CHAPTER 3

METHODOLOGY

3.1 Introduction

This chapter presents a details description of the proposed all techniques which is used in the proposed system. The details description of the system overview, system concept, System features and the proposed system structure.

3.2 Phases of Techniques

1. Phase One:
   The detailed description of the proposed technique of cryptography and mathematical equation of Rijndael Method.

2. Phase Two:
   The detailed description of the proposed technique of steganography and mathematical equation about Statistical Technique.

3. Phase Three:
   The description of the PE-File with the technique (without explaining the full detail of its structure because the proposed system does not deal with this structure but the work goes beyond it), which being used as a cover file.

4. Phase Four:
   The details description of the proposed technique of information:
   a. (Encryption and Hiding) in PE_File.
   b. (Retract and Decryption) from PE-File.
3.3 Phase One

In this phase described the cipher Rijndael. First presented the mathematical basis necessary for understanding the specifications followed by the design rationale and the description itself. Subsequently, the implementation aspects of the cipher and its inverse are treated. This is followed by the motivations of all design choices and the treatment of the resistance against known types of attacks. By giving security claims and goals, the advantages and limitations of the cipher, ways how it can be extended and how it can be used for functionality other than block encryption/decryption.

3.3.1 Advance Encryption Standard (AES)/ Rijndael.

AES is short for Advanced Encryption Standard and is a United States encryption standard defined in Federal Information Processing Standard (FIPS), published in November 2001. It was ratified as a federal standard in May 2002. One should not compare AES with RSA, another standard algorithm, as RSA is a different category of algorithm. Bulk encryption of information itself is seldom performed with RSA. RSA is used to transfer other encryption keys for use by AES for example, and for digital signatures. AES is a symmetric encryption algorithm processing data in block of 128 bits. Under the influence of a key, a 128-bit block is encrypted by transforming it in a unique way into a new block of the same size. AES is symmetric since the same key is used for encryption and the reverse transformation, decryption. The only secret necessary to keep for security is the key. AES may be configured to use different key-lengths, the standard defines 3 lengths and the resulting algorithms are named AES-128, AES-192 and AES-256 respectively to indicate the length in bits of the key. Each additional bit in the key effectively doubles the strength of the algorithm, when
defined as the time necessary for an attacker to stage a brute force attack, i.e. an 
exhaustive search of all possible key combinations in order to find the right 
one.
This is the second version of the Rijndael documentation. The main 
difference with the first version is the correction of a number of errors and 
inconsistencies, the addition of a motivation for the number of rounds, the 
addition of some figures in the section on differential and linear cryptanalysis, 
the inclusion of Brain Gladman’s performance figures and the specification of 
Rijndael extensions supporting block and key lengths of 160 and 224 bits 
(Selebrorg, 2004).

3.3.2 Mathematical Preliminaries

Several operations in Rijndael are defined at byte level, with bytes 
representing elements in the finite field GF ($2^8$). Other operations are defined in 
terms of 4-byte words. In this section introduced the basic mathematical concepts 
(Selebrorg, 2004).

3.3.2.1 The Field GF ($2^8$)

The elements of a finite field [LiNi86] can be represented in several 
different ways. For any prime power there is a single finite field, hence all 
representations of GF ($2^8$) are isomorphic. Despite this equivalence, the 
representation has an impact on the implementation complexity. For the classical 
polynomial had chosen representation. A byte $b$, consisting of bits $b_7 b_6 b_5 b_4 b_3 b_2$ 
$b_1 b_0$, is considered as a polynomial with coefficient in $\{0, 1\}$, $b_7 x^7 + b_6 x^6 + b_5 x^5 + b_4$ 
x$^4 + b_3 x^3 + b_2 x^2 + b_1 x + b_0 x$. Example; the byte with hexadecimal value ‘57’ 
(binary 01010111) corresponds with polynomial [$x^6 + x^4 + x^2 + x + 1$] (Jafer, 2006).
a) **Addition**

In the polynomial representation, the sum of two elements is the polynomial with coefficients that are given by the sum modulo 2 (i.e., 1+1 = 0) of the coefficients of the two terms. Example; ‘57’ + ‘83’ = ‘D4’, or with the polynomial notation; In binary notation had; “01010111” + “10000011” = “11010100”. Clearly, the addition corresponds with the simple bitwise EXOR (denoted by ⊕) at the byte level. All necessary conditions are fulfilled to have an Abelian group; internal, associative, neutral element (‘00’), inverse element (every element is its own additive inverse) and commutative. As every element is its own additive inverse, subtraction and addition is the same (Johannes, 2003).

b) **Multiplication**

In the polynomial representation, multiplication in GF (2^8) corresponds with multiplication of polynomial modulo an irreducible binary polynomial of degree 8. A polynomial is irreducible if it has no divisors other than 1 and itself. For Rijndael, this polynomial is called m(x) and given by m(x) = x^8 + x^4 + x^3 + x +1, or ‘11B’ in hexadecimal representation. Clearly, the result will be a binary polynomial of degree below 8. Unlike for addition, there is no simple operation at byte level. The multiplication defined above is associative and there is a neutral element (‘01’). For any binary polynomial b(x) of degree below 8, the extended algorithm of Euclid can be used to compute polynomials a(x), c(x) such that

\[ b(x) a(x) + m(x) c(x) = 1 \ldots \ldots \ldots (1) \]

Hence, a(x). b(x) mod m(x) = 1 or

\[ b^{-1}(x) = a(x) \mod m(x) \ldots \ldots \ldots (2) \]
Moreover, it holds that \(a(x) \cdot b(x) + c(x) = a(x) \cdot b(x) + a(x) + c(x)\). It follows that the set of 256 possible byte values, with the EXOR as addition and the multiplication defined as above has the structure of the finite field GF\( (2^8)\) (Johnson, 2005).

c) Multiplication by x

If there is multiply \(b(x)\) by the polynomial \(x\), then: \(b_7 x^8 + b_6 x^7 + b_4 x^5 + b_3 x^4 + b_2 x^3 + b_1 x^2 + b_0 x\). \(b(x)\) is obtained by reducing the above result modulo \(m(x)\). If \(b_7 = 0\), this reduction is the identity operation. If \(b_7 = 1\), \(m(x)\) must be subtracted (i.e., EXORed). It follows that multiplication by \(x\) (hexadecimal ‘02’) can be implemented at byte level as a left shift and a subsequent conditional bitwise EXOR with ‘1 B’. This operation is denoted by \(b = x \times (a)\). In dedicated hardware, xtime takes only 4 EXORs. Multiplication by higher powers of \(x\) can multiplication by repeated application of xtime. By adding intermediate results, multiplication by any constant can be implemented (Johnson, 2005).

