EXCERPT · FIPS 197 · PUBLISHED 26 NOVEMBER 2001 · UPDATED 9 MAY 2023

Advanced Encryption *Standard (AES)*

National Institute of Standards and Technology

Information Technology Laboratory · Federal Information Processing Standards Publication 197

Explanation The Advanced Encryption Standard (AES) specifies a FIPS-approved cryptographic algorithm that can be used to protect electronic data. The AES algorithm is a symmetric block cipher that can encrypt (encipher) and decrypt (decipher) digital information. The AES algorithm is capable of using cryptographic keys of 128, 192, and 256 bits to encrypt and decrypt data in blocks of 128 bits.

3.4 The State

Internally, the algorithms for the AES block ciphers are performed on a two-dimensional (four-by-four) array of bytes called the *state*. In the state array, denoted by s, each individual byte has two indices: a row index r in the range $0 \le r < 4$ and a column index c in the range $0 \le c < 4$. An individual byte of the state is denoted by either $s_{r,c}$ or $s[r, c]$.

In the specifications for the AES block cipher algorithms in Section 5, the first step is to copy the input array of bytes $in_0, in_1, \ldots, in_{15}$ to the state array s as follows:

$$s[r, c] = in[r + 4c] \quad \text{for } 0 \le r < 4 \text{ and } 0 \le c < 4. \tag{3.6}$$

A sequence of transformations is then applied to the state array, after which its final value is copied to the output array of bytes $out_0, out_1, \ldots, out_{15}$ as follows:

$$out[r + 4c] = s[r, c] \quad \text{for } 0 \le r < 4 \text{ and } 0 \le c < 4. \tag{3.7}$$

The correspondence between the indices of the input and output with the indices of the state array is illustrated in Fig. 1.

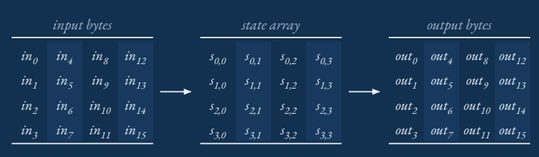


Figure 1. State array input and output

3.5 Arrays of Words

A *word* is a sequence of four bytes; a block consists of four words. The four columns of state array s are interpreted as an array v of four words as follows, in the notation of Fig. 1:

$$v_0 = \begin{pmatrix} s_{0,0} \\ s_{1,0} \\ s_{2,0} \\ s_{3,0} \end{pmatrix}, \quad v_1 = \begin{pmatrix} s_{0,1} \\ s_{1,1} \\ s_{2,1} \\ s_{3,1} \end{pmatrix}, \quad v_2 = \begin{pmatrix} s_{0,2} \\ s_{1,2} \\ s_{2,2} \\ s_{3,2} \end{pmatrix}, \quad v_3 = \begin{pmatrix} s_{0,3} \\ s_{1,3} \\ s_{2,3} \\ s_{3,3} \end{pmatrix}. \tag{3.8}$$

Thus, the column index c of s becomes the index for v, and the row index r of s becomes the index for the four bytes in each word.

Given a one-dimensional array u of words, $u[i]$ denotes the word that is indexed by i, and the sequence of four words $u[i], u[i + 1], u[i + 2], u[i + 3]$ is denoted by $u[i..i + 3]$.

4.2 Multiplication in GF(2^8)

The symbol • denotes multiplication in $\mathrm{GF}(2^8)$. Conceptually, this multiplication is defined on two bytes in two steps: 1) the two polynomials that represent the bytes are multiplied as polynomials, and 2) the resulting polynomial is reduced modulo the following fixed polynomial:

$$m(x) = x^8 + x^4 + x^3 + x + 1. \tag{4.3}$$

Within both steps, the individual coefficients of the polynomials are reduced modulo 2.

