Substitution Permutation Networks

Student: 470277

3rd October 2015

Abstract

A report that explores the definition of substitution permutation networks and determines the main aspects that make them useful in the creation of highly secure block ciphers: separation of confusion and diffusion and the avalanche effect. In the end, an example encipherment and decipherment with a two round AES program using such a network is shown. Finally it is concluded that SPNs are a useful base to build cryptosystems upon.

1 Introduction

The substitution-permutation network (SPN) is a cryptosystem design used in block ciphers such as the AES (Rijndael). Here we seek to explore what makes such a design useful in securing block ciphers such that it warrants their use. We omit any discussion of *modes of operation* necessary to secure multiblock encipherment and focus on single block encipherment.

We begin with the definition of SPNs, then explore their characteristics and give an example SPN which is AES where we see two round AES in action. The source for the two round AES is appended to the end of the report.

2 Definition of SPNs

An SPN is a block cipher design which is a type of a product cipher which is a cipher combining "two or more transformations in a manner intending that the resulting cipher is more secure than the individual components"[1]. Specifically the SPN is composed of number of stages or rounds each involving a layer of non-linear functions and a linear layer[2]. Before and after each round, round keys which are derived from a master key are XORed with the intermediate results[3]. The two layers are usually fixed and iterated for some number of rounds[3]. The decipherment is simply done in reverse meaning that the two layers need to also be invertible[3]. Figure 1 shows a simple example of a SPN with 16 bit input and output using S-Boxes as the non-linear functions, substituting 4 bits each, and a P-Box for the linear layer which permutes the resulting bits which are then XORed with a round key. Such boxes are a typical implementation of the layers.

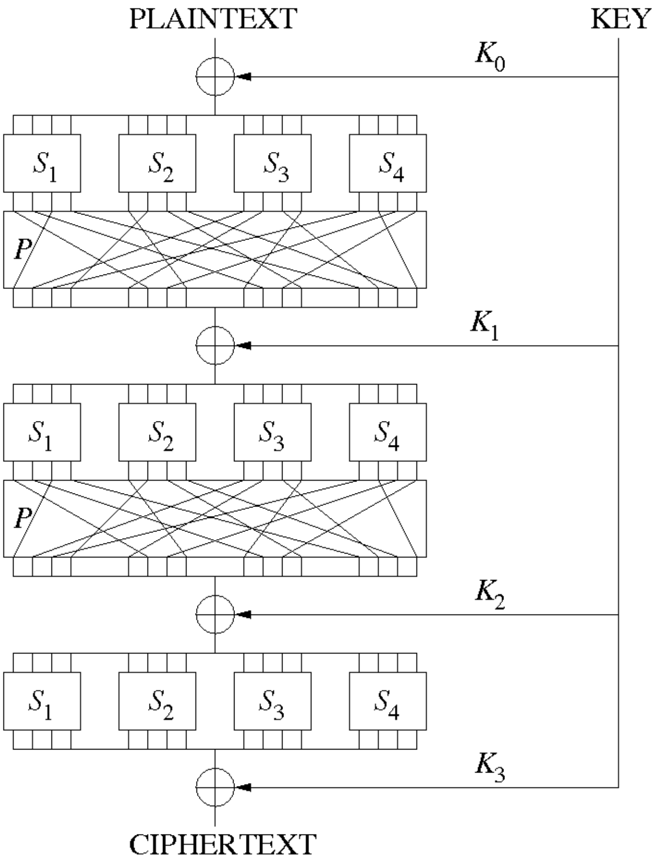

Figure 1: By GaborPete (Own work) [CC BY-SA 3.0 (http://creativecommons.org/licenses/by-sa/3.0) or GFDL (http://www.gnu.org/copyleft/fdl.html)], via Wikimedia Commons

3 The usefulness of SPNs

What makes the SPN useful in the design of highly secure block ciphers is that it makes it easier to design them as *pseudorandom permutations*[3], i.e. make their ciphertext be indistinguishable from a random permutation on the plaintext, which makes statistical analysis of the cryptogram to recover the key or plaintext very hard, even when the attack is a chosen-ciphertext attack and the attacker knows everything about the cipher except the key[3], leaving few options but to attain the key by brute force which has computational limitations for a sufficiently large key.

How the SPN makes easier to attain the pseudorandomness objective is a separation of concerns, specifically of *confusion* (attained via the non-linear layer) and *diffusion* (attained via the linear layer)[3][2]. This separation allows us to attain optimal confusion and diffusion[2]. Together, confusion and diffusion can be made to cause the essential *avalanche effect*[3].