3.3.2.2 Polynomials with Coefficients in GF \( (2^8)\)

Polynomials can be defined with coefficients in GF \((2^8)\). In this way, a 4-byte vector corresponds with a polynomial of degree below 4. Polynomials can be added by simply adding the corresponding coefficients. As the addition in GF \((2^8)\) is the bitwise EXOR, the addition of two vectors is a simple bitwise EXOR. Multiplication is more complicated. Assume that has two polynomials over GF \((2^8)\) (Jafer, 2006):

\[
a(x) = a_3 x^3 + a_2 x^2 + a_1 x + a_0 \quad \text{and} \quad b(x) = b_3 x^3 + b_2 x^2 + b_1 x + b_0 \ldots \ldots \ldots (3)
\]
a) **Multiplication by x**

If multiply \( b(x) \) by the polynomial \( x_1 \) then:

\[
b_3 x^4 + b_2 x^3 + b_1 x^2 + b_0 x \ldots \ldots \ldots (4)
\]

\( x \otimes b(x) \) is obtained by reducing the above result modulo \( 1 + x^4 \). This gives

\[
b_2 x^3 + b_1 x^2 + b_0 x + b_3 \ldots \ldots \ldots (5)
\]

The multiplication by \( x \) is equivalent to multiplication by a matrix as above with all

\( a_i = '00' \) except \( a_1 = '01' \). Let \( c(x) = x \otimes b(x) \). Then:

\[
\begin{bmatrix}
c_0 \\
c_1 \\
c_2 \\
c_3
\end{bmatrix} =
\begin{bmatrix}
00 & 00 & 00 & 01 \\
01 & 00 & 00 & 00 \\
00 & 01 & 00 & 00 \\
00 & 00 & 01 & 00
\end{bmatrix}
\begin{bmatrix}
b_0 \\
b_1 \\
b_2 \\
b_3
\end{bmatrix}
\]

Hence, multiplication by \( x \), or powers of \( x \), corresponds to a cyclic shift of the bytes inside the vector (Johnson, 2005).

### 3.3.3 Design Rationale

The three criteria taken into account in the design of Rijndael are the following:

a) Resistance against all known attacks;

b) Speed and code compactness on a wide range of platforms;

c) Design simplicity.

In most ciphers, the round transformation has the Feistel structure. In this structure typically part of the bits of the intermediates state are simply transposed unchanged to another position. The round transformation of Rijndael does not have the Feistel structure. Instead, the round transformation is composed of three distinct invertible uniform transformations, called layers. By “uniform”, means that every bit of the state is treated in a similar way. The specific choices for the different layers are for a large part based on the application of the Wide Trail...
Strategy, a design method to provide resistance against linear and differential cryptanalysis. In the Wide Trial Strategy, every layer has its own function:

- **The linear mixing layer**: guarantees high diffusion over multiple rounds.
- **The non-linear layer**: parallel application of S-boxes that have optimum worst-case nonlinearity properties.
- **The key addition layer**: A simple EXOR of the Round Key to the intermediate state.

Before the first round, a key addition layer is applied. The motivation for this initial key addition is the following. Any layer after the last key addition in the cipher (or before the first in the context of known-plaintext attacks) can be simply peeled off without knowledge of the key and therefore does not contribute to the security of the cipher. (e.g., the initial and final permutation in the DES). Initial or terminal key addition is applied in several designs, e.g., IDEA, SAFER and Blowfish. In order to make the cipher and its inverse more similar in structure, the linear mixing layer of the last round is different from the mixing layer in the other rounds. It can be shown that this does not improve or reduce the security of the cipher in any way. This is similar to the absence of the swap operation in the last round of the DES (Selebrorg, 2004).

### 3.3.4 Specification

Rijndael is an iterated block cipher with a variable block length and a variable key length. The block length and the key length can be independently specific to 128, 192 or 256 bits (Johannes, 2003).
3.3.4.1 The State, The Cipher Key and The Number of Rounds

The different transformations operate on the intermediate result, called the state. The state can be pictured as a rectangular array of bytes. This array has four rows, the number of columns is denoted byNb and is equal to the block length divided by 32. The Cipher Key is similarly pictured as a rectangular array with four rows. The number of columns of the Cipher Key is denoted byNk and is equal to the key length divided by 32. These representations are illustrated in Figure 1. In some instances, these blocks are also considered as one-dimensional array of 4-bytes vectors, where each vector consists of the corresponding column in the rectangular array representation. These arrays hence have lengths of 4, 6 or 8 respectively and indices in the ranges 0..3, 0..5, 0..7. 4-bytes vectors will sometimes be referred to as words. Where it is necessary to specify the four individual bytes within a 4-bytes vector or word the notation (a, b, c, d) will used where a, b, c and d are the bytes at positions 0, 1, 2 and 3 respectively within the column, vector or word being considered (Johannes, 2003).

![Figure 3.1: Example of State (with Nb = 6) and Cipher Key (with Nk = 4) layout.](image)

The input and output used by Rijndael at its external interface are considered to be one-dimensional arrays of 8-bit bytes numbered upwards from 0 to the 4*Nb-1. These blocks hence have lengths of 16, 24 or 3 bytes and array
indices in the ranges 0..15, 0..23 or 0..31. The Cipher Key is considered to be a one-dimensional arrays of 8-bit bytes numbered upwards from 0 to the $4^*N_k - 1$. These blocks hence have lengths of 16, 24 or 32 bytes and array indices in the ranges 0..15, 0..23 or 0..31. The cipher input bytes (the “plaintext” if the mode of use is ECB encryption) are mapped onto the state bytes in the order $a_{0.0}, a_{1.0}, a_{2.0}, a_{3.0}, a_{0.1}, a_{1.1}, a_{2.1}, a_{3.1}, a_{4.1}...$, and the bytes of the cipher key are mapped onto the array in the order $k_{0.0}, k_{1.0}, k_{2.0}, k_{3.0}, k_{0.1}, k_{1.1}, k_{2.1}, k_{3.1}, k_{4.1}...$ At the end of the cipher operation, the cipher output is extracted from the state by taking the state bytes in the same order. Hence if the one-dimensional index of a byte within a block is $n$ and the two-dimensional index is $(i, j)$. Had:

$$i = n \mod 4; \quad j = \lfloor n / 4 \rfloor; \quad n = I + 4^* j$$

Moreover, the index $i$ is also the byte number within a 4-byte vector or word and $j$ is index for the vector or word within the enclosing block. The number of rounds is denoted by $N_r$ and depends on the value $N_b$ and $N_k$. It is given in table one (Johannes, 2003).

