

## AES: Rijndael Cryptosystem

**Rijndael** is a secret key block cipher. Designed by Joan Daemen and Vincent Rijmen, it was chosen by NIST (National Institute of Standards and Technology) as the United States standard AES (Advanced Encryption Standard, FIPS 187), substituting DES. In this cryptosystem, key length and message block length can be 128, 192 or 256 bits. The standard fixed the message block length to be 128 bits, but allowed the three possible length values for the key.

### The finite field $GF(2^8)$

The elements of this field are **bytes**. Depending on the situation, we may express them in binary, hex or polynomial form.

The byte  $b_7b_6b_5b_4b_3b_2b_1b_0$  becomes the polynomial  $b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0$ .

For example,  $01010111=0x57$  would be  $x^6 + x^4 + x^2 + x + 1$ .

### Addition

Addition of two field elements corresponds to the usual addition of binary polynomials. For example,  $01010111+10000011$  is

$$(x^6 + x^4 + x^2 + x + 1) + (x^7 + x + 1) = x^7 + x^6 + x^4 + x^2 = 11010100$$

It also corresponds with XOR operation, which we denote  $\oplus$ . The identity element is  $00000000=0x00$ .

### Product

In order to multiply two elements of the finite field we should multiply them as binary polynomials and then reduce modulus  $m = x^8 + x^4 + x^3 + x + 1$ , that is, take the residue of the corresponding polynomial division. For example,

$$\begin{aligned} (x^6 + x^4 + x^2 + x + 1)(x^7 + x + 1) &= x^{13} + x^{11} + x^9 + x^8 + x^7 + \\ &\quad x^7 + x^5 + x^3 + x^2 + x + \\ &\quad x^6 + x^4 + x^2 + x + 1 \\ &= x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1 \\ &= x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1 \pmod{x^8 + x^4 + x^3 + x + 1} = x^7 + x^6 + 1. \end{aligned}$$

The identity element for this operation is  $00000001=0x01$ .

In  $GF(2^8)$  every element different from  $0x00$  has a multiplicative inverse. The inverse of the polynomial  $a$  is the unique polynomial  $b$  such that

$$ab = 1 \pmod{m}.$$

It can be computed using the extended Euclidean algorithm.

We can also write the elements different from  $0x00$  as powers of a certain generator  $g$ . For example, if we take  $g = x + 1 = 00000011 = 0x03$ , then

$$GF(2^8) = \{g, g^2, \dots, g^{254}, g^{255}(=g^0 = 1)\} \cup \{0\}$$

The product of two elements  $a = g^i$  and  $b = g^j$ , different from  $0x00$ , is  $ab = g^i g^j = g^{i+j} = g^{i+j \pmod{255}}$ , and the inverse of  $a$  is  $a^{-1} = (g^i)^{-1} = g^{-i} = g^{255-i}$ . In this case, product and computation of inverse can be performed by search in a table of 255 elements.

## Input of message and key

We convert the plaintext into a bit sequence of length multiple of 128, which is divided in blocks of length 128. Bits of such a block are arranged in a  $4 \times 4$  matrix, called **state**. The arrangement is done *by columns* and coefficients of the matrix are bytes. Namely, if the block is the byte array  $a_{00}a_{10}a_{20}a_{30}a_{01}a_{11}a_{21}a_{31} \dots a_{23}a_{33}$ , then the state is:

$a_{00}$	$a_{01}$	$a_{02}$	$a_{03}$
$a_{10}$	$a_{11}$	$a_{12}$	$a_{13}$
$a_{20}$	$a_{21}$	$a_{22}$	$a_{23}$
$a_{30}$	$a_{31}$	$a_{32}$	$a_{33}$

Similarly, key bits are placed in a matrix having 4 rows and **Nk** columns. The number Nk of columns will be 4, 6 or 8, depending on the key length being 128, 192 or 256, respectively. For example, if the key has length 192 we have  $Nk = 6$  and if we represent the key as the byte sequence  $k_{00}k_{10}k_{20}k_{30}k_{01} \dots k_{25}k_{35}$ , then the matrix is:

$$\text{clau} = \begin{array}{|c|c|c|c|c|c|} \hline k_{00} & k_{01} & k_{02} & k_{03} & k_{04} & k_{05} \\ \hline k_{10} & k_{11} & k_{12} & k_{13} & k_{14} & k_{15} \\ \hline k_{20} & k_{21} & k_{22} & k_{23} & k_{24} & k_{25} \\ \hline k_{30} & k_{31} & k_{32} & k_{33} & k_{34} & k_{35} \\ \hline \end{array}$$

