ADVANCED ENCRYPTION STANDARD (AES) IMPLEMENTATION

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Bachelor of Technology
In
Electronics and Communication Engineering

By
Aseem Jagadev
Roll No: 10509030
&
Vivek Senapati
Roll No: 10509012

Department of Electronics and Communication Engineering
National Institute of Technology, Rourkela
May, 2009

National Institute of Technology

Rourkela

CERTIFICATE

This is to certify that the thesis entitled, "*ADVANCED ENCRYPTION STANDARD (AES) IMPLEMENTATION* "submitted by "Aseem jagadev and Vivek Senapati" in partial fulfillment of the requirements for the award of Bachelor of Technology Degree in Electronics and communication Engineering at the National Institute of Technology, Rourkela (Deemed University) is an authentic work carried out by them under my supervision and guidance.

To the best of my knowledge, the matter embodied in the thesis has not been submitted to any other university / institute for the award of any Degree or Diploma.

DATE:11/5/2009

Prof. K. K. Mahapatra
Dept. of Electronics & Comm. Engineering
National Institute of Technology, Rourkela
Pin - 769008

National Institute of Technology
Rourkela

ACKNOWLEDGEMENT

We are thankful to **Dr. K. K. Mahapatra,** Professor in the department of Electronics and Communication Engineering, NIT Rourkela for giving us the opportunity to work under him and lending every support at every stage of this project work.

We would also like to convey our sincerest gratitude and ineptness' to all other faculty members and staff of Department of Electronics and Communication Engineering, NIT Rourkela, who bestowed their great effort and guidance at appropriate times without which it would have been very difficult on my part to finish the project work.

Date:

Aseem & Vivek
Dept. of Electronics & Comm. Engineering
National Institute of Technology, Rourkela
Pin - 769008

CONTENTS

1. Abstract.
2. Introduction to cryptography
3. Introduction to the Advanced Encryption Standard
4. Description of the AES algorithm
5. AES operations:
   - Sub Bytes,
   - Shift Row,
   - Mix Column
   - Add Round Key
6. The Rijndael Key Schedule
7. The Key Expansion
8. Implementation: The Key Schedule
9. Implementation: AES Encryption
10. AES Decryption.
11. Modifications
    - Arithmetic Operator,
    - Route Cipher,
    - Extended Key
12. Results
13. Conclusion.
14. References

ABSTRACT:

On October, 2, 2000, The National Institute of Standards and Technology (NIST) announced Rijndael as the new Advanced Encryption Standard (AES).The 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.

This thesis includes the complete step by step implementation of Advanced Encryption Technique, i.e. encrypting and decrypting 128 bit data using the AES and it's modification for enhanced reliability and security. The encryption process consists of the combination of various classical techniques such as substitution, rearrangement and transformation encoding techniques. The encryption and decryption modules include the Key Expansion module which generates Key for all iterations. The modifications include the addition of an arithmetic operation and

a route transposition cipher in the **attacks** iterative rounds. The Key expansion module is extended to double the number of iterative processing rounds in order to increase its immunity against unauthorized attacks**.**

Introduction to cryptography:

Cryptography is the science of information and communication security. Cryptography is the science of secret codes, enabling the confidentiality of communication through an insecure channel. It protects against unauthorized parties by preventing unauthorized alteration of use. It uses an cryptographic system to transform a plaintext into a cipher text, using most of the time a key.

There exists certain cipher that doesn't need a key at all. An example is a simple Caesar-cipher that obscures text by replacing each letter with the letter thirteen places down in the alphabet. Since our alphabet has 26 characters, it is enough to encrypt the cipher text again to retrieve the original message.

Introduction to the Advanced Encryption Standard:

The Advanced Encryption Standard, in the following referenced as AES, is the winner of the contest, held in 1997 by the US Government, after the **Data Encryption Standard** was found too weak because of its small key size and the technological advancements in processor power. Fifteen candidates were accepted in 1998 and based on public comments the pool was reduced to five finalists in 1999. In October 2000, one of these five algorithms was selected as the forthcoming standard: a slightly modified version of the Rijndael.

The Rijndael, whose name is based on the names of its two Belgian inventors, **Joan Daemen** and **Vincent Rijmen**, is a **Block cipher**, which means that it works on fixed-length group of bits, which are called *blocks*. It takes an input block of a certain size, usually 128, and produces a corresponding output block of the same size. The transformation requires a second input, which is the secret key. It is important to know that the secret key can be of any size (depending on the cipher used) and that AES uses three different key sizes: 128, 192 and 256 bits.

While AES supports only block sizes of 128 bits and key sizes of 128, 192 and 256 bits, the original Rijndael supports key and block sizes in any multiple of 32, with a minimum of 128 and a maximum of 256 bits.

Description of the Advanced Encryption Standard algorithm

AES is an iterated block cipher with a fixed block size of 128 and a variable key length. The different transformations operate on the intermediate results, called *state*. The state is a rectangular array of bytes and since the block size is 128 bits, which is 16 bytes, the rectangular array is of dimensions 4x4. (In the Rijndael version with variable block size, the row size is fixed to four and the number of columns varies. The number of columns is the block size divided by 32 and denoted **Nb**). The cipher key is similarly pictured as a rectangular array with four rows. The number of columns of the cipher key, denoted **Nk**, is equal to the key length divided by 32.

```
A state:
-----------------------------
| a0,0 | a0,1 | a0,2 | a0,3 |
| a1,0 | a1,1 | a1,2 | a1,3 |
| a2,0 | a2,1 | a2,2 | a2,3 |
| a3,0 | a3,1 | a3,2 | a3,3 |
-----------------------------

A key:
-----------------------------
| k0,0 | k0,1 | k0,2 | k0,3 |
| k1,0 | k1,1 | k1,2 | k1,3 |
| k2,0 | k2,1 | k2,2 | k2,3 |
| k3,0 | k3,1 | k3,2 | k3,3 |
-----------------------------
```

It is very *important* to know that the cipher input bytes are mapped onto the state bytes in the order a0,0, a1,0, a2,0, a3,0, a0,1, a1,1, a2,1, a3,1 ... and the bytes of the cipher key are mapped onto the array in the order k0,0, k1,0, k2,0, k3,0, k0,1, k1,1, k2,1, k3,1 ... At the end of the cipher operation, the cipher output is extracted from the state by taking the state bytes in the same order. AES uses a variable number of rounds, which are fixed: A key of size 128 has 10 rounds. A key of size 192 has 12 rounds. A key of size 256 has 14 rounds.

During each round, the following operations are applied on the state:

1. Sub Bytes: every byte in the state is replaced by another one, using the Rijndael S-Box

2. **Shift Row**: every row in the 4x4 array is shifted a certain amount to the left

3. **Mix Column**: a linear transformation on the columns of the state

4. **AddRoundKey**: each byte of the state is combined with a round key, which is a Different key for each round and derived from the Rijndael key schedule

Salient Features:

- The cipher key is expanded into a larger key, which is later used for the actual operations
- The round Key is added to the state before starting the with loop
- The **Final Round ()** is the same as **Round (),** apart from missing the **Mix Columns ()** operation.
- During each round, another part of the **Expanded Key** is used for the operations
- The Expanded Key shall always be derived from the Cipher Key and never be specified directly.

