UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Faculty of Engineering, Built Environment and Information Technology

EAS 410

Computer Engineering: Architecture and Systems

Practical 2 Report

Name and Surname	Student Number	Contribution	Signature
Willem A. Fourie	17028002	33	
Michelle Hartman	17090823	33	
Henk Botha	17011176	33	

By submitting this assignment we confirm that we have read and am aware of the University of Pretoria's policy on academic dishonesty and plagiarism and we declare that the work submitted in this assignment is our own as delimited by the mentioned policies. We explicitly declare that no parts of this assignment have been copied from current or previous students' work or any other sources (including the internet), whether copyrighted or not. We understand that we will be subjected to disciplinary actions should it be found that the work we submit here does not comply with the said policies.

Contents

I. AES Files

A. *AES.c*

```c
//
// Created by armandt on 2020/04/07.
//

#include "AES.h"
#include "stdio.h"
#include "math.h"
#include "stdlib.h"
#include "string.h"
#include "CipherModes.h"

int number_of_rounds = -1;
int expanded_key_size = -1;
int key_length = -1;

void set_number_of_rounds(int r){
    number_of_rounds = r;
}

void set_expanded_key_size(int s){
    expanded_key_size = s;
}

void set_key_length(int l){
    key_length = l;
    if(key_length == 128)    {
        number_of_rounds = 9;
        expanded_key_size = 176;
    }
    else if (key_length == 192) {
        number_of_rounds = 11;
        expanded_key_size = 208;
    }
    else if (key_length == 256) {
        number_of_rounds = 13;
        expanded_key_size = 240;
    }
}

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

unsigned char inverse_s_box[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

};
unsigned char multiply_2[] =
    {
        0x00, 0x02, 0x04, 0x06, 0x08, 0x0a, 0x0c, 0x0e, 0
           x10, 0x12, 0x14, 0x16, 0x18, 0x1a, 0x1c, 0x1e,
        0x20, 0x22, 0x24, 0x26, 0x28, 0x2a, 0x2c, 0x2e, 0
           x30, 0x32, 0x34, 0x36, 0x38, 0x3a, 0x3c, 0x3e,
        0x40, 0x42, 0x44, 0x46, 0x48, 0x4a, 0x4c, 0x4e, 0
           x50, 0x52, 0x54, 0x56, 0x58, 0x5a, 0x5c, 0x5e,
        0x60, 0x62, 0x64, 0x66, 0x68, 0x6a, 0x6c, 0x6e, 0
           x70, 0x72, 0x74, 0x76, 0x78, 0x7a, 0x7c, 0x7e,
        0x80, 0x82, 0x84, 0x86, 0x88, 0x8a, 0x8c, 0x8e, 0
           x90, 0x92, 0x94, 0x96, 0x98, 0x9a, 0x9c, 0x9e,
        0xa0, 0xa2, 0xa4, 0xa6, 0xa8, 0xaa, 0xac, 0xae, 0
           xb0, 0xb2, 0xb4, 0xb6, 0xb8, 0xba, 0xbc, 0xbe,
        0xc0, 0xc2, 0xc4, 0xc6, 0xc8, 0xca, 0xcc, 0xce, 0
           xd0, 0xd2, 0xd4, 0xd6, 0xd8, 0xda, 0xdc, 0xde,
        0xe0, 0xe2, 0xe4, 0xe6, 0xe8, 0xea, 0xec, 0xee, 0
           xf0, 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 multiply_3[] =
    {
        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 multiply_9[] =
    {
        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};

unsigned char multiply_11[] =
    {
        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, 0
```

```
            xae, 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};

unsigned char multiply_13[] =
    {
        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, 0
```

```
           xd9, 0xd4, 0xc3, 0xce, 0xed, 0xe0, 0xf7, 0xfa,
        0xb7, 0xba, 0xad, 0xa0, 0x83, 0x8e, 0x99, 0x94, 0
           xdf, 0xd2, 0xc5, 0xc8, 0xeb, 0xe6, 0xf1, 0xfc,
        0x67, 0x6a, 0x7d, 0x70, 0x53, 0x5e, 0x49, 0x44, 0
           x0f, 0x02, 0x15, 0x18, 0x3b, 0x36, 0x21, 0x2c,
        0x0c, 0x01, 0x16, 0x1b, 0x38, 0x35, 0x22, 0x2f, 0
           x64, 0x69, 0x7e, 0x73, 0x50, 0x5d, 0x4a, 0x47,
        0xdc, 0xd1, 0xc6, 0xcb, 0xe8, 0xe5, 0xf2, 0xff, 0
           xb4, 0xb9, 0xae, 0xa3, 0x80, 0x8d, 0x9a, 0x97};

unsigned char multiply_14[] =
    {
        0x00, 0x0e, 0x1c, 0x12, 0x38, 0x36, 0x24, 0x2a, 0
           x70, 0x7e, 0x6c, 0x62, 0x48, 0x46, 0x54, 0x5a,
        0xe0, 0xee, 0xfc, 0xf2, 0xd8, 0xd6, 0xc4, 0xca, 0
           x90, 0x9e, 0x8c, 0x82, 0xa8, 0xa6, 0xb4, 0xba,
        0xdb, 0xd5, 0xc7, 0xc9, 0xe3, 0xed, 0xff, 0xf1, 0
           xab, 0xa5, 0xb7, 0xb9, 0x93, 0x9d, 0x8f, 0x81,
        0x3b, 0x35, 0x27, 0x29, 0x03, 0x0d, 0x1f, 0x11, 0
           x4b, 0x45, 0x57, 0x59, 0x73, 0x7d, 0x6f, 0x61,
        0xad, 0xa3, 0xb1, 0xbf, 0x95, 0x9b, 0x89, 0x87, 0
           xdd, 0xd3, 0xc1, 0xcf, 0xe5, 0xeb, 0xf9, 0xf7,
        0x4d, 0x43, 0x51, 0x5f, 0x75, 0x7b, 0x69, 0x67, 0
           x3d, 0x33, 0x21, 0x2f, 0x05, 0x0b, 0x19, 0x17,
        0x76, 0x78, 0x6a, 0x64, 0x4e, 0x40, 0x52, 0x5c, 0
           x06, 0x08, 0x1a, 0x14, 0x3e, 0x30, 0x22, 0x2c,
        0x96, 0x98, 0x8a, 0x84, 0xae, 0xa0, 0xb2, 0xbc, 0
           xe6, 0xe8, 0xfa, 0xf4, 0xde, 0xd0, 0xc2, 0xcc,
        0x41, 0x4f, 0x5d, 0x53, 0x79, 0x77, 0x65, 0x6b, 0
           x31, 0x3f, 0x2d, 0x23, 0x09, 0x07, 0x15, 0x1b,
        0xa1, 0xaf, 0xbd, 0xb3, 0x99, 0x97, 0x85, 0x8b, 0
           xd1, 0xdf, 0xcd, 0xc3, 0xe9, 0xe7, 0xf5, 0xfb,
        0x9a, 0x94, 0x86, 0x88, 0xa2, 0xac, 0xbe, 0xb0, 0
           xea, 0xe4, 0xf6, 0xf8, 0xd2, 0xdc, 0xce, 0xc0,
        0x7a, 0x74, 0x66, 0x68, 0x42, 0x4c, 0x5e, 0x50, 0
           x0a, 0x04, 0x16, 0x18, 0x32, 0x3c, 0x2e, 0x20,
        0xec, 0xe2, 0xf0, 0xfe, 0xd4, 0xda, 0xc8, 0xc6, 0
           x9c, 0x92, 0x80, 0x8e, 0xa4, 0xaa, 0xb8, 0xb6,
        0x0c, 0x02, 0x10, 0x1e, 0x34, 0x3a, 0x28, 0x26, 0
           x7c, 0x72, 0x60, 0x6e, 0x44, 0x4a, 0x58, 0x56,
        0x37, 0x39, 0x2b, 0x25, 0x0f, 0x01, 0x13, 0x1d, 0
           x47, 0x49, 0x5b, 0x55, 0x7f, 0x71, 0x63, 0x6d,
        0xd7, 0xd9, 0xcb, 0xc5, 0xef, 0xe1, 0xf3, 0xfd, 0
           xa7, 0xa9, 0xbb, 0xb5, 0x9f, 0x91, 0x83, 0x8d};

unsigned char RCon[11] =
        {
                0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0
                   x40, 0x80, 0x1b, 0x36
        };

void key_expansion_core(unsigned char *in, unsigned char i)
{
    //rotate left
    unsigned char t = in[0];
    in[0] = in[1];
    in[1] = in[2];
    in[2] = in[3];
    in[3] = t;

    //S-box on all four bytes
    in[0] = s_box[in[0]];
    in[1] = s_box[in[1]];
    in[2] = s_box[in[2]];
    in[3] = s_box[in[3]];

    //RCon operation
    in[0] ^= RCon[i];
}

void key_expansion(unsigned char *input_key, unsigned char *expanded_key)
{
    //The first 16 bytes of the expanded key are simply the encryption key that the user entered.
    for (int i = 0; i < (key_length / 8); i++)
        expanded_key[i] = input_key[i];

    // Variables
    int bytes_generated = key_length / 8;
    int RCon_iteration = 1;
    unsigned char temp[4];

    while (bytes_generated < expanded_key_size)
    {
        // Assign previous four bytes in the expanded key to temp
        for (int i = 0; i < 4; i++)
            temp[i] = expanded_key[i + bytes_generated - 4];

        //Send t to the core key scheduler along with the RCon value.
        if (bytes_generated % 16 == 0)
            key_expansion_core(temp, RCon_iteration++);

        /*XOR the output of the core key scheduler with a four-byte block 16 bytes before the
```

```c
        expanded key (i.e bytes 0-3). The result becomes the next 4 bytes of the expanded key.*/
        for (unsigned char i = 0; i < 4; i++)
        {
            expanded_key[bytes_generated] = expanded_key[bytes_generated - 16] ^ temp[i];
            bytes_generated++;
        }
    }
}

void sub_bytes(unsigned char *state)
{
    for (int i = 0; i < 16; i++)
        state[i] = s_box[state[i]];
}
void inverse_sub_bytes(unsigned char *state)
{
    for (int i = 0; i < 16; i++)
        state[i] = inverse_s_box[state[i]];
}

void shift_rows(unsigned char *state)
{
    unsigned char tmp[16];

    tmp[0] = state[0];
    tmp[1] = state[5];
    tmp[2] = state[10];
    tmp[3] = state[15];

    tmp[4] = state[4];
    tmp[5] = state[9];
    tmp[6] = state[14];
    tmp[7] = state[3];

    tmp[8] = state[8];
    tmp[9] = state[13];
    tmp[10] = state[2];
    tmp[11] = state[7];

    tmp[12] = state[12];
    tmp[13] = state[1];
    tmp[14] = state[6];
    tmp[15] = state[11];

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