**Table 3.1:**

<table>
<thead>
<tr>
<th>$N_k$</th>
<th>$N_b = 4$</th>
<th>$N_b = 6$</th>
<th>$N_b = 8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N_k = 4$</td>
<td>10</td>
<td>12</td>
<td>14</td>
</tr>
<tr>
<td>$N_k = 6$</td>
<td>12</td>
<td>12</td>
<td>14</td>
</tr>
<tr>
<td>$N_k = 8$</td>
<td>14</td>
<td>14</td>
<td>14</td>
</tr>
</tbody>
</table>
3.3.4.2 Key Schedule

The Round Keys are derived from the Cipher Key by means of the key schedule. This consists of two components: the key Expansion and the Round Key Selection. The basic principle is the following (Johannes, 2003):

1. The total number of Round Key bits is equal to the block length multiplied by the number of rounds plus 1. (e.g., for a block length of 128 bits and 10 rounds, 1408 Rounds Key bits are needed).

2. The Cipher Key is expanded into an expanded key.

3. Round Keys are taken from this expanded key in the following way: the first Round Key consists of the first Nb words, the second one of the following Nb words, and so on.

a) Key Expansion

The expanded key is a linear array of 4-byte words and is denoted by \( W[Nb^* (nr+1)] \). The first \( Nk \) words contain the cipher key. All other words are defined recursively in terms of words with smaller indices. The key expansion function depends on the value of \( Nk \); there is a version for \( Nk \) equal to or below 6, and a version for \( Nk \) above 6. For \( Nk \leq 6 \), has (Selebrorg, 2004):

```c
Key expansion (byte Key [4* Nk] word W[Nb^* (nr+1)])
{
    for(i = 0; i < Nk; i++)
        W[i] = (Key[4*i],Key[4*i+1],Key[4*i+2],Key[4*i+3]);
    for(i = Nk; i < Nb * (Nr + 1); i++)
    {
        temp = W[i - 1];
        if (i % Nk == 0)
            temp = SubByte(RotByte(temp)) ^ Rcon[i / Nk];
        W[i] = W[i - Nk] ^ temp;
    }
}
```

**Figure 3.2: Function of key Expansion for \( Nk \leq 6 \).**
In this description, SubByte (W) is a function that returns a 4-byte word in which each byte is the result of applying the Rijndael S-box to the byte at the corresponding position in the input word. The function RotByte (w) returns a word in which the bytes are a cyclic permutation of those in its input such that the input word (a, b, c, d) produces the output word (b, c, d, a). It can be seen that the first Nk words are filled with the cipher key. Every following word w[i] is equal to the EXOR of the previous word W [i-1] and the word Nk position earlier W [i-Nk]. For words in positions that are a multiple of Nk, a transformation consists of a cyclic shift of the bytes in a word (RotByte), followed by the application of a table lookup to all four bytes of the word (SubByte). For Nk > 6, has (Selebrorg, 2004):

```c
KeyExpansion(byte Key[4*Nk] word W[ Nb*(Nn+1)])
{
    for(i = 0; i < Nk; i++)
        W[i] = [key[4*i],key[4*i+1],key[4*i+2],key[4*i+3]];

    for(i = Nk; i < Nb * (Nn + 1); i++)
    {
        temp = W[i - 1];
        if (i % Nk == 0)
            temp = SubByte(RotByte(temp)) ^ Rcon[i / Nk];
        else if (i % Nk == 4)
            temp = SubByte(temp);
        W[i] = W[i - Nk] ^ temp;
    }
}
```

**Figure 3.3: Function of key Expansion for Nk > 6.**

The difference with the scheme for Nk < is that for i-4 a multiple of Nk, SubByte is applied to W[i-1] prior to the EXOR. The round constants are independent of Nk and defined by:

\[ Rcon[i] = (RC[i], '00', '00', '00') \]
With $RC[I]$ representing an element in $GF(2^8)$ with a value of $x^{(i-1)}$ so that:

$$RC[1] = 1 \text{ (i.e. } '01')$$
$$RC[i] = x \text{ (i.e. } '02') \cdot (RC[i-1]) = x^{(i-1)}$$

b) Round Key Selection

Round Key I is given by the round key buffer words $W[Nb * i]$ to $W[Nb * (i+1)]$. This is illustrated in Figure 3.4. The Key schedule can be implemented without explicit use of the array $W[Nb * (Nr + 1)]$. For implementations where RAM is scare, the Round Keys can be computed on-the-fly using a buffer of $Nk$ words with almost no computational overhead (Seleborg, 2004).

![Figure 3.4: Key Expansion and Round Key Selection for Nb = 6 and Nk = 4.](image)

3.3.4.3 The Cipher

The cipher Rijndael consists of:

1. An initial Round Key addition;
2. $Nr$-1 Rounds;
3. A final round.

In pseudo C code, this gives:
The key expansion can be done on beforehand and Rijndael can be specified in terms of the expanded key (Jafer, 2006).

The Expanded Key shall always be derived from the Cipher Key and never be specified directly. There are however no restrictions on the selection of the Cipher Key itself (Jafer, 2006).

