SPL-1 Project Report

FileCryptoZipper

A File Manipulation Tool to Encrypt/Decrypt and Compress/Decompress with Algorithm Selection

Submitted by

Shahid-E-Kaiser Md. Tashrif

BSSE Roll No. : 1448

BSSE Session: 2021-2022

Submitted to

SPL-1 Coordinators

Supervised by

Toukir Ahammed

Lecturer

Institute of Information Technology

Supervisor's Approval: ________________

(signature)

Institute of Information Technology

University of Dhaka

[15-12-2023]

Table of Contents

1.Introduction....1

1.1.Security at the Core....1

1.2.Efficient Data Management....1

1.3.Real Life Applications....1

2.Background....2

2.1.Galois Field Arithmetic....2

2.2.Bitmasking....2

2.3.Fiestel Network....2

2.4.Block Cipher....2

2.5.Dictionary-Handling....2

3.Description of the project....2

3.1.AES Encryption....3

3.2.AES Decryption....4

3.3.BlowFish Encryption....4

3.4.BlowFish Decryption....5

3.5.Huffman Compression....5

3.6.Huffman Decompression....6

3.7.LZW Compression....6

3.8.LZW Decompression....6

4.Implementation....7

4.1.AES Encryption....7

4.1.1.Generate Keys and Expand....7

4.1.2.Read File to State Array....8

4.1.3.Encryption in 14 Rounds....9

4.1.4.Writing Encrypted Content....11

4.2.AES Decryption....11

4.2.1.Read Back Key....12

4.2.2.Read Encrypted Contents....12

4.2.3.Decrypt State Array....12

4.2.4.Write Back File....13

4.3.BlowFish Encryption....13

4.3.1.Key Expand....13

4.3.2.Read File & Encrypt....14

4.3.3.Write Back Encrypted File....14

4.4.BlowFish Decryption....15

4.4.1.Read Key....15

4.4.2.Read File & Decrypt……………………………………………………………………15
4.4.3.Write Back……………………………………………………………………………16
4.5.Huffman Compression………………………………………………………………16
4.5.1.Building Huffman Tree……………………………………………………………16
4.5.2.Writing Huff-Codes to File………………………………………………………17
4.6.Huffman Decompression……………………………………………………………18
4.6.1.Read File and Decompress………………………………………………………18
4.7.LZW Compression………………………………………………………………………19
4.7.1.Build Dictionary………………………………………………………………………19
4.7.2.Generating Compressed File………………………………………………………19
4.8.LZW Decompression……………………………………………………………………20
4.8.1.Reading Compressed File…………………………………………………………20
4.9.Comparison Report……………………………………………………………………21
5.User Interface……………………………………………………………………………21
5.1.Encryption/Decryption Example…………………………………………………21
5.2.Compression/Decompression Example…………………………………………23
5.3.Comparison Report Example………………………………………………………25
6.Challenges Faced…………………………………………………………………………25
6.1.Algorithm Complexity………………………………………………………………26
6.2.Key Management………………………………………………………………………26
6.3.Integration of Multiple Algorithms……………………………………………26
6.4.File Format Compatibility…………………………………………………………26
7.Conclusion…………………………………………………………………………………26
8.References…………………………………………………………………………………28

Index of Figures

Figure 1:AES Encryption……………………………………………………………………………………….……….4
Figure 2: Decryption………………………………………………………………………………………………...4
Figure 3:BlowFish Encryption………………………………………………………………………………..….5
Figure 4:LZW Compression…………………………………………………………………………………...6
Figure 5:Key Generation…………………………………………………………………………………………..7
Figure 6:Key Expansion……………………………………………………………………………………………8
Figure 7:Writing Key……………………………………………………………………………………….….8
Figure 8:Read to State……………………………………………………………………………………………9
Figure 9:Encryption………………………………………………………………………….……………………9
Figure 10:Substitute Operation………………………………………………………………………………10
Figure 11:Shift Row……………………………….…………………………………………………………………10
Figure 12:Mix Column……………………………………………………………………………………………11
Figure 13:Write Back Contents………………………………………………………….……………………..11
Figure 14:Read Keys…………………………………………………….……………………………………….11
Figure 15:Read Encrypted File…………….…………………………………………………………………12
Figure 16:Decryption in 14 Rounds….……………………………………………………………………….12
Figure 17:Write Back File……...........……………………………………………………………………...13
Figure 18:Key Expansion…….……………………………………………………………………………….13
Figure 19:Read & Encrypt Contents…………………..…………………………………………………….14
Figure 20:Writing Encrypted File………..………………………………………………………………..14
Figure 21:Key Read(BlowFish)………..…………………………………………………………………….15
Figure 22:Read & Decrypt Blocks…..…………………………………………………………………………15
Figure 23:Decryption Process……………..………………………………………………………………...16
Figure 24:Write File…….………………………………………………………………………………………16
Figure 25:Building Huffman Tree………………..……………………………………………………………17
Figure 26:Writing Huff-Codes……………………………….……………………………………………….18
Figure 27:Decompressing File………………………………………………………………………………18
Figure 28:Read & Create Dictionary…………………………………………………………………………19
Figure 29:Writing Index of Dictionary……………………………………………………………………..20
Figure 30:Decompression from Dictionary……………………………………………….………..20
Figure 31:Writing Report in File……………………………………………………………………………21
Figure 32:User Interface…………………………………………………………………….……………………22
Figure 33:Encryption UI………………………………………………………………………………………….22
Figure 34:File path Input………...…………………………………………………………………………….22
Figure 35:Original file & Encrypted File…………………………………………………………………….22

Figure 36:Decryption UI………………...23
Figure 37:Decrypted File………………………...23
Figure 38:Compression UI……………………………………………………………………………………………….23
Figure 39:Compression………………………………..24
Figure 40:Original File & Compressed File…………………………………………………………………..24
Figure 41:Decompression UI……………………………………………………………………………………..24
Figure 42:Decompressed File…………………………………………………………………………………….25
Figure 43:Comparison Report……………………………………………………………………………………….25

1. Introduction

FileCryptoZipper stands as file manipulation tool, offering users the ability to secure and optimize their digital assets with precision. This application enables users to seamlessly encrypt or decrypt text files, providing an additional layer of security through advanced encryption algorithms, including **AES (Advanced Encryption Standard)** and **Blowfish**. In addition, it offers efficient compression and decompression options, employing reputable algorithms such as **Huffman** and **LZW**.

1.1. Security at the Core

At the core of FileCryptoZipper lies a commitment to data security. The inclusion of AES and Blowfish encryption ensures that user's files are shielded with strong protection. This feature is particularly crucial for professionals and individuals alike, safeguarding confidential information during storage and transmission. Whether it is a business executive handling sensitive corporate documents or an individual securing personal files, FileCryptoZipper delivers a reliable protection against unauthorized access.

1.2. Efficient Data Management

In addition to its formidable encryption capabilities, the program optimizes file storage and transfer through its compression and decompression functionalities. The application employs Huffman and LZW algorithms, enabling users to reduce file sizes without compromising data integrity. This proves invaluable in scenarios where storage space is limited or when swift and efficient data transfer is imperative.

Furthermore, the application reports the time it took for encryption or decryption for letting the user know the efficiency of the algorithms. In case of compression, it reports the compressed and original size for determining effective compression for the specific file. By these reports, the user can know which algorithm is the best for a file.

1.3. Real-World Applications

FileCryptoZipper transcends theoretical concepts, finding practical applications in various real-life scenarios:

1. Business and Corporate Environments: Protect sensitive business documents and confidential information during storage and transmission, ensuring data integrity and security.

2. Individuals: Securely encrypt and compress personal files, preserving privacy and optimizing storage space.

3. Academic and Research: Enhance the security of research data and scholarly work through robust encryption, while efficiently managing file sizes for academic projects.

4. Legal and Medical Fields: Ensure the confidentiality of legal documents, medical records, and sensitive information through secure encryption and optimized file storage.

Overall, FileCryptoZipper is more than a tool. It's a reliable companion for individuals and organizations seeking a seamless integration of security and efficiency in their file management processes.

2. Background

The project has some complex calculations and terms that the user may struggle to understand. Some important terms are discussed below for clarification:

2.1.Galois Field Arithmetic In the **MixColumns** step of AES, Galois Field multiplication involves multiplying each element in a column by a fixed polynomial in the Galois Field GF(2^8). This multiplication operation provides diffusion and adds complexity to the encryption process, contributing to AES's strength. The specific polynomial used is x^4+1, which corresponds to an irreducible polynomial in GF(2^8). This multiplication in the Galois Field helps achieve the desired diffusion effect, enhancing the cryptographic strength of the AES algorithm.

2.2.Bitmasking It involves various bitmasking operations such as bitwise shifting, XOR, OR operations. Bitmasking is a bitwise operation in programming that involves using binary masks to manipulate specific bits within binary numbers. It includes operations like AND, OR, XOR, and shifting to efficiently set, clear, or toggle individual bits. In AES encryption, bitmasking is utilized for tasks such as substitution, permutation, and key manipulation, contributing to data transformation and algorithmic complexity. Example: 1010 << 1 == 0100 by shifting the bits 1 bit left.

2.3.Fiestel Network A Feistel Network is a cryptographic structure commonly used in block ciphers. It involves dividing the input into two halves, processing one half through a function that depends on the other half and a subkey, and then swapping the halves. This process is repeated through multiple rounds, enhancing the security and efficiency of the encryption or decryption process. The Feistel Network design allows for a balance between simplicity and effectiveness in symmetric key algorithms like DES and **Blowfish**.

2.4.Block Cipher A block cipher is a type of symmetric-key cryptographic algorithm that encrypts or decrypts fixed-size blocks of data, typically 64 or 128 bits. It operates on blocks of plaintext, transforming them into **Ciphertext** using a secret key. The encryption and decryption processes are reversible, and the same key is used for both operations. Block ciphers are widely used in securing data and communications, forming the basis for various cryptographic protocols and systems. Examples include DES, AES, and Blowfish.

2.5.Dictionary-Handling The dictionary is a dynamic data structure that starts with entries for individual characters. As the algorithm processes the input data, it expands the dictionary by adding new entries for encountered patterns. Each entry consists of a unique code and the corresponding pattern it represents. During compression, the algorithm looks up the longest matching pattern in the dictionary, outputs its code, and adds a new entry for the extended pattern. This dynamic dictionary allows LZW to efficiently represent repeated patterns with shorter codes, optimizing the compression process.

3. Description of the Project

My project is designed to take Filepath as input from the user to make operations on that file.

It will prompt the user for choosing from 5 options i.e. "**1.Encryption, 2.Decryption, 3.Compresssion, 4.Decompression and 5.Exit**" and after choosing one of it, the corresponding options again prompts to choose which algorithm is needed. The Encryption-Decryption option gives "**1.AES and 2.BlowFish**" option and the Compression-Decompression gives "**1.Huffman and 2.LZW**" option. Upon choosing, it performs certain operations and shows the taken time/compressed size. The user can perform operations on multiple files simultaneously as it stores the key and relevant data to read back when needed to revert the previous operation. It includes following attributes:

- Get filepath from user
- Prompt user for choosing options
- If (choice=1)
  Perform AES Encryption if chosen 1 after
  Perform BlowFish Encryption if chosen 2
- If (choice=2)
  Perform AES Decryption if chosen 1 after
  Perform BlowFish Decryption if chosen 2
- Upon choosing 3 and 4:
  Perform Huffman Compression/Decompression if chosen 1
  Perform LZW Compression/Decompression if chosen 2
- All these operations will create a report about their taken time and reduced/increased size in a text file "report.txt"

AES ENCRYPTION:

- Generate Key and Perform Key Expansion
- Read file in 128 bit chunks to bytestream and 4x4 state array
- Perform Encryption in 14 rounds:
  - Substitute Byte
  - Shift Row
  - Mix Column
  - Add Round Key
- Write the encrypted contents back to file

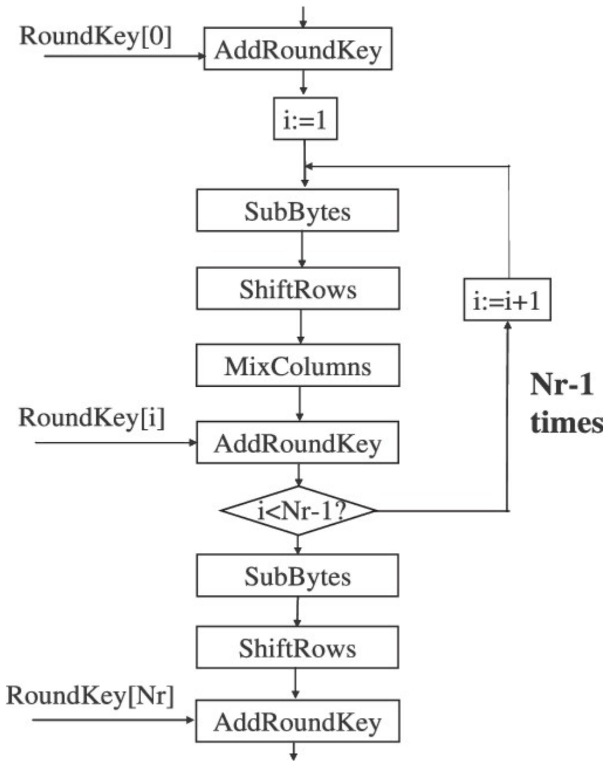


Figure 1:Flowchart of AES Encryption(source:www.reasearchgate.net)

AES DECRYPTION:

- Read back key from previous stored key
- Read encrypted file to state array
- Decrypt contents by reversing in 14 rounds
- Write decrypted contents back to file

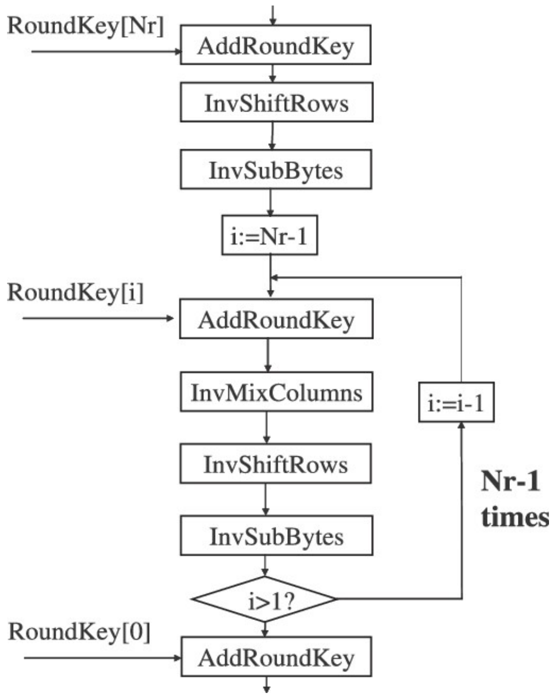


Figure 2: Flowchart of AES Decryption(source:www.researchgate.net)