```c
void inverse_shift_rows(unsigned char *state)
{
    unsigned char tmp[16];

    tmp[0] = state[0];
    tmp[5] = state[1];
    tmp[10] = state[2];
    tmp[15] = state[3];

    tmp[4] = state[4];
    tmp[9] = state[5];
    tmp[14] = state[6];
    tmp[3] = state[7];

    tmp[8] = state[8];
    tmp[13] = state[9];
    tmp[2] = state[10];
    tmp[7] = state[11];

    tmp[12] = state[12];
    tmp[1] = state[13];
    tmp[6] = state[14];
    tmp[11] = state[15];

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

void mix_columns(unsigned char *state)
{
    unsigned char tmp[16];

    tmp[0] = (unsigned char)(multiply_2[state[0]] ^ multiply_3[state[1]] ^ state[2] ^ state[3]);
    tmp[1] = (unsigned char)(state[0] ^ multiply_2[state[1]] ^ multiply_3[state[2]] ^ state[3]);
    tmp[2] = (unsigned char)(state[0] ^ state[1] ^ multiply_2[state[2]] ^ multiply_3[state[3]]);
    tmp[3] = (unsigned char)(multiply_3[state[0]] ^ state[1] ^ state[2] ^ multiply_2[state[3]]);

    tmp[4] = (unsigned char)(multiply_2[state[4]] ^ multiply_3[state[5]] ^ state[6] ^ state[7]);
    tmp[5] = (unsigned char)(state[4] ^ multiply_2[state[5]] ^ multiply_3[state[6]] ^ state[7]);
    tmp[6] = (unsigned char)(state[4] ^ state[5] ^ multiply_2[state[6]] ^ multiply_3[state[7]]);
    tmp[7] = (unsigned char)(multiply_3[state[4]] ^ state[5] ^ state[6] ^ multiply_2[state[7]]);

    tmp[8] = (unsigned char)(multiply_2[state[8]] ^ multiply_3[state[9]] ^ state[10] ^ state[11]);
    tmp[9] = (unsigned char)(state[8] ^ multiply_2[state[9]] ^ multiply_3[state[10]] ^ state[11]);
    tmp[10] = (unsigned char)(state[8] ^ state[9] ^ multiply_2[state[10]] ^ multiply_3[state[11]]);
    tmp[11] = (unsigned char)(multiply_3[state[8]] ^ state[9] ^ state[10] ^ multiply_2[state[11]]);

    tmp[12] = (unsigned char)(multiply_2[state[12]] ^ multiply_3[state[13]] ^ state[14] ^ state[15]);
    tmp[13] = (unsigned char)(state[12] ^ multiply_2[state[13]] ^ multiply_3[state[14]] ^ state[15]);
    tmp[14] = (unsigned char)(state[12] ^ state[13] ^ multiply_2[state[14]] ^ multiply_3[state[15]]);
    tmp[15] = (unsigned char)(multiply_3[state[12]] ^ state[13] ^ state[14] ^ multiply_2[state[15]]);
    tmp[16] = '\0';