AES operations: SubBytes, ShiftRow, MixColumn and AddRoundKey

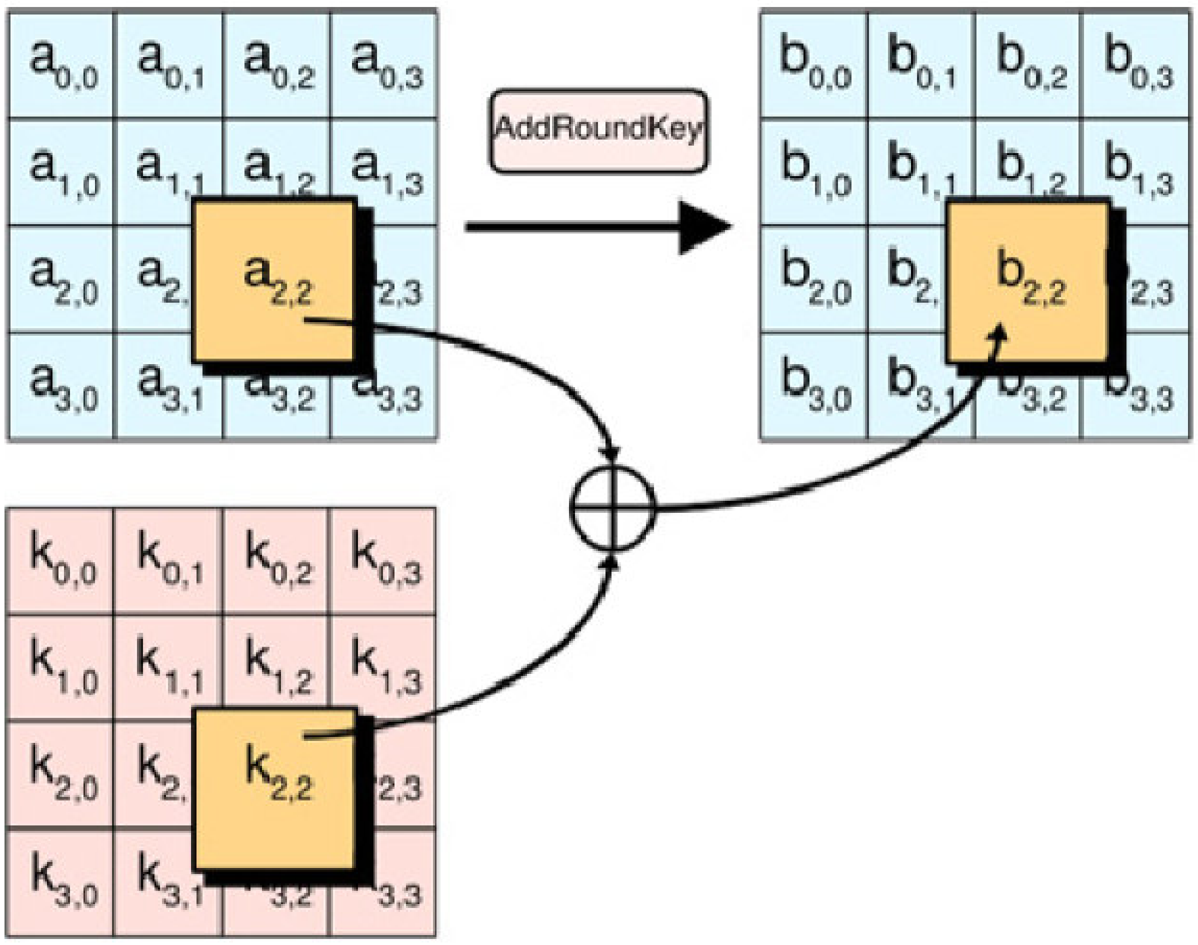
- **The AddRoundKey operation:**

In this operation, a Round Key is applied to the state by a simple bitwise XOR. The Round Key is derived from the Cipher Key by the means of the key schedule. The Round Key length is equal to the block key length (=16 bytes).

```
-----------------------------         -----------------------------    -----------------------------
| a0,0 | a0,1 | a0,2 | a0,3 |         | k0,0 | k0,1 | k0,2 | k0,3 |    | b0,0 | b0,1 | b0,2 | b0,3 |
| a1,0 | a1,1 | a1,2 | a1,3 |   XOR   | k2,0 | k2,1 | k2,2 | k2,3 |  = | b2,0 | b2,1 | b2,2 | b2,3 |
| a2,0 | a2,1 | a2,2 | a2,3 |         | k1,0 | k1,1 | k1,2 | k1,3 |    | b1,0 | b1,1 | b1,2 | b1,3 |
| a3,0 | a3,1 | a3,2 | a3,3 |         | k3,0 | k3,1 | k3,2 | k3,3 |    | b3,0 | b3,1 | b3,2 | b3,3 |
-----------------------------         -----------------------------    -----------------------------

where: b(i,j) = a(i,j) XOR k(i,j)
```

A graphical representation of this operation can be seen below:



• ***<u>The ShiftRow operation:</u>***

In this operation, each row of the state is cyclically shifted to the left, depending on the row index.

The 1st row is shifted 0 positions to the left.

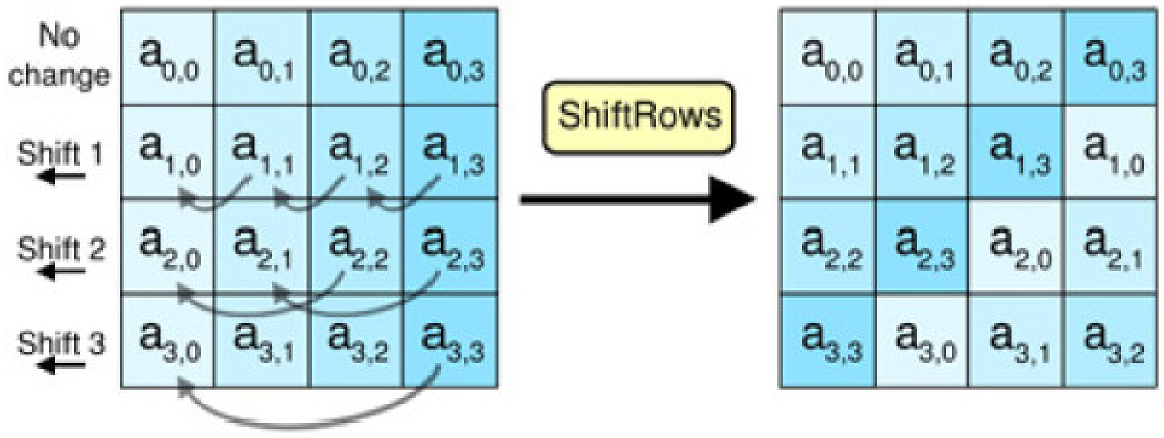
The 2nd row is shifted 1 position to the left.

The 3rd row is shifted 2 positions to the left.

The 4th row is shifted 3 positions to the left.

```
-----------------------------     -----------------------------
| a0,0 | a0,1 | a0,2 | a0,3 |     | a0,0 | a0,1 | a0,2 | a0,3 |
| a1,0 | a1,1 | a1,2 | a1,3 | ->  | a1,1 | a0,2 | a1,3 | a1,0 |
| a2,0 | a2,1 | a2,2 | a2,3 |     | a2,2 | a2,3 | a2,0 | a2,1 |
| a3,0 | a3,1 | a3,2 | a3,3 |     | a3,3 | a3,0 | a3,1 | a3,2 |
-----------------------------     -----------------------------
```

<u>A graphical representation of this operation can be found below</u>:


No change
Shift 1
Shift 2
Shift 3
$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$
ShiftRows
$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,1}$ $a_{1,2}$ $a_{1,3}$ $a_{1,0}$
$a_{2,2}$ $a_{2,3}$ $a_{2,0}$ $a_{2,1}$
$a_{3,3}$ $a_{3,0}$ $a_{3,1}$ $a_{3,2}$

The inverse of **Shift Row** is the same cyclically shift but to the right. It will be needed later for decoding.

• ***The SubBytes operation***:

The **SubBytes** operation is a non-linear byte substitution, operating on each byte of the state independently. The **substitution table (S-Box)** is invertible and is constructed by the composition of two transformations:

1. Take the multiplicative inverse in **Rijndael's finite field**
2. Apply an affine transformation which is documented in the Rijndael documentation.

Since the S-Box is independent of any input, pre-calculated forms are used. Each byte of the state is then substituted by the value in the S-Box whose index corresponds to the value in the state:

```
a(i,j) = SBox[a(i,j)]
```

The inverse of SubBytes is the same operation, using the inversed S-Box, which is also precalculated.

• ***<u>The MixColumn operation:</u>***

This section involves advance mathematical calculations in the **Rijndael's finite field**. It corresponds to the matrix multiplication with:

$$
\begin{matrix}
2 & 3 & 1 & 1 \\
1 & 2 & 3 & 1 \\
1 & 1 & 2 & 3 \\
3 & 1 & 1 & 2
\end{matrix}
$$

And that the addition and multiplication operations are different from the normal ones.

• ***<u>The Rijndael Key Schedule</u>***

- The Key Schedule is responsible for expanding a short key into a larger key, whose parts are used during the different iterations. Each key size is expanded to a different size:

An 128 bit key is expanded to an 176 byte key.

An 192 bit key is expanded to an 208 byte key.

An 256 bit key is expanded to an 240 byte key.

