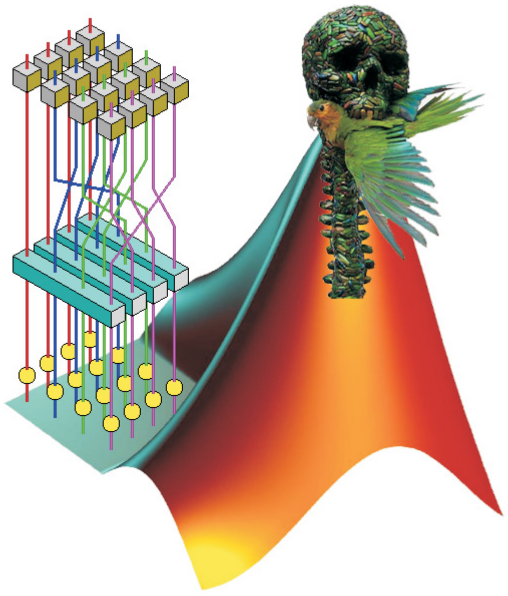


MATLAB Implementation of the

Advanced Encryption Standard

Jörg J. Buchholz
http://buchholz.hs-bremen.de

December 19, 2001

Contents

1 Introduction

This paper discusses a MATLAB implementation of the Advanced Encryption Standard (AES) [7]. AES is based on the block cipher Rijndael [4] [5] and became the designated successor of the Data Encryption Standard (DES) [8] which has been implemented in a tremendous number of cryptographic modules worldwide since 1977. MATLAB [1] is a matrix-oriented programming language, perfectly suited for the matrix-based data structure of AES.

Even though this implementation is fully operational, (i. e. it can be utilized to encrypt arbitrarily chosen plaintext into ciphertext and vice versa), the main optimization parameter of this implementation has not been execution speed but understandability. Assembler programmers might throw their hands up in horror, looking at shifting or substitution functions that have been coded algorithmically step-by-step instead of using a simple predefined lookup table; the primary goal of this ”educational” paper is to explain in greater detail *what* has to be done, rather than *how* it could be done for speed optimization reasons.

Also the question *why* certain algorithms have been chosen, e. g. with respect to the resistance against differential and linear cryptanalysis, is far beyond the scope of this paper. Interested readers are referred to the annex of the AES proposal [6] or a good book on cryptography [9]. Even Galois fields, the workhorse of modern cryptography, are introduced in a very pragmatic, engineer-friendly way, touching only as much mathematical background as necessary.

Furthermore, in order to minimize the number of if-then-else-conditions, a key length of 128 bits (16 bytes) has been implemented only; the extension to 24 or 32 bytes key lengths, as defined in [7], can easily be realized by altering the corresponding constants.

2 Finite Field Arithmetics

The following section introduces the different representation forms of a byte and discusses the basic arithmetics of finite fields.

A finite field, also called a Galois Field [3], [2], is a field with only finitely many elements. The finite field GF(2^8) e. g. consists of the $2^8 = 256$ different numbers ($0 \ldots 255$) represented by one byte (8 bits). Special xor- and modulo-operations, explained in detail in the following sections, make sure that the sum and the product of two finite field elements remain within the range of the original finite field.

2.1 Byte Representation Forms

The following four sections convert an example through the four usual representation forms of a finite field element.

2.1.1 Binary Representation

A byte consists of 8 bits, leading to the binary representation (index b) of an arbitrarily chosen example:

$$10100011_b \tag{1}$$

2.1.2 Decimal Representation

This example can be represented in decimal form (index d) by multiplying every bit by its corresponding power of two:

$$\begin{aligned} 1 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 &= 2^7 + 2^5 + 2^1 + 2^0 \\ &= 128 + 32 + 2 + 1 \\ &= 163_d \end{aligned} \tag{2}$$

Matlab uses the predefined function `bin2dec` (”binary to decimal”) to perform this conversion. Note the use of single quotation marks to input the binary representation as a string (character array):

```
>> bin2dec ('10100011')
ans =
   163
```

Example 1: Matlab example of `bin2dec`

2.1.3 Hexadecimal Representation

The numbers 0 ... 15 can be expressed by a group of four bits called a *nibble*. The numbers 10 ... 15 cannot be represented by a single decimal digit (0 ... 9) and are therefore "abbreviated" by the letters A ... F in hexadecimal notation (index h):

$$\begin{aligned}
0000_b &= 0_d = 0_h \\
0001_b &= 1_d = 1_h \\
0010_b &= 2_d = 2_h \\
0011_b &= 3_d = 3_h \\
0100_b &= 4_d = 4_h \\
0101_b &= 5_d = 5_h \\
0110_b &= 6_d = 6_h \\
0111_b &= 7_d = 7_h \\
1000_b &= 8_d = 8_h \\
1001_b &= 9_d = 9_h \\
1010_b &= 10_d = \mathrm{A}_h \\
1011_b &= 11_d = \mathrm{B}_h \\
1100_b &= 12_d = \mathrm{C}_h \\
1101_b &= 13_d = \mathrm{D}_h \\
1110_b &= 14_d = \mathrm{E}_h \\
1111_b &= 15_d = \mathrm{F}_h
\end{aligned}$$

The conversion from binary to hexadecimal is now very straightforward. The byte is divided into two nibbles and each nibble is represented by its hexadecimal digit:

$$10100011_b = \underbrace{1010}_{\mathrm{A}_h}\underbrace{0011}_{3_h}{}_b = \mathrm{A}3_h \tag{3}$$

For the conversion from hexadecimal back to decimal every hexadecimal digit is multiplied by its valence: The left digit is multiplied by 16, while the right one is multiplied by 1 and is therefore just added:

$$\mathrm{A}3_h = \mathrm{A} \cdot 2^4 + 3 \cdot 2^0 = 10 \cdot 16 + 3 \cdot 1 = 160 + 3 = 163_d \tag{4}$$

MATLAB offers the predefined function `hex2dec` to convert hexadecimal numbers back to their decimal representation:

```
>> hex2dec ('A3')
ans =
   163
```

Example 2: MATLAB example of `hex2dec`

2.1.4 Polynomial Representation

The polynomial representation of a byte is very similar to the conversion from binary to decimal in Equation (2). Substituting every 2 on the left hand side of Equation (2) by an x defines a polynomial using the bits of the binary form as coefficients of the powers of x:

$$1 \cdot x^7 + 0 \cdot x^6 + 1 \cdot x^5 + 0 \cdot x^4 + 0 \cdot x^3 + 0 \cdot x^2 + 1 \cdot x^1 + 1 \cdot x^0 = x^7 + x^5 + x + 1 \tag{5}$$

Note the fact, that the coefficients of this polynomial (representing a byte or GF(2^8) element) can only be 1 (or 0 respectively).

2.2 Polynomial Addition

"Usually" two polynomials are added by adding the coefficients of like powers of x according to Figure 1.

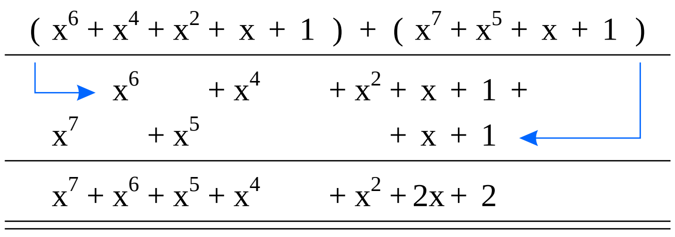

$(x^6 + x^4 + x^2 + x + 1) + (x^7 + x^5 + x + 1)$
$x^6 \quad + x^4 \quad + x^2 + x + 1 +$
$x^7 \quad + x^5 \quad + x + 1$
$x^7 + x^6 + x^5 + x^4 \quad + x^2 + 2x + 2$

Figure 1: "Classical" polynomial addition

Since this might lead to some coefficients of the resulting polynomial *not* being 0 or 1 (e. g. $2x$ and 2 in Figure 1), this "classical" sum does not represent a byte (i. e. an element of the original finite field).

In order to make sure that the resulting polynomial has only binary coefficients, the xor (exclusive or) operation depicted in Table 1 is used for the addition. Since the xor-"sum" of two 1's is not 2 but 0 ($1 \operatorname{xor} 1 = 0$), no 2-coefficient can appear.

x	y	x xor y
0	0	0
0	1	1
1	0	1
1	1	0

Table 1: xor operation

Figure 2 shows the bit-wise xor of two bytes (finite field elements), always resulting in another byte (element of the same finite field).

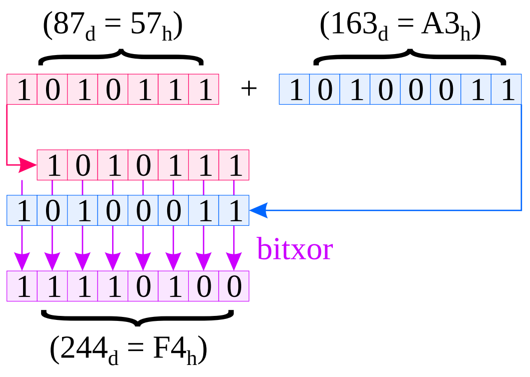


Figure 2: Binary polynomial addition

The resulting byte

$$244_d = \mathrm{F4}_h = 11110100_b = x^7 + x^6 + x^5 + x^4 + x^2 \tag{6}$$

directly corresponds to the polynomial of Figure 1, if the "non-binary" terms $2x$ and 2 are omitted there.

