



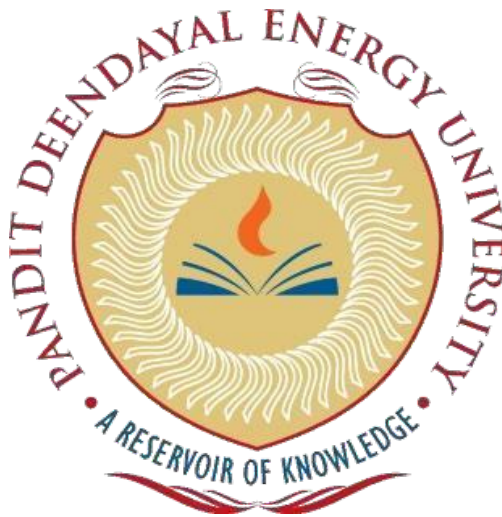
IMPLEMENTATION OF ADVANCED ENCRYPTION STANDARD (AES)

Subject: - Information Security Lab

Name: Kavita Bhatt

Roll No: - 21BIT265

Assignment



Submitted To: - Dr. Manish Mandloi

School of Technology

ICT Department

Aim: To implement AES-128 using MATLAB.

Apparatus required : MATLAB

Theory:

The Advanced Encryption Standard (AES) is a symmetric-key block cipher published by the National Institute of Standards and Technology (NIST) in December 2001.

In 1997, NIST started looking for a replacement for DES, which would be called the Advanced Encryption Standard or AES. The NIST specifications required a block size of 128 bits and three different key sizes of 128, 192, and 256 bits. The specifications also required that AES be an open algorithm, available to the public worldwide. The announcement was made internationally to solicit responses from all over the world. After the First AES Candidate Conference, NIST announced that 15 out of 21 received algorithms had met the requirements and been selected as the first candidates (August 1998). Algorithms were submitted from a number of countries; the variety of these proposals demonstrated the openness of the process and worldwide participation. After the Second AES Candidate Conference, which was held in Rome, NIST announced that 5 out of 15 candidates-MARS, RC6, Rijndael, Serpent, and Twofish - were selected as the finalists (August 1999). After the Third AES Candidate Conference, NIST announced that Rijndael, (pronounced like "Rain Doll"), designed by Belgian researchers Joan Daemen and Vincent Rijment, was selected as Advanced Encryption Standard (October 2000). In February 2001, NIST announced that a draft of the Federal Information Processing Standard (FIPS) was available for public review and comment. Finally, AES was published as FIPS 197 in the Federal Register in December 2001.

Criteria

The criteria defined by NIST for selecting AES fall into three areas: security, cost, and implementation. At the end, Rijndael was judged the best at meeting the combination of these criteria.

Security

The main emphasis was on security. Because NIST explicitly demanded a 128-bit key, this criterion focused on resistance to cryptanalysis attacks other than brute-force attack.

Cost

The second criterion was cost, which covers the computational efficiency and storage requirement for different implementations such as hardware, software, or smart cards.

Implementation

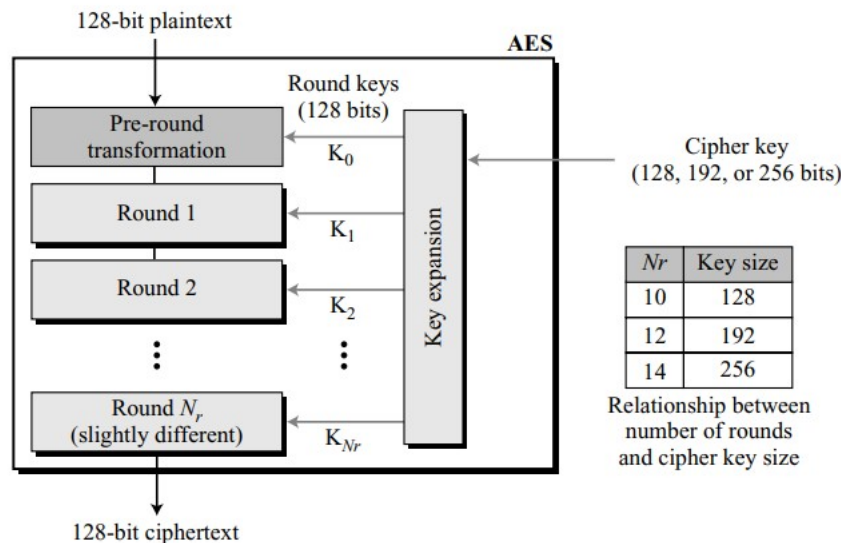
This criterion included the requirement that the algorithm must have flexibility (be implementable on any platform) and simplicity.

Rounds

AES is a non-Feistel cipher that encrypts and decrypts a data block of 128 bits. It uses 10, 12, or 14 rounds. The key size, which can be 128, 192, or 256 bits, depends on the number of rounds. Figure 1 shows the general design for the encryption algorithm (called cipher); the decryption algorithm (called inverse cipher) is similar, but the round keys are applied in the reverse order. In Figure 1, N_r defines the

number of rounds. The figure also shows the relationship between the number of rounds and the key size, which means that we can have three different AES versions; they are referred as AES-128, AES-192, and AES-256. However, the round keys, which are created by the key-expansion algorithm are always 128 bits, the same size as the plaintext or ciphertext block.