There is a relation between the cipher key size, the number of rounds and the Expanded Key size. For an 128-bit key, there is one initial AddRoundKey operation plus there are 10 rounds and each round needs a new 16 byte key, therefore we require 10+1 Round Keys of 16 byte, which equals 176 byte. The same logic can be applied to the two other cipher key sizes. The general formula is that:

```
ExpandedKeySize = (nbrRounds+1) * BlockSize
```

• ***Rotate:***

The 4-byte word is cyclically shifted 1 byte to the left:

```
---------------------     ---------------------
| 1d | 2c | 3a | 4f | -> | 2c | 3a | 4f | 1d |
---------------------     ---------------------
```

• ***Rcon:***

Just note that the Rcon values can be pre-calculated, which results in a simple substitution (a table lookup) in a fixed Rcon table.

• ***S-Box:***

The Key Schedule uses the same S-Box substitution as the main algorithm body.

• ***The Key Schedule Core:***

In the below code, word has a size of 4 bytes and i is the iteration counter from the Key Schedule

```
KeyScheduleCore (word)
{
Rotate(word);
SBoxSubstitution (word);
word[0] = word[0] XOR RCON[i];
}
```

- ***The Key Expansion:***

```
KeyExpansion(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;
}
}
```

• ***Implementation: The Key Schedule***

We will start the implementation of AES with the Cipher Key expansion. We intend to enlarge our input cipher key, whose size varies between 128 and 256 bits into a larger key, from which different RoundKeys can be derived.

• ***Implementation: S-Box***

The S-Box values can either be calculated on-the-fly to save memory or the pre-calculated values can be stored in an array. We will store the values in an array.

Here's the code for the 2 S-Boxes, it's only a table-lookup that returns the value in the array whose index is specified as a parameter of the function:

```
unsigned char sbox[256] =
{
//0 1 2 3 4 5 6 7 8 9 A B C D E F
0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76, //0
0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0, //1
0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15, //2
0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75, //3
0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84, //4
0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf, //5
0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8, //6
```

```
0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2, //7
0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73, //8
0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb, //9
0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79, //A
0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08, //B
0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a, //C
0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e, //D
0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf, //E
0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16 //F
};

unsigned char rsbox[256] =
{
 0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3, 0x9e, 0x81, 0xf3, 0xd7, 0xfb
, 0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87, 0x34, 0x8e, 0x43, 0x44, 0xc4, 0xde, 0xe9, 0xcb
, 0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d, 0xee, 0x4c, 0x95, 0x0b, 0x42, 0xfa, 0xc3, 0x4e
, 0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2, 0x76, 0x5b, 0xa2, 0x49, 0x6d, 0x8b, 0xd1, 0x25
, 0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0x65, 0xb6, 0x92
, 0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda, 0x5e, 0x15, 0x46, 0x57, 0xa7, 0x8d, 0x9d, 0x84
, 0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a, 0xf7, 0xe4, 0x58, 0x05, 0xb8, 0xb3, 0x45, 0x06
, 0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02, 0xc1, 0xaf, 0xbd, 0x03, 0x01, 0x13, 0x8a, 0x6b
, 0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4, 0xe6, 0x73
, 0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9, 0x37, 0xe8, 0x1c, 0x75, 0xdf, 0x6e
, 0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89, 0x6f, 0xb7, 0x62, 0x0e, 0xaa, 0x18, 0xbe, 0x1b
, 0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20, 0x9a, 0xdb, 0xc0, 0xfe, 0x78, 0xcd, 0x5a, 0xf4
, 0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xec, 0x5f
, 0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d, 0xe5, 0x7a, 0x9f, 0x93, 0xc9, 0x9c, 0xef
, 0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0, 0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61
, 0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26, 0xe1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0c, 0x7d
};
```

```
unsigned char getSBoxValue(unsigned char num)
{
return sbox[num];
}
unsigned char getSBoxInvert(unsigned char num)
{
return rsbox[num];
}
```

• ***Implementation: Rotate***

From the theoretical part, It is known already that Rotate takes a word (a 4-byte array) and rotates it 8 bit to the left. Since 8 bit correspond to one byte and our array type is character (whose size is one byte), rotating 8 bit to the left corresponds to shifting cyclically the array values one to the left.

<u>Here's the code for the Rotate function</u>::

```
void rotate(unsigned char *word)

{

unsigned char c;

int i;

c = word[0];

for (i = 0; i < 3; i++)

word[i] = word[i+1];

word[3] = c;

}
```

- ***<u>Implementation: Rcon</u>***

```
unsigned char Rcon[255] =

{

0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8,

0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3,

0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f,

0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d,

0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab,

0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d,
0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25,
0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01,
0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d,
0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa,
0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a,
0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02,
0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a,
0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef,
0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94,
0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04,
0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f,
0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5,
0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33,
0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb
};



unsigned char getRconValue(unsigned char num)
{
return Rcon[num];
}
```

• ***Implementation: Key Schedule Core***

Code applies the operations one after the other on the 4-byte word. The parameters are the 4-byte word and the iteration counter, on which **Rcon** depends.

```
void core(unsigned char *word, int iteration)
{
int i;
/* rotate the 32-bit word 8 bits to the left */
rotate(word);
/* apply S-Box substitution on all 4 parts of the 32-bit word */
for (i = 0; i < 4; ++i)
{
word[i] = getSBoxValue(word[i]);
}
/* XOR the output of the rcon operation with i to the first part (leftmost) only */
word[0] = word[0]^getRconValue(iteration);
}
```

• ***Implementation: Key Expansion***

```
enum keySize

{

SIZE_16 = 16,

SIZE_24 = 24,

SIZE_32 = 32

};
```

Our key expansion function basically needs only two things:

- the input cipher key
- the output expanded key

Since in C, it is not possible to know the size of an array passed as pointer to a function, the cipher key size (of type "enum key Size") is added and the expanded key size (of type size_t) to the parameter list of our function. The prototype looks like the following:

void expandKey(unsigned char *expandedKey, unsigned char *key, enum keySize, size_t expandedKeySize);

The keyexpansion function is shown below:

```
void expandKey(unsigned char *expandedKey,

unsigned char *key,

enum keySize size,

size_t expandedKeySize)

{

/* current expanded keySize, in bytes */

int currentSize = 0;

int rconIteration = 1;

int i;

unsigned char t[4] = {0}; // temporary 4-byte variable

/* set the 16,24,32 bytes of the expanded key to the input key */

for (i = 0; i < size; i++)

expandedKey[i] = key[i];

currentSize += size;

while (currentSize < expandedKeySize)

{

/* assign the previous 4 bytes to the temporary value t */

for (i = 0; i < 4; i++)
```

```
{
t[i] = expandedKey[(currentSize - 4) + i];
}
/* every 16,24,32 bytes we apply the core schedule to t
* and increment rconIteration afterwards
*/
if(currentSize % size == 0)
{
core(t, rconIteration++);
}
/* For 256-bit keys, we add an extra sbox to the calculation */
if(size == SIZE_32 && ((currentSize % size) == 16)) {
for(i = 0; i < 4; i++)
t[i] = getSBoxValue(t[i]);
}
/* We XOR t with the four-byte block 16,24,32 bytes before the new expanded
key.
* This becomes the next four bytes in the expanded key.
*/
for(i = 0; i < 4; i++) {
```

```
expandedKey[currentSize] = expandedKey[currentSize - size] ^ t[i];
currentSize++;
}
}
}
```

As it can be seen,no inner loops have been used to repeat an operation, the only inner loops are to iterate over the 4 parts of the temporary array t. The modulo operator have been used to check to apply the operation:

- *if(currentSize % size == 0)*: whenever we have have created n bytes of expandedKey (where n is the cipherkey size), we run the key expansion core once
- *if(size == SIZE_32 && ((currentSize % size) == 16))*: if we are expanding an 32-bit cipherkey and if we have already generated 16 bytes (as I explained above, in the 32-bit version we run the first loop only 3 times, which generates 12 bytes + the 4 bytes from the core), we add one additional S-Box substitution

• *Implementation: Using the Key Expansion*

Here are several test results:

The Key Expansion of an 128-bit key consisting of null characters (like the example above):

```
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

62 63 63 63 62 63 63 63 62 63 63 63 62 63 63 63

9b 98 98 c9 f9 fb fb aa 9b 98 98 c9 f9 fb fb aa

90 97 34 50 69 6c cf fa f2 f4 57 33 0b 0f ac 99

ee 06 da 7b 87 6a 15 81 75 9e 42 b2 7e 91 ee 2b

7f 2e 2b 88 f8 44 3e 09 8d da 7c bb f3 4b 92 90

ec 61 4b 85 14 25 75 8c 99 ff 09 37 6a b4 9b a7

21 75 17 87 35 50 62 0b ac af 6b 3c c6 1b f0 9b

0e f9 03 33 3b a9 61 38 97 06 0a 04 51 1d fa 9f

b1 d4 d8 e2 8a 7d b9 da 1d 7b b3 de 4c 66 49 41

b4 ef 5b cb 3e 92 e2 11 23 e9 51 cf 6f 8f 18 8e
```

<u>The Key Expansion of an 192-bit key consisting of null characters</u>:

```
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00 62 63 63 63 62 63 63 63

62 63 63 63 62 63 63 63 62 63 63 63 62 63 63 63

9b 98 98 c9 f9 fb fb aa 9b 98 98 c9 f9 fb fb aa

9b 98 98 c9 f9 fb fb aa 90 97 34 50 69 6c cf fa

f2 f4 57 33 0b 0f ac 99 90 97 34 50 69 6c cf fa

c8 1d 19 a9 a1 71 d6 53 53 85 81 60 58 8a 2d f9

c8 1d 19 a9 a1 71 d6 53 7b eb f4 9b da 9a 22 c8

89 1f a3 a8 d1 95 8e 51 19 88 97 f8 b8 f9 41 ab

c2 68 96 f7 18 f2 b4 3f 91 ed 17 97 40 78 99 c6

59 f0 0e 3e e1 09 4f 95 83 ec bc 0f 9b 1e 08 30

0a f3 1f a7 4a 8b 86 61 13 7b 88 5f f2 72 c7 ca

43 2a c8 86 d8 34 c0 b6 d2 c7 df 11 98 4c 59 70
```

The Key Expansion of an 256-bit key consisting of null characters:

```
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

62 63 63 63 62 63 63 63 62 63 63 63 62 63 63 63

aa fb fb fb aa fb fb fb aa fb fb fb aa fb fb fb

6f 6c 6c cf 0d 0f 0f ac 6f 6c 6c cf 0d 0f 0f ac

7d 8d 8d 6a d7 76 76 91 7d 8d 8d 6a d7 76 76 91

53 54 ed c1 5e 5b e2 6d 31 37 8e a2 3c 38 81 0e

96 8a 81 c1 41 fc f7 50 3c 71 7a 3a eb 07 0c ab

9e aa 8f 28 c0 f1 6d 45 f1 c6 e3 e7 cd fe 62 e9

2b 31 2b df 6a cd dc 8f 56 bc a6 b5 bd bb aa 1e

64 06 fd 52 a4 f7 90 17 55 31 73 f0 98 cf 11 19

6d bb a9 0b 07 76 75 84 51 ca d3 31 ec 71 79 2f

e7 b0 e8 9c 43 47 78 8b 16 76 0b 7b 8e b9 1a 62

74 ed 0b a1 73 9b 7e 25 22 51 ad 14 ce 20 d4 3b

10 f8 0a 17 53 bf 72 9c 45 c9 79 e7 cb 70 63 85
```

Implementation: AES Encryption

To implement the AES encryption algorithm, we proceed exactly the same way as for the key expansion, that is, we first implement the basic helper functions and then move up to the main loop. The functions take as parameter a *state*, which is, as already explained, a rectangular 4x4 array of bytes. We won't consider the state as a 2-dimensional array, but as a 1-dimensional array of length 16.

- **Implementation: subBytes**

It's a simple substitution with the S-Box value:

```
void subBytes(unsigned char *state)
{
int i;
/* substitute all the values from the state with the value in the SBox
* using the state value as index for the SBox
*/
for (i = 0; i < 16; i++)
state[i] = getSBoxValue(state[i]);
}
```

- **Implementation: shiftRows**

This function was split into two parts. The **shiftRows** function iterates over all the rows and then call **shiftRow** with the correct offset. **ShiftRow** shifts a 4-byte array by the given offset.

```
void shiftRows(unsigned char *state)
{
int i;
/* iterate over the 4 rows and call shiftRow() with that row */
for (i = 0; i < 4; i++)
shiftRow(state+i*4, i);
}

void shiftRow(unsigned char *state, unsigned char nbr)
{
int i, j;
unsigned char tmp;

/* each iteration shifts the row to the left by 1 */

for (i = 0; i < nbr; i++)

{

tmp = state[0];

for (j = 0; j < 3; j++)

state[j] = state[j+1];

state[3] = tmp;

}

}
```

- **<u>Implementation: addRoundKey</u>**

This is the part that involves the roundKey we generate during each iteration. We simply XOR each byte of the key to the respective byte of the state.

```
void addRoundKey(unsigned char *state, unsigned char *roundKey)

{

int i;

for (i = 0; i < 16; i++)
state[i] = state[i] ^ roundKey[i] ;

}
```

- **Implementation: mixColumns**

MixColumns involved the **galois** addition and multiplication and processes columns instead of rows. First of all, a function was needed that multiplies two number in the galois field.

```
unsigned char galois_multiplication(unsigned char a, unsigned char b)

{

unsigned char p = 0;

unsigned char counter;

unsigned char hi_bit_set;

for(counter = 0; counter < 8; counter++) {

if((b & 1) == 1)

p ^= a;

hi_bit_set = (a & 0x80);

a <<= 1;

if(hi_bit_set == 0x80)
a ^= 0x1b;

b >>= 1;

}

return p;

}
```

Spliting the function in 2 parts, the first one would generate a column

And then call mixColumn, which would then apply the matrix multiplication.

```
void mixColumns(unsigned char *state)

{

int i, j;

unsigned char column[4];

/* iterate over the 4 columns */

for (i = 0; i < 4; i++)

{

/* construct one column by iterating over the 4 rows */

for (j = 0; j < 4; j++)

{

column[j] = state[(j*4)+i];
}
/* apply the mixColumn on one column */
mixColumn(column);
/* put the values back into the state */
for (j = 0; j < 4; j++)
{
state[(j*4)+i] = column[j];
}
}
}
```

The mixColumn is simply a Galois multiplication of the column with the 4x4 Matrix provided in the theory. Since an addition corresponds to a XOR operation and we already have the multiplication function, the implementation is:

```
void mixColumn(unsigned char *column)
{
unsigned char cpy[4];
int i;
for(i = 0; i < 4; i++)
{
```

```
cpy[i] = column[i];

}

column[0] = galois_multiplication(cpy[0],2) ^

galois_multiplication(cpy[3],1) ^

galois_multiplication(cpy[2],1) ^

galois_multiplication(cpy[1],3);

column[1] = galois_multiplication(cpy[1],2) ^

galois_multiplication(cpy[0],1) ^

galois_multiplication(cpy[3],1) ^

galois_multiplication(cpy[2],3);

column[2] = galois_multiplication(cpy[2],2) ^

galois_multiplication(cpy[1],1) ^

galois_multiplication(cpy[0],1) ^

galois_multiplication(cpy[3],3);

column[3] = galois_multiplication(cpy[3],2) ^

galois_multiplication(cpy[2],1) ^

galois_multiplication(cpy[1],1) ^

galois_multiplication(cpy[0],3);

}
```

- **<u>Implementation: AES round</u>**

One AES round applies all four operations on the state consecutively.

```
void aes_round(unsigned char *state, unsigned char *roundKey)
{
subBytes(state);
shiftRows(state);
mixColumns(state);
addRoundKey(state, roundKey);
}
```

- **<u>Implementation: the main AES body</u>**

Now that we have all the small functions, we have taken the state, the expandedKey and the number of rounds as parameters and then called the operations one after the other. A little function called **createRoundKey()** was used to copy the next 16 bytes from the expandedKey into the **roundKey**, using the special mapping order.

```
Void   createRoundKey(unsigned char *expandedKey, unsigned char *roundKey)

{

int i,j;

/* iterate over the columns */

for (i = 0; i < 4; i++)

{

/* iterate over the rows */

for (j = 0; j < 4; j++)

roundKey[(i+(j*4))] = expandedKey[(i*4)+j];

}

}


void  aes_main(unsigned char *state, unsigned char *expandedKey, int nbrRounds)

{

int i = 0;

unsigned char roundKey[16];

createRoundKey(expandedKey, roundKey);

addRoundKey(state, roundKey);

for (i = 1; i < nbrRounds; i++) {
```

```
createRoundKey(expandedKey + 16*i, roundKey);

aes_round(state, roundKey);

}

createRoundKey(expandedKey + 16*nbrRounds, roundKey);

subBytes(state);

shiftRows(state);

addRoundKey(state, roundKey);

}
```

