INFORMATION SECURITY THROUGH DIGITAL IMAGE STEGANOGRAPHY USING MULTILEVEL AND COMPRESSION TECHNIQUE

Brajesh Mishra
Research Scholar, Integral University, Lucknow

Manish Madhava Tripathi
Associate Professor, CSE Department, Integral University, Lucknow

ABSTRACT-- One of the most important factor of information technology and communication has been the security of information. In the age of Internet it is genuine issue that leads to the development of various techniques that makes the data secure with one of the issue that how strong the concept is? Cryptography is created for securing the secrecy of communication and many methods have been developed to encrypt and decrypt the data in order to keep the message secure. On October, 2, 2000, The National Institute of Standards and Technology (NIST) announced Rijndael as the new Advanced Encryption Standard (AES). A predecessor to the AES was Data Encryption Standard (DES) which was considered to be insecure because of its vulnerability to brute force attacks. DES was a standard from 1977 and stayed until the mid 1990's. However, by the mid 1990s, it was clear that the DES's 56-bit key was no longer big enough to prevent attacks mounted on contemporary computers, which were thousands of times more powerful than those available when the DES was standardized. The AES is a 128 bit Symmetric block Cipher. But it is sometime not enough to make the content of data more secure, then hiding message concept developed that is called steganography. Steganography is the technique to hide information in other information. There are many carrier file formats are used, but the most popular are digital images. Digital images are more preferred because of their frequency on the Internet and there are large variety of steganographic techniques some of them are more complex and all of them have their respective strong and weak points. Systems without these weaknesses offer only a relative small capacity for steganographic messages. The best-known steganographic method that works in the spatial domain is the LSB (Least Significant Bit), which replaces the least significant bits of pixels selected to hide the information. The newly developed algorithm F5 withstands visual and statistical attacks, yet it still offers a large steganographic capacity. F5 implements matrix encoding to improve the efficiency of embedding. Thus it reduces the number of necessary changes. F5 employs permutative straddling to uniformly spread out the changes over the whole steganogram. The thesis proposed a concept that provide an image platform for user to input image and a textbox to insert texts. Once the proposed algorithm is adapted, the stego-image is encrypted and then compressed, which send over the communication channel to get received at receiver end. The image steganography, cryptography and data compression techniques are implemented to make the data strongly secure to not to get accessed by any intruder. This is a secure messaging technique that allows the message to get transferred from sender to receiver end by utilizing the minimum bandwidth with high security.

Keywords: Data security, steganography, compression, congestion, data encryption

1. INTRODUCTION

The idea of information hiding is nothing new in the history. As early as in ancient Greece there were attempts to hide a message in trusted media to deliver it across the enemy territory. The idea of using vinegar as an invisible ink to write the message on a paper is really old. This message is invisible to human eye until we heat the paper and the vinegar turns dark. In the modern era of digital communication, we can think about similar ideas and use for example digital images as a medium for hiding a message. The word “steganography” comes from two Greek words: steganos (covered) and graphs (writing) and often refers to secret writing or data hiding. In this thesis, we will constrain ourselves to (digital) image steganography (further referred as steganography) only to be able to practically demonstrate our results. The approach and final results from this thesis can be easily used for steganography in other digital media, such as video, music etc. In this chapter, we will describe the basics of image steganography and give some examples of algorithms how to hide data into an image [12]. We will start by defining some basic concepts of information hiding in terms of the so-called prisoners’ problem and giving the notation which will be used throughout the whole thesis. Furthermore, we will describe some techniques how to detect the presence of a message and hence how to break steganographic schemes. Finally, we will mention the problem of “public key steganography”, where similarly to public key cryptography we want to use different key (public key) to embed the message and another key (private key) to extract the message.

1.1 LSB Modification (Simple LSB Substitution Scheme)

LSB modification is perhaps the most popular method to embed a message into cover data. As its name suggests, this method works by modifying the least significant bit of one of the RGB values of the pixel data. The secret message data is then scattered pseudo-randomly across the image. This technique is analogous to the spread spectrum communication technique of frequency hopping.

This method is quite effective against human detection because it is difficult for the human eye to discern an LSB modified pixel. Also, any modifications that are made could easily be attributed to “noise” that may already be contained in the image.
However, computer generated images, such as those generated by vector drawing applications like Adobe Illustrator or Macromedia Flash, do not contain much noise and would therefore make a poor choice as cover data.

Wang et al. [8] proposed a model to describe their LSB substitution scheme. Suppose that the embedded data is secret image $C$, while the host multimedia is host image $H$. Both $C$ and $H$ are 8-bit gray images. Using simple LSB substitution, the rightmost $k$ least significant bits of $H$ will be replaced by $C$. $k$ is denoted as the length of LSB, i.e., $[\text{LSB}]=k$. $C$ is defined by decomposing the bit streams of $C$ into several $k$-bit units and treating each unit as a single pixel. Also, let $R$ be the $k$-bit residual image, which is derived by extracting the rightmost $k$ least significant bits from each pixel in the host image $H$. To increase security, the pixel location of $C$ is randomized by a bijection (i.e., one-to-one and onto) mapping function into a meaningless image $C''[12]$.