### 3.3.5 Implementation Aspects

The Rijndael cipher is suited to be implemented efficiently on a wide range of processors and in dedicated hardware. Will concentrate on 8-bit processors, typical for current smart cards and on 32-bit processors, typical for PCs (Jafer, 2006).
3.3.5.1 8-Bit Processor

On an 8-bit processor, Rijndael can be programmed by simply implementing the different component transformations. This is straightforward for rowshift and for the round key addition. The implementation of bytesub requires a table of 256 bytes. The round key addition; bytesub and rowshift can be efficiently combined and executed serially per State byte. Indexing overhead is minimised by explicitly coding the operation for every state byte. The transformation MixColumn requires matrix multiplication in the field GF ($2^8$). This can be implemented in an efficient way. Illustrate it for one column:

\[
\text{Tmp} = a[0] ^ a[1] ^ a[2] ^ a[3] \; / * \text{a is a byte array} */
\]

\[
\text{Tm} = a[0] ^ a[1] \; \text{Tm} = \text{xtime(Tm)} \; a[0] ^ = \text{Tm} ^ \text{Tmp} \; \]

\[
\text{Tm} = a[1] ^ a[2] \; \text{Tm} = \text{xtime(Tm)} \; a[1] ^ = \text{Tm} ^ \text{Tmp} \; \]

\[
\text{Tm} = a[2] ^ a[3] \; \text{Tm} = \text{xtime(Tm)} \; a[2] ^ = \text{Tm} ^ \text{Tmp} \; \]

\[
\text{Tm} = a[3] ^ a[0] \; \text{Tm} = \text{xtime(Tm)} \; a[3] ^ = \text{Tm} ^ \text{Tmp} \; \]

**Figure 3.7: Transformation MixColumn multiplication in the field GF ($2^8$).**

This description is for clarity. In practice, coding is of course done in assembly language. To prevent timing attacks, attention must be paid that xtime is implemented to take a fixed number of cycles, independent of the value of its argument. In practice this can be achieved by using a dedicated table-lookup (Johannes, 2003).

Obviously, implementing the key expansion in a single shot operation is likely to occupy too much RAM in a smart card. Moreover, in most applications, such as debit cards or electronic purses, the amount of data to be enciphered, deciphered or that is subject to a MAC is typically only a few blocks per session. Hence, not much performance can be gained by expanding the key only once for multiple applications of the block cipher. The key expansion can be implemented
in a cyclic buffer of 4*max (Nb, Nk) bytes. The round key is updated in between rounds. All operations in this key update can be implemented efficiently on byte level. If the cipher key length and the blocks length are equal or differ by a factor 2, the implementation is straightforward. If this is not the case, an additional buffer pointer is required (Johannes, 2003).

3.3.5.2 32-Bit Processor

a) Round Transformation

The different steps of the round transformation can be combined in a single set of table lookups, allowing for very fast implementations on processors with word length 32 or above. In this section, it is explained how this can be done. Expressed one column of the round output e in terms of bytes of the round input a. In this section, ai,j denotes the byte of a in row i and column j, aj denotes the column j of State a. For the key addition and the MixColumn transformation, it will be as shown in figure 3.8 (Johannes, 2003):

\[
\begin{bmatrix}
    e_{0,j} \\
e_{1,j} \\
e_{2,j} \\
e_{3,j}
\end{bmatrix} = \begin{bmatrix}
    d_{0,j} \\
d_{1,j} \\
d_{2,j} \\
d_{3,j}
\end{bmatrix} \oplus \begin{bmatrix}
    k_{0,j} \\
k_{1,j} \\
k_{2,j} \\
k_{3,j}
\end{bmatrix} \quad \text{and} \quad \begin{bmatrix}
    d_{0,j} \\
d_{1,j} \\
d_{2,j} \\
d_{3,j}
\end{bmatrix} = \begin{bmatrix}
    02 & 03 & 01 & 01 \\
    01 & 02 & 03 & 01 \\
    01 & 01 & 02 & 03 \\
    03 & 01 & 01 & 02
\end{bmatrix} \begin{bmatrix}
    c_{0,j} \\
c_{1,j} \\
c_{2,j} \\
c_{3,j}
\end{bmatrix}
\]

Figure 3.8: Shows the Key Addition and the MixColumn Transformation.

For the ShiftRow and the ByteSub transformations, it will be as shown in figure 3.9:

\[
\begin{bmatrix}
    c_{0,j} \\
c_{1,j} \\
c_{2,j} \\
c_{3,j}
\end{bmatrix} = \begin{bmatrix}
    b_{0,j} \\
b_{1,j-C1} \\
b_{2,j-C2} \\
b_{3,j-C3}
\end{bmatrix} \quad \text{and} \quad b_{i,j} = S[a_{i,j}]
\]

Figure 3.9: Shows the ShiftRow and the ByteSub Transformations.
In this expression the column indices must be taken modulo Nb. By substitution, the above expressions can be combined into:

\[
\begin{bmatrix}
e_{0,j} \\
e_{1,j} \\
e_{2,j} \\
e_{3,j}
\end{bmatrix} =
\begin{bmatrix}
02 & 03 & 01 & 01 \\
01 & 02 & 03 & 01 \\
01 & 01 & 02 & 03 \\
03 & 01 & 01 & 02
\end{bmatrix}
\begin{bmatrix}
a_{0,j} \\
a_{1,j-C1} \\
a_{2,j-C2} \\
a_{3,j-C3}
\end{bmatrix}
\oplus
\begin{bmatrix}
k_{0,j} \\
k_{1,j} \\
k_{2,j} \\
k_{3,j}
\end{bmatrix}
\]

**Figure 3.10: The Column Indices Taken Modulo Nb by Substitution.**

The matrix multiplication can be expressed as a Linear A combination of vectors:

\[
\begin{bmatrix}
e_{0,j} \\
e_{1,j} \\
e_{2,j} \\
e_{3,j}
\end{bmatrix} =
\begin{bmatrix}
02 & 01 \\
01 & 01 \\
01 & 01 \\
03 & 01
\end{bmatrix}
\begin{bmatrix}
a_{0,j} \\
a_{1,j-C1} \\
a_{2,j-C2} \\
a_{3,j-C3}
\end{bmatrix}
\oplus
\begin{bmatrix}
01 & 01 \\
02 & 01 \\
02 & 01 \\
03 & 02
\end{bmatrix}
\begin{bmatrix}
k_{0,j} \\
k_{1,j} \\
k_{2,j} \\
k_{3,j}
\end{bmatrix}
\]