    for (int i = 0; i < 17; i++)
        state[i] = tmp[i];
}

void inverse_mix_columns(unsigned char *state)
{
    unsigned char tmp[16];

    tmp[0] = (unsigned char)(multiply_14[state[0]] ^ multiply_9[state[3]] ^ multiply_13[state[2]] ^ multiply_11[state[1]]);
    tmp[1] = (unsigned char)(multiply_14[state[1]] ^ multiply_9[state[0]] ^ multiply_13[state[3]] ^ multiply_11[state[2]]);
    tmp[2] = (unsigned char)(multiply_14[state[2]] ^ multiply_9[state[1]] ^ multiply_13[state[0]] ^ multiply_11[state[3]]);
    tmp[3] = (unsigned char)(multiply_14[state[3]] ^ multiply_9[state[2]] ^ multiply_13[state[1]] ^ multiply_11[state[0]]);

    tmp[4] = (unsigned char)(multiply_14[state[4]] ^ multiply_9[state[7]] ^ multiply_13[state[6]] ^ multiply_11[state[5]]);
    tmp[5] = (unsigned char)(multiply_14[state[5]] ^ multiply_9[state[4]] ^ multiply_13[state[7]] ^ multiply_11[state[6]]);
    tmp[6] = (unsigned char)(multiply_14[state[6]] ^ multiply_9[state[5]] ^ multiply_13[state[4]] ^
        multiply_11[state[7]]);
    tmp[7] = (unsigned char)(multiply_14[state[7]] ^ multiply_9[state[6]] ^ multiply_13[state[5]] ^ multiply_11[state[4]]);

    tmp[8] = (unsigned char)(multiply_14[state[8]] ^ multiply_9[state[11]] ^ multiply_13[state[10]] ^ multiply_11[state[9]]);
    tmp[9] = (unsigned char)(multiply_14[state[9]] ^ multiply_9[state[8]] ^ multiply_13[state[11]] ^ multiply_11[state[10]]);
    tmp[10] = (unsigned char)(multiply_14[state[10]] ^ multiply_9[state[9]] ^ multiply_13[state[8]] ^ multiply_11[state[11]]);
    tmp[11] = (unsigned char)(multiply_14[state[11]] ^ multiply_9[state[10]] ^ multiply_13[state[9]] ^ multiply_11[state[8]]);

    tmp[12] = (unsigned char)(multiply_14[state[12]] ^ multiply_9[state[15]] ^ multiply_13[state[14]] ^ multiply_11[state[13]]);
    tmp[13] = (unsigned char)(multiply_14[state[13]] ^ multiply_9[state[12]] ^ multiply_13[state[15]] ^ multiply_11[state[14]]);
    tmp[14] = (unsigned char)(multiply_14[state[14]] ^ multiply_9[state[13]] ^ multiply_13[state[12]] ^ multiply_11[state[15]]);
    tmp[15] = (unsigned char)(multiply_14[state[15]] ^ multiply_9[state[14]] ^ multiply_13[state[13]] ^ multiply_11[state[12]]);
    tmp[16] = '\0';

    for (int i = 0; i < 17; i++)
        state[i] = tmp[i];
}

void add_round_key(unsigned char *state, unsigned char *round_key)
{
    for (int i = 0; i < 16; i++)
        state[i] ^= round_key[i];
}
void AES_encrypt(unsigned char *message, unsigned char *key)
{
    unsigned char state[16];
    for (int i = 0; i < 16; i++)
        state[i] = message[i];

    // int number_of_rounds = 9;

    unsigned char expanded_key[expanded_key_size];
    key_expansion(key, expanded_key);

    //Initial round
    add_round_key(state, key);

    //Mixing rounds
    for (int i = 0; i < number_of_rounds; i++)
    {
        sub_bytes(state);
        shift_rows(state);
        mix_columns(state);
        add_round_key(state, expanded_key + (16 * (i + 1)));
    }

    //Final round
    sub_bytes(state);
    shift_rows(state);
    add_round_key(state, expanded_key + expanded_key_size - 16);

    for (int i = 0; i < 16; i++)
        message[i] = state[i];
    message[17] = '\0';
}

void AES_decrypt(unsigned char *message, unsigned char *key)
{
    unsigned char state[16];
    for (int i = 0; i < 16; i++)
        state[i] = message[i];

    //int number_of_rounds = 9;

    unsigned char expanded_key[expanded_key_size];
    key_expansion(key, expanded_key);

    //Initial round
    add_round_key(state, expanded_key + expanded_key_size - 16);

    //Mixing rounds
    for (int i = 0; i < number_of_rounds; i++)
    {
        inverse_shift_rows(state);
        inverse_sub_bytes(state);
        add_round_key(state, expanded_key + expanded_key_size - 16 - (16 * (i + 1)));
        inverse_mix_columns(state);
    }

    //Final round
    inverse_shift_rows(state);
    inverse_sub_bytes(state);
    add_round_key(state, key);

    for (int i = 0; i < 16; i++)
        message[i] = state[i];
    message[17] = '\0';
}

void print_hex(const unsigned char *string, int count)
{
    unsigned char *p = (unsigned char *)string;

    for (int i = 0; i < count; ++i)
    {
        if (!(i % 16) && i)
            printf("\n");

        printf("%02x␣", p[i]);
    }
    printf("\n\n");
}

void print_hex_block(const char *string)
{
    unsigned char *p = (unsigned char *)string;

    for (int i = 0; i < 4; ++i)
    {
        int x = 0;
        for (int j = 0; j < 4; j++)
        {
            //              if (! (i % 16) && i)
            //                  printf("\n");
            x = j * 4;

            printf("%02x␣", p[i + x]);
        }
        printf("\n");
    }
    printf("\n\n");
}
```

