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

EHN 410

Network Security

Practical 2 Code

Group: 12

Name and Surname	Student Number	% Contribution
J.C.J. du Plessis	17010986	33.33
I. Eloff	17018082	33.33
R.S. Walters	17017051	33.33

By submitting this assignment we confirm that we have read and are 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.

May 5, 2020

Contents

1 AES.c

```c
#include "AES.h"


int main(int argc, char *argv[])
{
    int i;
    //             cbc/cfb  e/d    key     t     iv     fi     fo   streamlen
    bool args[8] = {false, false, false, false, false, false, false, false};
    int method = -1; // CBC if false, CFB if true
    int operation = -1; // encrypt if false, decrypt if true
    int width = -1; // AES128, AES192, AES256 macros
    int block_len = 16; // CFB8 , CFB64 , CFB128 (default) marcos
    bool file_output = false;
#if VERBOSE
    bool verbose = false; // Show all steps if true
#endif
    int message_len = 0;
    unsigned char *message = NULL;
    char *output_file_name = NULL;
    int IV[16];
    int user_key[32];

    char help_message[] = "\t./AES -arg1 value1 -arg2 value2...\n"
                          "\t\n"
                          "\tThe following arguments should then be given in this order:\n\n"
                          "\t-e (encryption), or\n"
                          "\t-d (decryption)\n\n"
                          "\t-cbc <len> (Cipher Block Chaining, <len> either 128, 192 or 256), or\n"
                          "\t-cfb <len> (Cipher Feedback, <len> either 128, 192 or 256)\n\n"
                          "\t-t <text to encrypt in ASCII or text to decrypt in HEX>, or\n"
                          "\t-fi <input file> and\n"
                          "\t-fo <output file>\n\n"
                          "\t-key <password in ASCII>\n\n"
                          "\t-iv <initialization vector in ASCII>\n\n"
                          "\t-streamlen <len> (length of the CFB stream if '-cfb' is given, either 8, 64 or
                          ↪  128)\n\n"
                          "\t-h help (will show this message)\n\n"
#if VERBOSE
                          "\t-verbose (will show all steps in the AES process)\n\n"
#endif
                          "\t\nRemember to add \"double quotes\" to ASCII inputs if spaces are present in the
                          ↪  string\n"
                          "\t\nExample usage:\n"
                          "\t1.\t./AES -e -cbc 128 -fi \"input.txt\" -fo \"encrypted.enc\" -key \"Very strong
                          ↪  password\" -iv \"Initialization vector\"\n"
                          "\t2.\t./AES -d -cfb 192 -fi \"encrypted.jpg\" -fo \"image.jpg\" -key \"Very strong
                          ↪  password\" -iv \"Initialization vector\" -streamlen 64\n"
                          "\t3.\t./AES -e -cbc 256 -t \"Text to encrypt\" -key \"Very strong password\" -iv
                          ↪  \"Initialization vector\"\n"
                          "\t4.\t./AES -d -cfb 128 -t C7D3CAAFEE6137 -key \"Very strong password\" -iv
                          ↪  \"Initialization vector\" -streamlen 8\n";
    //                                          ^^^^^^^^^^^^^^^ "Success"
```

```
// Greeting
printf("\nEHN 410 Group 12 Practical 2\n\n");

int arg;
for (arg = 1; arg < argc; arg++)
{
    if (strstr(argv[arg], "-cbc") != NULL) // Set chaining method to CBC and length
    {
        args[0] = true;
        method = false; //set the chaining method

        if (!strcmp(argv[arg + 1], "128"))
        {
            width = AES128;
            printf("AES128 with CBC selected\n");
        }
        else if (!strcmp(argv[arg + 1], "192"))
        {
            width = AES192;
            printf("AES192 with CBC selected\n");
        }
        else if (!strcmp(argv[arg + 1], "256"))
        {
            width = AES256;
            printf("AES256 with CBC selected\n");
        }
        else
        {
            printf("Parameter '%s' is not a valid length\n", argv[arg + 1]);
            printf("Valid parameters are '128', '192' and '256'\n");
            return EXIT_FAILURE;
        }

        arg++; // Skip over the value parameter that follows this parameter
    }
    else if (strstr(argv[arg], "-cfb") != NULL) // Set chaining method to CFB and length
    {
        args[0] = true;
        method = true; //set the chaining method

        if (!strcmp(argv[arg + 1], "128"))
        {
            width = AES128;
            printf("AES128 with CFB selected\n");
        }
        else if (!strcmp(argv[arg + 1], "192"))
        {
            width = AES192;
            printf("AES192 with CFB selected\n");
        }
        else if (!strcmp(argv[arg + 1], "256"))
        {
            width = AES256;
```

```
            printf("AES256 with CFB selected\n");
        }
        else
        {
            printf("Parameter '%s' is not a valid length\n", argv[arg + 1]);
            printf("Valid parameters are '128', '192' and '256'\n");
            return EXIT_FAILURE;
        }

        arg++; // Skip over the value parameter that follows this parameter
    }
    else if (strstr(argv[arg], "-e") != NULL) // Set operation encrypt
    {
        args[1] = true;
        operation = false;
        printf("Encryption selected\n");
    }
    else if (strstr(argv[arg], "-d") != NULL) // Set operation decrypt
    {
        args[1] = true;
        operation = true;
        printf("Decryption selected\n");
    }
    else if (strstr(argv[arg], "-key") != NULL) // Set the user key
    {
        args[2] = true;

        if (width == -1)
        {
            printf("The AES length must be specified before the key is given\n");
            printf("Specify this with '-cbc <length>' or '-cfb <length>'\n");
            return EXIT_FAILURE;
        }
        else
        {
            int user_key_size;

            if (width == AES128)
                user_key_size = AES128_USER_KEY_SIZE;
            else if (width == AES192)
                user_key_size = AES192_USER_KEY_SIZE;
            else
                user_key_size = AES256_USER_KEY_SIZE;

            char key[user_key_size + 1]; // +1 for null terminator

            for (i = 0; i < user_key_size + 1; i++) // Fill with zeroes to pad if needed + null terminator
                key[i] = '\0';

            strncpy(key, argv[arg + 1], user_key_size); // Copy the user key input

            // Convert from ASCII string to int array
            for (i = 0; i < user_key_size; i++)
                user_key[i] = (unsigned char) key[i]; // Get the integer value from the byte
```

```
            printf("Key (ASCII): \"%s\"\n", key);
            arg++; // Skip over the value parameter that follows this parameter
        }
    }
    else if (strstr(argv[arg], "-t") != NULL) // Set the input message
    {
        args[3] = true;

        if (operation == -1)
        {
            printf("The operation type must be specified before the message is given\n");
            printf("Specify this with '-e' for encryption or '-d' for decryption\n");
            return EXIT_FAILURE;
        }
        else if (operation) // Decrypt
        {
            // Take message as hex input
            char *parameter = argv[arg + 1];
            message_len = strlen(parameter) / 2; // 2 hex chars = 1 byte

            if (message_len > MAX_REQ_LEN)
            {
                printf("The message is too long, a maximum of %d bytes may be given with '-t'\n",
                ↪  MAX_REQ_LEN);
                return EXIT_FAILURE;
            }
            else
            {
                message = (unsigned char *) malloc((message_len + 17) * sizeof(unsigned char)); // + 17 if
                ↪  incomplete block to pad with zeroes

                for (i = message_len; i < message_len + 17; i++) // Fill with zeroes to pad if needed +
                ↪  null terminator
                    message[i] = '\0';

                // Convert from hex string to int array
                char current_number[2];
                for (i = 0; i < message_len; i++)
                {
                    strncpy(current_number, parameter, 2); // Retrieve one byte (two hex chars)
                    message[i] = (unsigned char) hex_convert(current_number, 2); // Get the integer value
                    ↪  from the byte
                    parameter += 2; // Move to the next byte
                }

                printf("Encrypted message (HEX): ");
                print_hex_string(message, message_len);
                printf("\n");
            }

        }
        else // Encrypt
        {
```

```
        // Take message as ASCII input
        message_len = strlen(argv[arg + 1]);
        if (message_len > MAX_REQ_LEN)
        {
            printf("The message is too long, a maximum of %d bytes may be given with '-t'\n",
            ↪  MAX_REQ_LEN);
            return EXIT_FAILURE;
        }
        else
        {
            message = (unsigned char *) malloc((message_len + 17) * sizeof(unsigned char)); // + 17 if
            ↪  incomplete block to pad with zeroes

            for (i = message_len; i < message_len + 17; i++) // Fill with zeroes to pad if needed +
            ↪  null terminator
                message[i] = '\0';

            strcpy((char *) message, argv[arg + 1]); // Copy the message input
            printf("Plaintext message (ASCII): \"%s\"\n", message);
        }
    }

    arg++; // Skip over the value parameter that follows this parameter
}
else if (strstr(argv[arg], "-iv") != NULL) // Set the initialization vector
{
    args[4] = true;
