SECURE IMAGE TRANSMISSION OVER A TCP IP NETWORK

(Communication Networks Project)

Chintalwar Sankalp,
Department of Electronics and Communication Engineering,
National Institute of technology,
Warangal, India,
chintalwarsankalp@gmail.com

Abstract

Over the past few years, Images/Videos(pictorial data) have overtaken the text format significantly. As a result of technological advancements, the usage of such formats have become more widespread, easier and powerful in day to day life. Thus there arises a need for these technologies to secure the data and also keep in view the privacy of end users. With data protection becoming such an integral aim to keep sensitive information out of the wrong hands, only proper data security and privacy measures will prevent data leakage while still ensuring that a company runs smoothly. Thus data manipulation provides a larger scope in this scenario. This project looks to secure transmit images over a TCP IP network, with necessary encryption to the images and effective transfer of the Data, using Socket programming in C/C++. By encrypting the Image at the transmitting end using AES algorithm, transmitting it over a TCP IP network, decrypting the image at receiving end, the functionality of this project is deemed to be attained.

Introduction

A network socket is an internal endpoint for sending or receiving data within a node on a computer network . The term socket is analogous to physical female connectors , communication between two nodes through a channel being visualized as a cable with two male connectors plugging into sockets at each node. Similarly, the term port (another term for a female connector) is used for external endpoints at a node, and the term socket is also used for an internal endpoint of local inter-process communication (IPC) (not over a network). However, the analogy is strained, as network communication need not be one-to-one or have a dedicated communication channel .

The Advanced Encryption Standard (AES), also known by its original name Rijndael is a specification for the encryption of electronic data established by the U.S. National Institute of Standards and Technology (NIST) in 2001 . AES is based on a design principle known as a substitution – permutation network, and is efficient in both software and hardware. AES is a variant of Rijndael which has a fixed block size of 128 bits, and a key size of 128, 192, or 256 bits.

Implementation

Overview of the processes to implement this project can be stated as

- To set up the TCP IP connection between the two nodes.
- Encryption of the selected image: The image file is converted into bitmap, the bitmap is then processed using AES using the key, thus giving back the encrypted image.
- Sending the packets of data over the network.
- Decryption of the Image using the key, which is only available to end users, and then converting the decrypted Bitmap into Image type.

Algorithm for Encryption

First the image Pixel Data is put into the array, then encrypted.

Encryption transforms data to an unintelligent form defined as cipher-text. - A repetition is called a round and the number of iterations of a loop is Nr. Value of Nr is 10, 12 or 14 for key size of 128, 192 or 256 bits length respectively. -

Encryption consists of four transformations:

1. SubBytes transformation.
2. Shift rows transformation.
3. MixColumns transformation.
4. Add-round key.

The AES algorithm operates on a 128-bit block of data and symmetrically executed (Nr – 1) times. The first and last rounds are different from other rounds: an additional AddRoundKey transformation at the beginning of the first round and no MixColumns transformation are performed in the last round. - Key length of 128 bits (AES- 128) for system generation is implemented here. A set of 10 keys is formulated for ten rounds each. - The description of each operation is provided below. The first step is conversion of 128 bit into a state which is a matrix of 4 x 4 matrix, each element being a byte (division of 128-bits into 16-bytes = S0-15)

$$\begin{bmatrix} S_0 & S_4 & S_8 & S_{12} \\ S_1 & S_5 & S_9 & S_{13} \\ S_2 & S_6 & S_{10} & S_{14} \\ S_3 & S_7 & S_{11} & S_{15} \end{bmatrix}$$

Fig. State Matrix- S.

.

SubBytes transformation: This transformation involves substitution of each of the bytes independently. A pre-calculated substitution table called S-box is used for swapping values depending on byte location. The S-box table tabulates a set of 256 numbers (from 0 to 255).The S-box is a permutation of each of the 256 numbers. Fig below shows the S-Box.

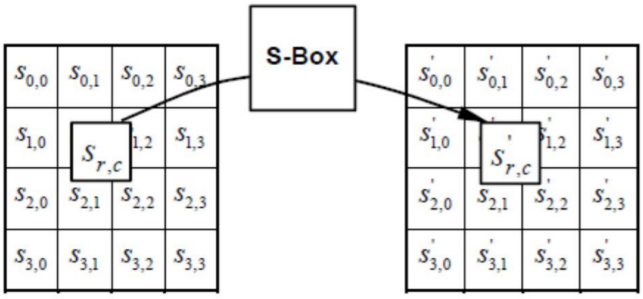


Fig. SubBytes Transformation.

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
a	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
b	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
c	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
d	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
e	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
f	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Fig. S-Box

Shift Rows transformation:

For this transformation, the rows of the state are episodically shifted by one. Leaving the first row as it is, second row is given a one byte left shift, third row is subject to two bytes left shift and subsequently fourth row is three bytes to the left shifted.

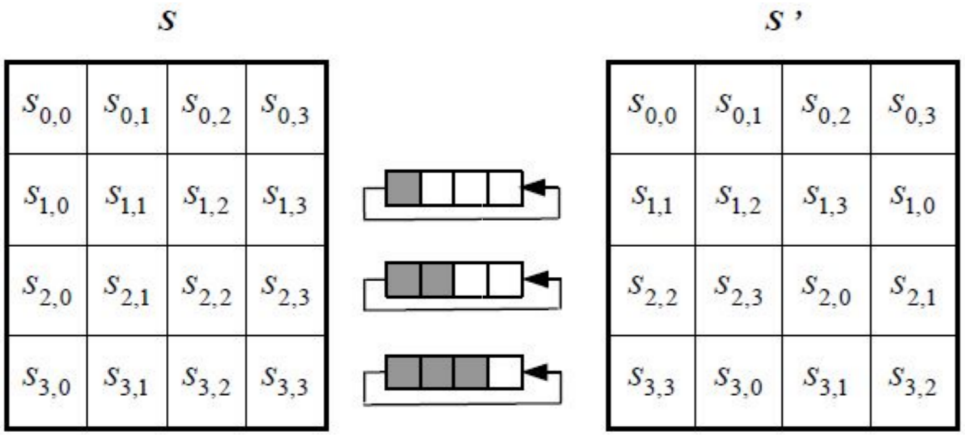


Fig. Shift Rows Transformation.

MixColumns transformation:

In MixColumns transformation, assuming the columns of the state as complex polynomial above GF (28) into multiplication of modulo x4 + 1 with a preset polynomial c(x), given by: c(x) = {03} x3+ {01}x2 + {01}x + {02}. The below figure provides the mathematical equivalent of this transformation.

$$\begin{bmatrix} S'_{0,c} \\ S'_{1,c} \\ S'_{2,c} \\ S'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 03 & 01 & 01 & 02 \\ 01 & 01 & 02 & 03 \\ 01 & 02 & 03 & 01 \end{bmatrix} \begin{bmatrix} S_{0,c} \\ S_{1,c} \\ S_{2,c} \\ S_{3,c} \end{bmatrix} \quad \text{..for c=>0 to 3}$$

Fig. Math. Eq. of MixColumns Trans.

ALGORITHM FOR KEY EXPANSION AND ADDITION:

The AddRoundKey Transformation and RoundKeys generated in Key Expansion are commonly used in both Encryption and Decryption. So they are defined separately.

AddRoundKey Transformation:

The AddRoundKey transformation includes round addition of a key termed RoundKey to the resultant state from the MixColumns alteration maneuver, implemented by a modest XOR operation bit by bit. The RoundKey to be used in each round is a derivative from the main key by means of the KeyExpansion set of rules. The entire encryption/ decryption process desires ten 128-bit RoundKey, designated as RoundKey1 to RoundKey10, which are generated in RoundKey Generation process.