(AES has defined three versions, with 10, 12, and 14 rounds. Each version uses a different cipher key size (128, 192, or 256), but the round keys are always 128 bits.)



General design of AES encryption cipher(Figure 1)

The number of round keys generated by the key-expansion algorithm is always one more than the number of rounds. In other words, we have Number of round keys = $N_r + 1$. We refer to the round keys as $K_0, K_1, K_2, \dots, K_{N_r}$.

Data Units

AES uses five units of measurement to refer to data: bits, bytes, words, blocks, and state. The bit is the smallest and atomic unit; other units can be expressed in terms of smaller ones. Figure 2 shows the non-atomic data units: byte, word, block, and state.

Bit

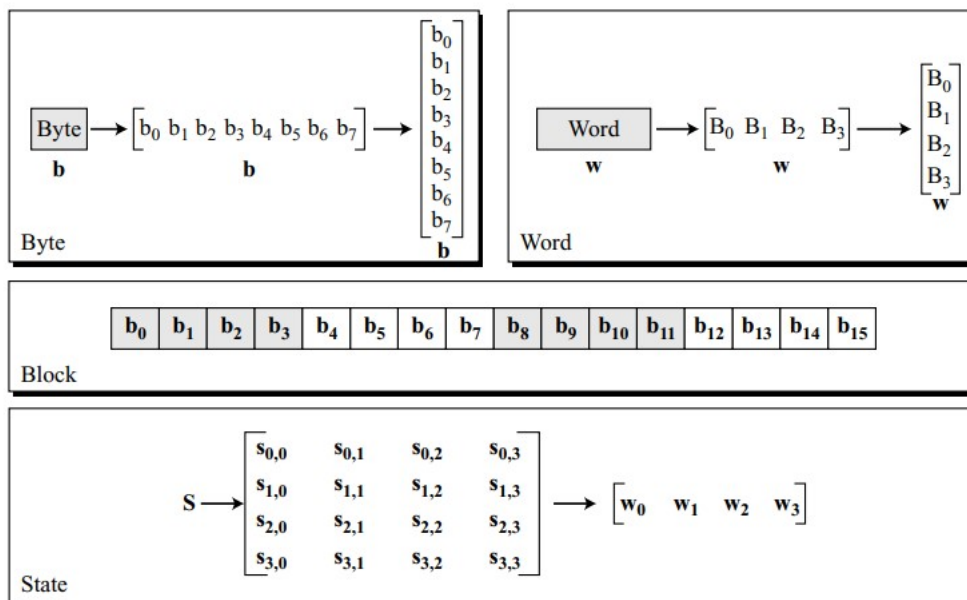
In AES, a bit is a binary digit with a value of 0 or 1. We use a lowercase letter to refer to a bit.

Byte

A byte is a group of eight bits that can be treated as a single entity, a row matrix (1×8) of eight bits, or a column matrix (8×1) of eight bits. When treated as a row matrix, the bits are inserted to the matrix from left to right; when treated as a column matrix, the bits are inserted into the matrix from top to bottom. We use a lowercase bold letter to refer to a byte.

Word

A word is a group of 32 bits that can be treated as a single entity, a row matrix of four bytes, or a column matrix of four bytes. When it is treated as a row matrix, the bytes are inserted into the matrix from left to right; when it is considered as a column matrix, the bytes are inserted into the matrix from top to bottom. We use the lowercase bold letter w to show a word.



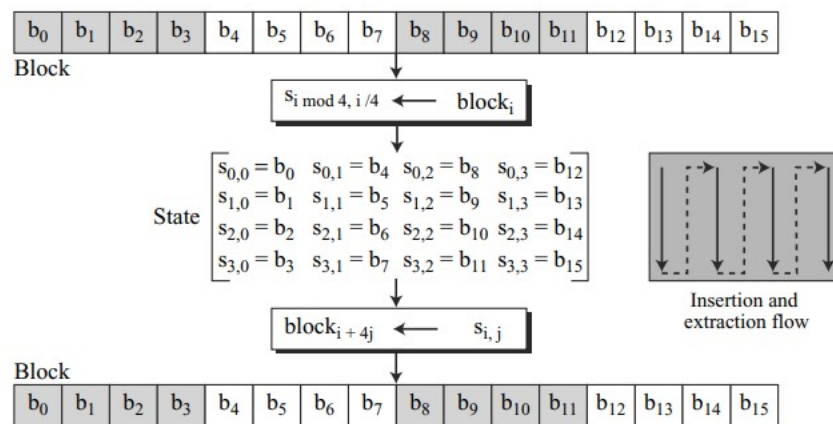
Data units used in AES(Figure 2)

Block

AES encrypts and decrypts data blocks. A block in AES is a group of 128 bits. However, a block can be represented as a row matrix of 16 bytes.