**Figure 3.11: Linear A Combination of Vectors.**

The multiplication factors \( S[a_{ij}] \) of the four vectors are obtained by performing a table lookup on input bytes \( a_{ij} \) in the S-box table \( S[256] \):

\[
\begin{align*}
T_0[a] &= \begin{bmatrix}
S[a] \\
S[a] \\
S[a] \\
S[a] \\
\end{bmatrix} \\
T_1[a] &= \begin{bmatrix}
S[a] \\
S[a] \\
S[a] \\
S[a] \\
\end{bmatrix} \\
T_2[a] &= \begin{bmatrix}
S[a] \\
S[a] \\
S[a] \\
S[a] \\
\end{bmatrix} \\
T_3[a] &= \begin{bmatrix}
S[a] \\
S[a] \\
S[a] \\
S[a] \\
\end{bmatrix}
\end{align*}
\]

**Figure 3.12: Defined Tables \( T_0 \) to \( T_3 \).**
These are 4 tables with 256 4-byte word entries and make up for 4Kbyte of total space. Using these tables, the round transformation can be expressed as:

\[ e_j = T_0[a_{3,j}] \oplus T_1[a_{4,j-C1}] \oplus T_2[a_{2,j-C2}] \oplus T_3[a_{3,j-C3}] \oplus k_j \]

... ... ...(6)

Hence, a table-lookup implementation with 4 Kbytes of tables takes only 4 table lookups and 4 EXORs per column per round. It can be seen that Ti[a] = Rotbyte (Ti-1[a]). At the cost of 3 additional rotations per round per column, the table-lookup implementation can be realized with only one table, i.e., with a total table size of 1Kbyte. Had (Johannes, 2003):

\[ e_j = k_j \oplus T_0[h_{4,j}] \oplus \text{Rotbyte}(T_3[h_{2,j-C1}] \oplus \text{Rotbyte}(T_2[h_{1,j-C2}] \oplus \text{Rotbyte}(T_1[h_{3,j-C3}])) \]

... ... ...(7)

The code-size (relevant in applets) can be kept small by including code to generate the tables instead of the tables themselves. In the final round, there is no MixColumn operation. This boils down to the fact that the tables must be used instead of the T tables. The need for additional tables can be suppressed by extracting the S table from the T tables by masking while executing the final round. Most operation in the key expansion can be implemented by 32-bit word EXORs. The additional transformations are the application of the S-box and a cyclic shift over 8-bits. This can be implemented very efficiently (Johnson, 2005).

b) Parallelism

It can be seen that there is considerable parallelism in the round transformation. All four component transformations of the round act in a parallel
way on bytes, rows or columns of the state. In the table-lookup implementation, all table lookups can in principle be done in parallel. The EXORs can be done in parallel for the most part also. The key expansion is clearly of a more sequential nature; the value of $W[i-1]$ is needed for the computation of $W[i]$. However, in most applications where speed is critical, the key expansion has to be done only once for a large number of cipher executions. In applications where the cipher key changes often (in extremis once per application of the Block Cipher), the key expansion and the cipher Rounds can be done in parallel (Johnson, 2005).

c) Hardware Suitability

The cipher is suited to be implemented in dedicated. There are several trade-offs between area and speed possible. Because the implementation in software on general-purpose processors is already very fast, the need for hardware implementations will very probably be limited to two specific cases (Johnson, 2005):

1. Extremely high speed chip with no area restrictions, the $T$ tables can be hardwired and the EXORs can be conducted in parallel.

2. Compact co-processor on a smart Card to speed up Rijndael execution, for this platform typically the s-box and the Xtime (or the complete MixColumn) operation can be hardwired.

3.3.5.3 The Inverse Cipher

In the table-lookup implementation it is essential that the only non-linear step (ByteSub) is the first transformation in a round and that the rows are shifted before MixColumn is applied. In the inverse of round, the order of the transformations in the round is reversed, and consequently the non-linear step
will end up being the last step of the inverse round and the rows are shifted after the application of (the inverse of) MixColumn. The inverse of a round can therefore not be implemented with the table lookups described above. This implementation aspect has been anticipated in the design. The structure of Rijndael is such that the sequence of transformations of its inverse is equal to that of the cipher itself, with the transformations replaced by their inverses and a change in the key schedule. This is shown in the following subsections (Jafer, 2006).

a) **Inverse of a Two-Round Rijndael Variant**

```c
InvRound(State, RoundKey)  
{  
  AddRoundKey(State, RoundKey);  
  InvMixColumn(State);  
  InvShiftRow(State);  
  InvByteSub(State);  
}
```

**Figure 3.13: Shows The Inverse of A Round.**

```c
InvFinalRound(State, RoundKey)  
{  
  AddRoundKey(State, RoundKey);  
  InvShiftRow(State);  
  InvByteSub(State);  
}
```

**Figure 3.14: The Inverse of the Final Round.**

b) **Algebraic Properties**

In deriving the equivalent structure of the inverse cipher, made using of two properties of the component transformations. First, the order of ShiftRow and ByteSub is indifferent. ShiftRow simply transposes the bytes and has no effect on the byte values. ByteSub works on individual bytes, independent of their position. Second, the sequence (Jafer, 2006):

```c
AddRoundKey(state, RoundKey) ;
```
InvMixColumn (State) ;

Can be replaced by:

InvMixColumn (State) ;
AddRoundKey (State, InvRoundKey) ;

With invRoundKey obtained by applying InvMixColumn to the corresponding RoundKey. This is based on the fact that for a linear transformation A, has A(x+k) = A(x) + A(k) (Jafer, 2006).

c) The Equivalent Inverse Cipher Structure

Using the properties described above, the inverse of the two-round Rijndael variant can be transformed into:

```
AddRoundKey(State, ExpandedKey+2*Nb) ;
InvByteSub(State);
InvShiftRow(State);
InvMixColumn(State);
AddRoundKey(State, I_ExpandedKey+Nb) ;
InvByteSub(State);
InvShiftRow(State);
AddRoundKey(State, ExpandedKey) ;
```