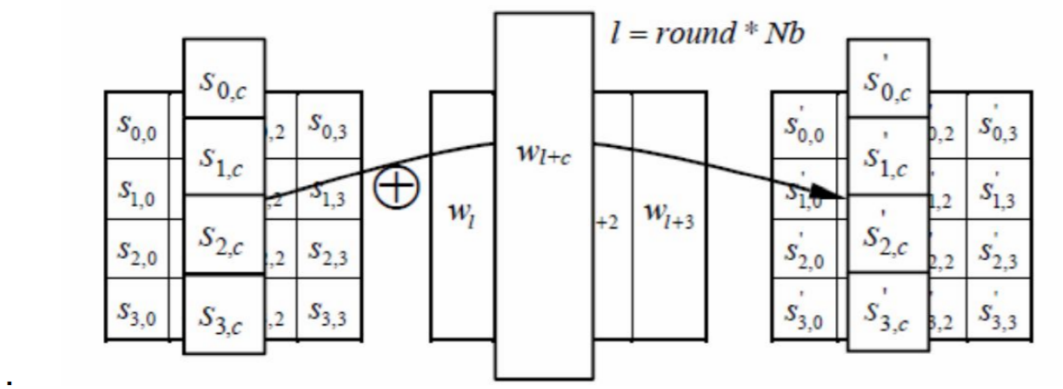


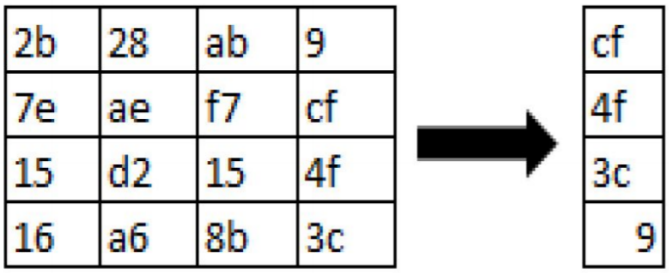
Fig. AddRoundKey Transformation.

Key Expansion:

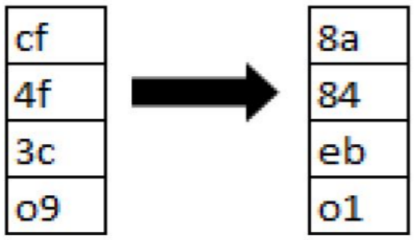
The spawning maneuver to derive wholly from the original secret input key, the RoundKeys to be utilized in every round is key expansion. The first RoundKey is original key in encryption. In decryption, the last of the generated key by key expansion will be the original key. As described before the secret round key will be second input to the plain text input before iterative steps of

the encryption or decryption commence. Elucidated below are the steps for generating 10 rounds of the round key:

Interchange the elements of 4th column of the (i-1) key in such a way that each element is shifted up by one row. This is shown below with example.



The values of the column are then substituted by SBox table values similar to SubBytes Transformation round as seen by eg. below.



The transformed byte matrix is given XOR operation with Row constant designated by Rcon. Rcon is reliant on on each round.

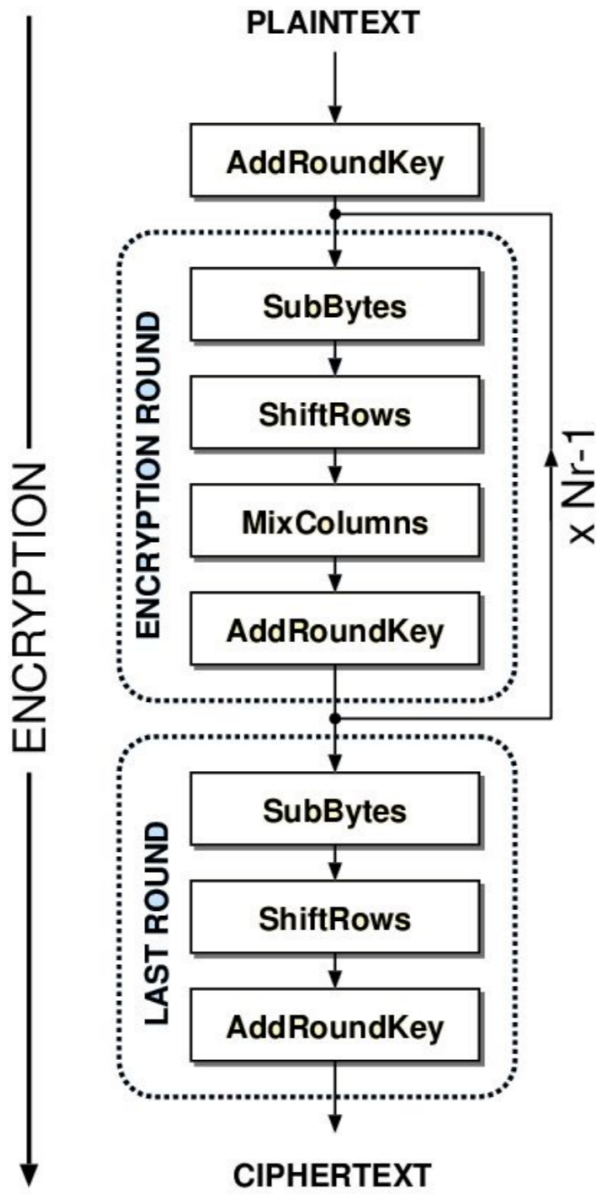
01	02	04	08	10	20	40	80	1b	36
00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	00	00	00	00

Rcon

Repeat the same for the remaing columns (2) .This will produce the complete key for ith round.

This whole progression takes place repeatedly for creating each of the 10 keys. The keys need to be stockpiled as and when computed as the (10-i)th round of decryption is equivalent to i'th round key of encryption.

FLOWCHART FOR ENCRYPTION:



ALGORITHM FOR DECRYPTION:

Decryption is a converse procedure of encryption .It is employed to decode the original plaintext out of an encrypted cipher-text. The key is the same or symmetric to the key used in encryption, but they are used in reverse order i.e. RoundKey10 to Original key from Round1 to Round10. The substitution box is the Inverse S box. The basic test of software or VHDL is that if the cipher data obtained from encryption is converted back to original the software can be said to be validated. There is an inverse relation to encrytion. Hence the round conversion of decoding algorithm deploys the tasks designated as AddRoundKey, InvMixColumns, InvShiftRows, and InvSubBytes respectively. The description of each step is given below.

InvMixColumn Transformation:

InvMixColumns transformation by considering columns a polynomial over GF (28) coefficients elements of which are the columns of the state are multiplied modulo (x4 + 1) by a fixed polynomial d(x) ={0b}x3 + {0d}x2 + {09}x + {0e}, where {0b}, {0d}; {09}, {0e} represent hexadecimal format. Fig. 10 shows InvMixColumns Transformation.

$$\begin{bmatrix} S'_{0,c} \\ S'_{1,c} \\ S'_{2,c} \\ S'_{3,c} \end{bmatrix} = \begin{bmatrix} 0E & 0B & 0D & 09 \\ 0B & 0D & 09 & 0E \\ 0D & 09 & 0E & 0B \\ 09 & 0E & 0B & 0D \end{bmatrix} \begin{bmatrix} S_{0,c} \\ S_{1,c} \\ S_{2,c} \\ S_{3,c} \end{bmatrix} \quad \text{..for c=>0 to 3}$$

InvSubBytes Transformation:

In InvSubBytes Transformation byte values of the state are replaced from a pre-calculated substitution table called InvS-box. This InvS-box table contains a permutation of each of 256 numbers (from 0 to 255) as shown below. Depending on the Byte location corresponding value is replaced. Fig. 11 shows InvSubBytes Transformation.