3.1 Confusion and diffusion

Shannon introduced these terms in connection to statistical analyses on the ciphertext which can be performed on many kinds of ciphers. An intuitive example is the breaking of the simple substitution cipher where letter frequencies are observed to recover the plaintext and consequently the key. This is possible due to the redundancy of the plaintext language. Diffusion counters this by "spreading out" the redundancy in the ciphertext such that the attacker needs to accept an impractical amount of ciphertext to pinpoint the statistical structure[4]. Figure 1 shows how the P-Box distributes the bits output from one S-Box among the other S-Boxes to that end.

Confusion seeks to counter the statistical analysis of the ciphertext with respect to finding the key by making the relation between the ciphertext and the key very complex[4], i.e. by making the resulting system highly non-linear so it's difficult to solve via linear analysis[5]. The details of how that is done is beyond the scope of this text.

3.2 The avalanche effect

The avalanche effect is used over the case when a change to one bit of the plaintext affects every bit of the resulting ciphertext (changing half of the bits on average)[3]. This can be achieved when the non-linear layer propagates the effects of changing a single input bit, i.e. for 1 changed input bit, at least 2 bits change in the output[3]. If this happens in one of the S-Boxes as in figure 1, the linear layer in turn receives the two changed bits and makes sure to distribute them to two different S-Boxes in later rounds that produce 4 changed bits and so on, until the number of bits changed equal the block size[3]. This characteristic also holds in decipherment, making the SPN not so easily malleable. Most changes to the ciphertext would result in nonsensical plaintext.

4 AES implementation of the SPN

Rijndael was accepted as the Advanced Encryption Standard (AES) in 2001. It is a example of a highly secure implementation of the SPN. It works on 4x4 byte matrices so the plaintext bytes are first placed into such a matrix. AES uses 4 layers:

- ByteSub Transformation: Serves as the non-linear layer. It uses the same S-Box for every byte of the input. It is carefully chosen to be highly non-linear, thus warding against linear and differential cryptanalysis attacks as well as interpolation attacks[6].
- ShiftRow Transformation: Part of the linear layer. It shifts the rows of the input matrix to the left. How far to the left depends on the row. It was added to resist "truncated differentials and the Square attack"[6].
- MixColumn Transformation: Part of the linear layer. It multiplies the input matrix by a fixed matrix. It causes diffusion among the bytes, such that if one byte changes in the column of the input matrix, all the bytes in the corresponding column in the output matrix change[6]. If two bytes change in the input matrix column, then at least 3 bytes change in the corresponding output matrix column[6].
- AddRoundKey: XORs the input matrix with a given round key.

In encipherment, the plaintext goes once through AddRoundKey and then through 10 rounds of all of the above layers in order, except with the last round omitting the MixColumn layer[6]. The round keys recursively from the columns of the master key, with the first round key being the master key itself. The key generation involves nonlinear mixing of columns via the same S-Box as the ByteSub layer uses[6]. The mixing is designed to

- resist attacks where the cryptanalyst knows a part of the key and tries to deduce the remaining bits[6], and
- to ensure that two distinct keys do not have a large number of keys in common[6].

We see that the AES fulfills the design requirement of an SPN, but it also contains various other design considerations to counter various types of cryptanalytic attacks.

4.1 Two round AES example

Here is an example single block enciphering CLI C program implementation of a two round 128 bit AES whose source files are at the end of this report. main.c is a minimal file for the interface which uses the high level functions encipher_block and decipher_block from the bottom of aes.h which in turn use lower level functions higher up. The S-Box and the inverse S-Box at the top of aes.h were taken from Wikipedia[7]. Additionally there is unittest.c which holds unit tests for some critical functions to make sure they work as expected. Here is the output of the unittest:

```
$ ./bin/unittest
[==========] Running 7 test(s).
[ RUN      ] test_xor_columns
[       OK ] test_xor_columns
```

```
[ RUN      ] test_byte_sub_transformation
[       OK ] test_byte_sub_transformation
[ RUN      ] test_shift_row_transformation
[       OK ] test_shift_row_transformation
[ RUN      ] test_mix_column_transformation
[       OK ] test_mix_column_transformation
[ RUN      ] test_round_key_addition
[       OK ] test_round_key_addition
[ RUN      ] test_generate_round_keys
[       OK ] test_generate_round_keys
[ RUN      ] test_encipher_decipher_block
[       OK ] test_encipher_decipher_block
[==========] 7 test(s) run.
[  PASSED  ] 7 test(s).
```