The LSB substitution scheme proposed by Wang et al. [8] in a more general situation. Let the secret image $C$ form a bit string $S = s_0s_1…s_{n-1}$, where $n$ is the number of secret bits and $s_i$ represents a bit, for $i = 0, 1, 2, …, n-1$. For the $k$-bit simple LSB substitution approach, the bits in host image $H$, which will be replaced by the bit string $S$, form a bit string $R = r_0r_1…r_{m-1}$. We will refer to the bit string $S$ as the secret string and to the bit string $R$ as the replaced string. As shown in Fig. 2, each bit $s_i$ in $S$ will replace the bit $r_i$ in $R$. The value of $k$ is usually equal to 1, 2, 3, or 3 and each of the $k$ adjacent bits of $R$ in Fig. 2 come from the same pixel of the host image. The idea of Wang et al.’s LSB substitution scheme is that each $k$-bit string of $S$ is transformed into another $k$-bit string before replacing $R$. Then we extend the $k$-bit string to the $l$-bit string, denoted as the matching string, where $l \geq k$. $l$ is the length of a matching string. Each $l$-bit string of $S$ will be transformed into another $l$-bit string before replacing $R$, therefore we call our new scheme as the $l$-bit transforming LSB substitution scheme. Note that the LSB substitution scheme proposed by Wang et al. is a special case of our new scheme with $l = k$.

![Figure 1.1: A comparison of an LSB altered color tone.](Image 134x481 to 421x303)

![Figure 1.2: An example of a bit string S and a bit string R, where l = 6 and k = 3.](Image 198x417 to 398x505)

While 23-bit true-color RGB data formats are best suited for LSB modification, it is possible to use this method with 8-bit color-index data formats. This can be tricky, however, because the palette is much smaller and pixel luminescence variation may be much greater and more easily detected. Therefore, it is wise to attempt LSB modification with a grayscale or monochromatic cover image. There are, however, problems with LSB modification. For one, this method will only work with raw image data. Lossy image compression formats, such as JPEG, do not store images in an RGB format and are therefore not as forgiving to simple bit manipulation. Another problem with LSB manipulation is that if the stego data is compressed with a lossy compression algorithm, the secret message may be destroyed.

![Figure 1.3: The model of LSB substitution scheme.](Image 28x318 to 72x814)

### 1.2 F5

While JPEG files are not as tolerant to bit manipulation as uncompressed image files, it is still possible to use them as cover data. The key is to know where to hide the information. The JPEG algorithm works by dividing an image into several 8 x 8 pixel matrices. Discrete cosine transform (DCT) coefficients are then calculated for each matrix.
The coefficients are then multiplied by a quantization matrix. The results are then rounded off to the nearest integer. The rounded numbers are then further compressed and the results are saved. It is in these DCT values where we can hide our data. A typical approach involves slightly altering a set of the largest DCT coefficients [14]. These larger values contain the most “energy” and would therefore produce the least amount of distortion in the image. Another approach is to choose DCT coefficients that fall into a particular range so as to avoid perception. The popular JPEG steganography algorithms, F5 and JSteg, both use DCT modification to embed data. And while both algorithms generally escape human detection, they are both detectable through statistical analysis [13].

Some well-known steganographic algorithms scatter the message over the whole carrier medium. Many of them have a bad time complexity. They get slower if we try to exhaust the steganographic capacity completely. Straddling is easy, if the capacity of the carrier medium is known exactly. However, we cannot predict the shrinkage for F3, because it depends on which bit is embedded in which position [35]. We merely can estimate the expected capacity.

The straddling mechanism used with F5 shuffles all coefficients using a permutation first. Then, F5 embeds into the permuted sequence. The shrinkage does not change the number of coefficients (only their values). The permutation depends on a key derived from a password. F5 delivers the steganographically changed coefficients in its original sequence to the Huffman coder. With the correct key, the receiver is able to repeat the permutation. The permutation has linear time complexity \( O(n) \). Ron Crandall [1] introduced matrix encoding as a new technique to improve the embedding efficiency. F5 possibly is the first implementation of matrix encoding. If most of the capacity is unused in a steganogram, matrix encoding decreases the necessary number of changes. Let us assume that we have a uniformly distributed secret message and uniformly distributed values at the positions to be changed. One half of the message causes changes, the other half does not. Without matrix encoding, we have an embedding efficiency of 2 bits per change. Because of the shrinkage produced by F3, the embedding efficiency is even a bit lower, e. g. 1.5 bits per change. For example, if we embed a very short message comprising only 217 bytes (1736 bits), F3 changes 1157 places in the image. F5 embeds the same message using matrix encoding with only 359 changes, i. e. with an embedding efficiency of 3.8 bits per change. The following example shows what happened in detail. We want to embed two bits \( x_1, x_2 \) in three modifiable bit places \( a_1, a_2, a_3 \) changing one place at most. We may encounter these four cases:

\[
x_1 = a_1 \oplus a_3, x_2 = a_2 \oplus a_3 \Rightarrow \text{change nothing}
\]

\[
x_1 = a_1 \oplus a_3, x_2 = a_2 \oplus a_3 \Rightarrow \text{change } a_1
\]

\[
x_1 = a_1 \oplus a_3, x_2 = a_2 \oplus a_3 \Rightarrow \text{change } a_2
\]

In all four cases we do not change more than one bit. In general, we have a code word \( a \) with \( n \) modifiable bit places for \( k \) secret message bits \( x \). Let \( f \) be a hash function that extracts \( k \) bits from a code word. Matrix encoding enables us to find a suitable modified code word \( a' \) for every \( a \) and \( x \) with \( x = f(a') \), such that the Hamming distance \( d(a, a') \leq d_{\text{max}} \). We denote this code by an ordered triple \((d_{\text{max}}, n, k)\); a code word with \( n \) places will be changed in no more than \( d_{\text{max}} \) places to embed \( k \) bits. F5 implements matrix encoding only for \( d_{\text{max}} = 1 \). For \( (1, n, k) \), the code words have the length \( n = 2k - 1 \). Neglecting shrinkage, we get a change density \( D(k) = \frac{1}{1 + n} = \frac{1}{2^k} \) and an embedding rate \( R(k) = \frac{k}{n+1} = \frac{k}{2^k - 1} \).

Using the change density and the embedding rate we can define the embedding efficiency \( W(k) \). It indicates the average number of bits we can embed per change: \( W(k) = R(k)D(k) = \frac{k}{2^k - 1} \). The embedding efficiency of the \((1, n, k)\) code is always larger than \( k \). Table 1 shows that the rate decreases with increasing efficiency. Hence, we can achieve high efficiency with very short messages only [15,16]. Below table shows the dependencies between the message bits \( x \) and the changed bit places \( a' \). We assign the dependencies with the “binary coding” of \( j \) to column \( a' \). So we can determine the hash function very fast. \( f(a) = \oplus_{i=1}^{n} a_i \).

<table>
<thead>
<tr>
<th>( k )</th>
<th>( N )</th>
<th>change density</th>
<th>Embedding rate</th>
<th>Embedding efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>50.00%</td>
<td>100.00%</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>25.00%</td>
<td>66.67%</td>
<td>2.67</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>6.25%</td>
<td>32.86%</td>
<td>3.33</td>
</tr>
<tr>
<td>4</td>
<td>15</td>
<td>12.50%</td>
<td>26.67%</td>
<td>3.27</td>
</tr>
<tr>
<td>5</td>
<td>31</td>
<td>3.12%</td>
<td>16.13%</td>
<td>5.16</td>
</tr>
<tr>
<td>6</td>
<td>63</td>
<td>1.56%</td>
<td>9.52%</td>
<td>6.09</td>
</tr>
<tr>
<td>7</td>
<td>127</td>
<td>0.78%</td>
<td>5.51%</td>
<td>7.06</td>
</tr>
<tr>
<td>8</td>
<td>255</td>
<td>0.39%</td>
<td>3.13%</td>
<td>8.03</td>
</tr>
<tr>
<td>9</td>
<td>511</td>
<td>0.20%</td>
<td>1.76%</td>
<td>9.02</td>
</tr>
</tbody>
</table>

\textit{Figure 1.4: Connection between change density and embedding rate}
We find the bit place
\[ s = x \oplus f(a) \]
that we have to change. The changed code word results in
\[ a' = a, \text{ if } s = 0 \] (\( x_a = f(a) \))
\[ a' = (a_1, a_2, \ldots, a_n), \text{ otherwise} \]

We can find an optimal parameter \( k \) for every message to embed and every carrier medium providing sufficient capacity, so that the message just fits into the carrier medium. For instance, if we want to embed a message with 1000 bits into a carrier medium with a capacity of 50000 bits, then the necessary embedding rate is \( R = 1000 : 50000 = 2\% \). This value is between \( R(k=8) \) and \( R(k=9) \) in Table 1. We choose \( k = 8 \), and are able to embed 50000 : 255 = 196 code words with a length \( n = 255 \). The (1, 255, 8) code could embed 196 \( \times \) 8 = 1568 bits. If we chose \( k = 9 \) instead, we could not embed the message completely[12,13].