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	52	09	6A	D5	30	36	A5	38	BF	40	A3	9E	81	F3	D7	FB
1	7C	E3	39	82	9B	2F	FF	87	34	83	43	44	C4	DE	E9	CB
2	54	7B	94	32	A6	C2	23	3D	EE	4C	95	0B	42	FA	C3	4E
3	08	2E	A1	66	28	D9	24	B2	76	5B	A2	49	6D	8B	D1	25
4	72	F8	F6	64	86	68	98	16	D4	A4	5C	CC	5D	65	B6	92
5	6C	70	48	50	FD	ED	B9	DA	5E	15	46	57	A7	8D	9D	84
6	90	D8	AB	00	8C	BC	D3	0A	F7	E4	58	05	B8	B3	45	06
7	D0	2C	1E	8F	CA	3F	0F	02	C1	AF	BD	03	01	13	8A	6B
8	3A	91	11	41	4F	67	DC	EA	97	F2	CF	CE	F0	B4	E6	73
9	96	AC	74	22	E7	AD	35	85	E2	F9	37	E8	1C	75	DF	6E
a	47	F1	1A	71	1D	29	C5	89	6F	B7	62	0E	AA	18	BE	1B
b	FC	56	3E	4B	C6	D2	79	20	9A	DB	C0	FE	78	CD	5A	F4
c	1F	DD	A8	33	88	07	C7	31	B1	12	10	59	27	80	EC	5F
d	60	51	7F	A9	19	B5	4A	0D	2D	E5	7A	9F	93	C9	9C	EF
e	A0	E0	3B	4D	AE	2A	F5	B0	C8	EB	BB	3C	83	53	99	61
f	17	2B	04	7E	BA	77	D6	26	E1	69	14	63	55	21	0C	7D

Fig. Inverse S-Box.

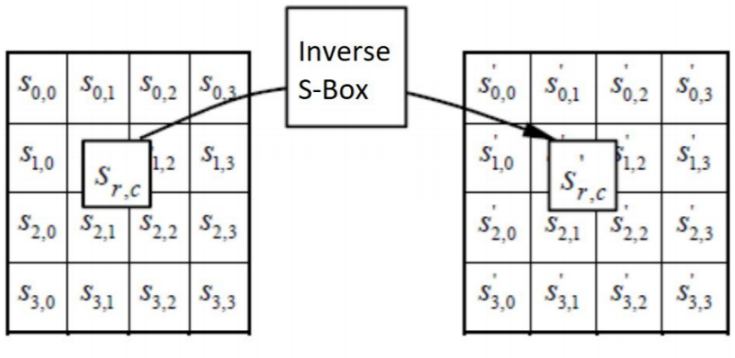


Fig. Inverse SubBytes Transformation.

InvShiftRows Transformation:

InvShiftRows operates similar to ShiftRows process though in the differing direction. Leaving the first row as it is, second row is given a one byte right shift, third row is subject to two bytes right shift and subsequently fourth row is three bytes to the right shifted.

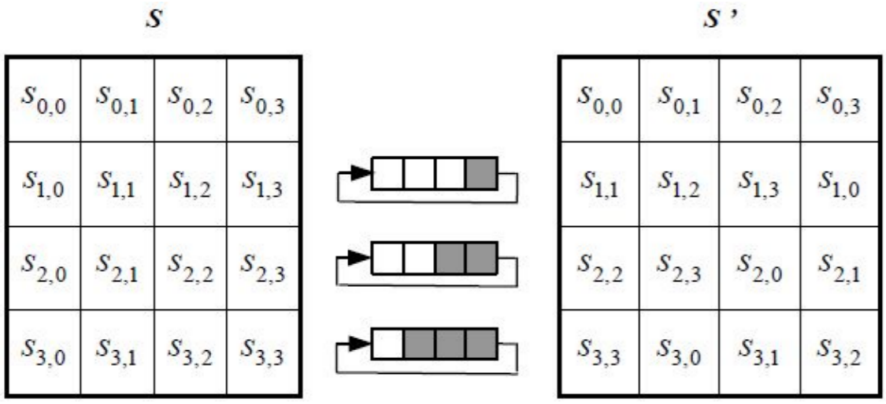
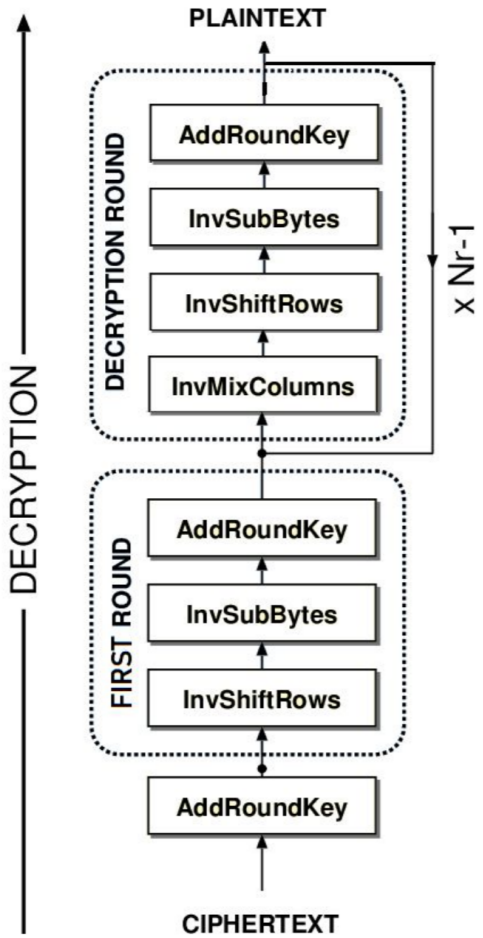


Fig. Inverse Shift Rows Transformation.

FLOWCHART FOR DECRYPTION:



Implementation of sendData() function:

This function uses the <socket.h> library to bind the sockets to the server given in the program and any connection from other computer gets the socket transferred, thus here sending the Image. The one sending the data acts like the server.

```
void senddata(){
    int sfd = socket(AF_INET,SOCK_STREAM,0);
    if(sfd<0)perror("0");
    struct sockaddr_in addr;
    int addrlen = sizeof(addr);
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = inet_addr("127.0.0.7");
    addr.sin_port = htons(port);
    if(bind(sfd,(struct sockaddr *)&addr,sizeof(addr))<0)
        perror("1");
    listen(sfd,3);
    int nsfd;
    nsfd = accept(sfd,(struct sockaddr *)&addr,(socklen_t*)&addrlen);
    if(nsfd<0){
        perror("2");
    }
    for(int i=0;i<10;i++)cout<<myvect[i]<<" ";
        cout<<endl;

    int cnt = 0;
    int size = myvect.size();
    send(nsfd,&size,sizeof(size),0);
    send(nsfd,&rows,sizeof(rows),0);
    send(nsfd,&columns,sizeof(columns),0);
    while(cnt<myvect.size()){
        int a[5000]={0};
        int d = 0;
        for(int i=0;i<5000;i++){
            if(cnt>=myvect.size()){i=50005;continue;}
            a[i] = myvect[cnt++];
        }
        send(nsfd,a,sizeof(a),0);
    }
    cout<<myvect.size()<<endl;
}
```

Implementation of getData() function:

This function connects to the provided IP, though the data transfer can be bi-directional, we implement only read here.

```
void getdata(char* arg){
    int sfd = socket(AF_INET,SOCK_STREAM,0);
    struct sockaddr_in addr;

    addr.sin_family = AF_INET;
    addr.sin_port = htons(port);

    addr.sin_addr.s_addr = inet_addr(arg);
    connect(sfd, (struct sockaddr *)&addr, sizeof(addr));

    int size = 0;
    read(sfd,&size,sizeof(size));
    read(sfd,&rows,sizeof(rows));
    read(sfd,&columns,sizeof(columns));
    int cnt = ceil((double)size/5000);int l = 0;
    while(cnt--){
        int a[5000]={0};
        read(sfd,a,sizeof(a));
        for(int i=0;i<5000;i++){
            if(l>=size){i=5005;continue;}
            myvect.push_back(a[i]);
            l++;
        }
    }
}
```

Complete program of Sender:

```
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/core/core.hpp>
#include <opencv2/opencv.hpp>
#include <string.h>
#include <iostream>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <string.h>
#include <arpa/inet.h>
#include <errno.h>
#include <netdb.h>
#include <ifaddrs.h>
#define port 8182

using namespace cv;
using namespace std;
//char ip = "172.30.101.142";
Vec3b buf;



unsigned char sbox[256] = {
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
};

unsigned char mul2[] =
{
        0x00,0x02,0x04,0x06,0x08,0x0a,0x0c,0x0e,0x10,0x12,0x14,0x16,0x18,0x1a,0x1c,0x1e,
```