State

AES uses several rounds in which each round is made of several stages. Data block is transformed from one stage to another. At the beginning and end of the cipher, AES uses the term data block; before and after each stage, the data block is referred to as a state. We use an uppercase bold letter to refer to a state. Although the states in different stages are normally called S , we occasionally use the letter T to refer to a temporary state. States, like blocks, are made of 16 bytes, but normally are treated as matrices of 4×4 bytes. In this case, each element of a state is referred to as $s_{r,c}$, where r (0 to 3) defines the row and the c (0 to 3) defines the column. Occasionally, a state is treated as a row matrix (1×4) of words. This makes sense, if we think of a word as a column matrix. At the beginning of the cipher, bytes in a data block are inserted into a state column by column, and in each column, from top to bottom. At the end of the cipher, bytes in the state are extracted in the same way, as shown in Figure 3.

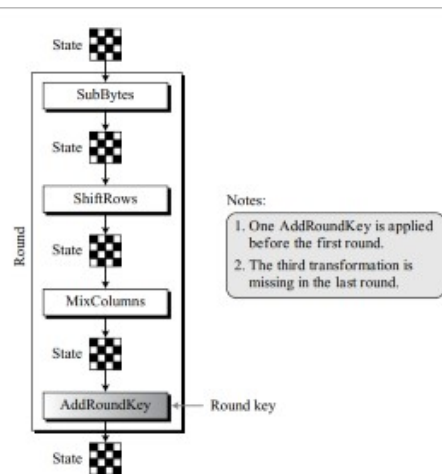


Block-to-state and state-to-block transformation(Figure 3)

Structure of Each Round

Figure 4 shows the structure of each round at the encryption side. Each round, except the last, uses four transformations that are invertible. The last round has only three transformations.

As Figure 4 shows, each transformation takes a state and creates another state to be used for the next transformation or the next round. The pre-round section uses only one transformation (AddRoundKey); the last round uses only three transformations (MixColumns transformation is missing).



Structure of each round at the encryption site(Figure 4)

TRANSFORMATIONS

To provide security, AES uses four types of transformations: substitution, permutation, mixing, and key-adding. We will discuss each here.

Substitution

AES, like DES, uses substitution. However, the mechanism is different. First, the substitution is done for each byte. Second, only one table is used for transformation of every byte, which means that if two bytes are the same, the transformation is also the same. Third, the transformation is defined by either a table lookup process or mathematical calculation in the GF(28) field. AES uses two invertible transformations.

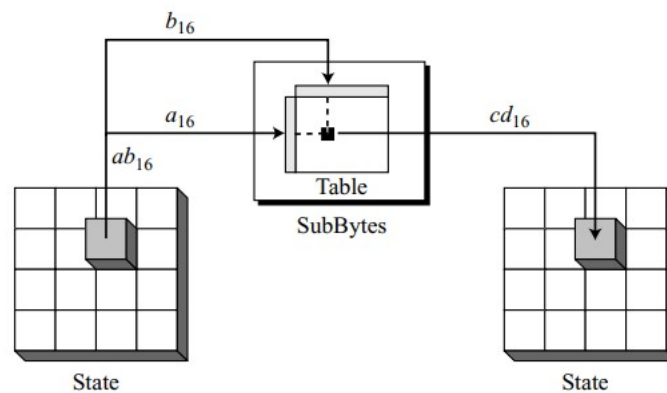
SubBytes

The first transformation, SubBytes, is used at the encryption site. To substitute a byte, we interpret the byte as two hexadecimal digits. The left digit defines the row and the right digit defines the column of the substitution table. The two hexadecimal digits at the junction of the row and the column are the new byte. Figure 5 shows the idea.

In the SubBytes transformation, the state is treated as a 4×4 matrix of bytes. Transformation is done one byte at a time. The contents of each byte is changed, but the arrangement of the bytes in the matrix remains the same. In the process, each byte is transformed independently. There are sixteen distinct byte-to-byte transformations

InvSubBytes

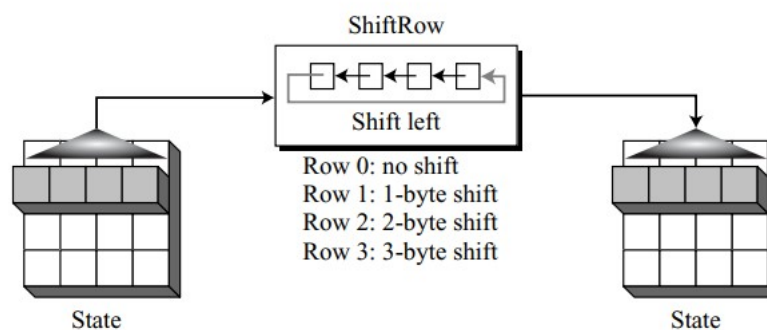
InvSubBytes is the inverse of SubBytes. It reverses the SubBytes operation used in encryption. Each byte of data is replaced with a corresponding byte from the inverse S-box. This nonlinear transformation recovers the original data by reversing the confusion introduced during encryption.



