SF} = 1\ or\ K_{SF}\ in\ selected\ primes)$

$\quad continue$

$\quad end$

$if\ \ GCD(K_{SF} and\ L)$

$\qquad done = 1$

$else$

$selected\ primes = K_{SF}$

$\quad end\ \ if$

$end\ \ while$

}

Example 16. Example of K_{SF} construction.

Let's consider L = 48

Steps:

Selectedprimes= [];

1. While done ≠ 1
2. Prime numbers in Z_{48} are

p=[2 3 5 7 11 13 17 19 23 29 31 37 41 43 47]

3. Select randomly from PN : K_{SF} =3 and check 3=GCD(3, 48) ==1 is false
4. Selectedprimes= K_{SF} and go to step 2
5. The next random number K_{SF} =13 → GCD(13 and 48)=1→ true
6. End

We implemented constructing shuffle key K_{SF} function as in Appendix A.3.2.5 taking the selected bits *SB* as an input. In line 3, the number of selected bits, *L*, are obtained. In lines 4-10, all prime numbers is obtained depending on the number of selected bits *L*. Then, in line 6, select randomly one of the prime numbers (*x*). In line 10, if the Greatest Common Divisor (GCD) between the *x* and *L* is equal to 1, then K_{SF} = x, otherwise, another prime number is selected.

After constructing shuffle key K_{SF}, shuffle bits *SHB* are obtained from Appendix A.3.2.6. Line 4, shuffle vector is created containing indices after shuffling. Then, in line 5, the shuffled bits *SHB* are obtained from selected bits *SB* using shuffle row.

Encoding and Compressing Phase Implementation

This section presents implementation of encoding and compressing phase.

In Appendix A.3.2.7 line 4, *K* groups are calculated which is defined by (7). In line 5, number of remainder bits are calculated. In lines 5-8, the remainder after division are stored in row vector in line 8. In lines 9-11, groups are created with size $K \times n$.

After create groups, we have to construct *H* matrix to obtain syndrome groups. For constructing *H* matrix, we use two methods for constructing: Gallager method and MacKay-Neal method. For Gallager method, we used implemented code [21], which takes the number of columns as an input and produced *H* matrix with size $r \times n$. For MacKay-Neal method, we used implemented code [22] which takes (*r*, *n*, method, noCycle, onePerCol) as inputs to produce *H* matrix with size $r \times n$. We have generated different 9 *H* matrices for each method and store them in ".mat" file format. Appendix A.3.2.8 shows the code of storing *H* matrices. Line 4, we construct Gallager *H* matrix which takes *n* as an input that determine the number of columns in *H* matrix. In lines

5-6, we construct MacKay-Neal H matrix which takes r and n, the other inputs determine the distribution of 1's in H matrix. For more details, see [22]. In line 7-11, we store the constructed H matrix with unique name. For Example, we use "HGST2_1" name for first H matrix constructed by Gallager for trial 2. "HGST2_2" name for second H matrix constructed by Gallager for trial 2, and so on. The detailed H matrix with its name are shown in Appendix. Details of Gallager and MacKay-Neal methods in Section 3.2.

After construction H matrix, in Appendix A.3.2.9 line 2, H matrix is transposed then, K groups are compressed using transposed H matrix according to (7) in line 4 to obtain syndrome groups.

Embedding Secret Data Phase Implementation

This section presents implementation of embedding secret data phase.

In Appendix A.3.2, in line 25, we implemented function to embed secret data. In line 27, the groups after embedding is inversed shuffled. Then, the modified MSB bits are replaced with original MSB bits in lines 29-31.

In Appendix A.3.2.10, in line 7-8, secret data are generated randomly with size $K(n-r)$ to be embedded. In line 9, the secret data is divided into K groups, each group with size n - r. In line 11, an embedded matrix defined with size $K \times n$. In line 12, secret data is embedded into embedded matrix in n - r space in each group. In line 13, the syndrome groups is assigned into embedded matrix in r space in each group.

In appendix A.3.2.11, the embedded group is reverse shuffle using constructed shuffle key. In line 7, the embedded matrix is converted into row vector. In line 9, the remainder bits are concatenated to the embedded matrix after conversion using horzcat function in MATLAB. In lines 12-13, the bits is reversed shuffle using shuffle key.

In Appendix A.3.2.12, collected MSBs are replaced with the inversed shuffle bits in line 2. In lines 6-5, MSB bits are collected and replaced from *EI2*. In lines 7-8, MSB bits are collected and replaced from *EI3*. In lines 9-10, MSB bits are collected and replaced from *EI4*. In lines 11-13, modified MSB bits are returned into *EI2* segment to get *EE2* segment. Same procedure for *EI3*, *EI4* in lines 14-19.

In Appendix A.3.2.13, in lines 2-6, *EI1*, *EE2*, *EE3*, *EE4* are composed according to (4) to obtain marked encrypted image.

3.1.3 Stage 3: Data Extraction and Image Recovery Implementation

This section presents implementation of Section 2.1.3. There are 3 options: data extraction, approximate image reconstruction, and lossless recovery.

Code 6 shows the implementation of data extraction and image recovery stage. In line 1, the data is extracted using selection key K_{SL}, shuffle key K_{SF}, L, n, and r from marked encrypted image *MEI*. In line 2, an approximate image is constructed using encryption key K_{ENC}. The last option in line 3, the data is extracted and the image is recovered.

Code 6: Data Extraction and Image Recovery Implementation

```
1.[extractedData]=DataExtraction(Marked_encrypted_image,selectionkey,Shufflekey,L,r,numberofbits);
2.[ApproximateImage,PSNR]=DecryptionAndEstimation(Marked_encrypted_image,EncryptionKey,imName,OriginalImage);
```

```
3.[RecievedImage,
   DecodedPSNR]=Recovery(Marked_encrypted_image,selectionkey,Shufflekey,H
   ,L,r,numberofbits,EncryptionKey,secretData,syndorm,kgroups,imName,fid2,Ori
   ginalImage);
```

Option 1: Data Extraction Implementation

This section present the implementation of option 1: data extraction.

In Appendix A.3.3, in line 2, the Marked encrypted image is decompose into 4 segments using (4) same as in Appendix A.3.2.1. In line 3, the MSB bits are collected same in Appendix A.3.2.2. In line 4, number of bits are selected using K_{SL} same as in Appendix A.3.2.4. In line 5, the bits are shuffled using shuffle key K_{SF} same Appendix A.3.2.6. The K groups are created same Appendix A.3.2.7. In line 7, the secret data are extracted by implementation function in Appendix A.3.3.1. In Appendix A.3.3.1 line 5, the secret data is extracted from K groups (K, $(r+1)\ldots n$). The extracted groups are converted into row vector in lines 6-8.

Code 7 shows the MATLAB code implementation of data extraction. In line 1, the marked encrypted image is decomposed into 4 segments. Then, in line 2, MSBs are collected from the last three segments. In line 3, number of bits are selected using the selection key. In line 4, the selected bits are shuffled using the shuffle key. After that, in line 5, the shuffled bits are divided into K groups. The secret data is extracted in line 6.

Code 7: Data Extraction Implementation

```
1. [V1,V2,V3,V4]=decompose(A);
2. [collectedbits]=collectBits(V2,V3,V4);
3. [selectedBits]=SelectBitsUsingSelectionKey(collectedbits,L,selectionKey);
4. [shuffledbits]=shufflebits(selectedBits,shuffleKey);
5. [kgroups,reminderBits]=createGroups(shuffledbits,n);
6. [extractedData]=extractData(kgroups,n,r);
```

Option 2: Approximate Image Reconstruction Implementation

This section present implementation of option 2: approximate image reconstruction.

In Appendix A.3.4 line 2, the marked encrypted image is decrypted using the encryption key according to (2) same in Appendix A.3.1 to obtain marked image. In line 6, marked image is decomposed into 4 segments using (4) same as in Appendix A.3.2.1. Then, in line 7, a reference image *BI* is constructed using interpolation function that we implemented in Appendix A.3.4.1. After that, reference image *BI* is decomposed using same as Appendix A.3.2.1. To get an approximate image *AI*, we implemented estimation function in Appendix A.3.4.2 using reference image *BI* and marked image *AI* according to (13) lines 9-12. Then, approximate segments after estimation is composed into one image to construct approximate image in line 13 same in Appendix A.3.2.13. In line 23, we use MATLAB PSNR function to get the PSNR of approximate image.

For bilinear interpolation, we implement function in Appendix A.3.4.1. In lines 3-5, X and Y coordinates are defined using meshgrid in MATLAB to expand the segment $EI1$. Then, in line 6, interpolated values are calculated using interp1 function in

MATLAB. Lines 7 and 8, interp1 function used bilinear extrapolation to calculated border values. In line 9, interpolated values are rounded and reference image *BI* is obtained.

In Appendix A.3.4.2, according to (13), in line 6, approximate segments is constructed.

Code 8 shows the MATLAB implementation of option 2, approximate image reconstruction. In line 1, the marked encrypted image is decrypted using the encryption key. Then, in line 2, the decrypted image is decomposed into 4 segments. Using the first segment, in line 3, the reference image is constructed using the bilinear interpolation. The reference image, in line 4, is divided into 4 segments. The MSBs estimation is calculated for the 4 segments in lines 5-8. After calculating the MSBs the approximate image is constructed by composing the 4 segments in line 9.

Code 8: Approximate Image Reconstruction Implementation

```
1.  [DecryptedImage]=decrypt(A,EncryptionKey);
2.  [A1,A2,A3,A4]=decompose(Marked_image);
3.  [B]=interplation(A1,Marked_image);
4.  [B1,B2,B3,B4]=decompose(B);
5.  [BB1] = calculate_approximate_image(A1, B1);
6.  [BB2]=calculate_approximate_image(A2, B2);
7.  [BB3]=calculate_approximate_image(A3, B3);
8.  [BB4]=calculate_approximate_image(A4, B4);
9.  [ approximateImage ] =compose(Marked_image,BB1,BB2,BB3,BB4);
```

Option 3: Lossless Recovery Implementation

This present implementation of option 3: lossless recovery.

Appendix A.3.5, shows lossless recovery steps. In this case, the data extracted perfectly in line 5 using same implementation in Appendix A.3.3. Then, in lines 7-20, the syndrome groups are extracted from marked encrypted image. In line 21, an approximate image is constructed then, in line 22, approximate image is encrypted. In lines, 23-28, encrypted approximate image is decomposed using implementation in AppendixA.3.2.1. Then MSB bits are collected from using implementation in Appendix A.3.2.2. After that, selected bits are obtained using implementation in Appendix A.3.2.4 using K_{SL}. Then, the selected bits are shuffled using implementation in Appendix A.3.2.6. Then, *K* groups are created using implementation in Appendix A.3.2.7. Using syndrome groups, *K* groups, and *H* matrix, we implemented decoding process in line 4 as given in Appendix A.3.5.1, (Section 3.3) to obtained decoded groups. Then, decoded groups are inversed shuffle using function in line 56. In line 57, the inversed shuffle bits are replaced with MSBs in in encrypted approximate image. After that, the segments after decoding are composed as given in Appendix A.3.2.13. To get decoded image, the composed image is decrypted as in line 61 as given in Appendix A.3.1.

Code 9 shows the MATLAB implementation of lossless recovery. In lines 1-6, the syndrome groups are extracted. In line 7, an approximate image is constructed. Then, in lines 8-14, the *K* groups ate obtained from approximate image using same steps in MSBs selection phase. In lines 15-18, using *H* matrix, extracted syndrome and *K* groups are used for decoding to recover the MSBs. The recovered MSBs are inversed shuffle in line 19 using the shuffle key. In lines 20-21, the MSBs are returned into

MSBs in segments in approximate image. Then, in lines 22-32, the recovered image is constructed after composing and decryption using encryption key.

Code 9: Lossless Recovery Implementation

```
1.  [E1,E2,E3,E4]=decompose(A);
2.  [collectedbits]=collectBits(E2,E3,E4);
3.  [selectedBits]=SelectBitsUsingSelectionKey(collectedbits,L,selectionKey);
4.  [shuffledbits]=shufflebits(selectedBits,Shufflekey);
5.  [kgroups,reminderBits]=createGroups(shuffledbits,numberofbits);
6.  [compressedData,compressedGroup]=GetCompressedData(kgroups,numberofbits,r);
7.  [ApproximateImage,ApproPSNR]=DecryptionAndEstimation(A,EncryptionKey,imName,OriginalImage);
8.  [EncryptedApproximateImage]=encrypt(ApproximateImage,EncryptionKey);
9.  [E1,E2,E3,E4]=decompose(EncryptedApproximateImage);
10. n=numberofbits;
11. [collectedbits]=collectBits(E2,E3,E4);
12. [selectedBits]=selectbits(collectedbits,selectionKey);
13. [shuffledbits]=shufflebits(selectedBits,Shufflekey);
14. [kgroupsappro,reminderBits]=createGroups(shuffledbits,n);
15. for i=1:r
16. [decodedString]=decodeStatisticsOriginal(compressedGroup(i,1:end),kgroupsappro(i,1:end),H);
    [decodedString]=decodeStatisticsOriginal(compressedGroup(i,1:end),kgroupsappro(i,1:end),H,kgroupsOriginal(i,1:end));
```

```
17. decoded(i,1:numberofbits)=decodedString;
18. end
19. [inverseShuffledBits]=inverseshuffle (decoded,reminderBits,Shufflekey );
20. [E1,E2,E3,E4]=decompose(EncryptedApproximateImage);
21. [EE2,EE3,EE4]=returnBitsAfterDecoding(inverseShuffledBits,E2,E3,E4,selecti
    onKey);
22. [decocedImage]=compose(ApproximateImage,E1,EE2,EE3,EE4);
23. [decocedImage]=decrypt(decocedImage,EncryptionKey);
```

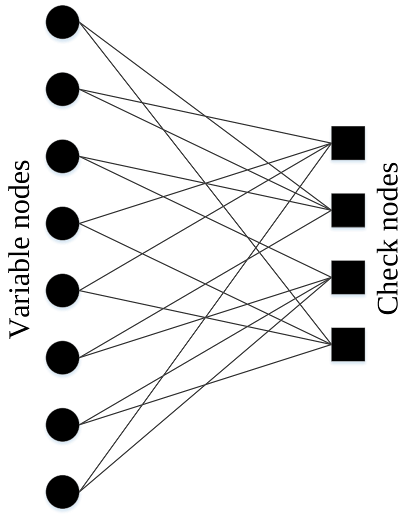
3.2. *H* Matrix Construction Methods

In encoding and compression phase, *H* matrix is used to compress groups of bits to obtain syndrome. In addition, in lossless recovery stage, *H* is used for recovered MSBs. However, the *H* matrix that is used is not exactly specified in [1] nor in its reference [10].Thus, we have generated different *H* matrices using Gallager method [11] and MacKay-Neal method [12].

An LDPC *H* matrix, is a binary matrix contains few number of 1's that are using for error correction. The *H* matrix can be represented via matrix and graphical representation. The graphical representation are used for decoding which will be described in Section 3.5. In Figure 18, an example of *H* matrix with size 4×8 represented by matrix and graphically is given. Figure 18 (a) The *H* matrix, Figure 18 (b) Graphical representation of *H* matrix

$$
H = \begin{bmatrix} 0\,1\,0\,1\,1\,0\,0\,1 \\ 1\,1\,1\,0\,0\,1\,0\,0 \\ 0\,0\,1\,0\,0\,1\,1\,1 \\ 1\,0\,0\,1\,1\,0\,1\,0 \end{bmatrix}
$$

(a)



(b)

Figure 19: Representation of *H* Matrix. (a) *H* Matrix (b) Graphical Representation of *H*. Columns of *H* Matrix are Represented as Variable Nodes in Left Side. Rows of *H* Matrix are Represented as a Check Nodes in Right Side

In the rows of *H* matrix are represented as variable nodes and the columns are represented as a check nodes. The 1's are represented as a connection between variable nodes and check nodes.

The *H* matrix presented by Gallager is regular, that means each column has w_c of 1's and each row has w_r of 1's. To construct *H* matrix, we have to define the number of 1's in each column which defines by w_c and the number of 1's in each rows defines by w_r. Then, the sub-H matrices are constructed based on the number of w_c. For example if w_c=4, then we have 4 sub matrices. Each column in each sub-matrix contains a single 1 and w_r of 1's in each row. The first rows in the first sub-matrix contains w_r successive ones ordered from left to right across the columns then the other sub-matrices are randomly chosen based on the first sub-matrix.

For construction, we determine the number of the columns (n), the number of 1's in each row (w_r) and number of 1's in each column (w_c). Then, to determine the size of H matrix, we have to know the number of the rows (r) using (17)

$$r = n \times (w_c/w_r) \quad (17)$$

For example, when n =20, WC = 3, w_r = 4 then r = (20×3)/4 = 15.

Thus, the H matrix size is r =15 and n= 20.

After that, we have to construct the first sub-matrix. We have to distribute the w_r 1's in each row sequentially. The other sub-matrices are constructed based on the first sub-matrix permuting the rows randomly [22]. We generated another different H matrices using MacKay-Neal [12] method. In this method, the 1's are added at one column at time from left to right. The location of 1's in each column are chosen randomly for rows are not full yet. We determined the number of the rows r and the number of columns (n). For example, r = 9 and n = 12 then the H matrix is

$$H = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

We have used implemented Gallager method to construct H matrices as in Appendix A.6 [22]. This function takes the number of columns n as an input. The number of 1's in columns (w_c) and rows (w_r) are assigned in lines 3 and 4 where w_c=4 and w_r =8. In line 5, number of rows r are determined. First sub matrix are generated in lines 6-12.

The generation of other sub-matrices are generated using permutation from sub-matrix in lines 14 – 23. We generated 10 different sizes of matrices 3 times matrices. In other words, we generated 10 ×3 matrices with different sizes with same ratio *R*=0.5. In Appendix A.6.1 the generated matrices are shown.

We have generated different *H* matrices using MacKay-Neal [23] method implemented using Appendix A.7.

3.3 Sum-Product Decoding Algorithm

In lossless recovery case, decoding algorithm is used to restore the original ones. Using *LLR* in (15) values, the *SGs* which are obtained after the secret data is extracted and *H* matrix, the original bits are restored using Sum-Product Decoding [16] and [20] algorithm. The decoding algorithm is described in Algorithm 14 in Section 2.1.3.3.

In Appendix A.3.5.1 shows the implementation of Sum-product decoding algorithm. In line 2, we defined the number of iteration equal to 15. In line 13, we initialized vector *z* to calculate *LLR* using *q*= 0.1. Then, in lines 20-28, we calculate the check nodes values in matrix N. In lines 34-51, vector *C* which contains variable nodes are connected to each check node. After calculating *C* vector , in lines 52-58, we calculate the inverse tanh using MATLAB function atanh and store values in *B* in line 59.a. using *E*, in lines 63-74, we calculate a vector J that it is decoded from received vector. To check if it’s decoded correct, in line 75, we multiply vector *J* with *H* transpose to get syndrome and comparing with received syndrome. If they are same, then *J* is the decoded vector. Else, recalculating using variable nodes values. In line 82, we declare vector *A* contains values of variable nodes in lines 80-91. In lines 92, *E* matrix is updated using values for vector *A*.

In Appendix A.4, we use same steps in section 3.1 except we define the selection ratio in line 6 when $\alpha \in [0.1, 1.0]$.