```c

void test_functionality(unsigned char *input_string, int key_length, unsigned char* key)
{
    printf("\n___________________________________________________________\n");
    unsigned char input[strlen(input_string)];
    strncpy(input, input_string, strlen(input_string));
    print_hex_block(input_string);

    printf("\nMix␣Columns\n");
    printf("___________________________________________________________\n");

    mix_columns(input_string);
    print_hex_block(input_string);
    printf("___________________________________________________________\n");

    printf("\nShift␣rows\n");
    strncpy(input_string, input, strlen(input_string));
    printf("___________________________________________________________\n");
    shift_rows(input_string);
    print_hex_block(input_string);
    printf("___________________________________________________________\n");

    printf("\nSub␣Bytes\n");
    strncpy(input_string, input, strlen(input_string));
    printf("___________________________________________________________\n");
    sub_bytes(input_string);
    print_hex_block(input_string);
    printf("___________________________________________________________\n");

    printf("\nExpanded␣key\n");
    strncpy(input_string, input, strlen(input_string));
    printf("___________________________________________________________\n");
    unsigned char expanded_key[expanded_key_size];
    set_key_length(key_length);
    key_expansion(key, expanded_key);
    print_hex(expanded_key, expanded_key_size);
    printf("______________________________________________________\n");
}


unsigned char* pad_and_encrypt(unsigned char * message, unsigned char * encrypted, int message_len, int key_len, unsigned char * k){
    unsigned char original_message[message_len + 1];     //make a copy of the message (maybe this helps)
    unsigned char key[key_len / 8];     //make a copy of the key

    for (int a = 0; a < key_len / 8; a++){
        key[a] = k[a];
    }

    for (int a = 0; a < message_len; a++){
        original_message[a] = message[a];
    }
    original_message[message_len] = '\0';

    set_key_length(key_len);

    //now do the padding
    int padded_message_len = message_len;
    if (message_len % 16 != 0){
        padded_message_len = (padded_message_len / 16 + 1) * 16;
    }

    unsigned char padded_message[padded_message_len + 1];
    for (int a = 0; a < padded_message_len; a++){
        if (a >= padded_message_len){
            padded_message[a] = '0';
        } else {
            padded_message[a] = original_message[a];
        }
    }
    padded_message[padded_message_len] = '\0';

    unsigned char temp[padded_message_len + 1];
    unsigned char * encrypted_message = temp;
    for (int a = 0; a < padded_message_len; a += 16){
        unsigned char block_to_encrypt[17];
        for (int b = 0; b < 16; b++){
            block_to_encrypt[b] = padded_message[a + b];
        }

        block_to_encrypt[16] = '\0';
        AES_encrypt(block_to_encrypt, key);

        for (int j = 0; j < 16; j++){
            encrypted_message[j + a] = block_to_encrypt[j];
        }
    }
    encrypted_message[padded_message_len] = '\0';   //very important for decryption

    for (int a = 0; a < padded_message_len + 1; a++){
//          printf("%x ", encrypted_message[a]);
        encrypted[a] = encrypted_message[a];
    }
//    encrypted[padded_message_len] = '\0';
//    printf("\n");

    return encrypted_message;
}

unsigned char * general_decrypt(unsigned char * message, int message_len, int key_len, unsigned char * k){
    int padded_message_len = message_len;
    unsigned char temp[padded_message_len + 1];        //the encrypted message should be the length of the padded original message
    unsigned char * decrypted_message = temp;
    unsigned char message_copy[padded_message_len + 1];

    set_key_length(key_len);

    for (int a = 0; a < padded_message_len; a++){
        message_copy[a] = message[a];
    }   //added this because C overwrites the contents of memory somewhere during the execution of this function

    for (int a = 0; a < padded_message_len; a += 16){
        unsigned char block_to_decrypt[17];

        for (int b = 0; b < 16; b++){
            block_to_decrypt[b] = message_copy[a + b];
        }
        block_to_decrypt[16] = '\0';


        AES_decrypt(block_to_decrypt, k);

        for (int b = 0; b < 16; b++){
            decrypted_message[b + a] = block_to_decrypt[b];
        }
    }

    decrypted_message[padded_message_len] = '\0';

    for (int a = 0; a < padded_message_len; a++){
//          printf("%x ", decrypted_message[a]);
        message[a] = decrypted_message[a];
    }

    message[padded_message_len] = '\0';
//      printf("\n");

    return decrypted_message;
}
```

B. AES.h

```c
//
// Created by armandt on 2020/04/07.
//

#ifndef AES_H
#define AES_H

/**
 * @brief The key expansion core is used in the key expansion method and contains 3 steps. 1) Rotate left. 2) S-box on all four bytes. 3) XOR with RCons
 * @param in A temporary 4 bytes used to generates the expanded key.
 * @param i The RCon iteration index
 */
void key_expansion_core(unsigned char* in, unsigned char i);
/**
 * @brief This method expands the original key to the appropriate expanded key, to provide enough round keys for the AES function.
 * @param input_key The original key.
 * @param expanded_key The final expanded to to be used by the AES algorithm.
 */
void key_expansion(unsigned char* input_key, unsigned char* expanded_key);
/**
 * @brief Uses the S-box table to perform a byte-by-byte substitution of the current state.
 * @param state The 128-bit block is copied to a state which is modified at each stage of the encryption.
 */
void sub_bytes(unsigned char* state);
/**
 * @brief Uses the inverse S-box table to perform a byte-by-byte substitution of the current state.
 * @param state The 128-bit block is copied to a state which is modified at each stage of the decryption.
 */
void inverse_sub_bytes(unsigned char* state);
/**
 * @brief A simple permutation which is performed row by row
 * @param state The 128-bit block is copied to a state which is modified at each stage of the encryption.
 */
```

```
void shift_rows(unsigned char* state);
/**
 * @brief A simple permutation which is performed row by row
 * @param state The 128-bit block is copied to a state which is modified at each stage of the decryption.
 */
void inverse_shift_rows(unsigned char* state);
/**
 * @brief A substitution that alters each byte in a column as a function of all of the bytes in the column.
 * @param state The 128-bit block is copied to a state which is modified at each stage of the decryption.
 */
void mix_columns(unsigned char* state );
/**
 * @brief A substitution that alters each byte in a column as a function of all of the bytes in the column.
 * @param state The 128-bit block is copied to a state which is modified at each stage of the decryption.
 */
void inverse_mix_columns(unsigned char* state );
/**
 * @brief A simple bitwise XOR of the current block with a portion of the expanded key.
 * @param state The 128-bit block is copied to a state which is modified at each stage of the decryption.
 * @param round_key The portion of the expanded key used in a particular round.
 */
void add_round_key(unsigned char* state, unsigned char* round_key);

/**
 * @brief Uses AES method to encrypt a message using the provided key. The encrypted
 * message is stored in the message array that it passed in.
 * @param message the plaintext that will be encrypted.
 * @param key The key used by the algorithm
 */
void AES_encrypt(unsigned char* message, unsigned char* key);
/**
 * @brief Uses AES method to decrypt a message using the provided key. The decrypted
 * message is stored in the message array that it passed in.
 * @param message The encrypted message that will be
```

```
decrypted.
 * @param key The key used by the algorithm
 */
void AES_decrypt(unsigned char* message, unsigned char* key);
/**
 * @brief Simply prints the input string in a hex format.
 * @param string The message to be printed in hex format.
 * @param count The number of characters that will be printed
 */
void print_hex(const unsigned char *string, int count);
/**
 * @brief Prints the input string in a hex, in a 4x4 block format.
 * @param string The message to be printed in hex format.
 */
void print_hex_block(const char *string);
/**
 * @brief Displays each individual functions results independently.
 * @param string The message to be passed into each individual function.
 * @param key_length The length of the input key.
 * @param key The input key.
 */
void test_functionality(unsigned char *input_string, int key_length, unsigned char* key);

