# Using image as cipher key in AES

**Razi Hosseinkhani[1] and Seyyed Hamid Haj Seyyed Javadi[2]**

[1] **Department of computer engineering, Science and Research Branch, Islamic Azad University, Tehran, Iran**

[2] **Department of Mathematics and Computer ScienceShahed University, Tehran, Iran**

**Abstract:**

This paper describes how cipher key can be generated from image. We use image to generate cipher key for AES algorithm. After this step , cipher key watermarked in image. S-Box generated by this key which it called Key-dependant S-box. These steps make AES algorithm more robust and more reliable.

**Keywords:** *AES, Encryption, Key dependant S-Box, Watermarking, Image key generation*

## 1 – Introduction

In cryptography, encryption is the process of transforming information (referred to as plaintext) using an algorithm (called cipher) to make it unreadable to anyone except those possessing special knowledge, usually referred to as a key. The result of the process is encrypted information (in cryptography, referred to as ciphertext). In many contexts, the word encryption also implicitly refers to the reverse process, decryption (e.g. “software for encryption” can typically also perform decryption), to make the encrypted information readable again (i.e. to make it unencrypted).

Encryption has long been used by militaries and governments to facilitate secret communication. Encryption is now commonly used in protecting information within many kinds of civilian systems. For example, the Computer Security Institute reported that in 2007, 71% of companies surveyed utilized encryption for some of their data in transit, and 53% utilized encryption for some of their data in storage. Encryption can be used to protect data "at rest", such as files on computers and storage devices. In recent years there have been numerous reports of confidential data such as customers' personal records being exposed through loss or theft of laptops or backup drives. Encrypting such files at rest helps protect them should physical security measures fail. Digital rights management systems which prevent unauthorized use or reproduction of copyrighted material and protect software against reverse engineering are another somewhat different example of using encryption on data at rest.

Encryption is also used to protect data in transit, for example data being transferred via networks (e.g. the Internet, e-commerce), mobile telephones, wireless microphones, wireless intercom systems, Bluetooth devices and bank automatic teller machines. There have been numerous reports of data in transit being intercepted in recent years. Encrypting data in transit also helps to secure it as it is often difficult to physically secure all access to networks [1].

Cipher algorithms have the two general categories: Private Key algorithms and public key algorithms. Private Key algorithms using single key to encrypt plain text and decrypt cipher text in sender and receiver side. Private Key algorithm samples are: DES (DES, 1977), 3DES and Advanced Encryption Standard [2] Public Key algorithms, such as the Rivest-Shamir-Adleman (RSA), using two different key for encrypt plain text and decrypt cipher text in sender and receiver sides.

Block cipher systems depend on the S-Boxes, which are fixed and no relation with a cipher key[3]. So only changeable parameter is cipher key. Since the only nonlinear component of AES is S-Boxes, they are an important source of cryptographic strength. So we intend use an image to create cipher key and watermarking this cipher key into image and Using Cipher Key to Generate Dynamic S-Box [4]. That cause increasing the strength of AES algorithms.

In section 2, we briefly introduce the AES algorithm [2]. In section 3, we show that how cipher key generated from image [4]. In Section 4, we explain how cipher key watermarked into an image. In section 5, we show that how S-Box will be generated from cipher key and in the final section we analyze experiments and investigate about results.

## 2 - Advanced Encryption Standard (AES):

In cryptography, the Advanced Encryption Standard (AES) is a symmetric-key encryption standard adopted by the U.S. government. The standard comprises three block ciphers, AES-128, AES-192 and AES-256, adopted from a larger collection originally published as Rijndael. Each of these ciphers has a 128-bit block size, with key sizes of 128, 192 and 256 bits, respectively. The AES ciphers have been analyzed extensively and are now used worldwide, as was the case with its predecessor, the Data Encryption Standard (DES) [2].

AES was announced by National Institute of Standards and Technology (NIST) as U.S. FIPS PUB 197 (FIPS 197) on November 26, 2001 after a 5-year standardization process in which fifteen competing designs were presented and evaluated before Rijndael was selected as the most suitable. It became effective as a Federal government standard on May 26, 2002 after approval by the Secretary of Commerce. It is available in many different encryption packages. AES is the first publicly accessible and open cipher approved by the NSA for top secret information.