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

unsigned char mul3[] =
{
	0x00,0x03,0x06,0x05,0x0c,0x0f,0x0a,0x09,0x18,0x1b,0x1e,0x1d,0x14,0x17,0x12,0x11,
	0x30,0x33,0x36,0x35,0x3c,0x3f,0x3a,0x39,0x28,0x2b,0x2e,0x2d,0x24,0x27,0x22,0x21,
	0x60,0x63,0x66,0x65,0x6c,0x6f,0x6a,0x69,0x78,0x7b,0x7e,0x7d,0x74,0x77,0x72,0x71,
	0x50,0x53,0x56,0x55,0x5c,0x5f,0x5a,0x59,0x48,0x4b,0x4e,0x4d,0x44,0x47,0x42,0x41,
	0xc0,0xc3,0xc6,0xc5,0xcc,0xcf,0xca,0xc9,0xd8,0xdb,0xde,0xdd,0xd4,0xd7,0xd2,0xd1,
	0xf0,0xf3,0xf6,0xf5,0xfc,0xff,0xfa,0xf9,0xe8,0xeb,0xee,0xed,0xe4,0xe7,0xe2,0xe1,
	0xa0,0xa3,0xa6,0xa5,0xac,0xaf,0xaa,0xa9,0xb8,0xbb,0xbe,0xbd,0xb4,0xb7,0xb2,0xb1,
	0x90,0x93,0x96,0x95,0x9c,0x9f,0x9a,0x99,0x88,0x8b,0x8e,0x8d,0x84,0x87,0x82,0x81,
	0x9b,0x98,0x9d,0x9e,0x97,0x94,0x91,0x92,0x83,0x80,0x85,0x86,0x8f,0x8c,0x89,0x8a,
	0xab,0xa8,0xad,0xae,0xa7,0xa4,0xa1,0xa2,0xb3,0xb0,0xb5,0xb6,0xbf,0xbc,0xb9,0xba,
	0xfb,0xf8,0xfd,0xfe,0xf7,0xf4,0xf1,0xf2,0xe3,0xe0,0xe5,0xe6,0xef,0xec,0xe9,0xea,
	0xcb,0xc8,0xcd,0xce,0xc7,0xc4,0xc1,0xc2,0xd3,0xd0,0xd5,0xd6,0xdf,0xdc,0xd9,0xda,
	0x5b,0x58,0x5d,0x5e,0x57,0x54,0x51,0x52,0x43,0x40,0x45,0x46,0x4f,0x4c,0x49,0x4a,
	0x6b,0x68,0x6d,0x6e,0x67,0x64,0x61,0x62,0x73,0x70,0x75,0x76,0x7f,0x7c,0x79,0x7a,
```

```
        0x3b,0x38,0x3d,0x3e,0x37,0x34,0x31,0x32,0x23,0x20,0x25,0x26,0x2f,0x2c,0x29,0x2a,
        0x0b,0x08,0x0d,0x0e,0x07,0x04,0x01,0x02,0x13,0x10,0x15,0x16,0x1f,0x1c,0x19,0x1a
};

unsigned char mainkey[16];
unsigned char roundkeys[10][16];
unsigned char rconkeys[10] = { 0x01, 0x02,0x04,0x08,0x10,0x20,0x40,0x80,0x1b,0x36 };

unsigned char keytemp1[16], keytemp2[4], tmp[16];

void roundkeyexpansion(int roundnum)
{
        int temp = keytemp1[3];
        keytemp1[3] = keytemp1[7];
        keytemp1[7] = keytemp1[11];
        keytemp1[11] = keytemp1[15];
        keytemp1[15] = temp;
        for (int i = 3; i < 16; i = i + 4) {
                keytemp1[i] = sbox[keytemp1[i]];
        }

        keytemp1[3] ^= rconkeys[roundnum];

        for (int i = 0; i < 13; i = i + 4) {
                keytemp1[i] = keytemp1[i + 3] xor keytemp1[i];
        }

        for (int i = 0; i < 13; i = i + 4) {
                keytemp1[i + 1] = keytemp1[i + 1] xor keytemp1[i];
        }
        for (int i = 0; i < 13; i = i + 4) {
                keytemp1[i + 2] ^= keytemp1[i + 1];
        }
```

```
	for (int i = 0; i < 4; i = i + 1) {
		keytemp1[(4 * i) + 3] = keytemp1[(4 * i) + 2] xor keytemp2[i];
	}
}

void assignkey() {
	keytemp1[0] = 0x54;
	keytemp1[4] = 0x68;
	keytemp1[8] = 0x61;
	keytemp1[12] = 0x74;
	keytemp1[1] = 0x73;
	keytemp1[5] = 0x20;
	keytemp1[9] = 0x6d;
	keytemp1[13] = 0x79;
	keytemp1[2] = 0x20;
	keytemp1[6] = 0x4b;
	keytemp1[10] = 0x75;
	keytemp1[14] = 0x6e;
	keytemp1[3] = 0x67;
	keytemp1[7] = 0x20;
	keytemp1[11] = 0x46;
	keytemp1[15] = 0x75;
	for (int i = 0; i <= 3; i++)
	{
		keytemp2[i] = keytemp1[4 * i + 3];
	}
}


void assignkey2(int round)
{
	for (int i = 0; i < 4; i = i + 1) {
		for (int j = 0; j < 16; j = j + 4)
```

```
                {
                        keytemp1[j + i] = roundkeys[round - 1][j + i];
                }
        }
        for (int i = 0; i <= 3; i++)
        {
                keytemp2[i] = keytemp1[4 * i + 3];
        }
}


void addKey(int roundnum) {
        for (int i = 0; i < 4; i = i + 1) {
                for (int j = 0; j < 16; j = j + 4)
                {
                        roundkeys[roundnum][j + i] = keytemp1[j + i];
                }
        }
}



void roundkeygen() {
        for (int i = 0; i < 10; i++)
        {
                if (i == 0)
                        assignkey();
                else
                        assignkey2(i);
                roundkeyexpansion(i);
                addKey(i);
        }
}
```

```
int plaintext[16] = {};


void subBytes() {
	for (int i = 0; i < 16; i++)
	{
		plaintext[i] = sbox[plaintext[i]];
	}
}

void shiftRowOnce(int k) {
	int temp = plaintext[k];
	for (int i = k; i < k + 3; i++)
	{
		plaintext[i] = plaintext[i + 1];
	}
	plaintext[k + 3] = temp;
}

void shiftRows() {
	shiftRowOnce(4);
	shiftRowOnce(8); shiftRowOnce(8);
	shiftRowOnce(12); shiftRowOnce(12); shiftRowOnce(12);
}




void mixColumns()
{
```

```
        tmp[0] = (unsigned char)mul2[plaintext[0]] ^ mul3[plaintext[4]] ^ plaintext[8] ^
plaintext[12];
        tmp[4] = (unsigned char)plaintext[0] ^ mul2[plaintext[4]] ^ mul3[plaintext[8]] ^
plaintext[12];
        tmp[8] = (unsigned char)plaintext[0] ^ plaintext[4] ^ mul2[plaintext[8]] ^
mul3[plaintext[12]];
        tmp[12] = (unsigned char)mul3[plaintext[0]] ^ plaintext[4] ^ plaintext[8] ^
mul2[plaintext[12]];

        tmp[1] = (unsigned char)mul2[plaintext[1]] ^ mul3[plaintext[5]] ^ plaintext[9] ^
plaintext[13];
        tmp[5] = (unsigned char)plaintext[1] ^ mul2[plaintext[5]] ^ mul3[plaintext[9]] ^
plaintext[13];
        tmp[9] = (unsigned char)plaintext[1] ^ plaintext[5] ^ mul2[plaintext[9]] ^
mul3[plaintext[13]];
        tmp[13] = (unsigned char)mul3[plaintext[1]] ^ plaintext[5] ^ plaintext[9] ^
mul2[plaintext[13]];

        tmp[2] = (unsigned char)mul2[plaintext[2]] ^ mul3[plaintext[6]] ^ plaintext[10] ^
plaintext[14];
        tmp[6] = (unsigned char)plaintext[2] ^ mul2[plaintext[6]] ^ mul3[plaintext[10]] ^
plaintext[14];
        tmp[10] = (unsigned char)plaintext[2] ^ plaintext[6] ^ mul2[plaintext[10]] ^
mul3[plaintext[14]];
        tmp[14] = (unsigned char)mul3[plaintext[2]] ^ plaintext[6] ^ plaintext[10] ^
mul2[plaintext[14]];

        tmp[3] = (unsigned char)mul2[plaintext[3]] ^ mul3[plaintext[7]] ^ plaintext[11] ^
plaintext[15];
        tmp[7] = (unsigned char)plaintext[3] ^ mul2[plaintext[7]] ^ mul3[plaintext[11]] ^
plaintext[15];
        tmp[11] = (unsigned char)plaintext[3] ^ plaintext[7] ^ mul2[plaintext[11]] ^
mul3[plaintext[15]];
```

