**Final Year B. Tech., Sem VII 2022-23**

**Cryptography And Network Security Lab**

**PRN No: 2019BTECS00038**

**Full name: Sadaf Najeem Mulla**

**Batch: B2**

**Assignment: 7**

**Title of assignment: Implementation of AES – Advanced Encryption Standard**

**Title:**

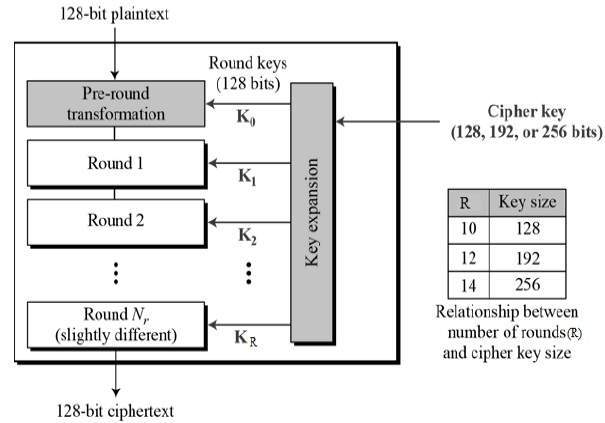
Implementation of Advanced Encryption Standard

**Aim:**

To develop and implement the Advanced Encryption Standard and to do encryption and decryption on the input plaintext

**Theory:**

* AES is an iterative rather than Feistel cipher. It is based on ‘substitution–permutation network’.
* Comprises of a series of linked operations, some of which involve replacing inputs by specific outputs (substitutions) and others involve shuffling bits around (permutations).
* AES performs all its computations on bytes rather than bits. Hence, AES treats the 128 bits of a plaintext block as 16 bytes. These 16 bytes are arranged in four columns and four rows for processing as a matrix
* the number of rounds in AES is variable and depends on the length of the key.
* AES uses 10 rounds for 128-bit keys, 12 rounds for 192-bit keys and 14 rounds for 256-bit keys. Each of these rounds uses a different 128-bit round key



* The features of AES are as follows
* Symmetric key symmetric block cipher
* 128-bit data, 128/192/256-bit keys
* Stronger and faster than Triple-DES
* Provide full specification and design details
* Software implementable in C and Java

**Implementation of Advanced Encryption Standard**

**Code:**

/\* encrypt.cpp

\* Performs encryption using AES 128-bit

\*/

#include <iostream>

#include <cstring>

#include <fstream>

#include <sstream>

#include "structures.h"

using namespace std;

/\* Serves as the initial round during encryption

\* AddRoundKey is simply an XOR of a 128-bit block with the 128-bit key.

\*/

void AddRoundKey(unsigned char \* state, unsigned char \* roundKey) {

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

state[i] ^= roundKey[i];

}

}

/\* Perform substitution to each of the 16 bytes

\* Uses S-box as lookup table

\*/

void SubBytes(unsigned char \* state) {

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

state[i] = s[state[i]];

}

}

// Shift left, adds diffusion

void ShiftRows(unsigned char \* state) {

unsigned char tmp[16];

/\* Column 1 \*/

tmp[0] = state[0];

tmp[1] = state[5];

tmp[2] = state[10];

tmp[3] = state[15];

/\* Column 2 \*/

tmp[4] = state[4];

tmp[5] = state[9];

tmp[6] = state[14];

tmp[7] = state[3];

/\* Column 3 \*/

tmp[8] = state[8];

tmp[9] = state[13];

tmp[10] = state[2];

tmp[11] = state[7];

/\* Column 4 \*/

tmp[12] = state[12];

tmp[13] = state[1];

tmp[14] = state[6];

tmp[15] = state[11];

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

state[i] = tmp[i];