SubBytes transformations(Figure 5)

ShiftRows

In the encryption, the transformation is called ShiftRows and the shifting is to the left. The number of shifts depends on the row number (0, 1, 2, or 3) of the state matrix. This means the row 0 is not shifted at all and the last row is shifted three bytes. Figure 6 shows the shifting transformation.



ShiftRows transformation(Figure 6)

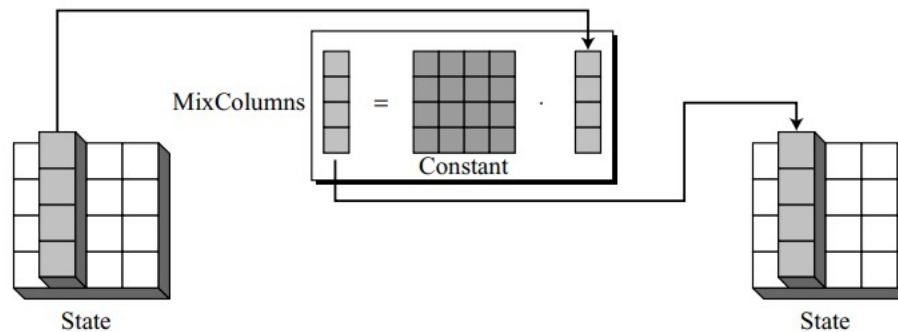
MixColumns

The MixColumns transformation operates at the column level; it transforms each column of the state to a new column. The transformation is actually the matrix multiplication of a state column by a constant square matrix. The bytes in the state column and constants matrix are interpreted as 8-bit words

(or polynomials) with coefficients in $GF(2)$. Multiplication of bytes is done in $GF(2^8)$ with modulus (10001101) or $(x^8 + x^4 + x^3 + x + 1)$. Addition is the same as XORing of 8-bit words. Figure 7 shows the MixColumns transformations.

InvMixColumns

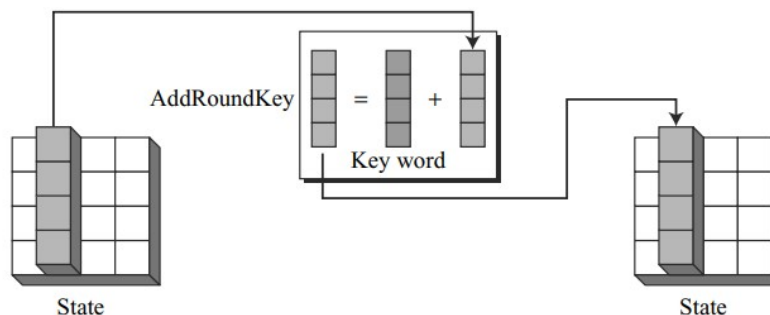
The InvMixColumns transformation is basically the same as the MixColumns transformation. If the two constant matrices are inverses of each other, it is easy to prove that the two transformations are inverses of each other.



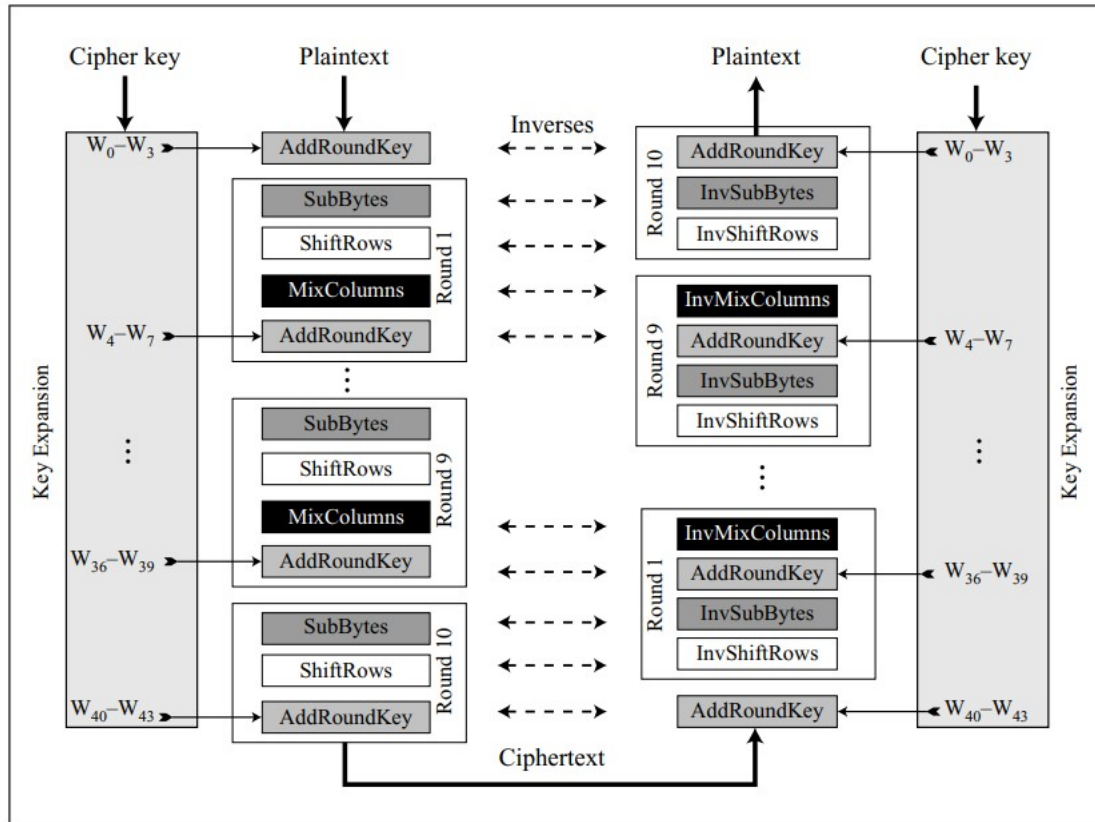
MixColumns transformation(Figure 7)