    //Functions wat Armandt by gesit het
    /**
 * @brief Sets the key length, number of rounds and expanded key size.
 * @param l Length of the key in bits
 */
    void set_key_length(int l);

/**
 * @brief Set the number_of_rounds variable
 * @param r Number of rounds
 */
void set_number_of_rounds(int r);

/**
 * @brief Set the expanded_key_size variable
 * @param s Expanded key size
 */
void set_expanded_key_size(int s);

```

```
/**
 * @brief A function that combines zero padding and encryption of an
 * arbitrarily-sized char array using AES_encrypt. This function will also
 * call the set_key_length function to initialise those variables.
 * @param message The char array that must be encrypted
 * @param message_len Length of the message to be encrypted in bytes.
 * @param encrypted The array where the encrypted message is stored
 * @param key_length The length of the key in bits
 * @param key The key used for encryption
 * @return Returns an array containing the encrypted message
 */
unsigned char* pad_and_encrypt(unsigned char * message, unsigned char * encrypted, int message_len, int key_length, unsigned char * key);

/**
 * @brief Takes a longer encrypted message and decrypts it, returning an array containing the decrypted message
 * @param message The encrypted message
 * @param massage_len The length of the message in bytes
 * @param key_length Length of the key in bits
 * @param key The key used for encryption
 * @return An array containing the message that has been decrypted (possibly padded with zeros)
 */
unsigned char* general_decrypt(unsigned char * message, int message_len, int key_length, unsigned char * key);


#endif //ARMANDT_MICHELLE_H
```

II. Cipher Mode Files

A. *CipherModes.c*

```
//
// Created by fouri on 2020/03/27.
//

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "CipherModes.h"
#include "AES.h"


void iterativeEncryptCBC(struct CBC *c){
    int blockSize = (*c).blockSize;
    int pSize = (*c).pSize;

    int round = 0;

    while
        (!((((round + 1) * blockSize - pSize) >= blockSize) || (round * blockSize - pSize == 0))){
            unsigned char temp[blockSize];

            int i = round * blockSize; //i is the index from where we will begin to copy chars
            int j = 0;                 //j is the number of items that have been copied over so far

            while ((i < pSize) && (j < blockSize))
            {
                temp[j] = (*c).plaintext[i];
                j++;
                i++;
            } //first, copy chars from p to temp without going beyond the scope of p
            while (j < blockSize)
            {
                temp[j] = 0;
                j++;
            } //pad with zeros if necessary

            //now we have temp, the block of input data we want to work with
            //        printf("XOR Output block: \t");
            if (round == 0)
            {
                for (int a = 0; a < blockSize; a++)
                {
                    temp[a] = temp[a] ^ (*c).iv[a];
                    //              printf("%x ", temp[a]);
                } //xor the plaintext with the IV
            }
            else
            {
                for (int a = 0; a < blockSize; a++)
                {
                    temp[a] = temp[a] ^ (*c).ciphertext[a + (round - 1) * blockSize]; //replace this with the previous block's ciphertext values
                } //xor the plaintext with the previous block of ciphertext
            }


            //call AES function on the temp array
            set_key_length((*c).keySize); //pass in the size of the key in bits
            unsigned char encrypted[blockSize + 1];
            pad_and_encrypt(temp, encrypted, blockSize, (*c).keySize, (*c).key);


            for (int a = 0; a < blockSize; a++)
            {
                (*c).ciphertext[a + blockSize * round] = encrypted[a];
            }
            round++;
        }
}


void iterativeDecryptCBC(struct CBC *c){
    int blockSize = (*c).blockSize;
    int cSize = (*c).cSize;

    int round = 0;

    while ((round * blockSize) - cSize != 0) {
        unsigned char temp[blockSize + 1];
        unsigned char *decrypted;

        //call aes decryption
        for (int a = 0 + round * blockSize; a < blockSize +
            round * blockSize; a++)
        {
            temp[a - round * blockSize] = (*c).ciphertext[a];
        }                       //copy ciphertext into temp
        temp[blockSize] = '\0'; //append an endline char

        general_decrypt(temp, blockSize, (*c).keySize, (*c).key);    //decrypt the block

        if (round == 0)
        {
            for (int a = 0; a < blockSize; a++)
            {
                temp[a] = temp[a] ^ (*c).iv[a];
                (*c).plaintext[a + blockSize * round] = temp[a];
            } //perform the XOR step using the IV and store the output in the plaintext array
        }
        else
        {
            for (int a = 0; a < blockSize; a++)
            {
                temp[a] = temp[a] ^ (*c).ciphertext[a + (round - 1) * blockSize];
                (*c).plaintext[a + blockSize * round] = temp[a];
            }//perform the xor step using the ciphertext block and store the output in the plaintext array
        }
        round++;
    }
}

void shiftBytesIn(unsigned char* shiftReg, int regSize, unsigned char* newData, int newDataSize){
    for (int a = 0; a < regSize - newDataSize; a++){
        shiftReg[a] = shiftReg[a + newDataSize];
    }

    int b = 0;
    for (int a = regSize - newDataSize; a < regSize; a++){
        shiftReg[a] = newData[b++];
    }
}


```

```c
void iterativeEncryptCFB(struct CFB *c){
    int blockSize = (*c).blockSize;
    int pSize = (*c).pSize;
    int shiftRegSize = (*c).shiftRegSize;

    int round = 0;

    while (round * blockSize <= pSize){
        //step 1: encrypt the IV/Shift Register using the provided Key, K
        unsigned char temp[shiftRegSize];
        for (int a = 0; a < shiftRegSize; a++)
        {
            temp[a] = (*c).shiftRegister[a];
        } //copy the shift register into a temp array

        unsigned char storage[shiftRegSize + 1];
        unsigned char keyCopy[c->keySize / 8];
        for (int a = 0; a < c->keySize / 8; a++)
        {
            keyCopy[a] = (*c).key[a];
        }
        pad_and_encrypt(temp, storage, shiftRegSize, (*c).keySize, keyCopy);

        //Step 2: XOR the LSB of the temp array with the plaintext
        int b = 0;
        for (int a = 0 + round * blockSize; a < round * blockSize + blockSize; a++)
        {
            //          (*c).ciphertext[a] = temp[b] = temp[b] ^ (*c).plaintext[a];
            (*c).ciphertext[a] = storage[b] = storage[b] ^ (*c).plaintext[a];
            b++;
        } //the first [blockSize] bytes of temp now contains the new block of ciphertext. And ciphertext has the new
        //data in it as well

        //Step 3: Shift the ciphertext into the shift register before starting the next round.
        shiftBytesIn((*c).shiftRegister, shiftRegSize, storage, blockSize);
        round++;
    }
}

```