2. LITERATURE REVIEW

The most of today’s steganographic systems use images as cover object because people often transmit digital images over email and other communication media. Several methods exist to utilize the concept of Steganography as well as plenty algorithms have been proposed in this regard. To gather knowledge in this particular research field, we have concentrated on some techniques and methods which are described below. Least significant bit (LSB) insertion is a common and simple approach to embed information in a cover object. For images as a covering media, the LSB of a pixel is replaced with an M’s bit. If we choose a 24-bit image as cover, we can store 3 bits in each pixel by modifying the LSBs of R, G and B array. To the human eye, the resulting stego image will look identical to the cover image [1, 2].

Hiding data in the features of images is also an important technique which uses the LSB modification concept. In this method, to hide data in an image the least significant bits (LSB) of each pixel is modified sequentially in the scan lines across the image in raw image format with the binary data. The portion, where the secret message is hidden is degraded while the rest remain untouched. An attacker can easily recover the hidden message by repeating the process [2, 3]. An interesting application of steganography and cryptography has been developed by Sutaoe, M.S., Khandare, M.V, where a steganography system is designed for encoding and decoding a secret file embedded into an image file using random LSB insertion method. In that method, the secret data are spread out among the cover image in a seemingly random manner. The key used to generate pseudorandom numbers, which will identify where, and in what order the hidden message is laid out. The advantage of this method is that it incorporates some cryptography in that diffusion is applied to the secret message [4]. The next interesting application of steganography is developed by M. Dobsicek, where the content is encrypted with one key and can be decrypted with several other keys. In this the relative entropy between encrypt and one specific decrypt key corresponds to the amount of information [5].

In 2007, Nameer N. EL-Emam proposed an algorithmic approach to obtain data security using LSB insertion steganographic method. In this approach, high security layers have been proposed to make it difficult to break through the encryption of the input data and confuse steganalysis too [6]. S. K. Bandypadhyay, Deb Nath Bhattacharyya, Swarnendu Mukherjee, Debashis Ganguliy, Poulam Das in 2008 has also proposed a heuristic approach to hide huge amount of data using LSB steganography technique. In their method, they have first encoded the data and afterwards the encoded data is hidden behind a cover image by modifying the least significant bits of each pixel of the cover image. The resultant stego-image was distortion less. Also, they have given much emphasis on space complexity of the data hiding technique [7]. There is also a good method proposed by G. Sahoo and R. K. Tiwari in 2008. Their proposed method works on more than one image using the concept of file hybridization. This particular method implements the cryptographic technique to embed two information files using steganography. And due to this reason they have used a stego key for the embedding process [8]. The method proposed by Adnan Abdul-Aziz Gutub, and Manal Mohammad Fattani in 2007 is also a good one. Their approach hides secret information bits within the letters benefiting from their inherited points. To note the specific letters holding secret bits, the scheme considers the two features, the existence of the points in the letters and the redundant Arabic extension character. They have used the pointed letters with extension to hold the secret bit ‘one’ and the un-pointed letters with extension to hold ‘zero’. This steganography technique is found attractive to other languages having similar texts to Arabic such as Persian and Urdu [9].

The paper “Multimedia Information Networking and Security (MINES)”, written by Yongzhen Zheng in 2011 proposes an approach to identify steganography software by Core Instructions Template Matching. First, we take the process of operations to the pixels in the LSB Replacement steganography algorithm as Core Instructions, and we have analyzed the implementation of Core Instructions in a variety of typical steganography software based on LSB Replacement Steganography algorithm, build the corresponding template for each of implementations. And then we determine whether the software is a LSB replacement steganography software according to the existence of codes matched with the Core Instructions Template [10].
The experimental results show that this method can identify some of the steganography software based on the LSB Replacement Steganography algorithm.

The paper “Digital image steganography: Survey and analysis of current methods” in Signal Processing, Volume 90, Issue 3, March 2010, Pages 727-752 by Abbas Cheddad, Joan Condell, Kevin Curran, Paul Mc Kevitt suggest that Steganography is the science that involves communicating secret data in an appropriate multimedia carrier, e.g., image, audio, and video files. It comes under the assumption that if the feature is visible, the point of attack is evident, thus the goal here is always to conceal the very existence of the embedded data. Steganography has various useful applications. However, like any other science it can be used for ill intentions. It has been propelled to the forefront of current security techniques by the remarkable growth in computational power, the increase in security awareness by, e.g., individuals, groups, agencies, government and through intellectual pursuit. Steganography's ultimate objectives, which are undetectability, robustness (resistance to various image processing methods and compression) and capacity of the hidden data, are the main factors that separate it from related techniques such as watermarking and cryptography.

This paper provides a state-of-the-art review and analysis of the different existing methods of steganography along with some common standards and guidelines drawn from the literature [11]. This paper concludes with some recommendations and advocates for the object-oriented embedding mechanism. Steganalysis, which is the science of attacking steganography, is not the focus of this survey but nonetheless will be briefly discussed.