    char iv[17]; // +1 for null terminator

    for (i = 0; i < 17; i++) // Fill with zeroes to pad if needed + null terminator
        iv[i] = '\0';

    strncpy(iv, argv[arg + 1], 16); // Copy the initialization vector input

    // Convert from ASCII string to int array
    for (i = 0; i < 16; i++)
        IV[i] = (unsigned char) iv[i]; // Get the integer value from the byte

    printf("Initialization Vector (ASCII): \"%s\"\n", iv);
    arg++; // Skip over the value parameter that follows this parameter
}
else if (strstr(argv[arg], "-fi") != NULL) // Read the input file
{
    args[5] = true;

    if (operation == -1)
    {
        printf("The operation type must be specified before the input file is given\n");
        printf("Specify this with '-e' for encryption or '-d' for decryption\n");
        return EXIT_FAILURE;
    }
    else
    {
        FILE *inputfileptr;
```

```
    inputfileptr = fopen(argv[arg + 1], "rb");

    if (inputfileptr == NULL)
    {
        printf("The input file specified \"%s\" could not be opened\n"
               "Make sure the file name and path is correct and that the file exists\n"
               "Give the input file in the following format: -fi <valid path to the existing
               ↪  file>\n", argv[arg + 1]);
        return EXIT_FAILURE;
    }

    fseek(inputfileptr, 0, SEEK_END); // Seek to end of file
    int size = ftell(inputfileptr); // Get current file pointer => length of the file
    fseek(inputfileptr, 0, SEEK_SET); // Seek back to beginning of file for use

    if (size < MAX_REQ_LEN)
    {
        file_output = true;
        message_len = size;

        message = (unsigned char *) malloc((message_len + 17) * sizeof(unsigned char)); // + 17 if
        ↪  incomplete block to pad with zeroes

        for (i = message_len; i < message_len + 17; i++) // Fill with zeroes to pad if needed +
        ↪  null terminator
            message[i] = '\0';

        // Take message as ASCII input
        int temp;
        for (i = 0; i < message_len; i++)
        {
            // fread( ) takes a void * as the first argument, int * has the same size as void * so
            ↪  read into an int and cast it to char
            fread(&temp, sizeof(unsigned char), 1, inputfileptr); // Read one byte from the file
            message[i] = (unsigned char) temp; // Write the byte to the message
        }

        if (operation) // Decrypt
            printf("Encrypted file input: \"%s\"\n", argv[arg + 1]);
        else // Encrypt
            printf("Plaintext file input: \"%s\"\n", argv[arg + 1]);

        fclose(inputfileptr);
    }
    else
    {
        printf("The file is too large, a maximum of %d bytes may be given with '-fi'\n",
        ↪  MAX_REQ_LEN);
        fclose(inputfileptr);
        return EXIT_FAILURE;
    }
}
```

```
    arg++; //skip over the value parameter that follows this parameter
}
else if (strstr(argv[arg], "-fo") != NULL) // Set the output file name
{
    args[6] = true;

    if (method == -1) // Method not specified yet
    {
        printf("The chaining method must be specified before the output file is given\n");
        printf("Specify this with '-cbc' for Cipher Block Chaining or '-cfb' for Cipher Feedback\n");
        return EXIT_FAILURE;
    }
    else
    {
        file_output = true;
        char *file_name = argv[arg + 1];
        char *output_file_path = NULL;

        output_file_path = (char *) malloc((strlen(argv[arg + 1]) + 1) * sizeof(char));
        output_file_path[0] = '\0';

        int pos = 0;
        while (strrchr(file_name, '/') != NULL)
        {
            output_file_path[pos++] = file_name[0]; // Retrieve the file path
            file_name++; // Move to next char
        }
        // file_name now contains the output filename

        if (output_file_path[0] != '\0')
        {
            printf("\nWARNING: A path was given with the output filename and will be discarded\n"
                   "You can find the output file in one of the output directories created\n\n");
        }
        free(output_file_path); // Discard the file path if one is given

        output_file_name = create_path(method, file_name); // Attempt to create the output directory
        // output_file_name now contains the file name + the output directory, must be freed later
    }

    arg++; //skip over the value parameter that follows this parameter
}
else if (strstr(argv[arg], "-streamlen") != NULL) // Set the stream length for CFB
{
    args[7] = true;

    if (!strcmp(argv[arg + 1], "8"))
    {
        block_len = CFB8;
        printf("8-bit CFB selected\n");
    }
    else if (!strcmp(argv[arg + 1], "64"))
    {
        block_len = CFB64;
```

```
                printf("64-bit CFB selected\n");
            }
            else if (!strcmp(argv[arg + 1], "128"))
            {
                block_len = CFB128;
                printf("128-bit CFB selected\n");
            }
            else
            {
                printf("Parameter '%s' is not a valid stream length\n", argv[arg + 1]);
                printf("Valid parameters for '-streamlen' are '8', '64' and '128'\n");
                return EXIT_FAILURE;
            }

            arg++; //skip over the value parameter that follows this parameter
        }
        else if (strstr(argv[arg], "-h") != NULL) // Show help
        {
            printf("\nUsage:\n%s", help_message);
            return EXIT_SUCCESS;
        }
#if VERBOSE
        else if (strstr(argv[arg], "-verbose") != NULL) // Enable verbose mode
        {
            verbose = true;
            printf("\nVerbose mode activated\nAll steps in the AES process will now be shown\n\n");
        }
#endif
        else
            printf("Invalid parameter: %s\n", argv[arg]);
    }

    if (!args[0] || !args[1] || !args[2] || message == NULL)
    {
        printf("All needed parameters are not given\n\nUsage:\n%s\n\n", help_message);

        printf("Do you want to perform tests? (y/n)  ");
        char c[2];
        scanf("%s", c);
        printf("\n\n");

        if (c[0] == 'y')
            test_functionality( );

        return EXIT_SUCCESS;
    }

    if (!args[4])
    {
        printf("The initialization vector was not set, setting to all zeroes\n");
        for (i = 0; i < 16; i++)
            IV[i] = 0;
    }
```

```
    if (!args[6] && file_output)
    {
        char file_name[14];

        if (operation) // Decrypt
            strcpy(file_name, "decrypted.txt");
        else // Encrypt
            strcpy(file_name, "encrypted.enc");

        output_file_name = create_path(method, file_name); // Attempt to create the output directory
        // output_file_name now contains the file name + the output directory, must be freed later

        printf("The output file is not specified, using default value of \"%s\"\n", output_file_name);
    }

    if (!args[7] && method)
        printf("The CFB stream length is not specified, using default value of 128-bits\n");

    // MAIN PROGRAM
    printf("\n\n");
    struct timeb start_time, end_time;

    if (operation) // Decrypt
    {
        printf("Decryption in process...\n\n");
        ftime(&start_time); // Get time before decryption starts

        if (method) // CFB
#if VERBOSE
            if (verbose)
                CFB_decrypt_verbose(width, message, message_len, block_len, IV, user_key);
            else
#endif
                CFB_decrypt(width, message, message_len, block_len, IV, user_key);
        else // CBC
#if VERBOSE
            if (verbose)
                CBC_decrypt_verbose(width, message, message_len, IV, user_key);
            else
#endif
                CBC_decrypt(width, message, message_len, IV, user_key);

        ftime(&end_time); // Get time after decryption ends

        // Remove trailing zeroes which could have been added in padding during encryption
        while (message[message_len] == '\0')
            message_len--;
        message_len++; // Re-add null terminator

        if (file_output)
        {
            write_to_file(output_file_name, message, message_len);
            printf("Plaintext file output: \"%s\"\n", output_file_name);
        }
```

```
        else
            printf("Decrypted (ASCII):\n\"%s\"\n\n", message);
    }
    else // Encrypt
    {
        printf("Encryption in process...\n\n");
        ftime(&start_time); // Get time before encryption starts

        if (method) // CFB
#if VERBOSE
            if (verbose)
                CFB_encrypt_verbose(width, message, message_len, block_len, IV, user_key);
            else
#endif
                CFB_encrypt(width, message, message_len, block_len, IV, user_key);
        else // CBC
#if VERBOSE
            if (verbose)
                CBC_encrypt_verbose(width, message, message_len, IV, user_key);
            else
#endif
                CBC_encrypt(width, message, message_len, IV, user_key);

        ftime(&end_time); // Get time after encryption ends

        // Determine the new message length
        int num_blocks = message_len / block_len;
        if (message_len % block_len != 0)
            num_blocks++;
        message_len = num_blocks * block_len; // New message length may be larger due to the required block
        ↪  size

        if (file_output)
        {
            write_to_file(output_file_name, message, message_len);
            printf("Encrypted file output: \"%s\"\n", output_file_name);
        }
        else
        {
            printf("Encrypted (HEX):\n");
            print_hex_string(message, message_len);
            printf("\n\n");
        }
    }

    // Calculate time elapsed in ms and print