Here is an example encipherment with the program with the key as "abcdefghijklmnop" and the plaintext block as "I attack at dawn", with each of the 16 characters representing a byte. The ciphertext is only shown as hexadecimal because not all characters of 8 bits are easily representable. The program writes the ciphertext into a temporary enciphered.txt file:

```
$ ./bin/main -e -k abcdefghijklmnop "I attack at dawn"
Input(ascii):          I attack at dawn
Input(hexadecimal):    492061747461636b206174206461776e
Output(hexadecimal):   c39156250ec1bf8343432bf1f1ef1294
```

When deciphering, the program reads the cipher from enciphered.txt:

```
$ ./bin/main -d -k abcdefghijklmnop
Input(hexadecimal):    c39156250ec1bf8343432bf1f1ef1294
Output(hexadecimal):   492061747461636b206174206461776e
Output(ascii):         I attack at dawn
```

If we change a single bit of the plaintext input, making I into H (the characters differ by 1 bit in ascii), we get:

```
$ ./bin/main -e -k abcdefghijklmnop "H attack at dawn"
Input(ascii):          H attack at dawn
Input(hexadecimal):    482061747461636b206174206461776e
Output(hexadecimal):   899156250ec1bfd7434333f1f1871294
```

We can see that there are many similarities between the resulting ciphertexts:

```
I attack at dawn: c39156250ec1bf8343432bf1f1ef1294
H attack at dawn: 899156250ec1bfd7434333f1f1871294
```

This can be attributed to the small number of rounds not being sufficient to create the avalanche effect. It is also worsened by the fact that the algorithm uses the 1st and 10th round of AES. The 10th round lacks the MixColumn step to further diffuse the changed input.

5 Conclusion

Rijndael has stood the test of time as the AES for 14 years and remains unbroken in the practical sense. It thus seems reasonable to conclude that SPNs are indeed useful in the implementation of highly secure block ciphers. The security level however depends how the implementation is carried out, as can easily be deduced from observing the two round AES. So it can be said that SPNs are a good starting point in block cipher design where the necessary confusion, diffusion and the avalanche effect are aspects that are easier than otherwise to work with then on.

References

[1] Menezes AJ, Oorschot PC, Vanstone SA. Chapter 7 - Block Ciphers. In: Handbook of Applies Cryptography. United States of America: CRC Press; 1996. p. 223.

[2] Petrovic S. Session 3 - Symmetric Ciphers 2 [Lecture slides]. University of Gjøvik; 2015.

[3] Katz J, Lindell Y. Chapter 5 - Practical Construction of Pesudorandom Permutations (Block Ciphers). In: Introduction to Modern Cryptography. United States of America: Chapman & Hall/CRC; 2008. p. 159.

[4] Shannon CE. Communication Theory of Secrecy Systems. 1949;28(4):656–715. The online retyped version was used. Available from: `http://netlab.cs.ucla.edu/wiki/files/shannon1949.pdf`.

[5] Biham E. On Matsui's linear cryptanalysis. In: De Santis A, editor. Advances in Cryptology — EUROCRYPT'94. vol. 950 of Lecture Notes in Computer Science. Springer Berlin Heidelberg; 1995. p. 341–355. Available from: `http://dx.doi.org/10.1007/BFb0053449`.

[6] Trappe W, Washington LC. Chapter 5 - The Advanced Encryption Standard: Rijndael. In: Introduction to Cryptography with Coding Theory. United States of America: Pearson Prentice Hall; 2006. p. 151.

[7] Rijndael S-Box, Wikipedia, the free encyclopedia; [updated 13.08.2015; cited 02.10.2015]. Available from: `https://en.wikipedia.org/wiki/Rijndael_S-box`.