## Enciphering algorithm

There is an initial transformation followed by **Nr** rounds. The number of rounds depends on Nk:

Nk	4	6	8
Nr	10	12	14

Through the KeyExpansion process, which is described below,  $Nr + 1$  round subkeys  $K_i$  are generated from the key  $K$ . These round subkeys are  $4 \times 4$  matrices. The first one is used in the initial transformation and then one in each round.

Let  $M$  be the 128 bit message block we want to encipher, represented as a  $4 \times 4$  state matrix as above. The enciphering algorithm has the following steps:

```

estat = AddRoundKey(M, K0)
For i=1 to Nr - 1
    estat = Roundi(estat, Ki)
EndFor
C = Roundfinal (estat, KNr)

```

where  $C$  is the ciphertext. For  $i = 1, \dots, Nr - 1$  the round Round<sub>i</sub> consists of the following operations:

```

estat = ByteSub(estat)
estat = ShiftRow(estat)
estat = MixColumn(estat)
estat = AddRoundKey(estat, Ki)

```

The Roundfinal omits the third one:

```

estat = ByteSub(estat)
estat = ShiftRow(estat)
estat = AddRoundKey(estat, KNr)

```

The input for the **AddRoundKey** function are two  $4 \times 4$  matrices and then it performs XOR to their coefficients:

$$A \oplus B = (a_{ij} \oplus b_{ij})_{i,j}$$

## ByteSub

This function operates byte by byte as follows:

- 1) For each byte different from  $0x00$  take the inverse in  $GF(2^8)$
- 2) Apply the following affine transformation (taking into account that the operations are mod 2): the image of the byte  $b_7b_6b_5b_4b_3b_2b_1b_0$  is the byte  $u_7u_6u_5u_4u_3u_2u_1u_0$ , where

$$\begin{pmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \\ u_7 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}.$$

## ShiftRow

The ShiftRow function leaves the first row unchanged and rotates to the left each other row of the state matrix: the second row is rotated one position, the third row two positions and the fourth row three positions. That is, if rows are numbered from 0 to 3, the  $i$ -th row is rotated  $i$  positions to the left.

## MixColumn

In this function, operations are performed over columns of the state matrix. Each column is considered as a polynomial of degree  $< 4$  with coefficients in  $GF(2^8)$ . This polynomial is multiplied by the polynomial  $c(Y) = 0x03Y^3 + 0x01Y^2 + 0x01Y + 0x02$  and finally we take the polynomial remainder of the division by  $Y^4 + 1$ . This is an endomorphism of the polynomial vector space and therefore it can be computed through matrix multiplication. Concretely, if  $(v_{0i}, v_{1i}, v_{2i}, v_{3i})$  is a column of the state, its image under the MixColumn function is  $(w_{0i}, w_{1i}, w_{2i}, w_{3i})$ , where

$$\begin{pmatrix} w_{0i} \\ w_{1i} \\ w_{2i} \\ w_{3i} \end{pmatrix} = \begin{pmatrix} 0x02 & 0x03 & 0x01 & 0x01 \\ 0x01 & 0x02 & 0x03 & 0x01 \\ 0x01 & 0x01 & 0x02 & 0x03 \\ 0x03 & 0x01 & 0x01 & 0x02 \end{pmatrix} \begin{pmatrix} v_{0i} \\ v_{1i} \\ v_{2i} \\ v_{3i} \end{pmatrix}.$$

## KeyExpansion

The key  $K$  is arranged in a  $4 \times Nk$  matrix. Its columns are denoted by  $clau(0), clau(1), \dots, clau(Nk - 1)$ .