3. IMPLEMENTATION
3.1. Steganography: Two steganography algorithms have been implemented in this work
3.1.1. LSB: Given an l-bit transforming LSB substitution scheme, we can construct a bipartite graph $G = (A \cup B, E)$, where vertex set $A = \{a0, a1, \ldots, aN-1\}$, vertex set $B = \{b0, b1, \ldots, bN-1\}$, and edge set $E = \{(ai, bj)\}$ for each $0 \leq i, j \leq N - 1$. $N$ is equal to $2l$. Suppose $S$ and $R$ are the secret string and the replaced string, respectively. Then, a maximal matching of $G$ has cardinality $N$ and represents a transformation which transforms the secret string $S$ into another secret string $S'$. We denote such maximal matching as $(S, S')$. Because we are only interested in the matching with cardinality $N$, any matching appearing in the following article is a maximal matching. When weights are assigned to all edges such that the weight of a matching $(S, S')$ is equal to $MSES_R$, where $MSES_R$ denotes the mean square error between host image $H$ and stego-image $Z$ with $R$ substituted by $S'$, a matching with minimum weight is an optimal solution. For any value of $l$, not all such weighted bipartite graphs exist. But, if $k$ divides $l$, the weighted bipartite graph can be created easily. Fig. 3.1 shows the case of $l = 3 \times k$, where $S0, S1, \ldots, Sm-1$ are all k-bit string of $S$ and $R0, R1, \ldots, Rm-1$ are all k-bit string of $R$, and where $m$ is the size of the host image. We scan strings $S$ and $R$ simultaneously, from left to right and process each l-bit string of $S$ and $R$ as follows. Consider each l-bit string $S3i+0||S3i+1||S3i+2$ of $S$ and l-bit string $R3i+0||R3i+1||R3i+2$ of $R$, where $i = 0, 1, \ldots, (m/3) - 1$ and the symbol “||” means the string concatenation operator. Let $x$ be the value of $S3i+0 ||S3i+1||S3i+2$, denoted as $val(S3i+0 ||S3i+1||S3i+2)$. Also, let $y0 = val(R3i+0)$, $y1 = val(R3i+1)$, and $y2 = val(R3i+2)$. Suppose $j$ is an integer in $0, 1, \ldots, N - 1$. $j$ can be represented as an l-bit string $J$, and let $j0, j1,$ and $j2$ be the value of left k-bit string, middle k-bit string, and right k-bit string of $J$, respectively. If the l-bit string $S3i+0||S3i+1||S3i+2$ matches to the l-bit string $J$, the l-bit string $J$ is used to substitute the l-bit string $R3i+0||R3i+1||R3i+2$. Therefore, the square error in this case is $(y0 - y0)2 + (j1 - y1)2 + (j2 - y2)2$. Assume that the weight of edge $(ax, by)$ is denoted as cost $(ax, by)$. The cost $(ax, by)$ in the current stage is calculated by adding $(y0 - y0)2 + (j1 - y1)2 + (j2 - y2)2$ to the cost $(ax, by)$ in the previous stage. Therefore we have the following equation, current cost $(ax, by) = \{previous \ cost \ (ax, by)\} + (y0 - y0)2 + (j1 - y1)2 + (j2 - y2)2$, for $j = 0, 1, \ldots, N - 1$.

![Figure 3.1 An example of the weight assignment of a bipartite graph for bit strings S and R, where l = 3 × k, x0, y1, and y2 are the values of the corresponding bit strings.](image-url)
Note that the optimal matching found by the matching approach must be kept in order to extract the embedded message. It needs $l \times 2l$ bits, denoted as string $O$, to represent the optimal matching. We describe the embedding process and extracting process as follows.

**Algorithm Embedding-Process** ($C$, $H$, $k$, $l$)
**Input:** a secret image $C$ and a host image $H$ for $l$-bit transforming LSB substitution with $k$-bit simple LSB substitution;
**Output:** stego-image $Z$ and optimal matching $O$;
   a) Get secret string $S$ from $C$ and replaced string $R$ from $H$
   b) Create bipartite graph $G$ and find an optimal matching $O$
   c) Transform $S$ into $S'$ by the optimal matching $O$
   d) Create stego-image $Z$ by replacing $R$ with $S'$

**Algorithm Extracting-Process** ($Z$, $O$, $k$, $l$)
**Input:** a stego-image $Z$ and an optimal matching $O$ for $l$-bit transforming LSB substitution with $k$-bit simple LSB substitution;
**Output:** secret image $C$;
   a) Get embedded string $S'$ from $Z$
   b) Transform $S'$ into $S$ by the optimal matching $O$
   c) Construct secret image $C$ from $S$