Thus, if $b(x)$ and $c(x)$ represent bytes b and c, then $b \bullet c$ is represented by the following modular reduction of their product as polynomials:

$$b(x)c(x) \mod m(x). \tag{4.4}$$

The modular reduction by $m(x)$ may be applied to intermediate steps in the calculation of $b(x)c(x)$; consequently, it is useful to consider the special case that $c(x) = x$ (i.e., $c = \{02\}$). In particular, the product $b \bullet \{02\}$ can be expressed as a function of b, denoted by $\mathrm{xTimes}(b)$, as follows:

$$\mathrm{xTimes}(b) = \begin{cases} \{b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0\ 0\} & \text{if } b_7 = 0 \\ \{b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0\ 0\} \oplus \{0\,0\,0\,1\,1\,0\,1\,1\} & \text{if } b_7 = 1. \end{cases} \tag{4.5}$$

Multiplication by higher powers of x (such as $\{04\}$, $\{08\}$, and $\{10\}$) can be implemented by the repeated application of xTimes(). For example, let $b = \{57\}$:

$$\begin{aligned} &\{57\} \bullet \{01\} = \{57\} \\ &\{57\} \bullet \{02\} = \mathrm{xTimes}(\{57\}) = \{ae\} \\ &\{57\} \bullet \{04\} = \mathrm{xTimes}(\{ae\}) = \{47\} \\ &\{57\} \bullet \{08\} = \mathrm{xTimes}(\{47\}) = \{8e\} \\ &\{57\} \bullet \{10\} = \mathrm{xTimes}(\{8e\}) = \{07\} \\ &\{57\} \bullet \{20\} = \mathrm{xTimes}(\{07\}) = \{0e\} \\ &\{57\} \bullet \{40\} = \mathrm{xTimes}(\{0e\}) = \{1c\} \\ &\{57\} \bullet \{80\} = \mathrm{xTimes}(\{1c\}) = \{38\}. \end{aligned} \tag{4.6}$$

These products facilitate the computation of any multiple of $\{57\}$. For example, because $\{13\} = \{10\} \oplus \{02\} \oplus \{01\}$, it follows that

$$\begin{aligned} \{57\} \bullet \{13\} &= \{57\} \bullet (\{01\} \oplus \{02\} \oplus \{10\}) \\ &= \{57\} \oplus \{ae\} \oplus \{07\} \\ &= \{fe\}. \end{aligned} \tag{4.7}$$

5. Algorithm Specifications

The general function for executing AES-128, AES-192, or AES-256 is denoted by Cipher(); its inverse is denoted by InvCipher().[1]

The core of the algorithms for Cipher() and InvCipher() is a sequence of fixed transformations of the state called a *round*. Each round requires an additional input called the *round key*; the round key is a block that is usually represented as a sequence of four words (i.e., 16 bytes).

An expansion routine, denoted by KeyExpansion(), takes the block cipher key as input and generates the round keys as output. In particular, the input to KeyExpansion() is represented as an array of words, denoted by *key*, and the output is an expanded array of words, denoted by *w*, called the *key schedule*.

The block ciphers AES-128, AES-192, and AES-256 differ in three respects: 1) the length of the key; 2) the number of rounds, which determines the size of the required key schedule; and 3) the specification of the recursion within KeyExpansion(). For each algorithm, the number of rounds is denoted by *Nr*, and the number of words of the key is denoted by *Nk*. (The number of words in the state is denoted by *Nb* for Rijndael in general; in this Standard, $Nb = 4$.) The specific values of *Nk*, *Nb*, and *Nr* are given in Table 3. No other configurations of Rijndael conform to this Standard.

For implementation issues relating to the key length, block size, and number of rounds, see Section 6.3.

Table 3. Key-Block-Round Combinations

	Key length		Block size		Number of rounds
	Nk	(in bits)	*Nb*	(in bits)	*Nr*
AES-128	4	128	4	128	10
AES-192	6	192	4	128	12
AES-256	8	256	4	128	14

The three inputs to Cipher() are: 1) the data input *in*, which is a block represented as a linear array of 16 bytes; 2) the number of rounds *Nr* for the instance; and 3) the round keys. Thus,

$$\begin{aligned} &\text{AES-128}(in, key) = \mathrm{Cipher}(in, 10, \mathrm{KeyExpansion}(key)) \\ &\text{AES-192}(in, key) = \mathrm{Cipher}(in, 12, \mathrm{KeyExpansion}(key)) \\ &\text{AES-256}(in, key) = \mathrm{Cipher}(in, 14, \mathrm{KeyExpansion}(key)). \end{aligned} \tag{5.1}$$

The inverse permutations are defined by replacing Cipher() with InvCipher() in Eq. 5.1.

