Advanced Encryption Standard

Sachin Tripathi

IIT(ISM), Dhanbad

Outline

- Overview of AES & Inside Algorithm

- Mathematics behind this Algorithm

Conclusions

Motivation behind AES

A replacement of DES was needed since Key size is too small.

In 1990's the cracking of DES algorithm became possible.

Around 50hrs of brute forcing allowed to crack the message.

3DES secure but slow

In 1997, National Institute of Standards and Technology (NIST) called for new proposal

NIST requirements

Block cipher must be supported with 128 bit block size

Three key must be supported 128, 192 and 256 bits

Efficiency in software and hardware

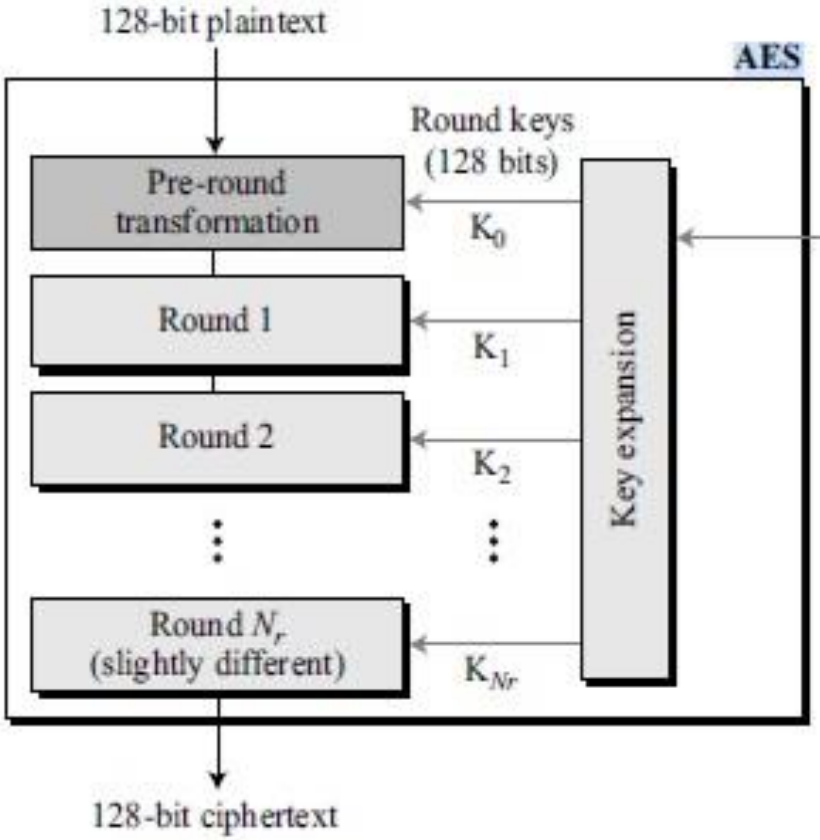
About AES

- AES is an encryption standard chosen by the National Institute of Standards and Technology (NIST) and accepted in worldwide as a desirable algorithm to encrypt sensitive data.
- It is the most widely used symmetric ciphers .
- In 2001 Rijndael algorithm designed by Rijment and Daemon of Belgium was declared as the winner of the competition.

In 1999, five finalist algorithms were announced:

- **Mars** by IBM Corporation
- **RC6** by RSA Laboratories
- **Rijndael**, by Joan Daemen and Vincent Rijmen
- **Serpent**, by Ross Anderson, Eli Biham and Lars Knudsen
- **Twofish**, by Bruce Schneier, John Kelsey, Doug Whiting, DavidWagner, Chris Hall and Niels Ferguson
- On October 2, 2000, **NIST** announced that it had chosen **Rijndael** as the **AES**.
- On November 26, 2001, **AES** was formally approved as a US federal standard.