```c

void iterativeDecryptCFB(struct CFB *c){
    int blockSize = (*c).blockSize;
    int pSize = (*c).pSize;
    int shiftRegSize = (*c).shiftRegSize;

    int round = 0;

    while ((round)*blockSize <= pSize){
        if (round == 0)
        {
            for (int a = 0; a < shiftRegSize; a++)
            {
                (*c).shiftRegister[a] = (*c).iv[a];
            } //the shift register should begin the same as the IV
        }
        else
        {
            //              printf("Shift Register: \t");
            //              printArr((*c).shiftRegister, shiftRegSize, 'x');
        }

        //step 1: encrypt the IV/Shift Register using the provided Key, K
        unsigned char temp[shiftRegSize];
        if (round == 0)
        {
            for (int a = 0; a < shiftRegSize; a++)
            {
                temp[a] = (*c).iv[a];
            } //copy the IV into a temp array
        }
        else
        {
            for (int a = 0; a < shiftRegSize; a++)
            {
                temp[a] = (*c).shiftRegister[a];
            } //copy the shift register into a temp array
        }

        unsigned char storage[shiftRegSize + 1];
        pad_and_encrypt(temp, storage, shiftRegSize, (*c).keySize, (*c).key);

        //step 2: XOR the first s bits of the encrypted output with the first s bits of ciphertext to get the plaintext
        int b = blockSize * round;
        for (int a = 0; a < blockSize; a++)
        {
            //          (*c).plaintext[a + b] = temp[a] ^ (*c).ciphertext[b + a];
            (*c).plaintext[a + b] = storage[a] ^ (*c).ciphertext[b + a];
        }

        //step 3: shift ciphertext block into shiftreg before running next step
        for (int a = 0; a < blockSize; a++)
        {
            temp[a] = (*c).ciphertext[a + b];
        } //copy ciphertext block into temp array (just so I can use the shiftBytesIn function more easily)

        shiftBytesIn((*c).shiftRegister, shiftRegSize, temp, blockSize);
        round++;
    }
}


void printArr(unsigned char *arr, int size, char format){
    if (format == 'c'){
        for (int a = 0; a < size; a++){
            printf("%c", arr[a]);
        }
    } else if (format == 'x'){
        for (int a = 0; a < size; a++){
            printf("%x ", arr[a]);
        }
    } else if (format == 'd'){
        for (int a = 0; a < size; a++){
            printf("%d ", arr[a]);
        }
    }

    printf("\n");
}

void readFile(unsigned char * filename, unsigned char * fileBuffer){
    FILE *f;
    f = fopen(filename, "rb");  //open binary file
    long int fileSize = 0;

    if (f == NULL){
```

```c
        printf("Error, file not found.\n");
        exit(0);
    } else {
        fileSize = getFileSize(filename);

        printf("Reading %Ld bytes from file.\n", fileSize);
        fread(fileBuffer, fileSize + 1, 1, f);
    }

    fclose(f);
}

void saveFile(unsigned char * filename, unsigned char * fileBuffer, int fileSize){
    FILE * f;
    f = fopen(filename, "wb");

    int numZeros = 0;   //the number of zeros added to a file when using CBC
    int a = fileSize - 1;
    while(fileBuffer[a] == 0){
        a--;
        numZeros++;
    }

    fileSize -= numZeros;   //this stops zeroes that were added for padding from being saved upon decryption

    if (f == NULL) {
        printf("Error. File could not be opened.\n");
    } else {
        fwrite(fileBuffer, fileSize, 1, f);
    }
    fclose(f);
}

long int getFileSize(unsigned char * filename){
    FILE *f;
    f = fopen(filename, "rb");   //open binary file
    long int fileSize = 0;

    if (f == NULL){
        printf("Error, file not found.\n");
    } else {
        fseek(f, 0L, SEEK_END);
        fileSize = ftell(f);
    }

    fclose(f);
    return fileSize;
}
```

B. *CipherModes.h*

```
//
// Created by fouri on 2020/03/30.
//

#ifndef cipher_modes_H
#define cipher_modes_H
#include <stdio.h>

/**
 * A structure to hold relevant info for doing cipher block chaining
 */
struct CBC{
    int pSize;          //size of plaintext in BYTES = no. of array indexes
    int cSize;          //size of the ciphertext created in BYTES
    int blockSize;      //size of each block in BYTES
    int keySize;        //size of the key in bits
    unsigned char* plaintext;   //array of chars
    unsigned char* key;
    unsigned char* ciphertext;
    unsigned char* iv;          //this holds the initialization vector
};

/**
 * The structure for the CFB methods. The shiftregister and IV should be initialized as having the same contents,
 * but not being the same object.
 */
struct CFB{
    int pSize;           //size of the plaintext message/ciphertext
    int shiftRegSize;    //bytes in the IV/Shift Register
    int blockSize;       //number of bytes processed per round
    int keySize;         //size of the key in bits
    unsigned char* plaintext;
    unsigned char* ciphertext;
    unsigned char* iv;  //init vector. THis does not get changed.
    unsigned char* shiftRegister;
    unsigned char* key;
};

/**
```

```c
 * @brief Takes a CBC struct and encrypts the data it contains, storing the ciphertext in the struct itself.
 * @param c The CBC structure used.
 */
void iterativeEncryptCBC(struct CBC * c);


/**
 * @brief Takes a CBC struct and and decrypts the data it contains iteratively rather than recursively.
 * @param c The CBC structure used.
 */
void iterativeDecryptCBC(struct CBC * c);

/**
 * @brief Acts as a shift register, shifting new data into an existing array.
 * @param shiftReg The register that is accepting new data
 * @param regSize The size of the accepting register
 * @param newData The array containing the new data
 * @param newDataSize The size of the new data array.
 */
void shiftBytesIn(unsigned char* shiftReg, int regSize, unsigned char* newData, int newDataSize);

/**
 * @brief Uses the cipher feedback mode to encrypt a message. Internally, this function uses the AES
 * encryption algorithm to encrypt the IV/Shift Register
 * @param c The CFB struct containing the plaintext and other relevant information
 */
void iterativeEncryptCFB(struct CFB *c);

/**
 * @brief Uses the cipher feedback mode to decrypt a message. Internally, this function uses the AES
 * decryption algorithm to decrypt the IV/Shift Register
 * @param c The CFB struct containing the plaintext and other relevant information
 */
void iterativeDecryptCFB(struct CFB *c);


/**
 * @brief Prints the contents of an array separated by spaces, with a newline at the end
 * @param arr The array containing the chars
 * @param size The size of the array
 * @param format The char that indicates to the printf function how it should display the data in the array
 */
void printArr(unsigned char *arr, int size, char format);