AddRoundKey

AddRoundKey also proceeds one column at a time. It is similar to MixColumns in this respect. MixColumns multiplies a constant square matrix by each state column; AddRoundKey adds a round key word with each state column matrix. The operation in MixColumns is matrix multiplication; the operation in AddRoundKey is matrix addition. Since addition and subtraction in this field are the same, the AddRoundKey transformation is the inverse of itself. Figure 8 shows the AddRoundKey transformation.



AddRoundKey transformation(Figure 8)



Cipher and reverse cipher design (Figure 9)

First, the order of SubBytes and ShiftRows is changed in the reverse cipher. Second, the order of MixColumns and AddRoundKey is changed in the reverse cipher. This difference in ordering is needed to make each transformation in the cipher aligned with its inverse in the reverse cipher. Consequently, the decryption algorithm as a whole is the inverse of the encryption algorithm. We have shown only three rounds, but the rest is the same. Note that the round keys are used in the reverse order. Note that the encryption and decryption algorithms in the original design are not similar.

Pseudocode for cipher :

```
Cipher (InBlock [16], OutBlock[16], w[0 ... 43])
{
    BlockToState (InBlock, S)

    S ← AddRoundKey (S, w[0...3])
    for (round = 1 to 10)
    {
        S ← SubBytes (S)
        S ← ShiftRows (S)
        if (round ≠ 10) S ← MixColumns (S)
        S ← AddRoundKey (S, w[4 × round, 4 × round + 3])
    }

    StateToBlock (S, OutBlock);
}
```

MATLAB IMPLEMENTATION

1) Main file(AES):

```
clc;
```

```
clear all;
```

```
close all;
```

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

```

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

```

```

invSBox = [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];

```

```

plainText = input("Enter Text ','s");
cipherKey = "HELLOEVERYONE";

```

```

TEXT_DEC = uint8(char(plainText));
npad = uint8(16 - (length(TEXT_DEC) - 16));

```

```

TEXT_DEC = [TEXT_DEC repmat(zeros, 1, npad)];

KEY_DEC = uint8(char(cipherKey));
npad = uint8(16 - length(KEY_DEC));
KEY_DEC = [KEY_DEC repmat(zeros, 1, npad)];
K = KEY_DEC;

ENCRYPT = [];
for i = 1:16:length(TEXT_DEC)
    block = Cipher(TEXT_DEC(i:i+15), K, 4, 4,10);
    ENCRYPT = [ENCRYPT block];
end

fprintf("Cipher Text %s",char(ENCRYPT));

DECRYPT = [];
for i = 1:16:length(TEXT_DEC)
    Dec_Block= InvCipher(ENCRYPT(i:i+15), K, 4, 4,10);
    DECRYPT = [DECRYPT Dec_Block];
end

DECRYPT = DECRYPT(:);
fprintf("\nPlain text: %s",char(DECRYPT(1:end-DECRYPT(end))));

```

2) SubWord:

```

function [out] = SubWord(in)
% A function that takes a four-byte input word and applies the S-box
% to each of the four bytes to produce an output word

mx = hex2dec('11B');
sbox = [0x63 0x7c 0x77 0x7b 0xf2 0x6b 0x6f 0xc5 0x30 0x01 0x67 0x2b 0xfe 0xd7 0xab 0x76;