    int elapsed_time = (int) (1000.0 * (end_time.time - start_time.time) + (end_time.millitm -
    ↪  start_time.millitm));
    printf("Operation took %u ms\n\n", elapsed_time);

    // Free dynamically allocated memory
    free(message);
    free(output_file_name);
```

```
    return EXIT_SUCCESS;
}


// Convert a char array to 4x4 block of hex
void char_blockify(unsigned char message[], int current_block[4][4], int start_pos)
{
    int byte_pos = start_pos;
    int row, col;
    for (col = 0; col < 4; col++)
    {
        for (row = 0; row < 4; row++)
            current_block[row][col] = message[byte_pos++];
    }
}


// Convert an integer array to 4x4 block of hex
void int_blockify(int message[16], int current_block[4][4])
{
    int row, col;
    for (col = 0; col < 4; col++)
    {
        for (row = 0; row < 4; row++)
            current_block[row][col] = message[row + (4 * col)];
    }
}


// Output a word to the terminal
void print_word(int word[], int length)
{
    int i;
    for (i = 0; i < length; i++)
        printf("%02X ", word[i]);
    printf("\n");
}


// Output a 4x4 block to the terminal as a block of hex
void print_block(int current_block[4][4])
{
    int row;
    for (row = 0; row < 4; row++)
        print_word(current_block[row], 4);
    printf("\n");
}


// Output the expanded key in rows of 16
void print_expanded_key(int width, int expanded_key[])
{
    int key_size;
```

```
    if (width == AES128)
        key_size = AES128_KEY_SIZE;
    else if (width == AES192)
        key_size = AES192_KEY_SIZE;
    else if (width == AES256)
        key_size = AES256_KEY_SIZE;
    else
        return;

    int i;
    for (i = 0; i < key_size; i += 16)
        print_word(expanded_key + i, 16);
    printf("\n");
}


// Print a c-string up to a certain length in hex
void print_hex_string(unsigned char hex_string[], int message_len)
{
    int i;
    for (i = 0; i < message_len; i++)
        printf("%02X", hex_string[i]);
}


// Write a message to a file
void write_to_file(char filename[], unsigned char message[], int message_len)
{
    FILE *outputfileptr;
    outputfileptr = fopen(filename, "wb");

    int i;
    int temp;
    for (i = 0; i < message_len; i++)
    {
        // fwrite( ) takes a void * as the first argument, int * has the same size as void * so cast the char to
        ↪  int to be written
        temp = (int) message[i]; // Get one char from the message
        fwrite(&temp, sizeof(unsigned char), 1, outputfileptr); // Write one byte to the file
    }

    fclose(outputfileptr);
}

// Create the output directory and return the full file path
char *create_path(int method, char *file_name)
{
    char output_file_path[11];
    char *output_file_name;

    if (method) // CFB
        strcpy(output_file_path, "CFB output");
    else // CBC
        strcpy(output_file_path, "CBC output");
```

```
    struct stat st = {0};
    int mkdir_result = 0;
    if (stat(output_file_path, &st) == -1) // Check if directory exists
        mkdir_result = mkdir(output_file_path, 0700); // Attempt to create directory if not existing

    if (mkdir_result < 0) // Could not create path
    {
        printf("\nERROR: The output directory \"%s\" could not be accessed\n"
               "It seems that this program does not have sufficient privileges to create directories, "
               "so the output will appear in the same directory as the executable\n\n", output_file_path);

        output_file_name = (char *) malloc((strlen(file_name) + 1) * sizeof(char));
        strcpy(output_file_name, file_name); // Copy only the filename
    }
    else // Path exists
    {
        // setup the path and filename for the desired output
        int len = strlen(output_file_path) + strlen(file_name) + 2;
        output_file_name = (char *) malloc(len * sizeof(char));
        strcpy(output_file_name, output_file_path); // Copy the path
        strcat(output_file_name, "/");
        strcat(output_file_name, file_name); // Copy the filename
    }

    return output_file_name;
}


// Convert block back to c-string
void char_unblockify(unsigned char message[], int current_block[4][4], int start_pos)
{
    int byte_pos = start_pos;
    int row, col;
    for (col = 0; col < 4; col++)
    {
        for (row = 0; row < 4; row++)
            message[byte_pos++] = current_block[row][col];
    }
}


// Shift first items in an array to the back or vice-versa
void AES_word_rotate(int word[], int length, int rotations, bool inverse)
{
    int old[length];
    int pivot = length - rotations;

    int i;
    for (i = 0; i < length; i++) // Set the old values aside for retrieval
        old[i] = word[i];

    if (inverse) // Shift items in the back to the front
    {
```

```
        for (i = 0; i < pivot; i++) // Populate the back with the items from the front
            word[i + rotations] = old[i];

        for (i = pivot; i < length; i++) // Populate the front with the other values from the back
            word[i - pivot] = old[i];
    }
    else // Shift the items in the front to the back
    {
        for (i = 0; i < pivot; i++) // Populate the front with the items from the back
            word[i] = old[i + rotations];

        for (i = pivot; i < length; i++) // Populate the back with the other values at the front
            word[i] = old[i - pivot];
    }
}


// Divide value up into its MSB and LSB Nibble and return the s_box value
int AES_s_box_transform(int input, bool inverse)
{   //           0 or 1      MSB             LSB
    return S_BOX[inverse][input >> 4][input & 0b00001111];
}


// Core key operation, transform of previous 4 bytes
void AES_key_scheduler(int word[4], int rcon)
{
    int byte_pos;
    AES_word_rotate(word, 4, 1, false); // Rotate the word
    for (byte_pos = 0; byte_pos < 4; byte_pos++) // Take the S-transform of the word
        word[byte_pos] = AES_s_box_transform(word[byte_pos], false);
    word[0] ^= rcon; // Add the round constant
}


// Exponentiation of 2, double the previous value except when 0x80 and max value of 0xFF
int AES_exp_2(int previous)
{
    if (previous == 0x80)
        return 0x1B;
    else if (previous * 2 >= 0xFF)
        return 0xFF;
    else
        return previous * 2;
}


// Main key expansion function
void AES_key_expansion(int width, int expanded_key[], int user_key[])
{
    /* !!!! WARNING !!!!
     * This function deliberately writes out of bounds (this is how it was created)
     * Be sure to allocate the correct expanded key size + 32 to be passed in for the expanded_key
     */
```

```
int user_key_size;
int expansion;
int sub_expansion;

if (width == AES128)
{
    expansion = AES128_EXPANSION;
    sub_expansion = AES128_SUB_EXPANSION;
    user_key_size = AES128_USER_KEY_SIZE;
}
else if (width == AES192)
{
    expansion = AES192_EXPANSION;
    sub_expansion = AES192_SUB_EXPANSION;
    user_key_size = AES192_USER_KEY_SIZE;
}
else if (width == AES256)
{
    expansion = AES256_EXPANSION;
    sub_expansion = AES256_SUB_EXPANSION;
    user_key_size = AES256_USER_KEY_SIZE;
}
else
    return;

int byte_pos;
int temp[4];

// Set first x bytes as the user key
int key_pos;
for (key_pos = 0; key_pos < user_key_size; key_pos++)
    expanded_key[key_pos] = user_key[key_pos];

// Last 4 bits into temp
for (key_pos = 0; key_pos < 4; key_pos++)
    temp[key_pos] = user_key[user_key_size - (4 - key_pos)];

// Fill the expanded key until the required length is reached
int expanded_pos;
int sub_pos;
int rcon = 1;
for (expanded_pos = 0; expanded_pos < expansion; expanded_pos++)
{
    AES_key_scheduler(temp, rcon);
    rcon = AES_exp_2(rcon);

    for (byte_pos = 0; byte_pos < 4; byte_pos++)
    {
        temp[byte_pos] ^= expanded_key[byte_pos + (user_key_size * expanded_pos)]; // Bitwise XOR with x
        ↪  bytes before
        expanded_key[byte_pos + user_key_size + (user_key_size * expanded_pos)] = temp[byte_pos]; // Expand
        ↪  key
    }
```

```
        // Perform the sub-expansion (will deliberately write out of bounds on last iteration)
        for (sub_pos = 0; sub_pos < sub_expansion; sub_pos++)
        {
            for (byte_pos = 0; byte_pos < 4; byte_pos++)
            {
                temp[byte_pos] ^= expanded_key[byte_pos + 4 + (user_key_size * expanded_pos) + (4 * sub_pos)];
                ↪  // Bitwise XOR with x bytes before
                expanded_key[byte_pos + 4 + user_key_size + (user_key_size * expanded_pos) + (4 * sub_pos)] =
                ↪  temp[byte_pos]; // Expand key
            }
        }
    }
}


// Substitute a block through the S-transform
void AES_sub_bytes(int current_block[4][4], bool inverse)
{
    int row, col;
    for (col = 0; col < 4; col++)
    {
        for (row = 0; row < 4; row++) // Perform S-transform on every byte
            current_block[row][col] = AES_s_box_transform(current_block[row][col], inverse);
    }
}


// The AES row shifting function
void AES_shift_rows(int current_block[4][4], bool inverse)
{
    /*
     * Rotate each word by the number of times equal to its index, i.e.
     * Row 0 stays the same
     * Row 1 is shifted once
     * Row 2 is shifted twice
     * Row 3 is shifted thrice
     */

    int row;
    for (row = 1; row < 4; row++)
        AES_word_rotate(current_block[row], 4, row, inverse);
}


// Finite field multiplication
int AES_dot_product(int a, int b)
{
    /*
     * Represent both numbers as polynomials, i.e.
     * 0b00000000 = 0
     * 0b00000001 = 1
     * 0b00000010 = x
     * 0b00000100 = x^2
```

```
     * ...
     * 0b1...      = x^n
     *
     * XOR pairs to put together, i.e.