### 2.1- Description of the cipher

AES has a fixed block size of 128 bits and a key size of 128, 192, or 256 bits, whereas Rijndael can be specified with block and key sizes in any multiple of 32 bits, with a minimum of 128 bits. The blocksize has a maximum of 256 bits, but the keysize has no theoretical maximum.

AES operates on a 4×4 matrix of bytes, termed the state (versions of Rijndael with a larger block size have additional columns in the state). Most AES calculations are done in a special finite field.

The AES cipher is specified as a number of repetitions of transformation rounds that convert the input plaintext into the final output of ciphertext. Each round consists of several processing steps, including one that depends on the encryption key. A set of reverse rounds are applied to transform ciphertext back into the original plaintext using the same encryption key.

### 2.2- High-level description of the algorithm

1. **KeyExpansion**: round keys are derived from the cipher key using Rijndael's key schedule
2. **Initial Round**
   1. **AddRoundKey**: each byte of the state is combined with the round key using bitwise xor
3. **Rounds**
   1. **SubBytes**: a non-linear substitution step where each byte is replaced with another according to a lookup table.
   2. **ShiftRows**: a transposition step where each row of the state is shifted cyclically a certain number of steps.
   3. **MixColumns**: a mixing operation which operates on the columns of the state, combining the four bytes in each column.
   4. **AddRoundKey**
4. **Final Round** (no MixColumns)
   1. **SubBytes**
   2. **ShiftRows**
   3. **AddRoundKey**

The Cipher is described in the pseudo code in Algorithm 1.

```
public word[] Cipher(byte[] plainText, byte[] cipherKey)
{
    state = new word[4];
    sBox = new newSbox(cipherKey);
    ks = new KeySchedule(cipherKey);
    for (int i = 0; i < 4; i++)
    {
        for (int j = 0; j < 4; j++)
        {
            if (state[j] == null)
                state[j] = new word();

            state[j].w[i] = plainText[i * 4 + j];
        }
    }
    AddRoundKey(0);
    for (int i = 1; i < Nr; i++)
    {
        SubBytes();
        ShiftRows();
        MixColumn();
        AddRoundKey(i);
    }

    SubBytes();
    ShiftRows();
    AddRoundKey(Nr);
    return state;
}
```

Algorithm 1.Pseudo Code for Cipher

## 3 – Generate Key from image:

To generate key, we need 16 points from image. Each of these points converted into one byte of Key. The following algorithm shows that how these points will be selected from image. In next step, we need to generate key bytes from these points.

Algorithm steps:

1- Height and width of the first point acquired with width, height and RGB color of center point.

2- Other points acquired with following function, that it uses RGB color to generate one byte of key. The SecretKeyGenerator described in the pseudo code in algorithm 2.

After executing the SecretKeyGenerator(Algorithm 2), secret key is ready to be watermarked in image.

**4- Watermarking secret key in image**

4.1– Bitmap files structure

The first 54 byte of BMP file is header which it's fixed in size. Other bytes have information about points color[5].


Figure 3: BMP file structure in brief

4.2- Watermarking algorithm

The watermarking algorithm puts Secret Key bytes into lower bits of image points so that the image size had to greater than 128 * 8 byte.

```
public SecretKeyGenerator(string address)
{
    bmp = new Bitmap(address);
    int centerX = bmp.Width / 2; int centerY = bmp.Height / 2;
    Color centerColor = bmp.GetPixel(centerX, centerY);
    int x = ((int)(centerColor.R * centerColor.G * centerColor.B
            * (bmp.Width + bmp.Height))) % bmp.Width;
    int y = 0;
    int[] points = new int[32];
    int i = 0, point = x, l = 10;
    while (i < 32)
    {
        point = 1 + (point * i * l);
        if (i % 2 == 0)
        {
            point += 1 + (point * i * (bmp.Height + l));
            point %= bmp.Width;
            point = Math.Abs(point);
        }
        else
        {
            point += 1 + (point * i * (bmp.Width + l));
            point %= bmp.Height;
            point = Math.Abs(point);
        }
        bool isExist = false;
        for (int j = 0; j < i; j += 1)
        {
            if (points[i] == point)
            {
                isExist = true;
                break;
            }
        }
        if (!isExist)
            points[i++] = point;
        l += 9;
    }
    for (int k = 0; k < 32; k += 2)
    {
        x = points[k];
        y = points[k + 1];
        Color tempColor = bmp.GetPixel(x, y);
        switch ((x + y) % 3)
        {
                case 0:
                Key.Add(tempColor.R);
                break;
            case 1:
                Key.Add(tempColor.G);
                break;
            case 2:
                Key.Add(tempColor.B);
                break;
        }

    }
return (Key);
}
```