main.c

```c
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include "aes.h"

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

    unsigned char key[16], block[16], direction;
    int c;

    while ((c = getopt(argc, argv, "edk:")) != -1) {
        if (c == 'e' || c == 'd') {
            direction = c;
        }
        else if (c == 'k') {
            for (int i = 0; i < 16; i++) {
                key[i] = *optarg;
                optarg++;
            }
        }
    }

    // Encipher and place result into enciphered.txt
    if (direction == 'e') {
        FILE* f = fopen("enciphered.txt", "w");
        printf("Input(ascii):        ");
        for (int i = 0; i < 16; i++) {
            block[i] = *argv[optind];
            argv[optind]++;
            printf("%c", block[i]);
        }
        printf("\nInput(hexadecimal):  ");
        for (int i = 0; i < 16; i++) {
            printf("%02x", block[i]);
        }
        encipher_block(block, key);
        printf("\nOutput(hexadecimal): ");
        for (int i = 0; i < 16; i++) {
            fprintf(f, "%c", block[i]);
            printf("%02x", block[i]);
        }
        printf("\n");
    }
    // Decipher contents of enciphered.txt and display
    else if (direction == 'd') {
        FILE* f = fopen("enciphered.txt", "r");
        printf("Input(hexadecimal):  ");
        for (int i = 0; i < 16; i++) {
            block[i] = fgetc(f);
            printf("%02x", block[i]);
        }
        decipher_block(block, key);
        printf("\nOutput(hexadecimal): ");
        for (int i = 0; i < 16; i++) {
```

```
            printf("%02x", block[i]);
        }
        printf("\nOutput(ascii):       ");
        for (int i = 0; i < 16; i++) {
            printf("%c", block[i]);
        }
        printf("\n");
    }

    return 0;
}
```