It can be seen that has again an initial Round Key addition, a round and a final round. The Round and the final round have the same structure as those of the cipher itself. This can be generalized to any number of rounds. Defined a round and the final round of the inverse cipher as follows (Jafer, 2006):

```
I_Round (State, I_RoundKey)
{
InvByteSub (State);
InvShiftRow (State);
InvMixColumn (State);
AddroundKey (State, I_RoundKey);
}

I_FinalRound (State, I_RoundKey)
{
InvByteSub (State);
InvShiftRow (State);
AddRoundKey (State, RoundKey0);
}
```
The Inverse of the Rijndael Cipher can now be expressed as follows:

```c
I_Rijndael (state, CipherKey) {
    I_KeyExpansion (CipherKey, I_ExpandedKey);
    AddRoundKey (State, I_ExpandedKey+ Nb*Nr);
    For(i=Nr-1; i>0; i--) Round(State,I_ExpandedKey+ Nb*i);
    FinalRound (State, I_ExpandedKey);
}
```

The key expansion for the Inverse Cipher is defined as follows:

1. Apply the key Expansion.
2. Apply InvMixColumn to all Rounds Keys except the first and the last one.

In pseudo C code, this gives:

```c
I_KeyExpansion(CipherKey,I_ExpandedKey) {
    KeyExpansion(CipherKey,I_ExpandedKey);
    For(i=1 ; i< Nr ; i++ )
        InvMixColumn(I_ExpandedKey + Nb*i);
}
```

d) Implementation of the Inverse Cipher

The choice of the MixColumn polynomial and the key expansion was partly based on cipher performance arguments. Since the inverse cipher is similar in structure, but uses a MixColumn transformation with another polynomial and (in some cases) a modified key schedule, performance degradation is observed on 8-bit processors. This asymmetry is due to the fact that the performance of the inverse cipher is considered to be less important than that of the cipher. In many application of a block cipher, the inverse cipher operation is not used. This is the case for the calculation of MACs, but also when the cipher is used in CFB-mode (Jafer, 2006).
3.4 Phase Two

In this phase described the statistical steganography technique and mathematical equation of this technique which is used in the proposed system.

3.4.1 Statistical Technique

Statistical steganography techniques utilize the existence of "1-bits" schemes, which embed one bit of information in a digital carrier. This is done by modifying the cover in such a way that some statistical characteristics change significantly if a "1" is transmitted. Otherwise, the cover is left unchanged. So the receiver must be able to distinguish unmodified covers from modified ones. A cover is divided into \( l \) disjoint blocks \( B_1 \ldots B_l \). A secret bit, \( m_i \), is inserted into the \( i \)th block by placing "1" into \( B_i \) if \( m_i = 1 \). Otherwise, the block is not changed in the embedding process. The detection of a specific bit is done via a test function which distinguishes modified block from unmodified block (Sellar, 2003):

\[
f(B_i) = \begin{cases} 
1 & \text{block } B_i \text{ was modified in the embedding process} \\
0 & \text{otherwise} 
\end{cases}
\]

\[... \ldots ... \ (8)\]

The function \( f \) can be interpreted as a hypothesis-test function and the test of null-hypothesis “block \( B_i \) was not modified” against the alternative hypothesis “block \( B_i \) was modified.” Therefore, the whole class of such steganography systems statistical steganography. The receiver successively applies \( f \) to all cover-block \( B_i \) in order to restore every bit of the secret message. The main question which remains to be solved is how such a function \( f \) in (8) can be constructed. If they interpret \( f \) as a hypothesis-testing function, they can use
the theory of hypothesis testing from mathematical statistics. Let us assume could find a formula \( h(B_i) \), which depends on some elements of the cover-block \( B_i \), and knew the distribution of \( h(B_i) \), in the unmodified block (i.e., the Hypothesis holds in this case) could then use standard procedure to test if \( h(B_i) \), equals or exceeds a specific value. If managed to alter \( h(B_i) \) in the embedding process in a way that its expected value is 0 if the block \( B_i \) was not modified, and expected value is much greater otherwise, could test whether \( h(B_i) \) equals zero under the given distribution of \( h(B_i) \). Statistical steganography techniques are, however, difficult to apply in many cases. First, a good test statistic \( h(B_i) \) must be found which allows distinction between modified and unmodified cover-blocks. Additionally, the distribution of \( h(B_i) \) must be known for a “normal” cover; in most cases, this is quite a difficult task. In practical implementations many (quite questionable) assumptions are made in order to determine a closed formula for this distribution. As an example, wanted to construct a statistical steganography algorithm out of pitas’ watermarking system, which is similar the patchwork approach of Bender et al. Suppose every cover-block \( B_i \) is a rectangular set of pixels \( p(i)n,m \). Furthermore, let \( S=\{s(i)n,m\} \) be a rectangular pseudorandom binary pattern of equal size, where the number of one is \( S \) equals the number of zeros. Would assume that both the sender and receiver have access to \( S \), which represents the stego-key in this application. The sender first splits the image block \( B_i \) into two sets, \( C_i \) and \( D_i \) of equal size (i.e., putting all pixels with indices \( (n,m) \) into set \( C \) where the corresponding key bit \( s_n,m \) equals zero) (Katzenbilker, 2000):

\[
C_i = \{p_n,m^{(i)} \in B_i | s_{n,m} = 1\} \\
D_i = \{p_n,m^{(i)} \in B_i | s_{n,m} = 0\}
\] ... ... ... (9)
The sender then adds a value \( k > 0 \) to all pixels in the subset \( C_i \) but leaves all pixels in \( D_i \) unchanged. In the last step, \( C_i \) and \( D_i \) are merged to form the marked image block \( B_i \). In order to extract the mark, the receiver reconstructs the sets \( C_i \) and \( D_i \). If the block contains a mark, all value in \( C_i \) will be larger than the corresponding values in the embedding step; thus testing the difference of the means of sets \( C_i \) and \( D_i \). If assumed that all pixels in both \( C_i \) and \( D_i \) are independent identically distributed random variables with an arbitrary distribution, the test statistic (Al_Mayyhee, 2005):

\[
q_i = \frac{C_i - D_i}{\sigma_i}
\]

where

\[
\sigma_i = \sqrt{\frac{\text{Var}[C_i] + \text{Var}[D_i]}{|S|/2}}
\]

\[\text{... ... ... (10)}\]

Where \( \overline{C_i} \) denotes the mean over all pixels in the set \( C_i \) and \( \text{Var}[C_i] \) the estimated variance of the random variables in \( C_i \), will follow a \( N(0,1) \) normal distribution asymptotically due to the central limit theorem. If a mark is embedded in the image block \( B_i \), the expected value of \( q \) will be greater than zero. The receiver is thus able to reconstruct the \( i \)th secret message bit by testing whether the statistic \( q_i \) of block \( B_i \) equals zero under the \( N(0,1) \) distribution (Katzenbisser, 2000; Sellar, 2003).
3.5 Phase Three

In this phase described the techniques related with PE-file, PE-file layout, PE-file format which is used in the proposed system.