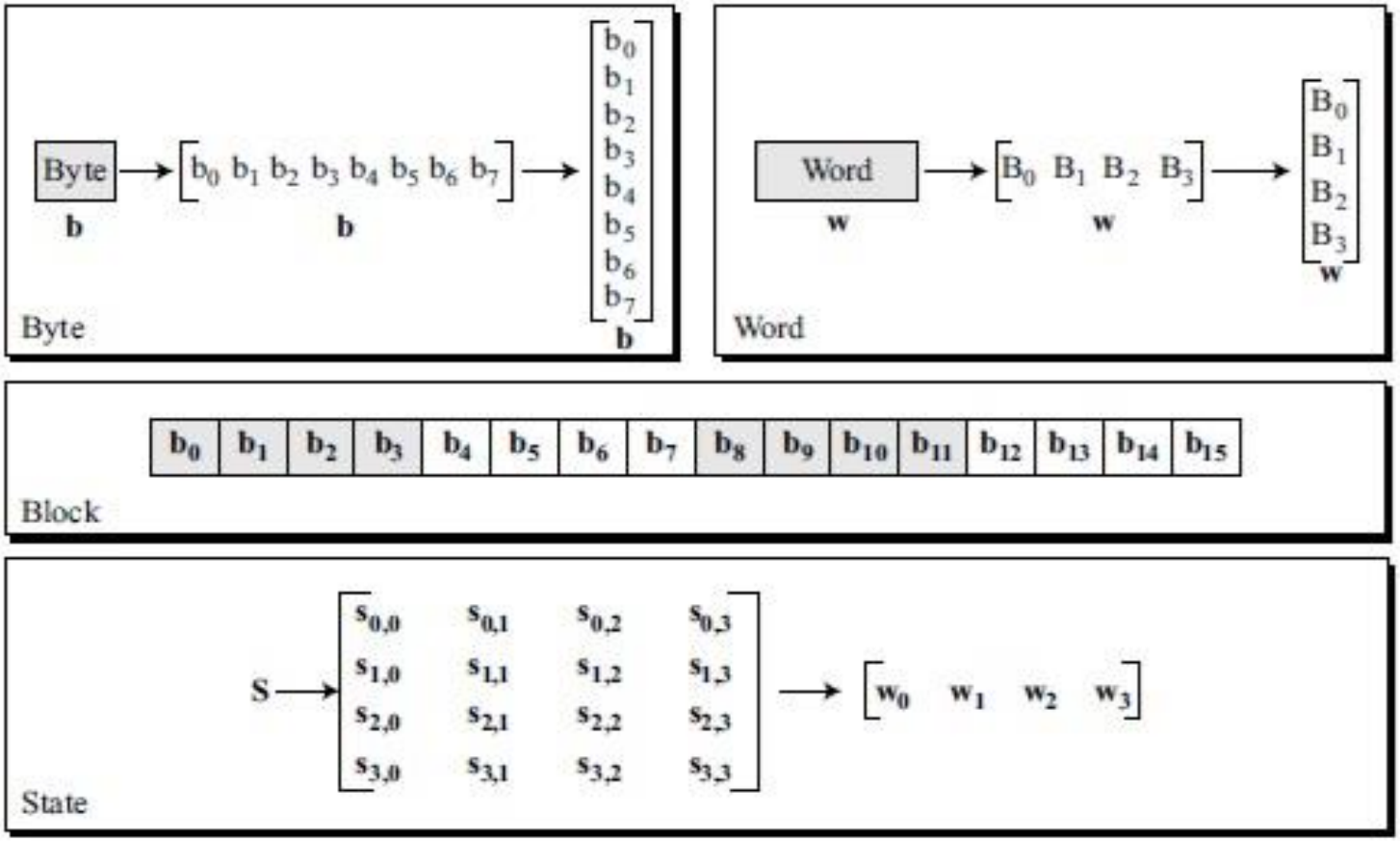
General Design of AES Encryption



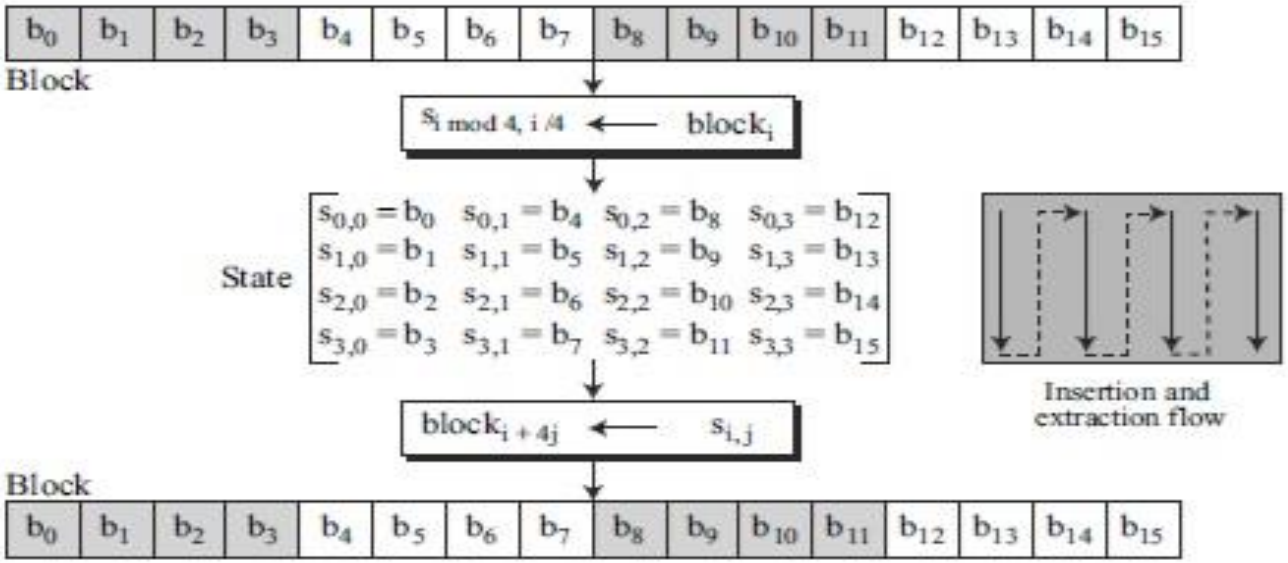
Nr	Key size
10	128
12	192
14	256

Relationship between number of rounds and cipher key size

Data Units Used in AES