aes.h

```c
static const unsigned char s_box[256] =
{
    0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5, 0x30, 0x01, 0x67, 0x2B, 0xFE, 0xD7, 0xAB, 0x76,
    0xCA, 0x82, 0xC9, 0x7D, 0xFA, 0x59, 0x47, 0xF0, 0xAD, 0xD4, 0xA2, 0xAF, 0x9C, 0xA4, 0x72, 0xC0,
    0xB7, 0xFD, 0x93, 0x26, 0x36, 0x3F, 0xF7, 0xCC, 0x34, 0xA5, 0xE5, 0xF1, 0x71, 0xD8, 0x31, 0x15,
    0x04, 0xC7, 0x23, 0xC3, 0x18, 0x96, 0x05, 0x9A, 0x07, 0x12, 0x80, 0xE2, 0xEB, 0x27, 0xB2, 0x75,
    0x09, 0x83, 0x2C, 0x1A, 0x1B, 0x6E, 0x5A, 0xA0, 0x52, 0x3B, 0xD6, 0xB3, 0x29, 0xE3, 0x2F, 0x84,
    0x53, 0xD1, 0x00, 0xED, 0x20, 0xFC, 0xB1, 0x5B, 0x6A, 0xCB, 0xBE, 0x39, 0x4A, 0x4C, 0x58, 0xCF,
    0xD0, 0xEF, 0xAA, 0xFB, 0x43, 0x4D, 0x33, 0x85, 0x45, 0xF9, 0x02, 0x7F, 0x50, 0x3C, 0x9F, 0xA8,
    0x51, 0xA3, 0x40, 0x8F, 0x92, 0x9D, 0x38, 0xF5, 0xBC, 0xB6, 0xDA, 0x21, 0x10, 0xFF, 0xF3, 0xD2,
    0xCD, 0x0C, 0x13, 0xEC, 0x5F, 0x97, 0x44, 0x17, 0xC4, 0xA7, 0x7E, 0x3D, 0x64, 0x5D, 0x19, 0x73,
    0x60, 0x81, 0x4F, 0xDC, 0x22, 0x2A, 0x90, 0x88, 0x46, 0xEE, 0xB8, 0x14, 0xDE, 0x5E, 0x0B, 0xDB,
    0xE0, 0x32, 0x3A, 0x0A, 0x49, 0x06, 0x24, 0x5C, 0xC2, 0xD3, 0xAC, 0x62, 0x91, 0x95, 0xE4, 0x79,
    0xE7, 0xC8, 0x37, 0x6D, 0x8D, 0xD5, 0x4E, 0xA9, 0x6C, 0x56, 0xF4, 0xEA, 0x65, 0x7A, 0xAE, 0x08,
    0xBA, 0x78, 0x25, 0x2E, 0x1C, 0xA6, 0xB4, 0xC6, 0xE8, 0xDD, 0x74, 0x1F, 0x4B, 0xBD, 0x8B, 0x8A,
    0x70, 0x3E, 0xB5, 0x66, 0x48, 0x03, 0xF6, 0x0E, 0x61, 0x35, 0x57, 0xB9, 0x86, 0xC1, 0x1D, 0x9E,
    0xE1, 0xF8, 0x98, 0x11, 0x69, 0xD9, 0x8E, 0x94, 0x9B, 0x1E, 0x87, 0xE9, 0xCE, 0x55, 0x28, 0xDF,
    0x8C, 0xA1, 0x89, 0x0D, 0xBF, 0xE6, 0x42, 0x68, 0x41, 0x99, 0x2D, 0x0F, 0xB0, 0x54, 0xBB, 0x16
};

static const unsigned char s_box_inverse[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
};

static unsigned char mix_col_matrix[4][4] =
{
    { 0x02, 0x03, 0x01, 0x01 },
    { 0x01, 0x02, 0x03, 0x01 },
    { 0x01, 0x01, 0x02, 0x03 },
    { 0x03, 0x01, 0x01, 0x02 }
};

static unsigned char inverse_mix_col_matrix[4][4] =
{
    { 0x0E, 0x0B, 0x0D, 0x09 },
    { 0x09, 0x0E, 0x0B, 0x0D },
    { 0x0D, 0x09, 0x0E, 0x0B },
    { 0x0B, 0x0D, 0x09, 0x0E }
```

```c
};

// Partial polynomial, omitting the leftmost 9th bit which doesn't have to be used here.
static const unsigned char rijndael_polynomial = 0x1B;

// Precondition: positions is positive.
// Postcondition: characters in row have cyclically shifted positions times to the left.
static void shift_left(unsigned char row[4], int positions) {

    char auxiliary_row[4];
    for (int i = 0; i < 4; i++) {
        auxiliary_row[i] = row[i];
    }

    for (int i = 0; i < 4; i++) {
        row[i] = auxiliary_row[(i + positions) % 4];
    }
}

// Precondition: positions is positive.
// Postcondition: characters in row have cyclically shifted positions times to the right.
static void shift_right(unsigned char row[4], int positions) {

    char auxiliary_row[4];
    for (int i = 0; i < 4; i++) {
        auxiliary_row[i] = row[i];
    }

    for (int i = 0; i < 4; i++) {
        row[i] = auxiliary_row[(i - positions + 4) % 4]; // Add 4 to match mathematical modulus
    }
}

// Precondition: None
// Postcondition: output = bX^n in GF(2^8) modulo the Rijndael polynomial.
static unsigned char multiply_by_term(unsigned char b, int n) {
    for (int i = 0; i < n; i++) {
        b <<= 1;
        if (b & 0x01) {                  // is rightmost bit 1?
            b &= 0xFE;                   // remove rightmost bit
            b ^= rijndael_polynomial;    // calculate as modulo the polynomial
        }
    }
    return b;
}

// Precondition: None
// Postcondition: output = ba in GF(2^8) modulo the Rijndael polynomial.
static unsigned char gf_multiply(unsigned char a, unsigned char b) {

    unsigned char result = 0x00;
    for (int i = 0; i < 8; i++) {
        if (a & (0x1 << i)) {                    // does a have term X^i?
