UNIVERSITÉ DE NANTES

Polytech'Nantes
École d'ingénieurs de l'université de Nantes

STAGE DE MASTER 2R

SYSTEMES ELECTRONIQUES ET GENIE ELECTRIQUE

PARCOURS SYSTEMES ELECTRONIQUES

ANNEE UNIVERSITAIRE 2006/2007

CRYPTO-COMPRESSION BASED ON CHAOS OF STILL IMAGES FOR THE TRANSMISSION

YIN XU

École Polytechnique de l'Université de Nantes
Laboratoire IREENA

Encadrants du stage :

Safwan EL ASSAD

Vincent RICORDEL

Stage effectué du (05/02/2007) au (09/07/2007)

Acknowledgments

I would like to express my gratitude to my supervisors Professor Safwan EL ASSAD and Professor Vincent RICORDEL, who have been giving me direction with great patience. As the tutor of the first part of my internship, Prof. RICORDEL coached me on the study of JPEG 2000 systems and I benefited a lot from his tutorials. His constructive suggestions on my research and careful correction of my submitted materials are deeply appreciated. And as the tutor of the second part of my internship, Prof. EL ASSAD pointed out a clear direction of the subject that I should follow, which spare me from the potential wasting of time. He was always encouraging me and thus providing me with more confidence to achieve the objects of the internship. Doctor. Abir AWAD helped me a lot on the program-performance comparisons of C++ and MATLAB and she also gave me many useful materials on my subject. I am really thankful.

I would also give my thankfulness to Professor Joseph SAILLARD and Professor Patrick LE CALLET for attending my final defense as a member of the jury despite the busy occupation of their own work. Of course also send great thankfulness to my two supervisors Prof. EL ASSAD and Prof. RICORDEL as the other two members of the jury.

Last but no least, I would like to thanks all the professors who have given lectures to us, especially to Professor Yide WANG, who has been always helping us to lead a happy and comfortable life in France, to Professor Bernard REMAUD and Professor Joseph SAILLARD, who have done a lot to introduce to us the splendid culture and magnific landscape of France.

Contents

Contents 1
Résumé 3
Abstract 5
Part 1. JPEG 2000 7
1. Context: digital image compression 7
1.1. Needs for image compression 7
1.2. Lossless and lossy compression 7
1.3. General structure of compression system 8
1.4. Entropy of the code 8
2. JPEG2000 9
2.1. General introduction 9
2.2. Basic architecture of the standard 9
2.3. Preprocessing 10
2.4. Wavelet transform 12
2.5. Quantization 12
2.6. Entropy coding 12
2.7. Structure of the code stream 13
2.8. Scalability 20
2.8.1. Quality scalability 21
2.8.2. Resolution scalability 21
2.9. JP2 file format syntax 22
2.10. Conclusion 28
Part 2. ECKBA 29
3. Context: chaotic cryptography 29
3.1. Cryptography 29
3.2. Symmetric-key encryption 30
3.3. Public-key encryption 31
3.4. Chaos based cryptography 32
4. Enhanced Chaotic Key-Based Algorithm 32
4.1. General idea 32
4.2. Three main blocks 33
4.2.1. PWLCM 33
4.2.2. CBC 34
4.2.3. SP-network 34
4.3. Encryption algorithm 34
4.4. Decryption algorithm 37
Part 3. Experiments 38
5. Programs 38
6. Results 39
Conclusions 43

References 44
Annex A. Common header file of the projects 45
Annex B. Source codes related to class ImgBMP 46
Annex C. Source codes related to class ImgJP2 55
Annex D. Bitmap Storage 65
D.1. BITMAPFILEHEADER 67
Members 67
Remarks 68
D.2. BITMAPINFO 68
Members 68
Remarks 69
D.3. BITMAPINFOHEADER 69
Members 70
Remarks 73
D.4. RGBQUAD 74
Members 74
Remarks 74
Annex E. Encrypted and decrypted images 75

Résumé

En général, la transmission des données et tout particulièrement la transmission d'images fixes et de vidéos doit satisfaire à deux objectifs qui sont la réduction du volume des données pour désencombrer les réseaux publics de communication, et la sécurité maximale de l'information transmise. Par sécurité de l'information, on entend confidentialité des données, intégrité des données, authentification des données et des communicants, et non répudiation des données. La confidentialité, qui nous intéresse dans cette étude, consiste à garder des données secrètes pour tous ceux qui ne sont pas autorisés à les connaître.

Afin de satisfaire ces deux objectifs, une approche crypto-compression est nécessaire.

Le schéma de chiffrement/déchiffrement ou crypto-système basé chaos sera implémenté par l'algorithme ECKBA (Enhanced Chaotic Key-Based Algorithm).

Pour le schéma de compression, les transformations multi-échelles semblent intéressantes car elles permettent de prendre en compte, en les séparant, les grandes structures et les petits détails contenus dans une image. De ce point de vue elles possèdent des similarités avec le système visuel humain. Un cadre applicatif envisagé est alors celui du cryptage d'un flux JPEG 2000.

Par conséquent, mon étude comprend les étapes suivantes : étude des images au format JPEG2000 et étude d'un algorithme ECKBA de chiffrement. Le but est d'appliquer l'algorithme de chiffrement en question à une image du format JPEG2000.

Dans la première partie de ce rapport, le système JPEG 2000 est étudié dans son ensemble. Fondamentalement il y a trois blocs principaux dans ce système de codage (donc aussi 3 blocs principaux dans la partie de décodage), à savoir : la transformation en ondelettes (DWT), la quantification et le codage entropique. Avant ces blocs, le pré-traitement de l'image est aussi nécessaire.

La transformation DWT peut être irréversible ou réversible. Le cas irréversible est mis en application à l'aide du filtre de Daubechies 9/7 et la transformation réversible est mise en application à l'aide du filtre de Le Gall 5/3.

La quantification est avec pertes, à moins que le pas de quantification soit de 1 et les coefficients soient des nombres entiers, comme ceux produits par l'ondelette réversible.

Le codage entropique est réalisé au moyen d'un système de codage arithmétique qui comprime les symboles binaires source relativement à un modèle adaptatif dont les probabilités sont liées à chacun de 18 contextes différents de codage.

Le flux de données d'une image JPEG 2000 est organisé comme une succession des couches, qui augmentent graduellement la qualité de l'image.

Un dossier au format JP2 représente une collection de boîtes. Certaines de ces boîtes sont indépendantes, et d'autres contiennent d'autres boîtes. La structure d'un dossier est une successionde boîtes nécessaires: « Signature box », « File Type box », « JP2 Header box », « Contigus Codestream box » et d'autres boîtes facultatives. Le début de la première boîte sera le premier octet du dossier, et le dernier octet de la dernière boîte sera le dernier octet du dossier. Un ordre particulier des boîtes dans le dossier n'est généralement pas nécessaire. Cependant, « JPEG 2000 Signature box » sera la première boîte d'un dossier JP2, la « File Type box » suivra immédiatement la « JPEG 2000 signature box » et la « JP2 Header box » tombera avant la « Contigus Codestream box ».

Le flux binaire JPEG 2000 valide est contenu dans la « Contigus Codestream Box ». Le contenu de la boîte est organisé par un ensemble de marqueurs, de segments et de flux binaires. Six types des marqueurs et de segments de marqueurs sont employés.

La deuxième partie se concentre sur l'algorithme ECKBA (Enhanced Chaotic Key-Based Algorithm). ECKBA a été conçu à l'origine par D. Socek et S. Li en 2005 [10], basé sur le travail de CKBA (Chaotic Key-Based Algorithm) proposé par Yen et Guo [11].

CKBA utilise comme carte chaotique la fonction logistique mono dimensionnelle, qui a été montrée inévitablement vulnérable vis-à-vis des attaques extérieures et par conséquent la sécurité maximale n'est pas assurée. Par ailleurs, la séquence des bits donnée par la fonction logistique n'est pas équilibrée.

Plusieurs processus ont été rajoutés dans l'algorithme CKBA pour le rendre plus efficace :

La taille de la clé a été augmentée jusqu'à 128 bits au lieu de 32 bits ; la fonction logistique 1-D dans CKBA a été remplacée par la fonction linéaire par morceau PWLCM (piecewise linear chaotic map) pour améliorer la propriété d'équilibre de la carte chaotique ; un générateur de permutation pseudo aléatoire PRPG (pseudo-random permutation generator) utilisant une seconde carte chaotique a été utilisé dans le procédé de chiffrage et de déchiffrage afin de rajouter de la diffusion au système (P-boîte) ; une opération complexe de substitution (S-boîte), ainsi que des ronds multiples pour le procédé de chiffrage et de déchiffrage ont été appliqués.

Finalement, l'approche fondamentale de l'algorithme ECKBA n'est pas complexe. Il utilise principalement trois opérations essentielles : PWLCM, CBC (Cipher Block Chaining) et ronds multiples de SP-réseau. Les octets représentant les valeurs de pixel d'une image sont organisés en séquences 1-D et traités octet par octet. Premièrement des paramètres pour la S-boîte (substitution) et la P-boîte (permutation) prochaines sont produits en employant la fonction linéaire par morceau PWLCM. Ensuite, la technique CBC est employant, qui consiste à utiliser le dernier octet crypté pour faire le cryptage du nouvel octet. Ce mode relie les cryptogrammes les uns aux autres et la répétition de certains blocs dans le message clair n'est plus décelable dans le message crypté. Finalement l'octet crypté entre en ronds multiples de SP-réceau.

Dans la troisième partie de ce rapport nous présentons les résultats obtenus de simulation faites en C++. Nous avons tout d'abord appliqué l'algorithme ECKBA sur des images BMP afin de faire des comparaisons du temps d'exécution avec le même algorithme réalisé sous MATLAB par l'équipe où je suis. Les résultats montrent comme attendus que C++ est plus rapide que Matlab.

Ensuite, et afin d'appliquer l'algorithme sur des images JP2, nous avons mis en oeuvre une version de l'application ECKBA pour les images en question. Les noms de ces deux applications sont : ECKBA_BMPv1 et ECKBA_JP2v1. La différence entre les deux programmes se situe seulement au niveau de la lecture et l'écriture des images à traiter.

Les algorithmes sont exécutés sous le logiciel d'exploitation Windows sur un processeur de 1.86GHz Intel® Pentium® M. Des résultats des expériences nous pouvons dire qu'une amélioration significative a été réalisée en utilisant le langage C++ comparé à celui MATLAB. L'évolution du temps de fonctionnement pour le chiffrage et le déchiffrage en C++ est généralement linéaire basée sur la taille des données, tandis que dans MATLAB il ne l'est pas.

Abstract

In general, the data transmission and particularly the transmission of fixed images and video have to satisfy two objects, namely, the volume reduction of the data for not charging the public communication networks too much, and the maximum safety of transmitted information. By safety of information, we mean data confidentiality, data integrity, data authentication, and not repudiation of the data. The confidentiality, which interests us in this study, consists in keeping secret data from all those that are not authorized.

In order to satisfy these two objectives, a crypto-compression approach is necessary.

The encryption/decryption diagram or cryptosystem will be based on chaos using the ECKBA (Enhanced Chaotic Key-Based Algorithm) algorithm.

For the compression method, the multi-scales transformations seem interesting because they make it possible to take into account the basic structures and at the same time the small details contained in an image. From this point of view it's similar with the human visual system. An applicative framework then considered is that of encoding of a flow JPEG2000.

Therefore, my study of this project consists of the following parts: the study of the JPEG2000-format images and the study of an encryption algorithm ECKBA. The goal is to apply the encryption algorithm to an image of JPEG2000 format.

In the first part of this report, the thriving image coding system JPEG 2000 coding system is given an overview. Basically there are three main blocks in this coding system (also 3 main blocks in the decoding part), namely DWT forward transformation, quantization and entropy coding. Before entering these main blocks, preprocessing of the image is needed.

The DWT transformation can be irreversible or reversible. The default irreversible transform is implemented by means of the Daubechies 9/7 filter and the default reversible transformation is implemented by means of the Le Gall 5/3 filter.

Quantization is lossy, unless the quantization step is 1 and the coefficients are integers, as produced by the reversible integer 5/3 wavelet.

Entropy coding is achieved by means of an arithmetic coding system that compresses binary symbols relative to an adaptive probability model associated with each of 18 different coding contexts.

The codestream of a JPEG 2000 image is organized as a succession of layers, which gradually increase the quality of the image.

A JP2 format file represents a collection of boxes. Some of those boxes are independent, and some of those boxes contain other boxes. The binary structure of a file is a contiguous sequence of boxes: Signature box, File Type box, JP2 Header box, Contiguous Codestream box and other optional boxes. The start of the first box shall be the first byte of the file, and the last byte of the last box shall be the last byte of the file. A particular order of the boxes in the file is not generally implied. However, the JPEG 2000 Signature box shall be the first box in a JP2 file, the File Type box shall immediately follow the JPEG 2000 Signature box and the JP2 Header box shall fall before the Contiguous Codestream box.

The valid and complete JPEG 2000 codestream is contained in the Contiguous Codestream Box. The contents of the box are organized as a set of markers, marker segments and bitstreams. Six types of markers and marker segments are used: delimiting, fixed information, functional, in bit stream, pointer, and informational.

The second part focuses on the cryptography algorithm Enhanced Chaotic Key-Based Algorithm (ECKBA). ECKBA was brought forward by D. Socek and S. Li in 2005 [10], based on the work of CKBA (Chaotic Key-Based Algorithm) proposed by Yen and Guo [11].

CKBA is based on a one-dimensional Logistic map, which has been shown unavoidably susceptible to chosen/known-plaintext attacks and that the claimed high security against ciphertext-only attacks was overestimated by the authors. In addition, the Logistic map yields unbalanced output.

In ECKBA, several steps were applied for enhancement: The key size was increased to 128-bits; the 1-D Logistic map in original CKBA was substituted by a piecewise linear chaotic map (PWLCM) to improve the balance property of the chaotic map; a pseudo-random permutation generator (PRPG) based on the new chaotic map was introduced as an additional component in encryption and decryption process, forming a permutation box (P-box) and adding diffusion to the system; a more complex substitution box (S-box) was applied; multiple rounds for encryption and decryption process were introduced.

In essence, the basic idea of ECKBA is not complex. There are mainly three blocks in this algorithm: PWLCM, CBC and multiple rounds of SP-network. The bytes representing pixel values of a plain-image are organized into a one-dimension sequence and processed one byte by one byte: Firstly parameters for the upcoming S-box (Substitution) and P-box (Permutation) are generated by using the Piecewise Linear Chaotic Map (PWLCM). Then each plain-byte is masked with the previous cipher-byte (the cipher-block chaining, CBC), before entering the SP-network, which consists of an S-box and a P-box, for multiple rounds.

The third part gives the specifications of the experiments, which is carried out in C++. Firstly, implementing ECKBA algorithms with BMP files was carried out to make comparisons of the running time performances with the MATLAB programs, which was realized by preceding colleagues and turned out to be not as good as expected. Then implementing ECKBA with JP2 files was carried out.

Two projects have been created for implementing ECKBA with BMP files and with JP2 files respectively. The names of these projects are: ECKBA_BMPv1 and ECKBA_JP2v1. Both the projects use the same class ECKBA for encryption and decryption, the differences are the images types to be processed.

The projects are executed under Windows Operating System on a 1.86GHz Intel® Pentium® M processor. From the results of the experiments we can say that significant improvement has been achieved by implementing the algorithms in C++ compared to that of in MATLAB. The running time for both encryption and decryption in C++ are generally linear based on the size of the image pixel data, whereas in MATLAB it's no the case. Moreover, decryption time does not increase as fast as encryption in MATLAB, which, however, should increase at the same pace. On the contrary, encryption and decryption time in C++ are basically as should be expected.

As for the ECKBA_JP2v1 project, the time for both encryption and decryption are linear to the size of the image buffer. Besides, the relationship between encryption time and decryption time with the same number of rounds are more ideal than that of the BMP situation.

Part 1. JPEG 2000

1. Context: digital image compression

1.1. Needs for image compression

Thanks to the exponential growth in computing power and the fast spread of the Internet, digital images are used universally today in such diverse fields as astronomy, remote sensing, medicine, photojournalism, graphic arts, law enforcement, advertisement, manufacturing and so on.

Despite the advantages, the large number of bits required to represent the images can be a problem, not only because of the space needed for the storage, but especially in those applications in which transmission of images from one end to another is frequently involved, resulting in large consumption of bandwidth and slow transmission speed. Fortunately, digital images generally contain a significant amount of redundancy, which can be taken advantage of to reduce the number of bits required to represent an image, hence saving memory needed for image storage and channel capacity for image transmission.

1.2. Lossless and lossy compression

As mentioned above, the number of bits actually required to represent an image can be reduced thanks to redundancy. In general, there are three types of redundancy in digital images [1]:

- Spatial redundancy, which is due to the correlation between neighboring pixel values;
- Spectral redundancy, which is due to the correlation between different color planes or spectral bands;
- Temporal redundancy, which is due to the correlation between different frames in a sequence of images.

Image compression aims to remove these redundancies so as to reduce the number of bits required to represent an image.

In general, image compression can be categorized into two groups: lossless (reversible) and lossy (irreversible).

In lossless compression, the reconstructed image after compression is identical to the original one, which means that no information is dumped during the compression. However, only a modest amount of compression can be achieved. Lossless compression is often demanded in medical applications to avoid legal disputation over the significance of errors introduced into the imagery. It is also often applied in cases where it is difficult to determine how to introduce an acceptable loss. In the cases where the image is to be extensively edited and recompressed lossless compression is also used to avoid unacceptable accumulative errors caused by multiple lossy compression operations.

In lossy compression, much higher compression rate can be achieved but at the cost of

degradations of the reconstructed image due to the compression. The primary goal of lossy compression is to minimize the number of bits required to represent an image with an allowable level of distortion. The more distortion we accept, the smaller the compressed data can be. The most commonly used employed measure of distortion is MSE (Mean Squared Error) [2] and PSNR (Peak Signal to Noise Ratio) [2]. The PSNR is expressed in dB (decibels). Good reconstructed images typically have PSNR values of 30dB or more.

1.3. General structure of compression system

Figure 1 depicts the general structure which is most commonly used in image compression system.

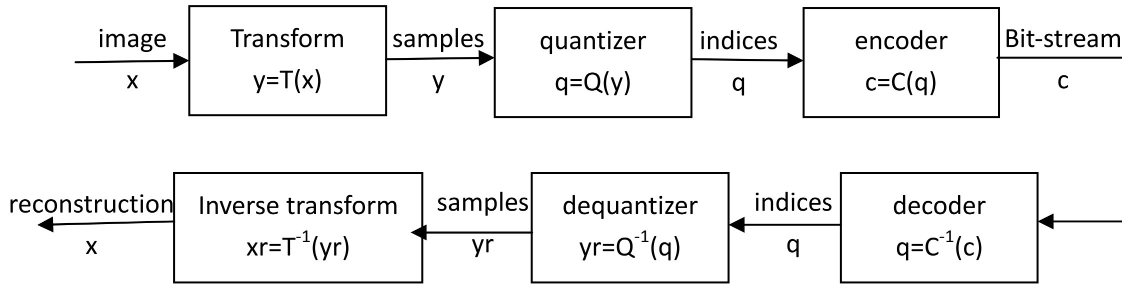


Figure 1: General structure of compression system.

The first step is to transform the original image samples into a form which enables comparatively simple quantization and coding operations. The decompressor employs the inverse transform, and no distortion is introduced by this step. The transform should capture the essence of statistical dependencies amongst the original image samples so that the transform samples and hence the quantization indices exhibit at most only very local dependencies. Ideally, they should be statistically independent. The transform should also separate irrelevant information from relevant information so that the irrelevant samples can be identified and quantized more heavily or even discarded. It is possible to construct transform which at least partially achieve both of these objectives simultaneously. [2]

The second step is to represent the transform samples approximately using a sequence of quantization indices. Quantization is solely responsible for introducing distortion, therefore the outcomes of each quantization index is generally much smaller than that for the transform samples; at the same time, the number of quantization indices may be smaller than that of the transform samples. Thus the quantization mapping introduces distortion and the decompressor uses an approximate inverse. For lossless compression, there should be no quantization.

Finally, the quantization indices are coded to form the final bit-stream. This step is invertible and introduces no distortion so that the decompressor can recover the quantization indices.

1.4. Entropy of the code

Any information-generating process can be viewed as a source that emits a sequence of symbols chosen from a finite alphabet. For example, a computer performs its computations on binary data, and such data may be considered as a sequence of symbols generated by a source with

a binary alphabet composed of 0 and 1. In the case of images, we can think of an n-bit image (each pixel of the image has a value of n bits) as being generated by a source with an alphabet of 2^n symbols representing the possible pixel values. By coding the source, which relates to an image, it is possible to measure the information conveyed. [1]

Source coding is the art of mapping each possible output from a given information source to a sequence of binary digits called “code bits”. If the mapping has the property that the code bits are entirely random, for example, taking values of 0 and 1 with equal probability, the code bits convey the maximum possible amount of information. Then, provide the mapping is invertible, one can identify the number of code bits with the amount of information in the original source output. [2]

The concept of “entropy” is defined in terms of the statistical properties of the source to measure the information conveyed. Entropy represents a lower bound on the average number of bits required to represent the source output. Moreover, it is possible to approach this lower bound arbitrarily closely as the complexity of the coding scheme grows without bound if allowed. [2]

2. JPEG2000

2.1. General introduction

The JPEG2000 standard was established to create a new image coding system for different types of still images (bi-level, gray level, color, multi-components), with different characteristics (natural images, scientific, medical, remote sensing, text, rendered graphics, etc) , allowing different imaging models (client/server, real-time transmission, image library archival, limited buffer and bandwidth resources, etc) , preferably within a unified system. [3] This coding system should provide low bit-rate operations with rate distortion and subjective image quality performances superior to existing standards, without sacrificing performances at other points in the rate-distortion spectrum, and at the same time incorporating many interesting features. Some of the most important features that this standard possess are listed as followed: [4]

Superior low bit-rate performance, Continuous-tone and bi-level compression, Lossless and lossy compression, Progressive transmission by pixel accuracy and resolution, Region-of-interest (ROI) coding, Open architecture, Robustness to bit errors, Protective image security, Content-based description, Side channel spatial information, Random code stream access and processing.

2.2. Basic architecture of the standard

Figure 2 [4] illustrates the block diagram of the JPEG2000 encoder (figure 2(a)) and decoder(figure 2(b)). The forward discrete wavelet transform (DWT) is first applied on the source image data. The transform coefficients are then quantized and entropy coded, before forming the final output codestream (bitstream). The decoder is the reverse of the encoder. The codestream is first entropy decoded, dequantized and inverse discrete wavelet transformed, thus resulting in the reconstructed image data.