Block to State Transformation



Example

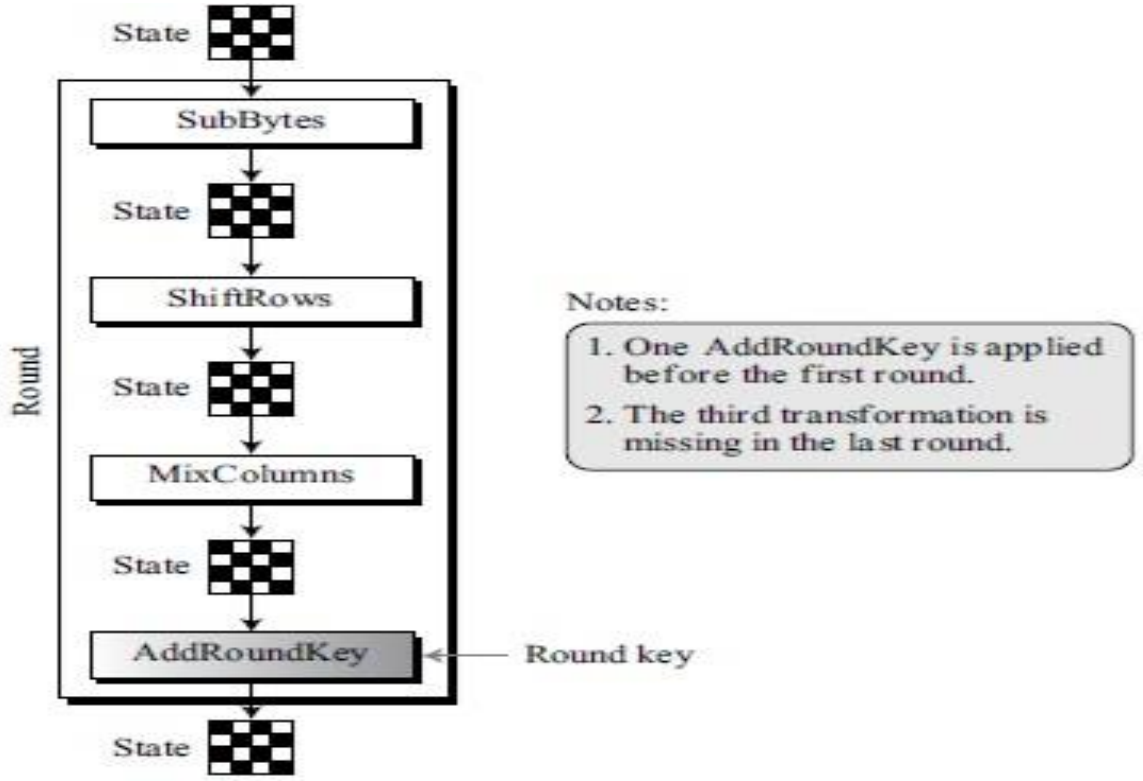
Let us see how a 16-character block can be shown as a 4 × 4 matrix. Assume that the text block is “AES uses a matrix”. We add two bogus characters at the end to get “AESUSESAMATRIXZZ”.