```
	tmp[15] = (unsigned char)mul3[plaintext[3]] ^ plaintext[7] ^ plaintext[11] ^
mul2[plaintext[15]];

	for (int i = 0; i < 16; i++)
		plaintext[i] = tmp[i];
}

void addroundkey(int round) {
	if (round == 0) {
		for (int i = 0; i < 16; i++) {
			plaintext[i] ^= mainkey[i];
		}

	}
	else
	{
		for (int i = 0; i < 16; i++) {
			plaintext[i] ^= roundkeys[round - 1][i];
		}
	}
}
void initialise() {

	mainkey[0] = 0x54;
	mainkey[4] = 0x68;
	mainkey[8] = 0x61;
	mainkey[12] = 0x74;
	mainkey[1] = 0x73;
	mainkey[5] = 0x20;
	mainkey[9] = 0x6d;
	mainkey[13] = 0x79;
	mainkey[2] = 0x20;
	mainkey[6] = 0x4b;
```

```
	mainkey[10] = 0x75;
	mainkey[14] = 0x6e;
	mainkey[3] = 0x67;
	mainkey[7] = 0x20;
	mainkey[11] = 0x46;
	mainkey[15] = 0x75;
}

void aes_encrypt() {
	addroundkey(0);

	subBytes();
	shiftRows();
	mixColumns();
	addroundkey(1);

	subBytes();
	shiftRows();
	mixColumns();
	addroundkey(2);

	subBytes();
	shiftRows();
	mixColumns();
	addroundkey(3);

	subBytes();
	shiftRows();
	mixColumns();
	addroundkey(4);

	subBytes();
	shiftRows();
```

```
        mixColumns();
        addroundkey(5);

        subBytes();
        shiftRows();
        mixColumns();
        addroundkey(6);

        subBytes();
        shiftRows();
        mixColumns();
        addroundkey(7);

        subBytes();
        shiftRows();
        mixColumns();
        addroundkey(8);

        subBytes();
        shiftRows();
        mixColumns();
        addroundkey(9);

        subBytes();
        shiftRows();
        addroundkey(10);
}



vector<int> myvect;

int rows, columns;
```

```
void senddata(){
	int sfd = socket(AF_INET,SOCK_STREAM,0);
	if(sfd<0)perror("0");
	struct sockaddr_in addr;
	int addrlen = sizeof(addr);
	addr.sin_family = AF_INET;
	addr.sin_addr.s_addr = inet_addr("127.0.0.7");
	addr.sin_port = htons(port);
	if(bind(sfd,(struct sockaddr *)&addr,sizeof(addr))<0)
		perror("1");
	listen(sfd,3);
	int nsfd;
	nsfd = accept(sfd,(struct sockaddr *)&addr,(socklen_t*)&addrlen);
	if(nsfd<0){
		perror("2");
	}
	for(int i=0;i<10;i++)cout<<myvect[i]<<" ";
		cout<<endl;

	int cnt = 0;
	int size = myvect.size();
	send(nsfd,&size,sizeof(size),0);
	send(nsfd,&rows,sizeof(rows),0);
	send(nsfd,&columns,sizeof(columns),0);
	while(cnt<myvect.size()){
		int a[5000]={0};
		int d = 0;
		for(int i=0;i<5000;i++){
			if(cnt>=myvect.size()){i=50005;continue;}
			a[i] = myvect[cnt++];
		}
		send(nsfd,a,sizeof(a),0);
```

```
	}
	cout<<myvect.size()<<endl;
}

int main() {
	roundkeygen();
	Mat image = imread("image1.jpg");
	rows = image.rows;
	columns = image.cols;
	cout<<"Image Height - "<<rows<<"\nImage Width - "<<columns<<"\n";
	cout << "\n\nEncryption Started\n\n";
	for (int i = 0; i < image.rows; i++)
	{
		for (int j = 0; j < image.cols; j++) {
			buf = image.at<Vec3b>(i, j);
			myvect.push_back(buf[0]);
			myvect.push_back(buf[1]);
			myvect.push_back(buf[2]);

		}
	}
	if (myvect.size() % 4 == 0)
	{
		for (int i = 0; i < myvect.size() / 16; i++) {
			for (int j = i * 16; j < (i * 16) + 16; j++)
			{
				plaintext[j % 16] = myvect[j];
			}
			aes_encrypt();
			for (int j = i * 16; j < (i * 16) + 16; j++)
			{
				myvect[j] = plaintext[j % 16];
			}
```

```
            }
        }


        for (int i = 0; i < image.rows*image.cols; i = i + 1)
        {
            buf[0] = myvect[i*3];
            buf[1] = myvect[(i*3) + 1];
            buf[2] = myvect[(i * 3) + 2];

            image.at<Vec3b>(i / image.cols, i%image.cols) = buf;
        }


        cout << "\n\nEncryption Finished\n\n";
        namedWindow("Encrypted Image", WINDOW_GUI_NORMAL);
        imshow("Encrypted Image", image);
        senddata();

        waitKey(0);

}
```

Complete program of Receiver:

```
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/core/core.hpp>
#include <opencv2/opencv.hpp>
#include<stdio.h>
#include<unistd.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <string.h>
#include <arpa/inet.h>
#define port 8182
#include <bits/stdc++.h>
using namespace cv;
using namespace std;


unsigned char sbox[256] = {
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
};

// Decryption: Inverse Rijndael S-box
unsigned char invsbox[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
};

// Decryption: Multiply by 9 for InverseMixColumns
unsigned char mul9[256] =
{
	0x00,0x09,0x12,0x1b,0x24,0x2d,0x36,0x3f,0x48,0x41,0x5a,0x53,0x6c,0x65,0x7e,0x77,
	0x90,0x99,0x82,0x8b,0xb4,0xbd,0xa6,0xaf,0xd8,0xd1,0xca,0xc3,0xfc,0xf5,0xee,0xe7,
	0x3b,0x32,0x29,0x20,0x1f,0x16,0x0d,0x04,0x73,0x7a,0x61,0x68,0x57,0x5e,0x45,0x4c,
	0xab,0xa2,0xb9,0xb0,0x8f,0x86,0x9d,0x94,0xe3,0xea,0xf1,0xf8,0xc7,0xce,0xd5,0xdc,
	0x76,0x7f,0x64,0x6d,0x52,0x5b,0x40,0x49,0x3e,0x37,0x2c,0x25,0x1a,0x13,0x08,0x01,
	0xe6,0xef,0xf4,0xfd,0xc2,0xcb,0xd0,0xd9,0xae,0xa7,0xbc,0xb5,0x8a,0x83,0x98,0x91,
```