3.5.1 Portable Executable File (PE-File)

The proposed system uses a portable executable file as a cover to embed an executable program as an example for the proposed system. This section is divided into three parts (Mega, 2004):

a) Techniques related with PE-file.

b) PE-file Layout.

c) PE-file format.

3.5.2 Techniques Related with PE-File.

Before looking inside the PE file, we should know special techniques some of which are (Martin, 2007):

a) General View of PE Files Sections

A PE file section represents code or data of some sort. While code is just code, there are multiple types of data. Besides read/write program data (such as global variables), other types of data in sections include application program interface (API) import and export tables, resources, and relocations. Each section has its own set of in-memory attributes, including whether the section contains code, whether it's read-only or read/write, and whether the data in the section is shared between all processes using the executable file. Sections have two alignment values, one within the desk file and the other in memory. The PE file header specifies both of these values, which can differ. Each section starts at an
offset that's some multiple of the alignment value. For instance, in the PE file, a
typical alignment would be 0x200. Thus, every section begins at a file offset
that's a multiple of 0x200. Once mapped into memory, sections always start on at
least a page boundary. That is, when a PE section is mapped into memory, the
first byte of each section corresponds to a memory page. On x86 CPUs, pages are
4KB aligned, while on the Intel Architecture IA-64, they're 8KB aligned.

b) **Relative Virtual Addresses (RVA)**

In an executable file, there are many places where an in-memory address
needs to be specified. For instance, the address of a global variable is needed
when referencing it. PE files can load just about anywhere in the process address
space. While they do have a preferred load address, you can't rely on the
executable file actually loading there. For this reason, it's important to have some
way of specifying addresses that are independent of where the executable file
loads. To avoid having hard coded memory addresses in PE files, RVAs are used.
An RVA is simply an offset in memory, relative to where the PE file was loaded.
For instance, consider an exe.File loaded at address 0x400000, with its code
section at address 0x401000. The RVA of the code section would be:

\[
\text{(Target address) } 0x401000 - \text{(load address) } 0x400000 = \text{(RAV)}
\]

To convert an RVA to an actual address, simply reverse the process: add
the RVA to the actual load address to find the actual memory address.
Incidentally, the actual memory address is called a Virtual Address (VA) in PE
parlance. Another way to think of a VA is that it's an RVA with the preferred
load address added in.
c) Importing Functions

When we use code or data from another DLL, we are importing it. When any PE files loads, one of the jobs of the windows loader is to locate all the imported functions and data and make those addresses available to the file being loaded.

3.5.3 PE-File Layout.

The PE file layout is shown in Figure 3.15. There are two unused spaces in PE file layout (Stive, 2008), and these unused spaces are suggested to hide a watermark. The size of the second unused space is different from one file to another (Stive, 2008).

<table>
<thead>
<tr>
<th>MS-DOS 2.0 Compatible.EXE Header</th>
<th>Base of Image Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unused</td>
<td>MS-DOS 2.0 Section</td>
</tr>
<tr>
<td>OEM Identifier</td>
<td>(For MS-DOS Compatibility Only)</td>
</tr>
<tr>
<td>OEM Information</td>
<td></td>
</tr>
<tr>
<td>Offset To PE Header</td>
<td></td>
</tr>
<tr>
<td>MS-DOS 2.0 Stub Program &amp; Relocation</td>
<td></td>
</tr>
<tr>
<td>Unused</td>
<td></td>
</tr>
<tr>
<td>PE Header</td>
<td></td>
</tr>
<tr>
<td>Section Headers</td>
<td></td>
</tr>
<tr>
<td>Image Pages</td>
<td></td>
</tr>
<tr>
<td>• Import info</td>
<td></td>
</tr>
<tr>
<td>• Export info</td>
<td></td>
</tr>
<tr>
<td>• Fix-up info</td>
<td></td>
</tr>
<tr>
<td>• Recourse info</td>
<td></td>
</tr>
<tr>
<td>• Debug info</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.15: Typical 32-bit Portable exe.File Layout (Stive, 2008).
3.5.4 PE-File Format

The header of PE format starts with MS-DOS stub that is used for printing a message, “This program cannot be run in DOS mode”, if the operating system can’t identify the PE on execution time. IMAGE_NT_HEADER located in the position after the MS-DOS stub has the information for the execution of a file, and consists of IMAGE_FILE_HEADER and IMAGE_OPTIONAL_HEADER. The IMAGE_FILE_HEADER has the information on the file, such as create time and machine type. The IMAGE_OPTIONAL_HEADER has the information on functions used in the file and on the start-address of the file on a memory, and the information is managed by IMAGE_DATA_DIRECTORY. A PE file except the header is composed of several sections that are basic unit of code or data within a PE or COFF file. IMAGE_SECTION_HEADER that is located in the position following to IMAGE_OPTIONAL_HEADER has the information on each section. The information consists of PointerToRawData, SizeOfRawData, VirtualAddress, and VirtualSize. The PointerToRawData and the SizeOfRawData respectively mean the position of each section and the size of each section on the file. The VirtualAddress and the VirtualSize respectively mean the position of each section and the size of each section on the memory. The size of each section on the file is a multiple of FileAlignment that is in IMAGE_OPTIONAL_HEADER. If the amount of the data of a section is smaller than the size of the section that is allotted on compile time, the slack space of the section occurs. The common sections used in the PE include a .text that has program binaries, .data, .idata that has information on export and import functions, .edata, and .rsrc section. An .idata section has the information on import functions used in executable files during the period of an execution (Stive, 2008).
3.6 Phase Four

In this phase described the system overview, system features, technique of information hiding in PE-File which is used in the proposed system and the proposed system structure.

