PRN: 2019BTECS00037

Name: Onkar Santosh Gavali

Batch: B2

**CNS Lab Assignment 7: Advanced Encryption Standard**

**Title:** Implementation of Advanced Encryption Standard

**Aim:** To implement AES using C/C++/Java/Python or any other programming language.

**Introduction:**

AES (Advanced Encryption Standard), also known as Rijndael encryption method in cryptography, is a block encryption standard adopted by the federal government of the United States. This standard is used to replace the original DES, which has been widely used all over the world and has become one of the most popular symmetric key algorithms.

Before the advent of AES, the most commonly used symmetric key algorithm was DES Encryption Algorithms It's in In 1977, it was published as the commercial encryption standard of the U.S. government. The main problem of DES is that the key length is short, and it is not suitable for the requirement of data encryption security in distributed open network. Therefore, in 1998, the U.S. government decided not to continue using DES as the federal encryption standard, and launched a campaign to solicit AES candidate algorithms. The basic requirements for AES are: faster than triple DES, at least as secure as triple DES, 128 bits of data packet length and 128/192/256 bits of key length.

After more than three years of selection, the Rijndael algorithm designed by Belgian cryptographers finally emerged as a new generation of advanced encryption standards, which was published by the National Institute of Standards and Technology (NIST) in 2001. FIPS PUB 197.

**Theory:**

AES algorithm (Rijndael algorithm) is a symmetric block cipher algorithm. The length of the data packet must be 128 bits, and the length of the key used should be 128, 192 or 256 bits.

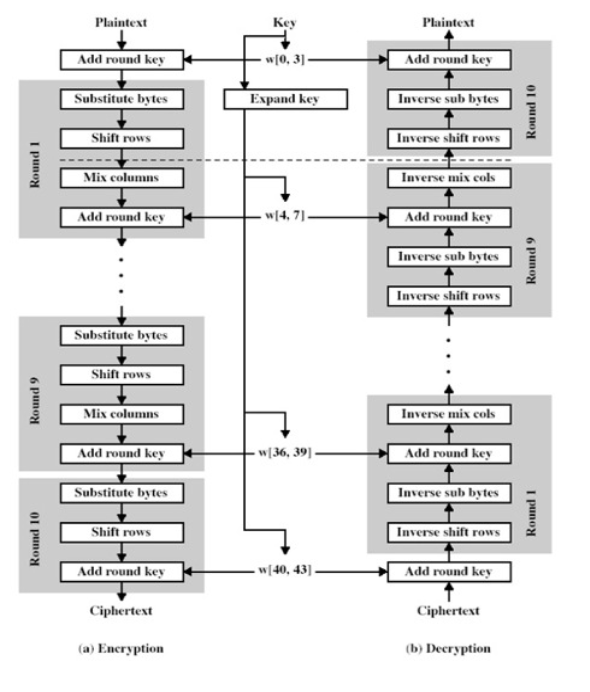
AES performs operations on bytes of data rather than in bits. Since the block size is 128 bits, the cipher processes 128 bits (or 16 bytes) of the input data at a time.

The number of rounds depends on the key length as follows :

* 128 bit key – 10 rounds
* 192 bit key – 12 rounds
* 256 bit key – 14 rounds

Each round comprises of 4 steps :

* SubBytes
* ShiftRows
* MixColumns
* Add Round Key



**Procedure:**

**Code=>**

#include <stdio.h>

#include <iostream>

#include <stdlib.h>

#include <string.h>

using namespace std;

// The number of columns comprising a state in AES. This is a parameter

// that could be 4, 6, or 8.  For this example we set it to 4.

#define Nb 4

// The number of rounds in AES Cipher. It is initialized to zero.

// The actual value is computed from the input.

int Nr=0;

// The number of 32 bit words in the key. It is initialized to zero.

// The actual value is computed from the input.

int Nk=0;

// in - the array that holds the plain text to be encrypted.

// out - the array that holds the cipher text.