- **Implementation: AES encryption**

Our parameters are the input plaintext, the key of size key Size and the output. First, we have calculated the number of rounds based on they key Size and then the expandedKeySize based on the number of rounds. Then we have to map the 16 byte input plaintext in the correct order to the 4x4 byte state, expand the key using our key schedule, encrypt the state using our main AES body and finally unmap the state again in the correct order to get the 16 byte output ciphertext.

```
char aes_encrypt(unsigned char *input,
unsigned char *output,
unsigned char *key,
enum keySize size)
{
/* the expanded keySize */
int expandedKeySize;
/* the number of rounds */
int nbrRounds;
/* the expanded key */
unsigned char *expandedKey;
/* the 128 bit block to encode */
unsigned char block[16];
int i,j;
/* set the number of rounds */
switch (size)
{
case SIZE_16:
nbrRounds = 10;
break;
case SIZE_24:
nbrRounds = 12;
break;
case SIZE_32:
nbrRounds = 14;
break;
default:
return UNKNOWN_KEYSIZE;
break;
}
```

• AES Decryption

Basically, we have inversed the whole encryption and applied all the operations backwards. As the key schedule stays the same, the only operations we need to implement are the inversed **subBytes, shiftRows and mixColumns**, while **addRoundKey** stays the same.

```
void invSubBytes(unsigned char *state)

{

int i;

/* substitute all the values from the state with the value in the SBox

* using the state value as index for the SBox

*/

for (i = 0; i < 16; i++)

state[i] = getSBoxInvert(state[i]);

}


void invShiftRows(unsigned char *state)

{

int i;

/* iterate over the 4 rows and call invShiftRow() with that row */

for (i = 0; i < 4; i++)


invShiftRow(state+i*4, i);

}

void invShiftRow(unsigned char *state, unsigned char nbr)

{
```

```
int i, j;
unsigned char tmp;
/* each iteration shifts the row to the right by 1 */
for (i = 0; i < nbr; i++)
{
tmp = state[3];
for (j = 3; j > 0; j--)
state[j] = state[j-1];
state[0] = tmp;
}
}
```

Rotation this time was done to the right and that we used the inversed S-Box for the substitution. As for the inversed mixColumns operation, the only difference was the multiplication matrix, which is the following:

14 11 13 9

9 14 11 13

13 9 14 11

11 13 9 14

```
void invMixColumns(unsigned char *state)
{
int i, j;
unsigned char column[4];
/* iterate over the 4 columns */
for (i = 0; i < 4; i++)
{
/* construct one column by iterating over the 4 rows */
for (j = 0; j < 4; j++)
{
column[j] = state[(j*4)+i];
}
/* apply the invMixColumn on one column */
invMixColumn(column);
/* put the values back into the state */
for (j = 0; j < 4; j++)
{
state[(j*4)+i] = column[j];
}
}
```

```
}

void invMixColumn(unsigned char *column)

{

unsigned char cpy[4];

int i;

for(i = 0; i < 4; i++)

{

cpy[i] = column[i];

}

column[0] = galois_multiplication(cpy[0],14) ^

galois_multiplication(cpy[3],9) ^

galois_multiplication(cpy[2],13) ^

galois_multiplication(cpy[1],11);

column[1] = galois_multiplication(cpy[1],14) ^

galois_multiplication(cpy[0],9) ^

galois_multiplication(cpy[3],13) ^

galois_multiplication(cpy[2],11);

column[2] = galois_multiplication(cpy[2],14) ^

galois_multiplication(cpy[1],9) ^

galois_multiplication(cpy[0],13) ^
galois_multiplication(cpy[3],11);

column[3] = galois_multiplication(cpy[3],14) ^

galois_multiplication(cpy[2],9) ^

galois_multiplication(cpy[1],13) ^

galois_multiplication(cpy[0],11);

}
```

One inversed AES round becomes:

```
void aes_invRound(unsigned char *state, unsigned char *roundKey)

{

invShiftRows(state);

invSubBytes(state);

addRoundKey(state, roundKey);

invMixColumns(state);

}
```

We have used our expanded key backwards, starting with the last 16 bytes and then moving towards the start:

```
void aes_invMain(unsigned char *state, unsigned char *expandedKey, int
nbrRounds)
{
int i = 0;
unsigned char roundKey[16];
createRoundKey(expandedKey + 16*nbrRounds, roundKey);
addRoundKey(state, roundKey);
for (i = nbrRounds-1; i > 0; i--) {
createRoundKey(expandedKey + 16*i, roundKey);
aes_invRound(state, roundKey);
}
createRoundKey(expandedKey, roundKey);
invShiftRows(state);
invSubBytes(state);
addRoundKey(state, roundKey);
}
```

This was the end of the Advanced Encryption Standard Implementation, ready to encrypt/decrypt messages of any size.

Modifications

In order to enhance the security and reliability of AES, we bring in three changes. In each iterative round, apart from the usual four above mentioned operations, we also include two new operations: The Arithmetic Operator and The Route Cipher. We also modify the key schedule so as to increase the number of the AES encryption rounds. For example, for 16 byte key, we generate 336 bit key instead of the usual 176 bit key. By this process, we are able to successfully process 20+1 rounds instead of the previous 10+1 rounds for the 16 byte key.

Lets have a look at the modifications and there implications.

Arithmetic Operation

In this operation, each element of the state is arithmetically added by a number depending on their row number.

The 1st row is added to 1.

The 2nd row is added to 2.

The 3rd row is added to 3.

The 4th row is added to 4.

To retain the symmetric nature of AES, during decryption we have inversed the process by subtracting the corresponding same numbers.

The 1st row is added to 1.

The 2nd row is added to 2.

The 3rd row is added to 3.

The 4th row is added to 4.

Code

```
void adds(unsigned char *state)
{
int i;
for (i = 0; i < 4; i++)
add(state+i*4, i);
}
void add(unsigned char *state, unsigned char nbr)
{
int i, j;



for (i = 0; i < nbr; i++)
{
for (j = 0; j <4; j++)
state[j] = state[j]+1;


}
```

Route cipher

In a route cipher, the plaintext was first written out in a grid of given dimensions, and then read off in a pattern given in the key. For example, using the same plaintext:

```
W R I O R F E O E
E E S V E L A N J
A D C E D E T C X
```

That would give a cipher text of:

EJXCTEDECDAEWRIORFEONALEVSE

Code

```
void hillcipher(unsigned char *state)
{
int a;
```

```
unsigned char temp2[16];

for(a=0;a<16;a++)

temp2[a] =state[a];

state[9]=temp2[0];

state[10]=temp2[1];

state[11]=temp2[2];

state[0]=temp2[3];

state[8]=temp2[4];

state[15]=temp2[5];

state[12]=temp2[6];

state[1]=temp2[7];

state[7]=temp2[8];

state[14]=temp2[9];

state[13]=temp2[10];

state[2]=temp2[11];

state[6]=temp2[12];

state[5]=temp2[13];

state[4]=temp2[14];

state[3]=temp2[15];
```