The specifications of Cipher(), KeyExpansion(), and InvCipher() are given in Sections 5.1, 5.2, and 5.3, respectively.

5.1 Cipher()

The rounds in the specification of Cipher() are composed of the following four byte-oriented transformations on the state:

- SubBytes() applies a substitution table (S-box) to each byte.
- ShiftRows() shifts rows of the state array by different offsets.
- MixColumns() mixes the data within each column of the state array.
- AddRoundKey() combines a round key with the state.

The four transformations are specified in Sections 5.1.1–5.1.4. In those specifications, the transformed bit, byte, or block is denoted by appending the symbol $'$ as a superscript on the original variable (i.e., by b'_i, b', $s'_{i,j}$, or s').

The round keys for AddRoundKey() are generated by KeyExpansion(), which is specified in Section 5.2. In particular, the key schedule is represented as an array w of $4 * (Nr + 1)$ words.

Cipher() is specified in the pseudocode in Alg. 1.

Algorithm 1 Pseudocode for Cipher()

```
1:  procedure Cipher(in, Nr, w)
2:      state ← in                                   ▷ See Sec. 3.4
3:      state ← AddRoundKey(state, w[0..3])          ▷ See Sec. 5.1.4
4:      for round from 1 to Nr − 1 do
5:          state ← SubBytes(state)                  ▷ See Sec. 5.1.1
6:          state ← ShiftRows(state)                 ▷ See Sec. 5.1.2
7:          state ← MixColumns(state)                ▷ See Sec. 5.1.3
8:          state ← AddRoundKey(state, w[4 * round..4 * round + 3])
9:      end for
10:     state ← SubBytes(state)
11:     state ← ShiftRows(state)
12:     state ← AddRoundKey(state, w[4 * Nr..4 * Nr + 3])
13:     return state                                 ▷ See Sec. 3.4
14: end procedure
```

The first step (Line 2) is to copy the input into the state array using the conventions from Sec. 3.4. After an initial round key addition (Line 3), the state array is transformed by *Nr* applications of the round function (Lines 4–12); the final round (Lines 10–12) differs in that the MixColumns() transformation is omitted. The final state is then returned as the output (Line 13), as described in Section 3.4.

5.1.1 SubBytes()

SubBytes() is an invertible, non-linear transformation of the state in which a substitution table, called an S-box, is applied independently to each byte in the state. The AES S-box is denoted by SBox().

Let b denote an input byte to SBox(), and let c denote the constant byte $\{01100011\}$. The output byte $b' = \mathrm{SBox}(b)$ is constructed by composing the following two transformations:

1. Define an intermediate value $\tilde{b}$, as follows, where b^{-1} is the multiplicative inverse of b, as described in Section 4.4:

$$\tilde{b} = \begin{cases} \{00\} & \text{if } b = \{00\} \\ b^{-1} & \text{if } b \ne \{00\}. \end{cases} \tag{5.2}$$

2. Apply the following affine transformation of the bits of $\tilde{b}$ to produce the bits of b':

$$b'_i = \tilde{b}_i \oplus \tilde{b}_{(i+4) \bmod 8} \oplus \tilde{b}_{(i+5) \bmod 8} \oplus \tilde{b}_{(i+6) \bmod 8} \oplus \tilde{b}_{(i+7) \bmod 8} \oplus c_i. \tag{5.3}$$

The matrix form of Eq. (5.3) is given by Eq. (5.4) below:

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} \tilde{b}_0 \\ \tilde{b}_1 \\ \tilde{b}_2 \\ \tilde{b}_3 \\ \tilde{b}_4 \\ \tilde{b}_5 \\ \tilde{b}_6 \\ \tilde{b}_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}. \tag{5.4}$$

Figure 2 illustrates how SubBytes() transforms the state.

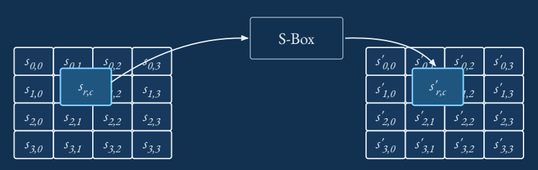


Figure 2. Illustration of SubBytes()

The AES S-box is presented in hexadecimal form in Table 4. For example, if $s_{r,c} = \{53\}$, then the substitution value would be determined by the intersection of the row with index '5' and the column with index '3' in Table 4, so that $s'_{r,c} = \{ed\}$.