As we said before, the KeyExpansion function generates  $Nr + 1$  subkeys. These subkeys are stored in matrix  $\mathbf{W}$ , having 4 rows and  $4(Nr + 1)$  columns.  $W(0), \dots, W(3)$  will be the subkey  $K_0$ ,  $W(4), \dots, W(7)$  the subkey  $K_1$  and so on. Columns of  $W$  are successively computed from the initial ones, according to the following algorithm <sup>1</sup>:

```

For  $i = 0$  to  $Nk - 1$ 
     $W(i) = clau(i)$ 
EndFor
For  $i = Nk$  to  $4 * (Nr + 1) - 1$ 
     $temp = W(i - 1)$ 
    If  $i = 0 \bmod Nk$ 
         $temp = ByteSub(RotByte(temp)) \oplus Rcon(i/Nk)$ 
    EndIf
    If  $Nk = 8$  and  $i = 4 \bmod Nk$ 
         $temp = ByteSub(temp)$ 
    EndIf
     $W(i) := W(i - Nk) \oplus temp$ 
EndFor

```

<sup>1</sup>ByteSub operates on each element of the column.

The **RotByte** function is rotation: column  $(a, b, c, d)$  becomes column  $(b, c, d, a)$ . On the other hand, **Rcon(i)** stands for the column  $(x^{i-1}, 0, 0, 0)$ , using polynomial notation, which means that we should take the remainder of division by  $m(x)$ .

## Deciphering algorithm

The deciphering algorithm has the same structure as the enciphering one: an initial transformation followed by  $N_r$  rounds. We use round subkeys obtained from **InvKeyExp**, denoted by  $\text{invK}_0, \text{invK}_1, \dots, \text{invK}_{N_r}$ . Let  $C$  be the block to decrypt, arranged as a  $4 \times 4$  matrix. The deciphering algorithm is:

```

estat = AddRoundKey(C, invKNr)
For i = Nr - 1 to 1
    estat = InvRoundi(estat, invKi)
EndFor
M = InvRoundfinal(estat, invK0)

```

For  $i = N_r - 1, \dots, 1$ , the transformations in  $\text{InvRound}_i$  are

```

estat = InvByteSub(estat)
estat = InvShiftRow(estat)
estat = InvMixColumn(estat)
estat = AddRoundKey(estat, invKi)

```

And in  $\text{InvRoundfinal}$  we have:

```

estat = InvByteSub(estat)
estat = InvShiftRow(estat)
estat = AddRoundKey(estat, invK0)

```

The new functions are precisely the inverses of the ones we have in the enciphering algorithm:

- **InvByteSub**. First we have to apply the inverse affine transformation  $\mathbf{y} = \mathbf{A}^{-1}(\mathbf{x} - \mathbf{v})$ , where

$$A^{-1} = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

Then, if the result is not  $0x00$ , we should take inverses in  $GF(2^8)$ .

- **InvShiftRow**. Apply to each row the same rotation as before, but now to the right.
- **InvMixColumn**. Each column is now multiplied by the inverse polynomial

$$d(Y) = 0x0B Y^3 + 0x0D Y^2 + 0x09 Y + 0x0E.$$

Using matrices:

$$\begin{pmatrix} v_{0i} \\ v_{1i} \\ v_{2i} \\ v_{3i} \end{pmatrix} = \begin{pmatrix} 0x0E & 0x0B & 0x0D & 0x09 \\ 0x09 & 0x0E & 0x0B & 0x0D \\ 0x0D & 0x09 & 0x0E & 0x0B \\ 0x0B & 0x0D & 0x09 & 0x0E \end{pmatrix} \begin{pmatrix} w_{0i} \\ w_{1i} \\ w_{2i} \\ w_{3i} \end{pmatrix}.$$

- **InvKeyExp**. Apply to the key  $K$  the KeyExpansion function, to obtain a matrix  $W$ . The initial 4 and the final 4 columns of  $W$  remain unchanged. To the remaining ones, we apply  $\text{InvMixColumn}$ . Therefore,  $\text{invK}_0 = K_0$ ,  $\text{invK}_{N_r} = K_{N_r}$ , and  $\text{invK}_i = \text{InvMixColumn}(K_i)$  for  $1 \leq i \leq N_r - 1$ .