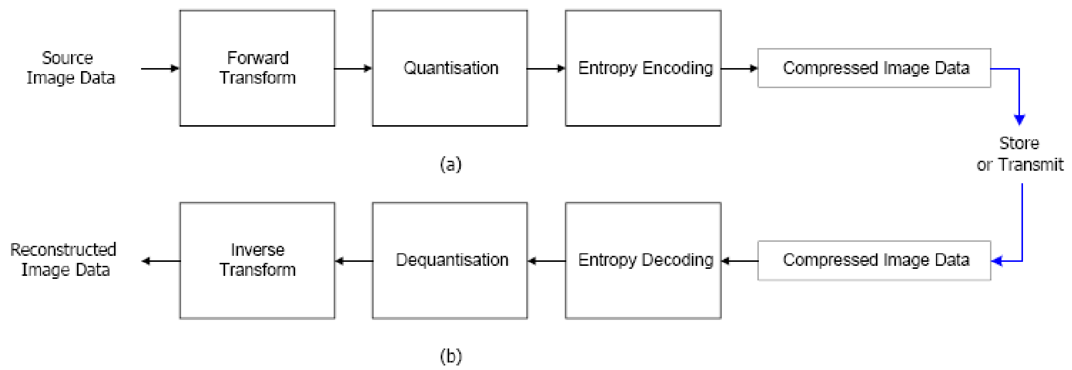


Figure 2: Block diagrams of the JPEG2000 (a) encoder and (b) decoder.

Before being wavelet transformed, there are several steps called preprocessing applied to the original image, namely image tiling, DC level shifting and component transformations. We will see more details of these steps in the following parts of this report.

2.3. Preprocessing

An overall illustration for the preprocessing is showed below in Figure 3.

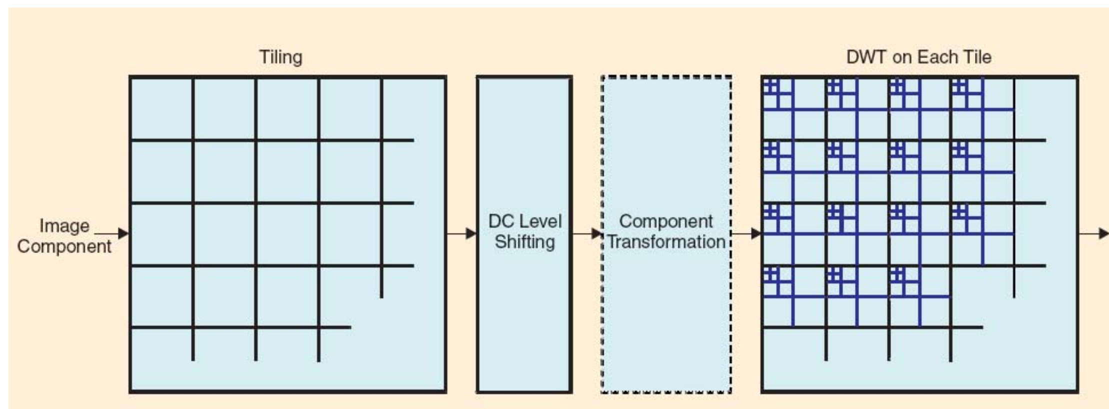


Figure 3: Tiling, DC level shifting, color transform (optional) and DWT of each image tile

One important operation is the tiling of an image. A tile is the basic unit on which the JPEG 2000 standard operates on. Tiling refers to the partition of the original image into rectangular non-overlapping blocks (tiles), which are compressed independently, as though they were entirely distinct images. All operations, including component mixing, wavelet transform, quantization and entropy coding are performed independently on the image tiles.

All tiles have exactly the same dimensions, except maybe those at the right and lower boundary of the image. Arbitrary tile sizes are allowed, up to and including the entire image (i.e. the whole image is regarded as one tile). Since smaller tiles create more tiling artifacts compared to larger tiles, tiling results in reduced quality. In other words, larger tiles perform visually better than smaller tiles.

Tiling reduces memory requirements and since they are also reconstructed independently,

they can be used for decoding specific parts of the image instead of the whole image.

Before the computation of the forward DWT on each image tile, all samples of the image tile component are DC level shifted by subtracting the same quality (i.e. the component depth). This operation is performed only on samples of components that are unsigned. If color transformation is used, it is performed prior to computation of the forward component transform. Otherwise it is performed prior to the wavelet transform. [4]

It is conclude from [3] that the component transformations have the following properties:

JPEG 2000 supports multiple-component images. Different components need not have the same bit depths nor need to all be signed or unsigned. For reversible (i.e., lossless) systems, the only requirement is that the bit depth of each output image component has to identical to the bit depth of the corresponding input image component.

Component transformations improve compression and allow for visually relevant quantization. The standard supports two different component transformations, one irreversible component transformation (ICT) that can be used for lossy coding and one reversible component transformation (RCT) that may be used for lossless or lossy coding, and all this in addition to encoding without color transformation. The block diagram of the JPEG 2000 multicomponent encoder is depicted in Figure 4 (Without restricting the generality, only three components are shown in the figure. These components could correspond to the RGB of a color image.) .

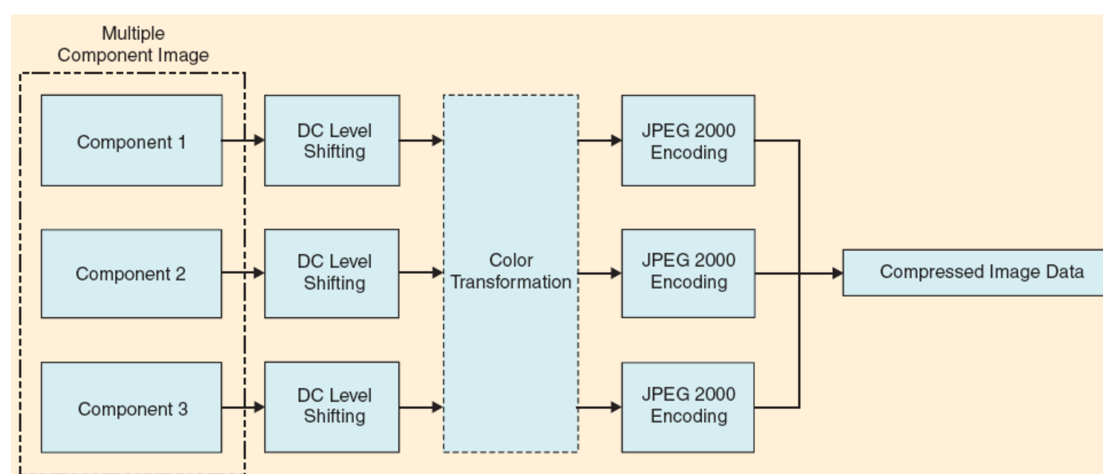


Figure 4: The JPEG 2000 multiple component encoder. Color transformation is optional. If employed, it can be irreversible or reversible.

Since the ICT may only be used for lossy coding, it may only be used with the 9/7 irreversible wavelet transform.

Since the RCT may be used for lossless or lossy coding, it may only be used with the 5/3 reversible wavelet transform. The RCT is a decorrelating transformation, which is applied to the three first components of an image. Three goals are achieved by this transformation, namely, color decorrelation for efficient compression, reasonable color space with respect to the human visual system for quantization, and ability of having lossless compression, i.e., exact reconstruction with finite integer precision. For the RGB components, the RCT can be seen as an approximation of a YUV transformation. All three of the components shall have the same sampling parameters and the same bit depth. There shall be at least three components if this transform is used.

2.4. Wavelet transform

Wavelet transform is used for the analysis of the tile components into different decomposition levels . These decomposition levels contain a number of subbands, which consist of coefficients that describe the horizontal and vertical spatial frequency characteristics of the original tile component. The three-level dyadic wavelet decomposition of the image “Lena” is shown in Figure 5.

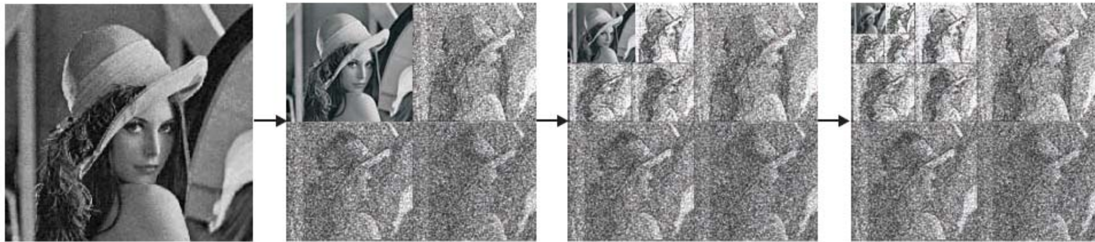


Figure 5: Three-level dyadic wavelet decomposition of the image “Lena”.

To perform the forward DWT the standard uses a one-dimensional (1-D) subband decomposition of a 1-D set of samples into low-pass and high-pass samples. Low-pass samples represent a down-sampled, low-resolution version of the original set. High-pass samples represent a down-sampled residual version of the original set, needed for the perfect reconstruction of the original set from the low-pass set. The DWT can be irreversible or reversible. The default irreversible transform is implemented by means of the Daubechies 9-tap/7-tap filter [4]. The default reversible transformation is implemented by means of the Le Gall 5-tap/3-tap filter.

2.5. Quantization

After transformation, all coefficients are quantized. Uniform scalar quantization with dead-zone about the origin is used in Part I and trellis coded quantization (TCQ) in Part II of the standard [3]. Quantization is the process by which the coefficients are reduced in precision. This operation is lossy, unless the quantization step is 1 and the coefficients are integers, as produced by the reversible integer 5/3 wavelet.

The quantization step-size is represented relative to the dynamic range of subband. In other words, the JPEG 2000 standard supports separate quantization step-sizes for each subband. However, one quantization step-size is allowed per subband. The dynamic range depends on the number of bits used to represent the original image tile component and on the choice of the wavelet transform. All quantized transform coefficients are signed values even when the original components are unsigned. These coefficients are expressed in a sign-magnitude representation prior to coding. For reversible compression, the quantization step-size is required to be one.

2.6. Entropy coding

Here, I will give a brief introduction of entropy coding since it’s not crucial for our

application to encrypt an image.

Entropy coding is achieved by means of an arithmetic coding system that compresses binary symbols relative to an adaptive probability model associated with each of 18 different coding contexts [3]. The MQ coding algorithm is used to perform this task and to manage the adaptation of the conditional probability models [4]. This algorithm has been selected in part for compatibility reasons with the arithmetic coding engine used by the JBIG2 compression standard and every effort has been made to ensure commonality between implementations and surrounding intellectual property issues for JBIG2 and JPEG 2000. The recursive probability interval subdivision of Elias coding is the basis for the binary arithmetic coding process. With each binary decision, the current probability interval is subdivided into two subintervals, and the code stream is modified (if necessary) so that it points to the base (the lower bound) of the probability subinterval assigned to the symbol, which occurred. Since the coding process involves addition of binary fractions rather than concatenation of integer code words, the more probable binary decisions can often be coded at a cost of much less than one bit per decision. [3]

As mentioned above, JPEG 2000 uses a very restricted number of contexts for any given type of bit. This allows rapid probability adaptation and decreases the cost of independently coded segments. The context models are always reinitialized at the beginning of each code block and the arithmetic coder is always terminated at the end of each block (i.e., once, at the end of the last sub bit plane). This is useful for error resilience also. (A code block is the fundamental entity for entropy coding)

2.7. Structure of the code stream

At the beginning, I would like to distinguish 2 concepts in order to avoid confusion in the up-coming text: the JPEG2000 “code stream” and the “bit stream” of a code block. The “code stream” refers to the whole image data after the JPEG2000 encoding, while “bit stream” refers to the entropy-coded data of a code block (we will see what is a code block later).

Between quantization and entropy coding, further divisions of each sub-band of each tile are applied, which lead to more benefits that I’ll explain later.

Firstly, each sub-band is divided into rectangles, overlapping or non-overlapping. Three spatially consistent rectangles at the same resolution level form a precinct (or a packet partition). The aim of precinct is to realize fast access to ROI (Region Of Interest), which, in my report, is out of concerns and thus will not be further explained. Yet a general understanding of such concept is possible in the graph that I’ll give later.

Secondly, each precinct is further divided into non-overlapping rectangles called code blocks, which form the basic input unit to the entropy coder. The size of the code block is typically 64×64 and not less than 32×32. See Figure 6 for a visual understanding.

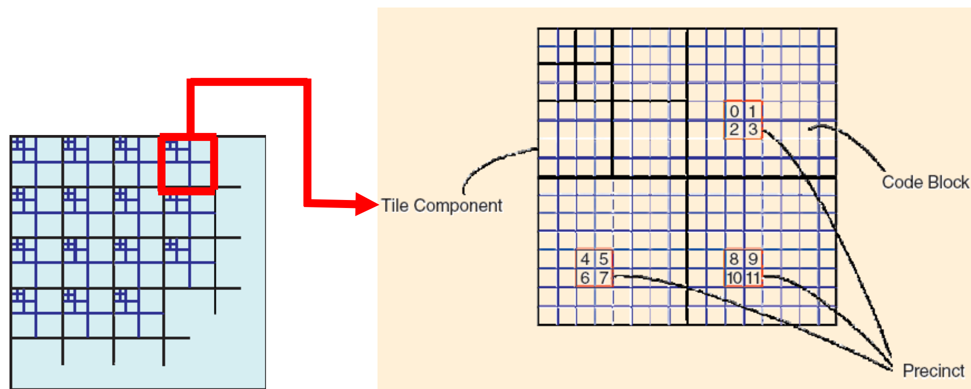


Figure 6: Partition of a tile into precincts and code blocks.

For each sub-band, the entropy coder visits the code blocks in raster orders: from the left-most code block to the right-most in the first row and continuing to the second row of code blocks, etc. See Figure 7. Each code block is a basic unit of coding and is coded entirely independently, without any reference to other code blocks either in the same or other sub-bands. Such independent block coding supplies significant benefits: spatial random access to the image content, parallel computations during coding or decoding, etc. [3]

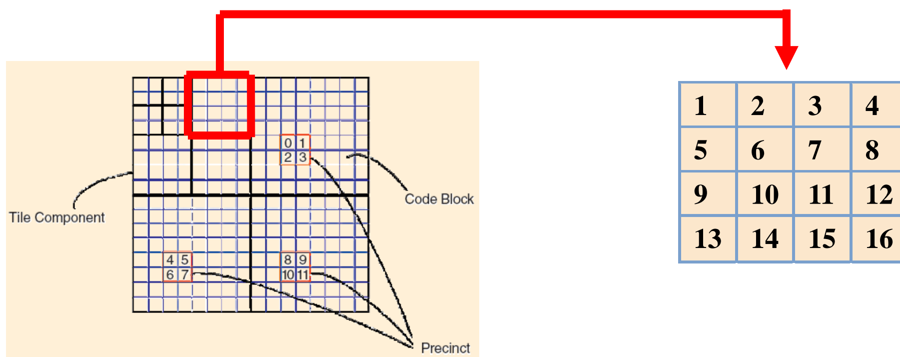


Figure 7: Visiting orders of code blocks for a sub-band.

For each code block, the entropy coder scans from the MSB (Most Significant Bit-plane) to the LSB (Least Significant Bit-plane). Each bit plane of a code block is scanned in a particular order: starting from the top left, the first 4 bits of the first column are scanned. Then the first 4 bits of the second column, until the width of the code block is covered. Then the next 4 rows are scanned vertically from left to right and so on. See figure 8.

Each individual bit plane of coefficients in a code block is coded using one and only one of the following three coding passes, namely the significance propagation, the magnitude refinement, and the cleanup pass. This process is called context modeling. The coding passes collect contextual information about the bit plane, for example, the importance of each individual coefficient bit. And the coder uses this information and its internal state to generate a compressed bit stream. Again, I'll not include the details of how these passes work to generate the bit stream and I'll focus on the structure of what comes out of such passes and coding scheme.

Different termination mechanisms allow different levels of independent extraction of this coding pass data. As mentioned previously, for each code block a separate bit stream is generated,

and during such process no information from other code blocks is utilized. The bit stream can be truncated to a variety of discrete lengths, incurring distortion. Truncation points of each bit stream (code block) are allocated by using rate distortion optimization [6]. During the encoding process, the lengths and the distortions are computed and temporarily stored with the compressed bit stream itself. In essence, the actual compression is achieved by truncating and/or re-quantizing the bit streams contained in each code block.

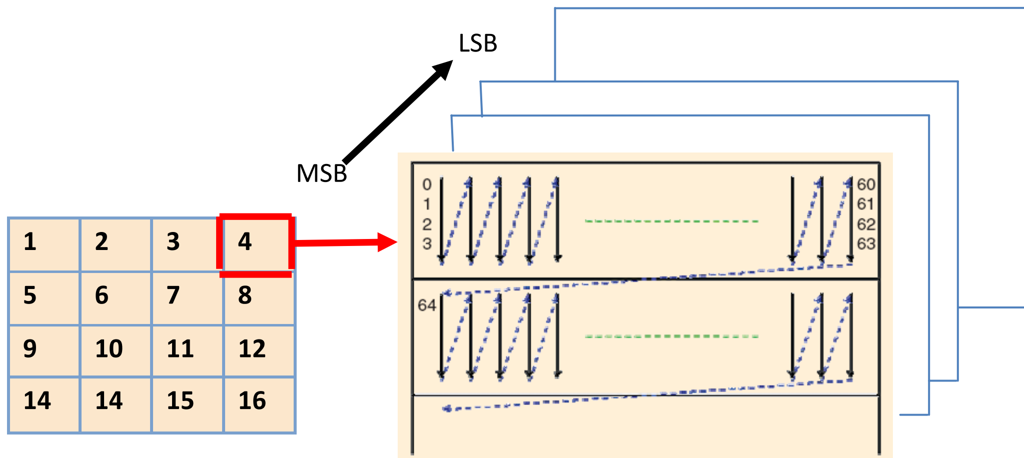


Figure 8: Scan pattern of bit planes of a code block.

The compressed bit streams from each code block in a precinct form the body of a packet. A collection of packets form a layer. A packet can be interpreted as one quality improvement for one resolution level at one spatial location, since precincts correspond roughly to spatial locations. Similarly, a layer can be interpreted as one quality improvement for the entire full resolution image. See Figure 9.

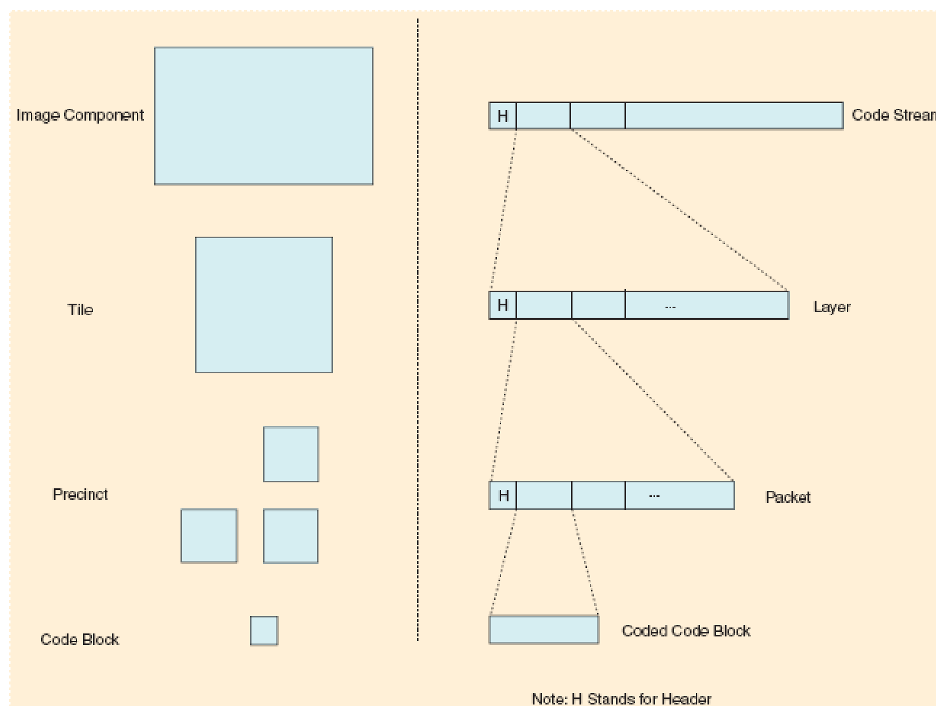


Figure 9: Conceprual correspodence between the spatial and the bit stream representation.

Thus, the code stream is organized as a succession of layers, each of which may contain the additional contributions from each code block. See Figure 10. It should be pointed out that some contributions of certain code blocks may be zero (i.e. the contribution of block 2 in layer 5 is zero in Figure 10) and the number of bits contributed by a code block is variable.

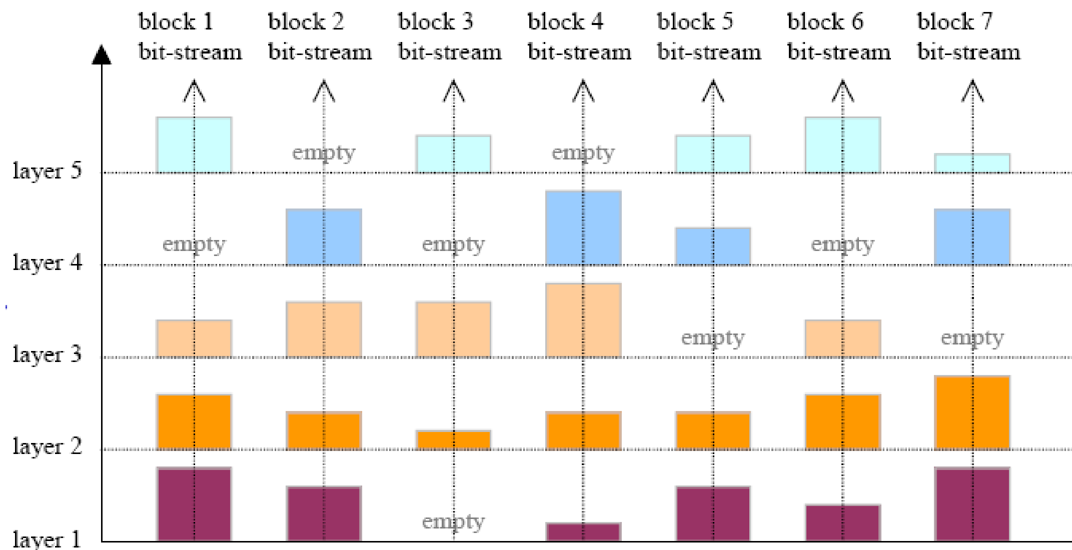


Figure 10: Different contributions of code blocks to different layers.

Each component is coded independently, and the coded data are interleaved on a layer basis. There are four scalabilities in the JPEG2000, namely resolution, quality, spatial location and component. Different types of scalabilities are achieved by the appropriate ordering of the packets within the bit stream.

Each layer successively and monotonically improves the image quality so that the more layers are decoded, the higher quality of the image can be gained. This allows the decoder to decode the image one quality level by one quality level, i.e. vertically from layer 1 to layer 5 in figure 10.

On the other side, the image can be decoded according to resolution levels. This means that code blocks from each sub-band successively improves the image resolution, i.e. horizontally from code-block 1 to code-block 7 in figure 10.

A detailed example of how such scalabilities perform in JPEG2000 is given below [5]. Take the 256×256 image "Lean" for example. The image is decomposed into 2 levels of resolutions as shown in Figure 11 and the code stream is shown in Figure 12. The size of each code block is 64×64. In this example, a layer is relevant to a bit plane.