3.4 Summary of Chapter 3

In this chapter we present:

1. The implementation of Qian-Zhang method when selection ratio α is fixed equal to 1 and when $\alpha \in [0.1, 1.0]$.

2. The implementation and algorithms of constructing selection key, shuffle key and encryption key.

3. *H* matrix construction methods using Gallager and MacKay-Neal.

4. Sum-product decoding algorithm

Chapter 4

EXPERIMENTAL SETTINGS AND RESULTS

In this chapter, we discuss conducted experiments in [2] using our defined parameters, in addition, we explain extended experiments on PSNR of decoded image, decoding time, embedding capacity and *H* matrix ratio.

4.1 Experimental Settings

Qian-Zhang method [1] (See Chapter 2), encodes grayscale images with size 512×512 using *H* matrix defined in [10], with r =3840 and n = 6336 and cross-over probability[1] $q = 0.1$. Using our defined parameters (selection key K_{SL}, encryption key K_{ENC}, shuffle key K_{SF}, *H* matrix), we conducted same experiments [1] in order to confirm results in [1].

Since, the *H* matrix are not defined clearly neither in [1] nor in its reference [10], we generated different *H* matrices with different sizes and different ratios using Gallager [11] and Mackay-Neal method [12].These experiments include relation between the quality of approximate image (PSNR) and embedding capacity using (10). Moreover, relation between the quality of decoded image (PSNR) and embedding capacity is studied in case of decoding fails. Since *H* matrix is not defined in [1] nor its reference [10], we had to find optimum *H* matrix construction method. We tested by experiments two *H* matrix construction methods, Gallager [11] and MacKay-Neal [12] to find optimum *H* matrix construction method and used it in conducted experiments in [2].

[1] A small probabilty that a most significant bit will be flipped.

We also extended experiments in [2] to find relation between H matrix size and both decoded PSNR and decoding time. Experiments conducted using MATLAB 2016 on a PC equipped with 2 GHz Intel Pentium Dual CPU E2180, 3 GB RAM, and Windows 10. Six of grayscale images with size 512×512 are used to conduct our experiments.

4.2 Comparison of Different H Matrix Construction Methods

We generated three different H matrices using Gallager [11] and Mackey Neal [12] with sizes (r = 64, n = 128) (r = 128, n = 256) and (r = 256, n = 512) (see Appendix B). For each size we generated 3 matrices for each method. In Appendix B.1, shows the parts of H matrix we used in our implementation with size $64\times$ 128. The other H matrices are constructed using implementation in Appendix B.2. We constructed H matrix using Gallager method by determining the number of columns in H matrix as shown below (Appendix B.2, line 3):

HG=Gallager_construction_LDPC (number_of_columns).

We constructed H matrix using MacKay-Neal method by determining the number of rows and columns in H matrix as shown below (Appendix B.2, line 4):

HM=makeLdpc (number_of_rows, number_of_columns, 0, 0, 1).

For each generated matrix six gray scale images have been tested by implementing Qian-Zhang method [1] in Appendix A.3. In Appendix A.3.2 lines 19-21, H matrix is loaded. In Appendix A.3.5 line 64, PSNR of decoded image are calculated for each H matrix. Appendix A.3 line 47 decoding time is calculated using tic and toc MATLAB functions.

In Appendix C.3, the details of experiments are shown. In Appendix C.3.1, PSNR for decoded image using Gallager method are shown for each run in Tables C.3.1.1, C.3.1.2 and C.3.1.3. In Appendix C.3.2, PSNR for decoded image using MacKay-Neal method is shown in Tables C.3.2.1, C.3.2.2 and C.3.2.3. We see from we see that PSNR for decoded image using Gallager method is better than using MacKay-Neal method. In Appendix C.3.3 in Tables (C.3.3.1, C.3.3.2 and C.3.3.3) show decoding time using Gallager method. In appendix C.3.4, decoding time using MacKay-Neal method is shown in Tables C.3.4.1, C.3.4.2 and C.3.4.3. We see that decoding time using Gallager method is less than using MacKay-Neal method.

Average PSNR of decoded image and decoding time is measured to compare between construction methods.

Table 1: Average Decoding Time for Each Image (seconds)

Image / Method	Baboon	Barbara	Lake	Lena	Man	Pepper
Gallager	852.648	775.785	432.138	380.004	446.409	430.336
MacKay-Neal	3830.568	5462.701	3910.733	3970.931	4654.994	4556.044

Table 2: Average PSNR for Each Decoded Image (dB)

Image / Method	Baboon	Barbara	Lake	Lena	Man	Peppers
Gallager	∞	∞	∞	∞	∞	∞
MacKay-Neal	32.937	24.459	26.582	26.957	26.6	26.4

We see from Table 1 that average decoding time for six tested images using Gallager is less than using MacKay-Neal and from Table 2 we see that for six tested images Gallager shows higher average PSNR than MacKay-Neal method.

We conclude that *H* matrix generated by Gallager method shows better performance than MacKay-Neal so we used Gallager in next conducted experiments.

4.3 Relation between PSNR of Approximate Image and Embedding Capacity

Depending on result from Section 5.1, we used Gallager method to generate different H matrices with sizes (r = 42, n = 210), (r = 64, n = 256), (r = 70, n = 210) and (r = 64, n = 128), put there different H matrices with different sizes and different ratios, in order to test relation between approximate PSNR and embedding capacity. Having selection ratio $\alpha \in [0.1, 1.0]$ starting with α =0.1 and incrementing by 0.1 obtain 10 values of α, Appendix A.4, line 5. Embedding capacity is obtained by substituting α values in (11).

In Appendix A.3.4, approximate image is constructed. Line 23 in Appendix A.3.4, PSNR of approximate image is calculated using psnr MATLAB function.

We found that approximate PSNR doesn't depend on embedding capacity. Since, approximate image is constructed using bilinear interpolation regardless the embedded secret data and H matrix size. Values for different embedding capacity is shown in Tables 3-6 as follows.

Table 3: PSNR (dB) of Approximate Image. H Matrix Size (r = 42, n = 210). α is selection ratio

α	Baboon	Barbara	Lake	Lena	Man	Peppers
0.1	24.698	25.749	32.333	34.42	31.983	32.097
0.2	24.698	25.749	32.333	34.42	31.983	32.097
0.3	24.698	25.749	32.333	34.42	31.983	32.097
0.4	24.698	25.749	32.333	34.42	31.983	32.097
0.5	24.698	25.749	32.333	34.42	31.983	32.097
0.6	24.698	25.749	32.333	34.42	31.983	32.097
0.7	24.698	25.749	32.333	34.42	31.983	32.097
0.8	24.698	25.749	32.333	34.42	31.983	32.097
0.9	24.698	25.749	32.333	34.42	31.983	32.097
1	24.698	25.785	32.097	34.432	31.983	32.097

Table 4: PSNR (dB) of Approximate Image. *H* Matrix Size (r = 64, n = 256). α is selection ratio

α	Baboon	Barbara	Lake	Lena	Man	Peppers
0.1	24.698	25.749	32.333	34.42	31.983	32.097
0.2	24.698	25.749	32.333	34.42	31.983	32.097
0.3	24.698	25.749	32.333	34.42	31.983	32.097
0.4	24.698	25.749	32.333	34.42	31.983	32.097
0.5	24.698	25.749	32.333	34.42	31.983	32.097
0.6	24.698	25.749	32.333	34.42	31.983	32.097
0.7	24.698	25.749	32.333	34.42	31.983	32.097
0.8	24.698	25.749	32.333	34.42	31.983	32.097
0.9	24.698	25.749	32.333	34.42	31.983	32.097
1	24.698	25.785	32.097	34.432	31.983	32.097

Table 5: PSNR (dB) of Approximate Image. *H* Matrix Size (r = 70, n = 210). α is selection ratio

α	Baboon	Barbara	Lake	Lena	Man	Peppers
0.1	24.698	25.749	32.333	34.42	31.983	32.097
0.2	24.698	25.749	32.333	34.42	31.983	32.097
0.3	24.698	25.749	32.333	34.42	31.983	32.097
0.4	24.698	25.749	32.333	34.42	31.983	32.097
0.5	24.698	25.749	32.333	34.42	31.983	32.097
0.6	24.698	25.749	32.333	34.42	31.983	32.097
0.7	24.698	25.749	32.333	34.42	31.983	32.097
0.8	24.698	25.749	32.333	34.42	31.983	32.097
0.9	24.698	25.749	32.333	34.42	31.983	32.097
1	24.698	25.785	32.097	34.432	31.983	32.097

Table 6: PSNR (dB) of Approximate Image. *H* Matrix Size (r = 64, n = 128). α is selection ratio

α	Baboon	Barbara	Lake	Lena	Man	Peppers
0.1	24.698	25.749	32.333	34.42	31.983	32.097
0.2	24.698	25.749	32.333	34.42	31.983	32.097
0.3	24.698	25.749	32.333	34.42	31.983	32.097
0.4	24.698	25.749	32.333	34.42	31.983	32.097
0.5	24.698	25.749	32.333	34.42	31.983	32.097
0.6	24.698	25.749	32.333	34.42	31.983	32.097
0.7	24.698	25.749	32.333	34.42	31.983	32.097
0.8	24.698	25.749	32.333	34.42	31.983	32.097
0.9	24.698	25.749	32.333	34.42	31.983	32.097
1	24.698	25.785	32.097	34.432	31.983	32.097

We see from Tables 3-6 that the approximate PSNR is constant with different *H* matrices and different selection ratio α. Hence, approximate PSNR embedding capacity doesn't depend on embedding capacity. Figure 19 shows the relation between PSNR of approximate image and embedding capacity with different selection ratio α.

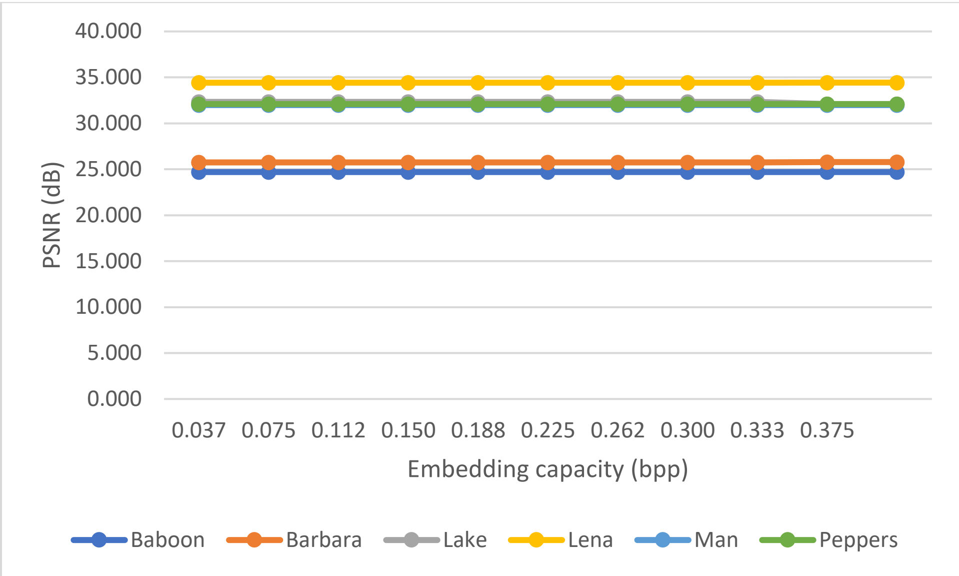


Figure 20: PSNR of Approximate Image of Baboon, Barbara, Lake, Lena, Man and Peppers Images. PSNR of Approximate Image is Constant for Different Selection Ratio α

In Appendix D.1, plots of relation between PSNR of approximate image and embedding capacity are shown for each *H* matrix.

4.4 Relation between PSNR of Decoded Image and Embedding Capacity

Depending on result from Section 5.1, we used Gallager method [11] to generate 3 *H* matrices with size (r = 16, n = 32), see Appendix E.1.

In Appendix E.1, screen shots of PSNR for decoded image for each run are shown in tables for six images. Tables E.1.1, E.1.2, E.1.3 show the PSNR of decoded images in Appendix A.1 Figure A.1.

Then we took the average PSNR of decoded each image (see Appendix E.1). In order to test relation between PSNR of decoded image and embedding capacity, selection ratio $\alpha \in (0,1]$ starting with $\alpha = 0.1$ is incremented by 0.1 to obtain 10 values of α. Embedding capacity is obtained by substituting α values in (10). Table 7 shows the relation between selection ratio α and embedding capacity when ratio R is fixed. According to (10), the embedding capacity increases when selection ratio α increases.

Table 7: Relation between Selection Ratio α and Embedding Capacity (bpp) with Fixed Ratio R

α	Embedding Capacity (bpp)
0.1	0.0375
0.2	0.075
0.3	0.1125
0.4	0.15
0.5	0.1875
0.6	0.225
0.7	0.2625
0.8	0.3
0.9	0.3375
1	0.375

We found that PSNR of decoded image depends on embedding capacity. Increasing embedding capacity by increasing selection ratio α in (10) will lead to decrease decoded PSNR. In addition, decoding time increases when embedding capacity increases.

We see from Table 8 that average PSNR of decoded image decreases when embedding capacity increases.

Table 8: Average PSNR of All Decoded Images (dB)

α	Baboon	Barbara	Lake	Lena	Man	Peppers
0.1	44.663	58.667	45.148	100	51.235	40.544
0.2	43.982	44.264	46.199	60.172	47.883	41.018
0.3	42.202	31.188	46.688	56.281	45.099	40.637
0.4	33.823	29.054	40.192	41.329	40.884	37.362
0.5	32.209	29.037	40.212	41.142	41.044	37.368
0.6	31.209	29.036	38.265	40.072	40.267	36.649
0.7	28.913	29.136	37.851	40.965	39.88	35.222
0.8	28.06	29.009	37.292	40.584	38.974	35.126
0.9	27.866	28.887	37.226	39.821	38.502	35.034
1	27.377	26.795	36.778	38.025	37.555	34.409

Table 9: Average Decoding Time for All Images (seconds). α is selection ratio

α	Baboon	Barbara	Lake	Lena	Man	Peppers
0.1	4.593	3.274	3.647	3.179	3.38	4.603
0.2	6.884	6.703	5.438	4.91	6.494	6.693
0.3	10.142	23.356	7.768	6.941	9.167	8.468
0.4	23.34	34.358	11.355	10.76	12.902	11.803
0.5	28.429	36.363	13.126	12.95	15.469	14.045
0.6	36.205	38.442	16.514	16.216	18.558	16.976
0.7	55.803	40.254	19.972	17.379	20.552	20.857
0.8	65.301	43.227	23.371	20.314	23.112	23.382
0.9	71.274	46.105	26.874	22.66	27.212	26.832
1	81.179	65.951	28.624	25.098	29.615	28.115

Appendix E.2 shows average PSNR of decoded image and average decoding time for each image in each run.

We see from Table 9 that average decoded time increased when embedding capacity increases. Figure 20 shows the relation between decoded PSNR and embedding capacity. As we see, the PSNR of decoded image decreases when the embedding capacity increases.

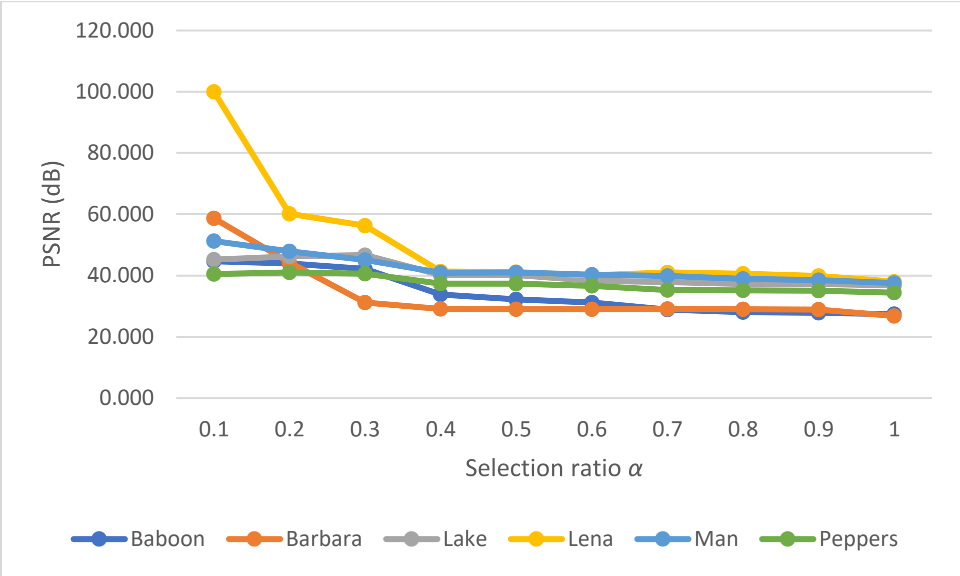


Figure 21: The Relation between PSNR of Decoded Image and α

Figure 21 shows the relation between decoding time and embedding capacity. As we see from the Figure 20 the decoding time increases when the embedding capacity increases.

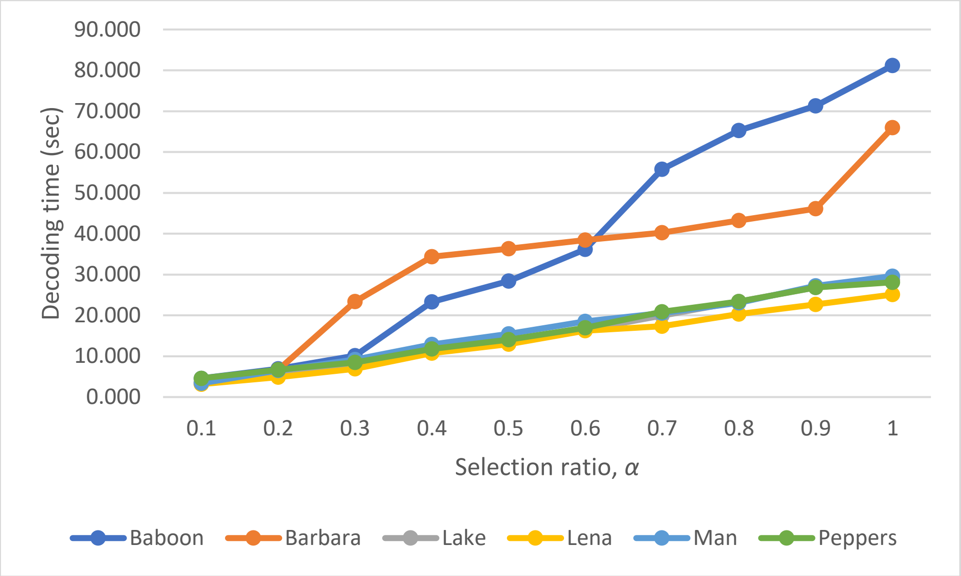


Figure 22: Relation between Decoding Time and α

4.5 Relation between H Matrix Size and PSNR of Decoded Image

We generated different *H* matrices with different sizes (r = 4, n = 8), (r = 8, n = 16), (r = 16, n = 32), (r = 32, n = 64), (r = 64, n = 128), (r = 128, n = 256), (r = 256, n = 512), (r = 512, n = 1024) and (r = 1024, n = 2048) and with same R=0.5 using Gallager method to find the relation between *H* matrix size and decoded PSNR when ratio is fixed. In Appendix F.1, 9 *H* matrices are constructed using Gallager method with different sizes and same ratio R=0.5 for each run. For each size we generate 3 *H* matrices. Appendix F.1.1, shows sample of *H* matrices for first run with sizes 4×8, 8×16. The other sizes and runs are constructed using code in Appendix B.2.