Algorithm 2: SecretKeyGenerator generate Secret Key by image

Figure 1: The original Image.

Figure 2: Selected points for create Secret Key in SecretKeyGenerator function.

```
void WatermarkKeyToImage(Bitmap bmp, byte[] key)
{
        byte[] temp = ReadFully(bmp, 0);
        int i = 0, j = 7;
        for (int k = 54; k < temp.Length && i < key.Length; k++)
        {
                temp[k] &=FE;
                temp[k] += (byte)((key[i] >> j--) & 1);
                if (j < 0){ i++; j = 7; }
        }
        WriteIntoFile(temp, bmp);
}
```

Algorithm 3: Watermark Secret key into image.

## 5 – Dynamic S-Box generation from cipher key algorithm:

5.1 – First Step :

We need primary S-Box to generate dynamic S-Box , that should has 16 rows and columns. We use S-Box generation algorithm that introduced in AES[2], to create primary S-Box as follows:

1- Take the mulltiplicative inverse in the finite feild $GF(2^8)$; the element {00} is mapped itself.

**2-** Apply the following affine transformation (over GF(2)):

$$b_i' = b_i \oplus b_{(i+4)mod8} \oplus b_{(i+5)mod8} \oplus b_{(i+6)mod8} \oplus b_{(i+7)mod8} \oplus c_i$$

Equation 1: Affine transformation that is used to create s-box

For $0 \leq i < 8$ ,where $b_i$ is the $i^{th}$ bit of the byte, and $c_i$ is the $i^{th}$ of a bytec with the value {63} or (01100011}. Here and elsewhere, a prime on a variable (e.g., $b^i$) indicate that the variable is to be uodated with the value on the right. In matrix form, the affine transformation element of the S-Box can be expressed as:

$$\begin{bmatrix} b_0' \\ b_1' \\ b_2' \\ b_3' \\ b_4' \\ b_5' \\ b_6' \\ b_7' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}.$$

Figure 4: Step 2 in S-Box generation in AES

5.2 – Second Step :

In this step, rows swapped with columns of primary S-Box in GenerateDynamicSbox(cipherKey) function. This function guarantees new S-Box remain one-for-one. This routin get cipher key as input and generate dynamic S-Box from cipher key. Note that in this paper if cipher key has 192 or 256 bits size, we use only first 128 bits of cipher key [4].

5.2.1 – GenerateDynamicSbox Algorithm :

IJCSI International Journal of Computer Science Issues, Vol. 9, Issue 2, No 2, March 2012
ISSN (Online): 1694-0814
www.IJCSI.org


```
1:void GenerateDynamicSbox(byte[16] key)
2:{
3:      byte rowIndex, columnIndex;
4:      byte shiftCount = GetShiftCount(key);

5:      byte[,] sBox = GeneratePrimarySbox();

6:      for(int i = 0; i < 16;i++)
7:      {
8:              GetProperIndex(key[i], out rowIndex, out colun
9:              ShiftRow(rowIndex, shiftCount, sBox);
10:             ShiftColumn(columnIndex, shiftCount, sBox);
11:             Swap(rowIndex, columnIndex, sBox);
12:     }
```

Algorithm 4: The GenerateDynamicSbox(cipherKey) function generate dynamic S-Box form cipher key.

In line 4, GetShiftCount(cipherKey) get cipherKey as input and return number of shift that should be applied to rows and columns before replacing with each other.

In line 5, GeneratePrimarySbox () generate primary S-Box according 4.1.