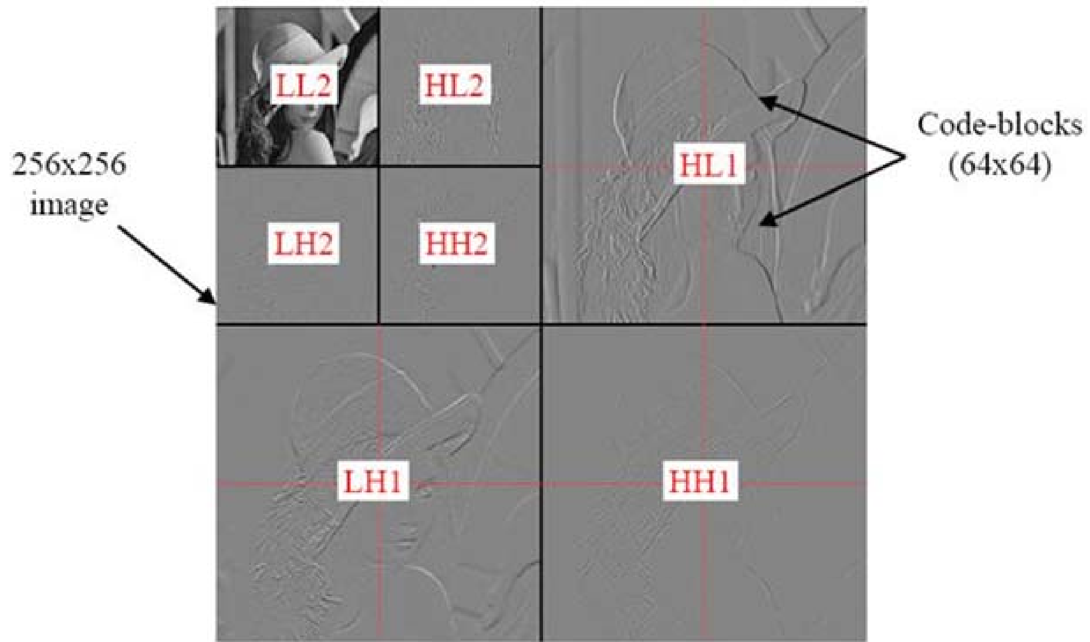


Figure 11: 2-level DWT decompositions of “Lena”.

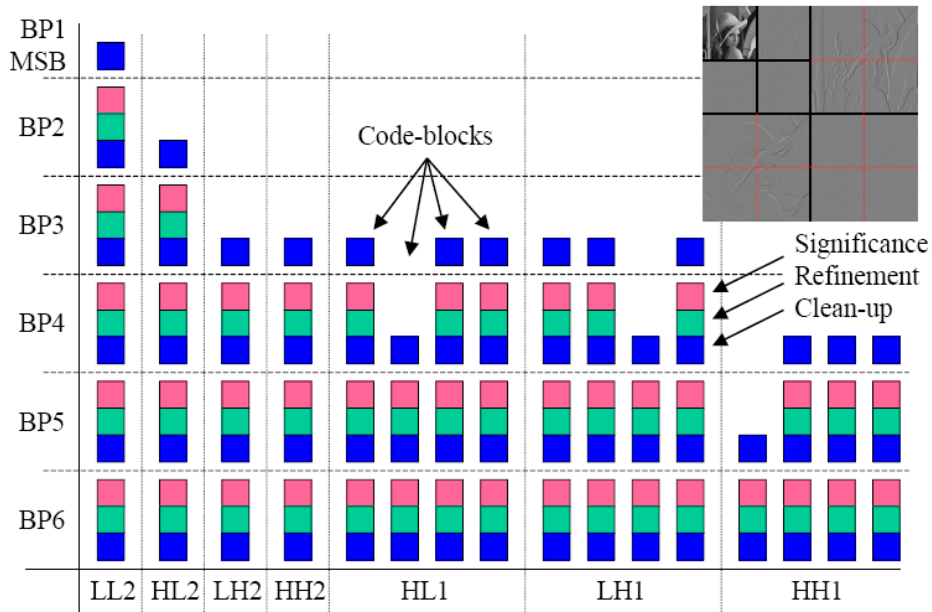


Figure 12: The bit-plane pass of coded data of “Lena”.

We will see in the following different resolution levels and quality layers from Figure 13 to Figure 17.

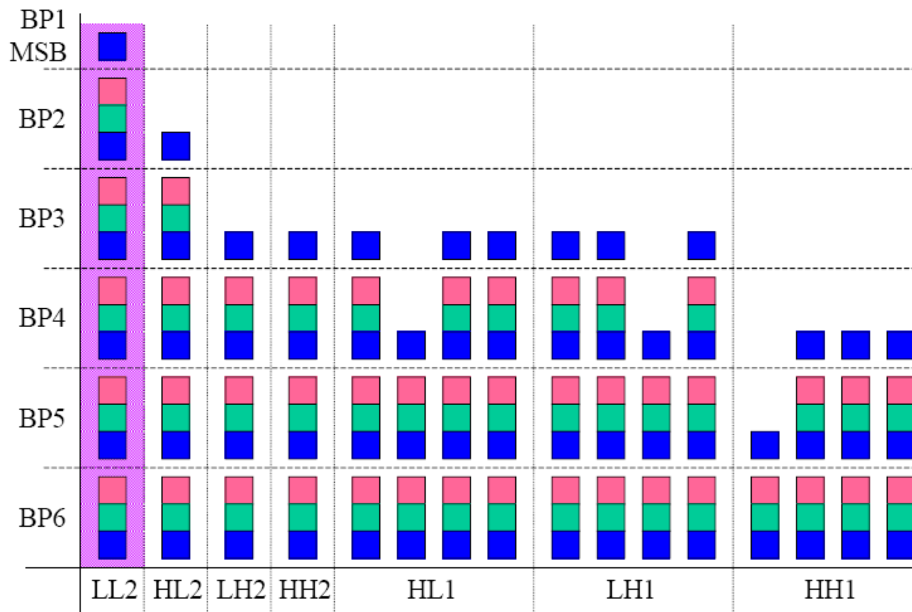


Figure 13: Lowest resolution, highest quality.

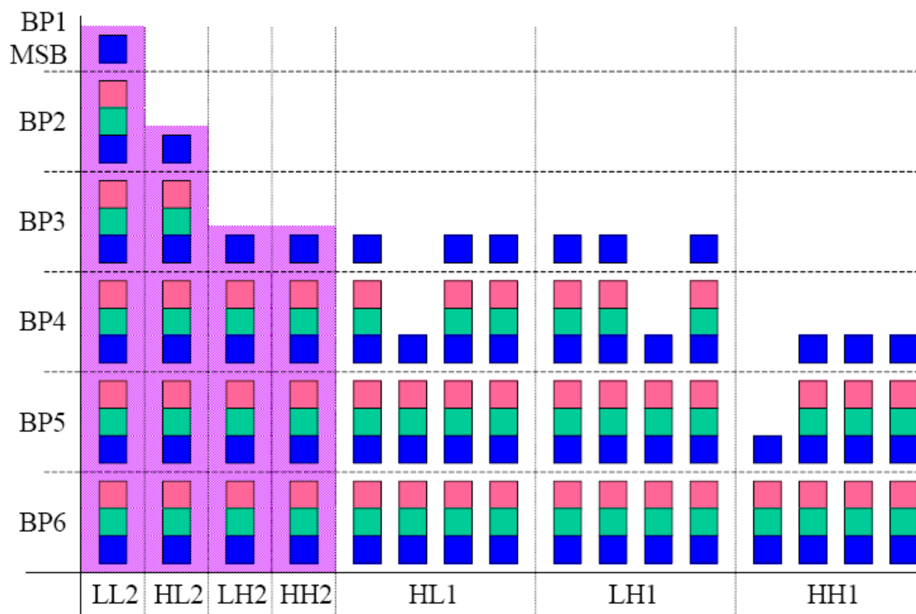


Figure 14: Medium resolution, highest quality.

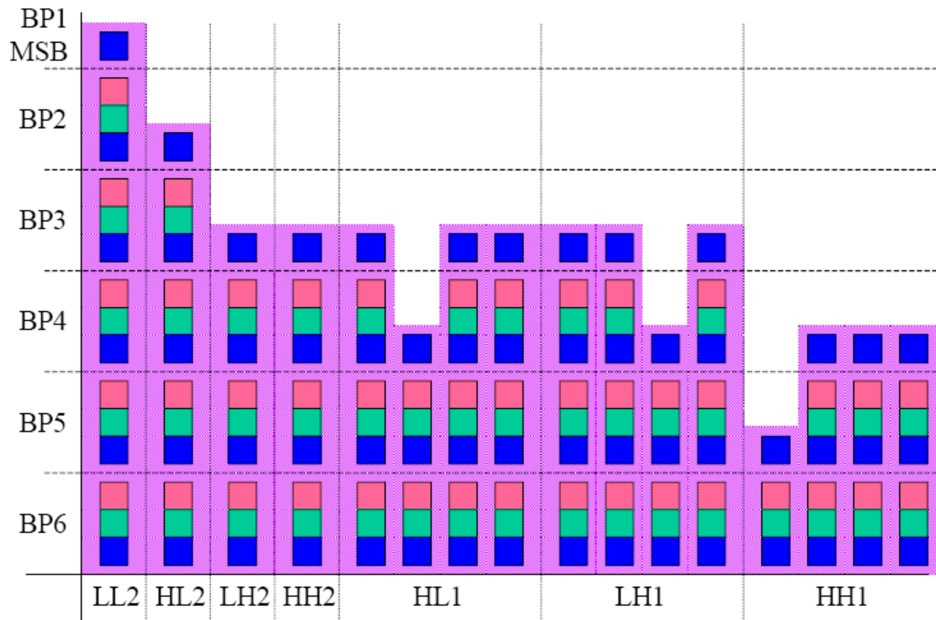


Figure 15: Highest resolution, highest quality.

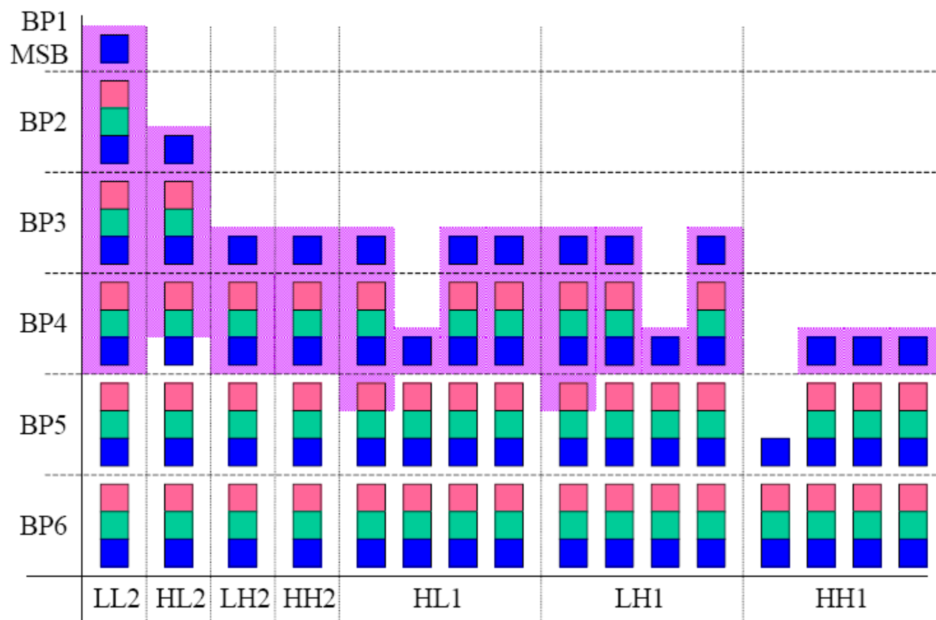


Figure 16: Highest resolution, target quality.

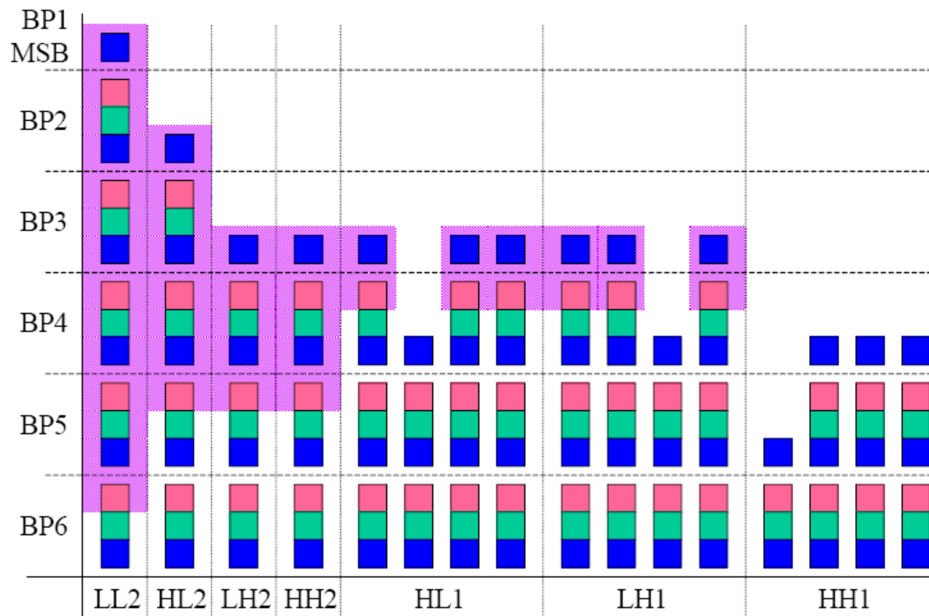


Figure 17: Highest resolution, target visual quality.

2.8. Scalability

In the above example, we've already seen different scalabilities in JPEG2000. Here, I'll explain a little bit more about this concept and give other examples.

In general, scalability in image coding means the ability to achieve more than one qualities and/or resolutions for an image. This property is practical in applications which require images to be simultaneously available for decoding at a variety of resolutions or qualities. It involves generating a coded representation of an image so as to facilitate the extraction of the image with more than one quality and/or resolution and enabling decoders of different complexities to decode various portions of the code stream according to their performances.

There are many advantages of scalable compression [3]: the target bit rate or reconstruction resolution needn't be known at the time of compression; the image needn't be compressed multiple times to achieve a target bit rate; resilience to transmission errors is available (i.e. the most important data of the lower layer can be sent over the channel with better error performance, while the less critical enhancement layer data can be sent over the channel with poor error performance), and so on.

The most important types of scalabilities are quality (SNR) scalability and resolution scalability, both supported by the JPEG2000 system, which are of great importance for Internet and database access applications and bandwidth scaling for robust delivery.

There are another two scalabilities supported by JPEG2000 system, namely spatial location scalability and component scalability. Spatial location scalability means that particular areas of an image can be decoded before the whole image being decoded, which is correspond to ROI in JPEG2000. Component scalability means that an image can be decoded on a component-by-component basis (i.e. for an YCrCb-image, the luminance component can be decoded and displayed).

Here, I'll explain further about the first two scalabilities.

2.8.1. Quality scalability

Quality scalability is also called SNR (Signal to Noise Ratio) scalability. It involves generating at least two image layers of the same resolution but with different qualities, from a single image source. The lower layer provides a basic quality and the higher layers provide enhancement qualities. See Figure 18 for an example [3]. This image is losslessly compressed and then decompressed at 0.125b/p, 0.25b/p, and 0.5b/p. It's obvious that the more the quality layers are decoded, the clearer the image is (the higher the quality of the image is), which is why we call it quality scalability.

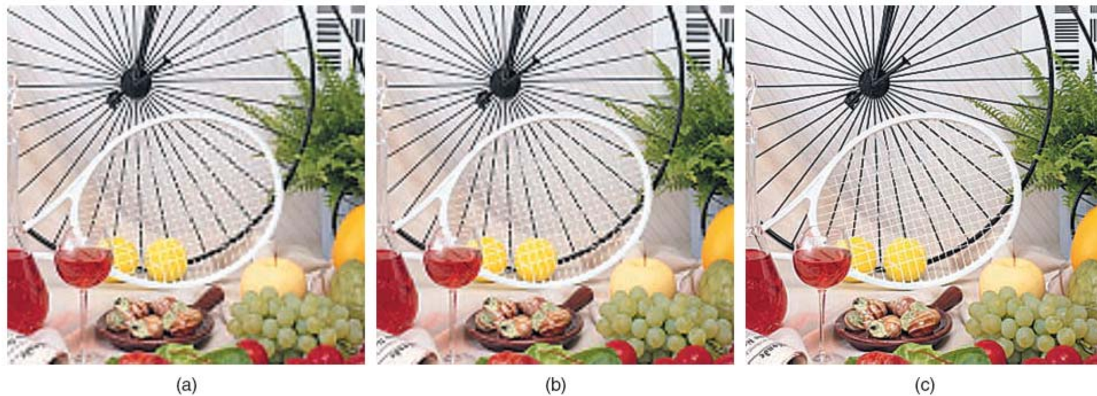


Figure 18: Example of quality scalability: (a) 0.125b/p, (b) 0.25b/p, (c) 0.5b/p.

2.8.2. Resolution scalability

Resolution in image processing means decoding the same image into various sizes by adding sub-bands information of the image. For example, in Figure 5, the image "Lena" is decomposed into 3 levels. By decoding only the lowest frequency part we obtain an image which is one 64th of the original size. When decoding the corresponding 3 sub-bands of the same level, an image which is one 16th of the original size can be obtained. This process can be continued for another 2 times until the original image is displayed.

Thus resolution scalability is intended for applications where to have at least 2 resolution layers is necessary. The lower layer provides the basic spatial resolution and the enhancement layer employs the spatially interpolated lower layer and carries the full spatial resolution of the input image source. It is practical for fast database access as well as for delivering different resolutions to terminals with different capabilities in terms of display and bandwidth. See Figure 19 for an example.

Figure 19: Example of resolution scalability.

At last, it should also be pointed out that JPEG2000 supports a combination of resolution and quality scalability. Therefore it is possible to progress by resolution at a given quality level and then change the progression to quality at the last achieved resolution level.

2.9. JP2 file format syntax

This sub-chapter is based on the ISO international standard ISO/IEC 15444-1 [12] [13].

A JP2 file represents a collection of boxes, some of those boxes are independent, and some of those boxes contain other boxes. The binary structure of a file is a contiguous sequence of boxes. The start of the first box shall be the first byte of the file, and the last byte of the last box shall be the last byte of the file. The binary structure of a box is defined Figure 20.

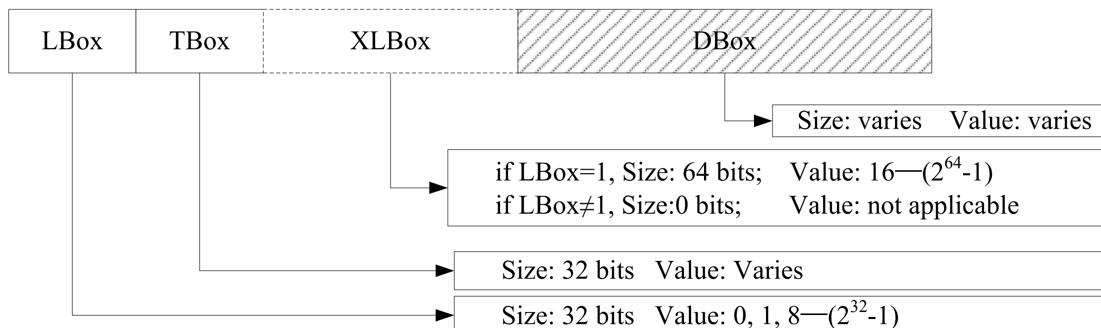


Figure 20: Organization of a box.

LBox: Box Length. This field specifies the length of the box, stored as a 4-byte big endian unsigned integer. This value includes all of the fields of the box, including the length and type. If the value of this field is 1, then the XLBox field shall exist and the value of that field shall be the actual length of the box. If the value of this field is 0, then the length of the box was not known when the LBox field was written. In this case, this box contains all bytes up to the end of the file.

If a box of length 0 is contained within another box (its superbox), then the length of that superbox shall also be 0. This means that this box is the last box in the file. The values 2-7 are reserved for ISO use.

TBox: Box Type. This field specifies the type of information found in the DBox field. The value of this field is encoded as a 4-byte big endian unsigned integer. However, boxes are generally referred to by an ISO/IEC 646 character string translation of the integer value.

XLBox: Extended Length. This field specifies the actual length of the box if the value of the LBox field is 1. This field is stored as an 8-byte big endian unsigned integer. The value includes all of the fields of the box, including the LBox, TBox and XLBox fields.

DBox: Box Contents. This field contains the actual information contained within this box. The format of the box contents depends on the box type and will be defined individually for each type.

For example, consider the illustration in Figure 21 of a sequence of boxes, including on box that contains other boxes:

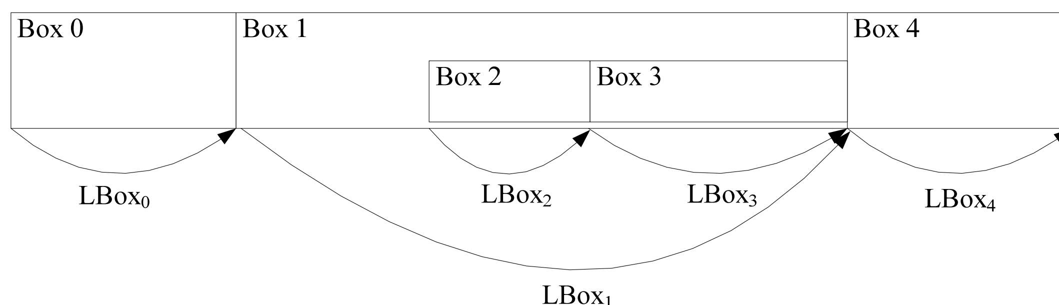


Figure 21: Illustration of box lengths.

Logically, the structure of a JP2 file is as shown in Figure 22. Boxes with dashed borders are optional in conforming JP2 files. Superbox refers to those boxes which contain other boxes in their contents. The figure only specifies the containment relationship between the boxes in the file. A particular order of those boxes in the file is not generally implied. However, the JPEG 2000 Signature box shall be the first box in a JP2 file, the File Type box shall immediately follow the JPEG 2000 Signature box and the JP2 Header box shall fall before the Contiguous Codestream box.

The JPEG 2000 Signature box identifies that the format of the file was defined by the JPEG 2000 ISO/IEC 15444-1 Standard, as well as provides a small amount of information which can help determine the validity of the rest of the file. The JPEG 2000 Signature box shall be the first box in the file, and all files shall contain one and only one JPEG 2000 Signature box.

The File Type box specifies the ISO/IEC 15444-1 Standard which completely defines all of the contents of the file, as well as a separate list of readers, defined by other Recommendations | International Standards, with which the file is compatible, and thus the file can be properly interpreted within the scope of that other standard. This box shall immediately follow the JPEG 2000 Signature box. This differentiates between the standard which completely describes the file, from other standards that interpret a subset of the file. All files shall contain one and only one File Type box.

The JP2 Header box contains generic information about the file, such as number of components, colourspace, and grid resolution. This box is a superbox. Within a JP2 file, there shall be one and only one JP2 Header box. The JP2 Header box may be located anywhere within the

file after the File Type box but before the Contiguous Codestream box. It also must be at the same level as the JPEG 2000 Signature and File Type boxes (it shall not be inside any other superbox within the file).