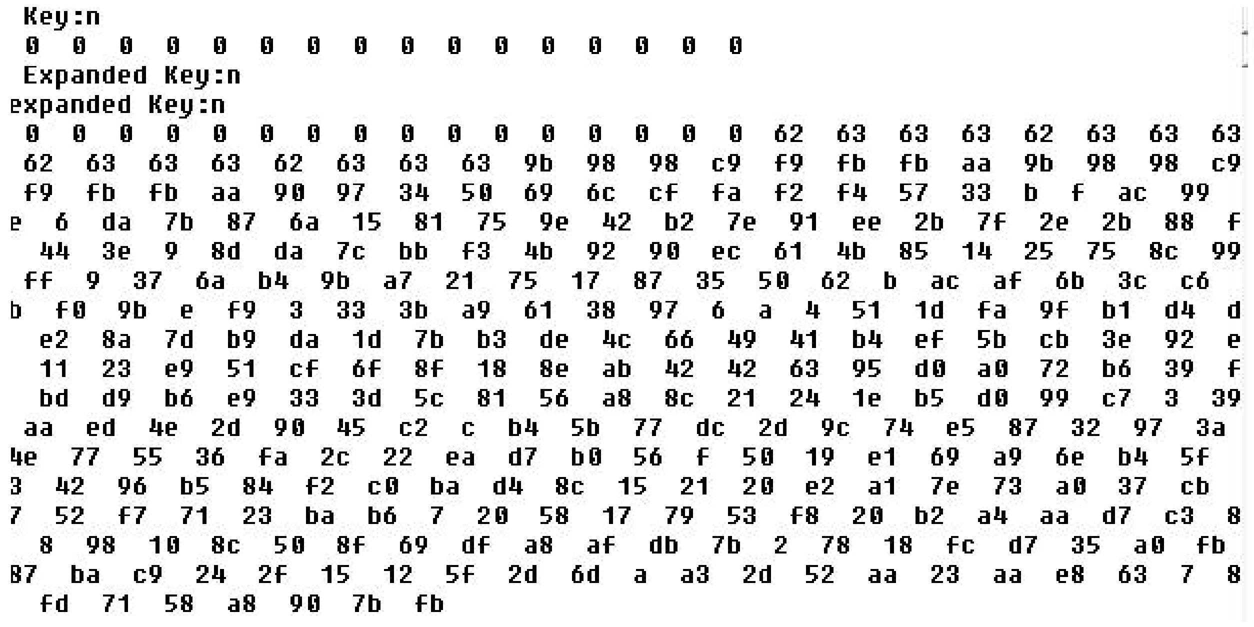
}

During Decryption we have to just reverse the process, replacing the elements in their original positions.

Now, we include both these new functions in each of our iterative rounds.

Extending the Key Schedule

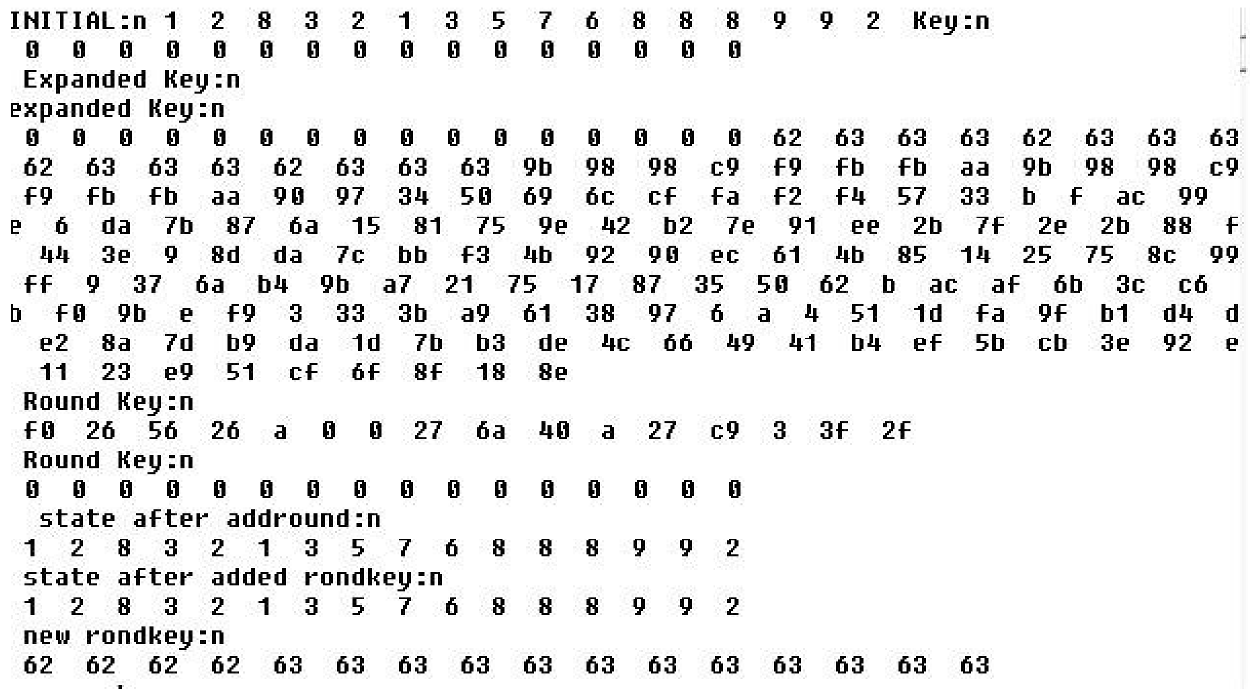
We have also extended the key schedule. We have followed the key schedule process but we haven't stopped at the earlier specifications, rather we continued doing so in order to enable more computing iterative rounds, giving the attacker an even tougher code to break. For example, for 16 byte null key, we generate the following 336 bit extended key which facilitates the proper operation of 20+1 rounds, i.e. double the number of rounds earlier.



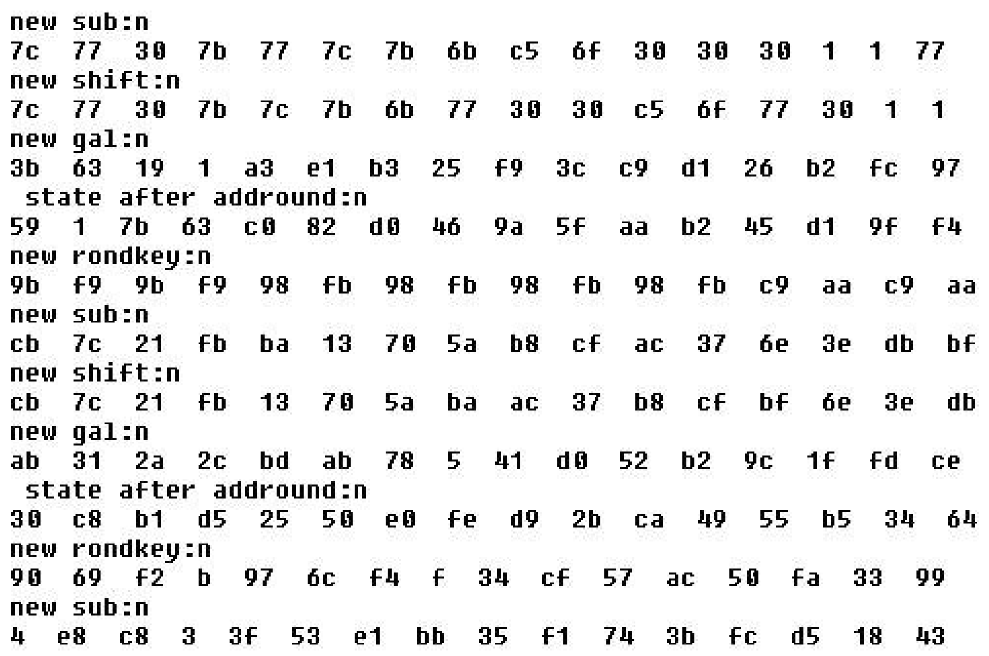
```
 Key:n
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 Expanded Key:n
expanded Key:n
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  62  63  63  63  62  63  63  63
 62  63  63  63  62  63  63  63  9b  98  98  c9  f9  fb  fb  aa  9b  98  98  c9
 f9  fb  fb  aa  90  97  34  50  69  6c  cf  fa  f2  f4  57  33  b  f  ac  99
e  6  da  7b  87  6a  15  81  75  9e  42  b2  7e  91  ee  2b  7f  2e  2b  88  f
  44  3e  9  8d  da  7c  bb  f3  4b  92  90  ec  61  4b  85  14  25  75  8c  99
 ff  9  37  6a  b4  9b  a7  21  75  17  87  35  50  62  b  ac  af  6b  3c  c6
b  f0  9b  e  f9  3  33  3b  a9  61  38  97  6  a  4  51  1d  fa  9f  b1  d4  d
  e2  8a  7d  b9  da  1d  7b  b3  de  4c  66  49  41  b4  ef  5b  cb  3e  92  e
  11  23  e9  51  cf  6f  8f  18  8e  ab  42  42  63  95  d0  a0  72  b6  39  f
  bd  d9  b6  e9  33  3d  5c  81  56  a8  8c  21  24  1e  b5  d0  99  c7  3  39
 aa  ed  4e  2d  90  45  c2  c  b4  5b  77  dc  2d  9c  74  e5  87  32  97  3a
4e  77  55  36  fa  2c  22  ea  d7  b0  56  f  50  19  e1  69  a9  6e  b4  5f
3  42  96  b5  84  f2  c0  ba  d4  8c  15  21  20  e2  a1  7e  73  a0  37  cb
7  52  f7  71  23  ba  b6  7  20  58  17  79  53  f8  20  b2  a4  aa  d7  c3  8
  8  98  10  8c  50  8f  69  df  a8  af  db  7b  2  78  18  fc  d7  35  a0  fb
87  ba  c9  24  2f  15  12  5f  2d  6d  a  a3  2d  52  aa  23  aa  e8  63  7  8
  fd  71  58  a8  90  7b  fb
```