In line 6, start loop for 16 times (foreach byte of cipher key, only first 16 byte of cipher key is used).

In line 8, GetProperIndex(cipherKey[i], out rowIndex, out columnIndex) get byte of cipher key and return indexes of row and column that should be replaced with each other.

In line 9, ShiftRow(rowIndex, shiftCount, sBox) get row index of S-Box and shift each element of given row cyclically. It means if rowIndex = 0 and shiftCount = 1, first element of S-Box, sBox[0,1] replace with sBox[0,0] and sBox[0,2] replace with sBox[0,1] ... and sBox[0,0] replace with sBox[0,15]. (The first index of sBox determine rowIndex and second one determine columnIndex).

In line 10, ShiftColumn(columnIndex, shiftCount, sBox) get column index of S-Box and shift each element of given column cyclically. It means if columnIndex = 0 and shiftCount = 1, first element of S-Box, sBox[1,0] replace with sBox[0,0] and sBox[2,0] replace with sBox[1,0] ... and sBox[0,0] replace with sBox[15,0].

In line 11, Swap(rowIndex, columnIndex, sBox) get row and column index and then swapped them with each other. For example if rowIndex = 5 and columnIndex = 4 the Swap function swapping element at sBox[0,5] with sBox[4,0] and sBox[1,5] with sBox[4,1] and ... and finaly sBox[15,5] swap with sBox[4,15].

5.2.2 – GetShiftCount Algorithm:

This function get cipher key as input and then return number of shift count as output. If cipher key larger than 128 bit, only first 128 should be used.

```
1:byte GetShiftCount(byte[16] cipherKey)

2:{
3:      byte customizingFactor =00;
4:      byte shiftCount = 0;

5:      for(int i = 0 ; i < 16 ;i++)
6:      {
7:          shiftCount ^= (byte)((key[i] * (i + 1)) % (0xFF + 1));
8:      }

9:      return shiftCount ^ customizingFactor;
10:}
```

Algorithm 5: The ***GetShiftCount ()*** function used to getshift count before swapping rows with columns.

In line 4, customizingFactor value is in [0-255] range. This variable can customize the GetShiftCount return value and then customize GenerateDynamicSbox.

In line 5, start loop for 16 times (foreach byte of cipher key, only first 16 byte of cipher key is used).

In line 6, sign ^ means XOR operation and sign % means modulo in C#. This equation guarantees that changing only one bit of Cipher key cause changing the value of shiftCount.

.In line 9, shiftCount XOR with customizingFactor that cause generate 256 different customizing states for shiftCount value.

5.2.3 – GetProperIndex Algorithm:

This function gets byte of cipher key and then return rowIndex and columnIndex as output. This function using ***Shuffle exchange algorithm*** [9] that used in designing parallel algorithms.

```
1:void GetProperIndex (byte key, out byte rowIndex, out byte colun
2:{
3:        int[] rowUsedArray, columnUsedArray;

4:        rowIndex = key &0F;
5:        columnIndex = key >> 4;

6:        rowIndex = Shuffle (rowUsedArray, rowIndex);
7:        columnIndex = Shuffle (columnUsedArray, columnIndex

8:        rowUsedArray.Add(rowIndex);
9:        columnUsedArray.Add(columnIndex);

10:}
```

Algorithm6: The ***GetProperIndex*** function pseudo code

In line 3, rowUsedArray and columnUsedArray variables are using for saving index that used in previous steps.

In line 4, sign & means AND operation in C#.

In line 5, sign >> means shift right *n*-times in C#.

In line 6, ***Shuffle*** function get rowIndex number and return next available rowIndex number if given rowIndex is in rowUsedArray.

In line 7, ***Shuffle*** function get columnIndex number and return next available columnIndex number if given columnIndex is in columnUsedArray.

In line 8, current rowIndex add to rowUsedArray array.

In line 9, current columnIndex add to columnUsedArray array.
This causes that every row and column only one time returns with this function thus every row and column is used for one time in ***GenerateDynamicSbox***.

## 6 – Experimental results :

In general, in S-Box, $n$ input bits are first represented as one of $2^n$ different characters. The set of $2^n$ chracters are then permuted so that each character is transposed to one of the others in the set. The character is then converted back to an $n$-bit output. It can be easily shown that there are $(2^n)!$ differnet substitution or connection patterns possible.