}

}

/\* MixColumns uses mul2, mul3 look-up tables

\* Source of diffusion

\*/

void MixColumns(unsigned char \* state) {

unsigned char tmp[16];

tmp[0] = (unsigned char) mul2[state[0]] ^ mul3[state[1]] ^ state[2] ^ state[3];

tmp[1] = (unsigned char) state[0] ^ mul2[state[1]] ^ mul3[state[2]] ^ state[3];

tmp[2] = (unsigned char) state[0] ^ state[1] ^ mul2[state[2]] ^ mul3[state[3]];

tmp[3] = (unsigned char) mul3[state[0]] ^ state[1] ^ state[2] ^ mul2[state[3]];

tmp[4] = (unsigned char)mul2[state[4]] ^ mul3[state[5]] ^ state[6] ^ state[7];

tmp[5] = (unsigned char)state[4] ^ mul2[state[5]] ^ mul3[state[6]] ^ state[7];

tmp[6] = (unsigned char)state[4] ^ state[5] ^ mul2[state[6]] ^ mul3[state[7]];

tmp[7] = (unsigned char)mul3[state[4]] ^ state[5] ^ state[6] ^ mul2[state[7]];

tmp[8] = (unsigned char)mul2[state[8]] ^ mul3[state[9]] ^ state[10] ^ state[11];

tmp[9] = (unsigned char)state[8] ^ mul2[state[9]] ^ mul3[state[10]] ^ state[11];

tmp[10] = (unsigned char)state[8] ^ state[9] ^ mul2[state[10]] ^ mul3[state[11]];

tmp[11] = (unsigned char)mul3[state[8]] ^ state[9] ^ state[10] ^ mul2[state[11]];

tmp[12] = (unsigned char)mul2[state[12]] ^ mul3[state[13]] ^ state[14] ^ state[15];

tmp[13] = (unsigned char)state[12] ^ mul2[state[13]] ^ mul3[state[14]] ^ state[15];

tmp[14] = (unsigned char)state[12] ^ state[13] ^ mul2[state[14]] ^ mul3[state[15]];

tmp[15] = (unsigned char)mul3[state[12]] ^ state[13] ^ state[14] ^ mul2[state[15]];

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

state[i] = tmp[i];

}

}

/\* Each round operates on 128 bits at a time

\* The number of rounds is defined in AESEncrypt()

\*/

void Round(unsigned char \* state, unsigned char \* key) {

SubBytes(state);

ShiftRows(state);

MixColumns(state);

AddRoundKey(state, key);

}

// Same as Round() except it doesn't mix columns

void FinalRound(unsigned char \* state, unsigned char \* key) {

SubBytes(state);

ShiftRows(state);

AddRoundKey(state, key);

}

/\* The AES encryption function

\* Organizes the confusion and diffusion steps into one function

\*/

void AESEncrypt(unsigned char \* message, unsigned char \* expandedKey, unsigned char \* encryptedMessage) {

unsigned char state[16]; // Stores the first 16 bytes of original message

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

state[i] = message[i];

}

int numberOfRounds = 9;

AddRoundKey(state, expandedKey); // Initial round

for (int i = 0; i < numberOfRounds; i++) {

Round(state, expandedKey + (16 \* (i+1)));

}

FinalRound(state, expandedKey + 160);

// Copy encrypted state to buffer

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

encryptedMessage[i] = state[i];

}

}

int main() {

cout << "=============================" << endl;

cout << " 128-bit AES Encryption Tool " << endl;

cout << "=============================" << endl;

char message[1024];

cout << "Enter the message to encrypt: ";

cin.getline(message, sizeof(message));

cout << message << endl;

// Pad message to 16 bytes

int originalLen = strlen((const char \*)message);

int paddedMessageLen = originalLen;

if ((paddedMessageLen % 16) != 0) {

paddedMessageLen = (paddedMessageLen / 16 + 1) \* 16;

}

unsigned char \* paddedMessage = new unsigned char[paddedMessageLen];

for (int i = 0; i < paddedMessageLen; i++) {

if (i >= originalLen) {

paddedMessage[i] = 0;

}

else {

paddedMessage[i] = message[i];

}

}

unsigned char \* encryptedMessage = new unsigned char[paddedMessageLen];

string str;

ifstream infile;

infile.open("keyfile", ios::in | ios::binary);

if (infile.is\_open())