After generation 3 *H* matrices for each size, we took the average for each *H* matrix size for all images. In Appendix F.2, values of PSNR for decoded image for each *H* matrix is described in tables F.2.1, F.2.2, F.2.3for each image.

We found that increasing the size of *H* matrix keeping the ratio R constant will lead to increase the PSNR of decoded image.

Table 10: Average PSNR of Decoded Image Using Different *H* Matrices Sizes with *R*=0.5 (dB)

H matrix size	Baboon	Barbara	Lake	Lena	Man	Peppers
4×8	24.174	25.235	24.705	33.828	31.43	31.57
8×16	26.165	27.364	26.765	36.483	33.872	34.011
16×32	29.813	31.534	30.673	45.218	41.21	40.886
32×64	32.208	34.918	33.563	62.969	44.855	46.232
64×128	40.22	46.531	43.375	82.185	81.458	81.348
128×256	79.239	64.993	72.116	83.714	82.123	84.394
256×512	∞	∞	∞	∞	∞	∞
512×1024	∞	∞	∞	∞	∞	∞
1024×2048	∞	∞	∞	∞	∞	∞

We see from Table 10 that the PSNR of decoded image is getting better when the size of *H* matrix increases.

Figure 22 shows the relation between *H* size and average PSNR of decoded image. As we see, the PSNR for decoded images increases when the size of *H* matrix increase.

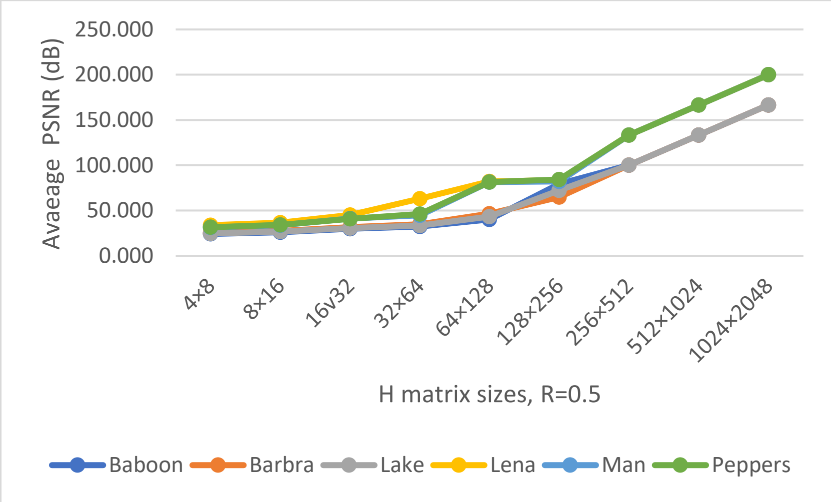


Figure 23: Average PSNR of All Images

In Appendix F.3, PSNR is shown for each decoded image.

4.6 Relation between H Matrix Size and Decoding Time

We generated different H matrices with different sizes (r = 4, n = 8), (r = 8, n = 16), (r = 16, n = 32), (r = 32, n = 64), (r = 64, n = 128), (r = 128, n = 256), (r = 256, n = 512), (r = 512, n = 1024) and (r = 1024, n = 2048) and with same R=0.5 using Gallager method to find the relation between H matrix size and decoding time when ratio is fixed. This relation helps selecting suitable H matrix size for specified decoding time and PSNR of decoded image. In Appendix F.1, 9 H matrices are constructed using Gallager method with different sizes and same ratio R=0.5 for each run. For each size we generate 3 H matrices, then we took the average for each H matrix size.

Sample of H matrices for first run with sizes 4×8, 8×16 are shown in Appendix F.1. The other sizes and runs are constructed using code in Appendix B.2.

We found that increasing the size of H matrix keeping the ratio R constant leads to the increase of the decoding time as shown in Appendix F.4.

Table 11: Average Decoding Time Using Different *H* Matrices Sizes with *R*=0.5 (Seconds)

H matrix size	Baboon	Barbara	Lake	Lena	Man	Peppers
4×8	28.32	23.07	12.21	11.71	12.99	13.25
8×16	61.02	54.39	21.07	19.08	22.52	23.75
16×32	91.48	80.55	31.07	28.96	33.39	35.29
32×64	203.23	181.43	69.67	61.29	73.20	76.37
64×128	323.28	308.09	133.88	126.09	144.62	156.29
128×256	659.09	600.13	300.64	263.37	325.63	315.96
256×512	1226.68	1152.50	796.61	737.73	843.49	890.90
512×1024	2870.23	2481.30	1851.25	1744.11	1952.29	1960.94
1024×2048	6459.64	5703.86	3770.50	3746.50	4020.32	3967.10

We see from Table 11 that the decoding time is getting better when the size of *H* matrix increases.

Figure 23 shows the relation between *H* matrix size and decoding time.

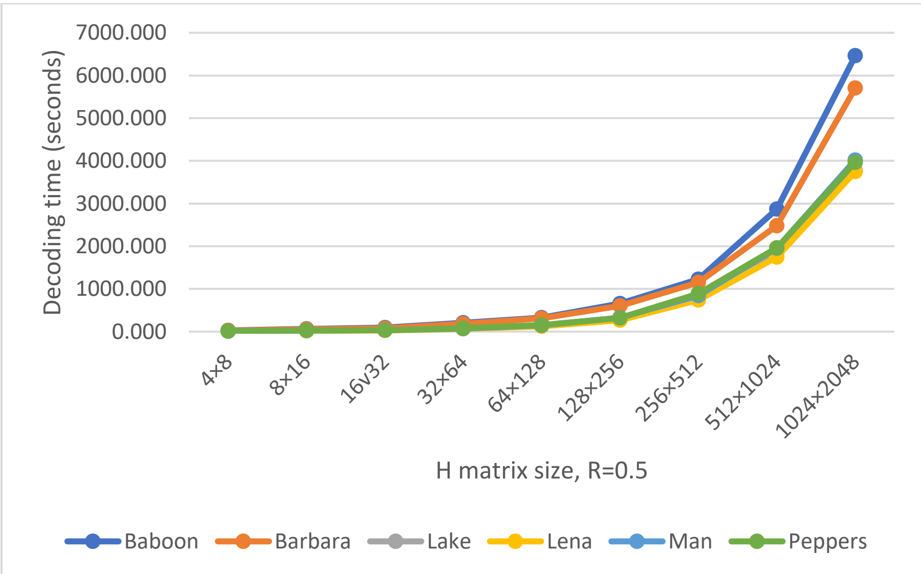


Figure 24: Average Decoding for Images

In Appendix F.5 Figures for decoding time of each decoded image**.**

4.7 Relation between H Matrix Ratio and PSNR of Decoded Image

We generated different *H* matrices (r = 42, n = 210), (r = 64, n = 256), (r = 70, n = 210) and (r = 64, n = 128) with different ratios (0.2, 0.25, 0.33, 0.5) respectively, to find the relation between *H* matrix ratio and PSNR of decoded image.

We found that decreasing ratio i.e. increasing the embedding capacity will lead to decrease PSNR of decoded image. For example, when R=0.33 according to (10), embedding capacity = 0.5 bpp. While, when R= 0.25 embedding capacity = 0.5625 bpp and R=0.2 embedding capacity =0.6 bpp.

Using (9), we expect that embedding capacity increase when the R=r / n decrease and selection ratio α is constant.

Let's consider α =1 and size of encrypted image is 512×512. According to (9) the embedding capacity of each ratio is calculated as shown in Table 12.

Table 12: Embedding Capacity with Different Ratios (bpp)

H matrix ratio	R=0.2	R=0.25	R=0.33	R=0.5
Embedding capacity	0.6	0.5625	0.5	0.375

Figure 24 shows the relation between R = r / n and embedding capacity. When R increases, the embedding capacity decreases. Thus, PSNR of decoded image increase since, the number of bits to be embedded decreases.

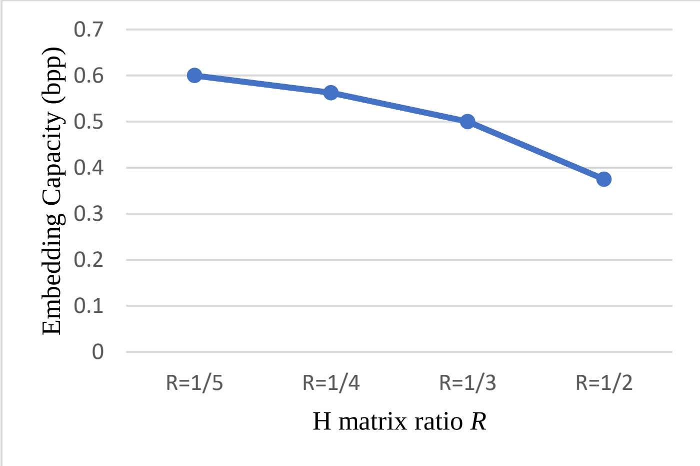


Figure 25: Relation between *R* and Embedding Capacity. When *R* Increases, Embedding Capacity Decreases

Table 13: PSNR of Decoded Images When *R*=0.2, *R*=0.25, *R*=0.33 and *R*=0.5 (dB)

Image	*R*=0.2	*R*=0.25	*R*=0.33	*R*=0.5
Baboon	24.698	25.425	29.341	34.909
Barbara	25.749	27.050	31.285	36.574
Lake	32.333	37.919	40.827	45.121
Lena	34.420	40.727	42.848	46.555
Man	31.983	37.524	39.490	44.374
Peppers	32.097	37.792	40.349	44.044

We see from Table 13 that average PSNR of decoded image increases when embedding capacity decreases. As we expected from (10) in Figure 25.

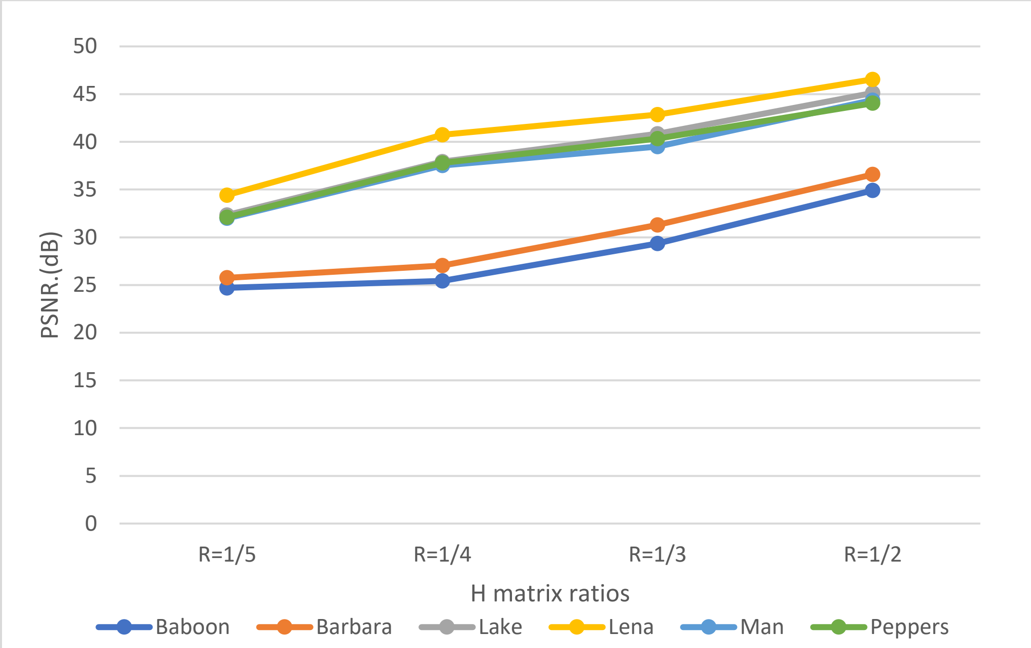


Figure 26: Relation between PSNR of Decoded Images and *H* Ratio, *R*=0.2, *R*=0.25, *R*=0.33 and *R*=0.5

The original image Figure 27(a) is encrypted into Figure 27(b). After encryption, the data hider collect 196608 bits, then with L= $3XY/4$ and α =1, by embedding 98304 bits with embedding capacity 0.375 bpp into encrypted image. Figure 27(c) shows the encrypted image that containing secret data. On the receiver side, secret data are extracted perfectly with error free when the embedding key is known. Figure 27(d) shows the approximate image after construction using the encryption key and the bilinear interpolation. The differences between the original image and the approximate one are shown in Figure 27(e). When the receiver knows the embedding and the encryption keys, the image is recovered perfectly which is shown in Figure 27(f). The other test image results (Baboon, Barbara, Lake, Man, Peppers) are shown in Appendix G.

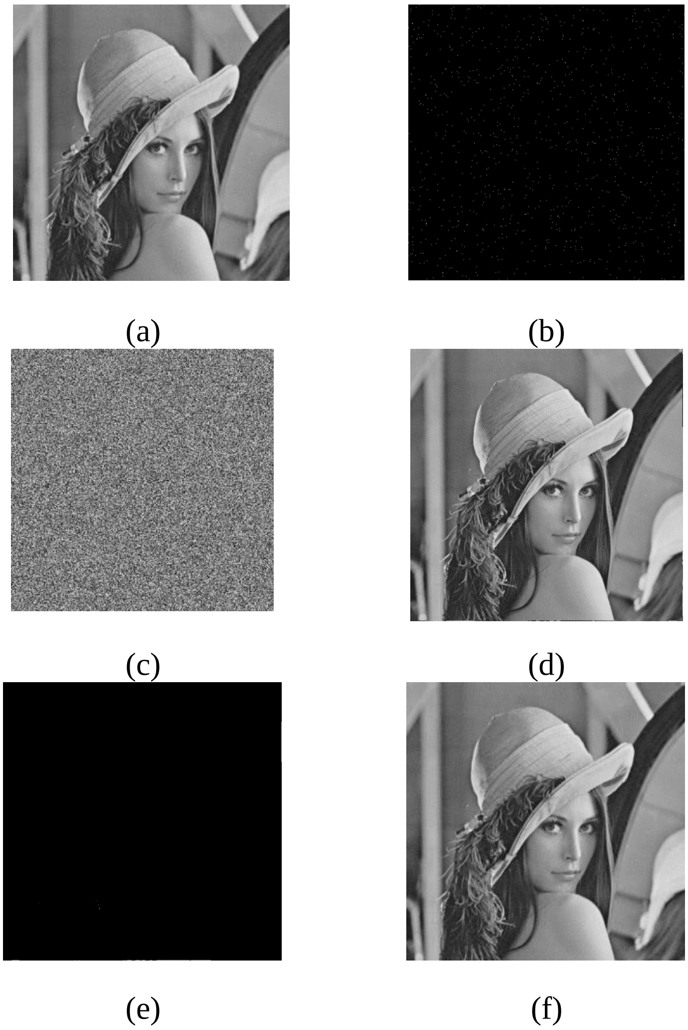


Figure 27: (a) The Original Image *Lena*. (b) The Encrypted Image (Stage 1). (c) Marked Encrypted Image (Stage 2). (d) The Approximate Image (Stage3, Option 2). (e) The Difference between the Original and the Approximate Images. (f) Perfectly Recovered Image (Stage3, Option3)

4.8 Comparison versus Qian-Zhang Scheme Results

By fixing *H* matrix ratio R=0.5 and α =1.0 (see Appendix A.3.2), we achieved maximum payload equal 98304 bits with embedding capacity 0.375 bpp (for both construction method , Gallager MacKay-Neal) which is higher than 77376 payload bits with embedding capacity 0.295 bpp in [1]. The payload bits are extracted perfectly. The secret data is generated randomly according to R which will equal $(3\times512\times512/4)\times(1-1/2)$ (See Appendix A.3.2.10, line 8). Embedding capacity is calculated using (10) (See Appendix A.3.2 line 34). Since the Qian-Zhang scheme [1] modifies the MSBs and using estimation algorithm to construct the approximate image, the quality of the image is fix, regardless the amount of the added payload bits (See Figure 18). Figure 28 shows the comparisons between our implementation results and Qian-Zhang scheme. Figure 28 shows that our results have same behavior as in Qian-Zhang scheme with little difference. In Figure 28(a) our results is less than Qian-Zhang scheme for Lena and Man images while in Figure 28(b) the results are comply for Baboon. For Lake image Figure 28(c) our results is better that Qian-Zhang scheme. The differences between our results and Qian-Zhang results refer to using different source of images which they are different in resolution and grayscale pixels values.

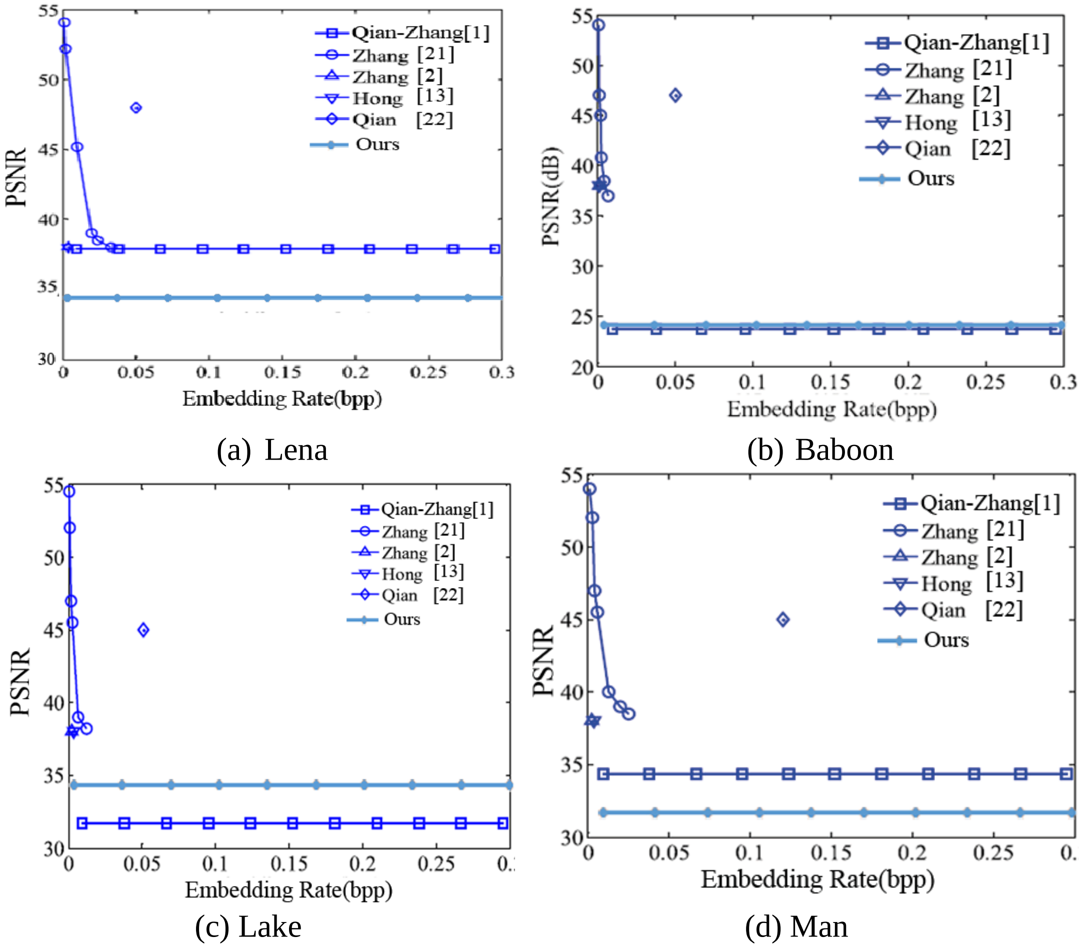


Figure 28: Comparison versus Qian-Zhang Scheme Results using the images. (a) Lena. (b) Baboon (c) Lake. (d) Man

Table 14 shows the comparison between our implementation results and Qian-Zhang results. By using the same size of grayscale images (512 × 512), the total number of bits to be embedded in Qian-Zhang scheme is 77376 while in our implementation the total number of bits to be embedded is 98304. Since, the size of used H matrix in Qian-Zhang scheme is 3840 × 6336 with R=0.61, while in our implementation we used H matrix size 1024 × 2048 with R= 0.5 which leads to compress more according to (11).