The cryptanalyst's task becomes computionally unfeasible as $n$ gets large, say n = 128; then $2^n = 10^{38}$, and $(10^{38})!$ is an <u>*astronomical*</u> number.

We experimentally compared keys that generated by same image with different opacity(second image 1% lighter than first image), then we generate dynamic S-Box from keys. S-Box tables illustrated in figure 5, 6.

*key_hex1*
*={3c,1a,69,a4,0b,69,82,53,12,df,07,fd,3d,00,51,5d}*

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 245 | 57 | 127 | 119 | 234 | 51 | 122 | 19 | 109 | 231 | 200 | 55 | 37 | 146 | 137 | 78 |
| 240 | 185 | 239 | 201 | 175 | 156 | 164 | 13 | 143 | 62 | 130 | 192 | 253 | 250 | 219 | 71 |
| 217 | 230 | 3 | 56 | 63 | 38 | 41 | 106 | 208 | 157 | 151 | 222 | 7 | 213 | 168 | 133 |
| 154 | 237 | 5 | 23 | 226 | 235 | 39 | 178 | 103 | 28 | 199 | 35 | 46 | 24 | 45 | 218 |
| 72 | 32 | 90 | 67 | 179 | 79 | 227 | 47 | 43 | 9 | 131 | 44 | 31 | 27 | 15 | 33 |
| 216 | 173 | 65 | 52 | 82 | 220 | 167 | 188 | 254 | 70 | 121 | 108 | 138 | 97 | 176 | 16 |
| 246 | 212 | 153 | 165 | 59 | 34 | 249 | 182 | 215 | 238 | 211 | 86 | 186 | 53 | 84 | 255 |
| 76 | 12 | 115 | 181 | 95 | 144 | 68 | 0 | 197 | 69 | 126 | 61 | 100 | 141 | 22 | 174 |
| 101 | 162 | 89 | 104 | 54 | 60 | 247 | 251 | 221 | 117 | 229 | 147 | 75 | 202 | 49 | 184 |
| 83 | 209 | 207 | 203 | 132 | 196 | 177 | 91 | 112 | 252 | 190 | 8 | 74 | 205 | 158 | 14 |
| 87 | 194 | 36 | 2 | 98 | 77 | 149 | 228 | 124 | 224 | 50 | 58 | 10 | 73 | 161 | 163 |
| 134 | 125 | 25 | 145 | 26 | 113 | 18 | 210 | 118 | 128 | 172 | 236 | 195 | 102 | 17 | 64 |
| 198 | 232 | 180 | 150 | 214 | 80 | 189 | 139 | 183 | 129 | 120 | 244 | 116 | 4 | 21 | 204 |
| 20 | 242 | 81 | 193 | 136 | 92 | 169 | 123 | 99 | 191 | 140 | 1 | 93 | 107 | 48 | 160 |
| 29 | 248 | 223 | 152 | 105 | 170 | 142 | 148 | 11 | 159 | 135 | 233 | 206 | 85 | 94 | 42 |
| 171 | 114 | 40 | 96 | 110 | 155 | 30 | 243 | 111 | 88 | 6 | 241 | 166 | 225 | 66 | 187 |

Figure 5. The dynamic S-Box generated with *key_hex*$_1$*.(S-Box1)*

We find 253 different between S-Box1 and S-Box2, thus approximately %99 of second S-Box is changed. The difference of S-Box1 and S-Box2 elements is illustrated in figure 7.