            result ^= multiply_by_term(b, i);    // sum the results of bX^i
        }
    }
```

```c
    return result;
}

// Precondition: None
// Postcondition: byte_matrix = multiplier × byte_matrix in GF(2^8) modulo Rijndael polynomial.
static void matrix_multiply(unsigned char multiplier[4][4], unsigned char byte_matrix[4][4]) {
    unsigned char new_matrix[4][4];
    unsigned char x = 0x00;
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++) {
            for (int m = 0; m < 4; m++) {
                x ^= gf_multiply(multiplier[i][m], byte_matrix[m][j]);
            }
            new_matrix[i][j] = x;
            x = 0x00;
        }
    }
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++) {
            byte_matrix[i][j] = new_matrix[i][j];
        }
    }
}

// Precondition: None
// Postcondition: byte_array bytes have been substituted via s_box.
static void substitute_bytes(unsigned char byte_array[4]) {
    for (int i = 0; i < 4; i++) {
        byte_array[i] = s_box[byte_array[i]];
    }
}

// Precondition: None
// Postcondition: byte_array bytes have been substituted via inverse s_box.
static void inverse_substitute_bytes(unsigned char byte_array[4]) {
    for (int i = 0; i < 4; i++) {
        byte_array[i] = s_box_inverse[byte_array[i]];
    }
}

// Precondition: None
// Postcondition: matrix has been filled with bytes from block in order, top row to bottom row,
//                filling each row from left to right.
static void block_to_matrix(unsigned char block[16], unsigned char matrix[4][4]) {
    int row = 0;
    for (int i = 0; i < 16; i++) {
        matrix[row][i % 4] = block[i];
        if ((i + 1) % 4 == 0) row++;
    }
}

// Precondition: None
// Postcondition: Inverse block_to_matrix.
static void matrix_to_block(unsigned char matrix[4][4], unsigned char block[16]) {
    int row = 0;
    for (int i = 0; i < 16; i++) {
```

```c
        block[i] = matrix[row][i % 4];
        if ((i + 1) % 4 == 0) row++;
    }
}

// ---------------- The 4 stages of a Rijndael round and inverses -----------------

// Precondition: None
// Postcondition: Each byte value B[i][j] of byte_matrix has been replaced with s_box[B[i][j]].
static void byte_sub_transformation(unsigned char byte_matrix[4][4]) {
    for (int i = 0; i < 4; i++) {
        substitute_bytes(byte_matrix[i]);
    }
}

// Precondition: None
// Postcondition: Each byte value B[i][j] of byte_matrix has been replaced with s_box_inverse[B[i][j]].
static void inverse_byte_sub_transformation(unsigned char byte_matrix[4][4]) {
    for (int i = 0; i < 4; i++) {
        inverse_substitute_bytes(byte_matrix[i]);
    }
}

// Precondition: None
// Postcondition: Each row B[i] of byte_matrix has been cyclically shifted i positions to the left.
static void shift_row_transformation(unsigned char byte_matrix[4][4]) {
    for (int i = 0; i < 4; i++) {
        shift_left(byte_matrix[i], i);
    }
}

// Precondition: None
// Postcondition: Each row B[i] of byte_matrix has been cyclically shifted i positions to the right.
static void inverse_shift_row_transformation(unsigned char byte_matrix[4][4]) {
    for (int i = 0; i < 4; i++) {
        shift_right(byte_matrix[i], i);
    }
}

// Precondition: None
// Postcondition: byte_matrix = mix_col_matrix × byte_matrix in GF(2^8) modulo Rijndael polynomial.
static void mix_column_transformation(unsigned char byte_matrix[4][4]) {
    matrix_multiply(mix_col_matrix, byte_matrix);
}

// Precondition: None
// Postcondition: byte_matrix = inverse_mix_col_matrix × byte_matrix in GF(2^8) modulo Rijndael polynomial.
static void inverse_mix_column_transformation(unsigned char byte_matrix[4][4]) {
    matrix_multiply(inverse_mix_col_matrix, byte_matrix);
}

// Precondition: None
// Postcondition: byte_matrix = byte_matrix ⊕ round_key.
static void round_key_addition(unsigned char byte_matrix[4][4], unsigned char round_key[4][4]) {