{

getline(infile, str); // The first line of file should be the key

infile.close();

}

else cout << "Unable to open file";

istringstream hex\_chars\_stream(str);

unsigned char key[16];

int i = 0;

unsigned int c;

while (hex\_chars\_stream >> hex >> c)

{

key[i] = c;

i++;

}

unsigned char expandedKey[176];

KeyExpansion(key, expandedKey);

for (int i = 0; i < paddedMessageLen; i += 16) {

AESEncrypt(paddedMessage+i, expandedKey, encryptedMessage+i);

}

cout << "Encrypted message in hex:" << endl;

for (int i = 0; i < paddedMessageLen; i++) {

cout << hex << (int) encryptedMessage[i];

cout << " ";

}

cout << endl;

// Write the encrypted string out to file "message.aes"

ofstream outfile;

outfile.open("message.aes", ios::out | ios::binary);

if (outfile.is\_open())

{

outfile << encryptedMessage;

outfile.close();

cout << "Wrote encrypted message to file message.aes" << endl;

}

else cout << "Unable to open file";

// Free memory

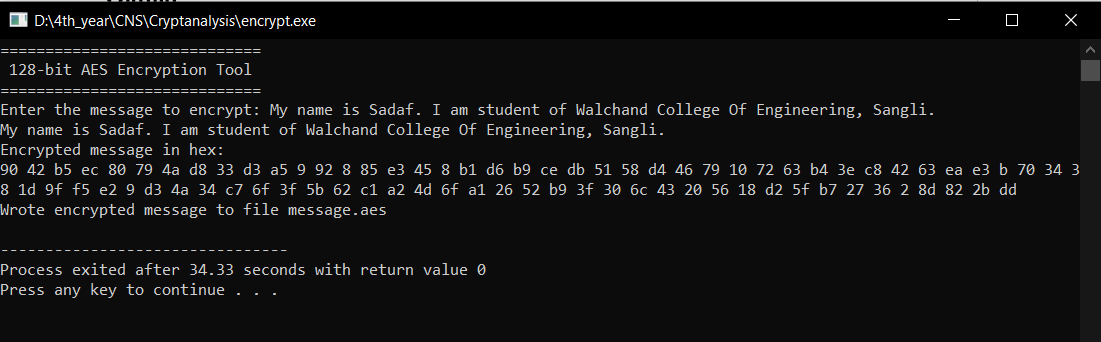
delete[] paddedMessage;

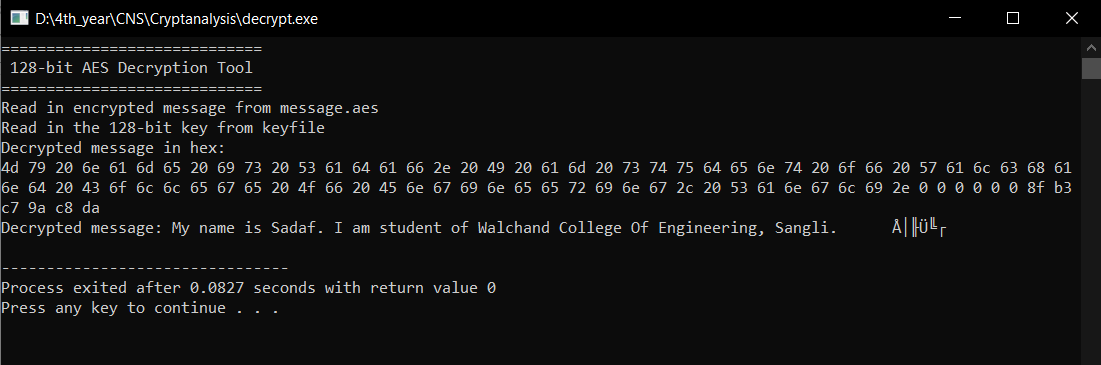
delete[] encryptedMessage;

return 0;

}

**Output:**





**Conclusion:**

Performed the experiment successfully. Encrypted the data

with the provided key. Output of this encryption is decrypted to match

the plaintext that was inputted by the user as shown in the above

diagram.