*key_hex2*
*={3e,1d,6b,a5,0e,0c,83,55,15,df,0a,df,3f,03,53,5f}*

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 191 | 114 | 228 | 171 | 47 | 200 | 65 | 225 | 84 | 232 | 229 | 213 | 145 | 93 | 169 | 20 |
| 25 | 102 | 17 | 201 | 250 | 245 | 193 | 240 | 71 | 163 | 251 | 175 | 156 | 105 | 124 | 192 |
| 117 | 217 | 119 | 14 | 54 | 247 | 69 | 204 | 81 | 64 | 194 | 241 | 113 | 56 | 19 | 21 |
| 223 | 206 | 4 | 6 | 110 | 152 | 226 | 219 | 39 | 118 | 125 | 22 | 161 | 35 | 87 | 1 |
| 142 | 120 | 242 | 44 | 27 | 51 | 111 | 160 | 73 | 146 | 103 | 179 | 41 | 7 | 95 | 132 |
| 210 | 97 | 222 | 209 | 157 | 162 | 57 | 218 | 134 | 88 | 253 | 83 | 177 | 13 | 34 | 32 |
| 212 | 155 | 180 | 30 | 233 | 130 | 85 | 40 | 207 | 243 | 151 | 8 | 138 | 82 | 90 | 9 |
| 178 | 144 | 197 | 165 | 59 | 208 | 5 | 182 | 92 | 238 | 128 | 86 | 221 | 99 | 23 | 153 |
| 224 | 239 | 135 | 237 | 123 | 190 | 38 | 18 | 66 | 147 | 184 | 244 | 236 | 149 | 28 | 109 |
| 166 | 74 | 106 | 58 | 62 | 94 | 63 | 150 | 195 | 46 | 77 | 203 | 122 | 42 | 67 | 101 |
| 187 | 45 | 76 | 248 | 133 | 11 | 26 | 116 | 2 | 80 | 60 | 159 | 168 | 216 | 214 | 170 |
| 154 | 199 | 43 | 33 | 234 | 140 | 249 | 174 | 98 | 231 | 211 | 55 | 143 | 24 | 61 | 78 |
| 49 | 131 | 254 | 16 | 31 | 121 | 189 | 139 | 75 | 176 | 12 | 37 | 220 | 227 | 100 | 68 |
| 252 | 50 | 172 | 255 | 185 | 89 | 10 | 29 | 164 | 235 | 127 | 181 | 126 | 0 | 215 | 186 |
| 148 | 129 | 198 | 70 | 108 | 79 | 158 | 196 | 188 | 48 | 173 | 52 | 3 | 137 | 72 | 36 |
| 115 | 202 | 107 | 104 | 15 | 112 | 167 | 230 | 141 | 246 | 183 | 205 | 96 | 91 | 136 | 53 |

Figure 6: The dynamic S-Box generated with *key_hex*$_2$*.(S-Box2)*

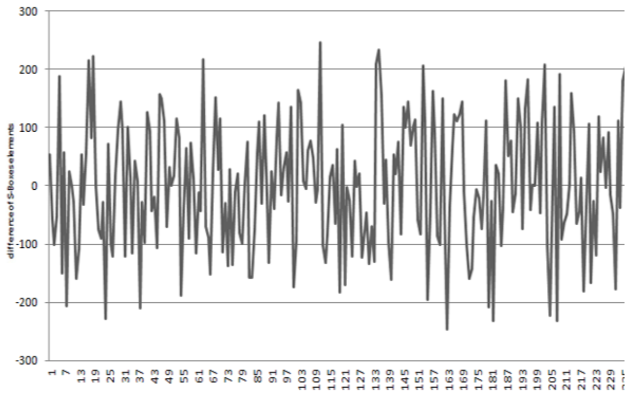


Figure 7: Plot of the difference of the S-Box elements (S-Box1 and S-Box2)

**7– Refrences**

[1] Encrption Availbale at : *http://en.wikipedia.org/wiki/Encryption*

[2] Federal Information Processing Standards, “Advanced Encryption Standard (AES)” Publication 197, November 26 - 2001

[3] Kazys KAZLAUSKAS, Janunius KAZLAUSKAS,” Key-Dependent S-Box Generation in AES Block Cipher System” , Inoformatica Volume: 20 – 2009

[4] Using Cipher Key to Generate Dynamic S-Box in AES Cipher System, *Razi Hosseinkhani, H.Haj Seyyed Javadi*, Internationl Journal Of Computer Sciences And Security, Volume 6, Issue 1

[5] BMP file format Available at : http:/*/en.wikipedia.org/wiki/BMP_file_format*

[6] Michael J.Quinn, Designing efficient algorithms for parallel computers, University of New Hamoshire, 1987