<u>RESULTS</u>

The prepared AES code was tested using various keys and key sizes. One of the results is given below. The data message was state[]= {1,2,8,3,2,1,3,5,7,6,8,8,8,9,9,2}. The key input was the 16 byte null vector. The encrypted message was found to be : 30 5f ad fb 4f bf d6 34 d4 af 5b d9 4e 34 9e 3. And the decryption gave back the original data. The step by step implementation is shown below.



```
INITIAL:n 1  2  8  3  2  1  3  5  7  6  8  8  8  9  9  2  Key:n
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 Expanded Key:n
expanded Key:n
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  62  63  63  63  62  63  63  63
 62  63  63  63  62  63  63  63  9b  98  98  c9  f9  fb  fb  aa  9b  98  98  c9
 f9  fb  fb  aa  90  97  34  50  69  6c  cf  fa  f2  f4  57  33  b  f  ac  99
e  6  da  7b  87  6a  15  81  75  9e  42  b2  7e  91  ee  2b  7f  2e  2b  88  f
  44  3e  9  8d  da  7c  bb  f3  4b  92  90  ec  61  4b  85  14  25  75  8c  99
 ff  9  37  6a  b4  9b  a7  21  75  17  87  35  50  62  b  ac  af  6b  3c  c6
b  f0  9b  e  f9  3  33  3b  a9  61  38  97  6  a  4  51  1d  fa  9f  b1  d4  d
  e2  8a  7d  b9  da  1d  7b  b3  de  4c  66  49  41  b4  ef  5b  cb  3e  92  e
  11  23  e9  51  cf  6f  8f  18  8e
 Round Key:n
 f0  26  56  26  a  0  0  27  6a  40  a  27  c9  3  3f  2f
 Round Key:n
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  state after addround:n
 1  2  8  3  2  1  3  5  7  6  8  8  8  9  9  2
 state after added rondkey:n
 1  2  8  3  2  1  3  5  7  6  8  8  8  9  9  2
 new rondkey:n
 62  62  62  62  63  63  63  63  63  63  63  63  63  63  63  63
```



```
new sub:n
7c  77  30  7b  77  7c  7b  6b  c5  6f  30  30  30  1  1  77
new shift:n
7c  77  30  7b  7c  7b  6b  77  30  30  c5  6f  77  30  1  1
new gal:n
3b  63  19  1  a3  e1  b3  25  f9  3c  c9  d1  26  b2  fc  97
 state after addround:n
59  1  7b  63  c0  82  d0  46  9a  5f  aa  b2  45  d1  9f  f4
new rondkey:n
9b  f9  9b  f9  98  fb  98  fb  98  fb  98  fb  c9  aa  c9  aa
new sub:n
cb  7c  21  fb  ba  13  70  5a  b8  cf  ac  37  6e  3e  db  bf
new shift:n
cb  7c  21  fb  13  70  5a  ba  ac  37  b8  cf  bf  6e  3e  db
new gal:n
ab  31  2a  2c  bd  ab  78  5  41  d0  52  b2  9c  1f  fd  ce
 state after addround:n
30  c8  b1  d5  25  50  e0  fe  d9  2b  ca  49  55  b5  34  64
new rondkey:n
90  69  f2  b  97  6c  f4  f  34  cf  57  ac  50  fa  33  99
new sub:n
4  e8  c8  3  3f  53  e1  bb  35  f1  74  3b  fc  d5  18  43
```

```
new shift:n
4  e8  c8  3  53  e1  bb  3f  74  3b  35  f1  43  fc  d5  18
new gal:n
ca  34  bd  ae  7d  80  2f  6d  7a  60  7d  ed  ad  1a  7c  fb
 state after addround:n
5a  5d  4f  a5  ea  ec  db  62  4e  af  2a  41  fd  e0  4f  62
new rondkey:n
ee  87  75  7e  6  6a  9e  91  da  15  42  ee  7b  81  b2  2b
new sub:n
be  4c  84  6  87  ce  b9  aa  2f  79  e5  83  54  e1  84  aa
new shift:n
be  4c  84  6  ce  b9  aa  87  e5  83  2f  79  aa  54  e1  84
new gal:n
61  9f  38  63  a7  ef  5b  1c  44  14  48  e4  bd  46  cb  e7
 state after addround:n
8f  18  4d  1d  a1  85  c5  8d  9e  1  a  a  c6  c7  79  cc
new rondkey:n
7f  f8  8d  f3  2e  44  da  4b  2b  3e  7c  92  88  9  bb  90
new sub:n
73  ad  e3  a4  32  97  a6  5d  b  7c  67  67  b4  c6  b6  4b
new shift:n
73  ad  e3  a4  97  a6  5d  32  67  67  b  7c  4b  b4  c6  b6

new gal:n
68  63  f7  cf  a4  e7  82  f2  f7  2  f9  af  f3  5e  ff  ce
 state after addround:n
17  9b  7a  3c  8a  a3  58  b9  dc  3c  85  3d  7b  57  44  5e
new rondkey:n
ec  14  99  6a  61  25  ff  b4  4b  75  9  9b  85  8c  37  a7
new sub:n
f0  14  da  eb  7e  a  6a  56  86  eb  97  27  21  5b  1b  58
new shift:n
f0  14  da  eb  a  6a  56  7e  97  27  86  eb  58  21  5b  1b
new gal:n
2a  90  88  bf  1e  88  bc  2a  27  53  76  75  26  33  13  85
 state after addround:n
c6  84  11  d5  7f  ad  43  9e  6c  26  7f  ee  a3  bf  24  22
new rondkey:n
21  35  ac  c6  75  50  af  1b  17  62  6b  f0  87  b  3c  9b
new sub:n
b4  5f  82  3  d2  95  1a  b  50  f7  d2  28  a  8  36  93
new shift:n
b4  5f  82  3  95  1a  b  d2  d2  28  50  f7  93  a  8  36
new gal:n
96  b2  5a  aa  7b  19  6c  88  30  b  31  7e  bd  c7  d6  4c
```

```
 state after addround:n
b7  87  f6  6c  e  49  c3  93  27  69  5a  8e  3a  cc  ea  d7
new rondkey:n
e  3b  97  51  f9  a9  6  1d  3  61  a  fa  33  38  4  9f
new sub:n
a9  17  42  50  ab  3b  2e  dc  cc  f9  be  19  80  4b  87  e
new shift:n
a9  17  42  50  3b  2e  dc  ab  be  19  cc  f9  e  80  4b  87
new gal:n
b4  c5  7c  38  8  e0  e5  8a  e7  90  c0  80  79  15  40  b7
 state after addround:n
ba  fe  eb  69  f1  49  e3  97  e4  f1  ca  7a  4a  2d  44  28
new rondkey:n
b1  8a  1d  4c  d4  7d  7b  66  d8  b9  b3  49  e2  da  de  41
new sub:n
f4  bb  e9  f9  a1  3b  11  88  69  a1  74  da  d6  d8  1b  34
new shift:n
f4  bb  e9  f9  3b  11  88  a1  74  da  69  a1  34  d6  d8  1b
new gal:n
fe  52  fb  ab  2a  3a  81  43  7b  64  c0  2c  20  aa  6a  26
 state after addround:n
4f  d8  e6  e7  fe  47  fa  25  a3  dd  73  65  c2  70  b4  67


 last rondkey:n
 b4  3e  23  6f  ef  92  e9  8f  5b  e2  51  18  cb  11  cf  8e
 last sub:n
 84  61  8e  94  bb  a0  2d  3f  a  c1  8f  4d  25  51  8d  85
 last shift:n
 84  61  8e  94  a0  2d  3f  bb  8f  4d  a  c1  85  25  51  8d
  state after addround:n
 30  5f  ad  fb  4f  bf  d6  34  d4  af  5b  d9  4e  34  9e  3
 dddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
dddddddddddddddddddddddddddddddddddddddddddddd:n
 rond key:n
 10  0  10  0  b4  3e  56  26  ef  92  0  27  6a  40  a  27
  first rond key:n
 b4  3e  23  6f  ef  92  e9  8f  5b  e2  51  18  cb  11  cf  8e
  state after addround:n
 84  61  8e  94  a0  2d  3f  bb  8f  4d  a  c1  85  25  51  8d
 last shift:n
 84  61  8e  94  bb  a0  2d  3f  a  c1  8f  4d  25  51  8d  85
 last sub:n
 4f  d8  e6  e7  fe  47  fa  25  a3  dd  73  65  c2  70  b4  67
 dec. rond key:n
 b1  8a  1d  4c  d4  7d  7b  66  d8  b9  b3  49  e2  da  de  41
 state after addround:n
fe  52  fb  ab  2a  3a  81  43  7b  64  c0  2c  20  aa  6a  26
last gal:n
f4  bb  e9  f9  3b  11  88  a1  74  da  69  a1  34  d6  d8  1b
last shift:n
f4  bb  e9  f9  a1  3b  11  88  69  a1  74  da  d6  d8  1b  34
last sub:n
ba  fe  eb  69  f1  49  e3  97  e4  f1  ca  7a  4a  2d  44  28
dec. rond key:n
e  3b  97  51  f9  a9  6  1d  3  61  a  fa  33  38  4  9f
 state after addround:n
b4  c5  7c  38  8  e0  e5  8a  e7  90  c0  80  79  15  40  b7
last gal:n
a9  17  42  50  3b  2e  dc  ab  be  19  cc  f9  e  80  4b  87
last shift:n
a9  17  42  50  ab  3b  2e  dc  cc  f9  be  19  80  4b  87  e
last sub:n
b7  87  f6  6c  e  49  c3  93  27  69  5a  8e  3a  cc  ea  d7
dec. rond key:n
21  35  ac  c6  75  50  af  1b  17  62  6b  f0  87  b  3c  9b
 state after addround:n
96  b2  5a  aa  7b  19  6c  88  30  b  31  7e  bd  c7  d6  4c


96  b2  5a  aa  7b  19  6c  88  30  b  31  7e  bd  c7  d6  4c
last gal:n
b4  5f  82  3  95  1a  b  d2  d2  28  50  f7  93  a  8  36
last shift:n
b4  5f  82  3  d2  95  1a  b  50  f7  d2  28  a  8  36  93
last sub:n
c6  84  11  d5  7f  ad  43  9e  6c  26  7f  ee  a3  bf  24  22
dec. rond key:n
ec  14  99  6a  61  25  ff  b4  4b  75  9  9b  85  8c  37  a7
 state after addround:n
2a  90  88  bf  1e  88  bc  2a  27  53  76  75  26  33  13  85
last gal:n
f0  14  da  eb  a  6a  56  7e  97  27  86  eb  58  21  5b  1b
last shift:n
f0  14  da  eb  7e  a  6a  56  86  eb  97  27  21  5b  1b  58
last sub:n
17  9b  7a  3c  8a  a3  58  b9  dc  3c  85  3d  7b  57  44  5e
dec. rond key:n
7f  f8  8d  f3  2e  44  da  4b  2b  3e  7c  92  88  9  bb  90
 state after addround:n
68  63  f7  cf  a4  e7  82  f2  f7  2  f9  af  f3  5e  ff  ce
last gal:n
73  ad  e3  a4  97  a6  5d  32  67  67  b  7c  4b  b4  c6  b6
```

```
last shift:n
73  ad  e3  a4  32  97  a6  5d  b  7c  67  67  b4  c6  b6  4b
last sub:n
8f  18  4d  1d  a1  85  c5  8d  9e  1  a  a  c6  c7  79  cc
dec. rond key:n
ee  87  75  7e  6  6a  9e  91  da  15  42  ee  7b  81  b2  2b
 state after addround:n
61  9f  38  63  a7  ef  5b  1c  44  14  48  e4  bd  46  cb  e7
last gal:n
be  4c  84  6  ce  b9  aa  87  e5  83  2f  79  aa  54  e1  84
last shift:n
be  4c  84  6  87  ce  b9  aa  2f  79  e5  83  54  e1  84  aa
last sub:n
5a  5d  4f  a5  ea  ec  db  62  4e  af  2a  41  fd  e0  4f  62
dec. rond key:n
90  69  f2  b  97  6c  f4  f  34  cf  57  ac  50  fa  33  99
 state after addround:n
ca  34  bd  ae  7d  80  2f  6d  7a  60  7d  ed  ad  1a  7c  fb
last gal:n
4  e8  c8  3  53  e1  bb  3f  74  3b  35  f1  43  fc  d5  18
last shift:n
4  e8  c8  3  3f  53  e1  bb  35  f1  74  3b  fc  d5  18  43


last sub:n
30  c8  b1  d5  25  50  e0  fe  d9  2b  ca  49  55  b5  34  64
dec. rond key:n
9b  f9  9b  f9  98  fb  98  fb  98  fb  98  fb  c9  aa  c9  aa
 state after addround:n
ab  31  2a  2c  bd  ab  78  5  41  d0  52  b2  9c  1f  fd  ce
last gal:n
cb  7c  21  fb  13  70  5a  ba  ac  37  b8  cf  bf  6e  3e  db
last shift:n
cb  7c  21  fb  ba  13  70  5a  b8  cf  ac  37  6e  3e  db  bf
last sub:n
59  1  7b  63  c0  82  d0  46  9a  5f  aa  b2  45  d1  9f  f4
dec. rond key:n
62  62  62  62  63  63  63  63  63  63  63  63  63  63  63  63
 state after addround:n
3b  63  19  1  a3  e1  b3  25  f9  3c  c9  d1  26  b2  fc  97
last gal:n
7c  77  30  7b  7c  7b  6b  77  30  30  c5  6f  77  30  1  1
last shift:n
7c  77  30  7b  77  7c  7b  6b  c5  6f  30  30  30  1  1  77
last sub:n
1  2  8  3  2  1  3  5  7  6  8  8  8  9  9  2


last sub:n
1  2  8  3  2  1  3  5  7  6  8  8  8  9  9  2
 state after addround:n
1  2  8  3  2  1  3  5  7  6  8  8  8  9  9  2
FIIIIIIIIIIIIINAAAAAAAAAAAAAAAAALLLLL:n
1  2  8  3  2  1  3  5  7  6  8  8  8  9  9  2
```