BLOWFISH ENCRYPTION:

- Expand key
- Read file in 64 bit blocks dividing into 2 32bit blocks
- Encrypting both block
- Write encrypted contents in file

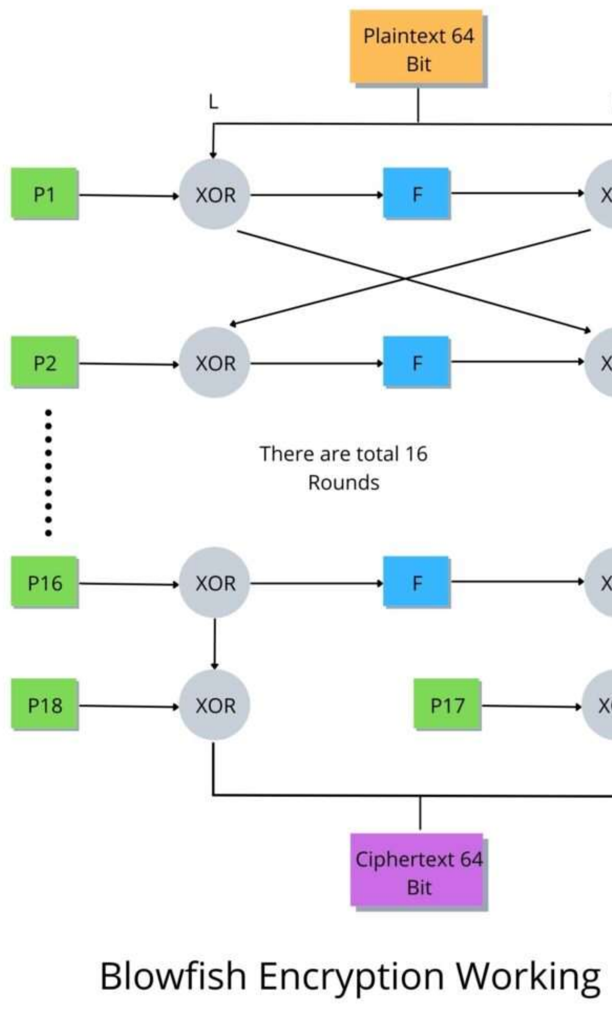


Figure 3: Flowchart of Blowfish Encryption(source:www.cybermeteoroid.com)

BLOWFISH DECRYPTION:

- Read key from file
- Read file in 64 bits
- Decrypting blocks
- Writing decrypted contents in file

HUFFMAN COMPRESSION:

- Read frequencies of file characters
- Make Huffman tree
- Replace Huffman codes with file generating compressed file

HUFFMAN DECOMPRESSION:

- Read file bit by bit
- Traverse the Huffman tree to get original character
- Write it in new decompressed file

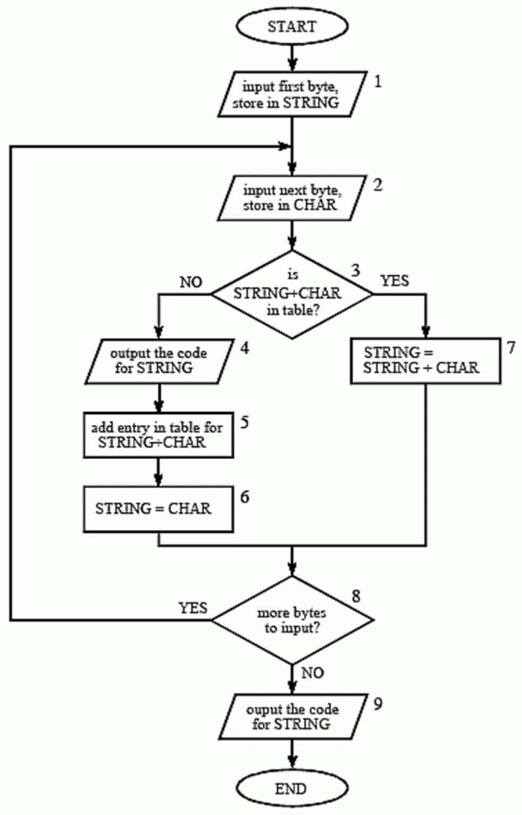

START
input first byte, store in STRING
1
input next byte, store in CHAR
2
is STRING+CHAR in table?
3
NO
YES
output the code for STRING
4
add entry in table for STRING+CHAR
5
STRING = CHAR
6
STRING = STRING + CHAR
7
more bytes to input?
8
YES
NO
ouput the code for STRING
9
END

FIGURE 27-7
LZW compression flowchart. The variable, *CHAR*, is a single byte. The variable, *STRING*, is a variable length sequence of bytes. Data are read from the input file (box 1 & 2) as single bytes, and written to the compressed file (box 4) as 12 bit codes. Table 27-3 shows an example of this algorithm.

Figure 4: Flowchart of LZW Compression(www.researchgate.net)

LZW COMPRESSION:

- Read files repeated sequences to make a dictionary structure
- Mapping index codes to sequences
- Replacing codes to generate compressed file

LZW DECOMPRESSION:

- Read bit by bit and get character from dictionary
- Writing decompressed characters to new file

4. Implementation

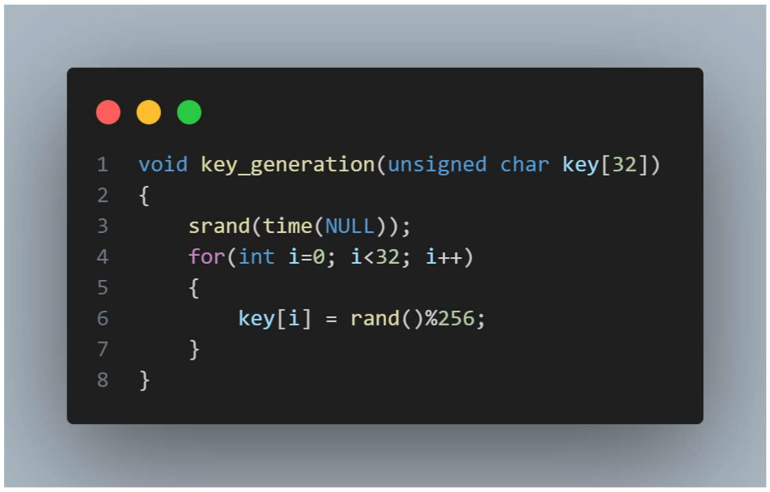
The implementation and testing of the project is described in detail below.

4.1 AES Encryption

The AES encryption performs following operations

4.1.1 Generate Keys and Expand

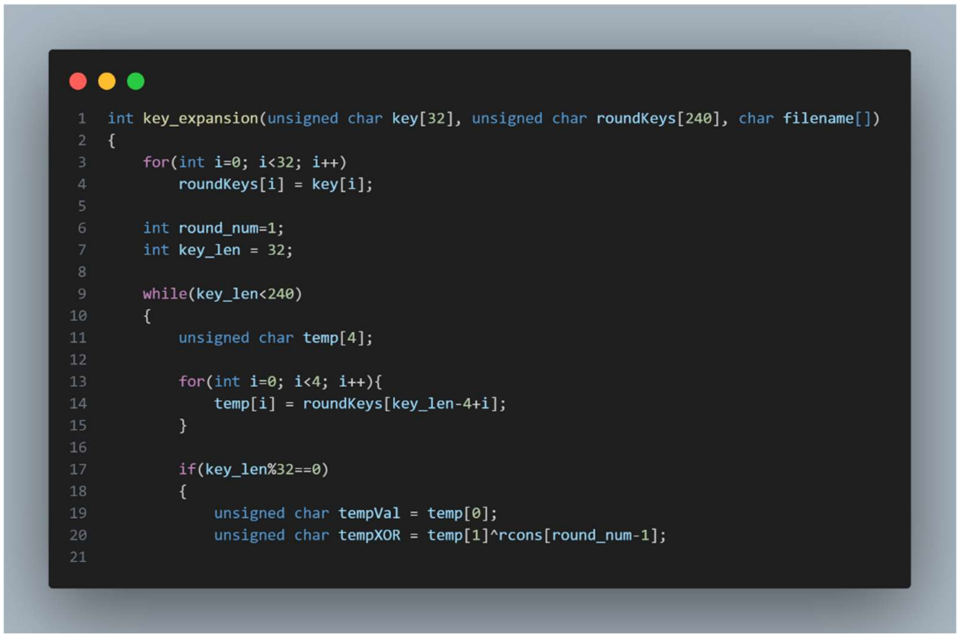
After choosing the algorithm, it will generate keys with random function and store it in key array as following.



```c
void key_generation(unsigned char key[32])
{
    srand(time(NULL));
    for(int i=0; i<32; i++)
    {
        key[i] = rand()%256;
    }
}
```

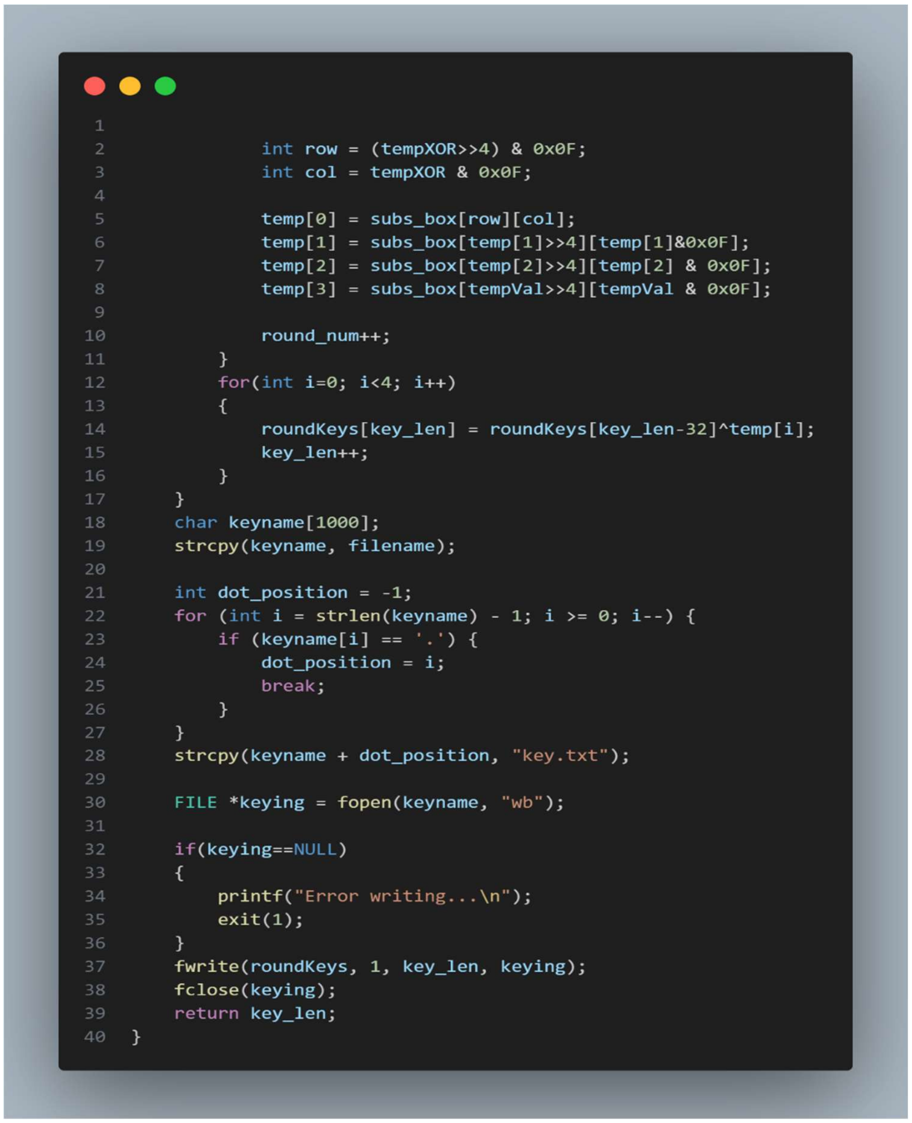
Figure 5: Key generation

Then it goes through the key expansion algorithm to get 240 keys and it writes it in a file.



```c
int key_expansion(unsigned char key[32], unsigned char roundKeys[240], char filename[])
{
    for(int i=0; i<32; i++)
        roundKeys[i] = key[i];

    int round_num=1;
    int key_len = 32;

    while(key_len<240)
    {
        unsigned char temp[4];

        for(int i=0; i<4; i++){
            temp[i] = roundKeys[key_len-4+i];
        }

        if(key_len%32==0)
        {
            unsigned char tempVal = temp[0];
            unsigned char tempXOR = temp[1]^rcons[round_num-1];

```

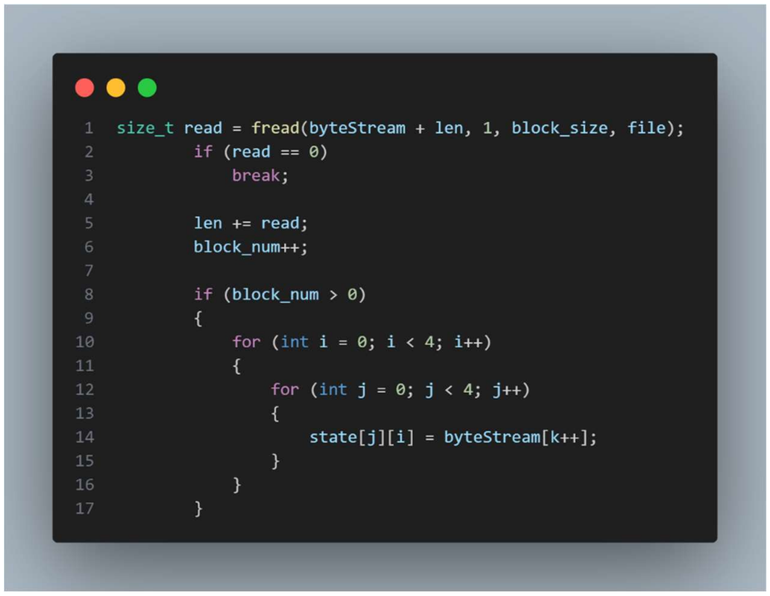
Figure 6: key expansion



```c

                int row = (tempXOR>>4) & 0x0F;
                int col = tempXOR & 0x0F;

                temp[0] = subs_box[row][col];
                temp[1] = subs_box[temp[1]>>4][temp[1]&0x0F];
                temp[2] = subs_box[temp[2]>>4][temp[2] & 0x0F];
                temp[3] = subs_box[tempVal>>4][tempVal & 0x0F];

                round_num++;
            }
            for(int i=0; i<4; i++)
            {
                roundKeys[key_len] = roundKeys[key_len-32]^temp[i];
                key_len++;
            }
        }
        char keyname[1000];
        strcpy(keyname, filename);

        int dot_position = -1;
        for (int i = strlen(keyname) - 1; i >= 0; i--) {
            if (keyname[i] == '.') {
                dot_position = i;
                break;
            }
        }
        strcpy(keyname + dot_position, "key.txt");

        FILE *keying = fopen(keyname, "wb");

        if(keying==NULL)
        {
            printf("Error writing...\n");
            exit(1);
        }
        fwrite(roundKeys, 1, key_len, keying);
        fclose(keying);
        return key_len;
    }
```