The bit-wise xor operation `bitxor` is a build-in function of MATLAB and is used throughout AES, whenever two bytes are added:

```
>> bitxor (87, 163)
ans =
   244
```

Example 3: MATLAB example of `bitxor`

2.3 Polynomial Multiplication

Two polynomials are multiplied by multiplying each summand of the first polynomial by (every summand of) the second polynomial and adding the coefficients of like powers (see Figure 3).

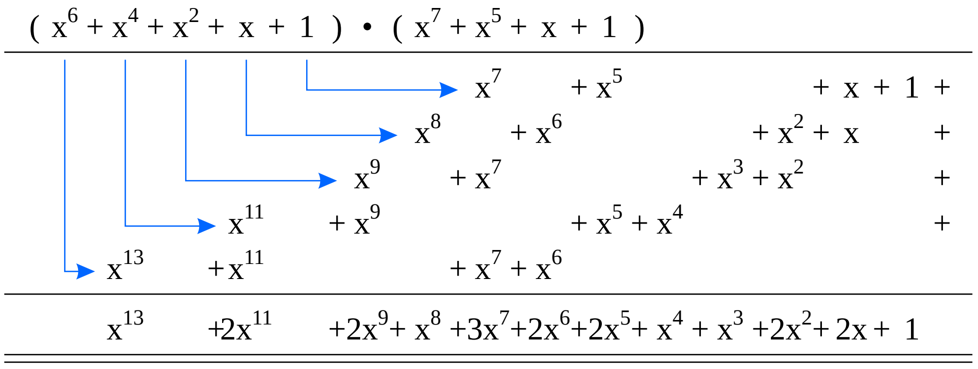

(x^6 + x^4 + x^2 + x + 1) • (x^7 + x^5 + x + 1)
x^7 + x^5 + x + 1 +
x^8 + x^6 + x^2 + x +
x^9 + x^7 + x^3 + x^2 +
x^11 + x^9 + x^5 + x^4 +
x^13 + x^11 + x^7 + x^6
x^13 +2x^11 +2x^9 + x^8 +3x^7 +2x^6 +2x^5 + x^4 + x^3 +2x^2 + 2x + 1

Figure 3: "Classical" polynomial multiplication

Once again, some coefficients of the resulting polynomial in Figure 3 are 2 or even 3 and have to be treated differently. The generalization of the xor-concept would now omit every power having an even coefficient and reduce every odd coefficient to 1, leading to a polynomial of

$$x^{13} + x^8 + x^7 + x^4 + x^3 + 1 \tag{7}$$

On the bit level (see Figure 4) the same result is achieved by shifting the second byte one bit to the left for every bit in the first byte. If a bit in the first byte is 0, a 0-byte is used instead of the second byte. Finally all corresponding bits are xor'ed.

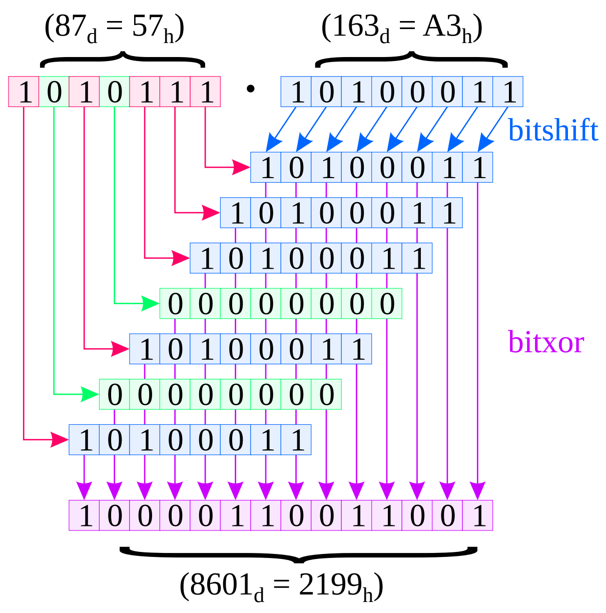
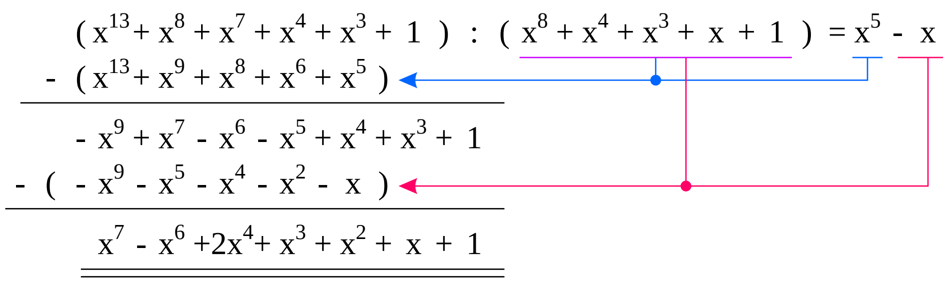


Figure 4: Binary polynomial multiplication

Unfortunately the resulting polynomial (7) has a degree greater than 7, can therefore not be expressed in one byte (i. e. it is not a GF(2^8) element) and has to be transformed back into the "byte range" by the modulo division described in the next section.

2.4 Polynomial Division

The manual algorithm to divide two polynomials is depicted in Figure 5.



(x^13 + x^8 + x^7 + x^4 + x^3 + 1) : (x^8 + x^4 + x^3 + x + 1) = x^5 - x
- (x^13 + x^9 + x^8 + x^6 + x^5)
- x^9 + x^7 - x^6 - x^5 + x^4 + x^3 + 1
- (- x^9 - x^5 - x^4 - x^2 - x)
x^7 - x^6 + 2x^4 + x^3 + x^2 + x + 1

Figure 5: "Classical" polynomial division

The greatest power of the numerator (x^{13}) is divided by the greatest power of the denominator (x^8) yielding the first resulting term (x^5). This term is multiplied by the complete denominator ($\rightarrow x^{13} + x^9 + x^8 + x^6 + x^5$) and subtracted from the numerator, resulting in a new numerator ($-x^9 + x^7 - x^6 - x^5 + x^4 + x^3 + 1$). This procedure is repeated until the greatest power of the new numerator has become less than the greatest power of the denominator. The final numerator $(x^7 - x^6 + 2x^4 + x^3 + x^2 + x + 1)$ is the remainder of this modulo operation. Applying the "generalized xor-rules" (even coefficients $\rightarrow 0$, odd coefficients $\rightarrow 1$) to the remainder leaves the desired byte-conform polynomial:

$$x^7 + x^6 + x^3 + x^2 + x + 1 \tag{8}$$

The bit level operations illustrated in Figure 6 achieve the same result by bit-wise shift and xor operations: The denominator is shifted to the left until its most significant bit (MSB) matches the MSB of the numerator. The subtraction is then performed via xor, resulting in a new, smaller numerator. The shifting and xor-ing is repeated, until the resulting numerator (the remainder) fits into one byte.

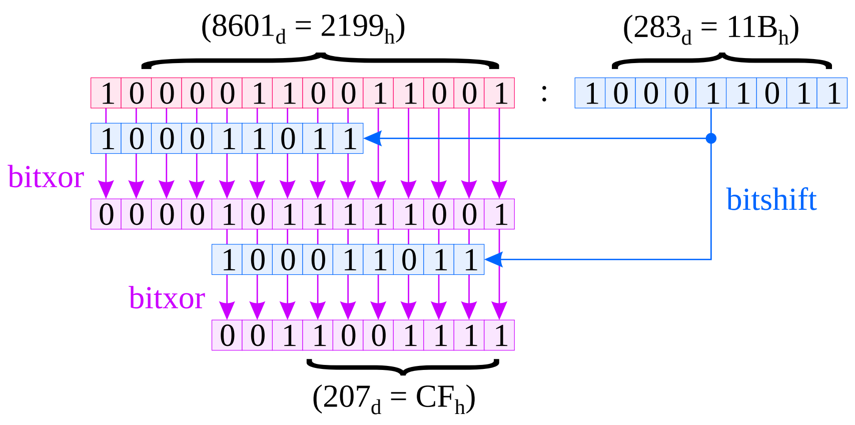


Figure 6: Binary polynomial division

2.5 `poly_mult` Implementation

The AES-function `poly_mult` (Listing 1) performs the multiplication of two polynomials (`a` and `b`) in GF(2^8) using a third polynomial (`mod_pol`) for the modular reduction.

```
1   function ab = poly_mult (a, b, mod_pol)
2   ab = 0;
3   for i_bit = 1 : 8
4       if bitget (a, i_bit)
5           b_shift = bitshift (b, i_bit - 1);
6           ab = bitxor (ab, b_shift);
7       end
8   end
9   for i_bit = 16 : -1 : 9
10      if bitget (ab, i_bit)
11          mod_pol_shift = bitshift (mod_pol, i_bit - 9);
12          ab = bitxor (ab, mod_pol_shift);
13      end
14  end
```

Listing 1: MATLAB function `poly_mult`

After the initialization (Line 2) `poly_mult` accomplishes the multiplication (Lines 3 . . . 8) and the modular reduction (Line 9 . . . 14) in two very similar loops.

For the multiplication (compare Figure 4) every bit (Line 3) of the first factor `a` is tested (Line 4) and if it is present, the second factor `b` is shifted to the left (Line 5) and xor-ed to the accumulating result `ab` (Line 6).

The modular reduction (compare Figure 6) interprets the intermediate result `ab` as the numerator, loops through all the bits of the numerator, starting with the MSB (Line 9) and shifts (Line 11) the modulo polynomial `mod_pol` ”under” detected (Line 10) bits of the numerator, in order to perform the appropriate xor-operation (Line 12).