### 3.1.2. F5: The algorithm F5 has the following coarse structure:
1. Start JPEG compression. Stop after the quantisation of coefficients.
2. Initialise a cryptographically strong random number generator with the key derived from the password.
3. Instantiate a permutation (two parameters: random generator and number of coefficients).
4. Determine the parameter $k$ from the capacity of the carrier medium, and the length of the secret message.
5. Calculate the code word length $n = 2k - 1$.
6. Embed the secret message with $(1, n, k)$ matrix encoding.
   a) Fill a buffer with $n$ nonzero coefficients.
   b) Hash this buffer (generate a hash value with $k$ bit-places.).
   c) Add the next $k$ bits of the message to the hash value (bit by bit, xor).
   d) If the sum is 0, the buffer is left unchanged. Otherwise the sum is the buffer’s index $1 \ldots n$, the absolute value of its element has to be decremented.
   e) Test for shrinkage, i.e. whether we produced a zero. If so, adjust the buffer (eliminate the 0 by reading one more nonzero coefficient, i.e. repeat step 6a beginning from the same coefficient). If no shrinkage occurred, advance to new coefficients behind the actual buffer. If there is still message data continue with step 6a.
   f) Continue JPEG compression (Huffman coding etc.).

### 3.2 Encryption: Two types of encryption strategy has been used in this work symmetric and Asymmetric, for symmetric encryption AES has been used and for Asymmetric encryption RSA has been used.

**3.2.1 AES:** For the AES algorithm, the length of the input block, the output block and the State is 128 bits. This is represented by $Nb = 4$, which reflects the number of 32-bit words (number of columns) in the State. For the AES algorithm, the length of the Cipher Key, $K$, is 128, 192, or 256 bits. The key length is represented by $Nk = 4, 6$, or $8$, which reflects the number of 32-bit words (number of columns) in the Cipher Key [6,19]. For the AES algorithm, the number of rounds to be performed during the execution of the algorithm is dependent on the key size. The number of rounds is represented by $Nr$, where $Nr = 10$ when $Nk = 4$, $Nr = 12$ when $Nk = 6$, and $Nr = 14$ when $Nk = 8$. For both its Cipher and Inverse Cipher, the AES algorithm uses a round function that is composed of four different byte-oriented transformations: 1) byte substitution using a substitution table (S-box), 2) shifting rows of the State array by different offsets, 3) mixing the data within each column of the State array, and 4) adding a Round Key to the State.

The Cipher is described in the pseudo code in Fig. 3.2. The individual transformations-SubBytes(), ShiftRows(), MixColumns(), and AddRoundKey()- process the State and are described in the following subsections, the array $w[[]]$ contains the key schedule, which is described. As shown in above figure, all $Nr$ rounds are identical with the exception of the final round, which does not include the MixColumns() transformation. Appendix B presents an example of the Cipher, showing values for the State array at the beginning of each round and after the application of each of the four transformations described in the following sections.
Cipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)]) begin byte state[4,Nb] state = = AddRoundKey(state, w[0,Nb-1]) for round = 1 step 1 to Nr-1 SubBytes(state) ShiftRows(state) MixColumns(state) AddRoundKey(state, w[round*Nb, (round+1)*Nb-1]) end for SubBytes(state) ShiftRows(state) AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1]) out = state end

The SubBytes() transformation is a non-linear byte substitution that operates independently on each byte of the State using a substitution table (S-box). This S-box, which is invertible, is constructed by composing two transformations:

1. Take the multiplicative inverse in the finite field GF(2^8), described in Sec. 3.2; the element {00} is mapped to itself.
2. Apply the following affine transformation (over GF(2)):

   \[ b = b' + b'' + b''' + b'\prime \]

   \((i + 4) \bmod 8 (i + 5) \bmod 8 (i + 6) \bmod 8 \)

   \(i = 0 \) for \(0 \leq i < 8\), \(b_i\) is the \(i\)th bit of the byte, and \(c_i\) is the \(i\)th bit of a byte \(c\) with the value \(63\) or \(01100011\). Here and elsewhere, a prime on a variable (e.g., \(b'\)) indicates that the variable is to be updated with the value on the right.

ShiftRows() Transformation In the ShiftRows() transformation, the bytes in the last three rows of the State are cyclically shifted over different numbers of bytes (offsets). The first row, \(r = 0\), is not shifted. Specifically, the ShiftRows() transformation proceeds as follows:

\[ s = s \bmod 4 \quad \text{and} \quad 0 \leq c < Nb, \quad (3,3) \quad r, c, r+c \bmod (r, Nb) \bmod Nb \]