Figure 7:Writing key

4.1.2 Read File to State Array



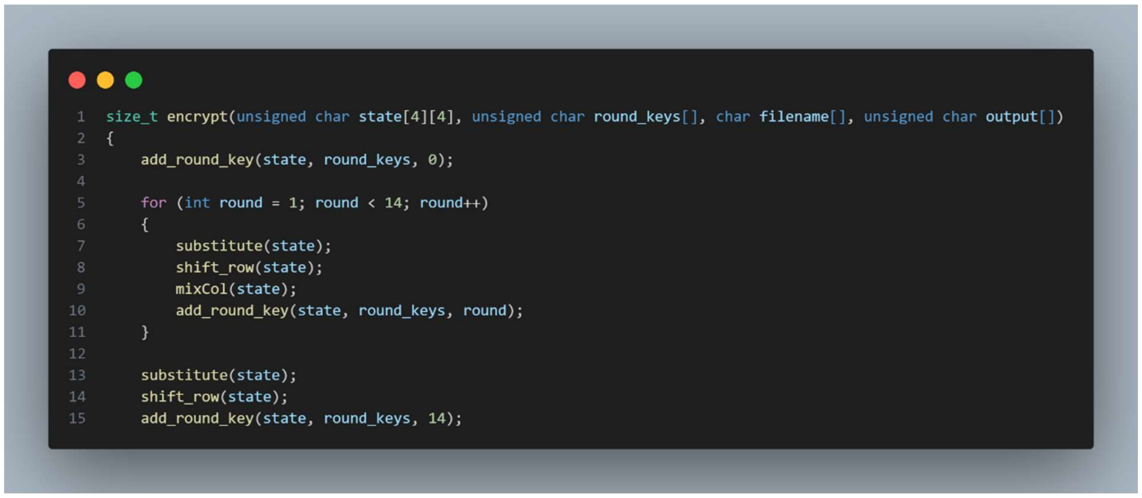
```c
size_t read = fread(byteStream + len, 1, block_size, file);
        if (read == 0)
            break;

        len += read;
        block_num++;

        if (block_num > 0)
        {
            for (int i = 0; i < 4; i++)
            {
                for (int j = 0; j < 4; j++)
                {
                    state[j][i] = byteStream[k++];
                }
            }
        }
```

Figure 8: Read to State

We read the file in binary mode to the bytestream array in 128 bit chunk and transfer it to 128 bit state array.

4.1.3 Encryption in 14 Rounds

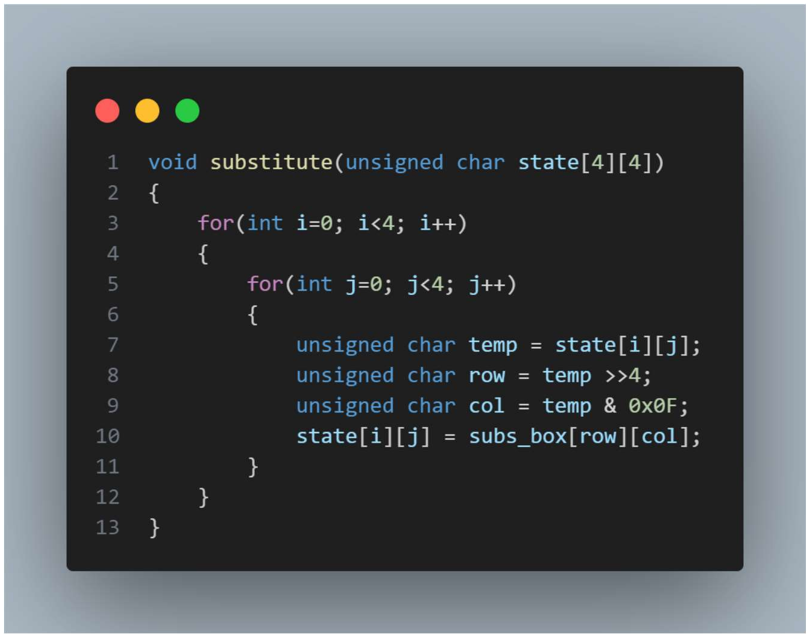
The state array is encrypted by the 4 operations in 14 rounds except the last round where mix column is missing.



```c
size_t encrypt(unsigned char state[4][4], unsigned char round_keys[], char filename[], unsigned char output[])
{
    add_round_key(state, round_keys, 0);

    for (int round = 1; round < 14; round++)
    {
        substitute(state);
        shift_row(state);
        mixCol(state);
        add_round_key(state, round_keys, round);
    }

    substitute(state);
    shift_row(state);
    add_round_key(state, round_keys, 14);
```

Figure 9: Encryption

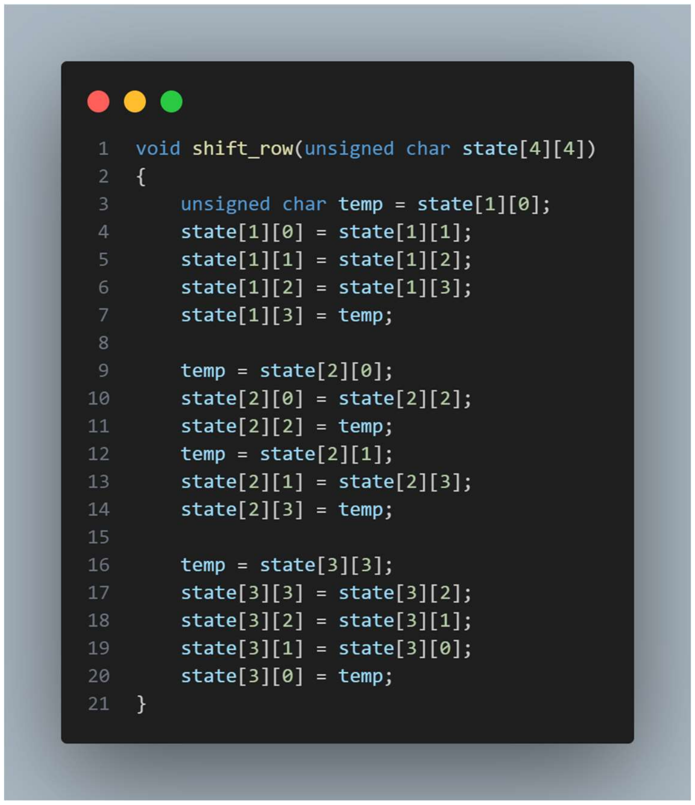
The Substitute part takes hex values from subs_box to replace the state array.



```c
void substitute(unsigned char state[4][4])
{
    for(int i=0; i<4; i++)
    {
        for(int j=0; j<4; j++)
        {
            unsigned char temp = state[i][j];
            unsigned char row = temp >>4;
            unsigned char col = temp & 0x0F;
            state[i][j] = subs_box[row][col];
        }
    }
}
```

Figure 10: Substitute Operation

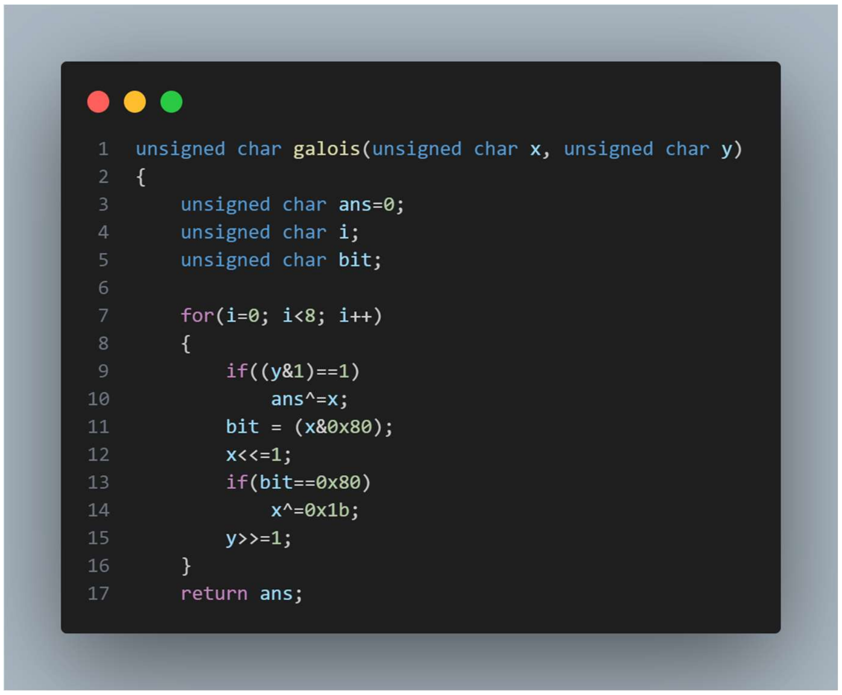
The Shift row shifts the first row 0 cell, second row 1 cell and third row 2 cell to the left circularly.



```c
void shift_row(unsigned char state[4][4])
{
    unsigned char 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;

    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;

    temp = state[3][3];
    state[3][3] = state[3][2];
    state[3][2] = state[3][1];
    state[3][1] = state[3][0];
    state[3][0] = temp;
}
```

Figure 11: Shift row

Then the mix column performs galois field multiplication with the state array and a constant array to produce diffusion.

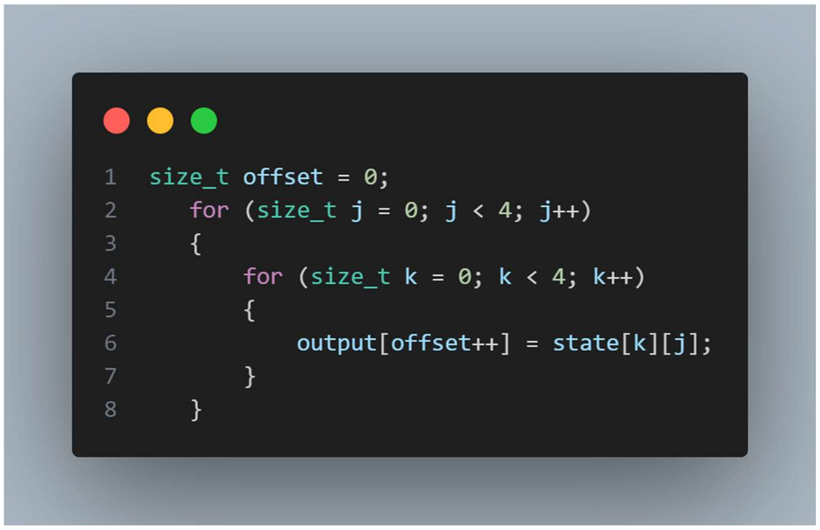


```c
unsigned char galois(unsigned char x, unsigned char y)
{
    unsigned char ans=0;
    unsigned char i;
    unsigned char bit;

    for(i=0; i<8; i++)
    {
        if((y&1)==1)
            ans^=x;
        bit = (x&0x80);
        x<<=1;
        if(bit==0x80)
            x^=0x1b;
        y>>=1;
    }
    return ans;
```

Figure 12: Mix column

4.1.4 Writing Encrypted Content

Then we write the encrypted bytes in binary mode back into the input file thus getting the complete encrypted file.



```c
size_t offset = 0;
   for (size_t j = 0; j < 4; j++)
   {
       for (size_t k = 0; k < 4; k++)
       {
           output[offset++] = state[k][j];
       }
   }
```

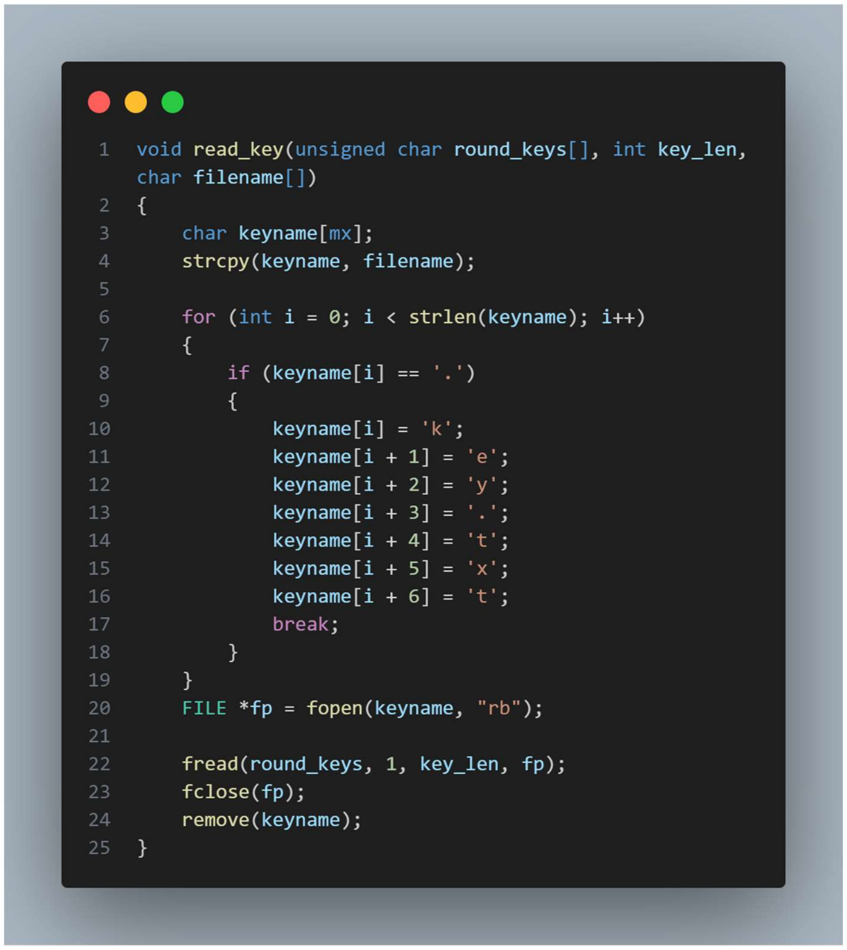
Figure 13: Write back contents

4.2 AES Decryption

The user chooses AES decryption and it starts following operations:

4.2.1 Read Back Key

We read the stored keys used in encryption to use it for decryption.