The MATLAB Example 4 summarizes the operations depicted in the previous sections by multiplying the bytes

$$87_d = 1010111_b = x^6 + x^4 + x^2 + x + 1 \tag{9}$$

and

$$163_d = 10100011_b = x^7 + x^5 + x + 1 \tag{10}$$

using the standard AES modulo polynomial

$$283_d = 100011011_b = x^8 + x^4 + x^3 + x + 1 \tag{11}$$

resulting in

$$87_d \bullet 163_d \mod 283_d = 207_d \tag{12}$$

by a call to the function `poly_mult`:

```
>> poly_mult (87, 163, 283)
ans =
   207
```

Example 4: MATLAB example of `poly_mult`

3 aes_demo

The MATLAB program `aes_demo` demonstrates the use of the AES-package.

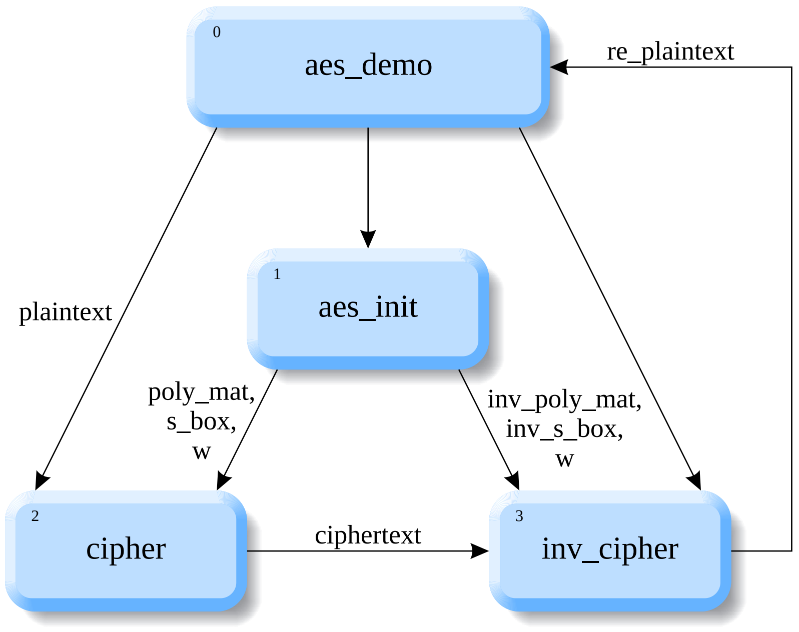


Figure 7: "Main" AES demonstration program `aes_demo`

As depicted in Figure 7 and Listing 2 the first task of the "main" program `aes_demo` is a call to `aes_init` in Line 1. This initialization routine supplies [1] the actual en- and decryption functions (`cipher` and `inv_cipher`) with the expanded key schedule `w`, the substitution tables `s_box` and `inv_s_box`, and the polynomial matrices `poly_mat` and `inv_poly_mat`. These quantities have to be generated only once and can be used by any subsequent en- or decipher call.

The Lines 2 ... 4 define the 16 byte (128 bit) of exemplary `plaintext` to be encrypted. In Line 5 this input block is passed to the encryption function `cipher`, which returns the corresponding 16 byte of `ciphertext`.

In order to demonstrate the decryption process too, the ciphertext is then forwarded to the decipher function `inv_cipher` in Line 6, resulting in the reprocessed plaintext block `re_plaintext`, which can be compared to and certainly has to equal the original `plaintext`.

[1] Figure 7 indicates that `aes_init` directly passes its parameters to `cipher` and `inv_cipher`. This is not really the case. In the actual implementation, the parameters are first returned from `aes_init` to `aes_demo` and then passed to `cipher` and `inv_cipher` by `aes_demo`.

```
[s_box, inv_s_box, w, poly_mat, inv_poly_mat] = aes_init;
plaintext_hex = {'00' '11' '22' '33' '44' '55' '66' '77' ...
                 '88' '99' 'aa' 'bb' 'cc' 'dd' 'ee' 'ff'};
plaintext = hex2dec (plaintext_hex);
ciphertext = cipher (plaintext, w, s_box, poly_mat);
re_plaintext = inv_cipher (ciphertext, w, inv_s_box, inv_poly_mat);
```

Listing 2: ”Main” program `aes_demo`

4 aes_init

Before doing any actual en- or decryption, the initialization function `aes_init` has to be called once.

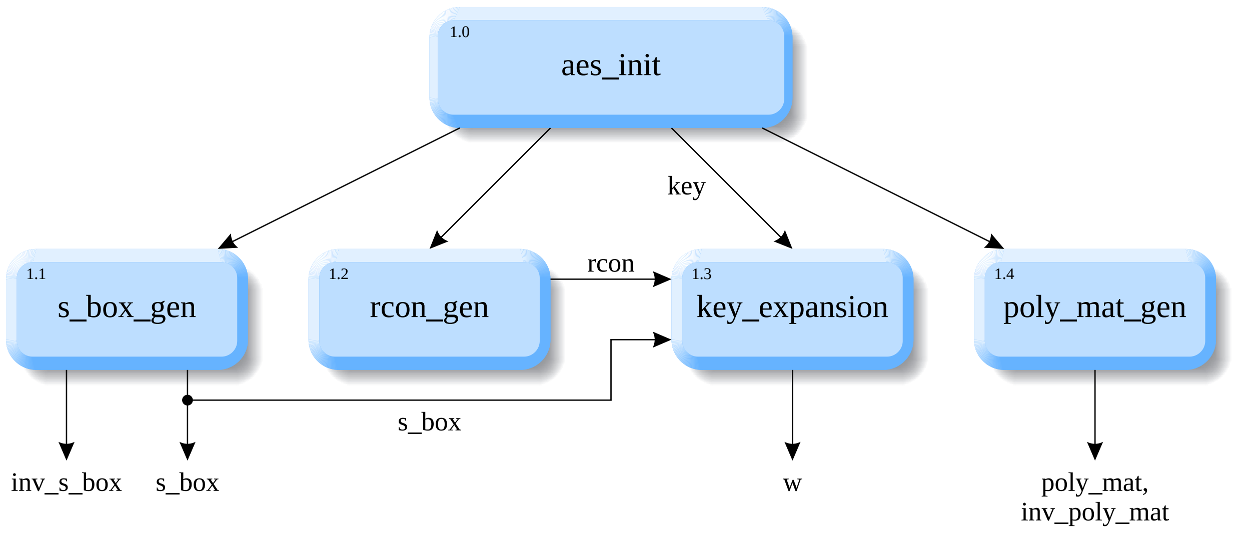


Figure 8: Initialization function `aes_init`

`aes_init` (Figure 8 and Listing 3) generates the two substitution tables `s_box` and `inv_s_box` by a call to `s_box_gen` (Line 2), defines the round constant vector `rcon` (Line 3) and an exemplary `key` (Lines 4 . . . 6) and computes the expanded key schedule `w` (Line 7). Additionally the two polynomial matrices `poly_mat` and `inv_poly_mat` are generated (Line 8).

```
1  function [s_box, inv_s_box, w, poly_mat, inv_poly_mat] = aes_init
2  [s_box, inv_s_box] = s_box_gen;
3  rcon = rcon_gen;
4  key_hex = {'00' '01' '02' '03' '04' '05' '06' '07' ...
5             '08' '09' '0a' '0b' '0c' '0d' '0e' '0f'};
6  key = hex2dec(key_hex); w = key_expansion (key, s_box, rcon);
7  [poly_mat, inv_poly_mat]= poly_mat_gen;
```

Listing 3: Initialization function `aes_init`

5 s_box_gen

The substitution tables (S-boxes) `s_box` and `inv_s_box` are used by the expanded key schedule function `key_expansion` and the en- and decrypting functions `cipher` and `inv_cipher` to directly substitute a byte (element of GF(2^8)) by another byte of the same finite field.

In any speed optimized real-world application the substitution tables would definitely be hard coded a priori in a constant (see Example 7); but in the scope of this educational paper it seems to be interesting, *how* the S-boxes can be generated.

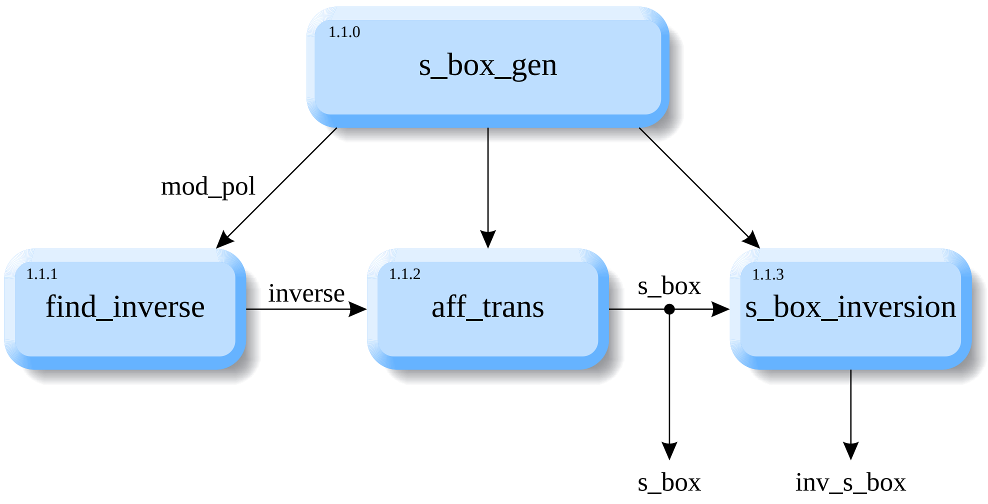


Figure 9: S-box generation function `s_box_gen`

The function `s_box_gen` (Figure 9 and Listing 4) creates the S-boxes (Line 1) by searching for the inverses of all elements of GF(2^8) (Lines 4 . . . 6) by the use of `find_inverse` (Line 5) and by applying affine transformations to all inverses (Lines 7 . . . 9) via `aff_trans` (Line 8). Finally the inverse S-Box `inv_s_box`, to be used in `inv_cipher`, is constructed from `s_box` by `s_box_inversion` (Line 10).