Table 14: Comparison versus Qian-Zhang Scheme Results

Qian-Zhang	**Ours**
Grayscale images with size 512 × 512	Grayscale images with size 512 × 512
Total number of collected bits=196608	Total number of collected bits=196608
Selection ratio, α =1.0	Selection ratio, α =1.0
H matrix size : r = 3840 , n = 6336	H matrix size: r = 1024, n = 2048
H matrix ratio, $R = r/n$ = 0.61	H matrix ratio, $R = r/n$ = 0.5
31 groups.	96 groups.
Total number of bits to be embedded =77376	Total number of bits to be embedded =98304
Embedding capacity E_{emb} = 0.2952 bpp	Embedding capacity E_{emb} = 0.375 bpp

4.9 Summary of Chapter 4

In this chapter, we have compared our results with experiments in [2], we found that:

1. PSNR of approximate image is constant when embedding capacity varies (See Figure 18 , Tables 5-8)
2. In the case of decoding fails, the PSNR of decoded image decreases when embedding capacity increases (See Figure 20).
3. We extend our experiments to find the effect of H matrix on decoding time and PSNR of decoded image (See Figure 19, Table 9).
4. By fixing ratio R and increasing H matrix size, both PSNR of decoded image and decoding time increase (See Figure 21, Figure 22).
5. When H matrix ratio R increases the embedding capacity decreases, and PSNR of decoded image increases (See Figure 24).

Chapter 5

CONCLUSION

This thesis aimed to investigate and study Qian-Zhang scheme [1], several fundamental parameters where not clearly specified and shown in [2] such as method of constructing LDPC *H* matrix, selection key K_{SL}, encryption key K_{ENC}, and shuffle key K_{SF}. Our implementation shows that the data extracted perfectly. In this thesis, we studied two *H* matrix construction methods Gallager and MacKay-Neal. We found that matrices constructed by Gallager provide less decoding time with higher PSNR. We studied the effect of *H* matrices size and ratio on the PSNR of decoded images. We found that by fixing the ratio and increasing the size of *H* matrix improves the PSNR of the decoded image. In addition, we studied the effect of *H* matrices size and ratio on the decoding time. We found that by fixing ratio and increasing the size of *H* matrix increases decoding time exponentially (See Figure 22). We obtained after decoding PSNR of 40.629, 41.659 dB for $H_{256\times512}$, $H_{512\times1024}$, respectively while for $H_{1024\times2048}$, the image was recovered perfectly. On the other hand, the time of decoding increases with the matrix size growth: (1426.90, 3668.93, and 5721.153 seconds, respectively). Moreover, increasing *H* matrix ratio $R = r / n$ leads to the decrease of embedding capacity and increase of PSNR of decoded image. Decreasing of *R* (we considered 0.5, 0.33, 0.25, 0.2) leads to the increase of the embedding capacity (0.375, 0.5, 0.5625, and 0.6 bits per pixel (bpp), respectively). In addition, decreasing *R* (0.5, 0.33, 0.25, 0.2) leads to the decrease of the PSNR (122.96, 64.95, 32.186 and 29.252 dB respectively).;.

We constructed LDPC *H* matrix using Gallager method [11], and we defined a selection key K_{SL}, encryption key K_{ENC}, and shuffle key K_{SF}. Then, we implemented Qian-Zhang [1]. We managed to obtain similar PSNR of approximate image results in [1] with little difference, for example in Lena image PSNR value in [1] higher than ours by 3.5 bpp. Since we use different *H* matrix size and method construction in addition, the images are obtained from another source. By extending experiments in [1], we found that the relation between PSNR of approximate image keeps unchanged when the embedding capacity varies. We also found that when decoding fails, the PSNR of decoded image decreases when embedding capacity increases by 0.0375 bpp, since the number of selected bits increase. Our experimental results showed that using *H* matrix constructed by Gallager with ratio R=0.5 leads to better embedding capacity by 1.27% than in Qian-Zhang [1].

Results obtained on PSNR of the decoded image and time dependence on the matrix size may be used for making decisions on Qian-Zhang scheme selection parameters and may be used for choosing suitable *H* matrix size to meet specified decoding time.

REFERENCES

[1] Z. Qian and X. Zhang, "Reversible Data Hiding in Encrypted Images With Distributed Source Encoding," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 26, no. 4, pp. 636–646, 2016.

[2] X. Zhang, "Reversible data hiding in encrypted image," *IEEE Signal Process. Lett.*, vol. 18, no. 4, pp. 255–258, 2011.

[3] D. Slepian and J. Wolf, "Noiseless coding of correlated information sources," *IEEE Trans. Inf. Theory*, vol. 19, no. 4, pp. 471–480, 1973.

[4] C. C. Chang, T. S. Nguyen, and Y. Liu, "A reversible data hiding scheme for image interpolation based on reference matrix," in *2017 5th International Workshop on Biometrics and Forensics (IWBF)*, 2017, pp. 1–6.

[5] X. Xie and C. C. Chang, "Reversible data hiding in encrypted images using reformed JPEG compression," in *2017 5th International Workshop on Biometrics and Forensics (IWBF)*, 2017, pp. 1–5.

[6] Z. Qian, X. Zhang, and G. Feng, "Reversible Data Hiding in Encrypted Images Based on Progressive Recovery," *IEEE Signal Process. Lett.*, vol. 23, no. 11, pp. 1672–1676, 2016.

[7] F. Huang, J. Huang, and Y. Q. Shi, "New Framework for Reversible Data Hiding in Encrypted Domain," *IEEE Trans. Inf. Forensics Secur.*, vol. 11, no. 12, pp.

2777–2789, 2016.

[8] R. Jiang, H. Zhou, W. Zhang, and N. Yu, “Reversible Data Hiding in Encrypted Three-Dimensional Mesh Models,” *IEEE Trans. Multimed.*, vol. 20, no. 1, pp. 55–67, 2018.

[9] M. P. C. Fossorier, M. Mihaljevic, and H. Imai, “Reduced complexity iterative decoding of low-density parity check codes based on belief propagation,” *IEEE Trans. Commun.*, vol. 47, no. 5, pp. 673–680, 1999.

[10] D. Varodayan, A. Aaron, and B. Girod, “Rate-adaptive codes for distributed source coding,” *Signal Processing*, vol. 86, no. 11, pp. 3123–3130, 2006.

[11] R. G. Gallager, “Low-density parity-check codes.,” *IRE Trans. Inf. Theory*, vol. 8, no. 1, pp. 21–28, 1962.

[12] D. J. C. MacKay and R. M. Neal, “Near Shannon limit performance of low density parity check codes,” *Electron. Lett.*, vol. 33, no. 6, pp. 457–458, 1997.

[13] W. Hong, T. S. Chen, and H. Y. Wu, “An Improved Reversible Data Hiding in Encrypted Images Using Side Match,” *IEEE Signal Process. Lett.*, vol. 19, no. 4, pp. 199–202, 2012.

[14] X. Zhang, “Separable Reversible Data Hiding in Encrypted Image,” *IEEE Trans. Inf. Forensics Secur.*, vol. 7, no. 2, pp. 826–832, 2012.

[15] W. Liu, W. Zeng, L. Dong, and Q. Yao, "Efficient Compression of Encrypted Grayscale Images," *IEEE Trans. Image Process.*, vol. 19, no. 4, pp. 1097–1102, 2010.

[16] S. Johnson, *Introducing Low-Density Parity-Check Codes*. 2010.

[17] W. Hong, T. S. Chen, and H. Y. Wu, "An improved reversible data hiding in encrypted images using side match," *IEEE Signal Process. Lett.*, vol. 19, no. 4, pp. 199–202, 2012.

[18] Y.-C. Chen, C.-W. Shiu, and G. Horng, "Encrypted signal-based reversible data hiding with public key cryptosystem," *J. Vis. Commun. Image Represent.*, vol. 25, no. 5, pp. 1164–1170, 2014.

[19] Z. Yin, B. Luo, and W. Hong, "Separable and error-free reversible data hiding in encrypted image with high payload," *Sci. World J.*, vol. 2014, 2014.

[20] A. D. Liveris, Z. Xiong, and C. N. Georghiades, "Compression of binary sources with side information at the decoder using LDPC codes," *IEEE Commun. Lett.*, vol. 6, no. 10, pp. 440–442, 2002.

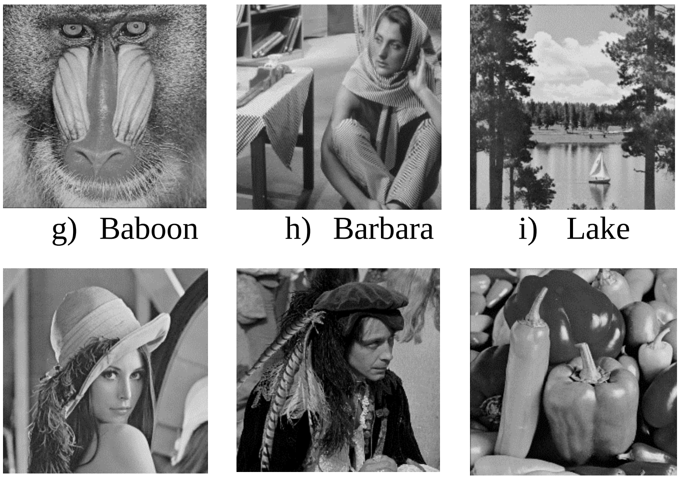
[21] X. Zhang, "Separable reversible data hiding in encrypted image," IEEE Trans. Inf. Forensics Security, vol. 7, no. 2, pp. 826–832, Apr. 2012.

[22] Z. Qian, X. Han, and X. Zhang, “Separable reversible data hiding in encrypted images by *n*-nary histogram modification,” in *Proc. 3rd Int. Conf. Multimedia Technol. (ICMT)*, Guangzhou, China, 2013, pp. 869–876.

APPENDICES

Appendix A: Qian-Zhang scheme implementation

Appendix A.1 Grayscale images used in our experiments



g) Baboon h) Barbara i) Lake

j) Lena k) Man l) Peppers

Figure A.1. Images used in Qian-Zhang scheme implementation

A.2 Images conversion

```
clc;
clear all
P = 'imagesPgm\';
D = dir(fullfile(P,'*.pgm'));
C = cell(size(D));
for k = 1:numel(D)
I = imread(fullfile(P,D(k).name));
C{k} = I(:);
kk=num2str(k);
h='.bmp';
saveN = sprintf('%s%s','images/',kk,h);
imwrite(I,saveN);
```

```
end
```

Appendix A.3 Qian-Zhang Scheme for $\alpha = 1.0$

```
clc;
clear all;
FileName = dir('images3/*.bmp');%read images from folder
nfiles = length(FileName); %get the number of images
DecodedPSNR={'name','HG1','HG2','HG3','HG4','HG5','HG6','HG7','HG8','HG9'};
DecodedTime={'name','HG1','HG2','HG3','HG4','HG5','HG6','HG7','HG8','HG9'};
sizes=[8,16,32,64,128,256,512,1024,2048]; % define the columns of H matrices which is equal to n
selectionSeed=4;
selectionRatio=1.0;
for ii=1:nfiles
imName='';
imName=FileName(ii).name;
rN = sprintf('%s','images3/',imName);
OriginalImage= imread(rN);%read the original image
DecodedPSNR(ii+1,1)={imName};
DecodedTime(ii+1,1)={imName};
[m,n] = size(OriginalImage);
%% encrption
load('EncryptionKey.mat');
[EncryptedImage]=encrypt(OriginalImage,EncryptionKey);
```

```
EncryptedImage=double(EncryptedImage);
saveN = sprintf('%s','EncryptedImages/EncryptedImage_',imName);
imwrite(EncryptedImage,saveN); % store the encrypted image
EncryptedImage=uint8(EncryptedImage);
%% hide data
for j=1:9
n=sizes(j); // get n=number of bits
[Marked_encrypted_image,selectionkey,Shufflekey,L,r,H,syndorm,kgroups]=HideData(EncryptedImage,selectionRatio,selectionSeed,n, imName,j);
%% data extraction
[extractedData]=DataExtraction(Marked_encrypted_image,selectionkey,Shufflekey,L,r,n);
% re=double(secretData)-double(extractedData);
% X = nnz(re);
% non_zero_Ptg=(X/(m*n))*100;
% zero_Ptg=(1-(non_zero_Ptg/100))*100;
% if(zero_Ptg==100)
% fprintf('\n Secert Data is extracted %0.4f \n', zero_Ptg);
% end
%
[ree,cee]=size(extractedData);
ebeee=cee/(512*512);
disp(ebeee);
%% get the ApproximateImage
```

43. [ApproximateImage,PSNR]=DecryptionAndEstimation(Marked_encrypted_image,EncryptionKey,imName,OriginalImage);
44. %% decode and get the original image
45. tic;
46. [RecievedImage, DecodedPSNR]=Recovery(Marked_encrypted_image,selectionkey,Shufflekey,H,L,r,n,EncryptionKey,secretData,syndorm,kgroups,imName,OriginalImage);
47. DecodedTime=toc;
48. GDecodedPSNR(ii+1,j+1)={DecodedPSNR};
49. GDecodedTime(ii+1,j+1)={DecodedTime};
50. end
51. % end for read files
52. end
53. xlswrite('GRun_1.xlsx',GDecodedPSNR,1);
54. xlswrite('GRun_1.xlsx',GDecodedTime,2);

Appendix A.3.1. Stage 1: Image Encryption

1. function [EncryptedImage] = encrypt(OriginalImage,EncryptionKey)
2. [M,N] = size(OriginalImage);
3. OriginalImage=OriginalImage';
4. binary=de2bi(OriginalImage,8,2,'left-msb'); % convert from pixel into binary
5. binaryImage=xor(binary,EncryptionKey); % encrypt using encryption key
6. EncryptedImage=bi2de(binaryImage,'left-msb'); % convert binary into pixel
7. EncryptedImage=reshape(EncryptedImage,M,N); % get encrypted image
8. EncryptedImage=EncryptedImage';
9. end

Appendix A.3.1.1. Generate Encryption Key

```
s_fieldnames = 'EncryptionKey';
a_nums = randi([0 1], 512*512, 8);
% % create the variable containing the values of a_nums
eval([s_fieldnames '=a_nums;']);
% save it in a mat file
save('EncryptionKey',s_fieldnames);
```

Appendix A.3.2. Stage 2: Data Hiding

```
function[MarkedEncryptedImage,selectionkey,shufflekey,L,r,H,syndorm,kgroups]= HideData(encryptedImage,selectionRatio,selectionSeed,n,imName,j)
% 1. decompose
[E1,E2,E3,E4]=decompose(encryptedImage);
[collectedbits]=collectBits(E2,E3,E4);
% create selection key
T=length(collectedbits); % total number of collected bits
L=floor(selectionRatio*T); % number of selected bits
[selectionkey]=createSelectionKey(collectedbits,selectionSeed,selectionRatio);
[selectedBits]=selectbits(collectedbits,selectionkey);
[shufflekey]=generateshuffelkey(selectedBits);
% 4. ShuffleBits
[shuffledbits]=shufflebits(selectedBits,shufflekey);
% 5. createGroups
[kgroups,reminderBits]=createGroups(shuffledbits,n); % return the group to multiply with H matrix
siize=size(kgroups);
```

16. Krows=siize(1,1);

17. Kcols=siize(1,2);

18. nu=int2str(j);

19. Hname=strcat('HGST2_',nu);

20. Hname2=strcat(Hname,'.mat');

21. load(Hname2);

22. % 6. getSyndrom

23. [syndorm]=GetSyndorme(kgroups,H);

24. % 7. embedData

25. [image_after_embedding,r,secretData] = embedData(kgroups,syndorm);

26. % 8. inverse shuffle

27. [inverseSuhffledBits]=inverseshuffle
    (image_after_embedding,reminderBits,shufflekey);

28. %9. return back to MSB with remainder

29. [EE2,EE3,EE4]=returnBits(inverseSuhffledBits,E2,E3,E4,selectionkey);

30. %10. compose into 1 image

31. [MarkedEncryptedImage]=compose(A,E1,EE2,EE3,EE4);

32. saveN = sprintf('%s','MEIimages/MarkedEncryptedImage_',imName);

33. imwrite(MarkedEncryptedImage,saveN);

34. embCap=secretData/(512*512);

35. end

Appendix A.3.2.1 Decompose Encrypted Image

1. function [E1,E2,E3,E4]=decompose(encryptedImage)

2. % to check of n and m is power of 2

3. [m,n] = size(encryptedImage);

```
[f,e] = log2(n);
if f == 0.5000
else
return;
end
[f1,e1] = log2(m);
if f1 == 0.5000
else
return;
end
% end of check
E1 = encryptedImage (1:2:end,1:2:end);  %  E1 odd matrix
E2 = encryptedImage (1:2:end,2:2:end);
E3 = encryptedImage (2:2:end,1:2:end);
E4 = encryptedImage (2:2:end,2:2:end) ; % E4 even matrix
end
```

Appendix A.3.2.2 Collect MSBs From E2,E3,E4

```
function [collectedBits] = collectBits(E2,E3,E4)
 [m,n] = size(E2); % M/2 * N/2
siz=(m) * (n) ;
%from E2
bits2=de2bi(E2,[],2,'left-msb'); % CONVERT THE PIXELS INTO BINARY coluns by couluns
c=bits2(1:siz,1); % get only the MSB from the plane
b2=c'; % convert from column to row
```

```
%from E3
bits3=de2bi(E3,[],2,'left-msb');
c=bits3(1:siz,1);
b3=c';
%form E4
bits4=de2bi(E4,[],2,'left-msb');
c=bits4(1:siz,1);
b4=c';
collectedBits =  horzcat(b2,b3,b4);
end
```

Appendix A.3.2.3 Create Selection Key

```
function[selectionKey,L]=createSelectionKey(collectedbits,selectionSeed,selectionRatio)
selectionSeed=4;
T=length(collectedbits);  % total number of collected bits
L=selectionRatio*T;  % determine the number of bits to be selected according to the selection ratio
rng(selectionSeed);
selectionKey=zeros(1,L);
index=1;
while index <= L
y = floor(randi(T,1,1)); % rand
if(ismember(y,selectionKey)) % this is return the index
continue;
end
```

```
selectionKey(1,index)=y; % here added the selected bits according to the KSL
(we add the index of each selected bit)
index=index+1;
end
end
```

Appendix A.3.2.4 Select Bits Using Selection Key

```
function [selectedBits] = selectbits(collectedbits,selectionKey)
selectedBits=collectedbits((selectionKey));
end
```

Appendix A.3.2.5 Shuffle Key Construction

```
function [ shuffleKey ] = generateshuffelkey(selectedBits)
done=0;
L=length(selectedBits);
selectedPrimes=zeros(1,1);
while done==0
p = primes(L);
x= p(randi(numel(p)));
if((x==1) ||(ismember(x,selectedprimes)) )
    continue;
end
if gcd(x,L)==1
done=1;
x= shuffleKey
else
selectedprimes=x;
```

```
end
end
end
```