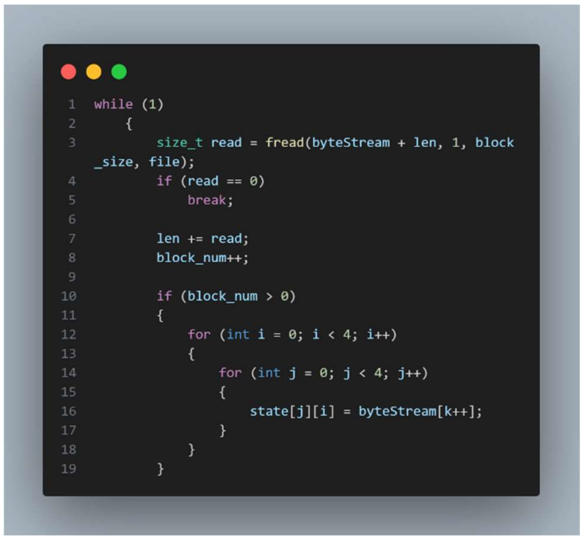


```c
void read_key(unsigned char round_keys[], int key_len,
char filename[])
{
    char keyname[mx];
    strcpy(keyname, filename);

    for (int i = 0; i < strlen(keyname); i++)
    {
        if (keyname[i] == '.')
        {
            keyname[i] = 'k';
            keyname[i + 1] = 'e';
            keyname[i + 2] = 'y';
            keyname[i + 3] = '.';
            keyname[i + 4] = 't';
            keyname[i + 5] = 'x';
            keyname[i + 6] = 't';
            break;
        }
    }
    FILE *fp = fopen(keyname, "rb");

    fread(round_keys, 1, key_len, fp);
    fclose(fp);
    remove(keyname);
}
```

Figure 14: Read keys

4.2.2 Read Encrypted Contents

Here, we read the file to be decrypted in 128 bit blocks to be passed to decryption method.

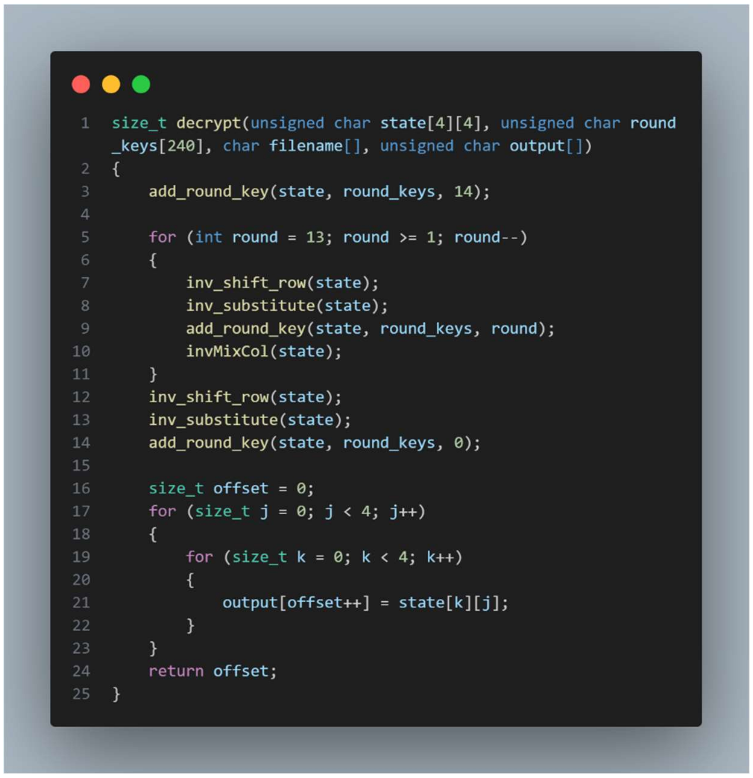


```c
while (1)
    {
        size_t read = fread(byteStream + len, 1, block_size, file);
        if (read == 0)
            break;

        len += read;
        block_num++;

        if (block_num > 0)
        {
            for (int i = 0; i < 4; i++)
            {
                for (int j = 0; j < 4; j++)
                {
                    state[j][i] = byteStream[k++];
                }
            }
        }
```

Figure 15: Read Encrypted File

4.2.3 Decrypt State Array

We pass the state array to the decrypt function to apply inverse sub-byte, inverse shift row, inverse mix column, inverse add round key.

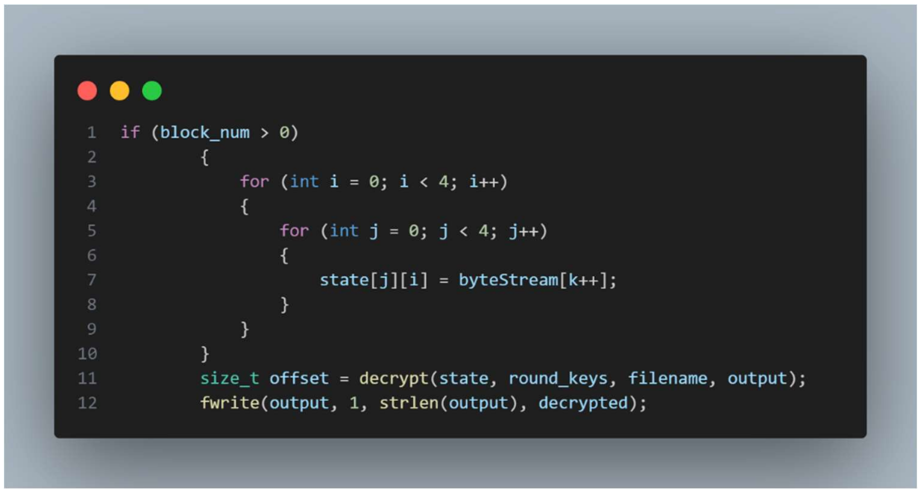


```c
size_t decrypt(unsigned char state[4][4], unsigned char round_keys[240], char filename[], unsigned char output[])
{
    add_round_key(state, round_keys, 14);

    for (int round = 13; round >= 1; round--)
    {
        inv_shift_row(state);
        inv_substitute(state);
        add_round_key(state, round_keys, round);
        invMixCol(state);
    }
    inv_shift_row(state);
    inv_substitute(state);
    add_round_key(state, round_keys, 0);

    size_t offset = 0;
    for (size_t j = 0; j < 4; j++)
    {
        for (size_t k = 0; k < 4; k++)
        {
            output[offset++] = state[k][j];
        }
    }
    return offset;
}
```

Figure 16: Decryption in 14 rounds

4.2.4 Write Back File

After decryption, we take the state array to a bytestream and write it to the original file getting the previous original file.



```c
if (block_num > 0)
        {
            for (int i = 0; i < 4; i++)
            {
                for (int j = 0; j < 4; j++)
                {
                    state[j][i] = byteStream[k++];
                }
            }
        }
        size_t offset = decrypt(state, round_keys, filename, output);
        fwrite(output, 1, strlen(output), decrypted);
```

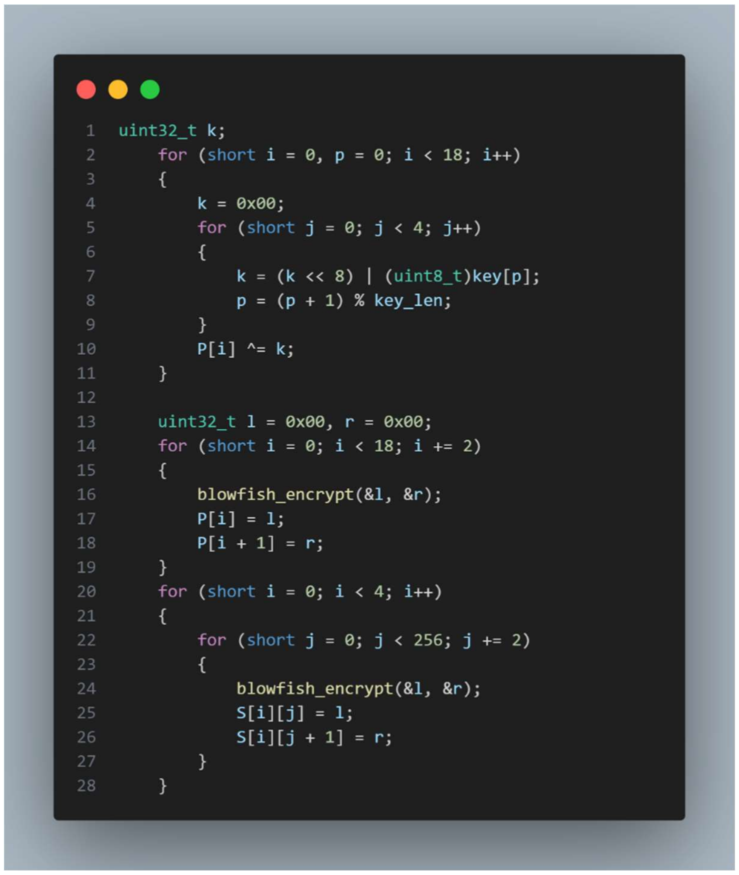
Figure 17: Write back file

4.3 BlowFish Encryption

Here we discuss the process of blowfish encryption algorithm and how it works.

4.3.1 Key Expand

The P array denoting the keys get expanded by the key expansion algorithm also initializing the 4 Substitution box arrays. They are written in file for later decryption.

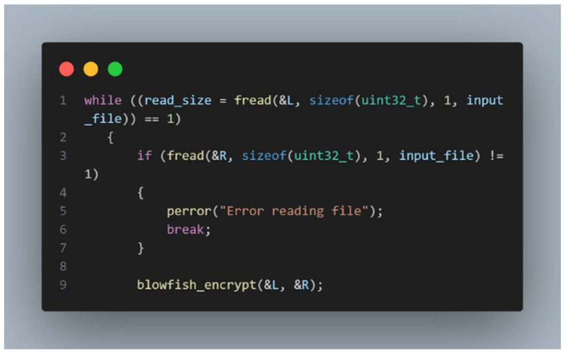


```c
uint32_t k;
    for (short i = 0, p = 0; i < 18; i++)
    {
        k = 0x00;
        for (short j = 0; j < 4; j++)
        {
            k = (k << 8) | (uint8_t)key[p];
            p = (p + 1) % key_len;
        }
        P[i] ^= k;
    }

    uint32_t l = 0x00, r = 0x00;
    for (short i = 0; i < 18; i += 2)
    {
        blowfish_encrypt(&l, &r);
        P[i] = l;
        P[i + 1] = r;
    }
    for (short i = 0; i < 4; i++)
    {
        for (short j = 0; j < 256; j += 2)
        {
            blowfish_encrypt(&l, &r);
            S[i][j] = l;
            S[i][j + 1] = r;
        }
    }
```

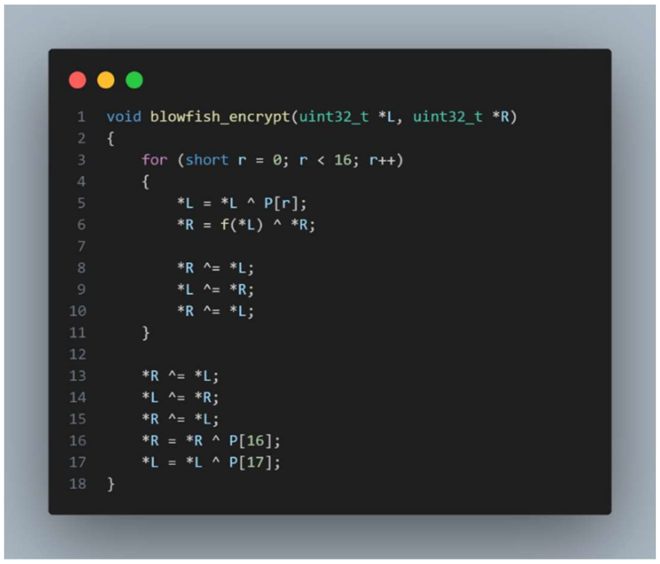
Figure 18: Key Expansion

4.3.2 Read File and Encrypt

We read the user provided file into 2 32bit variables, **L** and **R** and pass them to encrypt function. It then performs L XOR Key, which goes through an **F function.** Then it is XOR-ed with R variable. The whole process is repeated in 18 rounds.



```
while ((read_size = fread(&L, sizeof(uint32_t), 1, input
_file)) == 1)
    {
        if (fread(&R, sizeof(uint32_t), 1, input_file) !=
1)
        {
            perror("Error reading file");
            break;
        }

        blowfish_encrypt(&L, &R);
```

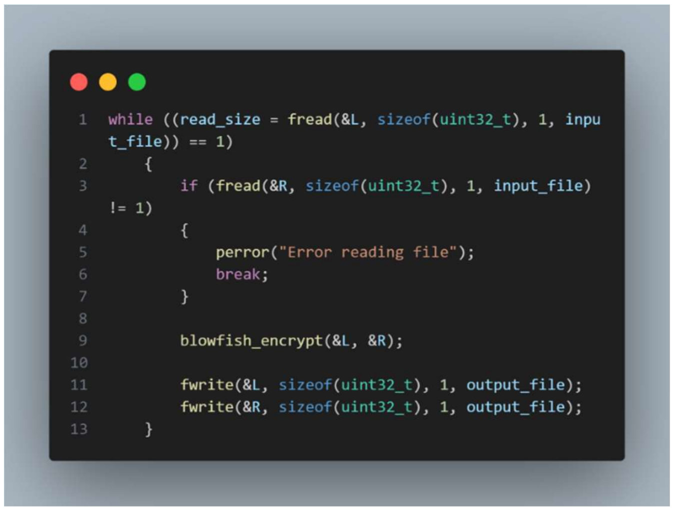


```
void blowfish_encrypt(uint32_t *L, uint32_t *R)
{
    for (short r = 0; r < 16; r++)
    {
        *L = *L ^ P[r];
        *R = f(*L) ^ *R;

        *R ^= *L;
        *L ^= *R;
        *R ^= *L;
    }

    *R ^= *L;
    *L ^= *R;
    *R ^= *L;
    *R = *R ^ P[16];
    *L = *L ^ P[17];
}
```

Figure 19: Reading and Encrypting contents

4.3.3 Write Back Encrypted File

After the encryption, we write them back to the original file as we do in AES.



```
while ((read_size = fread(&L, sizeof(uint32_t), 1, inpu
t_file)) == 1)
    {
        if (fread(&R, sizeof(uint32_t), 1, input_file)
!= 1)
        {
            perror("Error reading file");
            break;
        }

        blowfish_encrypt(&L, &R);

        fwrite(&L, sizeof(uint32_t), 1, output_file);
        fwrite(&R, sizeof(uint32_t), 1, output_file);
    }
```

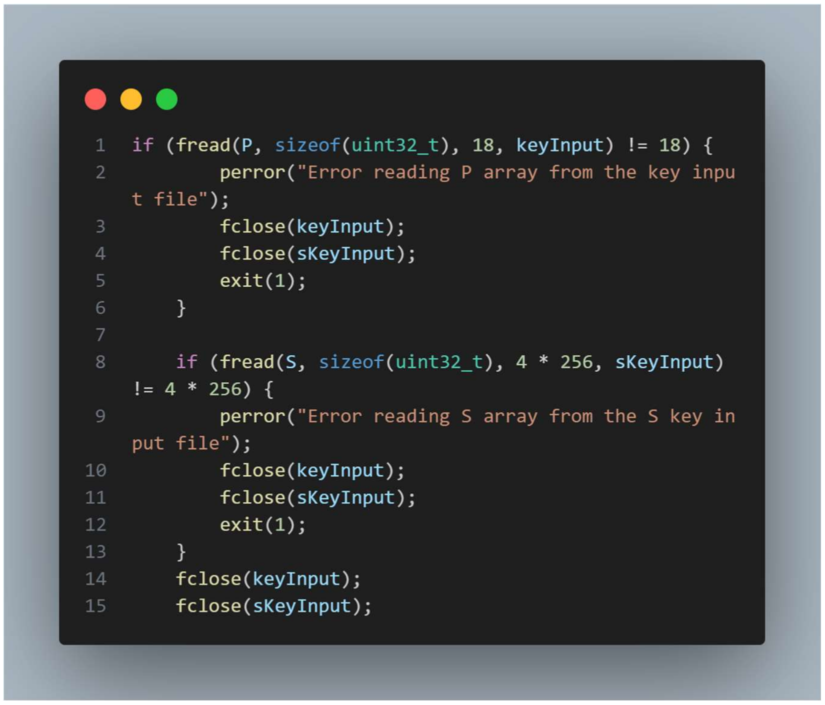
Figure 20: Writing Encrypted file

4.4 BlowFish Decryption

The Decryption reverses the effects on the file by doing the reverse operation again on the encrypted contents on the file.