// state - the array that holds the intermediate results during encryption.

unsigned char in[1024], out[1024], state[4][Nb];

// The array that stores the round keys.

unsigned char RoundKey[240];

// The Key input to the AES Program

unsigned char Key[32];

int getSBoxValue(int num) {

   int 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

   return sbox[num];

}

// The round constant word array, Rcon[i], contains the values given by

// x to the power (i-1) being powers of x (x is denoted as {02}) in the

// field GF(28).  Note that i starts at 1, not 0).

int 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  };

// This function produces Nb(Nr+1) round keys. The round keys are used in

// each round to encrypt the states.

void KeyExpansion() {

   int i,j;

   unsigned char temp[4],k;

   // The first round key is the key itself.

   for (i=0 ; i < Nk ; i++) {

      RoundKey[i\*4] = Key[i\*4];

      RoundKey[i\*4+1] = Key[i\*4+1];

      RoundKey[i\*4+2] = Key[i\*4+2];

      RoundKey[i\*4+3] = Key[i\*4+3];

   }

   // All other round keys are found from the previous round keys.

   while (i < (Nb \* (Nr+1))) {

      for (j=0 ; j < 4 ; j++) {

   temp[j] = RoundKey[(i-1) \* 4 + j];

      }

      if (i % Nk == 0) {

   // This function rotates the 4 bytes in a word to the left once.

   // [a0,a1,a2,a3] becomes [a1,a2,a3,a0]

   // Function RotWord()

   k = temp[0];

   temp[0] = temp[1];

   temp[1] = temp[2];

   temp[2] = temp[3];

   temp[3] = k;

   // SubWord() is 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.

   // Function Subword()

   temp[0] = getSBoxValue(temp[0]);

   temp[1] = getSBoxValue(temp[1]);

   temp[2] = getSBoxValue(temp[2]);

   temp[3] = getSBoxValue(temp[3]);

   temp[0] =  temp[0] ^ Rcon[i/Nk];

      } else if (Nk > 6 && i % Nk == 4) {

   // Function Subword()

   temp[0] = getSBoxValue(temp[0]);

   temp[1] = getSBoxValue(temp[1]);

   temp[2] = getSBoxValue(temp[2]);

   temp[3] = getSBoxValue(temp[3]);

      }

      RoundKey[i\*4+0] = RoundKey[(i-Nk)\*4+0] ^ temp[0];

      RoundKey[i\*4+1] = RoundKey[(i-Nk)\*4+1] ^ temp[1];

      RoundKey[i\*4+2] = RoundKey[(i-Nk)\*4+2] ^ temp[2];

      RoundKey[i\*4+3] = RoundKey[(i-Nk)\*4+3] ^ temp[3];

      i++;

   }

}

// This function adds the round key to state.

// The round key is added to the state by an XOR function.

void AddRoundKey(int round) {

   int i,j;

   for (i=0 ; i < Nb ; i++) {

      for (j=0 ; j < 4 ; j++) {

   state[j][i] ^= RoundKey[round \* Nb \* 4 + i \* Nb + j];

      }

   }

}

// The SubBytes Function Substitutes the values in the

// state matrix with values in an S-box.

void SubBytes() {

   int i,j;

   for (i=0 ; i < 4 ; i++) {

      for (j=0 ; j < Nb ; j++) {

   state[i][j] = getSBoxValue(state[i][j]);

      }

   }

}

// The ShiftRows() function shifts the rows in the state to the left.

// Each row is shifted with different offset.

// Offset = Row number. So the first row is not shifted.

void ShiftRows() {

   unsigned char temp;

   // Rotate first row 1 columns to left

   temp = state[1][0];

   state[1][0] = state[1][1];

   state[1][1] = state[1][2];

   state[1][2] = state[1][3];

   state[1][3] = temp;

   // Rotate second row 2 columns to left

   temp = state[2][0];

   state[2][0] = state[2][2];

   state[2][2] = temp;

   temp = state[2][1];

   state[2][1] = state[2][3];

   state[2][3] = temp;

   // Rotate third row 3 columns to left

   temp = state[3][0];

   state[3][0] = state[3][3];

   state[3][3] = state[3][2];

   state[3][2] = state[3][1];

   state[3][1] = temp;

}

// xtime is a macro that finds the product of {02} and the argument to

// xtime modulo {1b}

#define xtime(x)   ((x<<1) ^ (((x>>7) & 1) \* 0x1b))