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++) {
```

```
            byte_matrix[j][i] ^= round_key[i][j]; // round_keys are ordered by column
        }
    }
}


//------------------------- Round key generation -------------------------

// Precondition: None
// Postcondition: output = a ⊕ b
static void xor_columns(unsigned char a[4], unsigned char b[4], unsigned char output[4]) {
    for (int i = 0; i < 4; i++) {
        output[i] = a[i] ^ b[i];
    }
}

// Precondition: column_number is a multiple of 4
// Postcondition: output = T(a), where T is the Rijndael column function in the round key generation.
static void transform(unsigned char a[4], unsigned char output[4], int column_number) {
    for (int i = 0; i < 4; i++) {
        output[i] = a[i];
    }
    shift_left(output, 1);
    substitute_bytes(output);
    output[0] ^= 0x01 << column_number / 4;
}

// Precondition: None
// Postcondition: round_keys holds the first 3 round keys as matrices corresponding to key.
static void generate_round_keys(unsigned char round_keys[3][4][4], unsigned char key[16]) {

    unsigned char expanded_round_keys[12][4]; // Treated as 12 columns of 4
    unsigned char auxiliary_column[4];

    // Generate first round_key by copying key into matrix
    block_to_matrix(key, expanded_round_keys);

    // Expand matrix to other columns
    for (int i = 4; i < 12; i++) {
        if (i % 4) { // i is not a multiple of 4?
            xor_columns(expanded_round_keys[i-4], expanded_round_keys[i-1], expanded_round_keys[i]);
        }
        else {
            transform(expanded_round_keys[i-1], auxiliary_column, i);
            xor_columns(expanded_round_keys[i-4], auxiliary_column, expanded_round_keys[i]);
        }
    }

    // Gather columns into the respective round keys
    for (int m = 0; m < 3; m++) {
        for (int i = 0; i < 4; i++) {
            int exp_col = i + m*4;
            for (int j = 0; j < 4; j++) {
                round_keys[m][i][j] = expanded_round_keys[exp_col][j];
            }
        }
```

```
    }
}



// ------------------------- Block level encipherment/decipherment -----------------------------

// Precondition: None
// Postcondition: block has been 2 round AES enciphered with key.
static unsigned char encipher_block(unsigned char block[16], unsigned char key[16]) {

    // Prepare block and round keys
    unsigned char round_keys[3][4][4], block_matrix[4][4];
    block_to_matrix(block, block_matrix);
    generate_round_keys(round_keys, key);

    // ARK
    round_key_addition(block_matrix, round_keys[0]);

    // 1st round: BS SR MC ARK
    byte_sub_transformation(block_matrix);
    shift_row_transformation(block_matrix);
    mix_column_transformation(block_matrix);
    round_key_addition(block_matrix, round_keys[1]);

    // 2nd round: BS SR ARK
    byte_sub_transformation(block_matrix);
    shift_row_transformation(block_matrix);
    round_key_addition(block_matrix, round_keys[2]);

    matrix_to_block(block_matrix, block);
}

// Precondition: None
// Postcondition: block has been 2 round AES deciphered with key.
static unsigned char decipher_block(unsigned char block[16], unsigned char key[16]) {

    // Prepare block and round keys
    unsigned char round_keys[3][4][4], block_matrix[4][4];
    block_to_matrix(block, block_matrix);
    generate_round_keys(round_keys, key);

    // 2nd round inversed: ARK ISR IBS
    round_key_addition(block_matrix, round_keys[2]);
    inverse_shift_row_transformation(block_matrix);
    inverse_byte_sub_transformation(block_matrix);

    // 1st round inversed: ARK IMC ISR IBS
    round_key_addition(block_matrix, round_keys[1]);
    inverse_mix_column_transformation(block_matrix);
    inverse_shift_row_transformation(block_matrix);
    inverse_byte_sub_transformation(block_matrix);

    // ARK
    round_key_addition(block_matrix, round_keys[0]);