Line 2 defines the standard AES modular reduction polynomial `mod_pol` declared in Equation (11) and Line 3 takes care of the fact that the inverse of 0 is defined as 0. Note that MATLAB arrays (vectors and matrices) start with an index of 1.

```
function [s_box, inv_s_box] = s_box_gen
mod_pol = bin2dec ('100011011');
inverse(1) = 0;
for i = 1 : 255
    inverse(i + 1) = find_inverse (i, mod_pol);
end
for i = 1 : 256
    s_box(i) = aff_trans (inverse(i));
end
inv_s_box = s_box_inversion (s_box);
```

Listing 4: S-box generation function `s_box_gen`

5.1 `find_inverse`

The first step in the S-box generating process is to search for the multiplicative inverses of all elements of the finite field GF(2^8). In other words: For all possible 256 byte values b, find the byte b^{-1} that satisfies

$$b \bullet b^{-1} = 1 \tag{13}$$

where $\bullet$ denotes the polynomial multiplication defined in `poly_mult`.

The standard algorithm to perform such an inversion is called extended Euclidean algorithm, a C++ implementation of which can be found e. g. in [9]. This educational paper chooses a different, more pragmatic inversion approach, that is extremely slow on the one hand but very straightforward on the other hand:

Line 1 of Listing 5 declares the function `find_inverse` that is called with the input parameters `b_in`, which is the byte to be inverted and the modulo polynomial `mod_pol`, with respect to which the inversion has to take place.

The algorithm itself simply loops (Lines 2 ... 8) through all possible byte values and test-wise computes the product of the byte to be inverted (`b_in`) and the current test candidate (`i`) in Line 3. If the product ”accidently” equals 1 (Line 4), Equation (13) is met, the inverse is found (Line 5) and the loop (and the function) enjoys its well deserved break (Line 6).

```
function b_inv = find_inverse (b_in, mod_pol)
for i = 1 : 255
    prod = poly_mult (b_in, i, mod_pol);
    if prod == 1
        b_inv = i;
        break
    end
end
```

Listing 5: Inverse finding function `find_inverse`

The functionality of `find_inverse` is demonstrated in Example 5. The AES-Matlab function `find_inverse` is called in order to search for the inverse of 152_d with respect to AES' standard modulo polynomial 283_d. The answer is 42_d, which is immediately proven by multiplying 152_d by 42_d with a result of 1.

```
>> find_inverse (152, 283)
ans =
    42
>> poly_mult (152, 42, 283)
ans =
     1
```

Example 5: Matlab example of `find_inverse`

5.2 aff_trans

After the inverses of all bytes have been found, the second step of the S-box creation process is an affine transformation, consisting of a polynomial multiplication with a specific constant ($31_d = 00011111_b$) modulo another constant ($257_d = 100000001_b$) and the xor-addition of a third constant ($99_d = 01100011_b$):

$$b_{out} = b_{in} \bullet 31_d \mod 257_d \oplus 99_d \tag{14}$$

where b_{in} represents the input byte to be transformed, $\oplus$ denotes the bit-wise xor operation, and the output byte after the transformation is returned in b_{out}.

Line 1 of Listing 6 declares the Matlab function `aff_trans` transforming the input byte `b_in` into the output byte `b_out`. The three constants are defined in binary form in Lines 2 ... 4, Line 5 does the modulo multiplication and in Line 6 the bit-wise xor operation is performed.

```
function b_out = aff_trans (b_in)
mod_pol = bin2dec ('100000001');
mult_pol = bin2dec ('00011111');
add_pol = bin2dec ('01100011');
temp = poly_mult (b_in, mult_pol, mod_pol);
b_out = bitxor (temp, add_pol);
```

Listing 6: Affine transformation function `aff_trans`

Example 6 shows the affine transformation of 42_d (which is the result of the inverse finding Example 5) into 70_d.

```
>> aff_trans (42)
ans =
    70
```

Example 6: MATLAB example of `aff_trans`

5.3 `s_box_inversion`

The inverse S-box is used in the decrypting function `inv_cipher` to revert the substitution carried out via the S-box.

The corresponding AES-MATLAB function declared in Line 1 of Listing 7 takes the S-box (`s_box`) as its input and generates the inverse S-box `inv_s_box` in a single loop (Lines 2 . . . 4). The loop runs through all elements of the S-box, interprets the current S-box element value as an index into the inverse S-box and inserts the values 0 . . . 255 at the appropriate places in the inverse S-box (Line 3).

Note the ±1 offsets in Line 3, once again resulting from the fact, that MATLAB arrays start with an index of 1.

```
1   function inv_s_box = s_box_inversion (s_box)
2   for i = 1 : 256
3       inv_s_box(s_box(i) + 1) = i - 1;
4   end
```

Listing 7: S-box inversion function `s_box_inversion`

Example 7 presents AES' S-box and inverse S-box, resulting from a call to `s_box_gen`. The `1` in the parameter list of `s_box_gen(1)` switches the verbose mode of the actually implemented version of `s_box_gen` on, persuading the function to display intermediate results.

In order to use the S-box e. g. to substitute the byte 152_d by 70_d according to Example 5 and Example 6, the element of the S-box with an index of 152 has to be found. Keeping in mind that the indexing starts at 0 and that one row of the S-box in Example 7 holds 16 elements, the hexadecimal value 46_h (which indeed represents the expected decimal value of 70_d) can be found in the tenth row and the ninth column:

$$152 = (10 - 1) \cdot 16 + (9 - 1)$$

The backsubstitution from 70_d back to 152_d finds the value $98_h = 152_d$ in the fifth row and the seventh column of the inverse S-box `inv_s_box`:

$$70 = (5 - 1) \cdot 16 + (7 - 1)$$

```
>> s_box_gen(1);
********************************************
*                                          *
*       S - B O X   C R E A T I O N        *
*                                          *
*   (this might take a few seconds ;-))    *
*                                          *
********************************************

    s_box : 63 7c 77 7b f2 6b 6f c5 30 01 67 2b fe d7 ab 76
            ca 82 c9 7d fa 59 47 f0 ad d4 a2 af 9c a4 72 c0
            b7 fd 93 26 36 3f f7 cc 34 a5 e5 f1 71 d8 31 15
            04 c7 23 c3 18 96 05 9a 07 12 80 e2 eb 27 b2 75
            09 83 2c 1a 1b 6e 5a a0 52 3b d6 b3 29 e3 2f 84
            53 d1 00 ed 20 fc b1 5b 6a cb be 39 4a 4c 58 cf
            d0 ef aa fb 43 4d 33 85 45 f9 02 7f 50 3c 9f a8
            51 a3 40 8f 92 9d 38 f5 bc b6 da 21 10 ff f3 d2
            cd 0c 13 ec 5f 97 44 17 c4 a7 7e 3d 64 5d 19 73
            60 81 4f dc 22 2a 90 88 46 ee b8 14 de 5e 0b db
            e0 32 3a 0a 49 06 24 5c c2 d3 ac 62 91 95 e4 79
            e7 c8 37 6d 8d d5 4e a9 6c 56 f4 ea 65 7a ae 08
            ba 78 25 2e 1c a6 b4 c6 e8 dd 74 1f 4b bd 8b 8a
            70 3e b5 66 48 03 f6 0e 61 35 57 b9 86 c1 1d 9e
            e1 f8 98 11 69 d9 8e 94 9b 1e 87 e9 ce 55 28 df
            8c a1 89 0d bf e6 42 68 41 99 2d 0f b0 54 bb 16

inv_s_box : 52 09 6a d5 30 36 a5 38 bf 40 a3 9e 81 f3 d7 fb
            7c e3 39 82 9b 2f ff 87 34 8e 43 44 c4 de e9 cb
            54 7b 94 32 a6 c2 23 3d ee 4c 95 0b 42 fa c3 4e
            08 2e a1 66 28 d9 24 b2 76 5b a2 49 6d 8b d1 25
            72 f8 f6 64 86 68 98 16 d4 a4 5c cc 5d 65 b6 92
            6c 70 48 50 fd ed b9 da 5e 15 46 57 a7 8d 9d 84
            90 d8 ab 00 8c bc d3 0a f7 e4 58 05 b8 b3 45 06
            d0 2c 1e 8f ca 3f 0f 02 c1 af bd 03 01 13 8a 6b
            3a 91 11 41 4f 67 dc ea 97 f2 cf ce f0 b4 e6 73
            96 ac 74 22 e7 ad 35 85 e2 f9 37 e8 1c 75 df 6e
            47 f1 1a 71 1d 29 c5 89 6f b7 62 0e aa 18 be 1b
            fc 56 3e 4b c6 d2 79 20 9a db c0 fe 78 cd 5a f4
            1f dd a8 33 88 07 c7 31 b1 12 10 59 27 80 ec 5f
            60 51 7f a9 19 b5 4a 0d 2d e5 7a 9f 93 c9 9c ef
            a0 e0 3b 4d ae 2a f5 b0 c8 eb bb 3c 83 53 99 61
            17 2b 04 7e ba 77 d6 26 e1 69 14 63 55 21 0c 7d
```

Example 7: MATLAB call to `s_box_gen`

6 rcon_gen

The round constant matrix is used in the key expansion scheme (`key_expansion`). It is a 10×4 matrix of zeros except for the first column, which contains byte-conform powers of 2 (see Example 9).