// MixColumns function mixes the columns of the state matrix

void MixColumns() {

   int i;

   unsigned char Tmp,Tm,t;

   for (i=0 ; i < Nb ; i++) {

      t = state[0][i];

      Tmp = state[0][i] ^ state[1][i] ^ state[2][i] ^ state[3][i] ;

      Tm = state[0][i] ^ state[1][i] ;

      Tm = xtime(Tm);

      state[0][i] ^= Tm ^ Tmp ;

      Tm = state[1][i] ^ state[2][i] ;

      Tm = xtime(Tm);

      state[1][i] ^= Tm ^ Tmp ;

      Tm = state[2][i] ^ state[3][i] ;

      Tm = xtime(Tm);

      state[2][i] ^= Tm ^ Tmp ;

      Tm = state[3][i] ^ t ;

      Tm = xtime(Tm);

      state[3][i] ^= Tm ^ Tmp ;

   }

}

// Cipher is the main function that encrypts the PlainText.

void Cipher() {

   int i,j,round=0;

   //Copy the input PlainText to state array.

   for (i=0 ; i < Nb ; i++) {

      for (j=0 ; j < 4 ; j++) {

   state[j][i] = in[i\*4 + j];

      }

   }

   // Add the First round key to the state before starting the rounds.

   AddRoundKey(0);

   // There will be Nr rounds.

   // The first Nr-1 rounds are identical.

   // These Nr-1 rounds are executed in the loop below.

   for (round=1 ; round < Nr ; round++) {

      SubBytes();

      ShiftRows();

      MixColumns();

      AddRoundKey(round);

   }

   // The last round is given below.

   // The MixColumns function is not here in the last round.

   SubBytes();

   ShiftRows();

   AddRoundKey(Nr);

   // The encryption process is over.

   // Copy the state array to output array.

   for (i=0 ; i < Nb ; i++) {

      for (j=0 ; j < 4 ; j++) {

   out[i\*4+j]=state[j][i];

      }

   }

}

int fillBlock (int sz, char \*str, unsigned char \*in) {

   int j=0;

   while (sz < strlen(str)) {

      if (j >= Nb\*4) break;

      in[j++] = (unsigned char)str[sz];

      sz++;

   }

   // Pad the block with 0s, if necessary

   if (sz >= strlen(str)) for ( ; j < Nb\*4 ; j++) in[j] = 0;

   return sz;

}

int main(int argc, char \*\*argv) {

   int i;

   if (argc != 2) {

      cerr << "Usage: " << argv[0] << " <keysize: 1=128, 2=192, 3=256>\n";

      exit(0);

   }

   switch (atoi(argv[1])) {

   case 1: Nk = 4; break;

   case 2: Nk = 6; break;

   case 3: Nk = 8; break;

   default: Nk = 4; break;

   }

   // Calculate Nr from Nk and, implicitly, from Nb

   Nr = Nk + 6;

   // The key values are placed here

   Key[0]  = 0x2b;  Key[1]  = 0x7e;  Key[2]  = 0x15;  Key[3]  = 0x16;

   Key[4]  = 0x28;  Key[5]  = 0xae;  Key[6]  = 0xd2;  Key[7]  = 0xa6;

   Key[8]  = 0xab;  Key[9]  = 0xf7;  Key[10] = 0x15;  Key[11] = 0x88;

   Key[12] = 0x09;  Key[13] = 0xcf;  Key[14] = 0x4f;  Key[15] = 0x3c;

   // Get the input string

   char str[1024];

   fgets(str, 1024, stdin);

   // The KeyExpansion routine is called before encryption.

   KeyExpansion();

   // sz is the cursor into the input string

   int sz=0;

   // Each iteration encrypts one block = Nb\*4 bytes = 128 bits in this case

   while (sz < strlen(str)) {

      // Fill the array 'in' with the next plaintext block

      sz = fillBlock (sz, str, in);

      // The block is encrypted here - the result is in the array 'out'

      Cipher();

      // Output the encrypted block.

      cout<<"Encrypted text: ";

      for (i=0 ; i < Nb\*4 ; i++) cout << (int)out[i] << " ";

   }

   printf("\n\n");

}

**Conclusion:**

The experiment is performed by writing a code for encryption & decryption and tested.