     * 0b101101 = x^5 + x^3 + x^2 + 1
     */

    int i;
    int result = 0;

    // Expand polynomial
    /*
     * (polynomial a) * (polynomial b)
     * Multiplying a polynomial by x^n is equal to a n left shift
     * XOR the resulting polynomials together
     */
    int position = 128; // = 2^7 = 0b10000000 => x^7
    for (i = 7; i >= 0; i--)
    {
        if ((a & position) == position) // See if (a) has the power of x currently looked at
            result ^= b << i; // Shift (b) left by the power of x currently looked at if present
        position = position >> 1; // Make power of x one smaller
    }

    if (result < 0xFF) // Already smaller, modulo is the result
        return result;

    // Calculate modulo
    // Polynomial long division with irreducible polynomial x^8 + x^4 + x^3 + x + 1 => 0b100011011
    position = 65536; // = 2^16 = 0b1000000000000000 => x^16
    for (i = 16; i > 7; i--)
    {
        if ((result & position) == position) // See if (result) has the power of x currently looked at
            result ^= 0b100011011 << (i - 8); // Subtract a multiple of the irreducible polynomial if present
        position = position >> 1; // Make power of x one smaller
    }

    return result; // Remainder after long division was done
}


// Perform the dot product of the block and the prime matrix
void AES_mix_cols(int current_block[4][4], bool inverse)
{
    // Matrix dot operation with the prime matrix
    int row, col, out;
    int new_block[4][4];
    int multiply[4];
    for (out = 0; out < 4; ++out)
    {
        for (row = 0; row < 4; row++)
        {
            for (col = 0; col < 4; col++) // Calculate sub dot products
                multiply[col] = AES_dot_product(PRIME_MATRIX[inverse][row][col], current_block[col][out]);
```

```
            new_block[row][out] = multiply[0] ^ multiply[1] ^ multiply[2] ^ multiply[3]; // Add sub dot
            ↪  products together
        }
    }

    // Copy the result over the input as output
    for (row = 0; row < 4; row++)
    {
        for (col = 0; col < 4; col++)
            current_block[row][col] = new_block[row][col];
    }
}


// XOR a block with the expanded key at a certain index
void AES_add_round_key(int current_block[4][4], int expanded_key[], int key_index)
{
    int col, row;
    for (col = 0; col < 4; col++)
    {
        for (row = 0; row < 4; row++) // Do column wise XOR with the matching index of the key
            current_block[row][col] ^= expanded_key[row + (col * 4) + key_index];
    }
}


// The AES encryption algorithm
bool AES_encrypt(int width, int current_block[4][4], int expanded_key[])
{
    int number_of_rounds;

    if (width == AES128)
        number_of_rounds = AES128_ROUNDS;
    else if (width == AES192)
        number_of_rounds = AES192_ROUNDS;
    else if (width == AES256)
        number_of_rounds = AES256_ROUNDS;
    else
        return EXIT_FAILURE;

    int key_index = 0; // Start at the beginning of the key and work forwards

    // Initial round, add round key
    AES_add_round_key(current_block, expanded_key, key_index);

    // Perform the normal rounds
    int round;
    for (round = 0; round < number_of_rounds - 1; round++)
    {
        key_index += 16; // Move to next section
        AES_sub_bytes(current_block, false); // Substitute bytes
        AES_shift_rows(current_block, false); // Shift rows
        AES_mix_cols(current_block, false); // Mix columns
        AES_add_round_key(current_block, expanded_key, key_index); // Add round key
```

```
    }

    // Last round is a special case
    key_index += 16; // Move to the last section
    AES_sub_bytes(current_block, false); // Substitute bytes
    AES_shift_rows(current_block, false); // Shift rows
    // No mix columns
    AES_add_round_key(current_block, expanded_key, key_index); // Add round key

    return EXIT_SUCCESS;
}


// The AES decryption algorithm
bool AES_decrypt(int width, int current_block[4][4], int expanded_key[])
{
    int number_of_rounds;
    int key_size;

    if (width == AES128)
    {
        number_of_rounds = AES128_ROUNDS;
        key_size = AES128_KEY_SIZE;
    }
    else if (width == AES192)
    {
        number_of_rounds = AES192_ROUNDS;
        key_size = AES192_KEY_SIZE;
    }
    else if (width == AES256)
    {
        number_of_rounds = AES256_ROUNDS;
        key_size = AES256_KEY_SIZE;
    }
    else
        return EXIT_FAILURE;

    int key_index = key_size - 16; // Start at the end of the key and work backwards

    // Initial round, add round key
    AES_add_round_key(current_block, expanded_key, key_index);

    // Perform the normal rounds
    int round;
    for (round = 0; round < number_of_rounds - 1; round++)
    {
        key_index -= 16; // Move to previous section
        AES_shift_rows(current_block, true); // Inverse shift rows
        AES_sub_bytes(current_block, true); // Inverse substitute bytes
        AES_add_round_key(current_block, expanded_key, key_index); // Add round key
        AES_mix_cols(current_block, true); // Inverse mix columns
    }

    // Last round is a special case
```

```
    key_index -= 16; // Move to the first section
    AES_shift_rows(current_block, true); // Inverse shift rows
    AES_sub_bytes(current_block, true); // Inverse substitute bytes
    AES_add_round_key(current_block, expanded_key, key_index); // Add round key
    // No mix columns

    return EXIT_SUCCESS;
}


// The Cipher Block Chaining encryption
bool CBC_encrypt(int width, unsigned char message[], int message_len, int IV[16], int user_key[])
{
    /* !!!! WARNING !!!!
     * The length of the message array should be large enough to accommodate any possible expansion that could
↪   take place
     * due to the block size requirements. It is recommended to allocate 16 extra characters at the end of
↪   message
     * to avoid writing out of bounds.
     */

    int key_size;

    if (width == AES128)
        key_size = AES128_KEY_SIZE;
    else if (width == AES192)
        key_size = AES192_KEY_SIZE;
    else if (width == AES256)
        key_size = AES256_KEY_SIZE;
    else
        return EXIT_FAILURE;

    int expanded_key[key_size + 32]; // Allocate more space since AES_key_expansion deliberately writes out of
    ↪   bounds
    AES_key_expansion(width, expanded_key, user_key);

    int i;
    int current_block[4][4];
    int current_vector[16];

    // Copy IV to not change its contents
    for (i = 0; i < 16; i++)
        current_vector[i] = IV[i];

    int message_pos;
    for (message_pos = 0; message_pos < message_len; message_pos += 16)
    {
        // XOR current vector with plaintext
        for (i = 0; i < 16; i++)
            message[message_pos + i] ^= current_vector[i];

        // Convert the encryption input to a block
        char_blockify(message, current_block, message_pos);
```

```
        // Encrypt to produce ciphertext block
        AES_encrypt(width, current_block, expanded_key);

        // Convert the block back to the string
        char_unblockify(message, current_block, message_pos);

        // Update current vector with ciphertext values
        for (i = 0; i < 16; i++)
            current_vector[i] = message[message_pos + i];
    }

    return EXIT_SUCCESS;
}


// The Cipher Block Chaining decryption
bool CBC_decrypt(int width, unsigned char message[], int message_len, int IV[16], int user_key[])
{
    int key_size;

    if (width == AES128)
        key_size = AES128_KEY_SIZE;
    else if (width == AES192)
        key_size = AES192_KEY_SIZE;
    else if (width == AES256)
        key_size = AES256_KEY_SIZE;
    else
        return EXIT_FAILURE;

    int expanded_key[key_size + 32]; // Allocate more space since AES_key_expansion deliberately writes out of
    ↪  bounds
    AES_key_expansion(width, expanded_key, user_key);

    int i;
    int current_block[4][4];
    int current_vector[2][16]; // current_vector[vector] => current, current_vector[!vector] => old
    bool vector = false; // For array switching, no array copy needed then

    // Copy IV to not change its contents
    for (i = 0; i < 16; i++)
        current_vector[vector][i] = IV[i];

    int message_pos;
    for (message_pos = 0; message_pos < message_len; message_pos += 16)
    {
        // Copy current ciphertext values to the old vector
        for (i = 0; i < 16; i++)
            current_vector[!vector][i] = message[message_pos + i];

        // Convert the decryption input to a block
        char_blockify(message, current_block, message_pos);

        // Decrypt the block
        AES_decrypt(width, current_block, expanded_key);
```

```
        // Convert the block back to the string
        char_unblockify(message, current_block, message_pos);

        // XOR current vector with decrypted text to produce plaintext
        for (i = 0; i < 16; i++)
            message[message_pos + i] ^= current_vector[vector][i];

        vector = !vector; // Switch the arrays so that the old vector becomes the current one
    }

    return EXIT_SUCCESS;
}


// The Cipher Feedback encryption algorithm
bool CFB_encrypt(int width, unsigned char message[], int message_len, int CFB_len, int IV[16], int user_key[])
{
    /* !!!! WARNING !!!!
     * The length of the message array should be large enough to accommodate any possible expansion that could
↪   take place
     * due to the block size requirements. It is recommended to allocate 16 extra characters at the end of
↪   message
     * to avoid writing out of bounds.