```
        0x4d,0x44,0x5f,0x56,0x69,0x60,0x7b,0x72,0x05,0x0c,0x17,0x1e,0x21,0x28,0x33,0x3a,
        0xdd,0xd4,0xcf,0xc6,0xf9,0xf0,0xeb,0xe2,0x95,0x9c,0x87,0x8e,0xb1,0xb8,0xa3,0xaa,
        0xec,0xe5,0xfe,0xf7,0xc8,0xc1,0xda,0xd3,0xa4,0xad,0xb6,0xbf,0x80,0x89,0x92,0x9b,
        0x7c,0x75,0x6e,0x67,0x58,0x51,0x4a,0x43,0x34,0x3d,0x26,0x2f,0x10,0x19,0x02,0x0b,
        0xd7,0xde,0xc5,0xcc,0xf3,0xfa,0xe1,0xe8,0x9f,0x96,0x8d,0x84,0xbb,0xb2,0xa9,0xa0,
        0x47,0x4e,0x55,0x5c,0x63,0x6a,0x71,0x78,0x0f,0x06,0x1d,0x14,0x2b,0x22,0x39,0x30,
        0x9a,0x93,0x88,0x81,0xbe,0xb7,0xac,0xa5,0xd2,0xdb,0xc0,0xc9,0xf6,0xff,0xe4,0xed,
        0x0a,0x03,0x18,0x11,0x2e,0x27,0x3c,0x35,0x42,0x4b,0x50,0x59,0x66,0x6f,0x74,0x7d,
        0xa1,0xa8,0xb3,0xba,0x85,0x8c,0x97,0x9e,0xe9,0xe0,0xfb,0xf2,0xcd,0xc4,0xdf,0xd6,
        0x31,0x38,0x23,0x2a,0x15,0x1c,0x07,0x0e,0x79,0x70,0x6b,0x62,0x5d,0x54,0x4f,0x46
};

// Decryption: Multiply by 11 for InverseMixColumns
unsigned char mul11[256] =
{
        0x00,0x0b,0x16,0x1d,0x2c,0x27,0x3a,0x31,0x58,0x53,0x4e,0x45,0x74,0x7f,0x62,0x69,
        0xb0,0xbb,0xa6,0xad,0x9c,0x97,0x8a,0x81,0xe8,0xe3,0xfe,0xf5,0xc4,0xcf,0xd2,0xd9,
        0x7b,0x70,0x6d,0x66,0x57,0x5c,0x41,0x4a,0x23,0x28,0x35,0x3e,0x0f,0x04,0x19,0x12,
        0xcb,0xc0,0xdd,0xd6,0xe7,0xec,0xf1,0xfa,0x93,0x98,0x85,0x8e,0xbf,0xb4,0xa9,0xa2,
        0xf6,0xfd,0xe0,0xeb,0xda,0xd1,0xcc,0xc7,0xae,0xa5,0xb8,0xb3,0x82,0x89,0x94,0x9f,
        0x46,0x4d,0x50,0x5b,0x6a,0x61,0x7c,0x77,0x1e,0x15,0x08,0x03,0x32,0x39,0x24,0x2f,
        0x8d,0x86,0x9b,0x90,0xa1,0xaa,0xb7,0xbc,0xd5,0xde,0xc3,0xc8,0xf9,0xf2,0xef,0xe4,
        0x3d,0x36,0x2b,0x20,0x11,0x1a,0x07,0x0c,0x65,0x6e,0x73,0x78,0x49,0x42,0x5f,0x54,
        0xf7,0xfc,0xe1,0xea,0xdb,0xd0,0xcd,0xc6,0xaf,0xa4,0xb9,0xb2,0x83,0x88,0x95,0x9e,
        0x47,0x4c,0x51,0x5a,0x6b,0x60,0x7d,0x76,0x1f,0x14,0x09,0x02,0x33,0x38,0x25,0x2e,
        0x8c,0x87,0x9a,0x91,0xa0,0xab,0xb6,0xbd,0xd4,0xdf,0xc2,0xc9,0xf8,0xf3,0xee,0xe5,
        0x3c,0x37,0x2a,0x21,0x10,0x1b,0x06,0x0d,0x64,0x6f,0x72,0x79,0x48,0x43,0x5e,0x55,
        0x01,0x0a,0x17,0x1c,0x2d,0x26,0x3b,0x30,0x59,0x52,0x4f,0x44,0x75,0x7e,0x63,0x68,
        0xb1,0xba,0xa7,0xac,0x9d,0x96,0x8b,0x80,0xe9,0xe2,0xff,0xf4,0xc5,0xce,0xd3,0xd8,
        0x7a,0x71,0x6c,0x67,0x56,0x5d,0x40,0x4b,0x22,0x29,0x34,0x3f,0x0e,0x05,0x18,0x13,
        0xca,0xc1,0xdc,0xd7,0xe6,0xed,0xf0,0xfb,0x92,0x99,0x84,0x8f,0xbe,0xb5,0xa8,0xa3
};
```

```
// Decryption: Multiply by 13 for InverseMixColumns
unsigned char mul13[256] =
{
        0x00,0x0d,0x1a,0x17,0x34,0x39,0x2e,0x23,0x68,0x65,0x72,0x7f,0x5c,0x51,0x46,0x4b,
        0xd0,0xdd,0xca,0xc7,0xe4,0xe9,0xfe,0xf3,0xb8,0xb5,0xa2,0xaf,0x8c,0x81,0x96,0x9b,
        0xbb,0xb6,0xa1,0xac,0x8f,0x82,0x95,0x98,0xd3,0xde,0xc9,0xc4,0xe7,0xea,0xfd,0xf0,
        0x6b,0x66,0x71,0x7c,0x5f,0x52,0x45,0x48,0x03,0x0e,0x19,0x14,0x37,0x3a,0x2d,0x20,
        0x6d,0x60,0x77,0x7a,0x59,0x54,0x43,0x4e,0x05,0x08,0x1f,0x12,0x31,0x3c,0x2b,0x26,
        0xbd,0xb0,0xa7,0xaa,0x89,0x84,0x93,0x9e,0xd5,0xd8,0xcf,0xc2,0xe1,0xec,0xfb,0xf6,
        0xd6,0xdb,0xcc,0xc1,0xe2,0xef,0xf8,0xf5,0xbe,0xb3,0xa4,0xa9,0x8a,0x87,0x90,0x9d,
        0x06,0x0b,0x1c,0x11,0x32,0x3f,0x28,0x25,0x6e,0x63,0x74,0x79,0x5a,0x57,0x40,0x4d,
        0xda,0xd7,0xc0,0xcd,0xee,0xe3,0xf4,0xf9,0xb2,0xbf,0xa8,0xa5,0x86,0x8b,0x9c,0x91,
        0x0a,0x07,0x10,0x1d,0x3e,0x33,0x24,0x29,0x62,0x6f,0x78,0x75,0x56,0x5b,0x4c,0x41,
        0x61,0x6c,0x7b,0x76,0x55,0x58,0x4f,0x42,0x09,0x04,0x13,0x1e,0x3d,0x30,0x27,0x2a,
        0xb1,0xbc,0xab,0xa6,0x85,0x88,0x9f,0x92,0xd9,0xd4,0xc3,0xce,0xed,0xe0,0xf7,0xfa,
        0xb7,0xba,0xad,0xa0,0x83,0x8e,0x99,0x94,0xdf,0xd2,0xc5,0xc8,0xeb,0xe6,0xf1,0xfc,
        0x67,0x6a,0x7d,0x70,0x53,0x5e,0x49,0x44,0x0f,0x02,0x15,0x18,0x3b,0x36,0x21,0x2c,
        0x0c,0x01,0x16,0x1b,0x38,0x35,0x22,0x2f,0x64,0x69,0x7e,0x73,0x50,0x5d,0x4a,0x47,
        0xdc,0xd1,0xc6,0xcb,0xe8,0xe5,0xf2,0xff,0xb4,0xb9,0xae,0xa3,0x80,0x8d,0x9a,0x97
};