```

```

0xca 0x82 0xc9 0x7d 0xfa 0x59 0x47 0xf0 0xad 0xd4 0xa2 0xaf 0x9c 0xa4 0x72 0xc0;
0xb7 0xfd 0x93 0x26 0x36 0x3f 0xf7 0xcc 0x34 0xa5 0xe5 0xf1 0x71 0xd8 0x31 0x15;
0x04 0xc7 0x23 0xc3 0x18 0x96 0x05 0x9a 0x07 0x12 0x80 0xe2 0xeb 0x27 0xb2 0x75;
0x09 0x83 0x2c 0x1a 0x1b 0x6e 0x5a 0xa0 0x52 0x3b 0xd6 0xb3 0x29 0xe3 0x2f 0x84;
0x53 0xd1 0x00 0xed 0x20 0xfc 0xb1 0x5b 0x6a 0xcb 0xbe 0x39 0x4a 0x4c 0x58 0xcf;
0xd0 0xef 0xaa 0xfb 0x43 0x4d 0x33 0x85 0x45 0xf9 0x02 0x7f 0x50 0x3c 0x9f 0xa8;
0x51 0xa3 0x40 0x8f 0x92 0x9d 0x38 0xf5 0xbc 0xb6 0xda 0x21 0x10 0xff 0xf3 0xd2;
0xcd 0x0c 0x13 0xec 0x5f 0x97 0x44 0x17 0xc4 0xa7 0x7e 0x3d 0x64 0x5d 0x19 0x73;
0x60 0x81 0x4f 0xdc 0x22 0x2a 0x90 0x88 0x46 0xee 0xb8 0x14 0xde 0x5e 0x0b 0xdb;
0xe0 0x32 0x3a 0x0a 0x49 0x06 0x24 0x5c 0xc2 0xd3 0xac 0x62 0x91 0x95 0xe4 0x79;
0xe7 0xc8 0x37 0x6d 0x8d 0xd5 0x4e 0xa9 0x6c 0x56 0xf4 0xea 0x65 0x7a 0xae 0x08;
0xba 0x78 0x25 0x2e 0x1c 0xa6 0xb4 0xc6 0xe8 0xdd 0x74 0x1f 0x4b 0xbd 0x8b 0x8a;
0x70 0x3e 0xb5 0x66 0x48 0x03 0xf6 0x0e 0x61 0x35 0x57 0xb9 0x86 0xc1 0x1d 0x9e;
0xe1 0xf8 0x98 0x11 0x69 0xd9 0x8e 0x94 0x9b 0x1e 0x87 0xe9 0xce 0x55 0x28 0xdf;
0x8c 0xa1 0x89 0x0d 0xbf 0xe6 0x42 0x68 0x41 0x99 0x2d 0x0f 0xb0 0x54 0xbb 0x16];
out = arrayfun(@(x) sbox(floor(x/16) + 1, mod(x, 16) + 1), double(in.x));
out = gf(out, 8, mx);
end

```

3) SubBytes:

```

% SubBytes() Transformation
% Transformation in the Cipher that processes the State using a nonlinear
% byte substitution table (S-box) that operates on each of the State bytes
% independently
% SUBSTITUTES BYTES OF INPUT FROM S-BOX

function [state] = SubBytes(state)

sbox = [0x63 0x7c 0x77 0x7b 0xf2 0x6b 0x6f 0xc5 0x30 0x01 0x67 0x2b 0xfe 0xd7 0xab 0x76;

```

```

0xca 0x82 0xc9 0x7d 0xfa 0x59 0x47 0xf0 0xad 0xd4 0xa2 0xaf 0x9c 0xa4 0x72 0xc0;
0xb7 0xfd 0x93 0x26 0x36 0x3f 0xf7 0xcc 0x34 0xa5 0xe5 0xf1 0x71 0xd8 0x31 0x15;
0x04 0xc7 0x23 0xc3 0x18 0x96 0x05 0x9a 0x07 0x12 0x80 0xe2 0xeb 0x27 0xb2 0x75;
0x09 0x83 0x2c 0x1a 0x1b 0x6e 0x5a 0xa0 0x52 0x3b 0xd6 0xb3 0x29 0xe3 0x2f 0x84;
0x53 0xd1 0x00 0xed 0x20 0xfc 0xb1 0x5b 0x6a 0xcb 0xbe 0x39 0x4a 0x4c 0x58 0xcf;
0xd0 0xef 0xaa 0xfb 0x43 0x4d 0x33 0x85 0x45 0xf9 0x02 0x7f 0x50 0x3c 0x9f 0xa8;
0x51 0xa3 0x40 0x8f 0x92 0x9d 0x38 0xf5 0xbc 0xb6 0xda 0x21 0x10 0xff 0xf3 0xd2;
0xcd 0x0c 0x13 0xec 0x5f 0x97 0x44 0x17 0xc4 0xa7 0x7e 0x3d 0x64 0x5d 0x19 0x73;
0x60 0x81 0x4f 0xdc 0x22 0x2a 0x90 0x88 0x46 0xee 0xb8 0x14 0xde 0x5e 0x0b 0xdb;
0xe0 0x32 0x3a 0x0a 0x49 0x06 0x24 0x5c 0xc2 0xd3 0xac 0x62 0x91 0x95 0xe4 0x79;
0xe7 0xc8 0x37 0x6d 0x8d 0xd5 0x4e 0xa9 0x6c 0x56 0xf4 0xea 0x65 0x7a 0xae 0x08;
0xba 0x78 0x25 0x2e 0x1c 0xa6 0xb4 0xc6 0xe8 0xdd 0x74 0x1f 0x4b 0xbd 0x8b 0x8a;
0x70 0x3e 0xb5 0x66 0x48 0x03 0xf6 0x0e 0x61 0x35 0x57 0xb9 0x86 0xc1 0x1d 0x9e;
0xe1 0xf8 0x98 0x11 0x69 0xd9 0x8e 0x94 0x9b 0x1e 0x87 0xe9 0xce 0x55 0x28 0xdf;
0x8c 0xa1 0x89 0x0d 0xbf 0xe6 0x42 0x68 0x41 0x99 0x2d 0x0f 0xb0 0x54 0xbb 0x16];
state = arrayfun(@(x) sbox(floor(x/16) + 1, mod(x, 16) + 1),double(state));
end