     */

    int key_size;

    if (width == AES128)
        key_size = AES128_KEY_SIZE;
    else if (width == AES192)
        key_size = AES192_KEY_SIZE;
    else if (width == AES256)
        key_size = AES256_KEY_SIZE;
    else
        return EXIT_FAILURE;

    int expanded_key[key_size + 32]; // Allocate more space since AES_key_expansion deliberately writes out of
    ↪   bounds
    AES_key_expansion(width, expanded_key, user_key);

    int i;
    int CFB_back = 16 - CFB_len; // Length from the back
    int current_block[4][4];
    int current_vector[16]; // AES requires 128 bit input
    unsigned char current_string[16];

    // Copy IV to not change its contents
    for (i = 0; i < 16; i++)
        current_vector[i] = IV[i];

    int message_pos;
    for (message_pos = 0; message_pos < message_len; message_pos += CFB_len)
    {
```

```
        // Convert the encryption input to a block
        int_blockify(current_vector, current_block);

        // Encrypt the block
        AES_encrypt(width, current_block, expanded_key);

        // Take first CFB_len bytes in the block and XOR with the plaintext bytes to get the ciphertext bytes
        char_unblockify(current_string, current_block, 0);
        for (i = 0; i < CFB_len; i++)
            message[message_pos + i] ^= current_string[i];
        // Discard the rest of the block

        // Shift the current vector to the left by CFB_len bytes
        for (i = 0; i < CFB_back; i++)
            current_vector[i] = current_vector[i + CFB_len];

        // Put the ciphertext bytes in the last CFB_len bytes
        for (i = 0; i < CFB_len; i++)
            current_vector[i + CFB_back] = message[message_pos + i];
    }

    return EXIT_SUCCESS;
}


// The Cipher Feedback decryption algorithm
bool CFB_decrypt(int width, unsigned char message[], int message_len, int CFB_len, int IV[16], int user_key[])
{
    int key_size;

    if (width == AES128)
        key_size = AES128_KEY_SIZE;
    else if (width == AES192)
        key_size = AES192_KEY_SIZE;
    else if (width == AES256)
        key_size = AES256_KEY_SIZE;
    else
        return EXIT_FAILURE;

    int expanded_key[key_size + 32]; // Allocate more space since AES_key_expansion deliberately writes out of
    ↪  bounds
    AES_key_expansion(width, expanded_key, user_key);

    int i;
    int CFB_back = 16 - CFB_len;
    int current_block[4][4];
    int current_vector[16]; // AES requires 128 bit input
    unsigned char current_string[16];

    // Copy IV to not change its contents
    for (i = 0; i < 16; i++)
        current_vector[i] = IV[i];

    int message_pos;
```

```
    for (message_pos = 0; message_pos < message_len; message_pos += CFB_len)
    {
        // Convert the encryption input to a block
        int_blockify(current_vector, current_block);

        // Encrypt the block
        AES_encrypt(width, current_block, expanded_key);

        // Shift the current vector to the left by CFB_len bytes
        for (i = 0; i < CFB_back; i++)
            current_vector[i] = current_vector[i + CFB_len];

        // Put the ciphertext bytes in the last CFB_len bytes
        for (i = 0; i < CFB_len; i++)
            current_vector[i + CFB_back] = message[message_pos + i];

        // Take first CFB_len bytes in the block and XOR with the ciphertext bytes to get the plaintext bytes
        char_unblockify(current_string, current_block, 0);
        for (i = 0; i < CFB_len; i++)
            message[message_pos + i] ^= current_string[i];
        // Discard the rest of the block
    }

    return EXIT_SUCCESS;
}

#if VERBOSE
// The AES encryption algorithm, but all steps will be printed
bool AES_encrypt_verbose(int width, int current_block[4][4], int expanded_key[])
{
    int number_of_rounds;

    if (width == AES128)
        number_of_rounds = AES128_ROUNDS;
    else if (width == AES192)
        number_of_rounds = AES192_ROUNDS;
    else if (width == AES256)
        number_of_rounds = AES256_ROUNDS;
    else
        return EXIT_FAILURE;

    int key_index = 0; // Start at the beginning of the key and work forwards

    printf("\n~~~~AES encrypt input block:~~~~\n");
    print_block(current_block);

    // Initial round, add round key
    AES_add_round_key(current_block, expanded_key, key_index);
    printf("Add round key (initial):\n");
    print_block(current_block);

    // Perform the normal rounds
    int round;
    for (round = 0; round < number_of_rounds - 1; round++)
```

```
    {
        key_index += 16; // Move to next section
        printf("\n----Round %d:----\n", round + 1);

        AES_sub_bytes(current_block, false); // Substitute bytes
        printf("Substitute bytes step:\n");
        print_block(current_block);

        AES_shift_rows(current_block, false); // Shift rows
        printf("Shift rows step:\n");
        print_block(current_block);

        AES_mix_cols(current_block, false); // Mix columns
        printf("Mix columns step:\n");
        print_block(current_block);

        AES_add_round_key(current_block, expanded_key, key_index); // Add round key
        printf("Add round key step:\n");
        print_block(current_block);
    }

    // Last round is a special case
    key_index += 16; // Move to the last section
    printf("\n----Last round:----\n");

    AES_sub_bytes(current_block, false); // Substitute bytes
    printf("Substitute bytes step:\n");
    print_block(current_block);

    AES_shift_rows(current_block, false); // Shift rows
    printf("Shift rows step:\n");
    print_block(current_block);

    // No mix columns
    printf("No mix columns step in the last round\n\n");

    AES_add_round_key(current_block, expanded_key, key_index); // Add round key
    printf("Add round key step:\n");
    print_block(current_block);

    return EXIT_SUCCESS;
}


// The AES decryption algorithm, but all steps will be printed
bool AES_decrypt_verbose(int width, int current_block[4][4], int expanded_key[])
{
    int number_of_rounds;
    int key_size;

    if (width == AES128)
    {
        number_of_rounds = AES128_ROUNDS;
        key_size = AES128_KEY_SIZE;
```

```
}
else if (width == AES192)
{
    number_of_rounds = AES192_ROUNDS;
    key_size = AES192_KEY_SIZE;
}
else if (width == AES256)
{
    number_of_rounds = AES256_ROUNDS;
    key_size = AES256_KEY_SIZE;
}
else
    return EXIT_FAILURE;

int key_index = key_size - 16; // Start at the end of the key and work backwards

printf("\n~~~~AES decrypt input block:~~~~\n");
print_block(current_block);

// Initial round, add round key
AES_add_round_key(current_block, expanded_key, key_index);
printf("Add round key (initial):\n");
print_block(current_block);

// Perform the normal rounds
int round;
for (round = 0; round < number_of_rounds - 1; round++)
{
    key_index -= 16; // Move to previous section
    printf("\n----Round %d:----\n", round + 1);

    AES_shift_rows(current_block, true); // Inverse shift rows
    printf("Inverse shift rows step:\n");
    print_block(current_block);

    AES_sub_bytes(current_block, true); // Inverse substitute bytes
    printf("Inverse substitute bytes step:\n");
    print_block(current_block);

    AES_add_round_key(current_block, expanded_key, key_index); // Add round key
    printf("Add round key step:\n");
    print_block(current_block);

    AES_mix_cols(current_block, true); // Inverse mix columns
    printf("Inverse mix columns step:\n");
    print_block(current_block);
}

// Last round is a special case
key_index -= 16; // Move to the first section
printf("\n----Last round:----\n");

AES_shift_rows(current_block, true); // Inverse shift rows
printf("Inverse shift rows step:\n");
```

```
    print_block(current_block);

    AES_sub_bytes(current_block, true); // Inverse substitute bytes
    printf("Inverse substitute bytes step:\n");
    print_block(current_block);

    AES_add_round_key(current_block, expanded_key, key_index); // Add round key
    printf("Add round key step:\n");
    print_block(current_block);

    // No mix columns
    printf("No inverse mix columns step in the last round\n\n");

    return EXIT_SUCCESS;
}


// The Cipher Block Chaining encryption
bool CBC_encrypt_verbose(int width, unsigned char message[], int message_len, int IV[16], int user_key[])
{
    /* !!!! WARNING !!!!
     * The length of the message array should be large enough to accommodate any possible expansion that could
↪   take place
     * due to the block size requirements. It is recommended to allocate 16 extra characters at the end of
↪   message
     * to avoid writing out of bounds.
     */

    int key_size;

    if (width == AES128)
        key_size = AES128_KEY_SIZE;
    else if (width == AES192)
        key_size = AES192_KEY_SIZE;
    else if (width == AES256)
        key_size = AES256_KEY_SIZE;
    else
        return EXIT_FAILURE;

    int expanded_key[key_size + 32]; // Allocate more space since AES_key_expansion deliberately writes out of
    ↪  bounds
    AES_key_expansion(width, expanded_key, user_key);

    int i;
    int current_block[4][4];
    int current_vector[16];

    // Copy IV to not change its contents
    for (i = 0; i < 16; i++)
        current_vector[i] = IV[i];

    int message_pos;
    for (message_pos = 0; message_pos < message_len; message_pos += 16)
    {
```

```
        printf("\n\n\n********Block %d:********\n", (message_pos / 16) + 1);

        // XOR current vector with plaintext
        for (i = 0; i < 16; i++)
            message[message_pos + i] ^= current_vector[i];

        // Convert the encryption input to a block
        char_blockify(message, current_block, message_pos);

        // Encrypt to produce ciphertext block
        AES_encrypt_verbose(width, current_block, expanded_key);

        // Convert the block back to the string
        char_unblockify(message, current_block, message_pos);

        // Update current vector with ciphertext values
        for (i = 0; i < 16; i++)
            current_vector[i] = message[message_pos + i];
    }

    printf("\n\n********Expanded key:********\n");
    print_expanded_key(width, expanded_key);

    return EXIT_SUCCESS;
}


// The Cipher Block Chaining decryption
bool CBC_decrypt_verbose(int width, unsigned char message[], int message_len, int IV[16], int user_key[])
{
    int key_size;

    if (width == AES128)
        key_size = AES128_KEY_SIZE;
    else if (width == AES192)
        key_size = AES192_KEY_SIZE;
    else if (width == AES256)
        key_size = AES256_KEY_SIZE;
    else
        return EXIT_FAILURE;

    int expanded_key[key_size + 32]; // Allocate more space since AES_key_expansion deliberately writes out of
    ↪  bounds
    AES_key_expansion(width, expanded_key, user_key);

    int i;
    int current_block[4][4];
    int current_vector[2][16]; // current_vector[vector] => current, current_vector[!vector] => old
    bool vector = false; // For array switching, no array copy needed then

    // Copy IV to not change its contents
    for (i = 0; i < 16; i++)
        current_vector[vector][i] = IV[i];
```

```
    int message_pos;
    for (message_pos = 0; message_pos < message_len; message_pos += 16)
    {
        printf("\n\n\n********Block %d:********\n", (message_pos / 16) + 1);

        // Copy current ciphertext values to the old vector
        for (i = 0; i < 16; i++)
            current_vector[!vector][i] = message[message_pos + i];

        // Convert the decryption input to a block
        char_blockify(message, current_block, message_pos);

        // Decrypt the block
        AES_decrypt_verbose(width, current_block, expanded_key);

        // Convert the block back to the string
        char_unblockify(message, current_block, message_pos);

        // XOR current vector with decrypted text to produce plaintext
        for (i = 0; i < 16; i++)
            message[message_pos + i] ^= current_vector[vector][i];

        vector = !vector; // Switch the arrays so that the old vector becomes the current one
    }

    printf("\n\n********Expanded key:********\n");
    print_expanded_key(width, expanded_key);

    return EXIT_SUCCESS;
}


// The Cipher Feedback encryption algorithm
bool CFB_encrypt_verbose(int width, unsigned char message[], int message_len, int CFB_len, int IV[16], int
↪  user_key[])
{
    /* !!!! WARNING !!!!