// Decryption: Multiply by 14 for InverseMixColumns
unsigned char mul14[256] =
{
        0x00,0x0e,0x1c,0x12,0x38,0x36,0x24,0x2a,0x70,0x7e,0x6c,0x62,0x48,0x46,0x54,0x5a,
        0xe0,0xee,0xfc,0xf2,0xd8,0xd6,0xc4,0xca,0x90,0x9e,0x8c,0x82,0xa8,0xa6,0xb4,0xba,
        0xdb,0xd5,0xc7,0xc9,0xe3,0xed,0xff,0xf1,0xab,0xa5,0xb7,0xb9,0x93,0x9d,0x8f,0x81,
        0x3b,0x35,0x27,0x29,0x03,0x0d,0x1f,0x11,0x4b,0x45,0x57,0x59,0x73,0x7d,0x6f,0x61,
        0xad,0xa3,0xb1,0xbf,0x95,0x9b,0x89,0x87,0xdd,0xd3,0xc1,0xcf,0xe5,0xeb,0xf9,0xf7,
        0x4d,0x43,0x51,0x5f,0x75,0x7b,0x69,0x67,0x3d,0x33,0x21,0x2f,0x05,0x0b,0x19,0x17,
        0x76,0x78,0x6a,0x64,0x4e,0x40,0x52,0x5c,0x06,0x08,0x1a,0x14,0x3e,0x30,0x22,0x2c,
        0x96,0x98,0x8a,0x84,0xae,0xa0,0xb2,0xbc,0xe6,0xe8,0xfa,0xf4,0xde,0xd0,0xc2,0xcc,
        0x41,0x4f,0x5d,0x53,0x79,0x77,0x65,0x6b,0x31,0x3f,0x2d,0x23,0x09,0x07,0x15,0x1b,
```

```
        0xa1,0xaf,0xbd,0xb3,0x99,0x97,0x85,0x8b,0xd1,0xdf,0xcd,0xc3,0xe9,0xe7,0xf5,0xfb,
        0x9a,0x94,0x86,0x88,0xa2,0xac,0xbe,0xb0,0xea,0xe4,0xf6,0xf8,0xd2,0xdc,0xce,0xc0,
        0x7a,0x74,0x66,0x68,0x42,0x4c,0x5e,0x50,0x0a,0x04,0x16,0x18,0x32,0x3c,0x2e,0x20,
        0xec,0xe2,0xf0,0xfe,0xd4,0xda,0xc8,0xc6,0x9c,0x92,0x80,0x8e,0xa4,0xaa,0xb8,0xb6,
        0x0c,0x02,0x10,0x1e,0x34,0x3a,0x28,0x26,0x7c,0x72,0x60,0x6e,0x44,0x4a,0x58,0x56,
        0x37,0x39,0x2b,0x25,0x0f,0x01,0x13,0x1d,0x47,0x49,0x5b,0x55,0x7f,0x71,0x63,0x6d,
        0xd7,0xd9,0xcb,0xc5,0xef,0xe1,0xf3,0xfd,0xa7,0xa9,0xbb,0xb5,0x9f,0x91,0x83,0x8d
};
```

```
unsigned char mainkey[16];
unsigned char roundkeys[10][16];
unsigned char rconkeys[10] = { 0x01, 0x02,0x04,0x08,0x10,0x20,0x40,0x80,0x1b,0x36 };

unsigned char keytemp1[16], keytemp2[4], tmp[16];

void roundkeyexpansion(int roundnum)
{
        int temp = keytemp1[3];
        keytemp1[3] = keytemp1[7];
        keytemp1[7] = keytemp1[11];
        keytemp1[11] = keytemp1[15];
        keytemp1[15] = temp;
        for (int i = 3; i < 16; i = i + 4) {
                keytemp1[i] = sbox[keytemp1[i]];
        }

        keytemp1[3] ^= rconkeys[roundnum];
```

```
	for (int i = 0; i < 13; i = i + 4) {
		keytemp1[i] = keytemp1[i + 3] xor keytemp1[i];
	}

	for (int i = 0; i < 13; i = i + 4) {
		keytemp1[i + 1] = keytemp1[i + 1] xor keytemp1[i];
	}
	for (int i = 0; i < 13; i = i + 4) {
		keytemp1[i + 2] ^= keytemp1[i + 1];
	}
	for (int i = 0; i < 4; i = i + 1) {
		keytemp1[(4 * i) + 3] = keytemp1[(4 * i) + 2] xor keytemp2[i];
	}
}

void assignkey() {
	keytemp1[0] = 0x54;
	keytemp1[4] = 0x68;
	keytemp1[8] = 0x61;
	keytemp1[12] = 0x74;
	keytemp1[1] = 0x73;
	keytemp1[5] = 0x20;
	keytemp1[9] = 0x6d;
	keytemp1[13] = 0x79;
	keytemp1[2] = 0x20;
	keytemp1[6] = 0x4b;
	keytemp1[10] = 0x75;
	keytemp1[14] = 0x6e;
	keytemp1[3] = 0x67;
	keytemp1[7] = 0x20;
	keytemp1[11] = 0x46;
	keytemp1[15] = 0x75;
	for (int i = 0; i <= 3; i++)
```

```
	{
		keytemp2[i] = keytemp1[4 * i + 3];
	}
}


void assignkey2(int round)
{
	for (int i = 0; i < 4; i = i + 1) {
		for (int j = 0; j < 16; j = j + 4)
		{
			keytemp1[j + i] = roundkeys[round - 1][j + i];
		}
	}
	for (int i = 0; i <= 3; i++)
	{
		keytemp2[i] = keytemp1[4 * i + 3];
	}
}


void addKey(int roundnum) {
	for (int i = 0; i < 4; i = i + 1) {
		for (int j = 0; j < 16; j = j + 4)
		{
			roundkeys[roundnum][j + i] = keytemp1[j + i];
		}
	}
}


void roundkeygen() {
```

```
	for (int i = 0; i < 10; i++)
	{
		if (i == 0)
			assignkey();
		else
			assignkey2(i);
		roundkeyexpansion(i);
		addKey(i);
	}
}


int plaintext[16] = {};

void addroundkey(int round) {
	if (round == 0) {
		for (int i = 0; i < 16; i++) {
			plaintext[i] ^= mainkey[i];
		}

	}
	else
	{
		for (int i = 0; i < 16; i++) {
			plaintext[i] ^= roundkeys[round - 1][i];
		}
	}
}
void initialise() {

	mainkey[0] = 0x54;
	mainkey[4] = 0x68;
	mainkey[8] = 0x61;
```

```
	mainkey[12] = 0x74;
	mainkey[1] = 0x73;
	mainkey[5] = 0x20;
	mainkey[9] = 0x6d;
	mainkey[13] = 0x79;
	mainkey[2] = 0x20;
	mainkey[6] = 0x4b;
	mainkey[10] = 0x75;
	mainkey[14] = 0x6e;
	mainkey[3] = 0x67;
	mainkey[7] = 0x20;
	mainkey[11] = 0x46;
	mainkey[15] = 0x75;
}


void invSubBytes() {
	for (int i = 0; i < 16; i++)
	{
		plaintext[i] = invsbox[plaintext[i]];
	}
}

void invshiftRowOnce(int k) {
	int temp = plaintext[k + 3];
	for (int i = k + 3; i > k; i--)
	{
		plaintext[i] = plaintext[i - 1];
	}
	plaintext[k] = temp;
}

void invshiftRows() {
```

```
        invshiftRowOnce(4);
        invshiftRowOnce(8); invshiftRowOnce(8);
        invshiftRowOnce(12); invshiftRowOnce(12); invshiftRowOnce(12);

}


void invMixColumns() {

        tmp[0] = (unsigned char)mul14[plaintext[0]] ^ mul11[plaintext[4]] ^ mul13[plaintext[8]] ^
mul9[plaintext[12]];
        tmp[4] = (unsigned char)mul9[plaintext[0]] ^ mul14[plaintext[4]] ^ mul11[plaintext[8]] ^
mul13[plaintext[12]];
        tmp[8] = (unsigned char)mul13[plaintext[0]] ^ mul9[plaintext[4]] ^ mul14[plaintext[8]] ^
mul11[plaintext[12]];
        tmp[12] = (unsigned char)mul11[plaintext[0]] ^ mul13[plaintext[4]] ^ mul9[plaintext[8]] ^
mul14[plaintext[12]];

        tmp[1] = (unsigned char)mul14[plaintext[1]] ^ mul11[plaintext[5]] ^ mul13[plaintext[9]] ^
mul9[plaintext[13]];
        tmp[5] = (unsigned char)mul9[plaintext[1]] ^ mul14[plaintext[5]] ^ mul11[plaintext[9]] ^
mul13[plaintext[13]];
        tmp[9] = (unsigned char)mul13[plaintext[1]] ^ mul9[plaintext[5]] ^ mul14[plaintext[9]] ^
mul11[plaintext[13]];
        tmp[13] = (unsigned char)mul11[plaintext[1]] ^ mul13[plaintext[5]] ^ mul9[plaintext[9]] ^
mul14[plaintext[13]];

        tmp[2] = (unsigned char)mul14[plaintext[2]] ^ mul11[plaintext[6]] ^ mul13[plaintext[10]] ^
mul9[plaintext[14]];
        tmp[6] = (unsigned char)mul9[plaintext[2]] ^ mul14[plaintext[6]] ^ mul11[plaintext[10]] ^
mul13[plaintext[14]];
        tmp[10] = (unsigned char)mul13[plaintext[2]] ^ mul9[plaintext[6]] ^ mul14[plaintext[10]] ^
mul11[plaintext[14]];
```