Appendix A.3.2.6 Shuffle Bits Using Shuffle Key

```
function [ shuffledbits ] = shufflebits(selectedbits,key)
sizeOfSelectedBits=length(selectedbits);
shuffleRow=(1:sizeOfSelectedBits);
shuffleRow=mod(key*(shuffleRow),sizeOfSelectedBits)+1;
shuffledbits=selectedbits(shuffleRow);
end
```

Appendix A.3.2.7 Create Groups

```
function [ kgroups,arrayrem ] = createGroups(shuffledbits,numberofbits)
% create groups from L bits
L=length(shuffledbits);
k=floor(L/numberofbits); % no. of groups
reminder=mod(L,numberofbits);
arrayrem=zeros(1,reminder);
x=k*numberofbits;
arrayrem(1:end)=shuffledbits(x+1:end);% store the reminder
C=shuffledbits(1:x);
kgroups=reshape(C,numberofbits,k);
kgroups=kgroups.';
end
```

Appendix A.3.2.8 Store Generated Matrices In “.mat” Files

```
clc;
clear all;
n=1024;
HG=Gallager_construction_LDPC(1024);
r=9;n=12;
% HM=makeLdpc(9, 12, 0, 0, 1);
s_fieldnames = 'H';
a_nums=HG;
eval([s_fieldnames '=a_nums;']);
% save it in a mat file
save('HGST2_8',s_fieldnames);
```

Appendix A.3.2.9 Get Syndrom Groups

```
function [synd]=GetSyndrom (kgroups,H)
HT=H.';
kgroups=double(kgroups);
synd=mod((kgroups*(HT)),2);
end
```

Appendix A.3.2.10 Embed Secret Data

```
function [ image_after_embedding,r,Data] = embedData(kgroups,synd)
groups_size=size(kgroups);
synd_size=size(synd);
K=groups_size(1,1); % no of the groups
n=groups_size(1,2); % no of bits in each group
```

6. r=synd_size(1,2); % no of bits in syn group
7. embedding_size=K*(n-r); % number of bits to be embedding
8. Data=randi([0 1],1,embedding_size); %divided these bits in to K groups
9. Data2=reshape(Data,n-r,K);
10. Data2=Data2.';
11. image_after_embedding=zeros(K,n);
12. image_after_embedding(1:end,r+1:end)=Data2(1:end,1:end);
13. image_after_embedding(1:end,1:r)=synd(1:end,1:end);
14. end

Appendix A.3.2.11 Inverse Shuffle Bits

1. function[inverseShuffleBits]=inverseshuffle(embeddedImage,reminder,shuffledkey)
2. sz=size(embeddedImage);
3. sz_row=sz(1,1); % no of rows
4. sz_col=sz(1,2);
5. siz=sz_row*sz_col;
6. embeddedImage=embeddedImage.';
7. B = reshape(embeddedImage,[1 siz]); % convert the groups into row vector
8. % now add the reminder bits into B (row vector);
9. C=horzcat(B,reminder);
10. sizeOfSelectedBits=length(C);
11. index=(1:sizeOfSelectedBits);
12. BB=mod(index*shuffledkey,sizeOfSelectedBits)+1;
13. inverseShuffleBits(BB)=C;

```
end
```

Appendix A.3.2.12 Replace Bits

```
function [EE2,EE3,EE4] =returnBits(inverseSuhffledBits,E2,E3,E4,selectionkey)
collectedbits(selectionkey)=inverseSuhffledBits;
[m2,n2]=size(E2);
siz=(m2) * (n2) ;
bits2=de2bi(E2,[],2,'left-msb'); % CONVERT THE PIXELS INTO BINARY
bits2(1:siz,1)=collectedbits(1,1:siz);
bits3=de2bi(E3,[],2,'left-msb'); % CONVERT THE PIXELS INTO BINARY
bits3(1:siz,1)=collectedbits(1,siz+1:siz*2);
bits4=de2bi(E4,[],2,'left-msb'); % CONVERT THE PIXELS INTO BINARY
bits4(1:siz,1)=collectedbits(1,(siz*2)+1:end);
bits2 = fliplr(bits2);
EE2 =bi2de(bits2);
EE2=reshape(EE2,m2,n2);
bits3 = fliplr(bits3);
EE3 =bi2de(bits3);
EE3=reshape(EE3,m2,n2);
bits4 = fliplr(bits4);
EE4 =bi2de(bits4);
EE4=reshape(EE4,m2,n2);
end
```

Appendix A.3.2.13 Compose Segments

```
1. function [MarkedEncryptedImage] =compose(encryptedImage,E1,EE2,EE3,EE4)
2. MarkedEncryptedImage= encryptedImage;
3. MarkedEncryptedImage(1:2:end,1:2:end)=E1;  %  E1 odd matrix
4. MarkedEncryptedImage(1:2:end,2:2:end)= EE2;
5. MarkedEncryptedImage(2:2:end,1:2:end)=EE3;
6. MarkedEncryptedImage(2:2:end,2:2:end)=EE4 ; % E4 even matrix
7. End
```

Appendix A.3.3 Data Extraction

```
1. function
   [extractedData]=DataExtraction(MarkedEncryptedImage,selectionKey,shuffleKe
   y,L,r,n)
2. [V1,V2,V3,V4]=decompose(A); % same as in (4);
3. [collectedbits]=collectBits(V2,V3,V4);
4.  [selectedBits]=SelectBitsUsingSelectionKey(collectedbits,L,selectionKey);
5. [shuffledbits]=shufflebits(selectedBits,shuffleKey);
6. [kgroups,reminderBits]=createGroups(shuffledbits,n);
7.  [extractedData]=extractData(kgroups,n,r);
8. end
```

Appendix A.3.3.1 Extract Data from Marked Encrypted Image

```
1. function [ extractedBits2,extractedSynd ] = extractData(kgroups,n,r)
2. [row,c]=size(kgroups);
3. k=row;
4. x=k*(n-r);
```

5. extractedSynd=kgroups(1:k,1:r);

6. extractedBits=kgroups(1:k,r+1:end);% i put -1

7. extractedBits=extractedBits.';

8. extractedBits2=reshape(extractedBits,1,x);

9. end

Appendix A.3.4 Approximate Image Reconstruction

1. function [approximateImage,PSNR]= DecryptionAndEstimation (MarkedEncryptedImage,EncryptionKey,imName,original)

2. [DecryptedImage]=decrypt(MarkedEncryptedImage,EncryptionKey);

3. Marked_image=DecryptedImage;

4. [m,n] = size(Marked_image);

5. Marked_image=double(Marked_image);

6. [A1,A2,A3,A4]=decompose(Marked_image);

7. [B]=interplation(A1,Marked_image);

8. [B1,B2,B3,B4]=decompose(B);

9. [BB1] = calculate_approximate_image(A1, B1);

10. [BB2]=calculate_approximate_image(A2, B2);

11. [BB3]=calculate_approximate_image(A3, B3);

12. [BB4]=calculate_approximate_image(A4, B4);

13. [approximateImage] =compose(Marked_image,BB1,BB2,BB3,BB4);

14. approximateImage = uint8(approximateImage);

15. original=uint8(original);

16. evaluate=uint8(original)-uint8(approximateImage);

17. [rows,cols] = find(evaluate);

```
indeces=horzcat(rows,cols);
X = nnz(evaluate);
non_zero_Ptg=(X/(m*n))*100;
zero_Ptg=(1-(non_zero_Ptg/100))*100;
fprintf('\n percentage for appro. %0.4f \n', zero_Ptg);
PSNR=psnr(approximateImage,original);
fprintf('\n The PSNR value for approximate Image is %0.4f \n', PSNR);
saveN = sprintf('%s','ApproximateImages/ApproximateImage_',imName);
imwrite(approximateImage,saveN);
saveN = sprintf('%s','DifferenceImages/diffDecoded_',imName);
imwrite(evaluate,saveN);
end
```

Appendix A.3.4.1 Bilinear Interpolation

```
function [ B ] = interplation( E1,A )
[m,n] = size(A);
[X,Y] = meshgrid(1:256,1:256);%//revise size as variable
E1=double(E1);
[X2,Y2] = meshgrid(1:0.5:256.5,1:0.5:256.5); %// Define expanded grid of points
B = interp2(X,Y,E1,X2,Y2,'linear');
B(512,1:511)=interp1(1:512,B(1:511,1:511),512,'linear','extrap');
B(1:512,512)=interp1(1:512,B(1:512,1:512),512,'linear','extrap');
B=round(B);
end
```

Appendix A.3.4.2 Calculate Approximate Image

```
function [ approximateImage ] = calculate_approximate_image( A, B )
[m,n] = size(A);
approximateImage=zeros(m,n);
for i=1:m
for j=1:n
if (abs(128+mod(A(i,j),128)-B(i,j)) < abs(mod(A(i,j),128)-B(i,j)))
approximateImage(i,j)=128+mod(A(i,j),128);
else
approximateImage(i,j)=mod(A(i,j),128);
end
end
end
end
```

Appendix A.3.5 Lossless Recovery

```
function [RecievedImage,DecodedPSNR] =Recovery(Marked_encrypted_image,selectionKey,Shufflekey,H,L,r,numberofbits,EncryptionKey,secertData,syndorm,kgroupsOriginal,imName,OriginalImage)
A=Marked_encrypted_image;
[M,N]=size(A);
OriginalImage=uint8(OriginalImage);
[extractedData]=DataExtraction(Marked_encrypted_image,selectionKey,Shufflekey,L,r,numberofbits);
```

6. %% DataExtraction---->DONE
7. [E1,E2,E3,E4]=decompose(A);
8. [collectedbits]=collectBits(E2,E3,E4);
9. [selectedBits]=selectbits(collectedbits,selectionKey);
10. [shuffledbits]=shufflebits(selectedBits,Shufflekey);
11. [kgroups,reminderBits]=createGroups(shuffledbits,numberofbits);
12. [compressedData,compressedGroup]=GetCompressedData(kgroups,numberofbits,r);
13. re=double(syndorm)-double(compressedGroup); % compressedGroup = syndrom
14. [m,n]=size(compressedGroup);
15. X = nnz(re);
16. non_zero_Ptg=(X/(m*n))*100;
17. zero_Ptg=(1-(non_zero_Ptg/100))*100;
18. if(zero_Ptg==100)
19. % fprintf('\n syndrome is extracted in Recovery Stage %0.4f \n', zero_Ptg);
20. end
21. [ApproximateImage,ApproPSNR]=DecryptionAndEstimation(A,EncryptionKey, imName,OriginalImage);% get the approximate Image
22. [EncryptedApproximateImage]=encrypt(ApproximateImage,EncryptionKey);% encrypt the approximate Image
23. [E1,E2,E3,E4]=decompose(EncryptedApproximateImage);
24. n=numberofbits;
25. [collectedbits]=collectBits(E2,E3,E4);
26. [selectedBits]=selectbits(collectedbits,selectionKey);
27. [shuffledbits]=shufflebits(selectedBits,Shufflekey);

```
[kgroupsappro,reminderBits]=createGroups(shuffledbits,n);
[krows,kcols]=size(kgroupsappro);
diff=double(kgroupsOriginal)-double(kgroupsappro);
[rows1,cols] = find(diff);
indeces=horzcat(rows1,cols);
len1=length(rows1);
[rrr,ccc]=size(indeces);
fprintf('\n the differences between the original and approx. %0.4f \n', rrr);
C=unique(indeces);
X = nnz(diff);
decoded=zeros(1,1);
[r,c]=size(kgroupsOriginal);
tic;
for i=1:r
[decodedString]=decodeStatisticsOriginal(compressedGroup(i,1:end),kgroupsappro(i,1:end),H);
decoded(i,1:numberofbits)=decodedString;
end
tDecoded=toc;
diff2=double(kgroupsOriginal)-double(decoded);
[rows2,cols] = find(diff2);
indeces2=horzcat(rows2,cols);
len2=length(rows2);
[rrr,ccc]=size(indeces2);
fprintf('\n the differences after decoding. %0.4f \n', rrr);
```

```
C2=unique(indeces2);
X2 = nnz(diff2);
%% reshuffle the decoded bits
decoded=uint8(decoded);
[inverseShuffledBits]=inverseshuffle (decoded,reminderBits,Shufflekey );
[E1,E2,E3,E4]=decompose(EncryptedApproximateImage);
[EE2,EE3,EE4]=returnBitsAfterDecoding(inverseShuffledBits,E2,E3,E4,selectionKey);
[decocedImage]=compose(ApproximateImage,E1,EE2,EE3,EE4);
[decocedImage]=decrypt(decocedImage,EncryptionKey);
decocedImage=uint8(decocedImage);
saveN = sprintf('%s','DecodedImages/decoced_',imName);
imwrite(decocedImage,saveN);
DecodedPSNR=psnr(uint8(decocedImage),OriginalImage);
fprintf('\n PSNR after decoding %0.4f \n', DecodedPSNR);
diffDecoded=uint8(OriginalImage)-uint8(decocedImage);
X = nnz(diffDecoded);
non_zero_Ptg=(X/(m*n))*100;
zero_Ptg=(1-(non_zero_Ptg/100))*100;
fprintf('\n percentage after decoding %0.4f \n', zero_Ptg);
saveN = sprintf('%s','DifferenceImages/diffDecoded_',imName);
imwrite(diffDecoded,saveN);
RecievedImage=decocedImage;
end
```

Appendix A.3.5.1 Sum-Product Decoding

```
function [z]= decodeStatisticsOriginal(synd,U,H)
Itermax=15;
synd=double(synd);
y=double(y);
HT=H.';
yy=0;
q=0.1;
N=zeros(size(H));
E=zeros(size(H));
r=zeros(size(y));
[rows,cols]=size(r);
[m,n]=size(H);
%Initialization z
for i=1:cols
ll=log((1-q)/q);
z(i)=(1-(2*U(i)))*(ll);
end
%Initialization N
Iter=1;
for i=1:n
for j=1:m
if(H(j,i)==0)
continue;
end
```

```
N(j,i)=z(i);
end
end
%% processing at check nodes
zz=1;
L=zeros(1,n);
z=zeros(1,n);
while(Iter<=Itermax)
%Check messages
for j=1:m
%create C vector (to check variable nodes each check node is connected)
% n : size of cols in H
x=1;
for k=1:n
if(H(j,k)==0)
continue;
end
C(x)=k;
x=x+1;
end
len=length(C);
%
for i=1:n
cc=ismember(i,C);
if(cc==0)
```

```
50. continue;
51. end
52. % move over C and calculate tanh
53. for t=1:len
54. if(i==C(t))
55. continue;
56. end
57. zz=tanh(N(j,C(t))/2)*zz;
58. end
59. vv =atanh((1-(2*synd(j)))*zz)*2;
        B(j,i)=vv;
60. zz=1;
61. end
62. end
63. for i=1:n
64. L(i)=z(i);
65. r2=0;
66. for j=1:m
67. r2=B(j,i)+r2;
68. end
69. L(i)=r2+L(i);
70. if(L(i)<0)
        J(i)=1;
71. end
72. if(L(i)>=0)
```

J(i)=0;

73. end

74. end

75. syn=mod((J*(HT)),2);

76. if(syn==synd)

77. break; %%finish

78. else

79. [rows,co]=size(H);

80. for i=1:n

81. x2=1; % just index

82. A=zeros(1,1);

83. for k=1:rows

84. if(H(k,i)==0)

85. continue;

86. end

    a. A(x2)=k;

    b. x2=x2+1;

87. end

88. len=length(A);

89. for j=1:rows

90. cc=ismember(j,A);

91. if(cc==1)

xx=0;

92. for jj=1:len

93. if(j==A(jj))

```
94. continue;
95. end
96. xx=B(A(jj),i)+ xx;
97. end
        N(j,i)=xx+z(i);
98. end
99. end
100.    end
101.    end
102.    Iter=Iter+1;
103.    end
104.    end
105.    end
```

Appendix A.4 Qian-Zhang for $\alpha \in [0.1, 1.0]$

```
1. clc;
2.  clear all;
3.  DecodedPSNR={'na','0.1','0.2','0.3','0.4','0.5','0.6','0.7','0.8','0.9','1.0'};
4.  Time={'na','0.1','0.2','0.3','0.4','0.5','0.6','0.7','0.8','0.9','1.0'};
5.  EmbeddingCapacity={'na','0.1','0.2','0.3','0.4','0.5','0.6','0.7','0.8','0.9','1.0'};
6.  selectionRatio=[0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0];
7.  FileName = dir('images3/*.bmp');
8.  nfiles = length(FileName);    % Number of files found
9.  for ii=1:nfiles
10. imName='';
11. imName=FileName(ii).name;
```

```
rN = sprintf('%s','images3/',imName);
OriginalImage= imread(rN);
DecodedPSNR(ii+1,1)={imName};
Time(ii+1,1)={imName};
EmbedddingCapacity(ii+1,1)={imName};
[m,n] = size(OriginalImage);
[m,n] = size(OriginalImage);
seed=4;
numberofbits=32;
%% encrption
load('EncryptionKey.mat');
[EncryptedImage]=encrypt(OriginalImage,EncryptionKey );
EncryptedImage=double(EncryptedImage);
saveN = sprintf('%s','images/EncryptedImage',imName);
imwrite(EncryptedImage,saveN);
EncryptedImage=uint8(EncryptedImage);
%% hide data
for j=1:10
selectionRatio=selection(j);
[Marked_encrypted_image,selectionkey,Shufflekey,L,r,H,syndorm,kgroups,selectionkey2,secretData,collectedbitsF,selectedBits,shuffledbits]=HideData(EncryptedImage,selectionRatio,seed,numberofbits, imName,EncryptionKey);
%% data extraction
[extractedData]=DataExtraction(Marked_encrypted_image,selectionkey2,Shufflekey,L,r,numberofbits);
```

```
re=double(secretData)-double(extractedData);
X = nnz(re);
non_zero_Ptg=(X/(m*n))*100;
zero_Ptg=(1-(non_zero_Ptg/100))*100;
if(zero_Ptg==100)
fprintf('\n Secert Data is extracted %0.4f \n', zero_Ptg);
end
%% get the ApproximateImage
[ApproximateImage,zero_Ptg,ApproPSNR]=DecryptionAndEstimation(Marked_encrypted_image,EncryptionKey,imName,OriginalImage);
[rr,cc]=size(secretData);
b3=cc/(512*512);
tic;
[RecievedImage,PSNR]=Recovery(Marked_encrypted_image,selectionkey,Shufflekey,H,L,r,numberofbits,EncryptionKey,secretData,syndorm,kgroups,imName,selectionkey2,OriginalImage,collectedbitsF,b3,selectedBits,shuffledbits);
t=toc;
Time(ii+1,j+1)={t};
DecodedPSNR(ii+1,j+1)={PSNR};
EmbedddingCapacity(ii+1,j+1)={b3};
end
end
xlswrite('results.xlsx',EmbedddingCapacity,'Sheet1');
xlswrite('results.xlsx',ApproPSNR,'Sheet2');
```

Appendix A.3.1 Encryption key.