     * The length of the message array should be large enough to accommodate any possible expansion that could
↪  take place
     * due to the block size requirements. It is recommended to allocate 16 extra characters at the end of
↪  message
     * to avoid writing out of bounds.
     */

    int key_size;

    if (width == AES128)
        key_size = AES128_KEY_SIZE;
    else if (width == AES192)
        key_size = AES192_KEY_SIZE;
    else if (width == AES256)
        key_size = AES256_KEY_SIZE;
    else
        return EXIT_FAILURE;
```

```
    int expanded_key[key_size + 32]; // Allocate more space since AES_key_expansion deliberately writes out of
    ↪  bounds
    AES_key_expansion(width, expanded_key, user_key);

    int i;
    int CFB_back = 16 - CFB_len; // Length from the back
    int current_block[4][4];
    int current_vector[16]; // AES requires 128 bit input
    unsigned char current_string[16];

    // Copy IV to not change its contents
    for (i = 0; i < 16; i++)
        current_vector[i] = IV[i];

    int message_pos;
    for (message_pos = 0; message_pos < message_len; message_pos += CFB_len)
    {
        printf("\n\n\n********Block %d:********\n", (message_pos / CFB_len) + 1);

        // Convert the encryption input to a block
        int_blockify(current_vector, current_block);

        // Encrypt the block
        AES_encrypt_verbose(width, current_block, expanded_key);

        // Take first CFB_len bytes in the block and XOR with the plaintext bytes to get the ciphertext bytes
        char_unblockify(current_string, current_block, 0);
        for (i = 0; i < CFB_len; i++)
            message[message_pos + i] ^= current_string[i];
        // Discard the rest of the block

        // Shift the current vector to the left by CFB_len bytes
        for (i = 0; i < CFB_back; i++)
            current_vector[i] = current_vector[i + CFB_len];

        // Put the ciphertext bytes in the last CFB_len bytes
        for (i = 0; i < CFB_len; i++)
            current_vector[i + CFB_back] = message[message_pos + i];
    }

    printf("\n\n********Expanded key:********\n");
    print_expanded_key(width, expanded_key);

    return EXIT_SUCCESS;
}


// The Cipher Feedback decryption algorithm
bool CFB_decrypt_verbose(int width, unsigned char message[], int message_len, int CFB_len, int IV[16], int
↪  user_key[])
{
    int key_size;
```

```
    if (width == AES128)
        key_size = AES128_KEY_SIZE;
    else if (width == AES192)
        key_size = AES192_KEY_SIZE;
    else if (width == AES256)
        key_size = AES256_KEY_SIZE;
    else
        return EXIT_FAILURE;

    int expanded_key[key_size + 32]; // Allocate more space since AES_key_expansion deliberately writes out of
    ↪  bounds
    AES_key_expansion(width, expanded_key, user_key);

    int i;
    int CFB_back = 16 - CFB_len;
    int current_block[4][4];
    int current_vector[16]; // AES requires 128 bit input
    unsigned char current_string[16];

    // Copy IV to not change its contents
    for (i = 0; i < 16; i++)
        current_vector[i] = IV[i];

    int message_pos;
    for (message_pos = 0; message_pos < message_len; message_pos += CFB_len)
    {
        printf("\n\n\n********Block %d:********\n", (message_pos / CFB_len) + 1);

        // Convert the encryption input to a block
        int_blockify(current_vector, current_block);

        // Encrypt the block
        AES_encrypt_verbose(width, current_block, expanded_key);

        // Shift the current vector to the left by CFB_len bytes
        for (i = 0; i < CFB_back; i++)
            current_vector[i] = current_vector[i + CFB_len];

        // Put the ciphertext bytes in the last CFB_len bytes
        for (i = 0; i < CFB_len; i++)
            current_vector[i + CFB_back] = message[message_pos + i];

        // Take first CFB_len bytes in the block and XOR with the ciphertext bytes to get the plaintext bytes
        char_unblockify(current_string, current_block, 0);
        for (i = 0; i < CFB_len; i++)
            message[message_pos + i] ^= current_string[i];
        // Discard the rest of the block
    }

    printf("\n\n********Expanded key:********\n");
    print_expanded_key(width, expanded_key);

    return EXIT_SUCCESS;
}
```

```
#endif


// Convert hex to int, done because the system hex converter is unreliable
int hex_convert(char hex_string[], int length)
{
    int result = 0;
    int base = 1;

    int i;
    for (i = length; i > 0; i--)
    {
        switch (hex_string[i - 1])
        {
            case '0': {break;}
            case '1': {result += base * 1; break;}
            case '2': {result += base * 2; break;}
            case '3': {result += base * 3; break;}
            case '4': {result += base * 4; break;}
            case '5': {result += base * 5; break;}
            case '6': {result += base * 6; break;}
            case '7': {result += base * 7; break;}
            case '8': {result += base * 8; break;}
            case '9': {result += base * 9; break;}
            case 'A': {result += base * 10; break;}
            case 'B': {result += base * 11; break;}
            case 'C': {result += base * 12; break;}
            case 'D': {result += base * 13; break;}
            case 'E': {result += base * 14; break;}
            case 'F': {result += base * 15; break;}
            case 'a': {result += base * 10; break;}
            case 'b': {result += base * 11; break;}
            case 'c': {result += base * 12; break;}
            case 'd': {result += base * 13; break;}
            case 'e': {result += base * 14; break;}
            case 'f': {result += base * 15; break;}
            default:
                {
                    printf("The input given (\'%c\') is not a valid HEX character\nTerminating...\n",
                    ↪  hex_string[i]);
                    exit(EXIT_FAILURE);
                }
        }

        base *= 16;
    }

    return result;
}


// Print out various tests to test the functionality of the other functions
void test_functionality( )
{
```

```
//    **** TESTING PURPOSES **** /*
int i;
int AES128_user_key[AES128_USER_KEY_SIZE] = {0x74, 0x65, 0x73, 0x74, 0x20, 0x66, 0x75, 0x6E, 0x63, 0x74,
↪  0x69, 0x6F,
                                             0x6E, 0x61, 0x6C, 0x69};
int AES192_user_key[AES192_USER_KEY_SIZE] = {0x74, 0x65, 0x73, 0x74, 0x20, 0x66, 0x75, 0x6E, 0x63, 0x74,
↪  0x69, 0x6F,
                                             0x6E, 0x61, 0x6C, 0x69, 0x74, 0x65, 0x73, 0x74, 0x20, 0x66,
                                             ↪  0x75, 0x6E};
int AES256_user_key[AES256_USER_KEY_SIZE] = {0x74, 0x65, 0x73, 0x74, 0x20, 0x66, 0x75, 0x6E, 0x63, 0x74,
↪  0x69, 0x6F,
                                             0x6E, 0x61, 0x6C, 0x69, 0x74, 0x65, 0x73, 0x74, 0x20, 0x66,
                                             ↪  0x75, 0x6E,
                                             0x63, 0x74, 0x69, 0x6F, 0x6E, 0x61, 0x6C, 0x69};
int AES128_expanded_key[AES128_KEY_SIZE + 32];
int AES192_expanded_key[AES192_KEY_SIZE + 32];
int AES256_expanded_key[AES256_KEY_SIZE + 32];
int test_word[4] = {0x3A, 0x65, 0x71, 0x1B};
int x;
int test_cols[4][4] = {{0x74, 0x20, 0x61, 0x73},
                       {0x68, 0x69, 0x20, 0x74},
                       {0x69, 0x73, 0x74, 0x2E},
                       {0x73, 0x20, 0x65, 0x2E}};

printf("----Testing word rotate----\n");
printf("Original\n");
print_word(test_word, 4);
AES_word_rotate(test_word, 4, 1, false);
printf("\nOne rotation\n");
print_word(test_word, 4);
AES_word_rotate(test_word, 4, 1, true);
printf("\nOne inverse rotation\n");
print_word(test_word, 4);
AES_word_rotate(test_word, 4, 2, false);
printf("\nTwo rotations\n");
print_word(test_word, 4);
AES_word_rotate(test_word, 4, 2, true);
printf("\nTwo inverse rotations\n");
print_word(test_word, 4);
AES_word_rotate(test_word, 4, 3, false);
printf("\nThree rotations\n");
print_word(test_word, 4);
AES_word_rotate(test_word, 4, 3, true);
printf("\nThree inverse rotations\n");
print_word(test_word, 4);
printf("\n\n");

printf("----Testing S-transform----\n");
printf("Original\n3A\n");
x = AES_s_box_transform(0x3A, false);
printf("\nS-transformed\n%02X\n", x);
x = AES_s_box_transform(x, true);
printf("\nInverse s-transformed\n%02X\n", x);
printf("\n\n");
```

```
printf("----Testing key scheduler----\n");
printf("Original\n");
print_word(test_word, 4);
AES_key_scheduler(test_word, 1);
printf("\nKey scheduled with rcon = 1\n");