```

4) ShiftRows:

```

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

function [state] = ShiftRows(state)
    for i = 2:4
        state(i, :) = circshift(state(i, :), 1-i);
    end
end

```

end

5) SBox:

```
function [sbox] = SBox()

mx = hex2dec('11B');
row = [1 1 1 1 1 0 0 0];

% Set up constants A and b.
A = gf(gallery('circul', row), 2);
b = gf([0; 1; 1; 0; 0; 0; 1; 1], 2);
xinv = gf(0:255, 8, mx);

% Compute inverses.
xinv(2:256) = 1./xinv(2:256);

y = gf(transpose(dec2bin(xinv.x,8) == '1'),2);

% Affine transformation.
z = A * y + repmat(b,1,256);

sbox = transpose(uint8(bin2dec(num2str(transpose(z.x))))); % Reformat to make sbox.

sbox = reshape(sbox, 16, 16);

end
```

6) RotWord:

```
function [out] = RotWord(in)

out = circshift(in, -1);

end
```

7) Rcon:

```
function [out] = Rcon(i)

mx = hex2dec('11B');

temp = gf([0x02, 0x00, 0x00, 0x00], 8, mx);
```

```

out = gf([0x01, 0x00, 0x00, 0x00], 8, mx);
for j = 1:(i-1)
    out = out .* temp;
end
out = out.x;
end

```

8) MixColumns:

```

function [state] = MixColumns(state)
    mx = hex2dec('11B');
    row = [2 3 1 1];
    A = gf(gallery('circul', row), 8, mx);
    state = A * state;
    state = state.x;
end

```

9) KeyExpansion:

```

function [w] = KeyExpansion(K, Nk, Nb, Nr)
    mx = hex2dec('11B');
    warning('off');
    w = gf(zeros(Nb*(Nr+1), 4), 8, mx);

    for i = 1:Nk
        j = 4*(i-1)+1;
        w(i,:) = [K(j) K(j+1) K(j+2) K(j+3)];
    end
    for i = Nk+1:Nb*(Nr+1)
        temp = w(i-1, :);
        if mod(i-1, Nk) == 0

```



```

        temp = SubWord(RotWord(temp)) + Rcon((i-1)/Nk);
elseif mod(i-1,Nk) == 4
    temp = SubWord(temp);
end

w(i, :) = w(i-Nk, :) + temp;
end

w = int32(w.x);
end

```

10) InvSubBytes:

```

function [state] = InvSubBytes(state)

invsbox = [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];

state = arrayfun(@(x) invsbox(floor(x/16) + 1, mod(x, 16) + 1), double(state));
end

```

11) InvShiftRows:

```
function [state] = InvShiftRows(state)

    for i = 2:4
        state(i, :) = circshift(state(i, :), i-1);
    end
end
```

12) InvSBox:

```
function invsbox = InvSBox()

    mx = hex2dec('11B');

    row = [0 1 0 1 0 0 1 0]; % set up constants A and b.
    A = gf(gallery('circul', row), 2);
    b = gf([0; 0; 0; 0; 0; 1; 0; 1], 2);
    y = gf(transpose(dec2bin(0:255) == '1'), 2); % Affine transformation.
    z = A * y + repmat(b,1,256);
    z = gf(bin2dec(num2str(transpose(uint32(z.x)))), 8, mx); % Reformat.
    zinv = gf(0:255, 8, mx); % Invert results.
    zinv(1:99) = 1./z(1:99);
    zinv(100) = 0;
    zinv(101:256) = 1./z(101:256);
    invsbox = uint8(uint32(zinv.x)); % Reformat to make inverse sbox.
    invsbox = reshape(invsbox, 16, 16);
end
```

13) InvMixColumns:

```
function [state] = InvMixColumns(state)

    mx = hex2dec('11B');
```

```

row = [14 11 13 9];
A = gf(gallery('circul', row), 8, mx);
state = A * state;
state = state.x;
end

```

14) InvCipher:

% DECRYPT

```
function [out] = InvCipher(in, K, Nk, Nb, Nr)
```

```

w = KeyExpansion(K, Nk, Nb, Nr);

```

```

in = reshape(in, Nb, 4);

```

```

state = int32(in);

```

```

state = AddRoundKey(state, w(Nr*Nb+1:(Nr+1)*Nb, :)', Nb);

```

```

for round = (Nr-1):-1:1

```

```

    state = InvShiftRows(state);

```

```

    state = InvSubBytes(state);