Listing 8 defines the standard AES modulo polynomial `mod_pol` in Line 2 and a round constant initial value of 1 in Line 3. The remaining powers of 2 are then iteratively (Lines 4 ... 6) computed by polynomial multiplication of the previous value by 2 in Line 5. Finally three zero columns are appended in Line 7.

```
1  function rcon = rcon_gen
2  mod_pol = bin2dec ('100011011');
3  rcon(1) = 1;
4  for i = 2 : 10
5      rcon(i) = poly_mult (rcon(i-1), 2, mod_pol);
6  end
7  rcon = [rcon(:), zeros(10, 3)];
```

Listing 8: Round constant generation function `rcon_gen`

The polynomial nature of this multiplication by 2 (which actually is just a bit-shifting to the left) does not have any influence on the result as long as the product does not exceed a value of 255 (one byte). Therefore the first 8 elements of the first column of `rcon` are (compare to Example 9):

$$\begin{aligned}
1_d &= 01_h = 00000001_b \\
2_d &= 02_h = 00000010_b \\
4_d &= 04_h = 00000100_b \\
8_d &= 08_h = 00001000_b \\
16_d &= 10_h = 00010000_b \\
32_d &= 20_h = 00100000_b \\
64_d &= 40_h = 01000000_b \\
128_d &= 80_h = 10000000_b
\end{aligned}$$

The ninth element would "normally" be

$$128_d \cdot 2 = 256_d = 100_h = 100000000_b$$

which cannot be represented in one byte and is therefore "folded back" into byte range by modular reduction, as demonstrated in Example 8.

```
>> poly_mult (128, 2, 283)
ans =
    27
>> dec2hex (ans)
ans =
1B
```

Example 8: Modular reduction of the ninth element of `rcon`

Example 9 demonstrates the call to `rcon_gen` in verbose mode.

```
>> rcon_gen(1);
********************************************
*                                          *
*        R C O N   C R E A T I O N         *
*                                          *
********************************************

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

Example 9: MATLAB call to `rcon_gen`

7 key_expansion

The key expansion function (Figure 10) takes the user supplied 16 bytes long `key` and utilizes the previously created round constant matrix `rcon` and the substitution table `s_box` to generate a 176 byte long key schedule `w`, which will be used during the en- and decryption processes.

The blue arrowed closed loop in Figure 10 indicates that the functions `rot_word` and `sub_bytes` are called iteratively by the key expansion function.

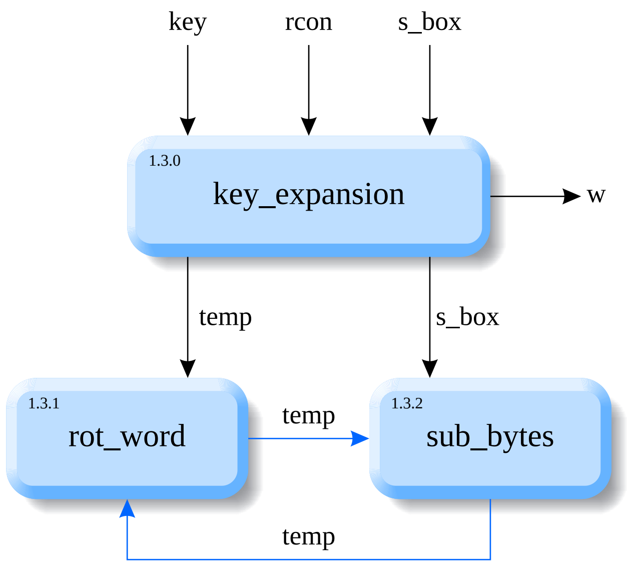


Figure 10: Key expansion function `key_expansion`

As depicted in Figure 11 the 16 bytes of the key vector are rearranged (row-wise) into a 4×4 initial key matrix ($k_{11} \ \ldots \ k_{44}$).

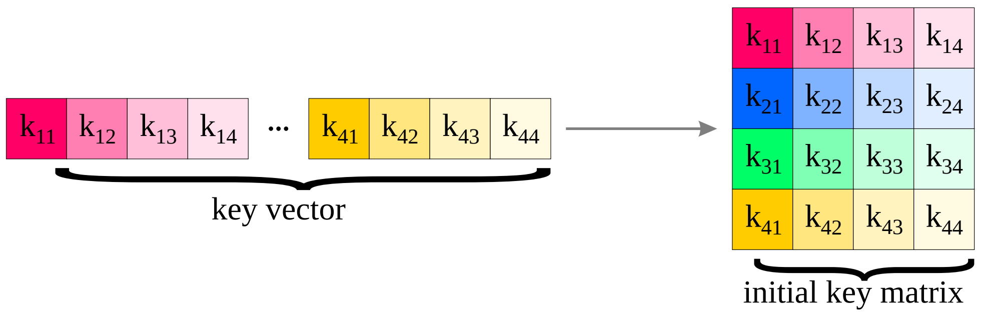


Figure 11: Row-wise reshape of the key row vector into the initial key matrix

The basic principle of the key expansion (Figure 12) is an element-wise xor-sum of two previous rows; the direct predecessor row and the row four rows up. The seventh row (k_{71} ... k_{74}) e. g. results from xor-ing the sixth row (k_{61} ... k_{64}) and the third row (k_{31} ... k_{34}).

Additionally every fourth row (row 5, row 9, ...) is created differently. Before applying the xor, the predecessor row is "rotated", "substituted" and "xor-ed" with its corresponding round constant (see Listing 9).

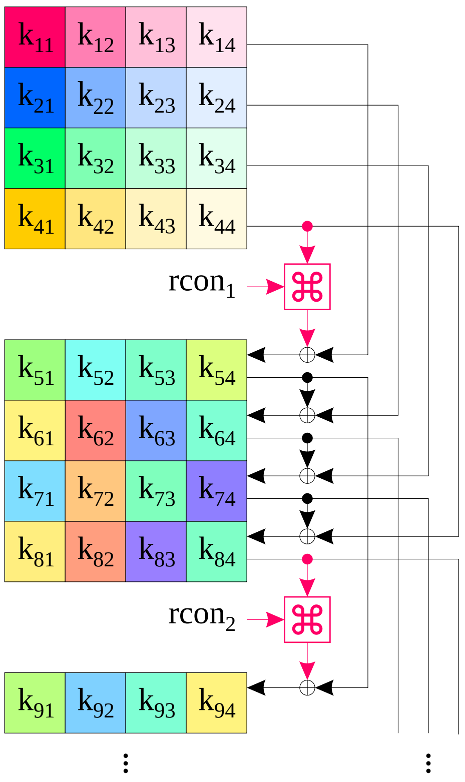


Figure 12: Key expansion

In Line 1 of Listing 9 the key expansion function with its input parameters `key` (user supplied key), `s_box` (substitution table), and `rcon` (round constant) is declared, returning the expanded key schedule `w`.

Line 2 utilizes the MATLAB command `reshape` to rearrange the 16 bytes of the key into a quadratic matrix of four rows, each holding four elements.

The effect of `reshape` is demonstrated in Example 10, rearranging the 4 elements of a row vector into a 2×2 matrix. Note the single quote character ('), which is MATLAB's transpose-operator, resulting in a row-wise insertion of the vector elements into the matrix.

```
>> 1:4
ans =
     1     2     3     4
>> reshape (ans, 2, 2)'
ans =
     1     2
     3     4
```

Example 10: MATLAB example of `reshape`

A loop (Lines 3 . . . 12) creates the remaining 40 rows of the key schedule. In Line 4 a buffer (`temp`) is filled with the previous row, which is xor-ed with the row four rows before in Line 11. Note the colon (:) as column index, which indicates that a complete row is addressed.

Line 5 checks if the row index matches 5, 9, 13, ... If this is the case, the buffered row is cyclically permuted (Line 6), substituted (Line 7) and xor-ed with the appropriate round constant (Lines 8 . . . 9), before it is finally xor-ed in Line 11 too.

```
 1  function w = key_expansion (key, s_box, rcon)
 2  w = (reshape (key, 4, 4))';
 3  for i = 5 : 44
 4      temp = w(i - 1, :);
 5      if mod (i, 4) == 1
 6          temp = rot_word (temp);
 7          temp = sub_bytes (temp, s_box);
 8          r = rcon ((i - 1)/4, :);
 9          temp = bitxor (temp, r);
10      end
11      w(i, :) = bitxor (w(i - 4, :), temp);
12  end
```

Listing 9: Key expansion function `key_expansion`

7.1 `rot_word`

The permutation function `rot_word` is declared in Line 1 of Listing 10. The input ”word” `w_in` is a row vector of four bytes and is cyclically permuted according to Figure 13. In MATLAB the permutation can easily be achieved by utilizing the new index vector in Line 2.

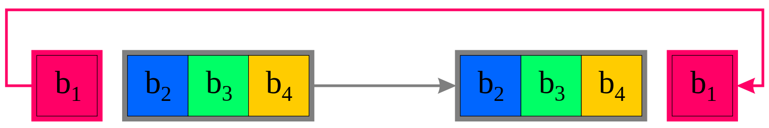


Figure 13: Word rotating

```
function w_out = rot_word (w_in)
w_out = w_in([2 3 4 1]);
```

Listing 10: Word rotating function `rot_word`

7.2 sub_bytes

The substitution function `sub_bytes` (Line 1 of Listing 11) applies the S-box to one or more input bytes `bytes_in` and truly manifests MATLAB's marvellous matrix manipulation mastery: In just one line of code (Line 2) the S-box is applied to a byte, a vector of bytes or even a complete matrix of bytes. This nice feature makes it feasible to utilize the same simple substitution function both for the substitution of a row vector in `key_expansion` and for the substitution of the state matrix in the encryption function `cipher`.