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\ & & & & \vdots & & & \\ & & & & \vdots & & & \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \end{bmatrix}$$

Appendix A.4.1 Selection keys depending on α .

$$\alpha = 0.1 \rightarrow K_{SL} = [190126 \quad 107591 \quad 191238 \quad 140539 \quad \cdots \quad \cdots \quad 82608 \quad 158627 \quad 153605 \quad 155714]$$

$$\alpha = 0.2 \rightarrow K_{SL} = [190126 \quad 107591 \quad 191238 \quad 140539 \quad \cdots \quad \cdots \quad 11421 \quad 124520 \quad 132931 \quad 85635]$$

$$\alpha = 0.3 \rightarrow K_{SL} = [190126 \quad 107591 \quad 191238 \quad 140539 \quad \cdots \quad \cdots \quad 185059 \quad 18782 \quad 157093 \quad 82356]$$

$$\alpha = 0.4 \rightarrow K_{SL} = [190126 \quad 107591 \quad 191238 \quad 140539 \quad \cdots \quad \cdots \quad 104635 \quad 173429 \quad 82457 \quad 182720]$$

$$\alpha = 0.5 \rightarrow K_{SL} = [190126 \quad 107591 \quad 191238 \quad 140539 \quad \cdots \quad \cdots \quad 76780 \quad 20130 \quad 193339 \quad 160105]$$

$$\alpha = 0.6 \rightarrow K_{SL} = \begin{bmatrix} 190126 & 107591 & 191238 & 140539 & \cdots & \cdots & 114946 & 159016 & 75903 & 172242 \end{bmatrix}$$

$$\alpha = 0.7 \rightarrow K_{SL} = \begin{bmatrix} 190126 & 107591 & 191238 & 140539 & \cdots & \cdots & 154162 & 101616 & 186494 & 165151 \end{bmatrix}$$

$$\alpha = 0.8 \rightarrow K_{SL} = \begin{bmatrix} 190126 & 107591 & 191238 & 140539 & \cdots & \cdots & 179383 & 33854 & 18658 & 115188 \end{bmatrix}$$

$$\alpha = 0.9 \rightarrow K_{SL} = \begin{bmatrix} 190126 & 107591 & 191238 & 140539 & \cdots & \cdots & 33464 & 121075 & 134712 & 122549 \end{bmatrix}$$

$$\alpha = 1.0 \rightarrow K_{SL} = \begin{bmatrix} 190126 & 107591 & 191238 & 140539 & \cdots & \cdots & 24554 & 15636 & 37396 & 13761 \end{bmatrix}$$

Appendix B. Different H Matrix Construction Methods

Appendix B.1. H matrices constructed using Gallager method

Appendix B.1.1: H matrix sample constructed for run 1 using Gallager method

H matrix with size 64 $\times$ 128

$$\begin{bmatrix}
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & & & & & & & & & & \vdots & & & & & & & & & & & \\
 & & & & & & & & & & \vdots & & & & & & & & & & & \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}$$

Appendix B.2. code for constructing H matrices using Gallager and MacKay-Neal implemented code.

```
clc;
clear all;
HG=Gallager_construction_LDPC(1024);
HM=makeLdpc(9, 12, 0, 0, 1);
s_fieldnames = 'H';
a_nums=HG;
eval([s_fieldnames '=a_nums;']);
% save it in a mat file
save('HGST4_8',s_fieldnames);
```

Wconstructed all *H* matrices using the above code.

We constructed *H* matrix using Gallager method by determining the number of columns in *H* matrix.

```
HG=Gallager_construction_LDPC(number_of_columns);
```

We constructed *H* matrix using MacKay-Neal method by determining the number of rows and columns in *H* matrix.

```
HM=makeLdpc(number_of_rows, number_of_columns, 0, 0, 1);
```

Appendix C.Comparison of Different H Matrix Construction Methods Results

Appendix C.3. Screenshots for Each Run (Section 4.2 results)

Appendix C.3.1 PSNR for decoded image using Gallager method

These results are taken from Appendix A.3 line 48 and the H matrices are used from Appendix B.1

Table C.3.1.1. PSNR for decoded images in run 1.

Run 1			
image	64×128	128×256	256×512
'Baboon.bmp'	34.90861	37.71688	Inf
'Barbara.bmp'	36.57365	40.82702	Inf
'Lake.bmp'	45.1205	48.71072	Inf
'Lena.bmp'	46.55473	51.1411	Inf
'Man.bmp'	44.37417	46.36989	Inf
'Peppers.bmp'	44.04416	53.1823	Inf

Table C.3.1.2 . PSNR for decoded images in run 2.

Run 2			
image	64×128	128×256	256×512
'Baboon.bmp'	42.1102	Inf	Inf
'Barbara.bmp'	50.62958	Inf	Inf
'Lake.bmp'	Inf	Inf	Inf
'Lena.bmp'	Inf	Inf	Inf
'Man.bmp'	Inf	Inf	Inf
'Peppers.bmp'	Inf	Inf	Inf

Table C.3.1.2 . PSNR for decoded images in run3.

Run 3			
image	64×128	128×256	256×512
'Baboon.bmp'	43.63988	Inf	Inf
'Barbara.bmp'	52.39049	54.1514	Inf
'Lake.bmp'	Inf	Inf	Inf
'Lena.bmp'	Inf	Inf	Inf
'Man.bmp'	Inf	Inf	Inf
'Peppers.bmp'	Inf	Inf	Inf

Appendix C.3.2 PSNR for decoded image using MacKay-Neal method

These results are taken from Appendix A.3 line 48 and the H matrices are used from Appendix B.2.

Table C.3.2.1. PSNR for decoded image in run 1.

Run 1			
image	64×128	128×256	256×512
'Baboon.bmp'	23.22244	23.19013	25.39934
'Barbara.bmp'	23.96336	23.70307	25.70974
'Lake.bmp'	26.95395	25.96255	26.82948
'Lena.bmp'	27.31193	26.16662	27.39133
'Man.bmp'	26.76558	25.85033	27.18347
'Peppers.bmp'	26.7537	25.80561	26.64054

Table C.3.2.2 .PSNR for decoded image in run 2

Run 2			
image	64×128	128×256	256×512
'Baboon.bmp'	42.1102	Inf	Inf
'Barbara.bmp'	50.62958	Inf	Inf
'Lake.bmp'	Inf	Inf	Inf
'Lena.bmp'	Inf	Inf	Inf
'Man.bmp'	Inf	Inf	Inf
'Peppers.bmp'	Inf	Inf	Inf

Table C.3.2.3. PSNR for decoded image in run 3

Run 3			
image	64×128	128×256	256×512
'Baboon.bmp'	36.17527	39.83777	41.65942
'Barbara.bmp'	37.99716	41.36387	43.35959
'Lake.bmp'	45.54802	49.38019	51.72102
'Lena.bmp'	49.38019	54.1514	55.40079
'Man.bmp'	47.61928	48.71072	51.1411
'Peppers.bmp'	45.85837	51.1411	50.172

Appendix C.3.3 Decoding time using Gallager method

These results are taken from Appendix A.3 line 49 and the H matrices are used from Appendix B.1

Table C.3.3.1. Decoding time in run 1

Run 1			
image	64×128	128×256	256×512
'Baboon.bmp'	422.688	867.914	1267.343
'Barbara.bmp'	406.9153	716.45	1203.991
'Lake.bmp'	143.7532	278.65	874.0114
'Lena.bmp'	134.4695	222.5902	782.9538
'Man.bmp'	146.7786	300.4301	892.0185
'Peppers.bmp'	156.7987	221.8305	912.3776

Table C.3.3.2. Decoding time in run 2

Run 2			
image	64×128	128×256	256×512
'Baboon.bmp'	246.3277	512.4624	1104.598
'Barbara.bmp'	263.2507	538.3843	1114.911
'Lake.bmp'	149.1373	367.0309	879.5724
'Lena.bmp'	135.0534	311.2852	793.9917
'Man.bmp'	159.2285	372.3432	899.5021
'Peppers.bmp'	158.9159	367.9129	888.3298

Table C.3.3.3. Decoding time in run 3

Run 3			
image	64×128	128×256	256×512
'Baboon.bmp'	300.8356	596.8973	1308.089
'Barbara.bmp'	254.1145	545.5668	1138.593
'Lake.bmp'	144.4779	339.4831	814.0754
'Lena.bmp'	108.7579	256.2376	636.2363
'Man.bmp'	127.8655	304.1008	738.9555
'Peppers.bmp'	153.1474	358.1463	871.9793

Appendix C.3.4 Decoding time using MacKay-Neal method

These results are taken from Appendix A.3 line 49 and the H matrices are used from Appendix B.2

Table C.3.4.1. Decoding time in run 1

Run 1			
image	64×128	128×256	256×512
'Baboon.bmp'	1808.244	3345.74	6337.719
'Barbara.bmp'	2413.731	4718.708	9255.665
'Lake.bmp'	1971.246	3593.738	6167.216
'Lena.bmp'	1639.259	3665.876	6607.658
'Man.bmp'	2147.971	4295.927	7521.084
'Peppers.bmp'	2052.906	4100.582	7514.644

Table C.3.4.2. Decoding time in run 2

Run 2			
image	64×128	128×256	256×512
'Baboon.bmp'	481.1471	1359.936	1694.186
'Barbara.bmp'	333.3271	940.5488	1075.772
'Lake.bmp'	135.9766	329.4257	341.5015
'Lena.bmp'	109.4876	294.4683	300.5307
'Man.bmp'	127.544	314.1997	368.9872
'Peppers.bmp'	133.884	334.1103	356.4929

Table C.3.4.3. Decoding time in run 3

Run 3			
image	64×128	128×256	256×512
'Baboon.bmp'	386.0721	840.524	2208.247
'Barbara.bmp'	270.3143	558.8403	1334.854
'Lake.bmp'	118.1959	205.7858	496.1075
'Lena.bmp'	97.66124	178.6593	388.7536
'Man.bmp'	110.0446	239.7486	522.3316
'Peppers.bmp'	121.4274	215.7534	573.9224

Appendix D. Relation between PSNR of Approximate Image and Embedding Capacity.

Appendix D.1. Figures of relation between PSNR pf approximate image and embedding capacity

These results obtained from Appendix A.4 line 54 and drew in Excel.

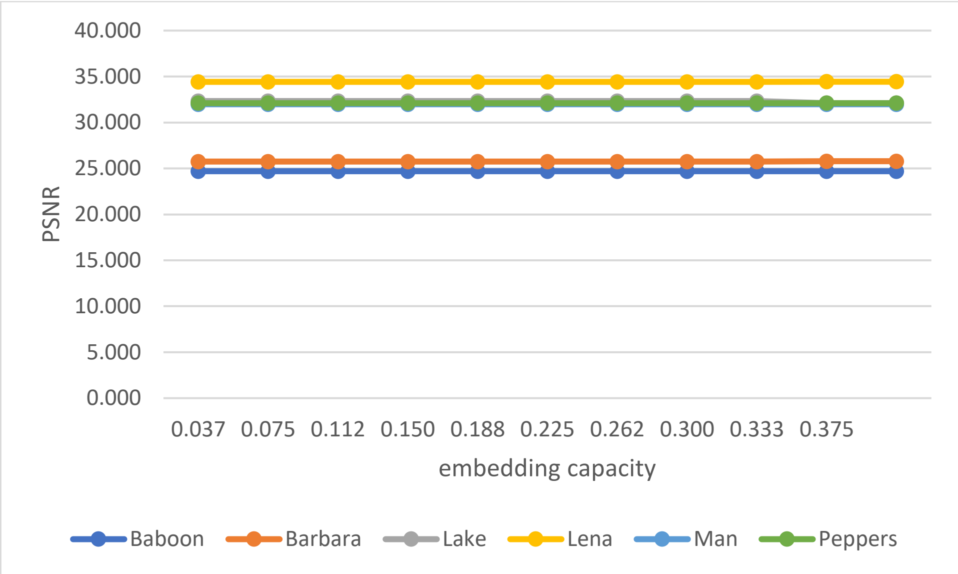


Figure D.1.1. PSNR of approximate image of Baboon, Barbara, Lake, Lena, Man and Peppers images with H matrix size 42×210. PSNR of approximate image is constant with different selection ratio

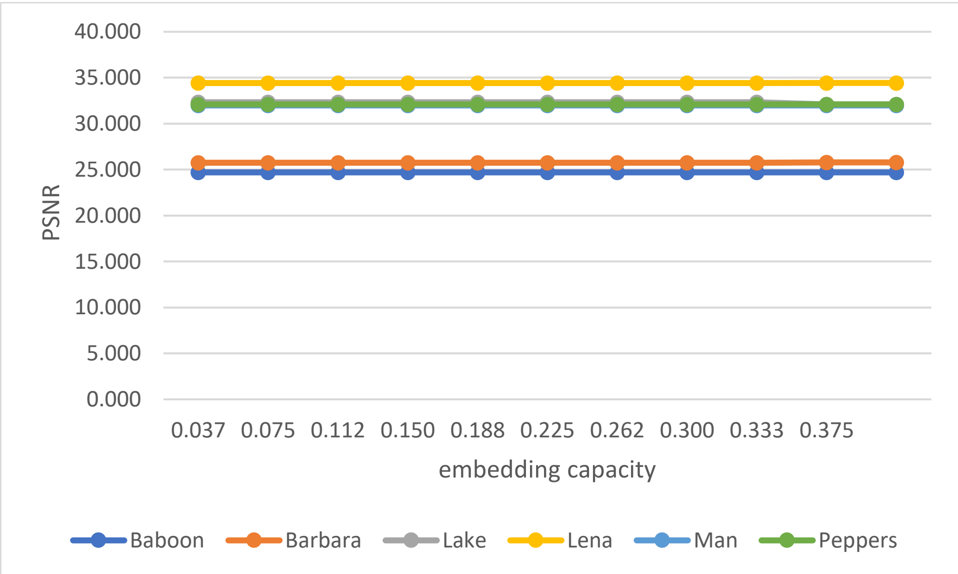


Figure D.1.2. PSNR of approximate image of Baboon, Barbara, Lake, Lena, Man and Peppers images with H matrix size 64×256. PSNR of approximate image is constant with different selection ratio α .

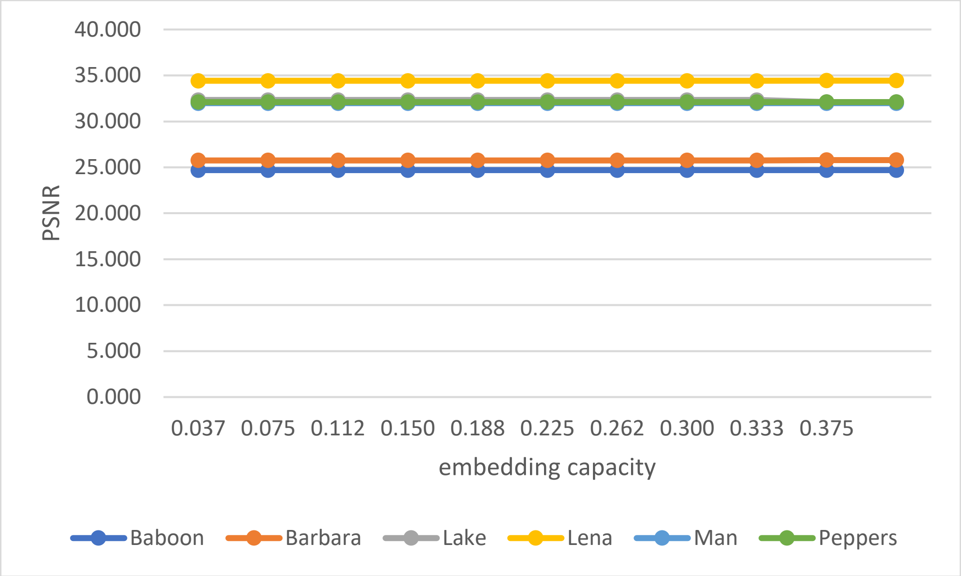


Figure D.1.3. PSNR of approximate image of Baboon, Barbara, Lake, Lena, Man and Peppers images with H matrix size 70×210. PSNR of approximate image is constant with different selection ratio α .

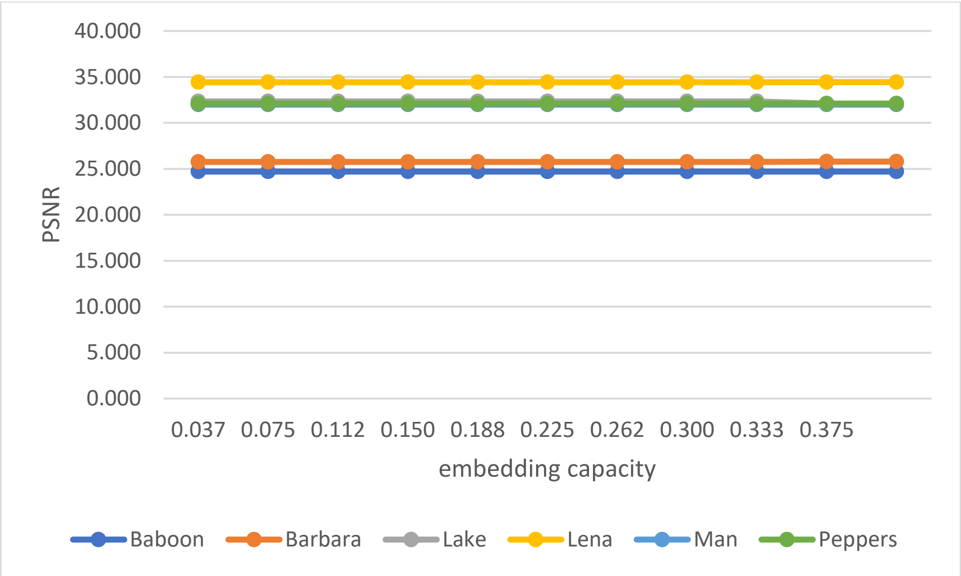


Figure D.1.4. PSNR of approximate image of Baboon, Barbara, Lake, Lena, Man and Peppers images with H matrix size 64×128. PSNR of approximate image is constant with different selection ratio α .

Appendix E. Relation between PSNR of Decoded Image and Embedding Capacity

Appendix E.1. Screen shots for 3 runs using 3 different H matrices with size 16×32

Results from Appendix A.4 line 49 – 50 shows the PSNR of decoded image and decoding time.