3.6.1 System Overview

The most important reason behind the idea of this system is that the programmers always need to create a back door for all of their developed applications, as a solution to many problems such that forgetting the password. This idea leads the customers to feel that all programmers have the ability to hack their system any time. At the end of this discussion all customers always are used to employ trusted programmers to build their own application.

Programmers want their application to be safe anywhere without the need to build ethic relations with their customers. In this system a solution is suggested for this problem. The solution is to hide the password in the executable file of the same system and then other application to be retracted by the customer himself. Steganography needs to know all files format to find a way for hiding information in those files. This technique is difficult because there are always large numbers of the file format and some of them have no way to hide information in them.
3.6.2 System Concept

Concept of this system can be summarized as hiding the password or any information beyond the end of an executable file so there is no function or routine (open-file, read, write, and close-file) in the operating system to extract it. This operation can be performed in two alternative methods:

- Building the file handling procedure independently of the operating system file handling routines. In this case we need cancelling the existing file handling routines and developing new functions which can perform our need, with the same names. This way needs the customer to install the system application manually as shown in Figure 3.16.

- Developing the file handling functions depending on the existing file handling routines. This way can be performed remotely as shown in Figure 3.17.

The advantage of the first method is it doesn't need any additional functions, which can be identified by the analysts. The disadvantage of this method is it needs to be installed (can not be operated remotely).

The advantage of the second method is it can be executed remotely and suitable for networks and the internet applications. So we choose this concept to implementation in this research.
Figure 3.16: First Method of the System Concept
3.6.3 System Features

This system has the following features:

a) The cover file can be executed normally after hiding operation. Because the hidden information already hide after the end of file and thus can not be manipulated as the exe.File. Therefore, the cover file still natural, working normally and not effected, such as if the cover is exe.Files (WINDOWES XP SETUP) after hiding operation it'll continued working. In other words, the exe.File can be installed of windows.
b) There is no limitation on the hidden file size where you can hide any file of any size regardless of the size of hidden information by structure on the property of the exe. File, so that the exe.File can not identify the size of the exe.File, so can using type of exe.File such as JDK whose contain number of different size (72MB, 77MB or 65MB), other world disparity in the size of the executable files, so can hide any size inside it without guessing the real size of the information hidden by the attacker. Furthermore, when hide after the end of exe.File, there is no limitation of the size files which must be hiding after the end of exe.File, open space of any size.

c) It's very difficult to extract the hidden information it's difficult to find out the information hiding, that is because of three reasons:

- The information hiding was encrypted before hiding of the information by AES method; this method very strong, 128-bit key would be in theory being in range of a military budget within 30-40 years. An illustration of the current status for AES is given by the following example, where we assume an attacker with the capability to build or purchase a system that tries keys at the rate of one billion keys per second. This is at least 1 000 times faster than the fastest personal computer in 2004. Under this assumption, the attacker will need about 10 000 000 000 000 000 000 000 years to try all possible keys for the weakest version.

- The attacker impossible guessing the information hiding inside the exe. File because of couldn't guess the real size of (exe.File and information hiding).

- The information hiding should be decrypted after retract of the information.
d) The hidden information can be of any type of multimedia files (Text, Audio, Video or Image) of any size without limitation and also can hidden all type of multimedia files in the same time inside the same cover, so can put (Text, Image, Video and Audio) in one folder and compressed them and then choose the compressed folder as a information hiding, in that way can hidden all in the same time.

e) Virus detection programmes can't detect such as files, the principle of antivirus check are checking from beginning to end. When checking the exe.Files by antivirus, will checked it from beginning to end of it ,since the principle of information hiding for that system is after end of file, the antivirus discontinue checking in the end of file so didn't mention to anything inside the exe.File while doing scanning.

3.6.4 The Proposed System Structure

To protect the hidden information from retraction the system encrypts the information by the built-in encryption algorithm provided by the VB.net which is mentioned in chapter three. The flow chart of the hiding operation can be performed as shown in Figure 3.19. The flow chart of retraction operation can be performed as shown in Figure 3.21.
Procedure: Hide Operation.

Input: Hidden file name, Cover file name.

Output: Stego-File.

Begin:

Step (1): Open the cover file (PE-file).
Step (2): Assign a pointer to the end of file.
Step (3): Write the file name after the end of file (PE-file).
Step (4): Assign a pointer to PE-file after the hidden file name.
Step (5): Encrypt the hidden file.
Step (6): Write the encrypt content to the file cover (PE-file).

End.

Figure 3.18: Algorithm of the Hiding Operation Procedure.
Figure 3.19: Flow Chart of the Hiding Operation.
Procedure: Retraction Operation.
Input: Stego-file.
Output: Hidden information file.

Begin:
Step (1): Select the cover file.
Step (2): Get the end of file.
Step (3): If end of file pointer exist.
Begin:
Step (3-1): Read the name of hidden file.
Step (3-2): Read the hidden data.
Step (3-3): Decrypt the data using the file name as a key.
Step (3-4): Create a file using hidden file name.
Step (3-5): Write into the created file the decrypted data.
End;
Else
Step (4): Display a message (no hidden information).
End.

Figure 3.20: Algorithm of the Retraction Operation Procedure.
Start Operation

Select the Name of the Cover File

Go to the End of the File Pointer

If the Pointer Found?

Yes

Read the Name of the Hidden File

Read the Encrypt Hidden Data

Decrypt the Data Using the File Name as a key

Create a File with the Hidden File Name and Write the Decrypted Data on it

End Operation

No

Display a Message "No Hidden Information"

Read the Name of the Hidden File

Read the Encrypt Hidden Data

Decrypt the Data Using the File Name as a key

Create a File with the Hidden File Name and Write the Decrypted Data on it

End Operation

Figure 3.21: Flow Chart of Retraction Operation
3.7 Conclusion

New methods proposed in this research are the advanced result from previous researches on the embedding information in executable files for hiding information. In addition, it has the strong advantages that don’t limit the amount of information to be hidden and proposed system to be undetectable with anti-virus software's and making it imperceptible by any anti-virus software. One of the important conclusions in of the proposed system is the solving of the problems that are related to the size of cover file, so the hiding method makes the relation between the cover and the message independent, finally. The encryption of the message increases the degree of security of hiding technique which is used in the proposed system.