/**
 * @brief Accepts a filename and opens the file. The function assumes that the file is located in
 * the root directory of the program. (Same folder as main.c)
 * @param filename The name of the file that must be encrypted.
 */
void readFile(unsigned char * filename, unsigned char * fileBuffer);

/**
 * @brief Saves the given array as a file, designated by the filename.
 * @param filename Name of the file, including the extension.
 * @param fileBuffer Array containing data.
 * @param fileSize Number of elements in fileBuffer.
 */
void saveFile(unsigned char * filename, unsigned char * fileBuffer, int fileSize);

/**
 * @brief Returns the size of the file in bytes.
 * @param filename The filename
 * @return File size in bytes
 */
long int getFileSize(unsigned char * filename);
#endif //ARMANDT_ARMANDT_H
```

III. Main and makefile

A. main.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <printf.h>
#include "string.h"
#include "CipherModes.h"
#include "AES.h"

int main(int argc, char *argv[])
{

    bool encrypt = false;
    bool decrypt = false;
    bool cbc = false;
    bool cfb = false;
    bool textIn = false;
    bool fileIn = false;
    unsigned char inputText[1000];
    unsigned char key[33];
    unsigned char iv[101];
    unsigned char *inFileName = NULL;
    unsigned char outFileName[100];
    unsigned char tempOutFileName[200];
    unsigned char newFileBuffer[1000000]; //approx 1mb file
    int keyLength = 0;  //the length of the key given by the user
    int streamLen = 0;
    int argLength = 0;  //the length of arguments like text, keys, etc.
    int plaintextLength = 0;

    struct CBC cbcStruct;
    struct CFB cfbStruct;

    for (int a = 1; a < argc; a++)
    {
        if (strcmp(argv[a], "-e") == 0)
        {
            encrypt = true;
            if (!((strcmp(argv[a + 1], "-cbc") == 0) || (strcmp(argv[a + 1], "-cfb") == 0))){
                keyLength = atoi(argv[a+1]);
            }   //if neither cbc nor cfb are used, then the keylength must still be entered.
        }
        else if (strcmp(argv[a], "-d") == 0)
        {
            decrypt = true;
        }
        else if (strcmp(argv[a], "-cbc") == 0)
        {
            cbc = true;
            keyLength = atoi(argv[a + 1]);
            if (!((keyLength == 128) || (keyLength == 192) || (keyLength == 256)))
            {
                printf("Incorrect key size entered. Closing.\n");
                return 0;
            }
            cbcStruct.keySize = keyLength;
            cbcStruct.blockSize = 16;
        }
        else if (strcmp(argv[a], "-cfb") == 0)
        {
            cfb = true;
            keyLength = atoi(argv[a + 1]);
            if (!((keyLength == 128) || (keyLength == 192) || (keyLength == 256)))
            {
                printf("Incorrect key size entered. Closing.\n");
                return 0;
            }
            cfbStruct.keySize = keyLength;
        }
        else if (strcmp(argv[a], "-t") == 0)
        {
            textIn = true;
            argLength = strlen(argv[a + 1]);
            unsigned char t[argLength + 1];
            for (int b = 0; b < argLength; b++){
                inputText[b] = argv[a+1][b];
            }
            inputText[argLength] = '\0';
//            plaintext = inputText;
            plaintextLength = argLength;
            if (cbc){
                if (encrypt){
                    cbcStruct.plaintext = inputText;
                    cbcStruct.pSize = plaintextLength;
                    cbcStruct.cSize = (cbcStruct.pSize / 16 + 1) * 16;
                } else {
                    cbcStruct.ciphertext = inputText;
                    cbcStruct.cSize = plaintextLength;
                    cbcStruct.pSize = cbcStruct.cSize;
                }
            } else if (cfb){
                if (encrypt){
                    cfbStruct.pSize = plaintextLength;
                    cfbStruct.plaintext = inputText;
                } else {
                    cfbStruct.pSize = plaintextLength;
                    cfbStruct.ciphertext = inputText;
                }
            }
            a++;
        }
        else if (strcmp(argv[a], "-key") == 0)
        {
            argLength = strlen(argv[a + 1]);
            for (int b = 0; b < argLength; b++){
                key[b] = argv[a+1][b];
            }
            key[argLength] = '\0';
//            key = key;

            if (cbc){
                cbcStruct.key = key;
            } else if (cfb) {
                cfbStruct.key = key;
                cfbStruct.shiftRegSize = 32;
            }
            a++;
        }
        else if (strcmp(argv[a], "-iv") == 0)
        {
            argLength = strlen(argv[a + 1]);
//            unsigned char iv[argLength + 1];
            for (int b = 0; b < argLength; b++){
                iv[b] = argv[a+1][b];
            }
            iv[argLength] = '\0';

            if (cbc) {
                cbcStruct.iv = iv;
                cbcStruct.blockSize = 16;
            } else if (cfb) {
                cfbStruct.iv = iv;
                cfbStruct.shiftRegSize = argLength;
            }
            a++;
        }
        else if (strcmp(argv[a], "-fi") == 0)
        {
            fileIn = true;
            argLength = strlen(argv[a + 1]);
            unsigned char temp[argLength + 1];
            for (int b = 0; b < argLength; b++){
                temp[b] = argv[a+1][b];
                tempOutFileName[b] = temp[b];
            }
            temp[argLength] = '\0';
            inFileName = temp;
            plaintextLength = getFileSize(inFileName);
            readFile(inFileName, newFileBuffer);

            for (int b = 0; b < argLength; b++){
                inFileName[b] = 0;
            }

            if (cbc) {
                if (encrypt){
                    cbcStruct.plaintext = newFileBuffer;
                    cbcStruct.pSize = plaintextLength;
                    cbcStruct.cSize = (cbcStruct.pSize / 16 + 1) * 16;
                } else {
                    cbcStruct.ciphertext = newFileBuffer;
                    cbcStruct.cSize = plaintextLength;
                    cbcStruct.pSize = plaintextLength;
                }
            } else if (cfb) {
                if (encrypt){
                    cfbStruct.plaintext = newFileBuffer;
                    cfbStruct.pSize = plaintextLength;
                } else {
                    cfbStruct.ciphertext = newFileBuffer;
                    cfbStruct.pSize = plaintextLength;
                }
            }
            a++;
        }
        else if (strcmp(argv[a], "-fo") == 0)
        {
            argLength = strlen(argv[a + 1]);
            if (argLength > 100){
                printf("The␣output␣file␣name␣entered␣is␣too␣long.␣Please␣use␣fewer␣than␣100␣characters.\n");