4.4.1 Read Key

In this step, the stored key and S-boxes are read back to the memory for using in the decryption process.

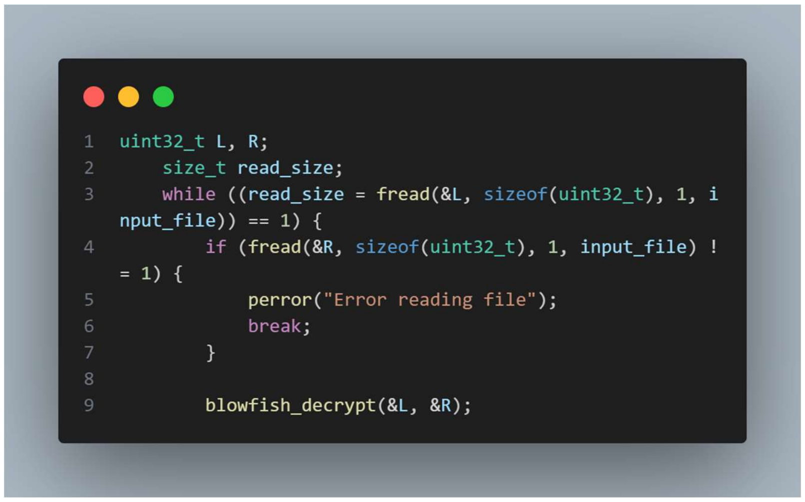


```
if (fread(P, sizeof(uint32_t), 18, keyInput) != 18) {
        perror("Error reading P array from the key input file");
        fclose(keyInput);
        fclose(sKeyInput);
        exit(1);
    }

    if (fread(S, sizeof(uint32_t), 4 * 256, sKeyInput) != 4 * 256) {
        perror("Error reading S array from the S key input file");
        fclose(keyInput);
        fclose(sKeyInput);
        exit(1);
    }
    fclose(keyInput);
    fclose(sKeyInput);
```

Figure 21: Key reading

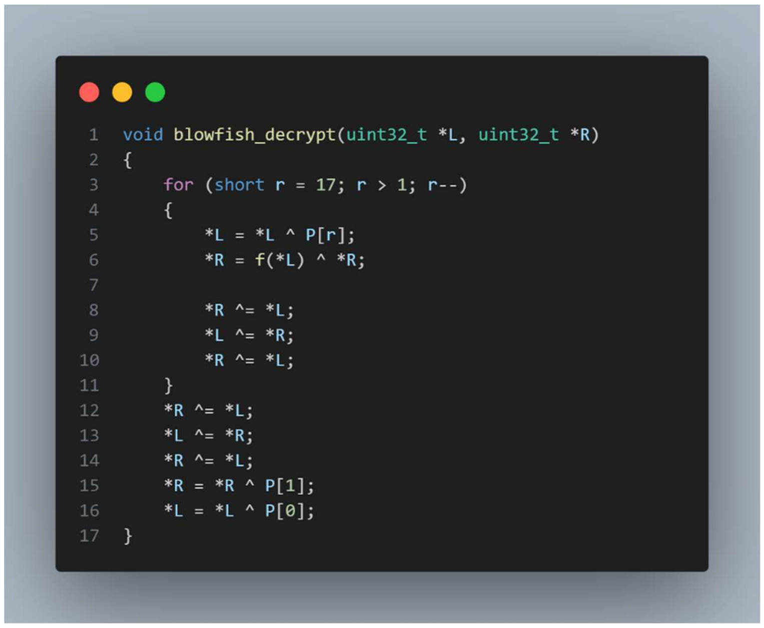
4.4.2 Read File and Decrypt

This step reads the decrypted file in 64 bit block the same way it did in Encryption process and decrypts by reverse XOR-ing the keys.



```
uint32_t L, R;
    size_t read_size;
    while ((read_size = fread(&L, sizeof(uint32_t), 1, input_file)) == 1) {
        if (fread(&R, sizeof(uint32_t), 1, input_file) != 1) {
            perror("Error reading file");
            break;
        }

        blowfish_decrypt(&L, &R);
```

Figure 22: Read and Decrypt blocks

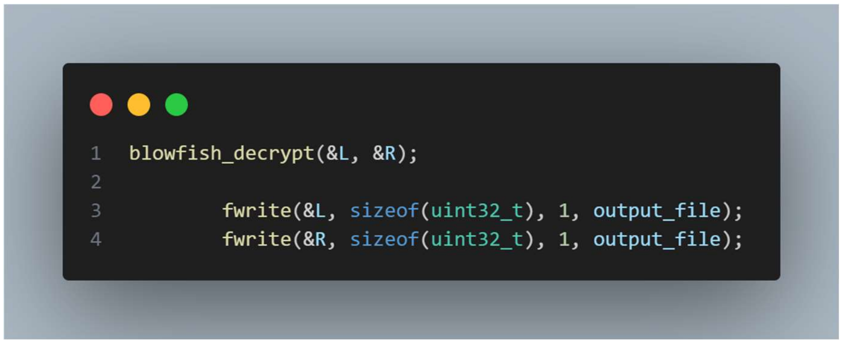


```c
void blowfish_decrypt(uint32_t *L, uint32_t *R)
{
    for (short r = 17; r > 1; r--)
    {
        *L = *L ^ P[r];
        *R = f(*L) ^ *R;

        *R ^= *L;
        *L ^= *R;
        *R ^= *L;
    }
    *R ^= *L;
    *L ^= *R;
    *R ^= *L;
    *R = *R ^ P[1];
    *L = *L ^ P[0];
}
```

Figure 23: Decryption Process

4.4.3 Write Back

After decrypting the blocks, we write them back to the original file to get the decrypted file.



```c
blowfish_decrypt(&L, &R);

        fwrite(&L, sizeof(uint32_t), 1, output_file);
        fwrite(&R, sizeof(uint32_t), 1, output_file);
```

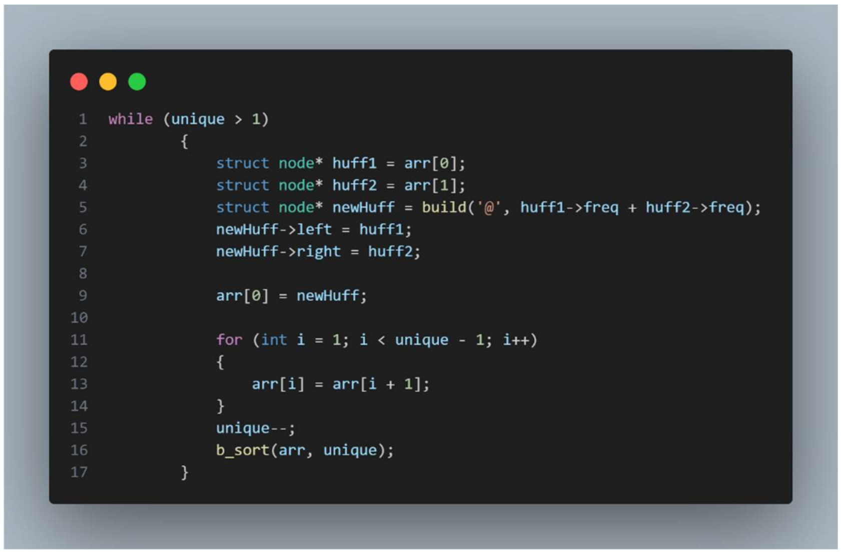
Figure 24: Write File

4.5 Huffman Compression

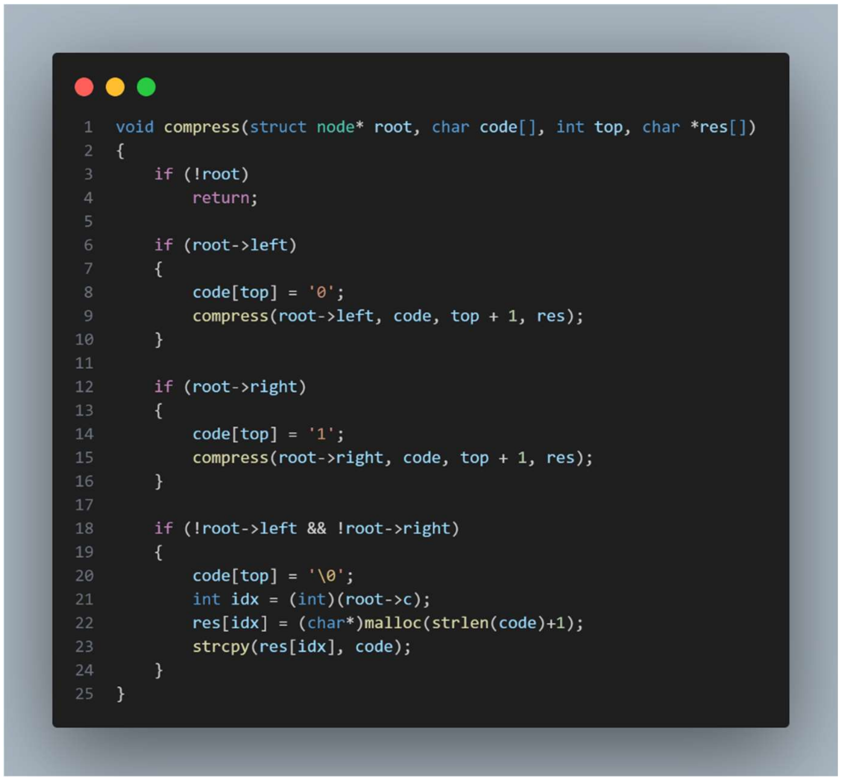
This algorithm takes user’s file to make a Huffman tree based on its content and compress it.

4.5.1 Building Huffman Tree

We count the frequencies of the file’s characters and make a Huffman tree with it. Then we traverse the tree and store the corresponding codes in a result array.



```c
while (unique > 1)
        {
            struct node* huff1 = arr[0];
            struct node* huff2 = arr[1];
            struct node* newHuff = build('@', huff1->freq + huff2->freq);
            newHuff->left = huff1;
            newHuff->right = huff2;

            arr[0] = newHuff;

            for (int i = 1; i < unique - 1; i++)
            {
                arr[i] = arr[i + 1];
            }
            unique--;
            b_sort(arr, unique);
        }
```

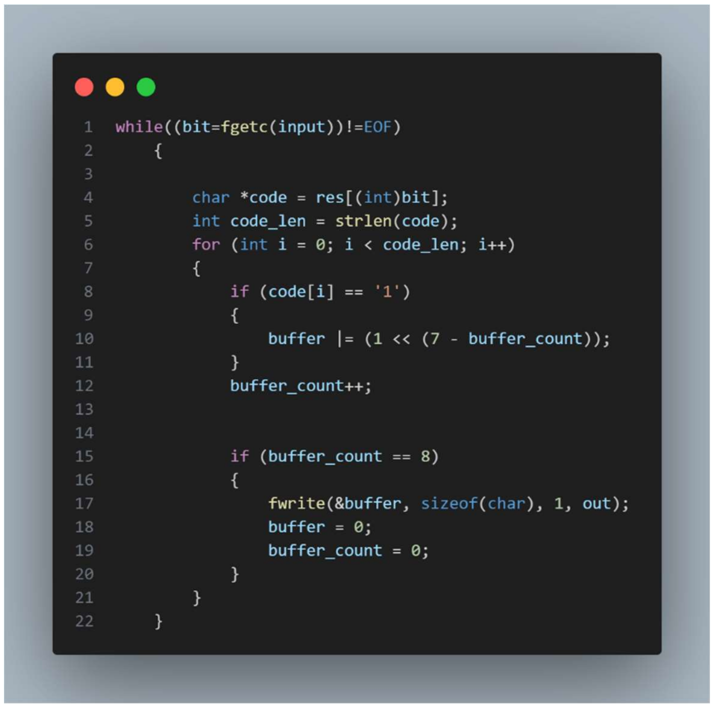


```c
void compress(struct node* root, char code[], int top, char *res[])
{
    if (!root)
        return;

    if (root->left)
    {
        code[top] = '0';
        compress(root->left, code, top + 1, res);
    }

    if (root->right)
    {
        code[top] = '1';
        compress(root->right, code, top + 1, res);
    }

    if (!root->left && !root->right)
    {
        code[top] = '\0';
        int idx = (int)(root->c);
        res[idx] = (char*)malloc(strlen(code)+1);
        strcpy(res[idx], code);
    }
}
```

Figure 25: Building Huffman Tree

4.5.2 Writing Huff-Codes to File

After getting the codes, we again read the file and get the corresponding codes of the read character and write it in a new file which is named as the compressed version of the file.



```c
while((bit=fgetc(input))!=EOF)
    {

        char *code = res[(int)bit];
        int code_len = strlen(code);
        for (int i = 0; i < code_len; i++)
        {
            if (code[i] == '1')
            {
                buffer |= (1 << (7 - buffer_count));
            }
            buffer_count++;


            if (buffer_count == 8)
            {
                fwrite(&buffer, sizeof(char), 1, out);
                buffer = 0;
                buffer_count = 0;
            }
        }
    }
```

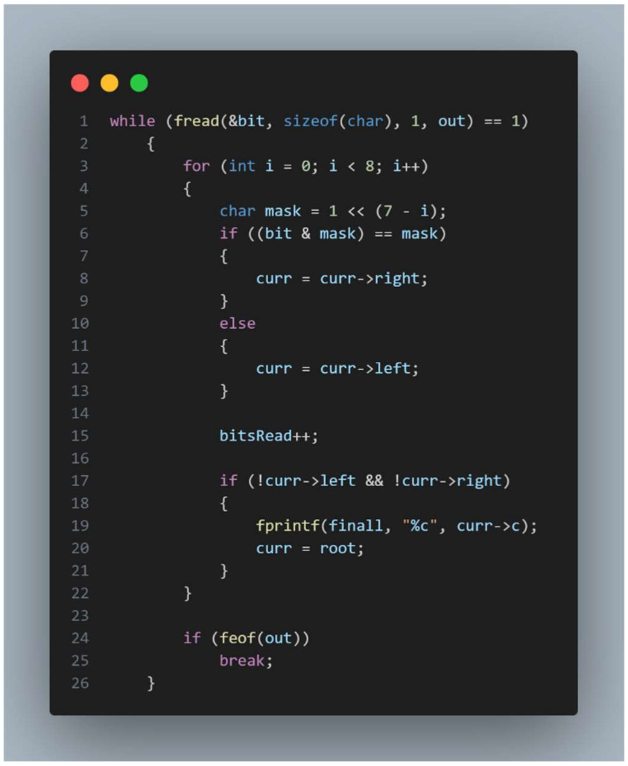
Figure 26: Writing compressed codes

4.6 Huffman Decompression

This step reads the compressed file to generate the decompressed i.e. original file.