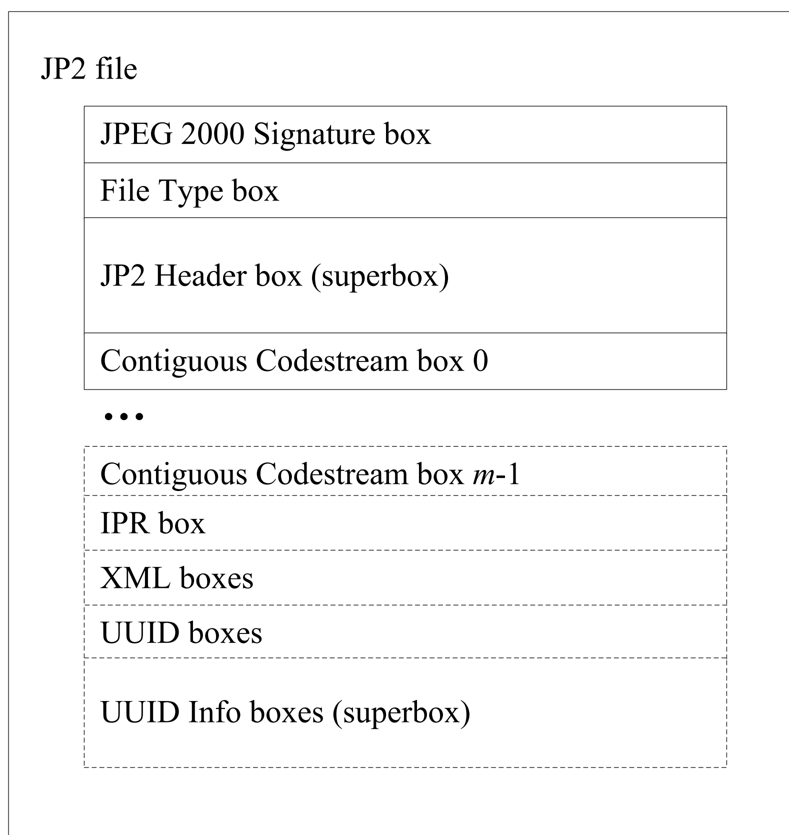


Figure 22: Conceptual structure of a JP2 file.

Boxes with type IPR, XML, UUID and UUID Info carry intellectual property rights information within a JP2 file. Inclusion of such information in a JP2 file is optional for conforming files.

The valid and complete JPEG 2000 codestream is contained in the Contiguous Codestream Box. When displaying the image, a conforming reader shall ignore all codestream after the first codestream found in the file. The contents of the box are organized as a set of markers, marker segments and bitstreams.

A marker segment, which is used to delimit and signal the characteristics of the codestream, includes a marker and associated parameters, called marker parameters. See Figure 23.

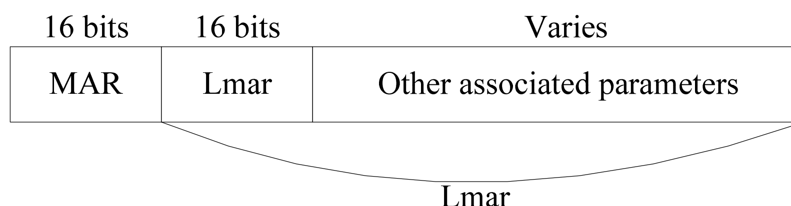


Figure 23: Structure of a marker segment.

MAR: Marker. Every marker is two bytes long. The first byte consists of a single 0xFF byte. The second byte denotes the specific marker and can have any value in the range 0x01 to 0xFE. Many of these markers are already used in ITU-T Rec. T.81 | ISO/IEC 10918-1 and ITU-T Rec.

T.84 | ISO/IEC 10918-3 and are regarded as reserve unless specifically used. Some markers, such as SOC, SOT, SOD, EOC, are independent and are not associated with any marker segment, they are used for delimiting particular parts of the codestream.

Lmar: Length of the marker parameters. In every marker segment the first two bytes after the marker shall be an unsigned big endian integer value that denotes the length in bytes of the marker parameters (including two bytes of this length parameter but not the two bytes of the marker itself).

Six types of markers and marker segments are used: delimiting, fixed information, functional, in bit stream, pointer, and informational. Delimiting marker and marker segments must be used to frame the headers and the data. Fixed information marker segments give required information about an image. The location of these marker segments, like delimiting marker segments, is specified. Functional marker segments are used to describe the coding functions used. In bit stream markers and marker segments are used for error resilience. Pointer marker segments point to specific offsets in the bit stream. Informational marker segments provide ancillary information. Table 1 lists the markers specified in the ISO/IEC 15444-1 [12].

Table 1: List of marker segments

	Name	Code	Main header	Tile-part header
Delimiting marker segments				
Start of codestream	SOC	0xFF4F	Required	Not allowed
Start of tile-part	SOT	0xFF90	Not allowed	Required
Start of data	SOD	0xFF93	Not allowed	Last marker
End of codestream	EOC	0xFFD9	Not allowed	Not allowed
Fixed information marker segments				
Image and tile size	SIZ	0xFF51	required	Not allowed
Functional marker segments				
Coding style default	COD	0xFF52	required	Optional
Coding style component	COC	0xFF53	Optional	Optional
Region-of –interest	RGN	0xFF5E	Optional	Optional
Quantization default	QCD	0xFF5C	Required	Optional
Quantization component	QCC	0xFF5D	Optional	Optional
Progression order default	POD	0xFF5F	Optional	Optional
Pointer marker segments				
Tile-part lengths, main header	TLM	0xFF55	Optional	Not allowed
Packet length, main header	PLM	0xFF57	Optional	Not allowed
Packet length, tile-part header	PLT	0xFF58	Not allowed	Optional
Packed packet headers, main header	PPM	0xFF60	Optional	Not allowed
Packed packet headers, tile-part header	PPT	0xFF61	Not allowed	Optional
In bit stream marker segments				
Start of packet	SOP	0xFF91	Not allowed	Optional, in bit stream
End of packet header	EPH	0xFF92	Not allowed	Optional, in bit stream
Informational marker segments				
Comment and extension	CME	0xFF64	Optional	Optional

The construction of the codestream is showed in Figure 24. The construction of the main header is showed in Figure 25 and the construction of a tile-part header is showed in Figure 26. All of the solid lines show required marker segments. The marker segments on the left are required to be in a specific location. The dashed lines show optional or possibly not required marker segments.

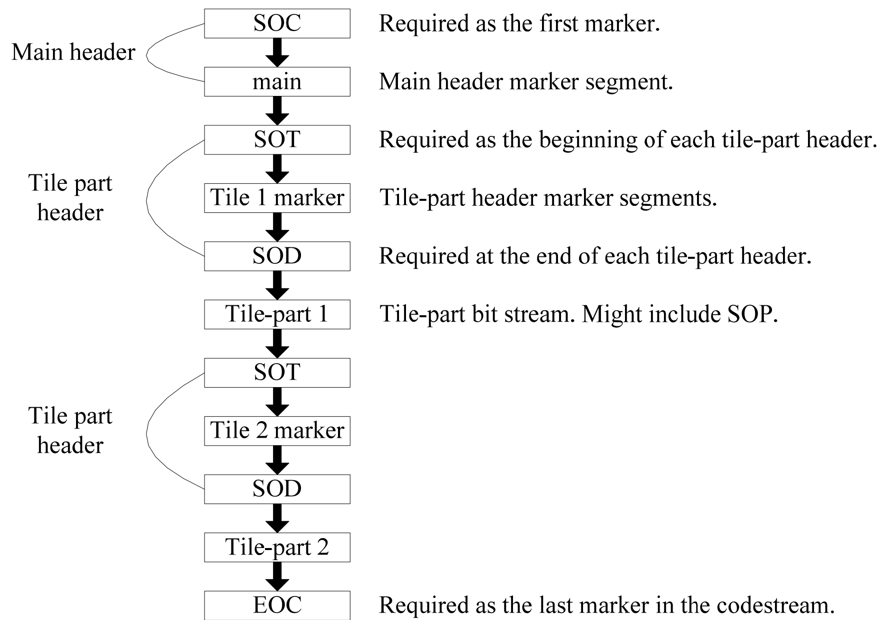


Figure 24: Construction of the codestream.

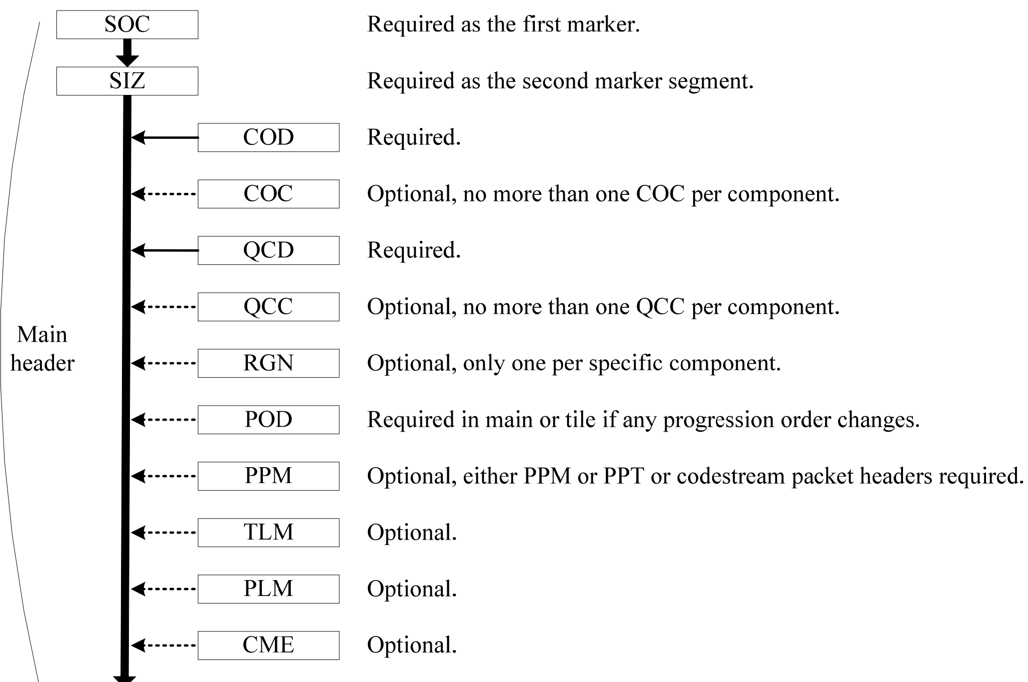


Figure 25: Construction of the main header.

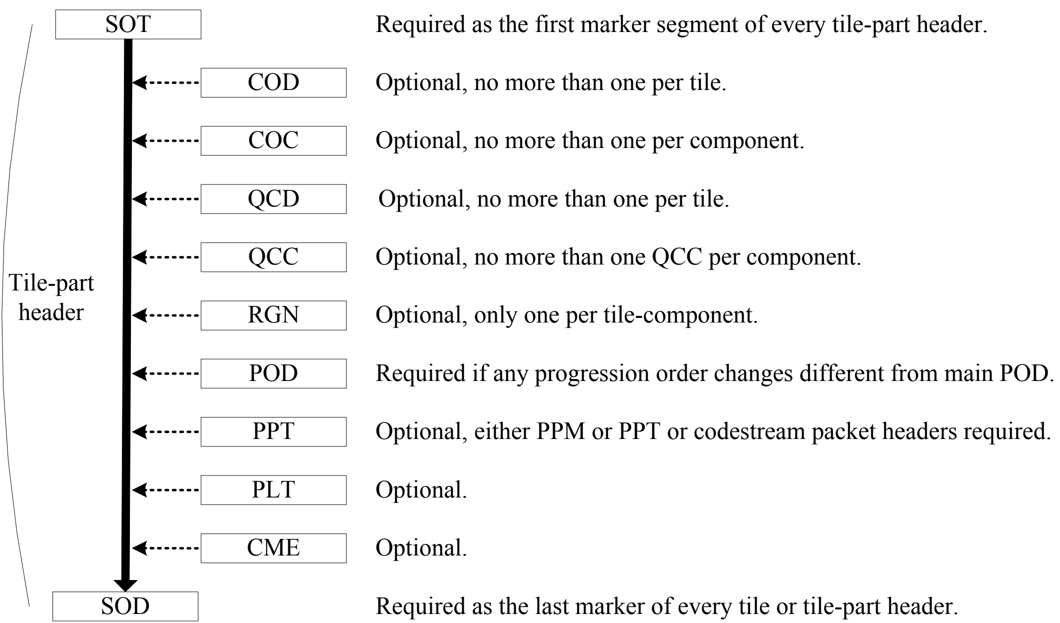


Figure 26: Construction of a tile-part header.

The real bit stream of a JPEG 2000 image is coded as the tile-parts in the codestream, therefore it's necessary to obtain the length of each tile-part before the extraction of the bit streams. There is one marker parameter that can be made use of for such purpose: the Psot parameter of the SOT marker segment. The structure of SOT market segments is showed in Figure 27.

16 bits	16 bits	16 bits	32 bits	8 bits	8bits
SOT	Lsot	Isot	Psot	TPsot	TNsot

Figure 27: Structure of SOT marker segment.

Psot: Length, in bytes, from the beginning of the first byte of this SOT marker segment of the tile-part to the end of the data of that tile-part. Figure 28 shows this alignment. Only the last tile-part in the codestream may contain a 0 for Psot. If the Psot is 0, this tile-part is assumed to contain all data until the EOC marker.

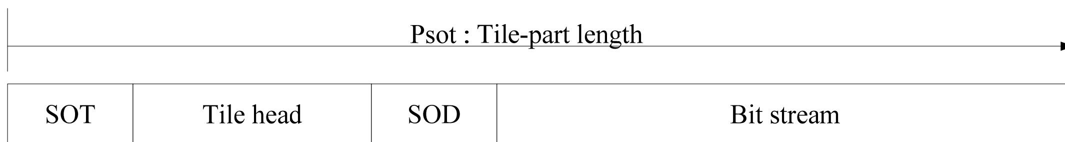


Figure 28: The length of a tile-part.

The length of the bit stream of a tile-part can be obtained by subtracting the length of SOT segment (12 bytes) from the length of the tile-part (the value of Psot parameter).

2.10. Conclusion

JPEG 2000 is a progressive coding system, which can be taken advantage of in our application to encrypt an image. Since only the high frequency part (the detail information) generally can not be utilized to identify an image, it seems practical to encrypt only the low frequency part so as to diminish the input data for the our encryption system for wireless transmission, where the ability to store and process data is limited.

Therefore the following steps can be considered: firstly, realize the extraction of the bitstreams of an JP2 file and the performance is evaluated. Afterwards, further experiments on the extraction of bitstreams based on resolution scalability of JPEG2000 images can be carried out to exam whether it’s feasible for our application. Thus arise several potential questions: how do we determine the levels of decomposition of an image so as to keep the size of the input to the encryption system as small as possible, and at the same time ensure the confidentiality of the image; is it possible to use other scalabilities such as quality scalability or component scalability to achieve our goals; etc.

All these will be experimented and further conclusion needed to be made.

Part 2. ECKBA

3. Context: chaotic cryptography

3.1. Cryptography

With a very long history, cryptography has been evolving ever since its birth several thousands years ago initialized by the Egyptians, but not until the break outs of the two world wars did cryptography claimed loud and clear its crucial role in protecting information security. Another event that profoundly accelerated the development of cryptography was the invention of computer, provoking demands for digital security in both local service and transmission to other ends.

Here, I introduce the concept of cryptography from [7]: Cryptography is the study of mathematical techniques related to aspects of information security such as confidentiality, data integrity, entity authentication, and data origin authentication.

There are 4 basic goals of cryptography to achieve information security, namely:

Confidentiality: aims at keeping the content of information from all unauthorized parties. There are numerous approaches to provide such service, ranging from physical protection to mathematical algorithms which render data unintelligible.

Data integrity: addresses and detects unauthorized alteration of data, including insertion, deletion and substitution.

Authentication: relates to identification of both parties in a communication and of the information being transmitted.

Non-repudiation: prevents an entity from denying previous commitments of actions.

A number of basic cryptographic primitives for information security are illustrated in Figure 29[7].

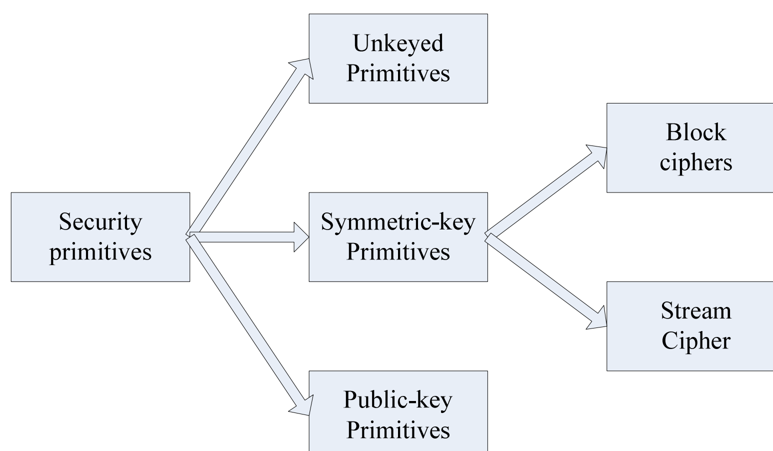


Figure 29: A taxonomy of cryptographic primitives.

Unkeyed primitives simply use techniques such as arbitrary length hash functions, one-way permutations and random sequences to protect the information security, rather than using the key

as a tool for security.

For symmetric-key primitives and public-key primitives, I will explain in more details in the following two sub-chapters.

3.2. Symmetric-key encryption

Roughly speaking, symmetric-key encryption means that, for each associated encryption/decryption key pair *(e, d)* [7], it is "computationally easy" to determine *d* knowing only *e*, and to determine *e* from *d*. An easy scheme of symmetric-key encryption is that *e=d*.

A two-party communication using symmetric-key encryption can be described by the block diagram of Figure 30. Alice and Bob denote the two communicating parties and Adersary denotes the illegal party who intends to crack the encrypted information.

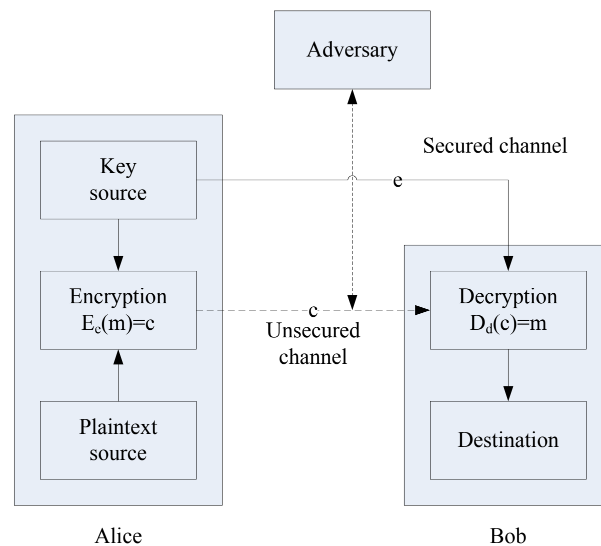


Figure 30: Two-party communication using encryption, with a secure channel for key exchange. The decryption key d can be efficiently computed from the encryption key e.

One of the major issues with symmetric-key systems is to find an efficient method to agree upon and exchange keys securely. This problem is referred to as the key distribution problem. It's assumed that all parties know the set of encryption/decryption transformation, for example, the encryption scheme. The only information required to be kept secret is the key *d*, which, in symmetric systems, means that the key *e* must also be kept secret, as d can be deduced from *e*. The classification of different symmetric-key encryption is showed in Figure 31.

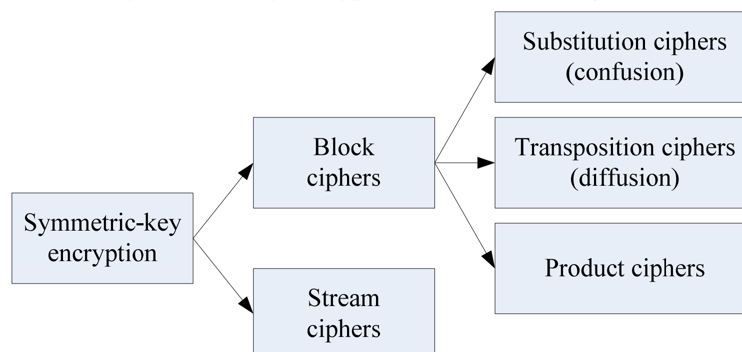


Figure 31: Classification of Symmetric-key encryption.

A block cipher is an encryption scheme which breaks up the plaintext messages to be transmitted into strings (called blocks) of a fixed length and decrypts one block at a time. There are three types of block ciphers:

Substitution ciphers replace symbols or groups of symbols by other symbols or groups of symbols.

Transposition ciphers simply permute the symbols in a block.

Simple substitution and transposition ciphers individually do not provide a very high level of security. To obtain strong ciphers it's necessary to combine the two types of ciphers together, forming product ciphers, among which come out several most practical and effective symmetric-key systems.

If a product cipher is made up of a composition of more than two transformations, each of which is either a substitution cipher or a transposition cipher, we call the composition of a substitution and a transposition a round. A substitution in a round is said to add confusion to the encryption process whereas a transposition is said to add diffusion.

The other type of symmetric cipher is called stream ciphers, which can be viewed as very simple block ciphers with block length equal to one. One advantage of stream ciphers is that the encryption transformation can change for each symbol of plaintext. Thus stream ciphers have no error propagation, which makes it useful in transmission channels where high probability of errors occurs.

3.3. Public-key encryption

For a pair of associated encryption/decryption transformations (E_e, D_d), if each pair has the property that by knowing E_e it is infeasible, given a random ciphertext c, to find the message m such that $E_e(m)=c$. This property implies that given e it is infeasible to determine the corresponding decryption key d.

Figure 32 shows how this scheme works between the two parties in the communication: Bob selects the key pair (e, d) and he sends the encryption key e called the public key to Alice over any channel but keeps the decryption key d called private key secure and secret. Alice may subsequently send a message m to Bob by applying the encryption transformation determined by Bob's public key to get $c=E_e(m)$. bob decrypts the ciphertext c by applying the inverse transformation *Dd* uniquely determined by d.

Different from symmetric-key ciphers, in public-key scheme the encryption key is transmitted to Alice over an unsecured channel, for example, the same channel where the ciphertext is transmitted. The encryption key e is thus called the public key since it need not be kept secret. Any entity can send encrypted messages to Bob, who is the only one capable of decrypting the messages.

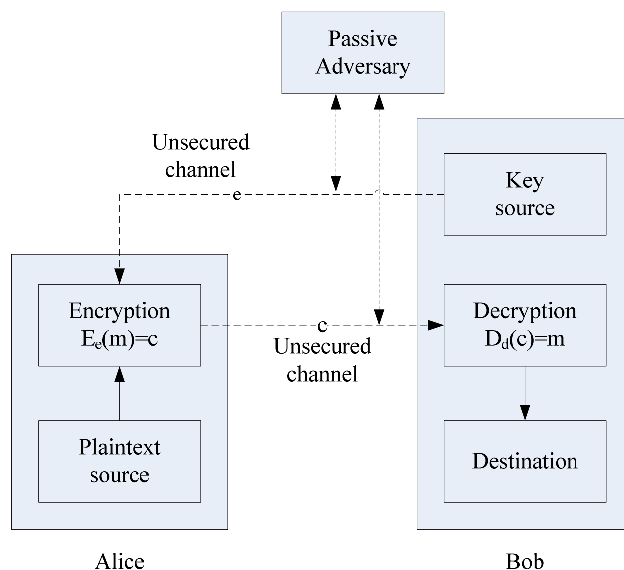


Figure 32: Public-key encryption and decryption communication.