Table 4. SBox(): substitution values for the byte xy (in hexadecimal format)

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

5.1.2 ShiftRows()

ShiftRows() is a transformation of the state in which the bytes in the last three rows of the state are cyclically shifted. The number of positions by which the bytes are shifted depends on the row index *r*, as follows:

$$s'_{r,c} = s_{r,(c+r) \bmod 4} \quad \text{for } 0 \le r < 4 \text{ and } 0 \le c < 4. \tag{5.5}$$

ShiftRows() is illustrated in Figure 3. In that representation of the state, the effect is to move each byte by *r* positions to the left in the row, cycling the left-most *r* bytes around to the right end of the row. The first row, where $r = 0$, is unchanged.

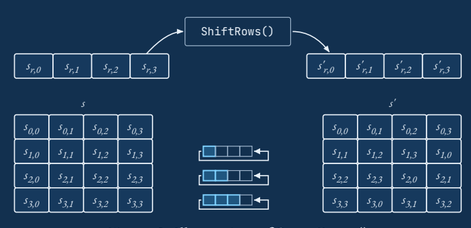


Figure 3. Illustration of ShiftRows()

5.1.3 MixColumns()

MixColumns() is a transformation of the state that multiplies each of the four columns of the state by a single fixed matrix, as described in Section 4.3, with its entries taken from the following word:

$$[a_0, a_1, a_2, a_3] = [\{02\}, \{01\}, \{01\}, \{03\}]. \tag{5.6}$$

Thus,

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad \text{for } 0 \le c < 4, \tag{5.7}$$

so that the individual output bytes are defined as follows:

$$\begin{aligned} s'_{0,c} &= (\{02\} \bullet s_{0,c}) \oplus (\{03\} \bullet s_{1,c}) \oplus s_{2,c} \oplus s_{3,c} \\ s'_{1,c} &= s_{0,c} \oplus (\{02\} \bullet s_{1,c}) \oplus (\{03\} \bullet s_{2,c}) \oplus s_{3,c} \\ s'_{2,c} &= s_{0,c} \oplus s_{1,c} \oplus (\{02\} \bullet s_{2,c}) \oplus (\{03\} \bullet s_{3,c}) \\ s'_{3,c} &= (\{03\} \bullet s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \bullet s_{3,c}). \end{aligned} \tag{5.8}$$

Figure 4 illustrates MixColumns().

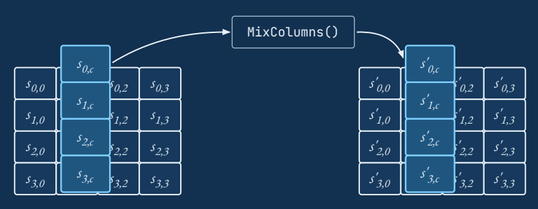


Figure 4. Illustration of MixColumns()

5.1.4 AddRoundKey()

AddRoundKey() is a transformation of the state in which a round key is combined with the state by applying the bitwise XOR operation. In particular, each round key consists of four words from the key schedule (described in Section 5.2), each of which is combined with a column of the state as follows:

$$[s'_{0,c}, s'_{1,c}, s'_{2,c}, s'_{3,c}] = [s_{0,c}, s_{1,c}, s_{2,c}, s_{3,c}] \oplus [w_{(4*round+c)}] \quad \text{for } 0 \le c < 4 \tag{5.9}$$

where *round* is a value in the range $0 \le round \le Nr$, and $w[i]$ is the array of key schedule words described in Section 5.2. In the specification of Cipher(), AddRoundKey() is invoked $Nr + 1$ times — once prior to the first application of the round function (see Alg. 1) and once within each of the *Nr* rounds, when $1 \le round \le Nr$.

The action of this transformation is illustrated in Fig. 5, where $l = 4 * round$. The byte address within words of the key schedule was described in Sec. 3.5.