4.6.1 Read File and Decompression

In this process, we read the compressed file bit by bit and convert it to the code again, then we traverse the Huffman tree to get its mapped character. We write it in a new file which gets to be decompressed file same as the original file.



```c
while (fread(&bit, sizeof(char), 1, out) == 1)
    {
        for (int i = 0; i < 8; i++)
        {
            char mask = 1 << (7 - i);
            if ((bit & mask) == mask)
            {
                curr = curr->right;
            }
            else
            {
                curr = curr->left;
            }

            bitsRead++;

            if (!curr->left && !curr->right)
            {
                fprintf(finall, "%c", curr->c);
                curr = root;
            }
        }

        if (feof(out))
            break;
    }
```

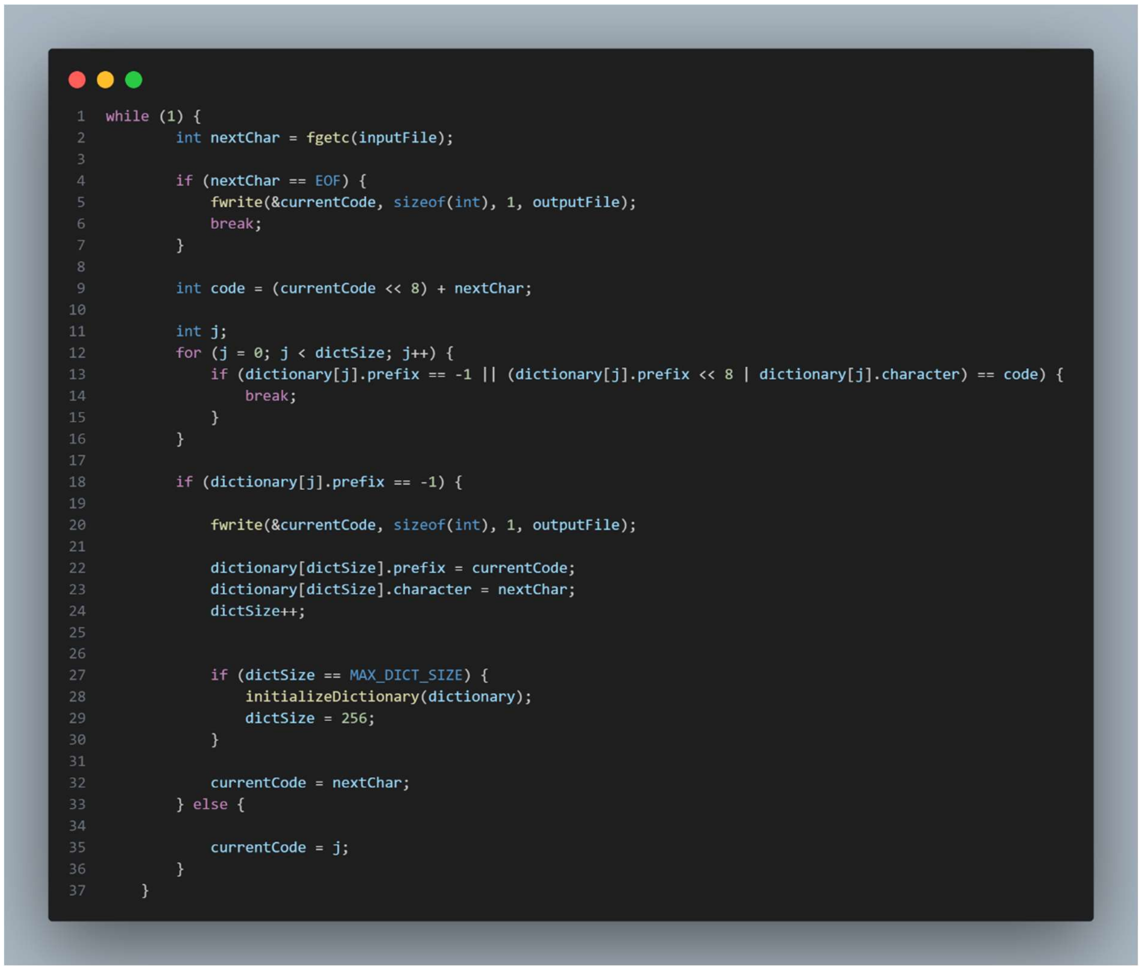
Figure 27: Decompressing codes

4.7 LZW Compression

This algorithm calculates repeated sequences of a file to compress them handling a dictionary structure.

4.7.1 Build Dictionary

We read the input file and keep track of repeated sequences and map them with the dictionary index. If the sequence is already in the dictionary we simple replace the index, else the index gets generated by incrementing dictionary size.

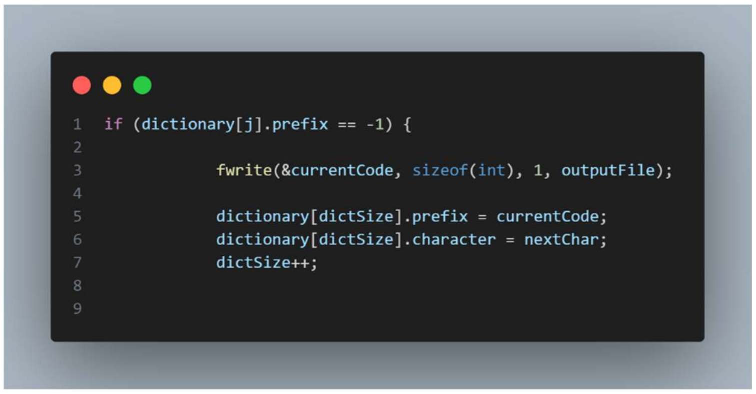


```c
while (1) {
        int nextChar = fgetc(inputFile);

        if (nextChar == EOF) {
            fwrite(&currentCode, sizeof(int), 1, outputFile);
            break;
        }

        int code = (currentCode << 8) + nextChar;

        int j;
        for (j = 0; j < dictSize; j++) {
            if (dictionary[j].prefix == -1 || (dictionary[j].prefix << 8 | dictionary[j].character) == code) {
                break;
            }
        }

        if (dictionary[j].prefix == -1) {

            fwrite(&currentCode, sizeof(int), 1, outputFile);

            dictionary[dictSize].prefix = currentCode;
            dictionary[dictSize].character = nextChar;
            dictSize++;


            if (dictSize == MAX_DICT_SIZE) {
                initializeDictionary(dictionary);
                dictSize = 256;
            }

            currentCode = nextChar;
        } else {

            currentCode = j;
        }
    }
```

Figure 28: Read & Create Dictionary

4.7.2 Generating Compressed File

After building dictionary, we simply replace the characters with their index of dictionary in the new file which gets to be the compressed file.



```c
if (dictionary[j].prefix == -1) {

            fwrite(&currentCode, sizeof(int), 1, outputFile);

            dictionary[dictSize].prefix = currentCode;
            dictionary[dictSize].character = nextChar;
            dictSize++;


```

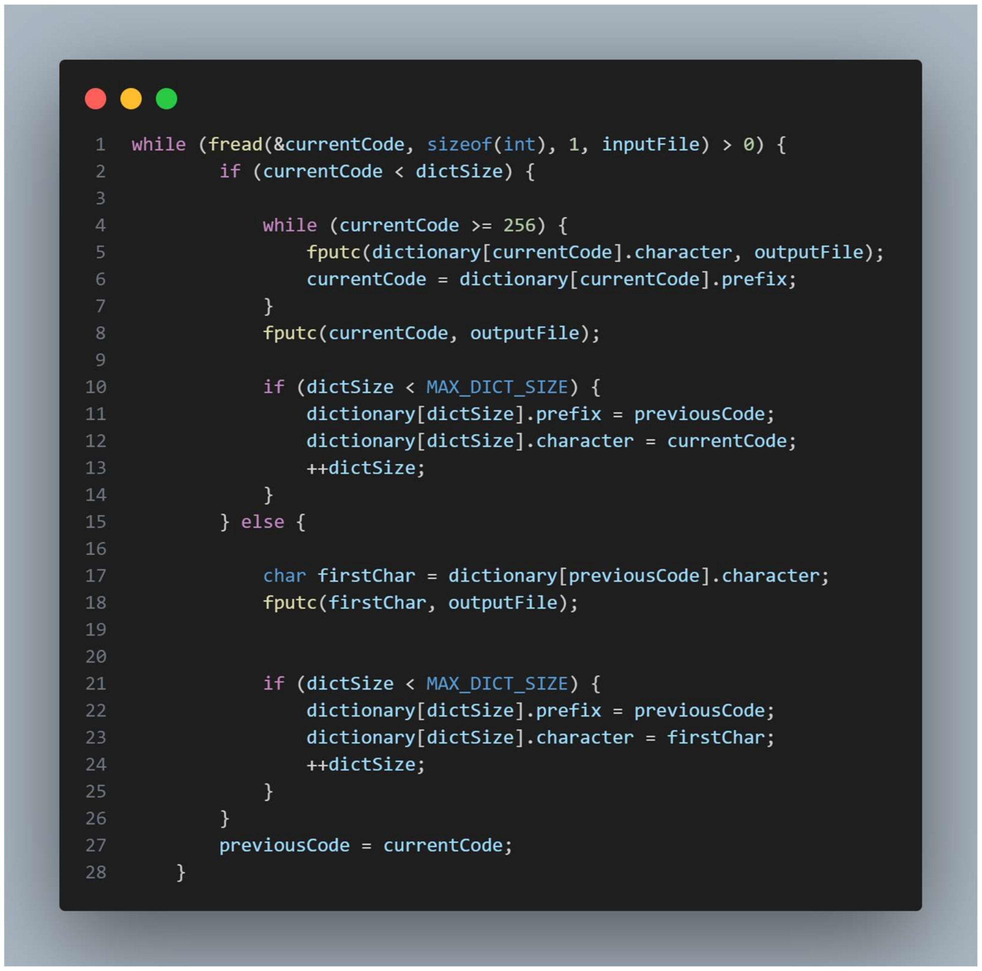
Figure 29: Writing Compressed contents

4.8 LZW Decompressing

This step decompresses by reading compressed file and using dictionary indexes.

4.8.1 Reading Compressed File

We read the compressed file bit by bit and search in the dictionary in the corresponding index. Then we write those characters in a new file which is same as the original file after decompression.

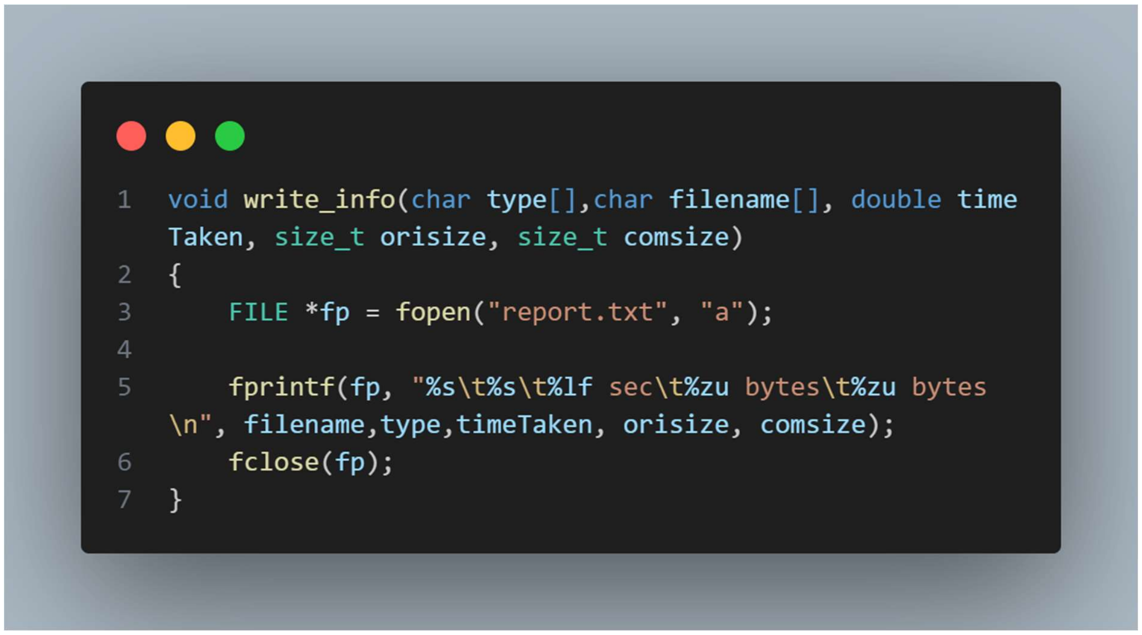


```c
while (fread(&currentCode, sizeof(int), 1, inputFile) > 0) {
        if (currentCode < dictSize) {

            while (currentCode >= 256) {
                fputc(dictionary[currentCode].character, outputFile);
                currentCode = dictionary[currentCode].prefix;
            }
            fputc(currentCode, outputFile);

            if (dictSize < MAX_DICT_SIZE) {
                dictionary[dictSize].prefix = previousCode;
                dictionary[dictSize].character = currentCode;
                ++dictSize;
            }
        } else {

            char firstChar = dictionary[previousCode].character;
            fputc(firstChar, outputFile);


            if (dictSize < MAX_DICT_SIZE) {
                dictionary[dictSize].prefix = previousCode;
                dictionary[dictSize].character = firstChar;
                ++dictSize;
            }
        }
        previousCode = currentCode;
    }
```

Figure 30: Decompression from Dictionary

4.9 Comparison Report

The Encryption, Decryption, Compression and Decompression operations will generate a detailed report about their time taken and file size before/after.