```

```
    matrix_to_block(block_matrix, block);
}
```

unittest.c

```
#include <stdio.h>
#include <stdarg.h>
#include <stddef.h>
#include <setjmp.h>
#include <cmocka.h>
#include "aes.h"

static void test_xor_columns(void **state) {
    unsigned char a[4] = { 0x11, 0x01, 0x10, 0x00 }, b[4] = { 0x01, 0x00, 0x11, 0x00 }, output[4];
    char old = output[0];
    xor_columns(a, b, output);
    assert_true(output[0] == 0x10);
    assert_true(output[1] == 0x01);
    assert_true(output[2] == 0x01);
    assert_true(output[3] == 0x00);
}

static void test_gf_multiply(void **state) {
    unsigned char a = 0x02, b = 0xCB, expected_result = 0x8B, result;
    result = gf_multiply(a, b);
    assert_true(result == expected_result);

    a = 0x03;
    b = 0xFF;
    expected_result = 0x1B;
    result = gf_multiply(a, b);
    assert_true(result == expected_result);

    a = 0x01;
    b = 0xAD;
    expected_result = b;
    result = gf_multiply(a, b);
    assert_true(result == expected_result);

    a = 0x01;
    b = 0x60;
    expected_result = b;
    result = gf_multiply(a, b);
    assert_true(result == expected_result);

    a = 0x03;
    b = 0x09;
    expected_result = 0x7B;
    result = gf_multiply(a, b);
    assert_true(result == expected_result);

}

static void test_byte_sub_transformation(void **state) {
    unsigned char byte_matrix[4][4] =
    {
        { 0x00, 0x00, 0x00, 0x00 },
        { 0x00, 0x00, 0x84, 0x00 },
        { 0x00, 0x00, 0x00, 0x00 },
```

```c
        { 0x00, 0x00, 0x00, 0x00 },
    };
    byte_sub_transformation(byte_matrix);
    assert_true(byte_matrix[0][0] == 0x63);
    assert_true(byte_matrix[1][2] == 0x5F);
    inverse_byte_sub_transformation(byte_matrix);
    assert_true(byte_matrix[0][0] == 0x00);
    assert_true(byte_matrix[1][2] == 0x84);
}

static void test_shift_row_transformation(void **state) {
    unsigned char byte_matrix[4][4] =
    {
        { 0x00, 0x00, 0x00, 0xF0 },
        { 0x00, 0x00, 0x84, 0x00 },
        { 0x00, 0x09, 0x00, 0x00 },
        { 0x08, 0x00, 0x00, 0x00 },
    };
    shift_row_transformation(byte_matrix);
    assert_true(byte_matrix[0][3] == 0xF0);
    assert_true(byte_matrix[1][2] == 0x00);
    assert_true(byte_matrix[1][1] == 0x84);
    assert_true(byte_matrix[2][1] == 0x00);
    assert_true(byte_matrix[2][3] == 0x09);
    assert_true(byte_matrix[3][0] == 0x00);
    assert_true(byte_matrix[3][1] == 0x08);
    inverse_shift_row_transformation(byte_matrix);
    assert_true(byte_matrix[0][3] == 0xF0);
    assert_true(byte_matrix[1][2] == 0x84);
    assert_true(byte_matrix[1][1] == 0x00);
    assert_true(byte_matrix[2][1] == 0x09);
    assert_true(byte_matrix[2][3] == 0x00);
    assert_true(byte_matrix[3][0] == 0x08);
    assert_true(byte_matrix[3][1] == 0x00);
}

static void test_mix_column_transformation(void **state) {
    unsigned char byte_matrix[4][4] =
    {
        { 0x00, 0x00, 0x60, 0x00 },
        { 0x00, 0x00, 0x00, 0x00 },
        { 0x00, 0x00, 0x00, 0x00 },
        { 0x00, 0x00, 0x09, 0x00 },
    };
    mix_column_transformation(byte_matrix);
    assert_true(byte_matrix[2][2] == 0x7B);
    inverse_mix_column_transformation(byte_matrix);
    assert_true(byte_matrix[2][2] == 0x00);
}

static void test_round_key_addition(void **state) {
    unsigned char byte_matrix[4][4] =
    {
        { 0x00, 0x00, 0x00, 0xA4 },
        { 0x05, 0x00, 0x00, 0x00 },
        { 0x00, 0x00, 0xE0, 0x00 },
```

```c
        { 0x00, 0x00, 0x00, 0x00 },
    };
    unsigned char round_key[4][4] =
    {
        { 0x00, 0x05, 0x00, 0x00 },
        { 0x00, 0x00, 0x00, 0x00 },
        { 0x00, 0x00, 0xE0, 0x00 },
        { 0xA4, 0x00, 0x00, 0x00 },
    };

    round_key_addition(byte_matrix, round_key);
    assert_true(byte_matrix[1][0] == 0x00);
    assert_true(byte_matrix[0][3] == 0x00);
    assert_true(byte_matrix[2][2] == 0x00);
    round_key_addition(byte_matrix, round_key);
    assert_true(byte_matrix[1][0] == 0x05);
    assert_true(byte_matrix[0][3] == 0xA4);
    assert_true(byte_matrix[2][2] == 0xE0);
}

static void test_generate_round_keys(void **state) {
    char key[16] = "abcdefghijklmnop";
    unsigned char round_keys[3][4][4];
    generate_round_keys(round_keys, key);
    assert_true(round_keys[0][0][0] == 'a');
    assert_true(round_keys[0][3][3] == 'p');
    assert_true(round_keys[1][0][1] == 0xCA);
}

static void test_encipher_decipher_block(void **state) {
    unsigned char key[16] = "abcdefghijklmnop", block[16] = "Hello it is good", previous_block[16];

    for (int i = 0; i < 16; i++) {
        previous_block[i] = block[i];
    }

    encipher_block(block, key);
    decipher_block(block, key);

    for (int i = 0; i < 16; i++) {
        assert_true(previous_block[i] == block[i]);
    }
}

int main(void) {
    const struct CMUnitTest tests[] = {
        cmocka_unit_test(test_xor_columns),
        cmocka_unit_test(test_byte_sub_transformation),
        cmocka_unit_test(test_shift_row_transformation),
        cmocka_unit_test(test_mix_column_transformation),
        cmocka_unit_test(test_round_key_addition),
        cmocka_unit_test(test_generate_round_keys),
        cmocka_unit_test(test_encipher_decipher_block)
    };
    return cmocka_run_group_tests(tests, NULL, NULL);
}
```