CONCLUSION:

The Advanced Encryption Technique was implemented successfully using ‘C’ language. Various data messages were encrypted using different keys and varying key sizes. The original data was properly retrieved via decryption of the cipher text. The modifications brought about in the code was tested and proved to be accurately encrypting and decrypting the data messages with even higher security and immunity against the unauthorized users.

REFERENCES:

[1] AES page available via http://www.nist.gov/CryptoToolkit.4

[2] Computer Security Objects Register (CSOR): http://csrc.nist.gov/csor/.

[3] J. Daemen and V. Rijmen, *AES Proposal: Rijndael*, AES Algorithm Submission, September 3, 1999, available at [1].

[4] J. Daemen and V. Rijmen, *The block cipher Rijndael*, Smart Card research and Applications, LNCS 1820, Springer-Verlag, pp. 288-296.

[5] B. Gladman's AES related home page http://fp.gladman.plus.com/cryptography_technology/.

[6] A. Lee, NIST Special Publication 800-21, *Guideline for Implementing Cryptography in the Federal Government*, National Institute of Standards and Technology, November 1999.

[7] A. Menezes, P. van Oorschot, and S. Vanstone, *Handbook of Applied Cryptography*, CRC Press, New York, 1997, p. 81-83.

[8] J. Nechvatal, ET. al., *Report on the Development of the Advanced Encryption Standard (AES)*, National Institute of Standards and Technology, October 2, 2000, available at [1].