where the shift value \(shift(Nb)\) depends on the row number, \(r\), as follows (recall that \(Nb = 4\)): \(shift(1,4) = 1\); \(shift(2,4) = 2\); \(shift(3,4) = 3\). (5.4) This has the effect of moving bytes to “lower” positions in the row (i.e., lower values of \(c\) in a given row), while the “lowest” bytes wrap around into the “top” of the row (i.e., higher values of \(c\) in a given row). MixColumns() Transformation The MixColumns() transformation operates on the State column-by-column, treating each column as a four-term polynomial as described in Sec. 4.3. The columns are considered as polynomials over GF(2^8) and multiplied modulo \(x^4 + 1\) with a fixed polynomial \(a(x)\), given by \(a(x) = (03)x^3 + (01)x^2 + (01)x + (02)\). As described in Sec. 3.3, this can be written as a matrix multiplication. Let \(s(x) = a(x)\cdot x(x)\) be the Affine Transformation in the AddRoundKey() transformation, a Round Key is added to the State by a simple bitwise XOR operation. Each Round Key consists of \(Nb\) words from the key schedule (described in Sec. 3.2). Those \(Nb\) words are each added into the columns of the State, such that where \(w[i]\) are the key schedule words described in Sec. 3.2, and \(round\) is a value in the range 0 \leq round \leq Nr. In the Cipher, the initial Round Key addition occurs when \(round = 0\), prior to the first application of the round function [17,18]. The application of the AddRoundKey().

Key Expansion

The AES algorithm takes the Cipher Key, \(K\), and performs a Key Expansion routine to generate a key schedule. The Key Expansion generates a total of \(Nb \cdot (Nr + 1)\) words: the algorithm requires an initial set of \(Nb\) words, and each of the \(Nr\) rounds requires \(Nb\) words of key data. The resulting key schedule consists of a linear array of 4-byte words, denoted \([w[i]]\), with \(i\) in the range 0 \leq i \leq Nb(Nr + 1).

The expansion of the input key into the key schedule proceeds according to the pseudo code.

SubWord() is a function that takes a four-byte input word and applies the S-box (Sec. 3.1.1) to each of the four bytes to produce an output word. The function RotWord() takes a word \([a0,a1,a2,a3]\) as input, performs a cyclic permutation, and returns the word \([a1,a2,a3,a0]\). The round constant word array, \(Rcon[i]\), contains the values given by \([x^{-1}, \{00\}, \{00\}, \{00\}]\), with \(x^{-1}\) being powers of \(x = (x + 2)\) in the field GF(2^8), as discussed in Sec. 3.2 (note that \(i\) starts at 1, not 0). From Fig. 11, it can be seen that the first \(Nk\) words of the expanded key are filled with the Cipher Key. Every following word, \(w[i]\), is equal to the XOR of the previous word, \(w[i-1]\), and the word \(Nk\) positions earlier, \(w[i-Nk]\). For words in positions that are a multiple of \(Nk\), a transformation is applied to \(w[i-Nk]\) prior to the XOR, followed by an XOR with a round constant, \(Rcon[i]\). This transformation consists of a cyclic shift of the bytes in a word (RotWord()), followed by the application of a table lookup to all four bytes of the word (SubWord()).

Inverse Cipher

The Cipher functions in Sec. 3.1 can be inverted and then implemented in reverse order to produce a straightforward Inverse Cipher for the AES algorithm. The individual transformations used in the Inverse-Cipher:

InvShiftRows(), InvSubBytes(), InvMixColumns(), and AddRoundKey() = process the State and are described in the following subsections. InvCipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)]) begin byte state[4,Nb] state = = AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1]) // See Sec. 5.1 for round = Nr-1 step -1 downto 1 InvShiftRows(state) // See Sec. 5.3.1 InvSubBytes(state) AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1]) InvMixColumns(state) end for InvShiftRows(state) InvSubBytes(state) AddRoundKey(state, w[0,Nb-1]) out = state end. Pseudo Code for the Inverse Cipher. InvShiftRows() Transformation InvShiftRows() is the inverse of the ShiftRows() transformation. The bytes in the last three rows
of the State are cyclically shifted over different numbers of bytes (offsets). The first row, $r = 0$, is not shifted. The bottom three rows are cyclically shifted by $\text{Nb} - \text{shift}(r, \text{Nb})$ bytes, where the shift value $\text{shift}(r, \text{Nb})$ depends on the row number, and is given in equation (see Sec. 3.1.2). Specifically, the $\text{InvShiftRows()}$ transformation proceeds as follows: $s = s$ for $0 < r < 4$ and $0 \leq c < \text{Nb}$.

**Transformation** $\text{InvSubBytes()}$ is the inverse of the byte substitution transformation, in which the inverse S-box is applied to each byte of the State. This is obtained by applying the inverse of the affine transformation (5.1) followed by taking the multiplicative inverse in $\mathbb{F}(2^8)$.