3.4. Chaos based cryptography

Chaotic systems are non linear systems or systems with very small linearity. They have very high sensitivity to even a tiny change of the initial condition or of the input parameters, which means two results of huge diverge in the same chaotic system may only caused by a small change of the values aforementioned. This property is interesting for cryptography [8].

Generally chaotic ciphers are classified into two categories: stream ciphers and block ciphers. Stream ciphers are generated by chaotic systems as a pseudo-random key-streams, it is then used to mask the plaintext. Block ciphers are generated by iterating/counter-iterating chaotic systems multiple times using the plaintext and/or the secret keys as initial conditions and/or control parameters. For example, ECKBA to be explained in details belongs to the category of block ciphers.

There are other ways of generating chaotic ciphers, such as chaotic S-box based block ciphers and searching based chaotic ciphers [9].

4. Enhanced Chaotic Key-Based Algorithm

4.1. General idea

This ECKBA (Enhanced 1-D Chaotic Key-Based Algorithm) algorithm is brought forward in [10], based on the work of CKBA (Chaotic Key-Based Algorithm) proposed by Yen and Guo in [11] .

CKBA is based on a one-dimensional Logistic map, which, has been shown unavoidably susceptible to chosen/known-plaintext attacks and that the claimed high security against ciphertext-only attacks was overestimated by the authors. In addition, the Logistic map yields unbalanced output.

In ECKBA, several steps were applied for enhancement [10]:

(1) The key size was increased to 128-bits.

(2) The 1-D Logistic map in original CKBA was substituted by a piecewise linear chaotic map (PWLCM) to improve the balance property of the chaotic map.

(3) A pseudo-random permutation generator (PRPG) based on the new chaotic map was introduced as an additional component in encryption and decryption process, forming a permutation box (P-box) and adding diffusion to the system.

(4) A more complex substitution box (S-box) was applied.

(5) Multiple rounds for encryption and decryption process were introduced.

In essence, the basic idea of ECKBA is not complex. The bytes representing pixel values of a plain-image are organized into a one-dimension sequence and processed one byte by one byte: Each plain-byte is masked with the previous cipher-byte (the cipher-block chaining, CBC), before entering the SP-network, which consists of an S-box and a P-box, for multiple rounds. See Figure 33.

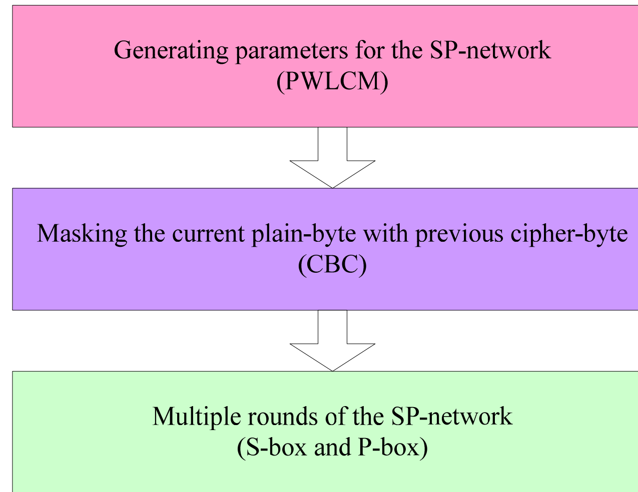


Figure 33: Basic idea for processing a byte of the image data.

4.2. Three main blocks

4.2.1. PWLCM

The Piecewise Linear Chaotic Map (PWLCM) is given as followed:

$$\begin{aligned} x(n) &= \mathcal{C}_\mu(x(n-1)) \\ &= \begin{cases} x(n-1)\cdot\frac{1}{\mu}, & \text{if } x(n-1)\in[0,\mu); \\ (x(n-1)-\mu)\frac{1}{0.5-\mu}, & \text{if } x(n-1)\in[\mu,0.5]; \\ \mathcal{C}_\mu(1-x(n-1)), & \text{if } x(n-1)\in[0.5,1); \end{cases} \end{aligned}$$

where the digital chaotic pseudo random sequence $x(n) \in (0, 1)$, $n \geqslant 0$, with the initial condition $x(0)$ and the positive real control parameter $\mu \in (0, 0.5)$. Thus there are two parameters for this map, namely the initial condition of the pseudo random sequence $x(0)$ and the control parameter μ.

4.2.2. CBC

In this block, the operation on the plain-byte is simple: implement bitwise exclusive-or (XOR) with the previous cipher-byte:

cipherbyte(i) = plainbyte(i) XOR cipherbyte(i-1)

Each cipher-byte is related to the immediately previous cipher-byte, forming the so-called Cipher Block Chaining (CBC) [10].

4.2.3. SP-network

- S-box

S-box, where 'S' stands for Substitution, is used for substituting the original byte to add confusion of the plain-image. The S-box transformation σ_r and its inverse σ_r^{-1} of ECKBA is defined as followed:

$$\sigma_r(u,v) = \begin{cases} u \oplus v, & \text{if } r \text{ is even;} \\ u + v \bmod 256, & \text{if } r \text{ is odd,} \end{cases}$$

$$\sigma_r^{-1}(u,v) = \begin{cases} u \oplus v, & \text{if } r \text{ is even;} \\ u - v \bmod 256, & \text{if } r \text{ is odd,} \end{cases}$$

where u and v are two bytes. To make sure the results of the transformation stay within the upper bound of one byte, modulo $u+v$ should be implemented.

- P-box

A pseudo-random permutation generator (PRPG) is introduced to add diffusion to the plain-image. For a byte, the number of all the possible permutations of its bits including itself is 8!(equals to 8*7*6*5*4*3*2*1). These are called permutation group of degree 8 and denoted by S_8. Denote a particular permutation by π_i, with $i \in (0, 8!)$ the index in the full symmetric group S_8 sorted in lexicographical Cartesian order. A demonstration example is given below:

Suppose $\pi \in S8$, the standard form of which is denoted as:

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 4 & 6 & 7 & 1 & 3 & 8 & 2 & 5 \end{pmatrix}$$

A byte B is denoted as B = $[b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8]$, where b_i is the ith MSB bit of the byte (the ith bit from the left of the byte). The forward permutation of B is π(B)=$[b_4, b_6, b_7, b_1, b_3, b_8, b_2, b_5]$ and the inverse permutation of B is π^{-1}(B)= $[b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8]$.

4.3. Encryption algorithm

The corresponding relationship between the ECKBA encryption algorithm and the flow diagram is given in Figure 34, where x_{i32} denotes a 32-bit integer variable and x denotes its normalized floating-point representation which corresponds to the relevant real interval, etc.

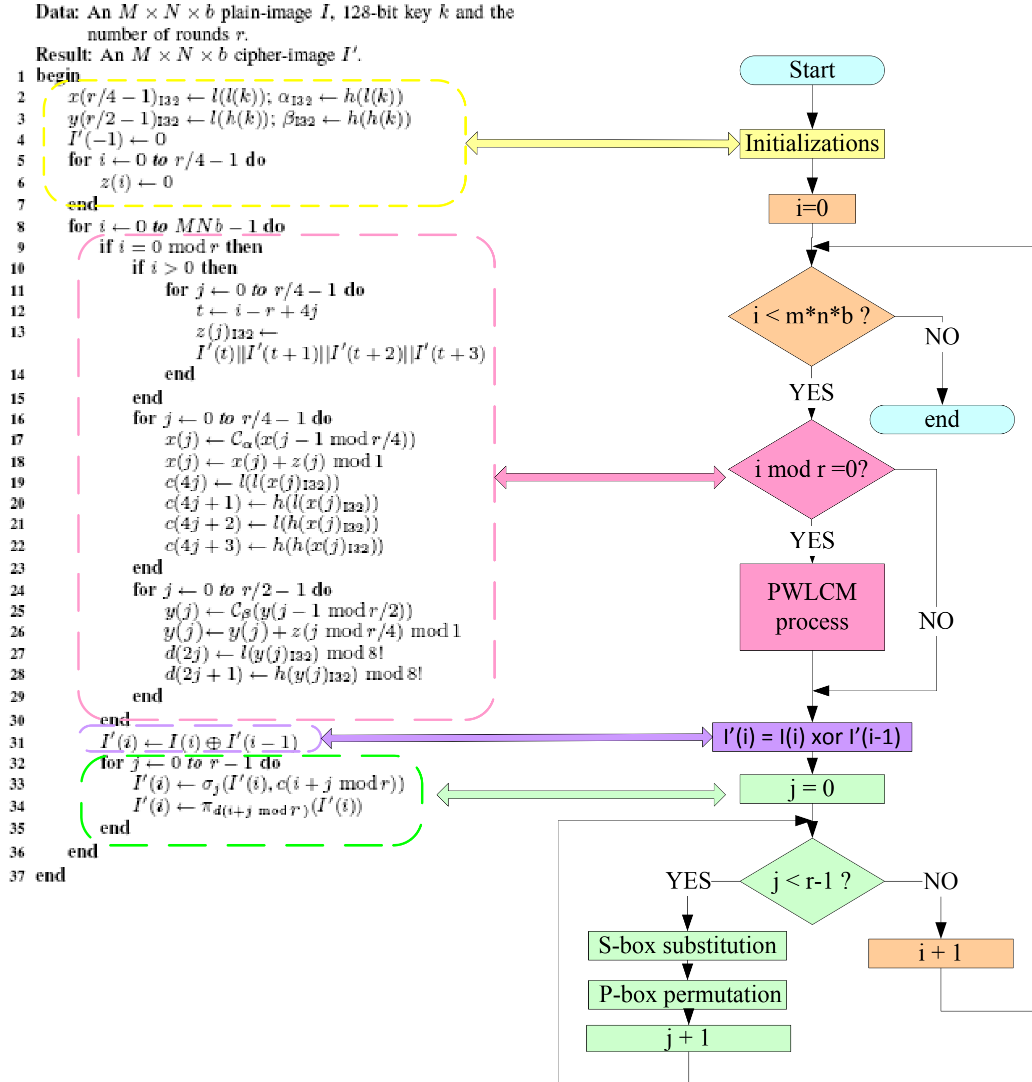


Figure 34: ECKBA encryption algorithm and its corresponding flow diagram.

Firstly, initializations of control parameters, initial conditions and other variables are carried out as enclosed by the yellow dot line corresponding to the yellow part in the flow diagram.

Secondly, a loop of M*N*b (the image data size) times is carried out, with M, N standing for the width and the height of the plain-image and b standing for the number of bytes representing one pixel. In each loop, one byte of the plain-image is processed at a time. Here, what should be paid attention to is that the basic unit to be processed in one loop is a byte, not a pixel value. A pixel value is represented by bits (8 bits make up a byte), which may be either an integer or a real value, for example, 8 bits/pixel of a BMP format image or 5.6bits/pixel of a JP2 format image. A byte is the basic unit of the storage device in a computer. Therefore it would be simpler and more convenient to use a byte as the basic unit of the process. Sometimes it’s not practical to calculate the image data size by M*N*b, for example, for a JP2 file the value of bits per pixel is usually real and the organization of the image data is not based on the width and height of the image. In such case, the number of loop times should be substituted, one example of which is the JP2 file image data extraction in the up-coming part of the report.

Within the loop, which is controlled by the orange part of the flow diagram, there are mainly

3 blocks performing different tasks as aforementioned. The codes enclosed by pink dot line (corresponding to the part in pink color in the flow diagram) generate parameters for the upcoming S-box (Substitution) and P-box (Permutation), by using the Piecewise Linear Chaotic Map (PWLCM). The codes enclosed in purple dot line (corresponding to the purple part in the flow diagram) simply masks a plain-byte with the cipher-byte generated in the previous loop. The codes enclosed in green dot line (corresponding to the part in green color of the flow diagram) perform the aforementioned Substitution(S-box) and Permutation (P-box) for number of r rounds.

In the algorithm, the 128-bit key k provides initial conditions and control parameters for the Piecewise Linear Chaotic Map to generate two pseudo-random chaotic sequences $\{x\}$ and $\{y\}$, each of which is used as control parameters for the substitution (S-box) and the permutation (P-box) respectively. For an n-bit segment x, denote its low significant half by $l(x)$ and its high significant half by $h(x)$.The structure of the key is as followed:

$h(h(k))$: 32 bits	$l(h(k))$: 32 bits	$h(l(k))$: 32 bits	$l(l(k))$: 32 bits
Control parameter of PWLCM to generate $\{y\}$	Initial condition of the pseudo-random chaotic sequence $\{y\}$	Control parameter of PWLCM to generate $\{x\}$	Initial condition of the pseudo-random chaotic sequence $\{x\}$

There are 5 arrays in the algorithm needed to be explained, of which the structures are shown below in Figure35.

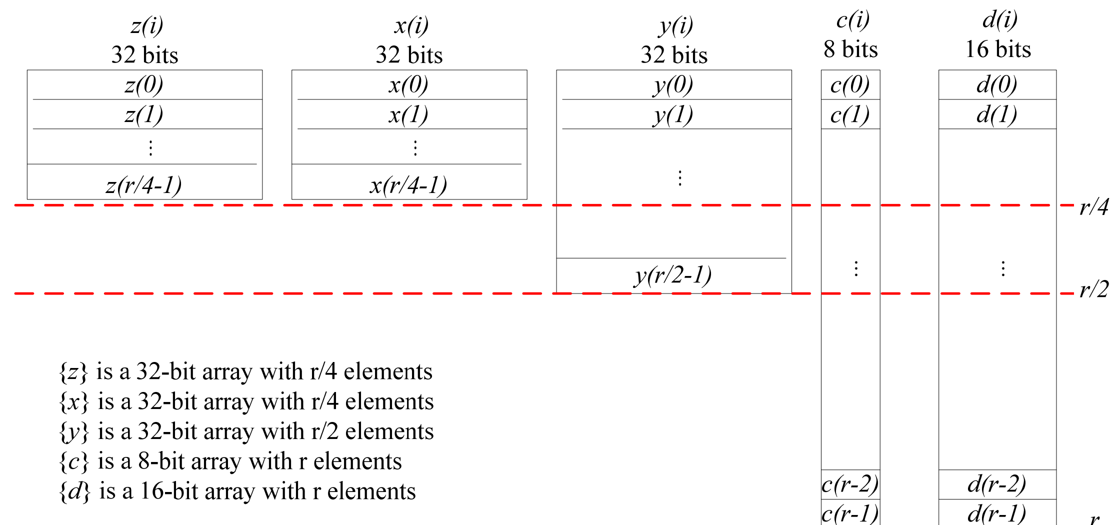


Figure 35: Structures of the arrays used in the algorithm.

The pseudo-random chaotic sequences $\{x\}$ and $\{y\}$ are treated as rings, therefore the initial conditions $l(l(k))$ of $\{x\}$ and $l(h(k))$ of $\{y\}$ are placed in $x(r/4-1)$ and $y(r/2-1)$ as they are the preceding elements of $x(0)$ and $y(0)$ respectively.

$\{z\}$ stores the last r newly generated cipher-bytes, in the way that 4 consecutive cipher-bytes form an element of z, see line 13 of encryption algorithm in figure 25. $\{z\}$ will be added to corresponding element of $\{x\}$ and $\{y\}$ (line 18 and line 26 of encryption algorithm), making a cipher-bytes based chaotic chain.

$\{x\}$ and $\{y\}$, the pseudo-random chaotic sequences, store the results of the Piecewise Linear

Chaotic Map (line 17 and line 25 of encryption algorithm). They are used to generate the elements of $\{c\}$ and $\{d\}$, which are the input parameter of S-box and P-box respectively. what should be carefully of is that the 32-bit segments *x(i)* and *y(i)* should be normalized by dividing 2^{32} before their entering the PWLCIM function as input parameters.

$\{c\}$ is input parameter of the substitution function σ , in which *c(i)* is *XOR*-ed with or added to the other input parameter, the current plain-byte being processed. Thus the elements' length of *c* should also be 8 bits as the length of the current plain-byte is 8 bits.

$\{d\}$, storing the indexes of the permutation group S_8, is the input parameter of the permutation function π. The range of these index is from 0 to 8!-1, which needs at least 2 bytes for storage resulting the 2-byte-length of the element of $\{d\}$.

4.4. Decryption algorithm

The inverse transformations to decrypt an encrypted image differ very little from the encryption algorithm [10] and the data structures of the arrays are the same. See Figure 36 with the differences marked out.

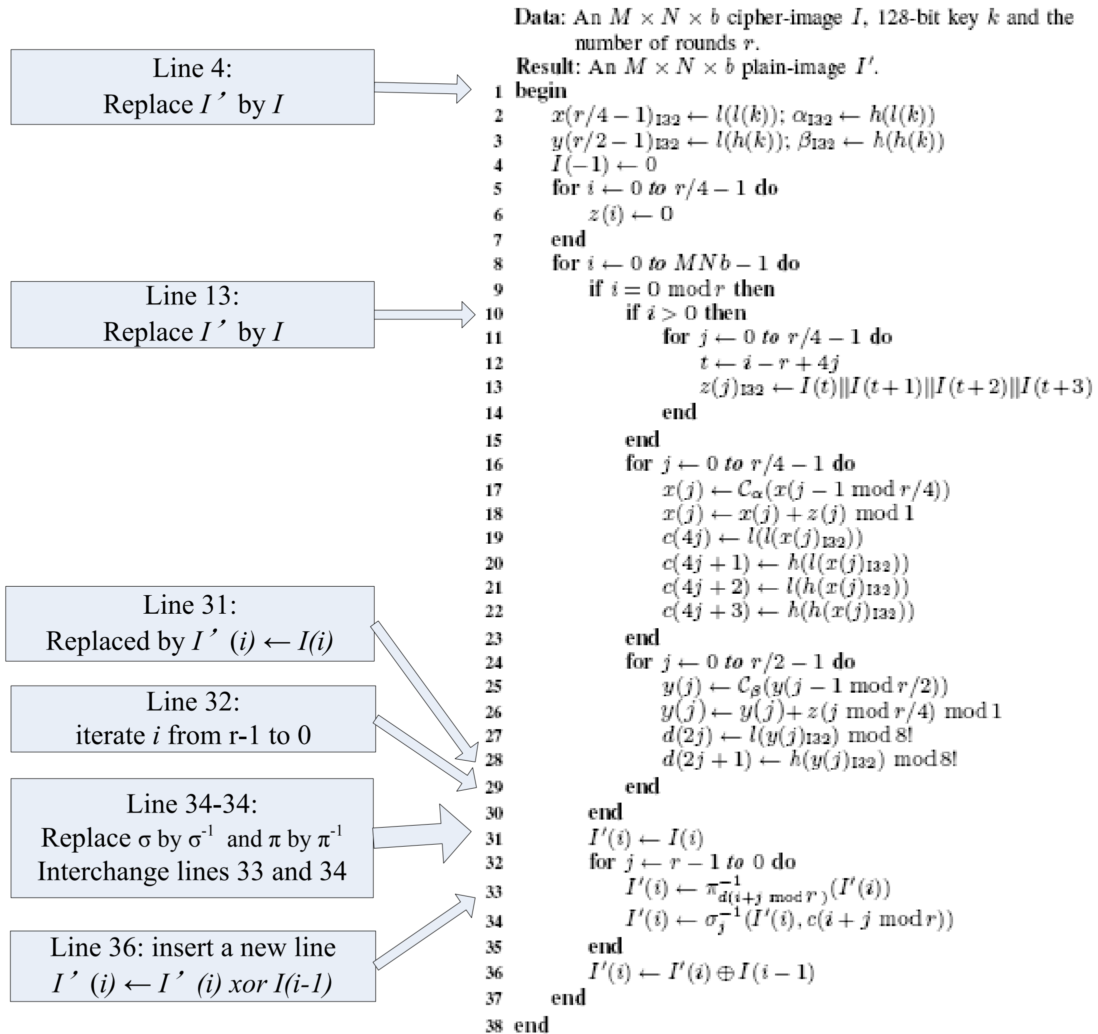


Figure 36: Decryption algorithm and changes compared with encryption.

Part 3. Experiments

5. Programs

Before my work, the algorithms of ECKBA dealing with BMP format files have been realized in MATLAB but the performance was not as good as expected. In order to achieve better performance of the algorithms, C++ is used for my experiments. Firstly, implementing ECKBA algorithms with BMP files was carried out to make comparisons of the running time performance with the MATLAB programs. Then implementing ECKBA with JP2 files was carried out.

Therefore two projects have been created for implementing ECKBA with BMP files and with JP2 files respectively. The names of these projects are: ECKBA_BMPv1 and ECKBA_JP2v1. Both the projects use the same class ECKBA for encryption and decryption, the differences are the images types to be processed. Project ECKBA_BMPv1 uses class ImgBMP to deal with the BMP format images whereas project ECKBA_JP2v1 uses class ImgJP2 to deal with the JP2 format images. The corresponding main functions (main.cpp) are also different considering the specific file formats.The headers and source codes are organized in the annexes listed in Table 2.

Table 2: Organization of the projects

Common files / Project names	Common headers: confg.h	Annex A
	Class ECKBA: ECKBA.h ECKBA.cpp	
ECKBA_BMPv1	Class ImgBMP: ImgBMP.h ImgBMP.cpp main.cpp	Annex B
ECKBA_JP2v1	Class ImgJP2: ImgJP2.h ImgJP2.cpp main.cpp	Annex C

Refer to Annex D about the syntax of a BMP file.

The project dealing with JP2 files assumes that the jp2 image files have their codestream box as the last box in the file and if it's not the case the program exits without any encryption and decryption.

The formats of input command line of both ECKBA_BMPv1 and ECKBA_JP2v1 are the same as specified in Table 3. Note that the programs do not exam the format of the input command line, thus the users should pay attention themselves.

Table 3: Instructions of the command line of the programs

Order	Parameter name	Instruction
1	target_image_name	The name of the image to be encrypted and decrypted. The suffix of the image file should be included.(e.g. boat.bmp lena.jp2)
2	encrypted_image_name	The name of the encrypted image file. The file will be created if it doesn't exist, otherwise the contents of the file will be rewrote. The suffix of the image file should be included.
3	decrypted_image_name	The name of the decrypted image file. The file will be created if it doesn't exist, otherwise the contents of the file will be rewrote. The suffix of the image file should be included.
4	rounds	Number of rounds of the SP-network. It should be an integer.
5	initial_condition_of_x	Initial condition of the pseudo-random chaotic sequence $\{x\}$. It should be an integer $\in [0, 2^{32}-1]$.
6	initial_condition_of_y	Initial condition of the pseudo-random chaotic sequence $\{y\}$. It should be an integer $\in [0, 2^{32}-1]$.
7	alpha	Control parameter of PWLCM for $\{x\}$. It should be an integer $\in [0, 2^{32}-1]$.
8	beta	Control parameter of PWLCM for $\{y\}$. It should be an integer $\in [0, 2^{32}-1]$.

6. Results

The projects are executed under Windows Operating System on a 1.86GHz Intel® Pentium® M processor. Refer to Annex E for the encrypted and decrypted image.

Running time of the ECKBA algorithms to process BMP format images written in C++ and using MATLAB has been listed in Table 4 and Table 5 below. Curves based on the data in Table 4 and Table 5 are shown from Figure 37 to Figure 40.