```
	tmp[14] = (unsigned char)mul11[plaintext[2]] ^ mul13[plaintext[6]] ^ mul9[plaintext[10]] ^
mul14[plaintext[14]];

	tmp[3] = (unsigned char)mul14[plaintext[3]] ^ mul11[plaintext[7]] ^ mul13[plaintext[11]] ^
mul9[plaintext[15]];
	tmp[7] = (unsigned char)mul9[plaintext[3]] ^ mul14[plaintext[7]] ^ mul11[plaintext[11]] ^
mul13[plaintext[15]];
	tmp[11] = (unsigned char)mul13[plaintext[3]] ^ mul9[plaintext[7]] ^ mul14[plaintext[11]] ^
mul11[plaintext[15]];
	tmp[15] = (unsigned char)mul11[plaintext[3]] ^ mul13[plaintext[7]] ^ mul9[plaintext[11]] ^
mul14[plaintext[15]];

	for (int i = 0; i < 16; i++) {
		plaintext[i] = tmp[i];
	}
}

void aes_decrypt() {
	addroundkey(10);

	invshiftRows();
	invSubBytes();
	addroundkey(9);

	invMixColumns();
	invshiftRows();
	invSubBytes();
	addroundkey(8);

	invMixColumns();
	invshiftRows();
	invSubBytes();
	addroundkey(7);
```

```
invMixColumns();
invshiftRows();
invSubBytes();
addroundkey(6);

invMixColumns();
invshiftRows();
invSubBytes();
addroundkey(5);

invMixColumns();
invshiftRows();
invSubBytes();
addroundkey(4);

invMixColumns();
invshiftRows();
invSubBytes();
addroundkey(3);

invMixColumns();
invshiftRows();
invSubBytes();
addroundkey(2);

invMixColumns();
invshiftRows();
invSubBytes();
addroundkey(1);

invMixColumns();
invshiftRows();
```

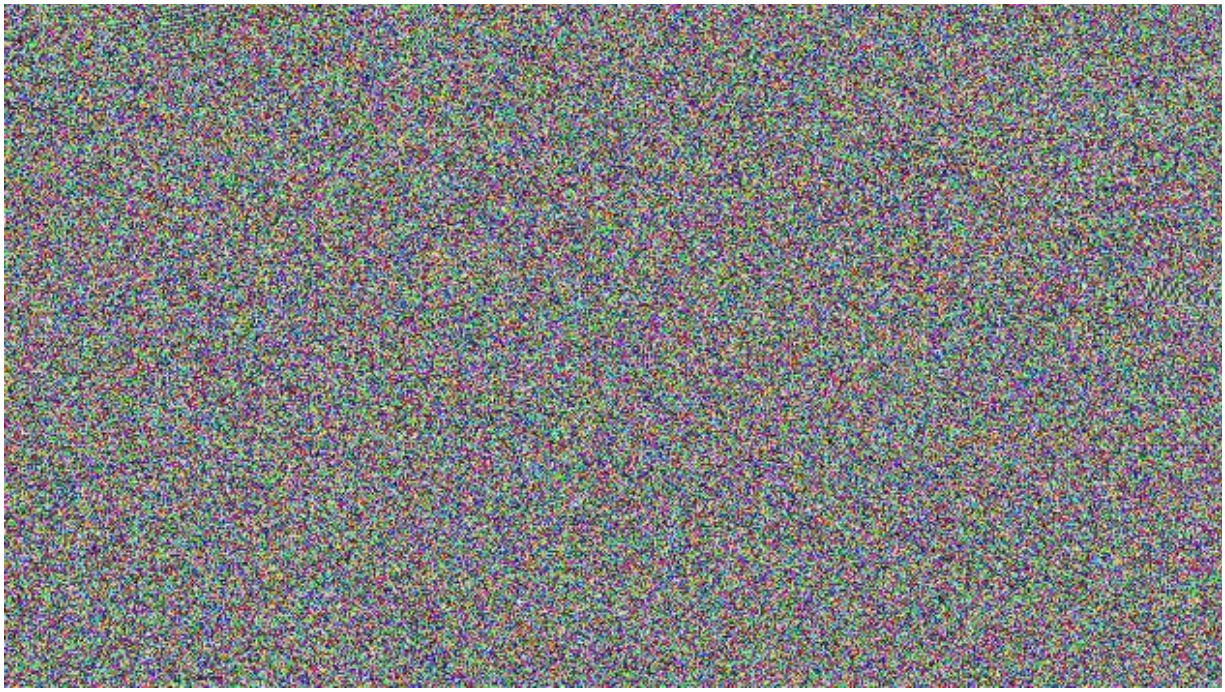
```
        invSubBytes();
        addroundkey(0);
}

vector<int> myvect;
Vec3b buf;
int rows, columns;
void getdata(char* arg){
        int sfd = socket(AF_INET,SOCK_STREAM,0);
        struct sockaddr_in addr;

        addr.sin_family = AF_INET;
        addr.sin_port = htons(port);

        addr.sin_addr.s_addr = inet_addr(arg);
        connect(sfd, (struct sockaddr *)&addr, sizeof(addr));

        int size = 0;
        read(sfd,&size,sizeof(size));
        read(sfd,&rows,sizeof(rows));
        read(sfd,&columns,sizeof(columns));
        int cnt = ceil((double)size/5000);int l = 0;
        while(cnt--){
                int a[5000]={0};
                read(sfd,a,sizeof(a));
                for(int i=0;i<5000;i++){
                        if(l>=size){i=5005;continue;}
                        myvect.push_back(a[i]);
                        l++;
                }
        }
}
int main(int argc,char** argv){
```

```
roundkeygen();
getdata(argv[1]);
Mat image(rows, columns, CV_8UC3, Scalar(0, 0, 0));

if (myvect.size() % 4 == 0)
{
	for (int i = 0; i < myvect.size() / 16; i++) {
		for (int j = i * 16; j < (i * 16) + 16; j++)
		{
			plaintext[j % 16] = myvect[j];
		}
		aes_decrypt();
		for (int j = i * 16; j < (i * 16) + 16; j++)
		{
			myvect[j] = plaintext[j % 16];
		}
	}
}

for (int i = 0; i < image.rows*image.cols; i = i + 1)
{
	buf[0] = myvect[i * 3];
	buf[1] = myvect[(i * 3) + 1];
	buf[2] = myvect[(i * 3) + 2];

	image.at<Vec3b>(i / image.cols, i%image.cols) = buf;
	//cout << "( " << i / image.cols << ",\n ";// << i % image.cols << " ),\n";
}

namedWindow("Image", WINDOW_GUI_NORMAL);
imshow("Image", image);
waitKey(0);
```

Results:

Given Image is-

Encrypted Image: (This image is sent from Sender to Receiver)-



Received Image: (After receiving the Encrypted Image and Decrypted) -

Applications:

The advent of mobile devices with high-resolution cameras on board, along with photo sharing has become highly popular. However, this development has raised another important issue that is privacy. Now-a-days, private data including photos are leaked from prominent photo sharing sites which are assumed as secure sites So, Encryption provides privacy and security.

Image Encryption can be implemented in transferring Highly confidential Data in Military sector, Government sector, etc.

Social Networking companies can encrypt images to secure the data theft from hackers. So Encryption is highly reliable.

Conclusion:

Secure Data Transmission over TCP-IP network has been achieved (loopback to same computer) and it's applications have been stated.