```

```

    state = AddRoundKey(state, w(round*Nb+1:(round+1)*Nb, :)', Nb);

```

```

    state = InvMixColumns(state);

```

```

end

```

```

state = InvShiftRows(state);

```

```

state = InvSubBytes(state);

```

```

state = AddRoundKey(state, w(1:Nb, :)', Nb);

```

```

out = state;

```

```

end

```

15) Cipher:

```
% ENCRYPT
```

```
function [out] = Cipher(in, K, Nk, Nb, Nr)
```

```
    in = reshape(in, Nb, 4);
```

```
    state = int32(in);
```

```
    w = KeyExpansion(K, Nk, Nb, Nr);
```

```
    state = AddRoundKey(state, w(1:Nb, :)', Nb);
```

```
    for round = 1:(Nr-1)
```

```
        state = SubBytes(state);
```

```
        state = ShiftRows(state);
```

```
        state = MixColumns(state);
```

```
        state = AddRoundKey(state, w(round*Nb+1:(round+1)*Nb, :)', Nb);
```

```
    end
```

```
    state = SubBytes(state);
```

```
    state = ShiftRows(state);
```

```
    state = AddRoundKey(state, w(Nr*Nb+1:(Nr+1)*Nb, :)', Nb);
```

```
    out = state;
```

```
end
```

16) AddRoundKey:

```
% Add Round Key
```

```
% In the AddRoundKey() transformation, a Round Key is added to the State
```

```
% by a simple bitwise XOR operation.
```

```
function [state] = AddRoundKey(state, w, Nb)
```

```
    mx = hex2dec('11B');
```

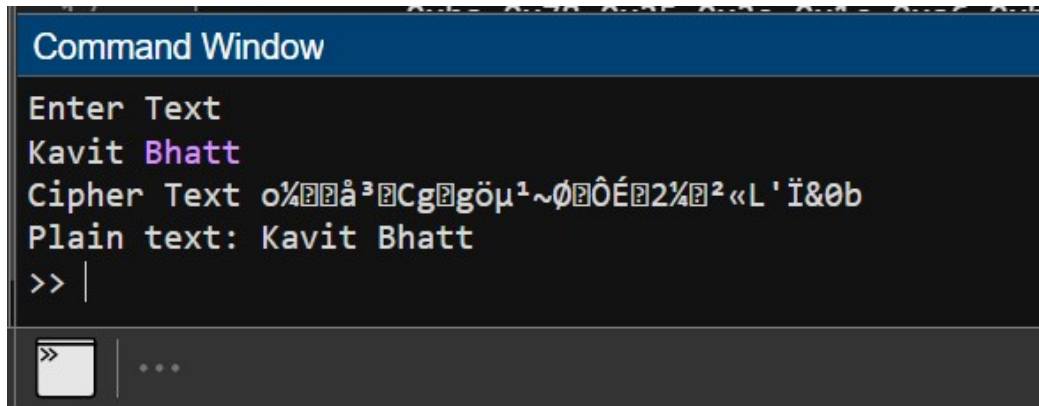
```
    w = gf(w, 8, mx);
```

```
    state = gf(state, 8, mx);
```

```
for c = 1:Nb
    state(c, :) = state(c, :) + w(c, :);
end

state = int16(state.x);
end
```

OUTPUT :



```
Command Window
Enter Text
Kavit Bhatt
Cipher Text o%å³Cgögöµ¹~øÔÉ²²«L'ï&øb
Plain text: Kavit Bhatt
>> |
```

Following is a brief review of the three characteristics of AES:

- **Security**

AES was designed after DES. Most of the known attacks on DES were already tested on AES; none of them has broken the security of AES so far.

Brute-Force Attack

AES is definitely more secure than DES due to the larger-size key (128, 192, and 256 bits). Let us compare DES with 56-bit cipher key and AES with 128-bit cipher key. For DES we need 256 (ignoring the key complement issue) tests to find the key; for AES we need 2^{128} tests to find the key. This means that if we can break DES in t seconds, we need $(2^{72} \times t)$ seconds to break AES. This would be almost impossible. In addition, AES provides two other versions with longer cipher keys. The lack of weak keys is another advantage of AES over DES.

Statistical Attacks

The strong diffusion and confusion provided by the combination of the SubBytes, ShiftRows, and MixColumns transformations removes any frequency pattern in the plaintext. Numerous tests have failed to do statistical analysis of the ciphertext.

Differential and Linear Attacks

AES was designed after DES. Differential and linear cryptanalysis attacks were no doubt taken into consideration. There are no differential and linear attacks on AES as yet.

- **Implementation**

AES can be implemented in software, hardware, and firmware. The implementation can use table lookup process or routines that use a well-defined algebraic structure. The transformation can be either byte-oriented or word-oriented. In the byte-oriented version, the whole algorithm can use an 8-bit processor; in the word-oriented version, it can use a 32-bit processor. In either case, the design of constants makes processing very fast.

- **Simplicity and Cost**

The algorithms used in AES are so simple that they can be easily implemented using cheap processors and a minimum amount of memory.