- Now we replace each character with an integer between 00 and 25. We then show each byte as an integer with two hexadecimal digits. For example, the character “S” is first changed to 18 and then written as 12 in hexadecimal. The state matrix is then filled up, column by column,

Text	A	E	S	U	S	E	S	A	M	A	T	R	I	X	Z	Z
Hexadecimal	00	04	12	14	12	04	12	00	0C	00	13	11	08	23	19	19

$$\begin{bmatrix} 00 & 12 & 0C & 08 \\ 04 & 04 & 00 & 23 \\ 12 & 12 & 13 & 19 \\ 14 & 00 & 11 & 19 \end{bmatrix}$$ State

Round Structure



At the decryption site inverse transformations are used :
InvSubByte, InvShiftRows,InvMixColumns and AddRoundKey

Substitution

AES, like DES, uses substitution. However, the mechanism is different.

- First, the substitution is done for each byte.
- Second, only one table is used for transformation of every byte, which means that if two bytes are the same, the transformation is also the same.
- Third, the transformation is defined by either a table lookup process or mathematical calculation in the GF(2^8) field. AES uses two invertible transformations.

SubByte Substitution

To substitute a byte, we interpret the byte as two hexadecimal digits. The left digit defines the row and the right digit defines the column of the substitution table. The two hexadecimal digits at the junction of the row and the column are the new byte

Transformation Table

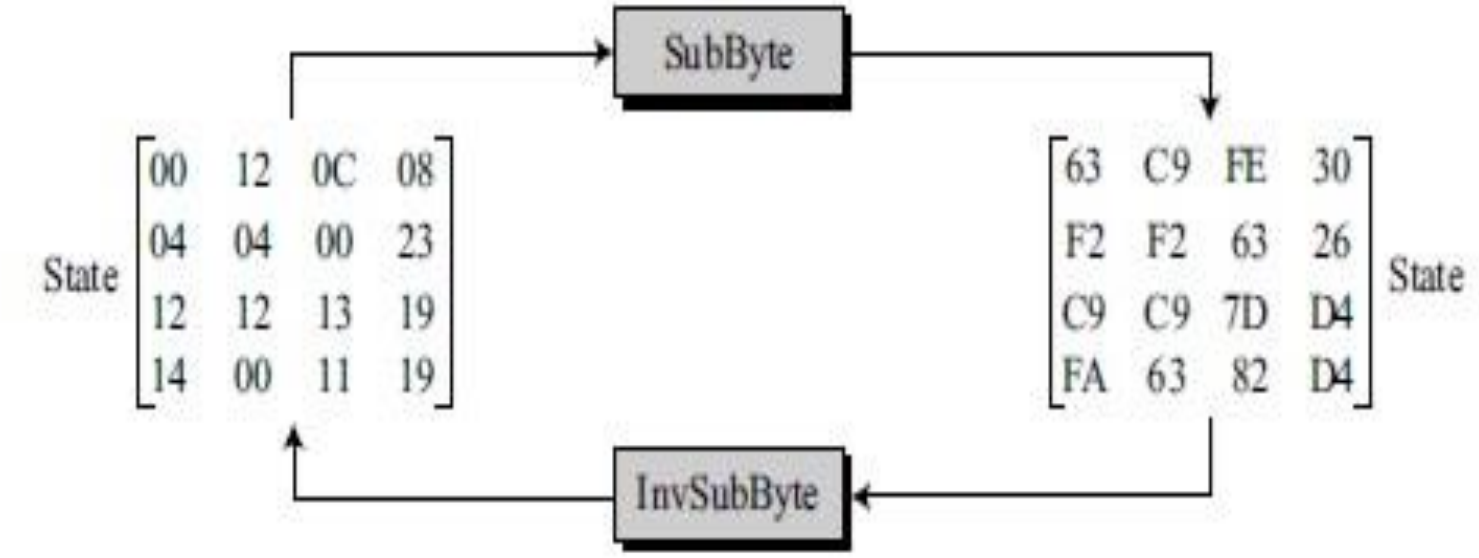
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	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