                return 0;
            }

            for (int b = 0; b < argLength - 2; b++){
                outFileName[b] = argv[a+1][b + 2];
            }
            outFileName[argLength] = '\0';

            for (int b = 199; b > 0; b--){
                if (tempOutFileName[b] == '/'){
                    if (cbc){
                        tempOutFileName[b + 1] = 'C';
                        tempOutFileName[b + 2] = 'B';
                        tempOutFileName[b + 3] = 'C';
                    } else {
                        tempOutFileName[b + 1] = 'C';
                        tempOutFileName[b + 2] = 'F';
                        tempOutFileName[b + 3] = 'B';
                    }
                    tempOutFileName[b + 4] = ' ';
                    tempOutFileName[b + 5] = 'O';
                    tempOutFileName[b + 6] = 'u';
                    tempOutFileName[b + 7] = 't';
                    tempOutFileName[b + 8] = 'p';
                    tempOutFileName[b + 9] = 'u';
                    tempOutFileName[b + 10] = 't';
                    tempOutFileName[b + 11] = '/';

                    for (int c = 0; c < argLength - 2; c++){
                        tempOutFileName[b + c + 11] = outFileName[c];
                    }// add the part of the path for the folders and the name for the output file
                    break;
                }
            }
            a++;
        }
        else if (strcmp(argv[a], "-streamlen") == 0)
        {
            streamLen = atoi(argv[a + 1]);


            if (cfb) {
                if (!((streamLen == 8) || (streamLen == 64) || (streamLen == 128))){
                    printf("Please try again and enter a valid streamlength.\n");
                    return 0;
                }
                cfbStruct.blockSize = atoi(argv[a + 1]);
            } else if (cbc){
                printf("Streamlen␣is␣not␣allowed␣for␣CBC␣operation.␣Please␣try␣again.\n");
                return 0;
            }
            a++;
        }
        else if (strcmp(argv[a], "-h") == 0)
        {
            printf("\n
            ===========================================================
            n");
            printf("The␣following␣commands␣are␣available:␣\n");
            printf("-e:␣\t\t\t\t␣Encryption\n");
            printf("-d:␣\t\t\t\t␣Decryption\n");
            printf("-cbc␣<len>:\t\t\t␣CBC␣Encryption/Decryption\n");
            printf("-cfb␣<len>:\t\t\t␣CFB␣Encryption/Decryption\n");
            printf("<len>:␣\t\t\t\t␣Key␣length:␣either␣128,␣192␣or␣256\n");
            printf("-t␣<text>:\t\t\t␣Enter␣the␣text␣to␣encrypt␣after␣this␣tag,␣surrounded␣by␣quotation␣marks.\n");
            printf("-key␣<password>:␣\t␣Enter␣the␣password␣after␣this␣tag.\n");
            printf("-iv␣<init␣vect>:␣\t␣Enter␣the␣initialisation␣vector␣after␣this␣tag.\n");
            printf("-fi␣<input␣file>:\t␣Enter␣the␣name␣of␣the␣input␣file.\n");
            printf("-fo␣<output␣file>:\t␣Enter␣the␣name␣of␣the␣output␣file.\n");
            printf("-streamlen␣<len>:␣\t␣Enter␣the␣streamlength␣after␣this␣tag.\n");
            printf("-h:␣\t\t\t\t␣Enter␣this␣tag␣to␣display␣this␣message.\n");
            printf("
            ===========================================================
            n");
        }
    }

    int s = 0;
    if (cbc){
        if (encrypt){
            s = cbcStruct.cSize;  //make an array to hold
```

```
                the ciphertext
        } else {
            s = cbcStruct.pSize;  //plaintext array will be same size as ciphertext
        }
    } else if (cfb){
        s = cfbStruct.pSize;    //both arrays are the same size for cfb
    } else {
        s = (plaintextLength / 16 + 1) * 16;
    }

    unsigned char newArray[s + 1];  //this will store cipher or plaintext

    if (cbc) {
        if (encrypt){
            cbcStruct.ciphertext = newArray;
            printf("Encryption has started.\n");
            iterativeEncryptCBC(&cbcStruct);
            printf("Done encrypting.\n");


            if (fileIn){
                saveFile(tempOutFileName, cbcStruct.ciphertext, cbcStruct.cSize);
                printf("File saved in the CBC folder.\n");
            } else if (textIn){
                printArr(cbcStruct.ciphertext, cbcStruct.cSize, 'x');
            }

//            iterativeDecryptCBC(&cbcStruct);
//            printArr(cbcStruct.plaintext, cbcStruct.pSize, 'c');

        } else {
            cbcStruct.plaintext = newArray;
            printf("Decryption has started.\n");
            iterativeDecryptCBC(&cbcStruct);
            printf("Done decrypting. \n");
//            printArr(cbcStruct.plaintext, cbcStruct.pSize, 'c');

            if (fileIn){
                saveFile(tempOutFileName, cbcStruct.plaintext, cbcStruct.pSize);
                printf("File saved in the CBC folder.\n");
            }
```

```

//            printArr(cbcStruct.plaintext, cbcStruct.pSize, 'c');
        }
    } else if (cfb) {
        unsigned char newShiftReg[cfbStruct.shiftRegSize];
        cfbStruct.shiftRegister = newShiftReg;
        for (int a = 0; a < cfbStruct.shiftRegSize; a++){
            newShiftReg[a] = iv[a];
        }

        if (encrypt){
            cfbStruct.ciphertext = newArray;
            printf("Encryption␣has␣started.\n");
            iterativeEncryptCFB(&cfbStruct);
            printf("Done␣encrypting.\n");
            // printArr(cfbStruct.ciphertext, cfbStruct.pSize, 'x');

            if (fileIn){
                saveFile(tempOutFileName, cfbStruct.ciphertext, cfbStruct.pSize);
                printf("File␣saved␣in␣the␣CFB␣folder.\n");
            }

//            iterativeDecryptCFB(&cfbStruct);
//            printArr(cfbStruct.plaintext, cfbStruct.pSize, 'c');
        } else {
            cfbStruct.plaintext = newArray;
            printf("Decryption␣has␣started.\n");
            iterativeDecryptCFB(&cfbStruct);
            printf("Done␣decrypting.\n");

            if (fileIn){
                saveFile(tempOutFileName, cfbStruct.plaintext, cfbStruct.pSize);
                printf("File␣saved␣in␣the␣CFB␣folder.\n");
            }
        }
    } else {
        if (encrypt){
            set_key_length(keyLength);
            test_functionality(inputText, keyLength, key);
//            pad_and_encrypt(inputText, newArray, plaintextLength, keyLength, key);
        } else {
            general_decrypt(inputText, plaintextLength, keyLength, key);
        }

    }   //neither cbc nor cfb

    return 0;
}
```

B. makefile

```
main: main.o CipherModes.o AES.o
	gcc -static main.o CipherModes.o AES.o -o main

main.o: main.c CipherModes.h AES.h
	gcc -c main.c

CipherModes.o: CipherModes.c CipherModes.h
	gcc -c CipherModes.c

AES.o: AES.c AES.h
	gcc -c AES.c

run:
	./main

clean:
	rm -f main CipherModes.o AES.o main.o
```