**InvMixColumns()** Transformation $\text{InvMixColumns()}$ is the inverse of the MixColumns() transformation. $\text{InvMixColumns()}$ operates on the State column-by-column, treating each column as a four-term polynomial as described in Sec. 3.3. The columns are considered as polynomials over $\mathbb{F}(2^8)$ and multiplied modulo $x^4 + 1$ with a fixed polynomial $a^{-1}(x)$, given by $a^{-1}(x) = \{0b\}x^3 + \{0d\}x^2 + \{09\}x + \{0e\}$.

For the Equivalent Inverse Cipher, the following pseudo code is added at the end of the Key Expansion routine (Sec. 3.2):

```plaintext
For i=0 step 1 to (Nr+1)*Nb-1 dw [ij] = w[i] end for round = 1 step 1 to Nr-1 InvMixColumns(dw[round*Nb, (round+1)*Nb-1]) // note change of type end for Note that, since InvMixColumns operates on a two-dimensional array of bytes while the Round Keys are held in an array of words, the call to InvMixColumns in this code sequence involves a change of type (i.e. the input to InvMixColumns() is normally the State array, which is considered to be a two-dimensional array of bytes, whereas the input here is a Round Key computed as a one-dimensional array of words) [7].
```

### 3.2.2. RSA: The RSA public-key cryptosystem, which was invented at MIT in 1966 by Ronald Rivest, Adi Shamir and Leonard Adleman.

The public key in this cryptosystem consists of the value $n$, which is called the modulus, and the value $e$, which is called the public exponent. The private key consists of the modulus $n$ and the value $d$, which is called the private exponent.

An RSA public-key / private-key pair can be generated by the following steps:

1. Generate a pair of large, random primes $p$ and $q$.
2. Compute the modulus $n$ as $n = pq$.
3. Select an odd public exponent $e$ between 3 and $n-1$ that is relatively prime to $p-1$ and $q-1$.
4. Compute the private exponent $d$ from $e$, $p$ and $q$. (See below.)
5. Output $(n, e)$ as the public key and $(n, d)$ as the private key.

The encryption operation in the RSA cryptosystem is exponentiation to the eth power modulo $n$: $c = \text{ENCRYPT} (m) = me \mod n$. The input $m$ is the message; the output $c$ is the resulting ciphertext. In practice, the message $m$ is typically some kind of appropriately formatted key to be shared. The actual message is encrypted with the shared key using a traditional encryption algorithm. This construction makes it possible to encrypt a message of any length with only one exponentiation. The decryption operation is exponentiation to the $d$th power modulo $n$:

$$m = \text{DECRYPT} (c) = cd \mod n.$$  

The relationship between the exponents $e$ and $d$ ensures that encryption and decryption are inverses, so that the decryption operation recovers the original message $m$. Without the private key $(n, d)$ (or equivalently the prime factors $p$ and $q$), it’s difficult (by CONJECTURE 6) to recover $m$ from $c$. Consequently, $n$ and $e$ can be made public without compromising security, which is the basic requirement for a public-key cryptosystem.

The fact that the encryption and decryption operations are inverses and operate on the same set of inputs also means that the operations can be employed in reverse order to obtain a digital signature scheme following Diffie and Hellman’s model. A message can be digitally signed by applying the decryption operation to it, i.e., by exponentiating it to the $d$th power: $s = \text{SIGN} (m) = md \mod n$. The digital signature can then be verified by applying the encryption operation to it and comparing the result with and/or recovering the message: $m = \text{VERIFY} (s) = se \mod n$ [8,9]. In practice, the plaintext $m$ is generally some function of the message, for instance a formatted one-way hash of the message. This makes it possible to sign a message of any length with only one exponentiation. Figure 1 gives a small example showing the encryption of values $m$ from 0 to 9 as well as decryptions of the resulting ciphertexts. The exponentiation is optimized as suggested above. To compute $m3 \mod n$, one first computes $m2 \mod n$ with one modular squaring, then $m3 \mod n$ with a modular multiplication by $m$. The decryption is done similarly: One first computes $c2 \mod n$, then $c3 \mod n$, $c6 \mod n$, and $c6 \mod n$ by alternating modular squaring and modular multiplication.

### Computing the Private Exponent

Let $n$ be the product of two distinct prime numbers $p$ and $q$, and let $e$ be the public exponent as defined above. Let $L = \text{LCM} (p-1, q-1)$ denote the least common multiple of $p-1$ and $q-1$. The private exponent $d$ for the RSA cryptosystem is any integer solution to the congruence $de \equiv 1 \mod L$. The value $d$ is the inverse of $e$ modulo $L$. The requirement that $e$ be relatively prime to $p-1$ and $q-1$ ensures that an inverse exists. Modular inverses are easy to find with the Extended Euclidean Algorithm or similar methods. The RSA
The compression technique is lossless. The algorithm is implemented which reduces its size and can be easily transmitted over communication channel to send it at receiver end. The compression technique is lossless so there are no chances for data loss.

5. REFERENCES