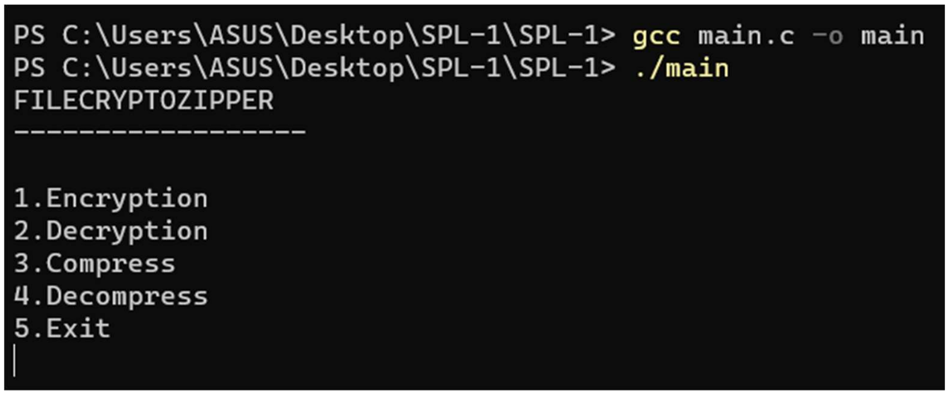


```c
void write_info(char type[],char filename[], double timeTaken, size_t orisize, size_t comsize)
{
    FILE *fp = fopen("report.txt", "a");

    fprintf(fp, "%s\t%s\t%lf sec\t%zu bytes\t%zu bytes\n", filename,type,timeTaken, orisize, comsize);
    fclose(fp);
}
```

Figure 31: Writing Report in File

5. User Interface

When executed, the program will look like following:

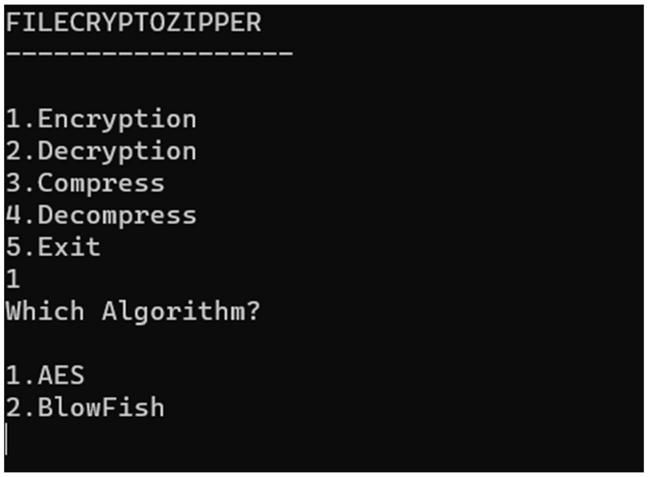


```
PS C:\Users\ASUS\Desktop\SPL-1\SPL-1> gcc main.c -o main
PS C:\Users\ASUS\Desktop\SPL-1\SPL-1> ./main
FILECRYPTOZIPPER
------------------

1.Encryption
2.Decryption
3.Compress
4.Decompress
5.Exit
```

Figure 32: User Interface

5.1 Encryption/Decryption Example

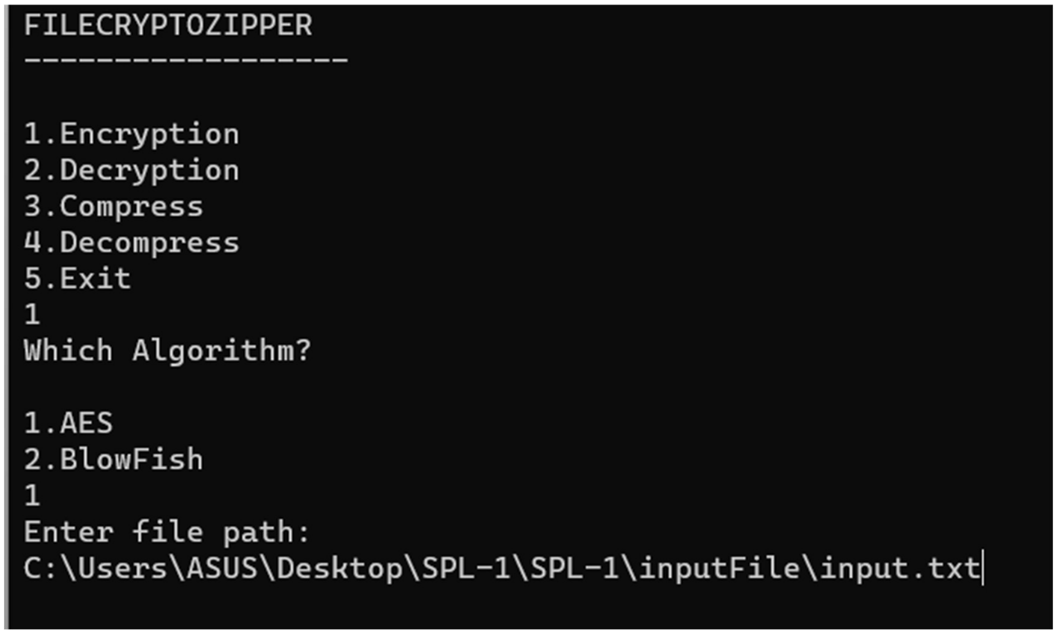
Suppose the user wants to encrypt a text file. So user will press 1 and get prompted for choosing algorithm.



```
FILECRYPTOZIPPER
------------------

1.Encryption
2.Decryption
3.Compress
4.Decompress
5.Exit
1
Which Algorithm?

1.AES
2.BlowFish
```

Figure 33: Encryption UI

We choose 1.AES, then it asks for a filepath where the text file is saved.

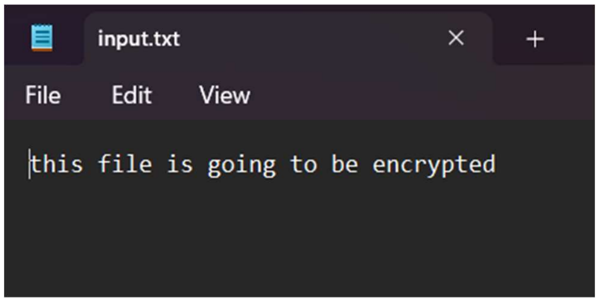


```
FILECRYPTOZIPPER
------------------

1.Encryption
2.Decryption
3.Compress
4.Decompress
5.Exit
1
Which Algorithm?

1.AES
2.BlowFish
1
Enter file path:
C:\Users\ASUS\Desktop\SPL-1\SPL-1\inputFile\input.txt
```

Figure 34: File path Input

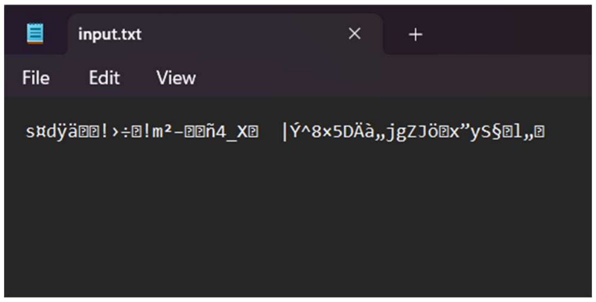
Then the file will be encrypted with the selected algorithm. The file will look like this:

Before Encryption:



```
this file is going to be encrypted
```

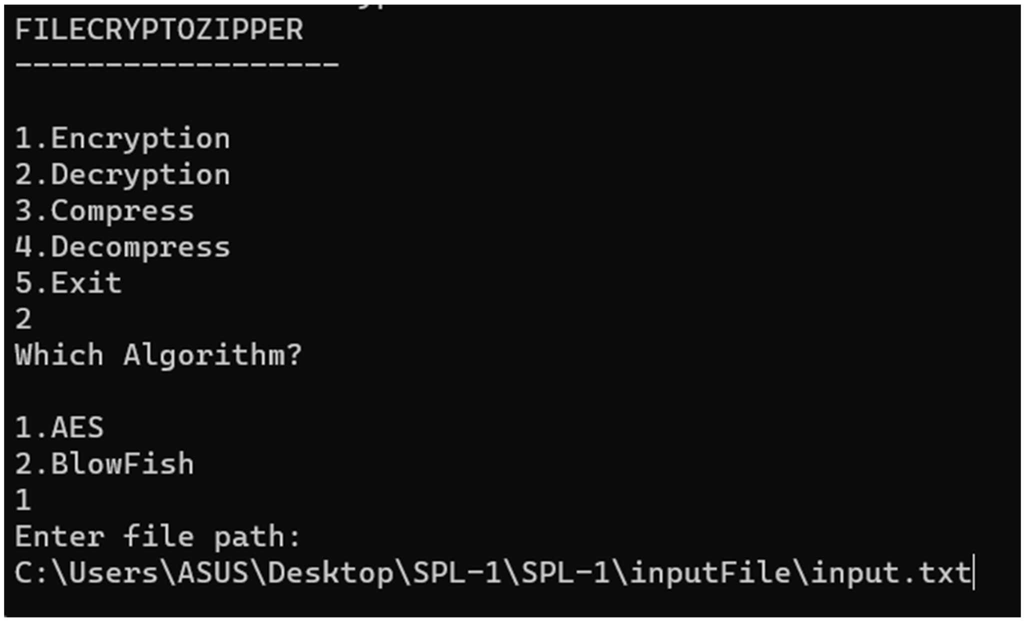
After Encryption



```
s¤dÿä��!›÷�!m²-��ñ4_X�  |Ý^8×5DÄà„jgZJö�x”yS§�l„�
```

Figure 35: Original & Encrypted File

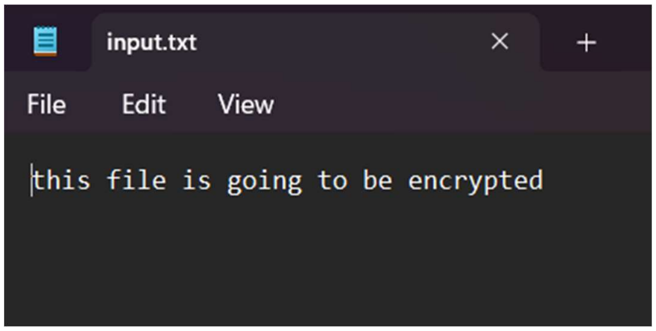
Now, we want to decrypt the file to get original contents. So we will chose option **2.Decryption,** then **1.AES** as we encrypted it with AES. Now we give the file path and hit enter.



```
FILECRYPTOZIPPER
------------------

1.Encryption
2.Decryption
3.Compress
4.Decompress
5.Exit
2
Which Algorithm?

1.AES
2.BlowFish
1
Enter file path:
C:\Users\ASUS\Desktop\SPL-1\SPL-1\inputFile\input.txt
```

Figure 36: Decryption UI

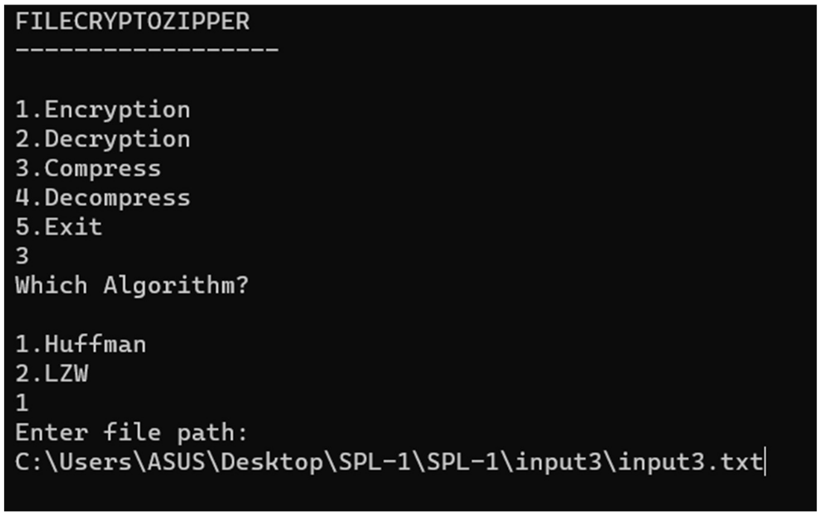
Now the file is decrypted and it is reverted back to the original file.



```
this file is going to be encrypted
```

Figure 37: Decrypted File

5.2 Compression/Decompression Example

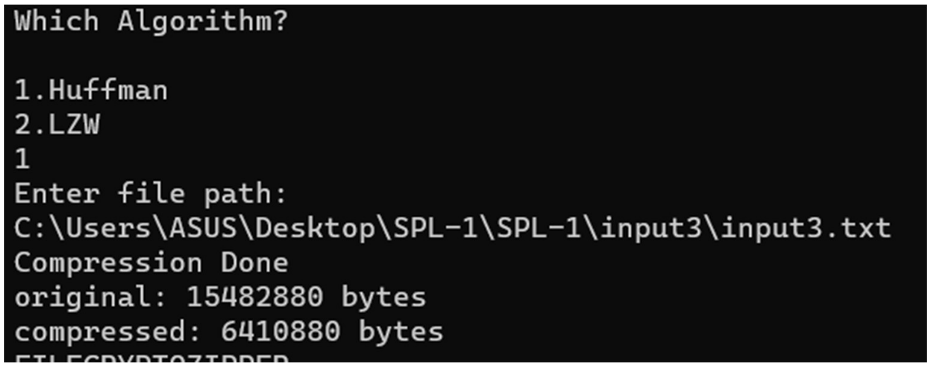


```
FILECRYPTOZIPPER
------------------

1.Encryption
2.Decryption
3.Compress
4.Decompress
5.Exit
3
Which Algorithm?

1.Huffman
2.LZW
1
Enter file path:
C:\Users\ASUS\Desktop\SPL-1\SPL-1\input3\input3.txt
```

Figure 38: Compression UI

The user chooses **3.Compression, 1.Huffman** and then gives the filepath.