Table E.1.1. Screen shot for the run 1 using H matrix with size 16×32. Output from

Decoded PSNR										
'na'	'0.1'	'0.2'	'0.3'	'0.4'	'0.5'	'0.6'	'0.7'	'0.8'	'0.89'	'1.0'
'Baboon.bmp'	45.25839	46.55473	44.60898	34.26136	32.19933	31.61076	28.94002	28.23797	28.16623	27.55462
'Barbara.bmp'	60.172	44.48999	31.20674	29.11009	28.90744	29.05266	29.22729	29.1169	28.74186	26.59265
'Lake.bmp'	49.75808	46.36989	49.38019	40.77681	40.086	39.13397	38.32509	37.97092	37.69227	36.65018
'Lena.bmp'	Inf	Inf	57.1617	41.3071	41.25106	41.19573	41.08715	40.98122	40.04363	38.15803
'Man.bmp'	52.39049	50.62958	46.94981	40.58159	41.25106	41.84691	41.53877	39.64122	39.41653	38.15803
'Peppers.bmp'	41.3071	41.78351	40.34929	37.59522	37.76651	36.63092	35.92319	35.48853	35.2444	34.59693

Decoding time										
'na'	'0.1'	'0.2'	'0.3'	'0.4'	'0.5'	'0.6'	'0.7'	'0.8'	'0.89'	'1.0'
'Baboon.bmp'	4.553546	7.207947	9.989396	24.04089	26.73861	32.92438	50.53063	55.12887	62.12613	72.29037
'Barbara.bmp'	2.95727	6.388064	21.13061	30.00963	31.27537	33.43192	32.38402	38.44484	40.66086	57.35785
'Lake.bmp'	3.116078	4.173732	6.765961	9.34774	9.747098	12.05504	17.25501	19.49169	22.86653	24.91006
'Lena.bmp'	3.135189	4.297842	4.758447	9.275991	11.31722	14.29564	14.79261	17.72381	20.01436	22.7438
'Man.bmp'	3.169711	6.058579	8.198942	12.76935	14.84527	17.43823	18.35466	20.01948	24.66623	25.50749
'Peppers.bmp'	4.518196	6.389718	7.994263	10.5923	12.50944	14.53996	19.27694	20.96444	25.20058	26.02837

Table E.1.2. Screen shot for the run 2 using H matrix with size 16×32

Decoded PSNR										
'na'	'0.1'	'0.2'	'0.3'	'0.4'	'0.5'	'0.6'	'0.7'	'0.8'	'0.89'	'1.0'
'Baboon.b	45.54802	43.54443	41.91126	33.1305	32.39049	31.09789	28.64912	27.67757	27.51439	27.34125
'Barbara.b	57.1617	46.1926	31.04978	28.82386	29.08635	28.87188	28.87188	28.85261	28.78266	26.74186
'Lake.bmp	41.19573	45.85837	44.48999	39.79774	40.62958	37.76651	37.38447	37.29399	37.0757	37.03333
'Lena.bmp	Inf	60.172	55.40079	41.1411	41.03386	39.60295	40.98122	40.72718	40.21565	37.94484
'Man.bmp	51.1411	45.85837	46.36989	41.1411	42.04287	40.62958	39.71877	39.30841	38.46939	37.43042
'Peppers.	39.23779	40.12879	40.98122	37.50029	37.57129	37.14004	34.78124	35.1341	34.8957	34.46657

Decoding time										
'na'	'0.1'	'0.2'	'0.3'	'0.4'	'0.5'	'0.6'	'0.7'	'0.8'	'0.89'	'1.0'
'Baboon.b	5.473112	7.181073	9.281864	23.15091	25.84418	36.93274	60.68582	74.82411	80.83109	87.86073
'Barbara.b	3.433396	7.012335	24.95245	36.99691	38.98896	40.91105	45.33478	45.3304	48.92857	71.40598
'Lake.bmp	3.968241	6.032774	7.74554	12.18162	14.60098	19.20528	21.36327	25.12496	28.81613	29.8356
'Lena.bmp	3.12614	5.311592	7.915263	11.27973	13.54071	17.29732	18.59696	21.39393	23.99623	26.00923
'Man.bmp	3.716287	6.631338	9.5647	13.42476	15.66311	19.45844	22.12255	26.07359	29.55342	32.53696
'Peppers.	4.782635	6.76876	8.413011	12.35968	14.70538	18.63446	22.39591	24.38323	28.77215	30.25371

Table E.1.3. Screen shot for the run 3 using H matrix with size 16×32

Decoded PSNR										
'na'	'0.1'	'0.2'	'0.3'	'0.4'	'0.5'	'0.6'	'0.7'	'0.8'	'0.89'	'1.0'
'Baboon.bmp'	43.1823	41.84691	40.086	34.07606	32.03619	30.91888	29.1511	28.26309	27.91633	27.23617
'Barbara.bmp'	Inf	42.1102	31.3071	29.22729	29.1169	29.18211	29.30841	29.05602	29.13739	27.05023
'Lake.bmp'	44.48999	46.36989	46.1926	40.00167	39.91894	37.89314	37.84204	36.61174	36.90864	36.65018
'Lena.bmp'	Inf	Inf	Inf	41.53877	41.1411	39.41653	40.82702	40.04363	39.2029	37.97092
'Man.bmp'	50.172	47.1617	41.97656	40.92921	39.83777	38.32509	38.38223	37.97092	37.61928	37.0757
'Peppers.bmp'	41.08715	41.1411	40.58159	36.99137	36.76756	36.17527	34.96062	34.75621	34.96062	34.16227

Decoding time										
'na'	'0.1'	'0.2'	'0.3'	'0.4'	'0.5'	'0.6'	'0.7'	'0.8'	'0.89'	'1.0'
'Baboon.bmp'	3.753821	6.264045	11.1543	22.82801	32.70518	38.7568	56.1913	65.95	70.86365	83.38525
'Barbara.bmp'	3.430226	6.708176	23.98624	36.06612	38.82556	40.98383	43.04409	45.90643	48.72464	69.09038
'Lake.bmp'	3.856352	6.106944	8.791316	12.53488	15.02886	18.28211	21.29641	25.49636	28.93823	31.1276
'Lena.bmp'	3.275215	5.120785	8.149749	11.72423	13.9922	17.05403	18.7478	21.82527	23.96893	26.54119
'Man.bmp'	3.252687	6.791309	9.73734	12.51106	15.89959	18.77812	21.17931	23.2423	27.41695	30.79951
'Peppers.bmp'	4.508908	6.921923	8.99801	12.45671	14.91977	17.75208	20.89691	24.79863	26.52282	28.06354

Appendix E.2, Average PSNR of decoded image and average decoding time for each image.

Figure E.2.1. Screen shot for PSNR of decoded image and decoding time for baboon in each run

Decoded PSNR										
	'0.1'	'0.2'	'0.3'	'0.4'	'0.5'	'0.6'	'0.7'	'0.8'	'0.89'	'1.0'
run 1	45.25839	46.55473	44.60898	34.26136	32.19933	31.61076	28.94002	28.23797	28.16623	27.55462
run 2	45.54802	43.54443	41.91126	33.1305	32.39049	31.09789	28.64912	27.67757	27.51439	27.34125
run 3	43.1823	41.84691	40.086	34.07606	32.03619	30.91888	29.1511	28.26309	27.91633	27.23617
avg	44.6629	43.98202	42.20208	33.82264	32.20867	31.20918	28.91341	28.05954	27.86565	27.37735

decoding time										
	'0.1'	'0.2'	'0.3'	'0.4'	'0.5'	'0.6'	'0.7'	'0.8'	'0.89'	'1.0'
run 1	4.553546	7.207947	9.989396	24.04089	26.73861	32.92438	50.53063	55.12887	62.12613	72.29037
run 2	5.473112	7.181073	9.281864	23.15091	25.84418	36.93274	60.68582	74.82411	80.83109	87.86073
run 3	3.753821	6.264045	11.1543	22.82801	32.70518	38.7568	56.1913	65.95	70.86365	83.38525
avg	4.593493	6.884355	10.14185	23.33994	28.42932	36.20464	55.80259	65.30099	71.27362	81.17878

Figure E.2.2. Screen shot for PSNR of decoded image and decoding time for barbara in each run

Decoded PSNR										
	'0.1'	'0.2'	'0.3'	'0.4'	'0.5'	'0.6'	'0.7'	'0.8'	'0.89'	'1.0'
run1	49.75808	46.36989	49.38019	40.77681	40.086	39.13397	38.32509	37.97092	37.69227	36.65018
run2	41.19573	45.85837	44.48999	39.79774	40.62958	37.76651	37.38447	37.29399	37.0757	37.03333
run3	44.48999	46.36989	46.1926	40.00167	39.91894	37.89314	37.84204	36.61174	36.90864	36.65018
avg	45.14793	46.19938	46.68759	40.19207	40.21151	38.26454	37.85053	37.29222	37.22554	36.7779
decoding time										
	'0.1'	'0.2'	'0.3'	'0.4'	'0.5'	'0.6'	'0.7'	'0.8'	'0.89'	'1.0'
run1	3.116078	4.173732	6.765961	9.34774	9.747098	12.05504	17.25501	19.49169	22.86653	24.91006
run2	3.968241	6.032774	7.74554	12.18162	14.60098	19.20528	21.36327	25.12496	28.81613	29.8356
run3	3.856352	6.106944	8.791316	12.53488	15.02886	18.28211	21.29641	25.49636	28.93823	31.1276
avg	3.64689	5.437817	7.767606	11.35475	13.12565	16.51414	19.97156	23.371	26.87363	28.62442

Figure E.2.3. Screen shot for PSNR of decoded image and decoding time for Lake in each run

Decoded PSNR										
	'0.1'	'0.2'	'0.3'	'0.4'	'0.5'	'0.6'	'0.7'	'0.8'	'0.89'	'1.0'
run1	49.75808	46.36989	49.38019	40.77681	40.086	39.13397	38.32509	37.97092	37.69227	36.65018
run2	41.19573	45.85837	44.48999	39.79774	40.62958	37.76651	37.38447	37.29399	37.0757	37.03333
run3	44.48999	46.36989	46.1926	40.00167	39.91894	37.89314	37.84204	36.61174	36.90864	36.65018
avg	45.14793	46.19938	46.68759	40.19207	40.21151	38.26454	37.85053	37.29222	37.22554	36.7779
decoding time										
	'0.1'	'0.2'	'0.3'	'0.4'	'0.5'	'0.6'	'0.7'	'0.8'	'0.89'	'1.0'
run1	3.116078	4.173732	6.765961	9.34774	9.747098	12.05504	17.25501	19.49169	22.86653	24.91006
run2	3.968241	6.032774	7.74554	12.18162	14.60098	19.20528	21.36327	25.12496	28.81613	29.8356
run3	3.856352	6.106944	8.791316	12.53488	15.02886	18.28211	21.29641	25.49636	28.93823	31.1276
avg	3.64689	5.437817	7.767606	11.35475	13.12565	16.51414	19.97156	23.371	26.87363	28.62442

Figure E.2.4. Screen shot for PSNR of decoded image and decoding time for Lena in each run

Decoded PSNR										
	'0.1'	'0.2'	'0.3'	'0.4'	'0.5'	'0.6'	'0.7'	'0.8'	'0.89'	'1.0'
run1	Inf	Inf	57.1617	41.3071	41.25106	41.19573	41.08715	40.98122	40.04363	38.15803
run2	Inf	60.172	55.40079	41.1411	41.03386	39.60295	40.98122	40.72718	40.21565	37.94484
run3	Inf	Inf	Inf	41.53877	41.1411	39.41653	40.82702	40.04363	39.2029	37.97092
avg	INF	60.172	56.28125	41.32899	41.14201	40.07174	40.96513	40.58401	39.82073	38.0246
decoding time										
	'0.1'	'0.2'	'0.3'	'0.4'	'0.5'	'0.6'	'0.7'	'0.8'	'0.89'	'1.0'
run1	3.135189	4.297842	4.758447	9.275991	11.31722	14.29564	14.79261	17.72381	20.01436	22.7438
run2	3.12614	5.311592	7.915263	11.27973	13.54071	17.29732	18.59696	21.39393	23.99623	26.00923
run3	3.275215	5.120785	8.149749	11.72423	13.9922	17.05403	18.7478	21.82527	23.96893	26.54119
avg	3.178848	4.910073	6.941153	10.75998	12.95004	16.21566	17.37912	20.31434	22.65984	25.09807

Figure E.2.5. Screen shot for PSNR of decoded image and decoding time for Man in each run

Decoded PSNR										
	'0.1'	'0.2'	'0.3'	'0.4'	'0.5'	'0.6'	'0.7'	'0.8'	'0.89'	'1.0'
run1	52.39049	50.62958	46.94981	40.58159	41.25106	41.84691	41.53877	39.64122	39.41653	38.15803
run2	51.1411	45.85837	46.36989	41.1411	42.04287	40.62958	39.71877	39.30841	38.46939	37.43042
run3	50.172	47.1617	41.97656	40.92921	39.83777	38.32509	38.38223	37.97092	37.61928	37.0757
avg	51.23453	47.88322	45.09876	40.88397	41.0439	40.26719	39.87993	38.97352	38.50173	37.55472

decoding time										
	'0.1'	'0.2'	'0.3'	'0.4'	'0.5'	'0.6'	'0.7'	'0.8'	'0.89'	'1.0'
run1	3.169711	6.058579	8.198942	12.76935	14.84527	17.43823	18.35466	20.01948	24.66623	25.50749
run2	3.716287	6.631338	9.5647	13.42476	15.66311	19.45844	22.12255	26.07359	29.55342	32.53696
run3	3.252687	6.791309	9.73734	12.51106	15.89959	18.77812	21.17931	23.2423	27.41695	30.79951
avg	3.379562	6.493742	9.166994	12.90172	15.46932	18.55827	20.55217	23.11179	27.2122	29.61465

Figure E.2.6. Screen shot for PSNR of decoded image and decoding time for Peppers in each run

Decoded PSNR										
	'0.1'	'0.2'	'0.3'	'0.4'	'0.5'	'0.6'	'0.7'	'0.8'	'0.89'	'1.0'
run1	41.3071	41.78351	40.34929	37.59522	37.76651	36.63092	35.92319	35.48853	35.2444	34.59693
run2	39.23779	40.12879	40.98122	37.50029	37.57129	37.14004	34.78124	35.1341	34.8957	34.46657
run3	41.08715	41.1411	40.58159	36.99137	36.76756	36.17527	34.96062	34.75621	34.96062	34.16227
avg	40.54401	41.0178	40.63737	37.36229	37.36845	36.64874	35.22168	35.12628	35.03358	34.40859

decoding time										
	'0.1'	'0.2'	'0.3'	'0.4'	'0.5'	'0.6'	'0.7'	'0.8'	'0.89'	'1.0'
run1	4.518196	6.389718	7.994263	10.5923	12.50944	14.53996	19.27694	20.96444	25.20058	26.02837
run2	4.782635	6.76876	8.413011	12.35968	14.70538	18.63446	22.39591	24.38323	28.77215	30.25371
run3	4.508908	6.921923	8.99801	12.45671	14.91977	17.75208	20.89691	24.79863	26.52282	28.06354
avg	4.603247	6.693467	8.468428	11.8029	14.04486	16.9755	20.85659	23.3821	26.83185	28.1152

Appendix F. Relation between H Matrix Size and PSNR of Decoded Image

Appendix F.1. 9 H matrices are constructed using Gallager method.

Appendix F.1.1 H matrices for first run

4×8

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix}$$

8×16

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Appendix F.2. PSNR of decoded image for each run

The results of the following tables are from Appendix A.3 line 48

Table F.2.1. Screen shot of run 1 for PSNR of decoded image

Gallager Run1									
image	4x8	8x16	16v32	32x64	64x128	128x256	256x512	512x1024	1024x2048
'Baboon.bmp'	23.12707	26.79541	28.98605	29.93536	34.90861	37.71688	Inf	Inf	Inf
'Barbara.bmp'	24.20823	28.11184	30.46389	31.09789	36.57365	40.82702	Inf	Inf	Inf
'Lake.bmp'	30.85742	35.05317	38.41109	39.03257	45.1205	48.71072	Inf	Inf	Inf
'Lena.bmp'	32.64384	37.31643	39.71877	40.77681	46.55473	51.1411	Inf	Inf	Inf
'Man.bmp'	30.32223	34.51353	37.59522	38.02356	44.37417	46.36989	Inf	Inf	Inf
'Peppers.bmp'	30.51528	35.1205	37.69227	37.89314	44.04416	53.1823	Inf	Inf	Inf

Table F.2.2. Screen shot of run 2 for PSNR of decoded image.

Gallager Run 2									
image	4x8	8x16	16v32	32x64	64x128	128x256	256x512	512x1024	1024x2048
'Baboon.bmp'	24.69795	25.60076	31.8278	36.28034	42.1102	Inf	Inf	Inf	Inf
'Barbara.bmp'	25.74877	26.83149	34.20603	39.41653	50.62958	Inf	Inf	Inf	Inf
'Lake.bmp'	32.33297	33.65922	49.38019	54.1514	Inf	Inf	Inf	Inf	Inf
'Lena.bmp'	34.42012	35.93954	52.39049	Inf	Inf	Inf	Inf	Inf	Inf
'Man.bmp'	31.98315	33.30564	47.38447	54.1514	Inf	Inf	Inf	Inf	Inf
'Peppers.bmp'	32.09665	33.44179	45.40079	55.40079	Inf	Inf	Inf	Inf	Inf

Table F.2.3. Screen shot of run 3 for PSNR of decoded image.

Gallager Run 3									
image	4x8	8x16	16v32	32x64	64x128	128x256	256x512	512x1024	1024x2048
'Baboon.bmp'	24.69795	26.09979	28.62472	30.4085	43.63988	Inf	Inf	Inf	Inf
'Barbara.bmp'	25.74877	27.14869	29.93125	34.23914	52.39049	54.1514	Inf	Inf	Inf
'Lake.bmp'	32.33297	33.91888	39.71877	43.45102	Inf	Inf	Inf	Inf	Inf
'Lena.bmp'	34.42012	36.1926	43.54443	48.1308	Inf	Inf	Inf	Inf	Inf
'Man.bmp'	31.98315	33.79711	38.64912	42.39049	Inf	Inf	Inf	Inf	Inf
'Peppers.bmp'	32.09665	33.46954	39.56503	45.40079	Inf	Inf	Inf	Inf	Inf

Appendix F.4. Decoding time for each run.

The results of the following tables are from Appendix A.3 line 49

Table F.4.1. Screen shot of run 1 for decoding time for six images.

Gallager Run1									
image	4×8	8×16	16v32	32×64	64×128	128×256	256×512	512×1024	1024×2048
'Baboon.bmp'	10.47673	54.14742	102.472	321.3167	422.688	867.914	1267.343	3007.546	6701.473
'Barbara.bmp'	10.68511	49.82742	96.92752	295.552	406.9153	716.45	1203.991	2647.709	6014.362
'Lake.bmp'	10.35693	21.95986	39.29046	98.06526	143.7532	278.65	874.0114	1997.799	3943.389
'Lena.bmp'	9.302672	18.90812	33.41373	76.17266	134.4695	222.5902	782.9538	1856.171	4030.259
'Man.bmp'	9.787246	22.3549	38.03738	97.39861	146.7786	300.4301	892.0185	1944.709	3992.399
'Peppers.bmp'	10.02246	21.81069	41.06289	101.9598	156.7987	221.8305	912.3776	2005.541	3955.918

Table F.4.2. Screen shot of run 2 for decoding time for six images.

Gallager Run2									
image	4×8	8×16	16v32	32×64	64×128	128×256	256×512	512×1024	1024×2048
'Baboon.bmp'	38.56491	50.08845	61.60266	106.3889	246.3277	512.4624	1104.598	2718.944	6355.402
'Barbara.bmp'	29.01061	48.25352	59.33074	112.0737	263.2507	538.3843	1114.911	2404.545	5554.317
'Lake.bmp'	14.63632	22.23997	28.22339	59.68289	149.1373	367.0309	879.5724	2013.097	4085.289
'Lena.bmp'	14.17402	19.32453	27.76404	56.43714	135.0534	311.2852	793.9917	1833.284	3926.428
'Man.bmp'	15.63008	22.08051	30.51494	62.6272	159.2285	372.3432	899.5021	1998.015	4054.955
'Peppers.bmp'	15.50616	22.82836	29.8847	62.65376	158.9159	367.9129	888.3298	1953.765	4070.833

Table F.4.3. Screen shot of run 3 for decoding time for six images.