print_word(test_word, 4);
printf("\n\n");

// Test cols changed by key scheduler
test_word[0] = 0x3A;
test_word[1] = 0x65;
test_word[2] = 0x71;
test_word[3] = 0x1B;

printf("----Testing exponentiation starting from 1----\n01 ");
x = 1;
for (i = 0; i < 20; i++)
    printf("%02X ", x = AES_exp_2(x));
printf("\n\n\n");

printf("----Testing key expansion----\n");
printf("AES128 expanded key\n");
AES_key_expansion(AES128, AES128_expanded_key, AES128_user_key);
print_expanded_key(AES128, AES128_expanded_key);
printf("AES192 expanded key\n");
AES_key_expansion(AES192, AES192_expanded_key, AES192_user_key);
print_expanded_key(AES192, AES192_expanded_key);
printf("AES256 expanded key\n");
AES_key_expansion(AES256, AES256_expanded_key, AES256_user_key);
print_expanded_key(AES256, AES256_expanded_key);
printf("\n");

printf("----Testing substitute bytes----\n");
printf("Original\n");
print_block(test_cols);
AES_sub_bytes(test_cols, false);
printf("Sub bytes\n");
print_block(test_cols);
AES_sub_bytes(test_cols, true);
printf("Inverse sub bytes should be same as original\n");
print_block(test_cols);
printf("\n");

printf("----Testing shift rows----\n");
printf("Original\n");
print_block(test_cols);
AES_shift_rows(test_cols, false);
printf("Shift rows\n");
print_block(test_cols);
AES_shift_rows(test_cols, true);
printf("Inverse shift rows should be same as original\n");
print_block(test_cols);
printf("\n");
```

```
printf("----Testing dot product----\n");
printf("57 dot 83 = ");
x = AES_dot_product(0x57, 0x83);
printf("%02X\n", x);
printf("83 dot 57 = ");
x = AES_dot_product(0x83, 0x57);
printf("%02X\n\n\n", x);

printf("----Testing mix cols----\n");
printf("Original\n");
print_block(test_cols);
AES_mix_cols(test_cols, false);
printf("Mix cols\n");
print_block(test_cols);
AES_mix_cols(test_cols, true);
printf("Inverse mix cols should be same as original\n");
print_block(test_cols);
printf("\n");

printf("----Testing add round key----\n");
printf("Original\n");
print_block(test_cols);
AES_add_round_key(test_cols, AES128_expanded_key, 0);
printf("Key added\n");
print_block(test_cols);
AES_add_round_key(test_cols, AES128_expanded_key, 0);
printf("Key added again should be same as original\n");
print_block(test_cols);
printf("\n");

printf("----Testing AES128----\n");
printf("Original\n");
print_block(test_cols);
AES_encrypt(AES128, test_cols, AES128_expanded_key);
printf("Encrypted\n");
print_block(test_cols);
AES_decrypt(AES128, test_cols, AES128_expanded_key);
printf("Decrypted should be same as before\n");
print_block(test_cols);
printf("\n");

printf("----Testing AES192----\n");
printf("Original\n");
print_block(test_cols);
AES_encrypt(AES192, test_cols, AES192_expanded_key);
printf("Encrypted\n");
print_block(test_cols);
AES_decrypt(AES192, test_cols, AES192_expanded_key);
printf("Decrypted should be same as before\n");
print_block(test_cols);
printf("\n");

printf("----Testing AES256----\n");
```

```
    printf("Original\n");
    print_block(test_cols);
    AES_encrypt(AES256, test_cols, AES256_expanded_key);
    printf("Encrypted\n");
    print_block(test_cols);
    AES_decrypt(AES256, test_cols, AES256_expanded_key);
    printf("Decrypted should be same as before\n");
    print_block(test_cols);

    // */ **** TESTING PURPOSES ****
}
```

2 AES.h

```
#ifndef EHN_PRAC2_AES_H
#define EHN_PRAC2_AES_H

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <stdbool.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/timeb.h>

/// The maximum length of an input to be handled.
#define MAX_REQ_LEN 104857600 // 100 MiB = 104857600 Bytes
/// Activate or deactivate verbose mode capabilities
#define VERBOSE 1

// Constants
#define AES128 0
#define AES192 1
#define AES256 2
#define AES128_ROUNDS 10
#define AES192_ROUNDS 12
#define AES256_ROUNDS 14
#define AES128_KEY_SIZE 176
#define AES192_KEY_SIZE 208
#define AES256_KEY_SIZE 240
#define AES128_USER_KEY_SIZE 16
#define AES192_USER_KEY_SIZE 24
#define AES256_USER_KEY_SIZE 32
#define AES128_EXPANSION 10
#define AES192_EXPANSION 8
#define AES256_EXPANSION 7
#define AES128_SUB_EXPANSION 3
#define AES192_SUB_EXPANSION 5
#define AES256_SUB_EXPANSION 7
#define CFB8 1
#define CFB64 8
#define CFB128 16 // Default
```

```
/// Provides a one-to-one mapping for the non-linear substitution of a byte.
const int S_BOX[2][16][16] = {{{0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5, 0x30, 0x01, 0x67, 0x2B, 0xFE,
↪  0xD7, 0xAB, 0x76}, // Forward
                               {0xCA, 0x82, 0xC9, 0x7D, 0xFA, 0x59, 0x47, 0xF0, 0xAD, 0xD4, 0xA2, 0xAF, 0x9C,
                               ↪  0xA4, 0x72, 0xC0},
                               {0xB7, 0xFD, 0x93, 0x26, 0x36, 0x3F, 0xF7, 0xCC, 0x34, 0xA5, 0xE5, 0xF1, 0x71,
                               ↪  0xD8, 0x31, 0x15},
                               {0x04, 0xC7, 0x23, 0xC3, 0x18, 0x96, 0x05, 0x9A, 0x07, 0x12, 0x80, 0xE2, 0xEB,
                               ↪  0x27, 0xB2, 0x75},
                               {0x09, 0x83, 0x2C, 0x1A, 0x1B, 0x6E, 0x5A, 0xA0, 0x52, 0x3B, 0xD6, 0xB3, 0x29,
                               ↪  0xE3, 0x2F, 0x84},
                               {0x53, 0xD1, 0x00, 0xED, 0x20, 0xFC, 0xB1, 0x5B, 0x6A, 0xCB, 0xBE, 0x39, 0x4A,
                               ↪  0x4C, 0x58, 0xCF},
                               {0xD0, 0xEF, 0xAA, 0xFB, 0x43, 0x4D, 0x33, 0x85, 0x45, 0xF9, 0x02, 0x7F, 0x50,
                               ↪  0x3C, 0x9F, 0xA8},
                               {0x51, 0xA3, 0x40, 0x8F, 0x92, 0x9D, 0x38, 0xF5, 0xBC, 0xB6, 0xDA, 0x21, 0x10,
                               ↪  0xFF, 0xF3, 0xD2},
                               {0xCD, 0x0C, 0x13, 0xEC, 0x5F, 0x97, 0x44, 0x17, 0xC4, 0xA7, 0x7E, 0x3D, 0x64,
                               ↪  0x5D, 0x19, 0x73},
                               {0x60, 0x81, 0x4F, 0xDC, 0x22, 0x2A, 0x90, 0x88, 0x46, 0xEE, 0xB8, 0x14, 0xDE,
                               ↪  0x5E, 0x0B, 0xDB},
                               {0xE0, 0x32, 0x3A, 0x0A, 0x49, 0x06, 0x24, 0x5C, 0xC2, 0xD3, 0xAC, 0x62, 0x91,
                               ↪  0x95, 0xE4, 0x79},
                               {0xE7, 0xC8, 0x37, 0x6D, 0x8D, 0xD5, 0x4E, 0xA9, 0x6C, 0x56, 0xF4, 0xEA, 0x65,
                               ↪  0x7A, 0xAE, 0x08},
                               {0xBA, 0x78, 0x25, 0x2E, 0x1C, 0xA6, 0xB4, 0xC6, 0xE8, 0xDD, 0x74, 0x1F, 0x4B,
                               ↪  0xBD, 0x8B, 0x8A},
                               {0x70, 0x3E, 0xB5, 0x66, 0x48, 0x03, 0xF6, 0x0E, 0x61, 0x35, 0x57, 0xB9, 0x86,
                               ↪  0xC1, 0x1D, 0x9E},
                               {0xE1, 0xF8, 0x98, 0x11, 0x69, 0xD9, 0x8E, 0x94, 0x9B, 0x1E, 0x87, 0xE9, 0xCE,
                               ↪  0x55, 0x28, 0xDF},
                               {0x8C, 0xA1, 0x89, 0x0D, 0xBF, 0xE6, 0x42, 0x68, 0x41, 0x99, 0x2D, 0x0F, 0xB0,
                               ↪  0x54, 0xBB, 0x16}},
                              {{0x52, 0x09, 0x6A, 0xD5, 0x30, 0x36, 0xA5, 0x38, 0xBF, 0x40, 0xA3, 0x9E, 0x81,
                              ↪  0xF3, 0xD7, 0xFB}, // Inverse
                               {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}};

/// Used for the transformation of a column in the mix columns operation.
const int PRIME_MATRIX[2][4][4] = {{{2, 3, 1, 1}, // Forward
                                    {1, 2, 3, 1},
                                    {1, 1, 2, 3},
                                    {3, 1, 1, 2}},
                                   {{14, 11, 13,  9}, // Inverse
                                    { 9, 14, 11, 13},
                                    {13,  9, 14, 11},
                                    {11, 13,  9, 14}}};


/**
 * The main function. Arguments as described in the README is passed to this function.
 * This function then uses the arguments to either encrypt or decrypt some input.
 * @param argc The number of arguments passed.
 * @param argv The arguments as C-strings.
 * @return Successful execution.
 */
int main(int argc, char *argv[]);


/**
 * Convert a char array to 4x4 block of hex.
 * @param message A c-string containing the message to be converted.
 * @param current_block The output as a 4x4 integer array.
 * @param start_pos The position from which to start the conversion in the string.
 */
void char_blockify(unsigned char message[], int current_block[4][4], int start_pos);


/**
 * Convert an integer array to 4x4 block of hex.