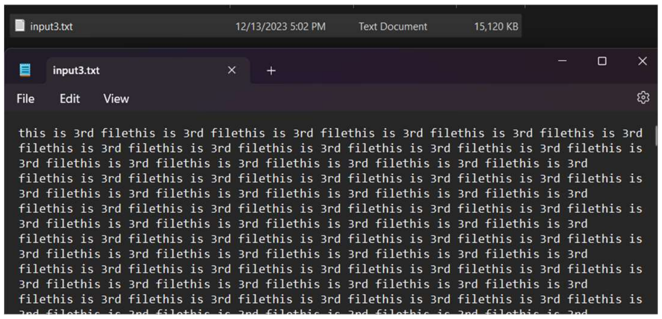
After compression, we can see how much the file size has been reduced.



```
Which Algorithm?

1.Huffman
2.LZW
1
Enter file path:
C:\Users\ASUS\Desktop\SPL-1\SPL-1\input3\input3.txt
Compression Done
original: 15482880 bytes
compressed: 6410880 bytes
```

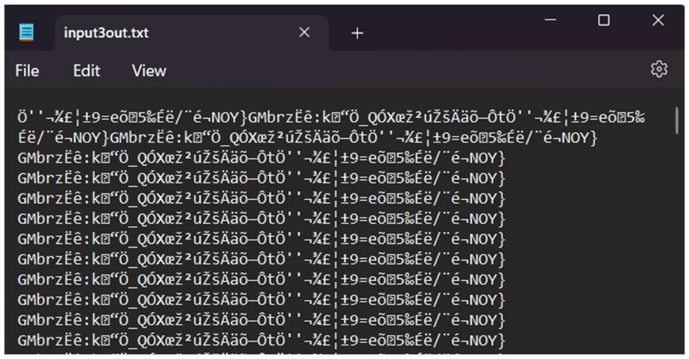
Figure 39: Compression

The file before and after compression looks like following:



```
input3.txt    12/13/2023 5:02 PM    Text Document    15,120 KB
```

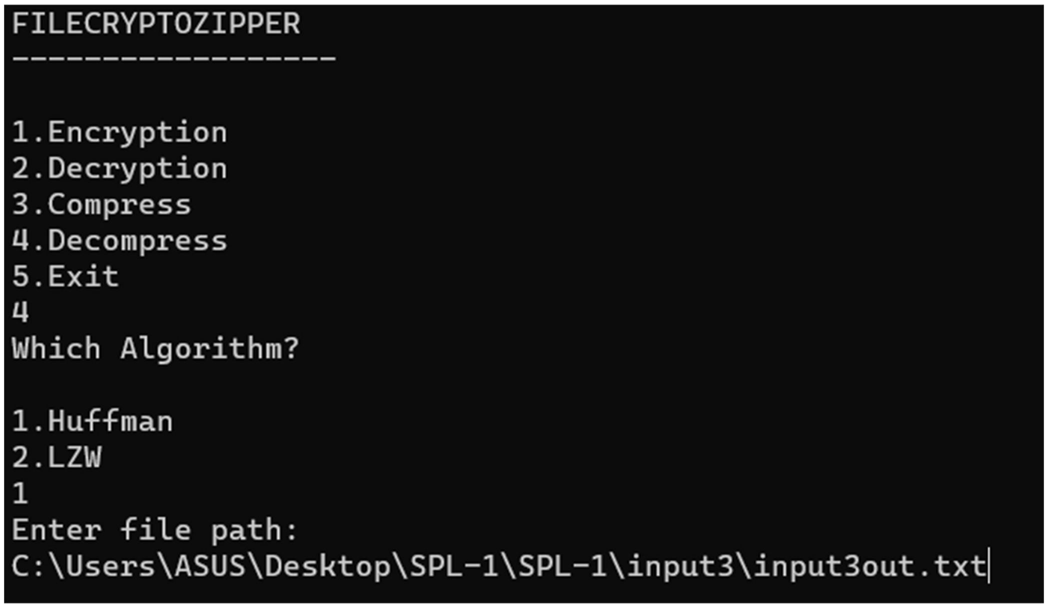
```
this is 3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd
filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is
3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd
filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is
3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd
filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is
3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd
filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is
3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd
filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is
3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd
filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is
```



```
Ö''¬¾£¦±9=eõ5‰Éë/¨é¬NOY}GMbrzËê:k“Ö_QÓXœž²úŽšÄäõ–Ôtö''¬¾£¦±9=eõ5‰
Éë/¨é¬NOY}GMbrzËê:k“Ö_QÓXœž²úŽšÄäõ–Ôtö''¬¾£¦±9=eõ5‰Éë/¨é¬NOY}
GMbrzËê:k“Ö_QÓXœž²úŽšÄäõ–Ôtö''¬¾£¦±9=eõ5‰Éë/¨é¬NOY}
GMbrzËê:k“Ö_QÓXœž²úŽšÄäõ–Ôtö''¬¾£¦±9=eõ5‰Éë/¨é¬NOY}
GMbrzËê:k“Ö_QÓXœž²úŽšÄäõ–Ôtö''¬¾£¦±9=eõ5‰Éë/¨é¬NOY}
GMbrzËê:k“Ö_QÓXœž²úŽšÄäõ–Ôtö''¬¾£¦±9=eõ5‰Éë/¨é¬NOY}
GMbrzËê:k“Ö_QÓXœž²úŽšÄäõ–Ôtö''¬¾£¦±9=eõ5‰Éë/¨é¬NOY}
GMbrzËê:k“Ö_QÓXœž²úŽšÄäõ–Ôtö''¬¾£¦±9=eõ5‰Éë/¨é¬NOY}
GMbrzËê:k“Ö_QÓXœž²úŽšÄäõ–Ôtö''¬¾£¦±9=eõ5‰Éë/¨é¬NOY}
GMbrzËê:k“Ö_QÓXœž²úŽšÄäõ–Ôtö''¬¾£¦±9=eõ5‰Éë/¨é¬NOY}
GMbrzËê:k“Ö_QÓXœž²úŽšÄäõ–Ôtö''¬¾£¦±9=eõ5‰Éë/¨é¬NOY}
GMbrzËê:k“Ö_QÓXœž²úŽšÄäõ–Ôtö''¬¾£¦±9=eõ5‰Éë/¨é¬NOY}
```

Figure 40: Original & Compressed File

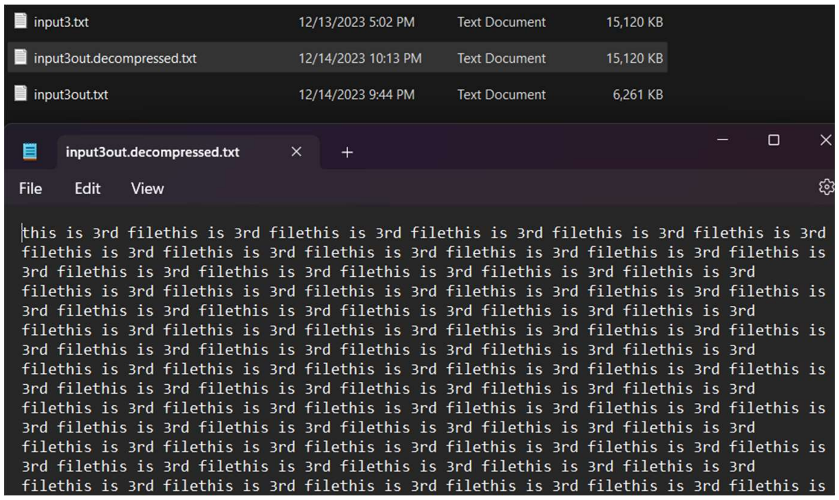
Again, if we want to decompress it, we choose **4.Decompress, 1.Huffman**, then write the filepath.



```
FILECRYPTOZIPPER
-------------------

1.Encryption
2.Decryption
3.Compress
4.Decompress
5.Exit
4
Which Algorithm?

1.Huffman
2.LZW
1
Enter file path:
C:\Users\ASUS\Desktop\SPL-1\SPL-1\input3\input3out.txt
```

Figure 41: Decompression UI

After hitting enter, the decompressed file gets generated as following:



```
input3.txt                     12/13/2023 5:02 PM    Text Document    15,120 KB
input3out.decompressed.txt     12/14/2023 10:13 PM   Text Document    15,120 KB
input3out.txt                  12/14/2023 9:44 PM    Text Document     6,261 KB
```

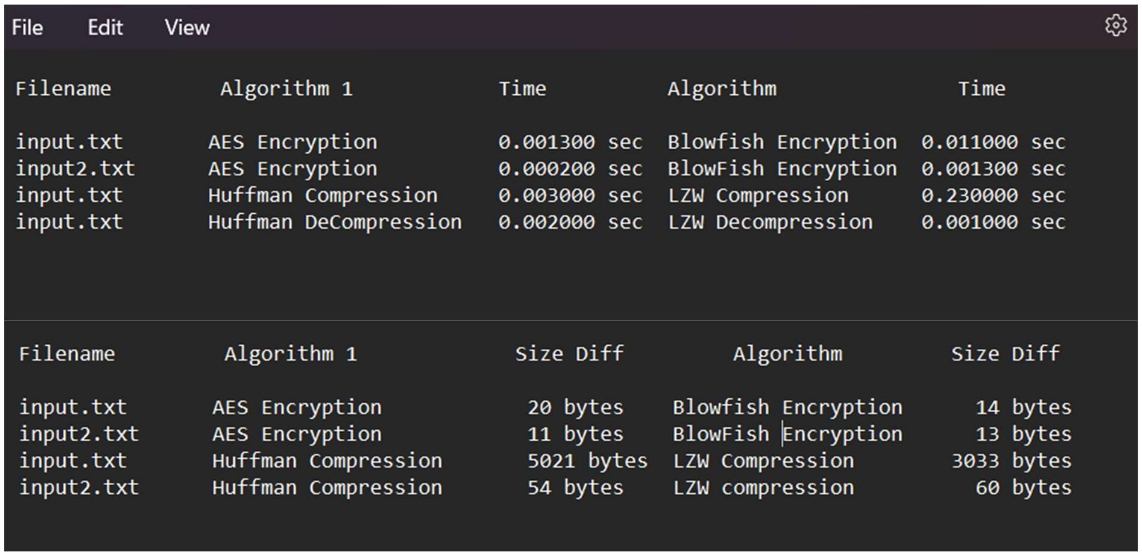
```
this is 3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd
filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is
3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd
filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is
3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd
filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is
3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd
filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is
3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd
filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is
3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd
filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is
3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd
filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is 3rd filethis is
```

Figure 42: Decompressed File

As we see, the decompressed file is same as original file as well as the file size. So no data is lost.

5.3 Comparison Report Sample

The operations done above automatically generate reports of their time taken and file size before/after in a "report.txt" file which will help the user know which algorithm was efficient and secure in those operations and also keeps a track of the work.



```
Filename        Algorithm 1            Time          Algorithm              Time

input.txt       AES Encryption         0.001300 sec  Blowfish Encryption    0.011000 sec
input2.txt      AES Encryption         0.000200 sec  BlowFish Encryption    0.001300 sec
input.txt       Huffman Compression    0.003000 sec  LZW Compression        0.230000 sec
input.txt       Huffman DeCompression  0.002000 sec  LZW Decompression      0.001000 sec

Filename        Algorithm 1            Size Diff     Algorithm              Size Diff

input.txt       AES Encryption          20 bytes     Blowfish Encryption      14 bytes
input2.txt      AES Encryption          11 bytes     BlowFish Encryption      13 bytes
input.txt       Huffman Compression     5021 bytes   LZW Compression        3033 bytes
input2.txt      Huffman Compression     54 bytes     LZW compression          60 bytes
```

Figure 43: Comparison Report

6. Challenges Faced

Doing this project, I faced several issues and difficulties as it is my first academic project. It brought new technical issues as well as code handling skills in the test. The Challenges were:

6.1.Algorithm Complexity:

AES and Blowfish encryption algorithms, as well as Huffman and LZW compression algorithms, was complex to implement correctly. Ensuring the proper understanding and accurate implementation of these algorithms posed a challenge for me as they use various sub-algorithms within it, making it larger and tough to implement correctly.

6.2.Key Management:

Managing encryption keys securely is crucial for the security of the tool. Generating, storing, and handling keys in a secure manner was challenging for me as I had to deal with file read/writing in binary mode which I was not used to.

6.3.Integration of Multiple Algorithms:

Integrating different encryption and compression algorithms seamlessly in a single tool was tricky for me. Ensuring that each component works correctly with others without compromising the integrity of the data is a non-trivial task. I had to make sure that adding another feature does not break or bring a bug to the entire project. So, before implementing anything, I had to brainstorm how I am going to design a generalized form which can handle these 4 algorithms.

6.4.File Format Compatibility:

Different file formats have different structures, and handling various types of files (text, images) correctly was also a test for me. Ensuring that my tool works well with files as well as images, I had to learn how Image(BMP) format works and store pixel data in it and manipulate it likewise which is a bit different that typical text file.

7. Conclusion

In conclusion, FileCryptoZipper represents a successful integration of encryption and compression functionalities into a user-friendly file manipulation tool. Enabling users to encrypt/decrypt with AES or Blowfish and compress/decompress using Huffman and LZW, this project delivers a versatile solution. The thoughtful implementation of security features, along with a simple interface, underscores the commitment to both functionality and user experience. Implementing this project taught me a lot of skills and lessons:

- Handling huge amount of codes in a single program was a first time project for me. I was used to maintaining 100-200 lines of code but this project had around 1500+ lines which made it much harder to keep a track. It taught me to efficiently plan and brainstorm the easier design for handling this big project.

- File I/O operations were not easy for me as I had difficulties understanding it. This project helped me deep dive into details of file operations which made me comfortable in working with files now as this is specifically a file manipulation tool. So I had to master a lot of syntax, logics of file I/O.

- Considering scalability factors, especially when dealing with large files, to ensure that the tool remains efficient and effective as data volumes increase. Primarily it worked with small files, but big data files were causing issues for which I had to optimize the code. It taught me to think of many cases for a program afterwards.

- Bitwise operations were the core for my project. As I had very little knowledge about this, I had to go through many lectures and videos to learn bitwise shifting, XOR operations. It will help me in the future for more reduces complexity of my logics in the code.

Future Works

The project can be extended by various aspects as it is open to implement more algorithms.

- I can modify it to support more file formats like videos, jpg image, pdf, doc or audio files considering that these are common to specific industries to compress or encrypt.
- I can implement it in a cloud storage devices where users can encrypt or compress their files. This could involve API for Google drive or AWS.
- It can be modified to have a password protection where the compression and decompression needs a user given password to give access to it.
- I can implement more Encryption and Compression Algorithms in the project to make it more robust.

References

1. https://www.geeksforgeeks.org/advanced-encryption-standard-aes/ , Geeksforgeeks-> AES Encryption and Decryption, last accessed on: 13/10/2023

2. https://en.wikipedia.org/wiki/Blowfish_(cipher), Wikipedia->Blowfish algorithm, last accessed on: 9/12/2023

3. https://www.geeksforgeeks.org/lzw-lempel-ziv-welch-compression-technique/ geeksforgeeks-> LZW Compression algorithm, last accessed on: 8/12/2023

4. https://www.programiz.com/dsa/huffman-coding, programiz->Huffman Algorithm, last accessed on: 11/09/2023

5. https://gibberlings3.github.io/iesdp/file_formats/ie_formats/bmp.htm, BMP Image format, last accessed on: 11/12/2023