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
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	CB	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

InvSubBytes Table

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	52	09	6A	D5	30	36	A5	38	BF	40	A3	9E	81	F3	D7	FB
1	7C	E3	39	82	9B	2F	FF	87	34	8E	43	44	C4	DE	E9	CB
2	54	7B	94	32	A6	C2	23	3D	EE	4C	95	0B	42	FA	C3	4E
3	08	2E	A1	66	28	D9	24	B2	76	5B	A2	49	6D	8B	D1	25
4	72	F8	F6	64	86	68	98	16	D4	A4	5C	CC	5D	65	B6	92
5	6C	70	48	50	FD	ED	B9	DA	5E	15	46	57	A7	8D	9D	84
6	90	D8	AB	00	8C	BC	D3	0A	F7	E4	58	05	B8	B3	45	06
7	D0	2C	1E	8F	CA	3F	0F	02	C1	AF	BD	03	01	13	8A	6B
8	3A	91	11	41	4F	67	DC	EA	97	F2	CF	CE	F0	B4	E6	73
9	96	AC	74	22	E7	AD	35	85	E2	F9	37	E8	1C	75	DF	6E
A	47	F1	1A	71	1D	29	C5	89	6F	B7	62	0E	AA	18	BE	1B
B	FC	56	3E	4B	C6	D2	79	20	9A	DB	C0	FE	78	CD	5A	F4
C	1F	DD	A8	33	88	07	C7	31	B1	12	10	59	27	80	EC	5F
D	60	51	7F	A9	19	B5	4A	0D	2D	E5	7A	9F	93	C9	9C	EF
E	A0	E0	3B	4D	AE	2A	F5	B0	C8	EB	BB	3C	83	53	99	61
F	17	2B	04	7E	BA	77	D6	26	E1	69	14	63	55	21	0C	7D

Example


SubByte
State
00 12 0C 08
04 04 00 23
12 12 13 19
14 00 11 19
63 C9 FE 30
F2 F2 63 26
C9 C9 7D D4
FA 63 82 D4
State
InvSubByte

Transformation using GF(2^8)

- AES also defines the transformation algebraically using the GF(2^8) field with the irreducible polynomials ($x^8 + x^4 + x^3 + x + 1$),
- The SubBytes transformation repeats a routine, called subbyte, sixteen times.The InvSubBytes repeats a routine called invsubbyte. Each iteration transforms one byte.
- In the subbyte routine, the multiplicative inverse of the byte (as an 8-bit binary string) is found in GF(2^8) with the irreducible polynomial as the modulus.
- if the byte is $(00)_{16}$, its inverse is itself.

- The inverted byte is then interpreted as a column matrix with the least significant bit at the top and the most significant bit at the bottom.
- This column matrix is multiplied by a constant square matrix,X, and the result, which is a column matrix, is added with a constant column matrix, y, to give the new byte.