Table 4: Running time of encryption in C++ and MATLAB

Image	M×N×b	Sum in bytes	*r*=4 Enc.time in seconds	*r*=8 Enc.time in seconds	
			C++	C++	MATLAB
boat	128×96×1	12288	0.088	0.156	13.641
mandril	176×144×1	25344	0.166	0.328	27.328
clown	256×256×1	65536	0.426	0.854	70.453
barb	512×512×1	262144	1.745	3.416	198.328
lena_lumi	433×433×3	562467	3.896	7.713	304.813

Table 5: Running time of decryption in C++ and MATLAB.

Image	M×N×b	Sum in bytes	r=4 Dec.time in seconds	r=8 Dec.time in seconds	
			C++	C++	MATLAB
boat	128×96×1	12288	0.078	0.156	12.969
mandril	176×144×1	25344	0.162	0.317	25.959
clown	256×256×1	65536	0.422	0.823	66.485
barb	512×512×1	262144	1.692	3.318	187.453
lena_lumi	433×433×3	562467	3.785	7.437	263.719

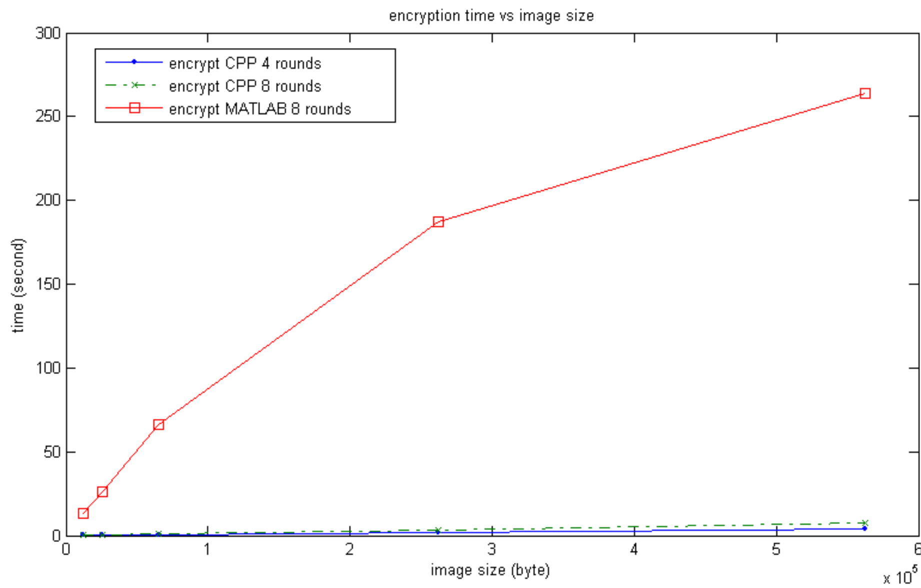


Figure 37: Comparisons of encryption time between C++ and MATLAB.

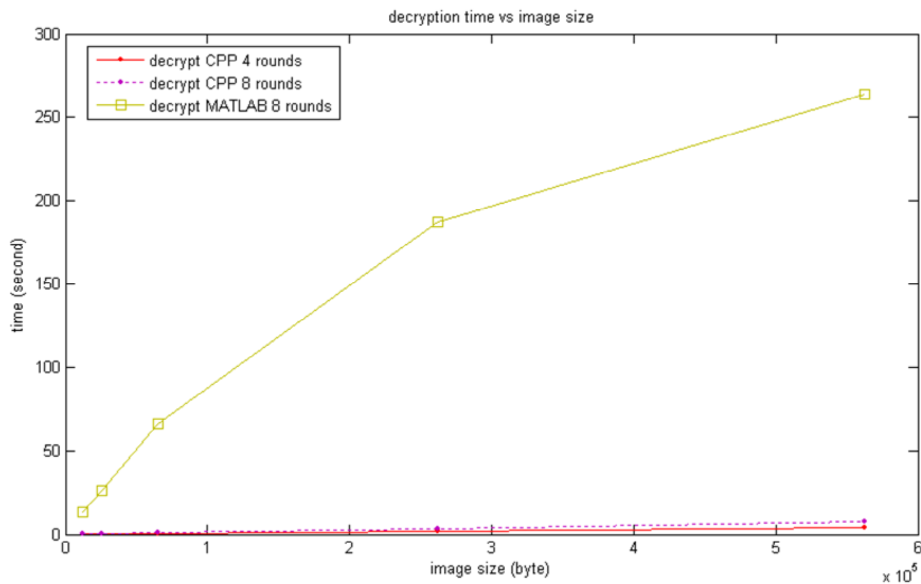


Figure 38: Comparisons of decryption time between C++ and MATLAB.

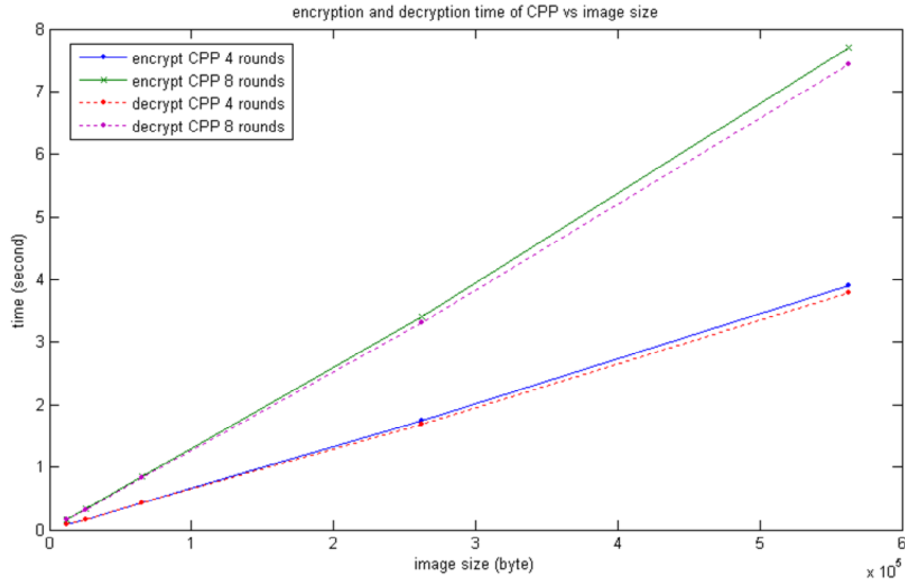


Figure 39: Comparisons of encryption and decryption time in C++.

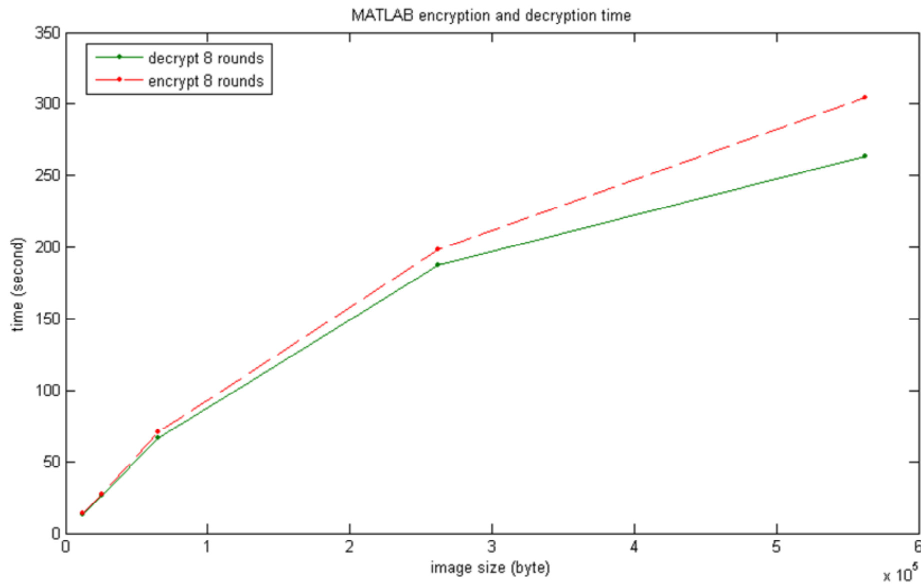


Figure 40: Comparisons of encryption and decryption time in MATLAB.

From the above data and curves, it's obvious to observe that significant improvement has been achieved by implementing the algorithms in C++ compared to that of in MATLAB. The running time for both encryption and decryption in C++ are generally linear based on the size of the image pixel data, whereas in MATLAB it's no the case. Moreover, decryption time does not increase as fast as encryption in MATLAB, which, however, should increase at the same pace. On the contrary, the curves of encryption and decryption time are basically as should be expected in C++.

Results of encryption and decryption time of project JP2v1 is shown in Table 6 and the corresponding curves are shown in Figure 41. It should be pointed out that the last testing image is

not the same as used in the BMPv1 project. Its original BMP format file is 1024×1024×3. The reason why the bit-depth of the images are not give is that the images' bit-depths are real values instead of integers after compressed by the JPEG 2000 coding system.

Table 6: Encryption and decryption time of JP2v1

Image	M×N	Sum in bytes	*r*=4 time in seconds		*r*=8 time in seconds	
			encryption	decryption	encryption	decryption
boat	128×96	8360	0.0502	0.0494	0.0968	0.1
mandril	176×144	21526	0.1346	0.1312	0.247	0.2562
clown	256×256	47485	0.281	0.2814	0.5468	0.5436
barb	512×512	204219	1.219	1.2094	2.3656	2.3654
lena	1024×1024	597546	3.578	3.5464	6.9374	6.9594

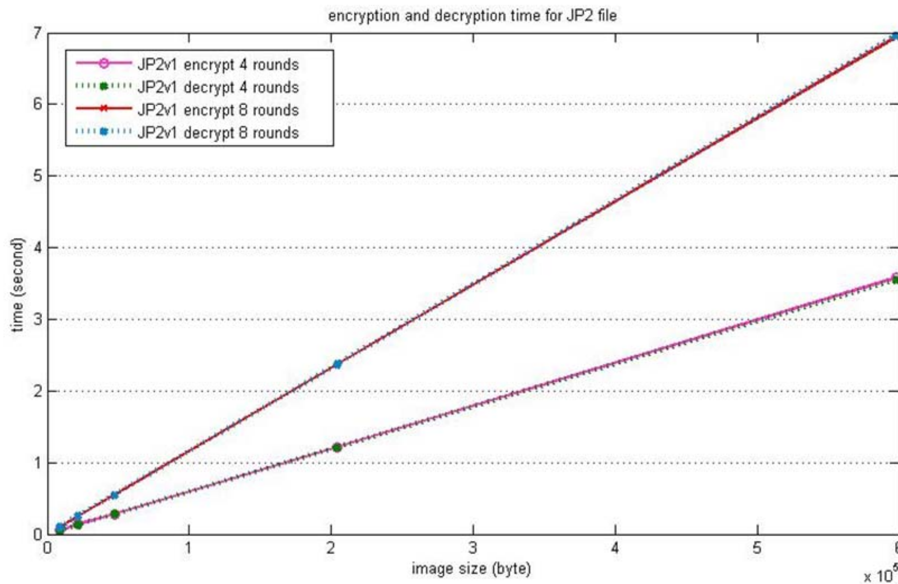


Figure 41: Encryption and decryption time for JP2 files.

As can be seen from Figure 40, the time for both encryption and decryption are linear to the size of the image buffer. Besides, the curve of encryption time and that of the decryption time with the same number of rounds are more ideal than that of the BMP situation.

Conclusions

The main work of my project is to implement the ECKBA algorithm taking into account the progressive image coding system JPEG 2000. The results of the experiments prove that ECKBA is a linear function based on the size of the image data, thus efforts need to be made to minimize this size, which naturally leads to consideration of the JPEG 2000 images.

The extraction the whole codestream of a JPEG 2000 image has been realized but this is not complete as no further work on extracting particular part of the codestream has been done. Thus the following tasks are suggested if any further exploration on this subject is to be continued.

Exam different codestream extraction schemes: resolution scalability and quality scalability.

Determine to which extent needs the codestream to be extract to balance between the simplicity of the encoding and decoding process and the consumption of the time for encryption and decryption.

Analyze this crypto-compression scheme.

References

[1] Majid Rabbani, Paul W. Jones. *Digital image compression techniques*. SPIE, 1991.

[2] David S. Taubman, Micheal W. Marcellin. *JPEG2000 image compression fundamentals, standards and practice*. KAP, 2002.

[3] Athanassios Skodras, Charilaos Christopoulos, Touradj Ebrahimi. The JPEG2000 still image compression standard. In *IEEE, Signal Processing Magazine,* Vol 18, Issue 5, pp. 36-58, September 2001.

[4] Charilaos Christopoulos, Athanassios Skodras, Touradj Ebrahimi. The JPEG2000 still image coding system: an overview. In *IEEE, Transactions on Consumer Electronics.* Vol. 46, No. 4, pp. 1103-1127, November 2000.

[5] Majid Rabbani, Diego Santa Cruz. The JPEG2000 still Image compression standard.

[6] D. Taubman. High performance scalable image compression with EBCOT. In *IEEE Trans. Image Processing*, Vol.9, No.7, pp.1158-1170, July 2000.

[7] A. Menezes, p. van Oorschot, S. Vanstone. *Handbook of applied cryptography*. CRC Press, Inc. 1997.

[8] D. Caragata, S. EL ASSAD. *Development of some encryption/decryption algorithms using chaos for secure communication systems.* Internal Research Report of IREENA, 2006.

[9] Shujun Li. *Analyses and new designs of digital chaotic ciphers.* PHD. Thesis, Xi'an Jiaotong University, 2003.

[10] Daniel Socek, Shujun Li, Spyros S. Magliveras, Borko Furht. Enhanced 1-D chaotic key-based algorithm for image encryption. In *SECURECOMM*, 2005, P406-P407.

[11] J. –C. Yen and J.-I.Guo. A new chaotic key-based design for image encryption and decryption. In *Proceeding of 2000 IEEE International Conference on Circuits and Systems (ISACS 2000)*, volume 4, pages49-52, 2000.

[12] *JPEG2000 Part I Final Committee Draft Version 1.0 (ISO/IEC FCD15444-1)*, ISO/IEC JTC 1/SC 29/WG 1 N1646R, March, 2000.

[13] *Annex I JP2 file format syntax*, ISO/IEC 15444-1:2004 | ITU-T Rec. T.800.

Annex A. Common header file of the projects

This annex gives the common header file config.h of the projects.

```
///////////////////////////////////
// config.h
///////////////////////////////////
#include <iostream.h>
#include <string.h>
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#define    WORD       unsigned short
#define    DWORD      unsigned long
#define    LONG       unsigned long
#define    BYTE       unsigned __int8

#define SizeofWORD sizeof(WORD)
#define SizeofDWORD sizeof(DWORD)
#define SizeofBYTE sizeof(BYTE)
#define SizeofLONG sizeof(LONG)
```

Annex B. Source codes related to class ImgBMP

This annex gives the header file ImgBMP.h and source files including ImgBMP.cpp and the corresponding main.cpp of the project ECKBA_BMPv1

```
//////////////////////////////////////////////////////////////////////
// ImgBMP.h: interface for the ImgBMP class.
//
//////////////////////////////////////////////////////////////////////

#if !defined(AFX_IMGBMP_H__F06B8888_C082_41E1_BD41_27B0B17E2736__INCLUDED
_)
#define AFX_IMGBMP_H__F06B8888_C082_41E1_BD41_27B0B17E2736__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include "confg.h"

typedef struct BITMAPFILEHEAER{
        WORD      bfType;
        DWORD     bfSize;
        WORD      bfReserved1;
        WORD      bfReserved2;
        DWORD     bfOffBits;
    }   BITMAPFILEHEADER;

typedef struct BITMAPINFOHEADER{
        DWORD biSize;
        LONG biWidth;
        LONG biHeight;
        WORD biPlanes;
        WORD biBitCount;
        DWORD biCompression;
        DWORD biSizeImage;
        LONG biXPelsPerMeter;
        LONG biYPelsPerMeter;
        DWORD biClrUsed;
        DWORD biClrImportant;
    } BITMAPINFOHEADER;

/*  typedef struct RGBQUAD{
        BYTE rgbBlue;
```

```
        BYTE rgbGreen;
        BYTE rgbRed;
        BYTE rgbReserved;
    } RGBQUAD;
*/

class ImgBMP
{
public:
    ImgBMP();
    virtual ~ImgBMP();

public:
    ImgBMP(char *FileName);        // constructor with an BMP file name
    DWORD GetPixelArrySize();    // return the size of the buffer storing image pixel data
    BYTE* GetPixelArry();            // return the pointer pointing to the buffer storing image pixel
data
    void DisplayPixelArry();      // to display the image pixel data
    void DisplayRGBQUADArry(); // to display the RGBQUAD data of the iamge
    void DisplayInfo();               // to display the header info. of the image
    // write the header info., RGBQUAD and the pixel data stroed in buffer to a file assigned by
FileName
    void Wt2File(char *FileName, BYTE *buffer);

protected:
    void LoadFileInfo(FILE *stream);        // to load the header info. of file stream
    void LoadRGBQUADArry(FILE *stream); // to load the RGBQUAD info. of file stream
    void LoadPixelArry(FILE *stream);      // to load the pixel data of file stream

    void WtHd2File(FILE *stream);             // to write the header info. to file stream
    void WtPixel2File(FILE *stream, BYTE *buffer);    // to write the buffer storing pixel data to
file stream

    int BMPHeaderSize;           // size of the BMP header
    int QUADArrySize;             // size of the QUAD array

    char *imgName;                   // name of the image

    BYTE *pQUADArry;               // buffer to stroe the QUAD array
    BYTE *pPixelArry;             // buffer to store the pixel data

    BITMAPFILEHEADER fh;        // data structure to store the bitmap file header
    BITMAPINFOHEADER info;    // data structure to store the bitmap info header
};
```

```
#endif
// !defined(AFX_IMGBMP_H__F06B8888_C082_41E1_BD41_27B0B17E2736__INCLUDED_)
//////////////////////////////////////////////////////////////////////
// ImgBMP.cpp: implementation of the ImgBMP class.
//
//////////////////////////////////////////////////////////////////////

#include "ImgBMP.h"

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////

ImgBMP::ImgBMP()
{

}

ImgBMP::~ImgBMP()
{
	free(pPixelArry);
	free(pQUADArry);
}

ImgBMP::ImgBMP(char *FileName)
{
	//initialize header size
	BMPHeaderSize=54;

	//initialize image name
	imgName=FileName;

	//open the BMP file in read and binary mode
	FILE *stream = fopen(imgName, "rb");
	//printf("the file name is:%s\n", FileName);
	if(stream == NULL){
		printf("Faiure in ImgBMP constructor: file %s can not be opened! Exiting....\n",
FileName);
		exit(-1);
	}
	//load the file header and the info header of the image
	LoadFileInfo(stream);
//	DisplayInfo();
```

```
	//load the RGBQUAD array of the image if there is one
	LoadRGBQUADArry(stream);
//	DisplayRGBQUADArry();

	//load the pixel array of the image
	LoadPixelArry(stream);
//	  DisplayPixelArry();

	fclose(stream);
}
// ------------------------------------------------------------------------------------
// LoadFileInfo(FILE *stream)
// to load both the BITMAPFILEHEADER and the BITMAPINFOHEADER into fh and info
// ------------------------------------------------------------------------------------
void ImgBMP::LoadFileInfo(FILE *stream)
{
	//load the BITMAPFILEHEADER of the image
	fread(&fh.bfType, sizeof(fh.bfType), 1, stream);
	fread(&fh.bfSize, sizeof(fh.bfSize), 1, stream);
	fread(&fh.bfReserved1, sizeof(fh.bfReserved1), 1, stream);
	fread(&fh.bfReserved2, sizeof(fh.bfReserved2), 1, stream);
	fread(&fh.bfOffBits, sizeof(fh.bfOffBits), 1, stream);

	//load the BITMAPINFOHEADER of the image
	fread(&info.biSize, sizeof(info.biSize), 1, stream);
	fread(&info.biWidth, sizeof(info.biWidth), 1, stream);
	fread(&info.biHeight, sizeof(info.biHeight), 1, stream);
	fread(&info.biPlanes, sizeof(info.biPlanes), 1, stream);
	fread(&info.biBitCount, sizeof(info.biBitCount), 1, stream);
	fread(&info.biCompression, sizeof(info.biCompression), 1, stream);
	fread(&info.biSizeImage, sizeof(info.biSizeImage), 1, stream);
	fread(&info.biXPelsPerMeter, sizeof(info.biXPelsPerMeter), 1, stream);
	fread(&info.biYPelsPerMeter, sizeof(info.biYPelsPerMeter), 1, stream);
	fread(&info.biClrUsed, sizeof(info.biClrUsed), 1, stream);
	fread(&info.biClrImportant, sizeof(info.biClrImportant), 1, stream);
}
// ------------------------------------------------------------------------------------
// DisplayInfo()
// to display the BITMAPFILEHEADER AND THE BITMAPINFOHEADER in the console
screen
// ------------------------------------------------------------------------------------
void ImgBMP::DisplayInfo()
{
```

```
    //display the BITMAPFILEHEADER
    printf("fh.bfType:%x\n",fh.bfType);
    printf("fh.bfSize:%d\n",fh.bfSize);
    printf("fh.bfReserved1:%d\n",fh.bfReserved1);
    printf("fh.bfReserved2:%d\n",fh.bfReserved2);
    printf("fh.bfOffBits:%d\n\n\n",fh.bfOffBits);

    //display the BITMAPINFOHEADER
    printf("info.biSize:%d\n",info.biSize);
    printf("info.biWidth:%d\n",info.biWidth);
    printf("info.biHeight:%d\n",info.biHeight);
    printf("info.biPlanes:%d\n",info.biPlanes);
    printf("info.biBitCount:%d\n",info.biBitCount);
    printf("info.biCompression:%d\n",info.biCompression);
    printf("info.biSizeImage:%d\n",info.biSizeImage);
    printf("info.biXPelsPerMeter:%d\n",info.biXPelsPerMeter);
    printf("info.biYPelsPerMeter:%d\n",info.biYPelsPerMeter);
    printf("info.biClrUsed:%d\n",info.biClrUsed);
    printf("info.biClrImportant:%d\n",info.biClrImportant);
}
// ---------------------------------------------------------------------------------
// LoadRGBQUADArry(FILE *stream)
// to load the RGBQUADArry into pQUADArry
// each element of pQUADArry is a BYTE
// ---------------------------------------------------------------------------------
void ImgBMP::LoadRGBQUADArry(FILE *stream)
{
    QUADArrySize=fh.bfOffBits-BMPHeaderSize;
    if(QUADArrySize==0){    //there is no RGBQUAD array in the image
        pQUADArry=NULL;
        return;
    }else{                  //load the RGBQUAD array into the memory pointed by pQUAD
        pQUADArry=(BYTE*)malloc(QUADArrySize);
        if(pQUADArry)
            fread(pQUADArry, SizeofBYTE, QUADArrySize, stream);
        else throw "can't allocate memory for the RGBQUAD array";
    }
}
// ---------------------------------------------------------------------------------
// DisplayRGBQUADArry()
// to display the RGBQUADArry in the console screen
// ---------------------------------------------------------------------------------
void ImgBMP::DisplayRGBQUADArry()
{
```

```
	if(QUADArrySize==0)
		printf("there is no RGBQUAD array in this image\n");
	else{
		BYTE *p=pQUADArry;
		for(int i=0; i<QUADArrySize; i++)
			cout<<p[i]<<endl;
	}
}
// ------------------------------------------------------------------------------------
// to load the pixel array into pPixelArry
// each element of pPixelArry is a BYTE
// ------------------------------------------------------------------------------------
void ImgBMP::LoadPixelArry(FILE *stream)
{
	pPixelArry=(BYTE*)malloc(fh.bfSize-fh.bfOffBits);
	//if(pPixelArry)
		fread(pPixelArry, SizeofBYTE, (fh.bfSize-fh.bfOffBits), stream);
	//else throw "can't allocate memory for the pixel array";
}

//???????????????????????????????????????????
// there is still a problem with writing the pixels into a file
// ------------------------------------------------------------------------------------
// to display pixel array into a file named "pixel value in byte.out"
// the element of each output unit is a byte
// ------------------------------------------------------------------------------------
void ImgBMP::DisplayPixelArry()
{
	FILE *stream=fopen("pixel value in byte.out", "wt");
	if(stream==NULL){
		printf("error occured when opening PIXEL VALUE IN BYTE.OUT\n");
		return;
	}else{
		BYTE *p=pPixelArry;
/*		BYTE temp;
		DWORD i=0, j;
		while(i<(100)){
			for(j=0; j<10; j++){
				//cout<<p[i]<<"\t";
				temp=p[i];
				fwrite(&temp, sizeof(BYTE), 1, stream);
				i++;
			}
			//cout<<endl;
```

```
			fwrite("\n",sizeof("\n"), 1, stream);
		}
*/
		fwrite(p, sizeof(BYTE), (fh.bfSize-fh.bfOffBits), stream);
 }
	fclose(stream);
}

//-----------------------------------------------------------------------------------------------------
// write the BITMAPFILEHEADER, BITMAPINFOHEADER AND RGBQUADQrry of this object to the file named "FileName"
// then append the given pixel buffer to the file to form a new file
//-----------------------------------------------------------------------------------------------------
void ImgBMP::Wt2File(char *FileName, BYTE *buffer)
{
	// creat the destination file
	FILE *streamDst = fopen(FileName,"wb");
	if(streamDst == NULL){
		printf("Faiure in FileWtHd: file %s can not be opened! Exiting....\n", FileName);
		exit(-1);
	}

	WtHd2File(streamDst);
	WtPixel2File(streamDst, buffer);
	fclose(streamDst);
}

void ImgBMP::WtHd2File(FILE *stream)
{
	//write the BITMAPFILEHEADER of the image
	fwrite(&fh.bfType, sizeof(fh.bfType), 1, stream);
	fwrite(&fh.bfSize, sizeof(fh.bfSize), 1, stream);
	fwrite(&fh.bfReserved1, sizeof(fh.bfReserved1), 1, stream);
	fwrite(&fh.bfReserved2, sizeof(fh.bfReserved2), 1, stream);
	fwrite(&fh.bfOffBits, sizeof(fh.bfOffBits), 1, stream);

	//write the BITMAPINFOHEADER of the image
	fwrite(&info.biSize, sizeof(info.biSize), 1, stream);
	fwrite(&info.biWidth, sizeof(info.biWidth), 1, stream);
	fwrite(&info.biHeight, sizeof(info.biHeight), 1, stream);
	fwrite(&info.biPlanes, sizeof(info.biPlanes), 1, stream);
	fwrite(&info.biBitCount, sizeof(info.biBitCount), 1, stream);
	fwrite(&info.biCompression, sizeof(info.biCompression), 1, stream);
	fwrite(&info.biSizeImage, sizeof(info.biSizeImage), 1, stream);
```

```
        fwrite(&info.biXPelsPerMeter, sizeof(info.biXPelsPerMeter), 1, stream);
        fwrite(&info.biYPelsPerMeter, sizeof(info.biYPelsPerMeter), 1, stream);
        fwrite(&info.biClrUsed, sizeof(info.biClrUsed), 1, stream);
        fwrite(&info.biClrImportant, sizeof(info.biClrImportant), 1, stream);

        //write the RGBQUADArry of the image
        if(QUADArrySize!=0)
                fwrite(pQUADArry, SizeofBYTE, QUADArrySize, stream);
}

void ImgBMP::WtPixel2File(FILE *stream, BYTE *buffer)
{
        fwrite(buffer,SizeofBYTE,(fh.bfSize-fh.bfOffBits),stream);
}
//--------------------------------------------------------------------------------------------------------
//to get the members of the object
//--------------------------------------------------------------------------------------------------------
BYTE* ImgBMP::GetPixelArry()
{
        return pPixelArry;
}

DWORD ImgBMP::GetPixelArrySize()
{
        return (fh.bfSize-fh.bfOffBits);
}

//################################################
// this is the main function of the ECKBA_BMPv1 project
//################################################

#include "ECKBA.h"
#include "ImgBMP.h"

void printPixel(char *title, BYTE *buffer)
{
        printf("\n\n%s\n",title);
        for (int i=0; i<100;i++){
                printf("%d \t", buffer[i]);
        }

}

void main(int argc, char *argv[])
```

```
{
	if(argc!=9){
		printf("\nThe format of the command parameters is not correct!\n");
		printf("\nThe format should be: target_image_name.bmp encrypted_image_name.bmp
decrypted_image_name.bmp number_of_rounds	initial_condition_of_x
initial_condition_of_y	alpha	beta\n");
		return;
	}

	//generate ImgBMP object
	ImgBMP image(argv[1]);
	DWORD PixelSize=image.GetPixelArrySize();
	BYTE *buf=image.GetPixelArry();
//	printPixel("first 100 pixel values of the original image:", buf);

	//generate ECKBA object
	BYTE	r=atoi(argv[4]);
	DWORD x=atol(argv[5]);
	DWORD y=atol(argv[6]);
	DWORD a=atol(argv[7]);
	DWORD b=atol(argv[8]);
	ECKBA	ECKBAObj(r,x,y,a,b);

	//encrypt
	ECKBAObj.Encrypt(PixelSize, buf);
	buf=ECKBAObj.GetCipherBuf();
//	printPixel("first 100 pixel values after encryption:", buf);

	//generate an encrypted image
	image.Wt2File(argv[2], buf);

	//decrypt
	ECKBAObj.Decrypt();
	buf=ECKBAObj.GetPlainBuf();
//	printPixel("first 100 pixel values after Decrypttion:", buf);

	//gennerate an decrypted image
	image.Wt2File(argv[3], buf);

	cout<<"\ntime for encryption: "<<ECKBAObj.GetEncrptTime()<<endl;
	cout<<"\ntime for decryption: "<<ECKBAObj.GetDecrptTime()<<endl;
}
```