```
1   function bytes_out = sub_bytes (bytes_in, s_box)
2   bytes_out = s_box(bytes_in + 1);
```

Listing 11: Byte substitution function `sub_bytes`

Example 11 illustrates a call to `key_expansion` in verbose mode. The 16 bytes of the user supplied key can be found in the 4 $\times$ 4 initial key matrix (`w(1:4, :)`). The next output demonstrates the cyclic permutation of the fourth row (`After rot_word`). After the application of the S-box, the current round constant is xor-ed and the fifth row of the key schedule (`w(05, :)`) is finally computed by another xor with the first row.

For the determination of the sixth row (`w(06, :)`) all that rotating, substituting and rcon xor-ing does not have to take place. The sixth row is just the xor-sum of the fifth and the second row.

```
>> key_expansion (key, s_box, rcon, 1);
********************************************
*                                          *
*        K E Y   E X P A N S I O N         *
*                                          *
********************************************

w(1:4, :) :        00 01 02 03
                   04 05 06 07
                   08 09 0a 0b
                   0c 0d 0e 0f

After rot_word :   0d 0e 0f 0c

After sub_bytes :  d7 ab 76 fe

rcon(05, :) :      01 00 00 00

After rcon xor :   d6 ab 76 fe

w(05, :) :         d6 aa 74 fd

w(06, :) :         d2 af 72 fa

w(07, :) :         da a6 78 f1

w(08, :) :         d6 ab 76 fe

After rot_word :   ab 76 fe d6 ...
```

Example 11: MATLAB call to `key_expansion`

8 poly_mat_gen

The polynomial matrices `poly_mat` and `inv_poly_mat` are used in the `mix_columns` function called by the en- and decryption functions `cipher` and `inv_cipher` respectively. Both matrices have a size of 4×4 and every row is a cyclic permutation (right shift) of the previous row (see Example 14).

The function `poly_mat_gen` (Listing 12) achieves the row-wise permutation of such a circulant matrix by a call to the `cycle` function in Line 5 and 9. The matrix to be permuted is assembled in Lines 2 … 4 and Lines 6 … 8 by defining its first row in hexadecimal representation in Lines 2 and 6, by converting the row to decimal (Lines 3 and 7) and by quadruplicating the row into a 4 $\times$ 4 matrix in Lines 4 and 8.

```
1  function [poly_mat, inv_poly_mat] = poly_mat_gen
2  row_hex = {'02' '03' '01' '01'};
3  row = hex2dec (row_hex)';
4  rows = repmat (row, 4, 1);
5  poly_mat = cycle (rows, 'right');
6  inv_row_hex = {'0e' '0b' '0d' '09'};
7  inv_row = hex2dec (inv_row_hex)';
8  inv_rows = repmat (inv_row, 4, 1);
9  inv_poly_mat = cycle (inv_rows, 'right');
```

Listing 12: Polynomial matrix generating function `poly_mat_gen`

The quadruplication process via the use of the MATLAB function `repmat` is illustrated in Example 12.

```
>> row = [2 3 1 1]
row =
     2     3     1     1
>> rows = repmat (row, 4, 1)
rows =
     2     3     1     1
     2     3     1     1
     2     3     1     1
     2     3     1     1
```

Example 12: Row quadruplication via `repmat`

8.1 `cycle`

The cyclic permutation of the matrix has been outsourced into the function `cycle`, enabling other functions like `shift_rows` and `inv_shift_rows` to use this functionality too.

As illustrated in Figure 14 the function `cycle` (called with the parameter `'right'`) cyclically permutes the rows of the input matrix. The first row is not shifted at all, the elements of the second row are shifted one position to the right (with the former right-most element reentering as the left-most one), and the elements of the third and fourth rows are shifted two and three positions to the right respectively.

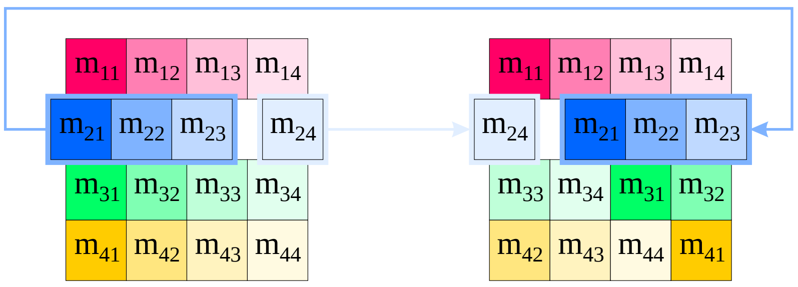

m_{11} m_{12} m_{13} m_{14}
m_{21} m_{22} m_{23} m_{24}
m_{31} m_{32} m_{33} m_{34}
m_{41} m_{42} m_{43} m_{44}
m_{11} m_{12} m_{13} m_{14}
m_{24} m_{21} m_{22} m_{23}
m_{33} m_{34} m_{31} m_{32}
m_{42} m_{43} m_{44} m_{41}

Figure 14: Cyclic permutation function `cycle` performs a row-wise right shift

Once again, the algorithm used in Listing 13 is pure overkill for a 4 $\times$ 4 matrix; a simple look-up table would do a much better job in an operational system. On the other hand, the code demonstrates MATLAB's interesting indexing capabilities to formulate a closed solution without a single `for`-loop.

The input parameters of the function `cycle` in Line 1 of Listing 13 are the matrix (`matrix_in`) to be permuted and the `direction` (`'left'` or `'right'`) into which the shifting has to take place.

In Lines 2 ... 6 a shift direction depending column vector (`col`) is defined (compare to Example 13). The appropriate row vector (`row`) is defined in Line 7. Both vectors are expanded (quadruplicated) into the corresponding matrices (`rows` and `cols`) in Lines 8 ... 9. In Line 10 an index matrix (`ind_mat`) is created by adding both matrices and "folding back" the result into the 1 ... 16 range via the modulo operator (`mod`). The index matrix is finally applied to the matrix to be permuted in Line 11, resulting in the permuted matrix `matrix_out`.

```
function matrix_out = cycle (matrix_in, direction)
if strcmp (direction, 'left')
    col = (0 : 5 : 15)';
else
    col = (16 : -3 : 7)';
end
row = 0 : 4 : 12;
cols = repmat (col, 1, 4);
rows = repmat (row, 4, 1);
ind_mat = mod (rows + cols, 16) + 1;
matrix_out = matrix_in (ind_mat);
```

Listing 13: Cyclic permutation function `cycle`

The elements of the index matrix are interpreted as linear indices into the two-dimensional matrix, as if the columns of the matrix were concatenated vertically. The element 7 in the third row and the fourth column of `ind_mat` in Example 13 e. g. addresses the element in the third row and the second column of the input matrix:

$$7 = (2 - 1) \cdot 4 + 3$$

```
>> col = (0 : 5 : 15)'
col =
     0
     5
    10
    15
>> row = 0 : 4 : 12
row =
     0     4     8    12
>> cols = repmat (col, 1, 4);
cols =
     0     0     0     0
     5     5     5     5
    10    10    10    10
    15    15    15    15
>> rows = repmat (row, 4, 1);
rows =
     0     4     8    12
     0     4     8    12
     0     4     8    12
     0     4     8    12
>> ind_mat = mod (rows + cols, 16) + 1
ind_mat =
     1     5     9    13
     6    10    14     2
    11    15     3     7
    16     4     8    12
```

Example 13: Generation of an index matrix

```
>> poly_mat_gen (1);
********************************************
*                                          *
*    P O L Y _ M A T   C R E A T I O N     *
*                                          *
********************************************

    poly_mat : 02 03 01 01
               01 02 03 01
               01 01 02 03
               03 01 01 02