Gallager Run 3									
image	4×8	8×16	16v32	32×64	64×128	128×256	256×512	512×1024	1024×2048
'Baboon.bmp'	35.9228	78.82967	110.3678	181.9725	300.8356	596.8973	1308.089	2884.198	6322.048
'Barbara.bmp'	29.5108	65.07362	85.37798	136.662	254.1145	545.5668	1138.593	2391.63	5542.89
'Lake.bmp'	14.56222	25.69115	34.49064	65.24599	144.4779	339.4831	814.0754	1858.856	3613.74
'Lena.bmp'	11.64157	19.01288	25.70931	51.2655	108.7579	256.2376	636.2363	1542.868	3282.807
'Man.bmp'	13.56617	23.13476	31.60919	59.58722	127.8655	304.1008	738.9555	1914.146	4013.594
'Peppers.bmp'	14.20992	26.59643	34.92618	64.4915	153.1474	358.1463	871.9793	1923.503	3874.55

Appendix G. Results of Our Implementation for All Images

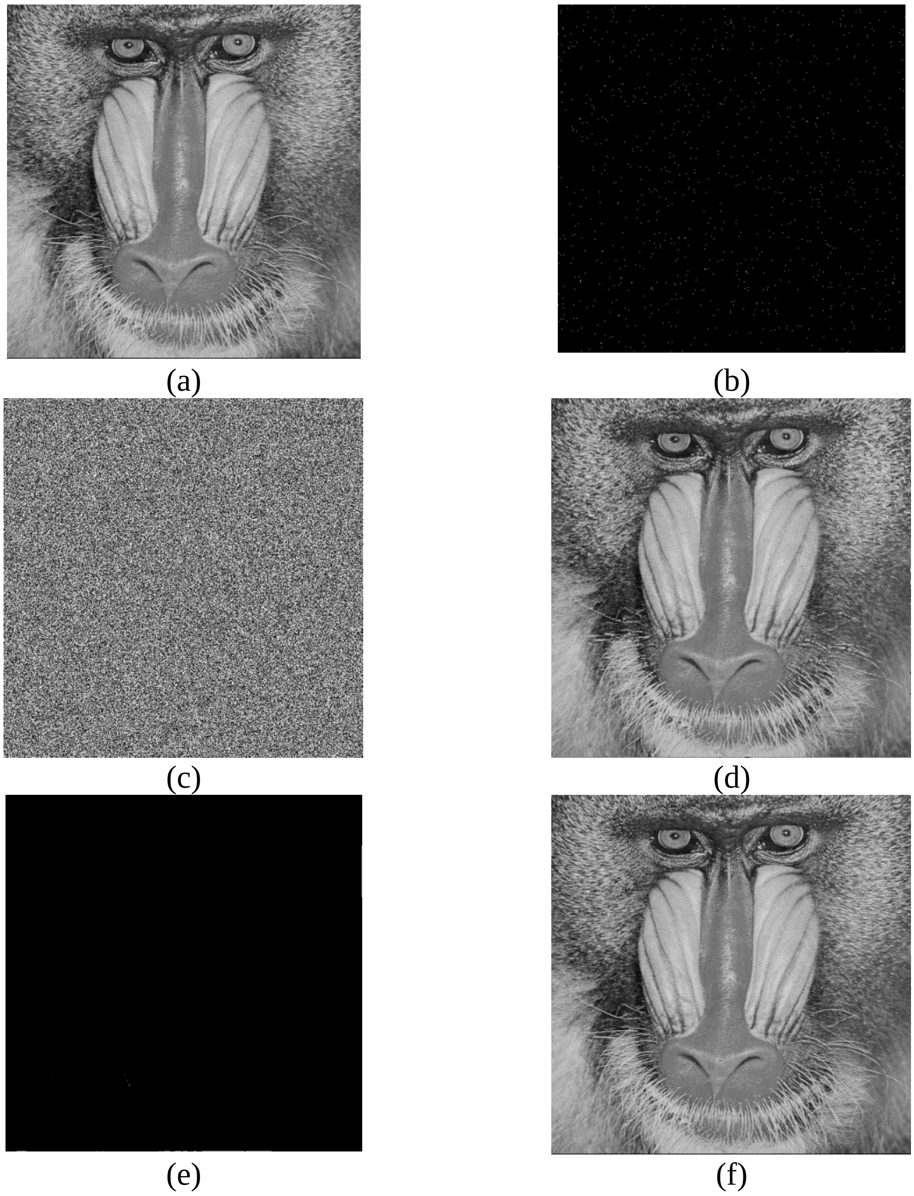


Figure G.1: (a) The Original Image *Baboon*. (B) The Encrypted Image (Stage 1). (C) Marked Encrypted Image (Stage 2). (D) The Approximate Image (Stage3, Option 2). (E) The Difference Between The Original And The Approximate Images. (F) Perfectly Recovered Image (Stage3, Option3).

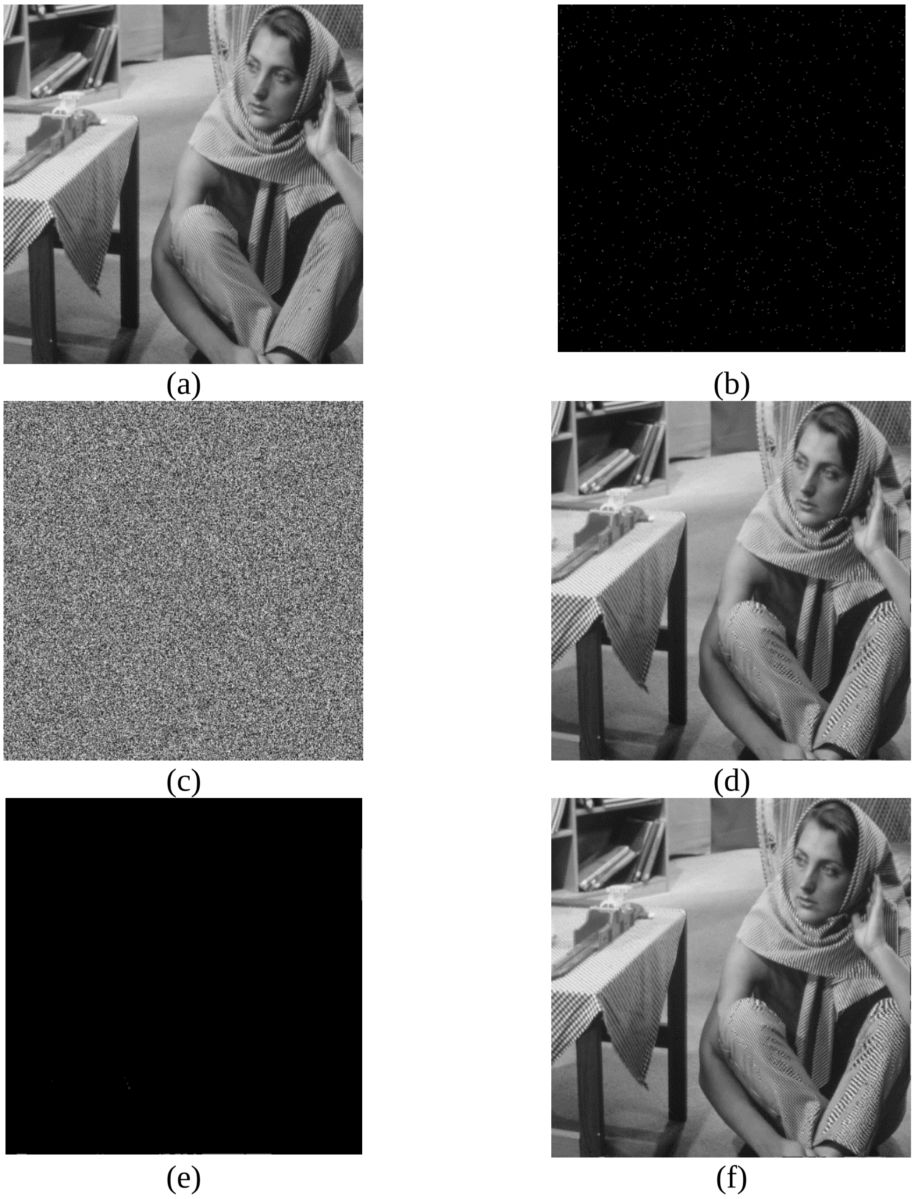


Figure G.2: (a) The Original Image *Barbara*. (B) The Encrypted Image (Stage 1). (C) Marked Encrypted Image (Stage 2). (D) The Approximate Image (Stage3, Option 2). (E) The Difference Between The Original And The Approximate Images. (F) Perfectly Recovered Image (Stage3, Option3).

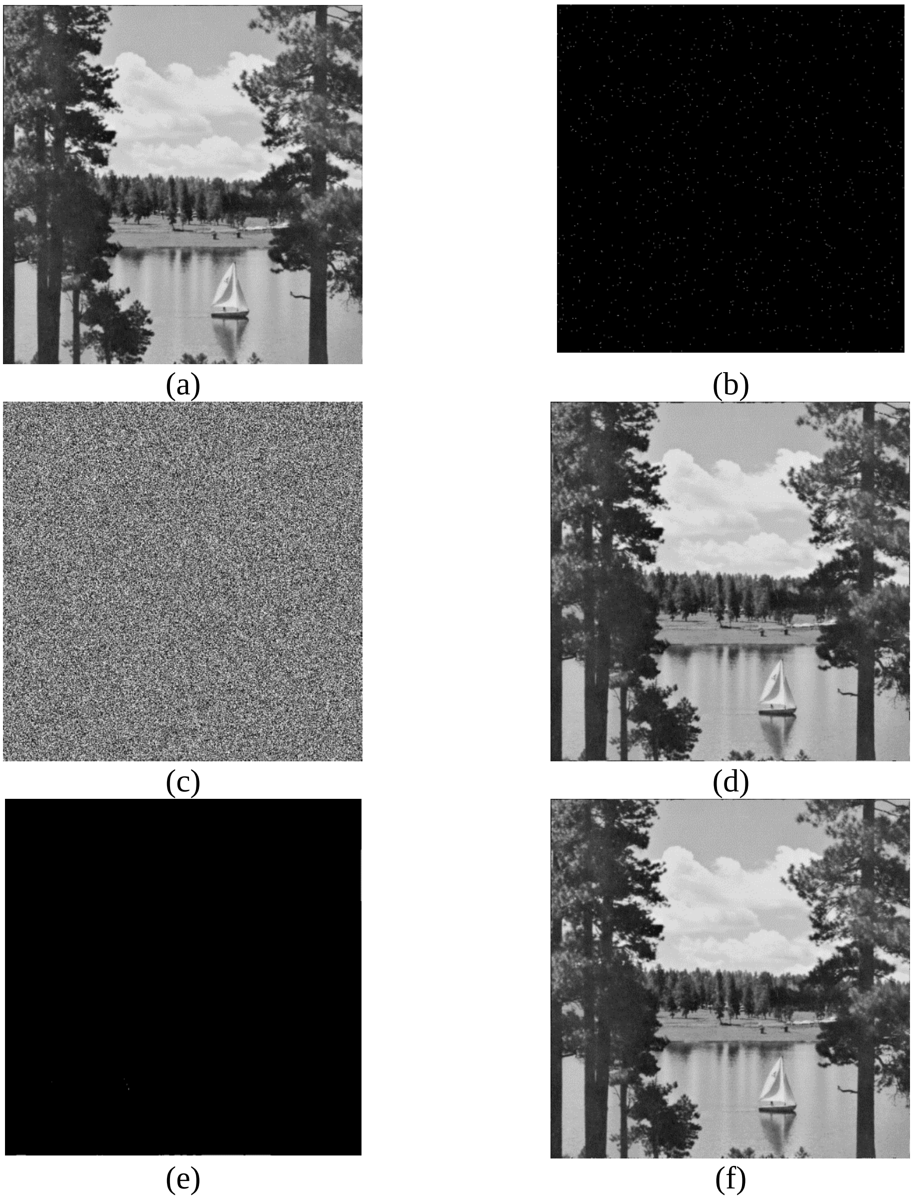


Figure G.3: (a) The Original Image *Lake*. (B) The Encrypted Image (Stage 1). (C) Marked Encrypted Image (Stage 2). (D) The Approximate Image (Stage3, Option 2). (E) The Difference Between The Original And The Approximate Images. (F) Perfectly Recovered Image (Stage3, Option3).

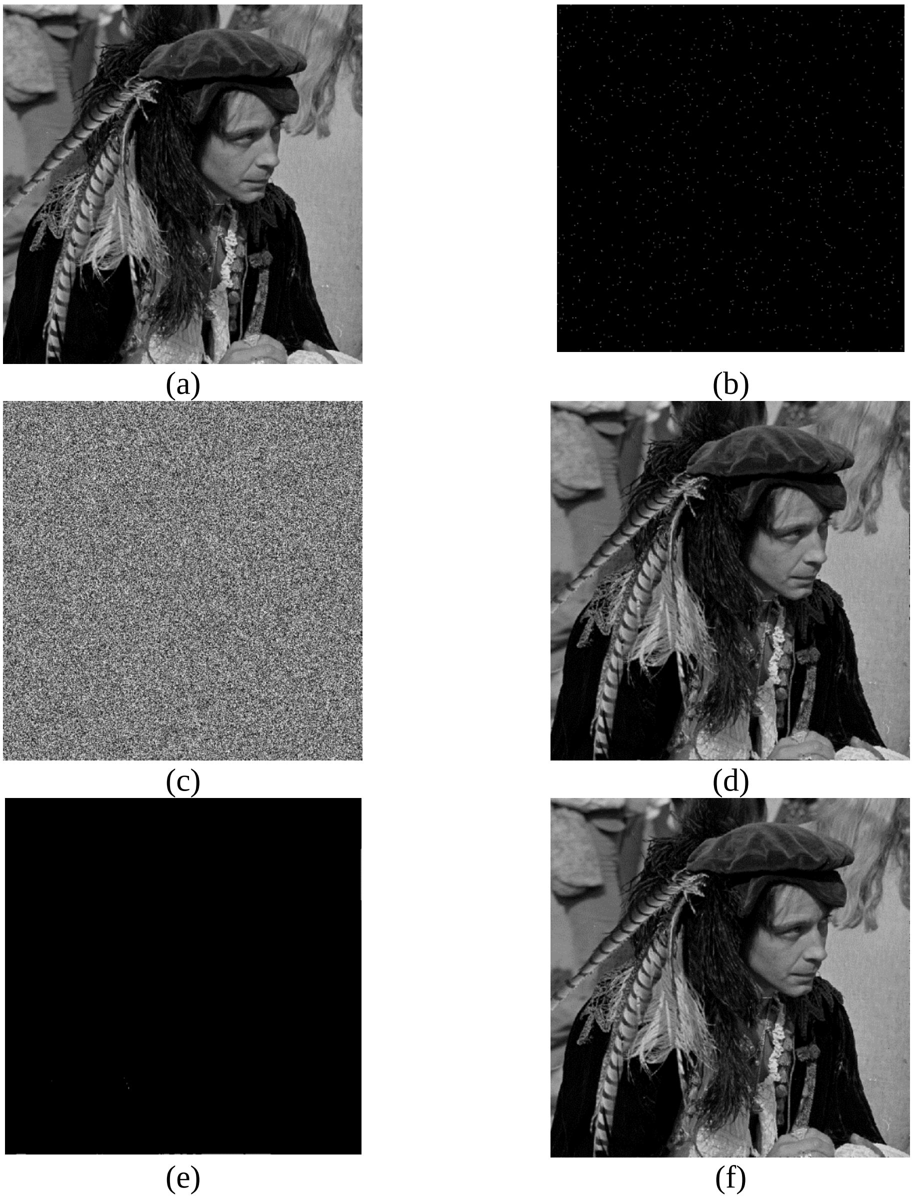


Figure G.4: (a) The Original Image *Man*. (B) The Encrypted Image (Stage 1). (C) Marked Encrypted Image (Stage 2). (D) The Approximate Image (Stage3, Option 2). (E) The Difference Between The Original And The Approximate Images. (F) Perfectly Recovered Image (Stage3, Option3).

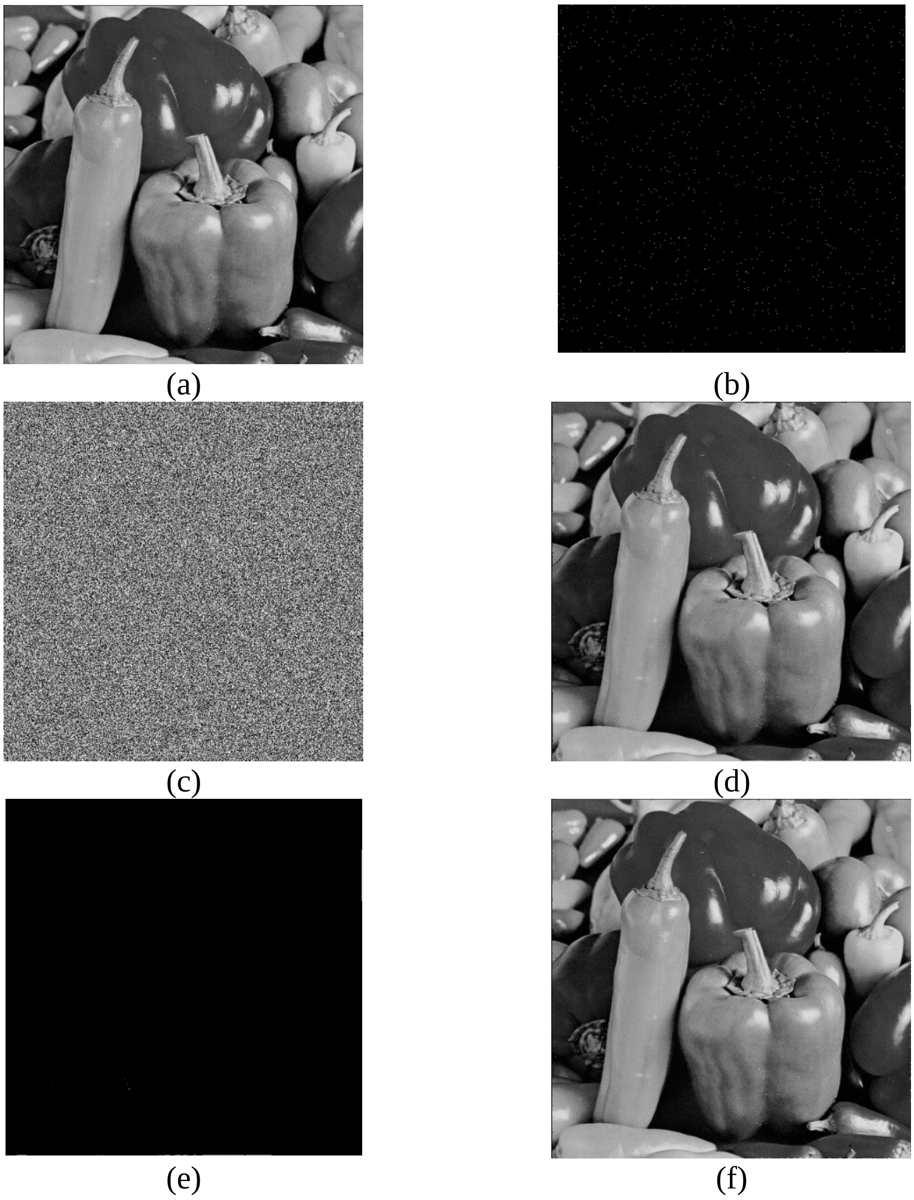


Figure G.5: (a) The Original Image *Peppers*. (B) The Encrypted Image (Stage 1). (C) Marked Encrypted Image (Stage 2). (D) The Approximate Image (Stage3, Option 2). (E) The Difference Between The Original And The Approximate Images. (F) Perfectly Recovered Image (Stage3, Option3).