 * @param message An integer array containing the values to be converted.
 * @param current_block The output as a 4x4 integer array.
 */
void int_blockify(int message[16], int current_block[4][4]);


/**
 * Output a word to the terminal.
 * @param word The word to be printed.
```

```
 * @param length The length of the word.
 */
void print_word(int word[], int length);


/**
 * Output a 4x4 block to the terminal as a block of hex.
 * @param current_block The block to be printed.
 */
void print_block(int current_block[4][4]);


/**
 * Output the expanded key in rows of 16.
 * @param width Use the macros AES128, AES192 or AES256 to select which width to use.
 * @param expanded_key The expanded key to print.
 */
void print_expanded_key(int width, int expanded_key[]);


// Print a c-string up to a certain length in hex
void print_hex_string(unsigned char hex_string[], int message_len);


 // Write a message to a file
void write_to_file(char filename[], unsigned char message[], int message_len);


// Create the output directory and return the full file path
char *create_path(int method, char *file_name);

/**
 * Convert block back to c-string.
 * @param message The output array, must exist before being passed in.
 * @param current_block The block to be converted.
 * @param start_pos The position to start converting in the output.
 */
void char_unblockify(unsigned char message[], int current_block[4][4], int start_pos);


/**
 * Shift last items in an array to the front or vice-versa.
 * @param word The array to be rotated, also the output.
 * @param length The length of the word.
 * @param rotations The number of rotations to perform.
 * @param inverse Rotate in the opposite direction if true.
 */
void AES_word_rotate(int word[], int length, int rotations, bool inverse);


/**
 * Divide value up into its MSB and LSB Nibble and return the s_box value.
 * @param input The value to be transformed.
 * @param inverse Perform the inverse transform if true.
```

```
 * @return The transformed value.
 */
int AES_s_box_transform(int input, bool inverse);


/**
 * Core key operation, transform of previous 4 bytes.
 * @param word The bytes to be transformed, also the output.
 * @param rcon The round constant to be used.
 */
void AES_key_scheduler(int word[4], int rcon);


/**
 * Exponentiation of 2, double the previous value except when 0x80 and max value of 0xFF.
 * @param previous The value to be used exponentiated.
 * @return The exponentiated value.
 */
int AES_exp_2(int previous);


/**
 * Main key expansion function.
 * @param width Use the macros AES128, AES192 or AES256 to select which width to use.
 * @param expanded_key The expanded key output, the correct length array (AESxxx_KEY_SIZE + 32) must exist and
↪  be passed in here.
 * Allocate more space since AES_key_expansion deliberately writes out of bounds.
 * @param user_key The user key to be expanded.
 */
void AES_key_expansion(int width, int expanded_key[], int user_key[]);


/**
 * Substitute a block through the S-transform.
 * @param current_block The block to be transformed, also the output.
 * @param inverse Perform the inverse transform if true.
 */
void AES_sub_bytes(int current_block[4][4], bool inverse);


/**
 * The AES row shifting function.
 * @param current_block The block to be shifted, also the output.
 * @param inverse Perform the inverse shift if true.
 */
void AES_shift_rows(int current_block[4][4], bool inverse);


/**
 * Finite field multiplication.
 * @param a The first value.
 * @param b The second value.
 * @return The result of the dot product.
 */
```

```
int AES_dot_product(int a, int b);


/**
 * Perform the dot product of the block and the prime matrix.
 * @param current_block The block to be used in the dot product, also the output.
 * @param inverse Perform the inverse dot product if true.
 */
void AES_mix_cols(int current_block[4][4], bool inverse);


/**
 * XOR a block with the expanded key at a certain index
 * @param current_block The block to which the round key should be added, also the output.
 * @param expanded_key The expanded key to use.
 * @param key_index The index in the key to start from.
 */
void AES_add_round_key(int current_block[4][4], int expanded_key[], int key_index);


/**
 * The AES encryption algorithm.
 * @param width Use the macros AES128, AES192 or AES256 to select which width to use.
 * @param current_block The block to be encrypted, also the output.
 * @param expanded_key The expanded key to be used.
 * @return Successful execution.
 */
bool AES_encrypt(int width, int current_block[4][4], int expanded_key[]);


/**
 * The AES decryption algorithm.
 * @param width Use the macros AES128, AES192 or AES256 to select which width to use.
 * @param current_block The block to be decrypted, also the output.
 * @param expanded_key The expanded key to be used.
 * @return Successful execution.
 */
bool AES_decrypt(int width, int current_block[4][4], int expanded_key[]);


/**
 * The Cipher Block Chaining encryption algorithm.
 * @param width Use the macros AES128, AES192 or AES256 to select which width to use.
 * @param message The message to be encrypted, also the output.
 * @param message_len The length of the message.
 * @param IV The initialization vector to be used.
 * @param user_key The user key to be used.
 * @return Successful execution.
 */
bool CBC_encrypt(int width, unsigned char message[], int message_len, int IV[16], int user_key[]);


/**
 * The Cipher Block Chaining decryption algorithm.
```

```
 * @param width Use the macros AES128, AES192 or AES256 to select which width to use.
 * @param message The message to be decrypted, also the output.
 * @param message_len The length of the message.
 * @param IV The initialization vector to be used.
 * @param user_key The user key to be used.
 * @return Successful execution.
 */
bool CBC_decrypt(int width, unsigned char message[], int message_len, int IV[16], int user_key[]);


/**
 * The Cipher Feedback encryption algorithm.
 * @param width Use the macros AES128, AES192 or AES256 to select which width to use.
 * @param message The stream to be encrypted, also the output.
 * @param message_len The length of the message.
 * @param CFB_len The length of the chain to use.
 * @param IV The initialization vector to be used.
 * @param user_key The user key to be used.
 * @return Successful execution.
 */
bool CFB_encrypt(int width, unsigned char message[], int message_len, int CFB_len, int IV[16], int user_key[]);


/**
 * The Cipher Feedback decryption algorithm.
 * @param width Use the macros AES128, AES192 or AES256 to select which width to use.
 * @param message The stream to be decrypted, also the output.
 * @param message_len The length of the message.
 * @param CFB_len The length of the chain to use.
 * @param IV The initialization vector to be used.
 * @param user_key The user key to be used.
 * @return Successful execution.
 */
bool CFB_decrypt(int width, unsigned char message[], int message_len, int CF_Blen, int IV[16], int user_key[]);


#if VERBOSE
/**
 * The verbose version of the AES encryption algorithm. Prints out intermediate results in the encryption
 * process.
 * @param width Use the macros AES128, AES192 or AES256 to select which width to use.
 * @param current_block The block to be encrypted, also the output.
 * @param expanded_key The expanded key to be used.
 * @return Successful execution.
 */
bool AES_encrypt_verbose(int width, int current_block[4][4], int expanded_key[]);

/**
 * The verbose version of the AES decryption algorithm. Prints out intermediate results in the decryption
 * process.
 * @param width Use the macros AES128, AES192 or AES256 to select which width to use.
 * @param current_block The block to be decrypted, also the output.
 * @param expanded_key The expanded key to be used.
 * @return Successful execution.
```

```
 */
bool AES_decrypt_verbose(int width, int current_block[4][4], int expanded_key[]);

/**
 * The verbose version of the Cipher Block Chaining encryption algorithm. Prints out intermediate results in
 * the encryption process.
 * @param width Use the macros AES128, AES192 or AES256 to select which width to use.
 * @param message The message to be encrypted, also the output.
 * @param message_len The length of the message.
 * @param IV The initialization vector to be used.
 * @param user_key The user key to be used.
 * @return Successful execution.
 */
bool CBC_encrypt_verbose(int width, unsigned char message[], int message_len, int IV[16], int user_key[]);

/**
 * The verbose version of the Cipher Block Chaining decryption algorithm. Prints out intermediate results
 * in the decryption process.
 * @param width Use the macros AES128, AES192 or AES256 to select which width to use.
 * @param message The message to be decrypted, also the output.
 * @param message_len The length of the message.
 * @param IV The initialization vector to be used.
 * @param user_key The user key to be used.
 * @return Successful execution.
 */
bool CBC_decrypt_verbose(int width, unsigned char message[], int message_len, int IV[16], int user_key[]);

/**
 * The verbose version of the Cipher Feedback encryption algorithm. Prints out intermediate results in the
↪  encryption
 * process.
 * @param width Use the macros AES128, AES192 or AES256 to select which width to use.
 * @param message The stream to be encrypted, also the output.
 * @param message_len The length of the message.
 * @param CFB_len The length of the chain to use.
 * @param IV The initialization vector to be used.
 * @param user_key The user key to be used.
 * @return Successful execution.
 */
bool CFB_encrypt_verbose(int width, unsigned char message[], int message_len, int CFB_len, int IV[16], int
↪  user_key[]);

/**
 * The verbose version of the Cipher Feedback decryption algorithm. Prints out intermediate results in the
↪  decryption
 * process.
 * @param width Use the macros AES128, AES192 or AES256 to select which width to use.
 * @param message The stream to be decrypted, also the output.
 * @param message_len The length of the message.
 * @param CFB_len The length of the chain to use.
 * @param IV The initialization vector to be used.
 * @param user_key The user key to be used.
 * @return Successful execution.
 */
```

```
bool CFB_decrypt_verbose(int width, unsigned char message[], int message_len, int CF_Blen, int IV[16], int
↪  user_key[]);
#endif


// Convert hex to int, done because the system hex converter is unreliable
int hex_convert(char hex_string[], int length);


// Print out various tests to test the functionality of the other functions
void test_functionality( );


#endif //EHN_PRAC2_AES_H
```