inv_poly_mat : 0e 0b 0d 09
               09 0e 0b 0d
               0d 09 0e 0b
               0b 0d 09 0e
```

Example 14: MATLAB call to `poly_mat_gen`

9 `cipher`

The function `cipher` is the real McCoy, doing the actual encryption of the 16 byte long input vector of `plaintext` into the output `ciphertext` vector as illustrated in Figure 15. Further input parameters of `cipher`, that have been created by the initialization function `aes_init`, are the substitution table `s_box`, the key schedule `w`, and the polynomial matrix `poly_mat`.

`cipher` rearranges the plaintext vector into the `state` matrix and iteratively loops the `state` through `add_round_key`, `sub_bytes`, `shift_rows`, and `mix_columns`.

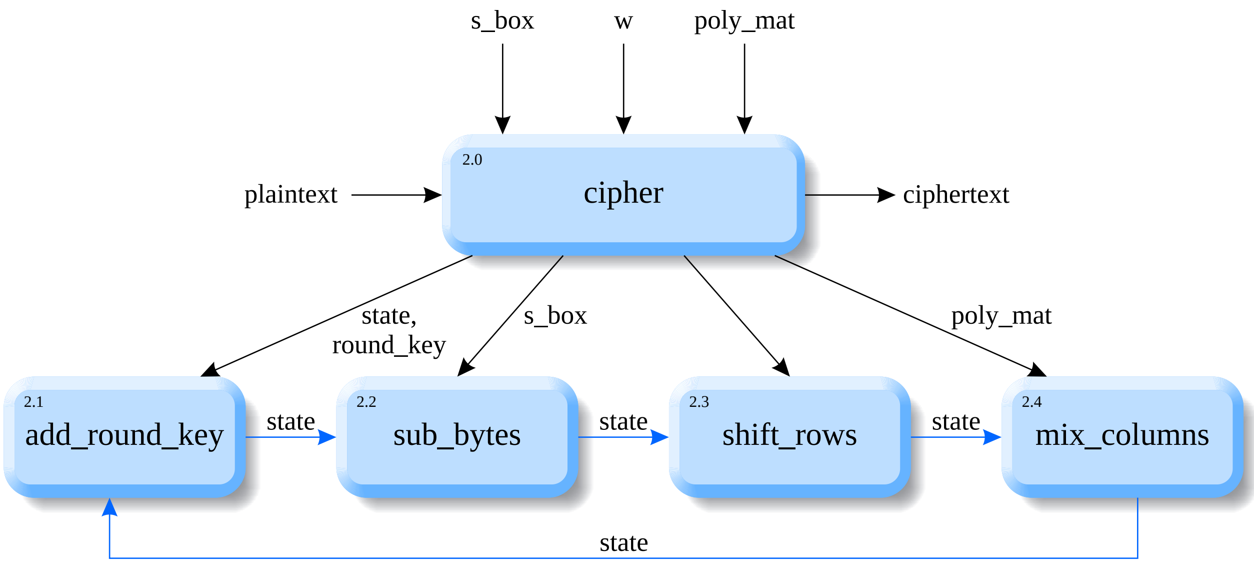


Figure 15: Encryption function `cipher`

Line 2 of Listing 14 column-wise rearranges the 16 elements (bytes) of the plaintext vector, which is illustrated in Figure 16 and demonstrated in Example 15 (`Initial_state`).

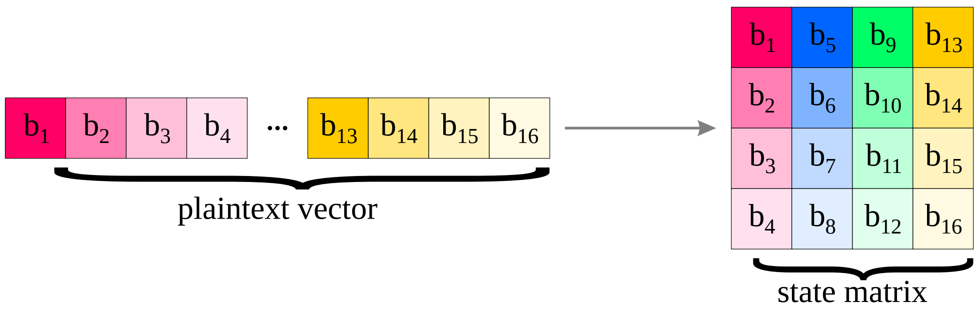

b_1 b_2 b_3 b_4 ... b_{13} b_{14} b_{15} b_{16}
plaintext vector
b_1 b_5 b_9 b_{13}
b_2 b_6 b_{10} b_{14}
b_3 b_7 b_{11} b_{15}
b_4 b_8 b_{12} b_{16}
state matrix

Figure 16: Column-wise reshape of the plaintext row vector into the state matrix

Line 3 extracts the first 4×4 matrix from the key schedule `w`. In order to match the column-oriented state matrix, this first round key (matrix) has to be transposed and is added to the initial state in Line 4 (compare to Example 15).

The state transformations inside the loop (Lines 5 ... 11) are repeated nine times. They consist of an application of the S-box in Line 6, a cyclical permutation of the state row elements in Line 7, a polynomial matrix multiplication in Line 8, the extraction of the current round key matrix in Line 9, and the binary addition of the round key in Line 10.

In the final round, which has been coded separately in Lines 12 ... 15, the `mix_columns` operation is missing.

Line 16 reshapes the final state (i. e. the `ciphertext`) back into a 16 byte long row vector.

```
 1  function ciphertext = cipher (plaintext, w, s_box, poly_mat)
 2  state = reshape (plaintext, 4, 4);
 3  round_key = (w(1:4, :))';
 4  state = add_round_key (state, round_key);
 5  for i_round = 1 : 9
 6      state = sub_bytes (state, s_box);
 7      state = shift_rows (state);
 8      state = mix_columns (state, poly_mat);
 9      round_key = (w((1:4) + 4*i_round, :))';
10      state = add_round_key (state, round_key);
11  end
12  state = sub_bytes (state, s_box);
13  state = shift_rows (state);
14  round_key = (w(41:44, :))';
15  state = add_round_key (state, round_key);
16  ciphertext = reshape (state, 1, 16);
```

Listing 14: Encryption function `cipher`

9.1 add_round_key

The `add_round_key` function declared in Line 1 of Listing 15 just has to perform a bitwise xor of the state matrix and the round key matrix. MATLAB can do this matrix operation in just one line of code (Line 2).

```
function state_out = add_round_key (state_in, round_key)
state_out = bitxor (state_in, round_key);
```

Listing 15: Round key adding function `add_round_key`

The functionality of `add_round_key` can be verified in Example 15 (`State at start of round 1`).

9.2 `shift_rows`

As illustrated in Figure 17 the function `shift_rows` cyclically permutes (shifts) the rows of the state matrix to the left.

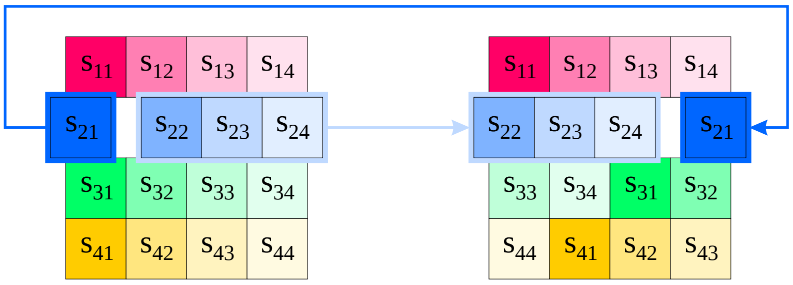

S_{11} S_{12} S_{13} S_{14}
S_{21} S_{22} S_{23} S_{24}
S_{31} S_{32} S_{33} S_{34}
S_{41} S_{42} S_{43} S_{44}
S_{11} S_{12} S_{13} S_{14}
S_{22} S_{23} S_{24} S_{21}
S_{33} S_{34} S_{31} S_{32}
S_{44} S_{41} S_{42} S_{43}

Figure 17: Row shifting function `shift_rows`

Fortunately, the cyclic permutation algorithm has already be defined in function `cycle`. Therefore, the function `shift_rows`, declared in Line 1 of Listing 16, simply has to call `cycle` with the appropriate shift direction parameter (`'left'`) in Line 2.

```
1   function state_out = shift_rows (state_in)
2   state_out = cycle (state_in, 'left');
```

Listing 16: Row shifting function `shift_rows`

Once again, Example 15 (`After shift_rows`) demonstrates the operation of `shift_rows`.

9.3 `mix_columns`

The `mix_columns` transformation computes the new state matrix $\mathbf{S}'$ by left-multiplying the current state matrix $\mathbf{S}$ by the polynomial matrix $\mathbf{P}$:

$$\mathbf{S}' = \mathbf{P} \bullet \mathbf{S} \tag{15}$$

The polynomial matrix **P** has already been defined in `poly_mat_gen`:

$$\begin{bmatrix} s'_{11} & s'_{12} & s'_{13} & s'_{14} \\ s'_{21} & s'_{22} & s'_{23} & s'_{24} \\ s'_{31} & s'_{32} & s'_{33} & s'_{34} \\ s'_{41} & s'_{42} & s'_{43} & s'_{44} \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \bullet \begin{bmatrix} s_{11} & s_{12} & s_{13} & s_{14} \\ s_{21} & s_{22} & s_{23} & s_{24} \\ s_{31} & s_{32} & s_{33} & s_{34} \\ s_{41} & s_{42} & s_{43} & s_{44} \end{bmatrix} \tag{16}$$

Every element of the left hand side matrix is the scalar product of the corresponding row of **P** and the column of **S**, e. g.:

$$s'_{23} = \begin{bmatrix} 1 & 2 & 3 & 1 \end{bmatrix} \bullet \begin{bmatrix} s_{13} \\ s_{23} \\ s_{33} \\ s_{43} \end{bmatrix} = 1 \bullet s_{13} \oplus 2 \bullet s_{23} \oplus 3 \bullet s_{33} \oplus 1 \bullet s_{43} \tag{17}$$

where $\bullet$ denotes the binary polynomial modulo multiplication (`poly_mult`) and $\oplus$ represents the corresponding bit-wise xor operation.

Since the declaration and definition of polynomial matrices as MATLAB-objects, having their own methods for multiplication, would go beyond the scope of this paper, the function `mix_columns`, declared in Line 1 of Listing 17, has to perform a complete standard matrix-matrix-multiplication involving the usual three nested loops (Lines 3, 4, and 6).

The input parameters of `mix_columns` are the state matrix (`state_in`) to be transformed (**S** in Equation 15) and the polynomial matrix **P** (`poly_mat`), which has been created in `poly_mat_gen`. Line 2 defines the AES modulo polynomial to be used in the inner polynomial multiplication in Lines 7 ... 10.

The double loop in Lines 3 ... 4 addresses every single element of the transformed matrix state $\mathbf{S}'$ (as in Equation 16) via its row and column indices. Line 5 initializes the state buffer `temp_state` which holds the cumulative sum during the element-wise computation of the right hand side expression of Equation 17 (e. g. $1 \bullet s_{13} \oplus 2 \bullet s_{23}$). The inner loop (Lines 6 ... 12) loops through all summands of Equation 17 (e. g. $3 \bullet s_{33}$), buffers every single product in `temp_prod` (Lines 7 ... 10) and accumulates (bit-wise xor) the current product to the state buffer in Line 11, which is finally inserted into the transformed state matrix (`state_out`) in Line 13.