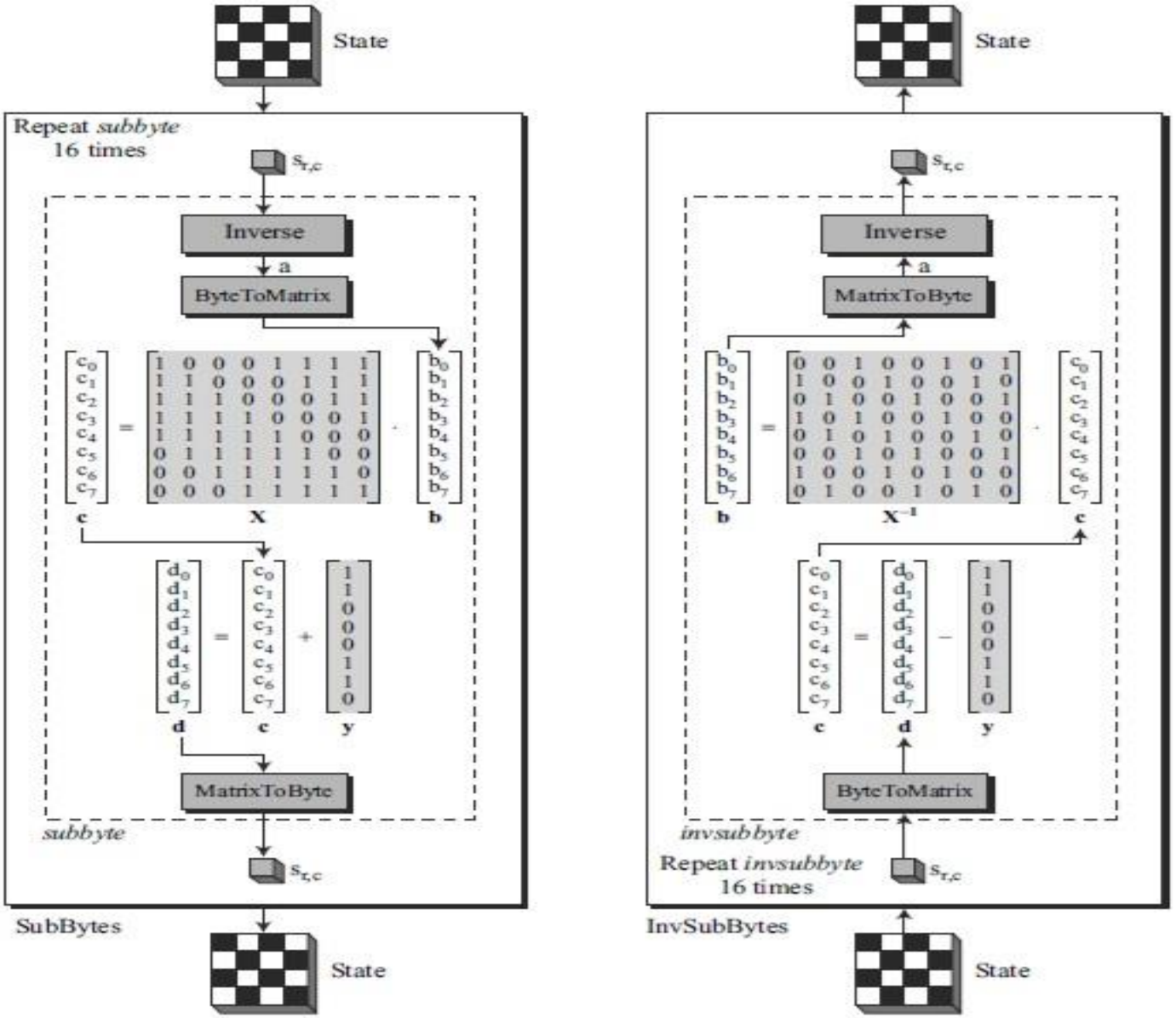
- Note that multiplication and addition of bits are done in GF(2).The invsubbyte is doing the same thing in reverse order.

- After finding the multiplicative inverse of the byte, the process is similar to the affine ciphers

- In the encryption, multiplication is first and addition is second.
- In the decryption, subtraction (addition by inverse) is first and division(multiplication by inverse) is second.
- It can be easily proved that two transformations are inverses of each other because addition or subtraction in GF(2) is actually the XOR operation.

$$\text{subbyte:} \quad \rightarrow \mathbf{d} = \mathbf{X}(s_{r,c})^{-1} \oplus \mathbf{y}$$

$$\text{invsubbyte:} \rightarrow [\mathbf{X}^{-1}(\mathbf{d} \oplus \mathbf{y})]^{-1} = [\mathbf{X}^{-1}(\mathbf{X}(s_{r,c})^{-1} \oplus \mathbf{y} \oplus \mathbf{y})]^{-1} = [(s_{r,c})^{-1}]^{-1} = s_{r,c}$$


State
Repeat subbyte
16 times
$s_{r,c}$
Inverse
a
ByteToMatrix
c
X
b
d
c
y
MatrixToByte
subbyte
$s_{r,c}$
SubBytes
State
State
$s_{r,c}$
Inverse
a
MatrixToByte
b
X^{-1}
c
c
d
y
ByteToMatrix
invsubbyte
Repeat invsubbyte
16 times
$s_{r,c}$
InvSubBytes
State

Example

Let us show how the byte 0C is transformed to FE by subbyte routine and transformed back to 0C by the invsubbyte routine

- subbyte:
  - The multiplicative inverse of 0C in GF(2^8) field is B0, which means b is (10110000).
  - Multiplying matrix X by this matrix results in c = (10011101)
  - The result of XOR operation is d = (11111110), which is FE in hexadecimal.