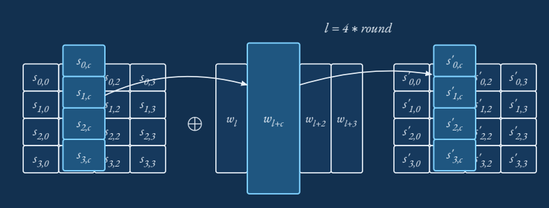


Figure 5. Illustration of AddRoundKey()

5.2 KeyExpansion()

KeyExpansion() is a routine that is applied to the key to generate $4 * (Nr + 1)$ words. Thus, four words are generated for each of the $Nr + 1$ applications of AddRoundKey() within the specification of Cipher(), as described in Section 5.1.4. The output of the routine consists of a linear array of words, denoted by $w[i]$, where i is in the range $0 \le i < 4 * (Nr + 1)$.

KeyExpansion() invokes 10 fixed words denoted by $Rcon[j]$ for $1 \le j \le 10$. These 10 words are called the *round constants*. For AES-128, a distinct round constant is called in the generation of each of the 10 round keys. For AES-192 and AES-256, the key expansion routine calls the first eight and seven of these same constants, respectively. The values of $Rcon[j]$ are given in hexadecimal notation in Table 5:

Table 5. Round constants

j	*Rcon[j]*	*j*	*Rcon[j]*
1	[01,00,00,00]	6	[20,00,00,00]
2	[02,00,00,00]	7	[40,00,00,00]
3	[04,00,00,00]	8	[80,00,00,00]
4	[08,00,00,00]	9	[1b,00,00,00]
5	[10,00,00,00]	10	[36,00,00,00]

The value of the left-most byte of $Rcon[j]$ in polynomial form is x^{j-1}. Note that for $j > 0$, these bytes may be generated by successively applying xTimes() to the byte represented by x^{j-1} (see Eq. 4.5).

Two transformations on words are called within KeyExpansion(): RotWord() and SubWord(). Given an input word represented as a sequence $[a_0, a_1, a_2, a_3]$ of four bytes,

$$\mathrm{RotWord}([a_0, a_1, a_2, a_3]) = [a_1, a_2, a_3, a_0], \tag{5.10}$$

and

$$\mathrm{SubWord}([a_0, \ldots, a_3]) = [\mathrm{SBox}(a_0), \mathrm{SBox}(a_1), \mathrm{SBox}(a_2), \mathrm{SBox}(a_3)]. \tag{5.11}$$

The expansion of the key proceeds according to the pseudocode in Alg. 2. The first *Nk* words of the expanded key are the key itself. Every subsequent word $w[i]$ is generated recursively from the preceding word, $w[i - 1]$, and the word *Nk* positions earlier, $w[i - Nk]$, as follows:

- If i is a multiple of *Nk*, then $w[i] = w[i - Nk] \oplus \mathrm{SubWord}(\mathrm{RotWord}(w[i - 1])) \oplus Rcon[i/Nk]$.
- For AES-256, if $i + 4$ is a multiple of 8, then $w[i] = w[i - Nk] \oplus \mathrm{SubWord}(w[i - 1])$.
- For all other cases, $w[i] = w[i - Nk] \oplus w[i - 1]$.

Algorithm 2 Pseudocode for KeyExpansion()

```
1:  procedure KeyExpansion(key)
2:      i ← 0
3:      while i ≤ Nk − 1 do
4:          w[i] ← key[4 * i..4 * i + 3]
5:          i ← i + 1
6:      end while                                    ▷ When the loop concludes, i = Nk.
7:      while i ≤ 4 * Nr + 3 do
8:          temp ← w[i − 1]
9:          if i mod Nk = 0 then
10:             temp ← SubWord(RotWord(temp)) ⊕ Rcon[i/Nk]
11:         else if Nk > 6 and i mod Nk = 4 then
12:             temp ← SubWord(temp)
13:         end if
14:         w[i] ← w[i − Nk] ⊕ temp
15:         i ← i + 1
16:     end while
17:     return w
18: end procedure
```

[1] Informally, these functions are sometimes called "encryption" and "decryption," but neutral terminology is appropriate because there are other applications of block ciphers besides encryption.

doi.org/10.6028/NIST.FIPS.197-upd1