Annex C. Source codes related to class ImgJP2

This annex gives the header file Img JP2.h and source files including Img JP2.cpp and the corresponding main.cpp of project ECKBA_JP2v1

```
//////////////////////////////////////////////////////////////////////
// ImgJP2.h: interface for the ImgJP2 class.
//
//////////////////////////////////////////////////////////////////////

#if !defined(AFX_IMGJP2_H__03F04A9F_6C25_4CD7_B174_97BD7972D68E__INCLUDED_
)
#define AFX_IMGJP2_H__03F04A9F_6C25_4CD7_B174_97BD7972D68E__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include "confg.h"


class ImgJP2
{
public:
	ImgJP2(FILE *streamIn);
	ImgJP2();
	virtual ~ImgJP2();
	bool LoadBox(BYTE *buf, DWORD *bufCount);
	bool LoadCdBoxHdr(BYTE *buf);
	bool LoadCdTileHdr(BYTE *buf, WORD *bufCount, DWORD *WholeTileSiz);
	WORD LoadCdMainHdr(BYTE *buf);
	void printBuf(char *title, BYTE *buf, int count);


protected:
	DWORD GetRghOdr(DWORD var);

	FILE *stream;

};

#endif
// !defined(AFX_IMGJP2_H__03F04A9F_6C25_4CD7_B174_97BD7972D68E__INCLUDED_)
```

```
////////////////////////////////////////////////////////////////////
// ImgJP2.cpp: implementation of the ImgJP2 class.
//
////////////////////////////////////////////////////////////////////

#include "ImgJP2.h"

////////////////////////////////////////////////////////////////////
// Construction/Destruction
////////////////////////////////////////////////////////////////////
ImgJP2::ImgJP2()
{

}

ImgJP2::~ImgJP2()
{

}

ImgJP2::ImgJP2(FILE *streamIn)
{
	stream=streamIn;

  }

//#####################################################################################
###
// to reverse the order of the bytes in var
// making the most significant byte the least significant byte, etc.
//#####################################################################################
###
DWORD ImgJP2::GetRghOdr(DWORD var)
{
	int i;
	DWORD t=0, mask=255;
	for(i=0; i<4; i++){
		t=t<<8;
		t=t+(var&mask);
		var=var>>8;
	}
	return t;
}
```

```
//###################################################################################
###
// to print the first count number of elements of the buf
//###################################################################################
###
void ImgJP2::printBuf(char *title, BYTE *buf, int count)
{
	printf("%s\n",title);
	for (int i=0; i<count; i++){
		printf("%x\t", buf[i]);
	}
	cout<<endl<<endl;

}

//###################################################################################
###
// to load the data box header including: LBox, TBox
// assuming: no XLBox exisits
//###################################################################################
###
bool ImgJP2::LoadCdBoxHdr(BYTE *buf)
{
	fread(buf, SizeofBYTE, 8, stream);

	// determine if the Tbox field is 0x 6a70 3263('jp2c')
	if(buf[4]==0x6a && buf[5]==0x70 && buf[6]==0x32 && buf[7]==0x63)
		return true;
	else
		return false;
}

//###################################################################################
###
// to load the code stream main header into the memory
//###################################################################################
###
WORD ImgJP2::LoadCdMainHdr(BYTE *buf)
{
	BYTE *ptr=buf;
	int length;

	//load the SOC marker into the buffer
```

```
	fread(ptr, SizeofBYTE, 2, stream);
	ptr=ptr+2;

	while(true){
		fread(ptr, SizeofBYTE, 2, stream);		// load the MAR field of the marker
		if(ptr[0]==0xff && ptr[1]==0x90){		// encounter SOT marker FF90
			fseek(stream, -2, SEEK_CUR);
			break;						// get out of the loop
		}else{						// not a SOT marker, load the rest of
the marker segment
			ptr=ptr+2;
			fread(ptr, SizeofBYTE, 2, stream);	// load the length of the maker segment, this
length doen't include the marker field
			length=ptr[0]<<8;				// to form the 2 bytes into one integer
			length=length+ptr[1];
			ptr=ptr+2;
			fread(ptr, SizeofBYTE, length-2, stream);
//			printBuf("market segment", ptr, length-2);
			ptr=ptr+length-2;
		}
	}
	return (WORD)(ptr-buf);
}

//##################################################################################
###
// load tile part header into the menory
// input parameter:
// buf - the buffer to store the tile part header including SOT and SOD
// siz - the actual size of the tile part header loaded
// WholeTileSiz - the Psot field of SOT marker indicating the length of the whole tile part
// return value:
// if the current marker is a tile part header return true, else(EOC) return false
//##################################################################################
###
bool ImgJP2::LoadCdTileHdr(BYTE *buf, WORD *bufCount, DWORD *WholeTileSiz)
{
	BYTE *ptr=buf;
	int length;
	DWORD Psot;

	//determine whether it's EOC
	fread(ptr, SizeofBYTE, 2, stream);
	if(ptr[0]==0xff && ptr[1]==0xd9){
```

```
		fseek(stream, -2, SEEK_CUR);
		return false;
	}
	ptr=ptr+2;
	//load the SOT marker(consist of 12 bytes) into the buffer
	fread(ptr, SizeofBYTE, 10, stream);

	//get the Psot field of SOT marker
	//Psot: length in bytes, from the beginning of the 1st byte of
	//this SOT marker segment of the tile-part to the end of
	//the data of that tile-part
	Psot=ptr[4]<<8;
	Psot=(Psot+ptr[5])<<8;
	Psot=(Psot+ptr[6])<<8;
	Psot=Psot+ptr[7];

	// move forward the pointer by 10 bytes
	//despite SOT marker of 2 bytes there lefts 10 bytes of the marker segment
	ptr=ptr+10;

	while(true){
		fread(ptr, SizeofBYTE, 2, stream);		// load MAR field of the marker
		if(ptr[0]==0xff && ptr[1]==0x93){		// encounter SOD marker FF93
			ptr=ptr+2;				// increase the pointer by 2
			break;					// get out of the loop
		}else{						// not a SOT marker, load the rest of
the marker segment
			ptr=ptr+2;
			fread(ptr, SizeofBYTE, 2, stream);	// load the length of the maker segment, this
length doen't include the marker field
			length=ptr[0]<<8;			// to form the 2 bytes into one integer
			length=length+ptr[1];
			ptr=ptr+2;
			fread(ptr, SizeofBYTE, length-2, stream);
//			printBuf("market segment", ptr, length-2);
			ptr=ptr+length-2;
		}
	}
	*bufCount=(WORD)(ptr-buf);
	*WholeTileSiz=Psot;
	return true;
}

//##############################################################################
```

```
###
// to load a box into the memory
// input parameters:
// buf - where the box will be loaded
// bufCount - number of bytes loaded into buf
// return value:
// true    - if the box has been successfully loaded
// false - if the length of the box is zero(the last box of the file), the box has not been loaded
//################################################################################
###
bool ImgJP2::LoadBox(BYTE *buf, DWORD *bufCount)
{
	DWORD count;
	fread(&count, SizeofDWORD, 1, stream);
	count=GetRghOdr(count);
	*bufCount=count;
	fseek(stream, -4, SEEK_CUR);
	if(count==0){
		return false;
	}else{
		fread(buf, SizeofBYTE, count, stream);
		return true;
	}
}
//############################################
// this is the main function of the ECKBA_JP2v1 project
//############################################

#include "confg.h"
#include "ImgJP2.h"
#include "ECKBA.h"

const DWORD MaxsizDWORD=10000000;

// !!!!!!!!!!!!!              ATTENTION          !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
// this program only handles the jp2 images with their codestream box as the last box in the file
// !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

void main(int argc, char *argv[])
{
	if(argc!=9){
		printf("\nThe format of the command parameters is not correct!\n");
		printf("\nThe format should be:\n target_image_name.jp2
encrypted_image_name.jp2    decrypted_image_name.jp2      rounds    initial_condition_of_x
```

```
initial_condition_of_y      alpha      beta\n");
                return;
        }

        BYTE *buffer=(BYTE*)calloc(MaxsizDWORD, 1);
        WORD sizWORD=0;
        DWORD sizDWORD=0;
        DWORD imageSiz=0;
        double timeEn=0, timeDe=0;

        //generate ECKBA object
        BYTE    r=atoi(argv[4]);
        DWORD x=atol(argv[5]);
        DWORD y=atol(argv[6]);
        DWORD a=atol(argv[7]);
        DWORD b=atol(argv[8]);
        ECKBA    ECKBAObj(r,x,y,a,b);

        //open the JP2 file in read and binary mode
        FILE *streamOrg = fopen(argv[1], "rb");
//      printf("the file name is:%s\n", argv[1]);
        if(streamOrg == NULL){
                printf("Faiure in opening file: %s Exiting....\n", argv[1]);
                exit(-1);
        }

        //creat the encrption file
        FILE *streamEn = fopen(argv[2],"wb");
        if(streamEn == NULL){
                printf("Faiure in opening file: %s    Exiting....\n", argv[2]);
                exit(-1);
        }

        //creat the decryption file
        FILE *streamDe = fopen(argv[3],"wb");
        if(streamDe == NULL){
                printf("Faiure in opening file: %s    Exiting....\n", argv[3]);
                exit(-1);
        }

        // creat the input image object
        ImgJP2    image(streamOrg);

        // load and wirte the boxes of the file until the last box
```

```
	while(image.LoadBox(buffer, &sizDWORD)){
		fwrite(buffer, SizeofBYTE, sizDWORD, streamEn);
		fwrite(buffer, SizeofBYTE, sizDWORD, streamDe);
	}

	// load and write the main header of the contiguous codestream box
	if(!image.LoadCdBoxHdr(buffer)){
		cout<<"the last box of the file is not a codestream box!\n";
		cout<<"\nthis program only handles the jp2 images with their codestream box as the last
box in the file\n";
		cout<<"no encryption and decryption will be applied to the image!\n";
		cout<<"the process will be teminated!\n"<<endl;
		exit(1);
	}
	fwrite(buffer, SizeofBYTE, 8, streamEn);
	fwrite(buffer, SizeofBYTE, 8, streamDe);

	// load the data of the codestream
	// 1	load the main header
	sizDWORD=image.LoadCdMainHdr(buffer);
	fwrite(buffer, SizeofBYTE, sizDWORD, streamEn);
	fwrite(buffer, SizeofBYTE, sizDWORD, streamDe);

	// 2	load tile parts, afterwards encrypt and decrypt the loaded data until encoutering EOC
	// for image.LoadTileHdr:
	//sizDWORD - the whole size of the tile part including header and data
	// !!ATTENTION!! here we should use &sizDWORD to make sure sizDWORD is assigned
the value of header size
	while(image.LoadCdTileHdr(buffer, &sizWORD, &sizDWORD)){

		// write the tile part header to the encrypted and decrypted files
		fwrite(buffer, SizeofBYTE, sizWORD, streamEn);
		fwrite(buffer, SizeofBYTE, sizWORD, streamDe);

		// get the actual sizDWORDe of the data by minusing the sizDWORDWORD of the
header
		sizDWORD=sizDWORD-sizWORD;
//		cout<<"sizDWORDe of the tile part data: "<<sizDWORD<<endl;

		// the tile part data may not be the last in the codestream
		if(sizDWORD!=0){
			imageSiz=imageSiz+sizDWORD;
			// load the tile part data into buffer
			fread(buffer, SizeofBYTE, sizDWORD, streamOrg);
```

```
//			image.printBuf("tile part data values:", buffer, 100);

			// encrypt the buffer and write the result into the encryted file
			ECKBAObj.Encrypt(sizDWORD, buffer);
			timeEn=timeEn+ECKBAObj.GetEncrptTime();
			fwrite(ECKBAObj.GetCipherBuf(), SizeofBYTE, sizDWORD, streamEn);

			// decryption and write the result into the decrypted file
			ECKBAObj.Decrypt();
			timeDe=timeDe+ECKBAObj.GetDecrptTime();
			fwrite(ECKBAObj.GetPlainBuf(), SizeofBYTE, sizDWORD, streamDe);
		}

		// the tile part data is the last one in the codestream
		else{
			// load the data until EOC
			BYTE *p=buffer;
			fread(p, SizeofBYTE, 1, streamOrg);
			while(true){
				p++;
				fread(p, SizeofBYTE, 1, streamOrg);
				if(*(p-1)==0xff && *p==0x09){		// encounter EOC marker
					p=p-2;	// move backwards the pointer to the end of the data, exlcuding
EOC marker
					fseek(streamOrg, -2, SEEK_CUR);	// set back the file stream poiner
correspondingly
					break;
				}
			}
			// calculate the size of the data
			sizDWORD=(DWORD)(p-buffer);
			imageSiz=imageSiz+sizDWORD;
//			image.printBuf("tile part data values:", buffer, 100);

			// encrypt the buffer and write the result into the encryted file
			ECKBAObj.Encrypt(sizDWORD, buffer);
			timeEn=timeEn+ECKBAObj.GetEncrptTime();
			fwrite(ECKBAObj.GetCipherBuf(), SizeofBYTE, sizDWORD, streamEn);

			// decryption and write the result into the decrypted file
			ECKBAObj.Decrypt();
			timeDe=timeDe+ECKBAObj.GetDecrptTime();
			fwrite(ECKBAObj.GetPlainBuf(), SizeofBYTE, sizDWORD, streamDe);
			break;
```

```
        }


    }

    // load and wirte the EOC marker
    fread(buffer, SizeofBYTE, 2, streamOrg);
    fwrite(buffer, SizeofBYTE, 2, streamEn);
    fwrite(buffer, SizeofBYTE, 2, streamDe);

    // display the encryption time and decryption time
    cout<<"time for encryption: "<<timeEn<<endl;
    cout<<"time for decryption: "<<timeDe<<endl;
    cout<<"the actual image size:"<<imageSiz<<endl;

    fclose(streamOrg);
    fclose(streamEn);
    fclose(streamDe);
}
```

Annex D. Bitmap Storage

Bitmaps should be saved in a file that uses the established bitmap file format and assigned a name with the three-character .bmp extension. The established bitmap file format consists of a **BITMAPFILEHEADER** structure followed by a **BITMAPINFOHEADER**, **BITMAPV4HEADER**, or **BITMAPV5HEADER** structure. An array of **RGBQUAD** structures (also called a color table) follows the bitmap information header structure. The color table is followed by a second array of indexes into the color table (the actual bitmap data).

The bitmap file format is shown in the following illustration.

BITMAPFILEHEADER
BITMAPINFOHEADER
RGBQUAD array
Color-index array

Windows 95, Windows NT 4.0: Replace the **BITMAPINFOHEADER** structure with the **BITMAPV4HEADER** structure.

Windows 98/Me, Windows 2000/XP: Replace the **BITMAPINFOHEADER** structure with the **BITMAPV5HEADER** structure.

The members of the **BITMAPFILEHEADER** structure identify the file; specify the size of the file, in bytes; and specify the offset from the first byte in the header to the first byte of bitmap data. The members of the **BITMAPINFOHEADER**, **BITMAPV4HEADER**, or **BITMAPV5HEADER** structure specify the width and height of the bitmap, in pixels; the color format (count of color planes and color bits-per-pixel) of the display device on which the bitmap was created; whether the bitmap data was compressed before storage and the type of compression used; the number of bytes of bitmap data; the resolution of the display device on which the bitmap was created; and the number of colors represented in the data. The **RGBQUAD** structures specify the RGB intensity values for each of the colors in the device's palette.

The color-index array associates a color, in the form of an index to an **RGBQUAD** structure, with each pixel in a bitmap. Thus, the number of bits in the color-index array equals the number of pixels times the number of bits needed to index the **RGBQUAD** structures. For example, an 8x8 black-and-white bitmap has a color-index array of 8 * 8 * 1 = 64 bits, because one bit is needed to index two colors. The Redbrick.bmp, mentioned in About Bitmaps, is a 32x32 bitmap with 16 colors; its color-index array is 32 * 32 * 4 = 4096 bits because four bits index 16 colors.

To create a color-index array for a top-down bitmap, start at the top line in the bitmap. The index of the **RGBQUAD** for the color of the left-most pixel is the first *n* bits in the color-index array (where *n* is the number of bits needed to indicate all of the **RGBQUAD** structures). The color of the next pixel to the right is the next *n* bits in the array, and so forth. After you reach the right-most pixel in the line, continue with the left-most pixel in the line below. Continue until you finish with the entire bitmap. If it is a bottom-up bitmap, start at the bottom line of the bitmap instead of the top line, still going from left to right, and continue to the top line of the bitmap.