## Messages and *padding*

Messages will be byte sequences. Therefore, 16 bytes are needed to fill a state matrix corresponding to a 128 bit block. We add the same kind of *padding* as in SHA, now for 128 bit blocks.

## Implementation: signatures

Define the class `aes` with the following methods:

```
public static byte[ ][ ][ ] keyExpansion(BigInteger K , int Nk, int Nr)
```

input: `K` is an integer representing the key, `Nk` is the number of columns of the key and `Nr` is the number of rounds;

output: list of the  $Nr + 1$  enciphering subkeys, first index refers to subkey, second one to the subkey row and the last one to the subkey column.

```
public static byte[ ][ ][ ] invKeyExpansion (BigInteger K, int Nk, int Nr)
```

input: `K` is an integer representing the key, `Nk` is the number of columns of the key and `Nr` is the number of rounds;

output: list of  $Nr + 1$  deciphering subkeys, first index refers to subkey, second one to the subkey row and the last one to the subkey column.

```
public static byte byteSub(byte subestat)
```

input: `subestat` is a byte;

output: a byte, obtained applying `ByteSub` to `subestat`.

```
public static byte invByteSub (byte subestat)
```

input: `subestat` is a byte;

output: a byte, obtained applying `InvByteSub` to `subestat`.

```
public static byte[ ][ ] shiftRow(byte[ ][ ] estat)
```

input: `estat` is a  $4 \times 4$  byte matrix;

output: a  $4 \times 4$  byte matrix, obtained applying `ShiftRow` to `estat`.

```
public static byte[ ][ ] invShiftRow(byte[ ][ ] estat)
```

input: `estat` is a  $4 \times 4$  byte matrix;

output: a  $4 \times 4$  byte matrix, obtained applying `InvShiftRow` to `estat`.

```
public static byte[ ][ ] mixColumn(byte[ ][ ] estat)
```

input: `estat` is a  $4 \times 4$  byte matrix;

output: a  $4 \times 4$  byte matrix, obtained applying `MixColumn` to `estat`.

```
public static byte[ ][ ] invMixColumn(byte[ ][ ] estat)
```

input: `estat` is a  $4 \times 4$  byte matrix;

output: a  $4 \times 4$  byte matrix, obtained applying `InvMixColumn` to `estat`.

```
public static byte[][] addRoundKey(byte[][] estat, byte[][] Ki)
```

input: **estat** and **K<sub>i</sub>** are  $4 \times 4$  byte matrices;

output: a  $4 \times 4$  byte matrix, obtained adding matrices **estat** and **K<sub>i</sub>** bit to bit.

```
public static byte[][] rijndael(byte[][] estat, byte[][][] W, int Nk, int Nr)
```

input: **estat** is a  $4 \times 4$  byte matrix, **Nk** is the key length divided by 32, **Nr** is the number of rounds and **W** is the matrix storing the enciphering subkeys;

output: a  $4 \times 4$  byte matrix, obtained applying the enciphering algorithm to **estat**.

```
public static byte[][] invRijndael(byte[][] estat, byte[][][] InvW, int Nk, int Nr)
```

input: **estat** is a  $4 \times 4$  byte matrix, **Nk** is the key length divided by 32, **Nr** is the number of rounds and **InvW** is the matrix storing the deciphering subkeys;

output: a  $4 \times 4$  byte matrix, obtained applying the deciphering algorithm to

```
public static byte[] xifrarAES (byte[] M, BigInteger K, int Lk)
```

input: **M** is a byte list representing the message to encipher, **K** is an integer representing the key and **Lk** is the key length (128, 192 or 256);

output: byte list representing the cryptogram obtained by enciphering the message **M** (with added padding) in **CBC mode** with key **K**.

```
public static byte[] desxifrarAES (byte[] C, BigInteger K, int Lk)
```

input: **C** is a byte list representing the cryptogram, **K** is an integer representing the key and **Lk** is the key length (128, 192 or 256);

output: byte list representing the message obtained after deciphering and padding removal.

## Test values

In FIPS 197 you can find detailed examples for `keyExpansion`, `rijndael` and `invRijndael`.