```
function state_out = mix_columns (state_in, poly_mat)
mod_pol = bin2dec ('100011011');
for i_col_state = 1 : 4
    for i_row_state = 1 : 4
        temp_state = 0;
        for i_inner = 1 : 4
            temp_prod = poly_mult (...
                        poly_mat(i_row_state, i_inner), ...
                        state_in(i_inner, i_col_state), ...
                        mod_pol);
            temp_state = bitxor (temp_state, temp_prod);
        end
        state_out(i_row_state, i_col_state) = temp_state;
    end
end
```

Listing 17: Column mixing function `mix_columns`

Due to the intricate nature of the `mix_columns` function, its functionality cannot be verified in Example 15 at a glance.

```
>> cipher (in, w, s_box, poly_mat, 1);
********************************************
*                                          *
*               C I P H E R                *
*                                          *
********************************************

Initial state :                  00 44 88 cc
                                 11 55 99 dd
                                 22 66 aa ee
                                 33 77 bb ff

Initial round key :              00 04 08 0c
                                 01 05 09 0d
                                 02 06 0a 0e
                                 03 07 0b 0f

State at start of round 1 :      00 40 80 c0
                                 10 50 90 d0
                                 20 60 a0 e0
                                 30 70 b0 f0

After sub_bytes :                63 09 cd ba
                                 ca 53 60 70
                                 b7 d0 e0 e1
                                 04 51 e7 8c

After shift_rows :               63 09 cd ba
                                 53 60 70 ca
                                 e0 e1 b7 d0
                                 8c 04 51 e7

After mix_columns :              5f 57 f7 1d
                                 72 f5 be b9
                                 64 bc 3b f9
                                 15 92 29 1a

Round key :                      d6 d2 da d6
                                 aa af a6 ab
                                 74 72 78 76
                                 fd fa f1 fe ...
```

Example 15: MATLAB call to `cipher`

10 `inv_cipher`

The decryption function `inv_cipher` (Figure 18) step-by-step reverses the transformations of the encryption process. The input parameter of `inv_cipher` are the `ciphertext` to be decrypted (back) into `plaintext`, the inverse S-box (`inv_s_box`), the key schedule `w`, and the inverse polynomial matrix `inv_poly_mat`.

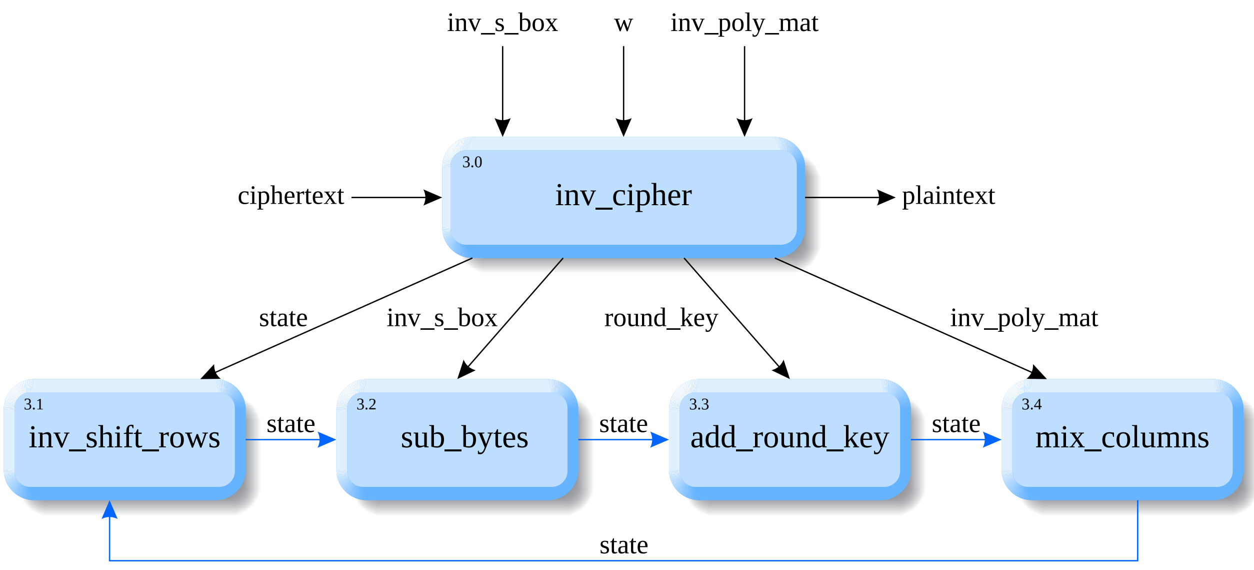


Figure 18: Decryption function `inv_cipher`

Listing 18 reveals the substantial similarity between the en- and the decryption function. The ciphertext is reshaped into the state matrix in Line 2. The first round key to be used here (Line 3) is the last one that has been used in `cipher`. As a consequence, the xor operation in Line 4 directly reverses the final `add_round_key` call in `cipher`. The last but one operation of `cipher` was a call to `shift_rows`, the effect of which is neutralized by the call to `inv_shift_rows` in Line 6. The `sub_bytes` transformation in Line 7 utilizes the inverse substitution table `inv_s_box`, reversing the substitution in `cipher`. The same is true for the call to `mix_columns` in Line 10, which uses the inverse polynomial matrix `inv_poly_mat` and therefore compensates the corresponding call in `cipher`.

As in `cipher`, but in the opposite chronological order, it takes nine identical rounds (Line 5) of row shifting, byte substituting and column mixing and a final tenth round (again with a missing `mix_columns` call) to end up with the reshaped plaintext in Line 16.

```
function plaintext = inv_cipher (ciphertext, w, inv_s_box, inv_poly_mat)
state = reshape (ciphertext, 4, 4);
round_key = (w(41:44, :))';
state = add_round_key (state, round_key);
for i_round = 9 : -1 : 1
    state = inv_shift_rows (state);
    state = sub_bytes (state, inv_s_box);
    round_key = (w((1:4) + 4*i_round, :))';
    state = add_round_key (state, round_key);
    state = mix_columns (state, inv_poly_mat);
end
state = inv_shift_rows (state);
state = sub_bytes (state, inv_s_box);
round_key = (w(1:4, :))';
state = add_round_key (state, round_key);
plaintext = reshape (state, 1, 16);
```

Listing 18: Decryption function `inv_cipher`

10.1 `inv_shift_rows`

The function `inv_shift_rows` is supposed to reverse the effect of the corresponding function `shift_rows` in the encryption process. Since `shift_rows` performs left shifts, `inv_shift_rows` (Listing 19) simply has to shift all rows of the state matrix (back) to the right.

```
function state_out = inv_shift_rows (state_in)
state_out = cycle (state_in, 'right');
```

Listing 19: Inverse row shifting function `inv_shift_rows`

Example 16 demonstrates the application of `inv_cipher` to the ciphertext resulting from a previous call to `cipher`.

```
>> inv_cipher (out, w, inv_s_box, inv_poly_mat, 1);
********************************************
*                                          *
*       I N V E R S E   C I P H E R        *
*                                          *
********************************************

Initial state :                   69 6a d8 70
                                  c4 7b cd b4
                                  e0 04 b7 c5
                                  d8 30 80 5a

Initial round key :               13 e3 f3 4d
                                  11 94 07 2b
                                  1d 4a a7 30
                                  7f 17 8b c5

State at start of round 9 :       7a 89 2b 3d
                                  d5 ef ca 9f
                                  fd 4e 10 f5
                                  a7 27 0b 9f

After inv_shift_rows :            7a 89 2b 3d
                                  9f d5 ef ca
                                  10 f5 fd 4e
                                  27 0b 9f a7

After inv_sub_bytes :             bd f2 0b 8b
                                  6e b5 61 10
                                  7c 77 21 b6
                                  3d 9e 6e 89

Round key :                       54 f0 10 be
                                  99 85 93 2c
                                  32 57 ed 97
                                  d1 68 9c 4e

After add_round_key :             e9 02 1b 35
                                  f7 30 f2 3c
                                  4e 20 cc 21
                                  ec f6 f2 c7 ...
```

Example 16: MATLAB call to `inv_cipher`

List of Figures

List of Tables

List of Listings

List of Examples

References

[1] The Mathworks: MATLAB , *The Language of Technical Computing.* http://www.mathworks.com/products/matlab, (2001).

[2] The Mathworks: *Galois Field Computations.* http://www.mathworks.com/access/helpdesk/help/toolbox/comm/tutor3.shtml, Communications Toolbox, (2001).

[3] J. Daemen, L. R. Knudsen, and V. Rijmen: *The Galois Field GF(2^8).* http://www.ddj.com/documents/s=936/ddj9710e/9710es1.htm, Dr. Dobb's Journal, (October 1997).

[4] V. Rijmen: *The block cipher Rijndael.* http://www.esat.kuleuven.ac.be/~rijmen/rijndael/, (2001).

[5] J. Daemen, V. Rijmen: *AES proposal: Rijndael.* http://www.esat.kuleuven.ac.be/~rijmen/rijndael/rijndaeldocV2.zip, (1999).

[6] J. Daemen: *Annex to AES proposal Rijndael.* http://www.esat.kuleuven.ac.be/~rijmen/rijndael/PropCorr.PDF, (1998).

[7] National Institute of Standards and Technology: *Specification for the Advanced Encryption Standard (AES).* http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf, (2001).

[8] National Institute of Standards and Technology: *Data Encryption Standard (DES).* http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf, (2001).

[9] B. Schneier: *Applied Cryptography.* Addison-Wesley, (1996).