The following hexadecimal output shows the contents of the file Redbrick.bmp.

```
0000 42 4d 76 02 00 00 00 00 00 76 00 00 00 28 00 0010 00 00
20 00 00 00 20 00 00 00 01 00 04 00 00 00 0020 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 0030 00 00 00 00 00 00 00 00 00
00 00 80 00 00 80 0040 00 00 00 80 80 00 80 00 00 00 80 00 80 00
80 80 0050 00 00 80 80 80 00 c0 c0 c0 00 00 00 ff 00 00 ff 0060
00 00 00 ff ff 00 ff 00 00 00 ff 00 ff 00 ff ff 0070 00 00 ff ff
ff 00 00 00 00 00 00 00 00 00 00 0080 00 00 00 00 00 00 00
00 00 00 00 00 00 09 00 0090 00 00 00 00 00 00 11 11 01 19 11 01
10 10 09 09 00a0 01 09 11 11 01 90 11 01 19 09 09 91 11 10 09 11
00b0 09 11 19 10 90 11 19 01 19 19 10 10 11 10 09 01 00c0 91 10
91 09 10 10 90 99 11 11 11 11 19 00 09 01 00d0 91 01 01 19 00 99
11 10 11 91 99 11 09 90 09 91 00e0 01 11 11 11 91 10 09 19 01 00
11 90 91 10 09 01 00f0 11 99 10 01 11 11 91 11 11 19 10 11 99 10
09 10 0100 01 11 11 11 19 10 11 09 09 10 19 10 10 10 09 01 0110
11 19 00 01 10 19 10 11 11 01 99 01 11 90 09 19 0120 11 91 11 91
01 11 19 10 99 00 01 19 09 10 09 19 0130 10 91 11 01 11 11 91 01
91 19 11 00 99 90 09 01 0140 01 99 19 01 91 10 19 91 91 09 11 99
11 10 09 91 0150 11 10 11 91 99 10 90 11 01 11 11 19 11 90 09 11
0160 00 19 10 11 01 11 99 99 99 99 99 99 99 99 09 99 0170 99 99
99 99 99 99 00 00 00 00 00 00 00 00 00 0180 00 00 00 00 00 00
90 00 00 00 00 00 00 00 00 00 0190 00 00 00 00 00 00 99 11 11 11
19 10 19 19 11 09 01a0 10 90 91 90 91 00 91 19 19 09 01 10 09 01
11 11 01b0 91 11 11 11 10 00 91 11 01 19 10 11 10 01 01 11 01c0
90 11 11 11 91 00 99 09 19 10 11 90 09 90 91 01 01d0 19 09 91 11
01 00 90 10 19 11 00 11 11 00 10 11 01e0 01 10 11 19 11 00 90 19
10 91 01 90 19 99 00 11 01f0 91 01 11 01 91 00 99 09 09 01 10 11
91 01 10 91 0200 99 11 10 90 91 00 91 11 00 10 11 01 10 19 19 09
0210 10 00 99 01 01 00 91 01 19 91 19 91 11 09 10 11 0220 00 91
00 10 90 00 99 01 11 10 09 10 10 19 09 01 0230 91 90 11 09 11 00
90 99 11 11 11 90 19 01 19 01 0240 91 01 01 19 09 00 91 10 11 91
99 09 09 90 11 91 0250 01 19 11 11 91 00 91 19 01 00 11 00 91 10
```

```
11 01 0260 11 11 10 01 11 00 99 99 99 99 99 99 99 99 99 99 0270
99 99 99 99 99 90
```

The following table shows the data bytes associated with the structures in a bitmap file.

Structure	Corresponding bytes
BITMAPFILEHEADER	0x00 0x0D
BITMAPINFOHEADER	0x0E 0x35
RGBQUAD array	0x36 0x75
Color-index array	0x76 0x275

D.1. BITMAPFILEHEADER

The **BITMAPFILEHEADER** structure contains information about the type, size, and layout of a file that contains a DIB.

```
typedef struct tagBITMAPFILEHEADER {
    WORD bfType;
    DWORD bfSize;
    WORD bfReserved1;
    WORD bfReserved2;
    DWORD bfOffBits;
    } BITMAPFILEHEADER, *PBITMAPFILEHEADER;
```

Members

bfType

Specifies the file type, must be BM.

bfSize

Specifies the size, in bytes, of the bitmap file.

bfReserved1

Reserved; must be zero.

bfReserved2

Reserved; must be zero.

bfOffBits

Specifies the offset, in bytes, from the beginning of the **BITMAPFILEHEADER** structure to the bitmap bits.

Remarks

A **BITMAPINFO** or **BITMAPCOREINFO** structure immediately follows the **BITMAPFILEHEADER** structure in the DIB file. For more information, see Bitmap Storage.

D.2. BITMAPINFO

The **BITMAPINFO** structure defines the dimensions and color information for a DIB.

```
typedef struct tagBITMAPINFO {
  BITMAPINFOHEADER bmiHeader;
  RGBQUAD bmiColors[1];
  } BITMAPINFO, *PBITMAPINFO;
```

Members

bmiHeader

Specifies a **BITMAPINFOHEADER** structure that contains information about the dimensions of color format.

.

bmiColors

The **bmiColors** member contains one of the following:

- An array of **RGBQUAD**. The elements of the array that make up the color table.
- An array of 16-bit unsigned integers that specifies indexes into the currently realized logical palette. This use of **bmiColors** is allowed for functions that use DIBs. When **bmiColors** elements contain indexes to a realized logical palette, they must also call the following bitmap functions:

  CreateDIBitmap

CreateDIBPatternBrush

CreateDIBSection

The i*Usage* parameter of **CreateDIBSection** must be set to DIB_PAL_COLORS.

The number of entries in the array depends on the values of the **biBitCount** and **biClrUsed** members of the **BITMAPINFOHEADER** structure.

The colors in the **bmiColors** table appear in order of importance. For more information, see the Remarks section.

Remarks

A DIB consists of two distinct parts: a **BITMAPINFO** structure describing the dimensions and colors of the bitmap, and an array of bytes defining the pixels of the bitmap. The bits in the array are packed together, but each scan line must be padded with zeroes to end on a **LONG** data-type boundary. If the height of the bitmap is positive, the bitmap is a bottom-up DIB and its origin is the lower-left corner. If the height is negative, the bitmap is a top-down DIB and its origin is the upper left corner.

A bitmap is packed when the bitmap array immediately follows the **BITMAPINFO** header. Packed bitmaps are referenced by a single pointer. For packed bitmaps, the **biClrUsed** member must be set to an even number when using the DIB_PAL_COLORS mode so that the DIB bitmap array starts on a **DWORD** boundary.

Note The **bmiColors** member should not contain palette indexes if the bitmap is to be stored in a file or transferred to another application.

Unless the application has exclusive use and control of the bitmap, the bitmap color table should contain explicit RGB values.

D.3. BITMAPINFOHEADER

The **BITMAPINFOHEADER** structure contains information about the dimensions and color format of a DIB.

Windows 95, Windows NT 4.0: Applications can use the **BITMAPV4HEADER** structure for added functionality.

Windows 98/Me, Windows 2000/XP: Applications can use the **BITMAPV5HEADER** structure for added functionality. However, these are used only in the **CreateDIBitmap** function.

```
typedef struct tagBITMAPINFOHEADER{
   DWORD biSize;
   LONG biWidth;
   LONG biHeight;
   WORD biPlanes;
   WORD biBitCount;
   DWORD biCompression;
   DWORD biSizeImage;
   LONG biXPelsPerMeter;
   LONG biYPelsPerMeter;
   DWORD biClrUsed;
   DWORD biClrImportant;
   } BITMAPINFOHEADER, *PBITMAPINFOHEADER;
```

Members

biSize

Specifies the number of bytes required by the structure.

biWidth

Specifies the width of the bitmap, in pixels.

Windows 98/Me, Windows 2000/XP: If **biCompression** is BI_JPEG or BI_PNG, the **biWidth** member specifies the width of the decompressed JPEG or PNG image file, respectively.

biHeight

Specifies the height of the bitmap, in pixels. If **biHeight** is positive, the bitmap is a bottom-up DIB and its origin is the lower-left corner. If **biHeight** is negative, the bitmap is a top-down DIB and its origin is the upper-left corner.

If **biHeight** is negative, indicating a top-down DIB, **biCompression** must be either BI_RGB or BI_BITFIELDS. Top-down DIBs cannot be compressed.

Windows 98/Me, Windows 2000/XP: If **biCompression** is BI_JPEG or BI_PNG, the **biHeight** member specifies the height of the decompressed JPEG or PNG image file, respectively.

biPlanes

Specifies the number of planes for the target device. This value must be set to 1.

biBitCount

Specifies the number of bits-per-pixel. The **biBitCount** member of the **BITMAPINFOHEADER** structure determines the number of bits that define each pixel and the maximum number of colors in the bitmap. This member must be one of the following values.

Value	Meaning
0	**Windows 98/Me, Windows 2000/XP:** The number of bits-per-pixel is specified or is implied by the JPEG or PNG format.
1	The bitmap is monochrome, and the **bmiColors** member of **BITMAPINFO** contains two entries. Each bit in the bitmap array represents a pixel. If the bit is clear, the pixel is displayed with the color of the first entry in the **bmiColors** table; if the bit is set, the pixel has the color of the second entry in the table.
4	The bitmap has a maximum of 16 colors, and the **bmiColors** member of **BITMAPINFO** contains up to 16 entries. Each pixel in the bitmap is represented by a 4-bit index into the color table. For example, if the first byte in the bitmap is 0x1F, the byte represents two pixels. The first pixel contains the color in the second table entry, and the second pixel contains the color in the sixteenth table entry.
8	The bitmap has a maximum of 256 colors, and the **bmiColors** member of **BITMAPINFO** contains up to 256 entries. In this case, each byte in the array represents a single pixel.
16	The bitmap has a maximum of 2^16 colors. If the **biCompression** member of the **BITMAPINFOHEADER** is BI_RGB, the **bmiColors** member of **BITMAPINFO** is NULL. Each **WORD** in the bitmap array represents a single pixel. The relative intensities of red, green, and blue are represented with five bits for each color component. The value for blue is in the least significant five bits, followed by five bits each for green and red. The most significant bit is not used. The **bmiColors** color table is used for optimizing colors used on palette-based devices, and must contain the number of entries specified by the **biClrUsed** member of the **BITMAPINFOHEADER**. If the **biCompression** member of the **BITMAPINFOHEADER** is BI_BITFIELDS, the **bmiColors** member contains three **DWORD** color masks that specify the red, green, and blue components, respectively, of each pixel. Each **WORD** in the bitmap array represents a single pixel. **Windows NT/Windows 2000/XP:** When the **biCompression** member is BI_BITFIELDS, bits set in each **DWORD** mask must be contiguous and should not overlap the bits of another mask. All the bits in the pixel do not have to be used. **Windows 95/98/Me:** When the **biCompression** member is BI_BITFIELDS, the system supports only the following 16bpp color masks: A 5-5-5 16-bit image, where the blue mask is 0x001F, the green mask is 0x03E0, and the red mask is 0x7C00; and a 5-6-5 16-bit image, where the blue mask is 0x001F, the green mask is 0x07E0, and the red mask is 0xF800.
24	The bitmap has a maximum of 2^24 colors, and the **bmiColors** member of **BITMAPINFO** is NULL. Each 3-byte triplet in the bitmap array represents the relative intensities of blue, green, and red,

	respectively, for a pixel. The **bmiColors** color table is used for optimizing colors used on palette-based devices, and must contain the number of entries specified by the **biClrUsed** member of the **BITMAPINFOHEADER**.
32	The bitmap has a maximum of 2^32 colors. If the **biCompression** member of the **BITMAPINFOHEADER** is BI_RGB, the **bmiColors** member of **BITMAPINFO** is NULL. Each **DWORD** in the bitmap array represents the relative intensities of blue, green, and red, respectively, for a pixel. The high byte in each **DWORD** is not used. The **bmiColors** color table is used for optimizing colors used on palette-based devices, and must contain the number of entries specified by the **biClrUsed** member of the **BITMAPINFOHEADER**. If the **biCompression** member of the **BITMAPINFOHEADER** is BI_BITFIELDS, the **bmiColors** member contains three **DWORD** color masks that specify the red, green, and blue components, respectively, of each pixel. Each **DWORD** in the bitmap array represents a single pixel. **Windows NT/ 2000:** When the **biCompression** member is BI_BITFIELDS, bits set in each **DWORD** mask must be contiguous and should not overlap the bits of another mask. All the bits in the pixel do not need to be used. **Windows 95/98/Me:** When the **biCompression** member is BI_BITFIELDS, the system supports only the following 32-bpp color mask: The blue mask is 0x000000FF, the green mask is 0x0000FF00, and the red mask is 0x00FF0000.

biCompression

Specifies the type of compression for a compressed bottom-up bitmap (top-down DIBs cannot be compressed).

This member can be one of the following values.

Value	Description
BI_RGB	An uncompressed format.
BI_RLE8	A run-length encoded (RLE) format for bitmaps with 8 bpp. The compression format is a 2-byte format consisting of a count byte followed by a byte containing a color index. For more information, see Bitmap Compression.
BI_RLE4	An RLE format for bitmaps with 4 bpp. The compression format is a 2-byte format consisting of a count byte followed by two word-length color indexes. For more information, see Bitmap Compression.
BI_BITFIELDS	Specifies that the bitmap is not compressed and that the color table consists of three **DWORD** color masks that specify the red, green, and blue components, respectively, of each pixel. This is valid when used with 16- and 32-bpp bitmaps.
BI_JPEG	**Windows 98/Me, Windows 2000/XP:** Indicates that the image is a JPEG image.
BI_PNG	**Windows 98/Me, Windows 2000/XP:** Indicates that the image is a PNG image.

biSizeImage

Specifies the size, in bytes, of the image. This may be set to zero for BI_RGB bitmaps.

Windows 98/Me, Windows 2000/XP: If **biCompression** is BI_JPEG or BI_PNG, **biSizeImage** indicates the size of the JPEG or PNG image buffer, respectively.

biXPelsPerMeter

Specifies the horizontal resolution, in pixels-per-meter, of the target device for the bitmap. An application can use this value to select a bitmap from a resource group that best matches the characteristics of the current device.

biYPelsPerMeter

Specifies the vertical resolution, in pixels-per-meter, of the target device for the bitmap.

biClrUsed

Specifies the number of color indexes in the color table that are actually used by the bitmap. If this value is zero, the bitmap uses the maximum number of colors corresponding to the value of the **biBitCount** member for the compression mode specified by **biCompression**.

If **biClrUsed** is nonzero and the **biBitCount** member is less than 16, the **biClrUsed** member specifies the actual number of colors the graphics engine or device driver accesses. If **biBitCount** is 16 or greater, the **biClrUsed** member specifies the size of the color table used to optimize performance of the system color palettes. If **biBitCount** equals 16 or 32, the optimal color palette starts immediately following the three **DWORD** masks.

When the bitmap array immediately follows the **BITMAPINFO** structure, it is a packed bitmap. Packed bitmaps are referenced by a single pointer. Packed bitmaps require that the **biClrUsed** member must be either zero or the actual size of the color table.

biClrImportant

Specifies the number of color indexes that are required for displaying the bitmap. If this value is zero, all colors are required.

Remarks

The **BITMAPINFO** structure combines the **BITMAPINFOHEADER** structure and a color table to provide a complete definition of the dimensions and colors of a DIB. For more information about DIBs, see Device-Independent Bitmaps and **BITMAPINFO**.

An application should use the information stored in the **biSize** member to locate the color table in a **BITMAPINFO** structure, as follows:

```
pColor = ((LPSTR)pBitmapInfo + (WORD)(pBitmapInfo->bmiHeader.biSize));
```

Windows 98/Me, Windows 2000/XP: The **BITMAPINFOHEADER** structure is extended to allow a JPEG or PNG image to be passed as the source image to **StretchDIBits**.

D.4. RGBQUAD

The **RGBQUAD** structure describes a color consisting of relative intensities of red, green, and blue.

```
typedef struct tagRGBQUAD {
   BYTE rgbBlue;
   BYTE rgbGreen;
   BYTE rgbRed;
   BYTE rgbReserved;
   } RGBQUAD;
```

Members

rgbBlue

Specifies the intensity of blue in the color.

rgbGreen

Specifies the intensity of green in the color.

rgbRed

Specifies the intensity of red in the color.

rgbReserved

Reserved; must be zero.

Remarks

The **bmiColors** member of the **BITMAPINFO** structure consists of an array of **RGBQUAD** structures.

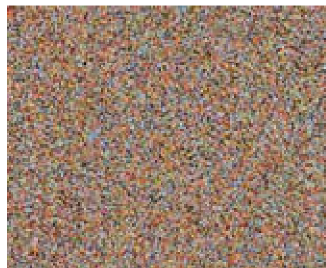
Annex E. Encrypted and decrypted images

1. The encrypted and decrypted images for project ECKBA_BMPv1 is shown as followed:

boat_en.bmp

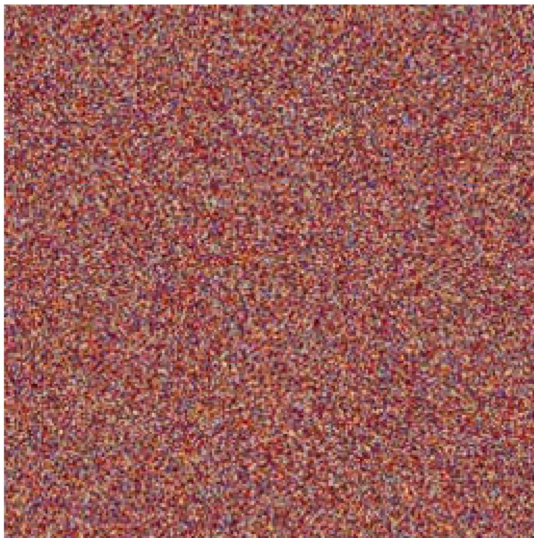
boat_de.bmp

mandril_en.bmp



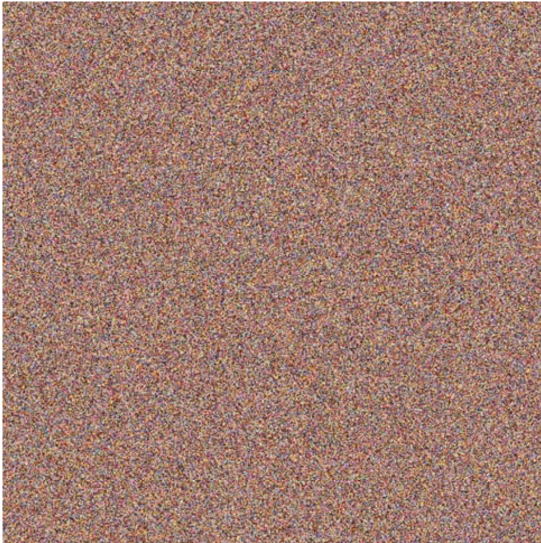
mandril_de.bmp

clown_en.bmp

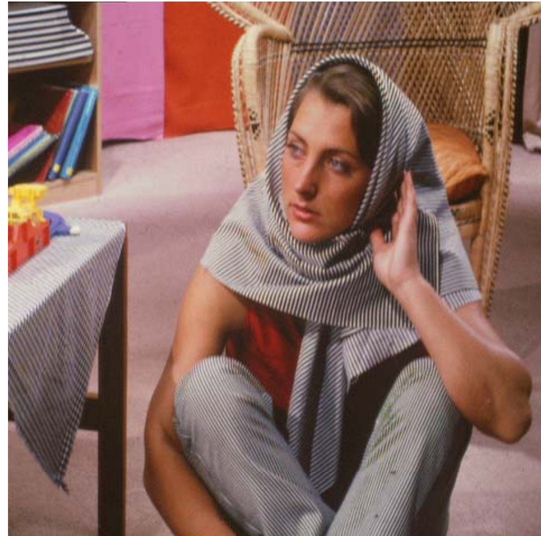


clown_de.bmp

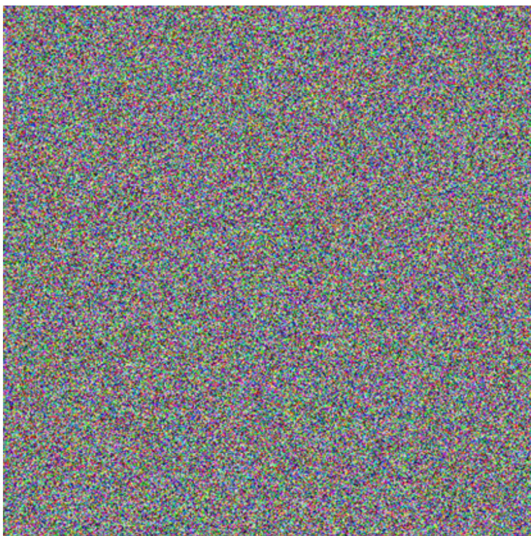
barb_en.bmp



barb_de.bmp



Lena_lumi_en.bmp



lena_lumi_de.bmp

2. The encrypted and decrypted images for project ECKBA_JP2v1 is shown as followed:

01boatEn.jp2

01boatDe.jp2

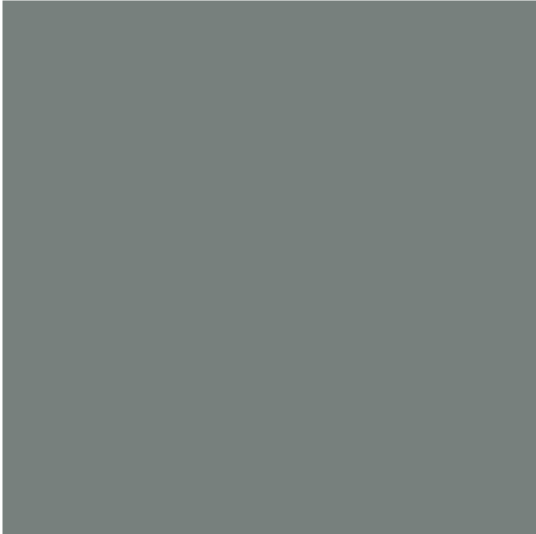
01mandrilEn.jp2

01mandrilDe.jp2

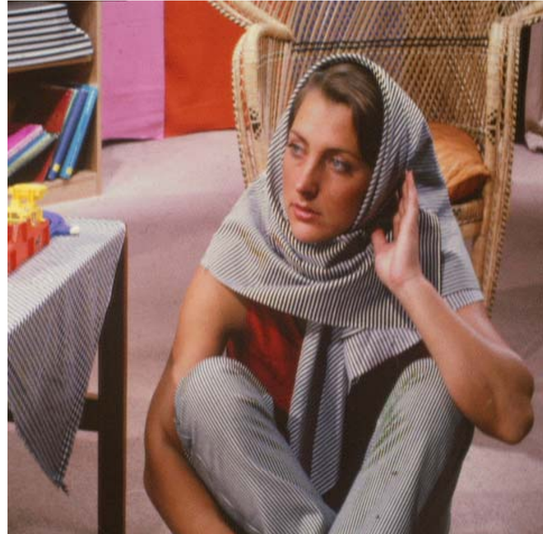
01clowmEn.jp2

01clownDe.jp2

01barbEn.jp2



01barbDe.jp2

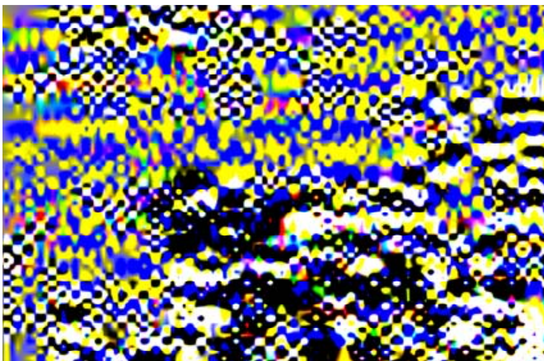


lena.jp2

01lenaEn.jp2

01lenaDe.jp2

01flowerEn.jp2



01flowerDe.jp2