- invsubbyte:
  - The result of XOR operation is c = (10011101)
  - The result of multiplying by matrix X^{-1} is (11010000) or B0
  - The multiplicative inverse of B0 is 0C.

Algorithm

Pseudocode for SubBytes transformation

```
SubBytes (S)
{
   for (r = 0 to 3)
     for (c = 0 to 3)
           S_r,c = subbyte (S_r,c)
}

subbyte (byte)
{
   a ← byte^-1                 // Multiplicative inverse in GF(2^8) with inverse of 00 to be 00
    ByteToMatrix (a, b)
    for (i = 0 to 7)
    {
        c_i ← b_i ⊕ b_(i+4)mod 8 ⊕ b_(i+5)mod 8 ⊕ b_(i+6)mod 8 ⊕ b_(i+7)mod 8
        d_i ← c_i ⊕ ByteToMatrix (0x63)
    }
    MatrixToByte (d, d)
    byte ← d
}
```

- The ByteToMatrix routine transforms a byte to an 8×1 column matrix. The MatrixToByte routine transforms an 8×1 column matrix to a byte.

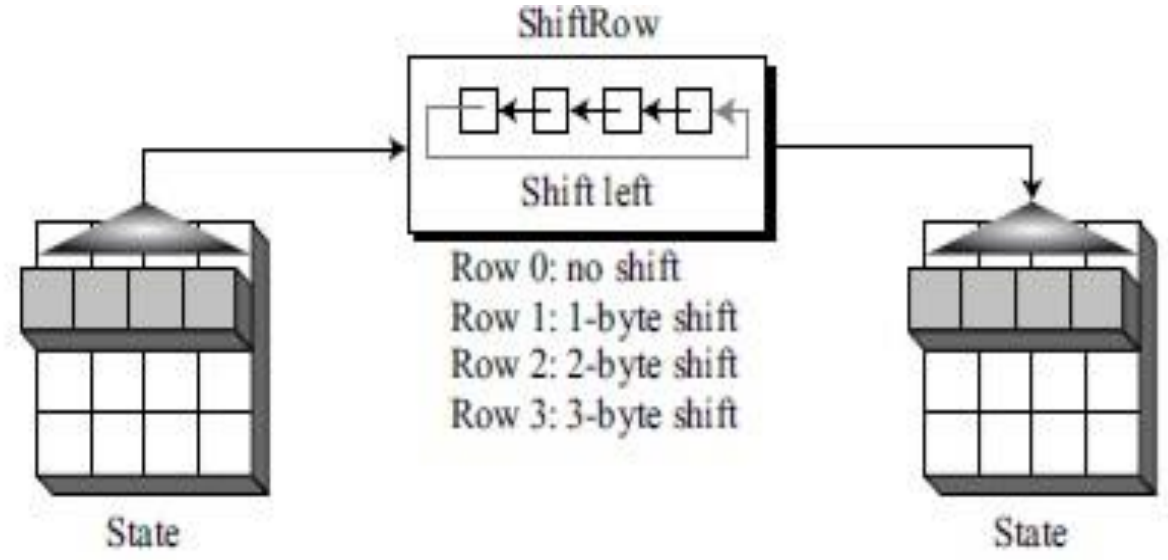
- The expansion of these routines and the algorithm for InvSubBytes are left as exercises.

- Although the multiplication and addition of matrices in the subbyte routine are an affine-type transformation and linear, the replacement of the byte by its multiplicative inverse in $GF(2^8)$ is nonlinear. This step makes the whole transformation nonlinear.

Permutation

- Another transformation found in a round is shifting, which permutes the bytes. Unlike DES, in which permutation is done at the bit level, shifting transformation in AES is done at the byte level; the order of the bits in the byte is not changed.
  - In the encryption, the transformation is called ShiftRows and the shifting is to the left.
  - The number of shifts depends on the row number (0, 1, 2, or 3) of the state matrix. This means the row 0 is not shifted at all and the last row is shifted three bytes.

ShiftRows Transformation



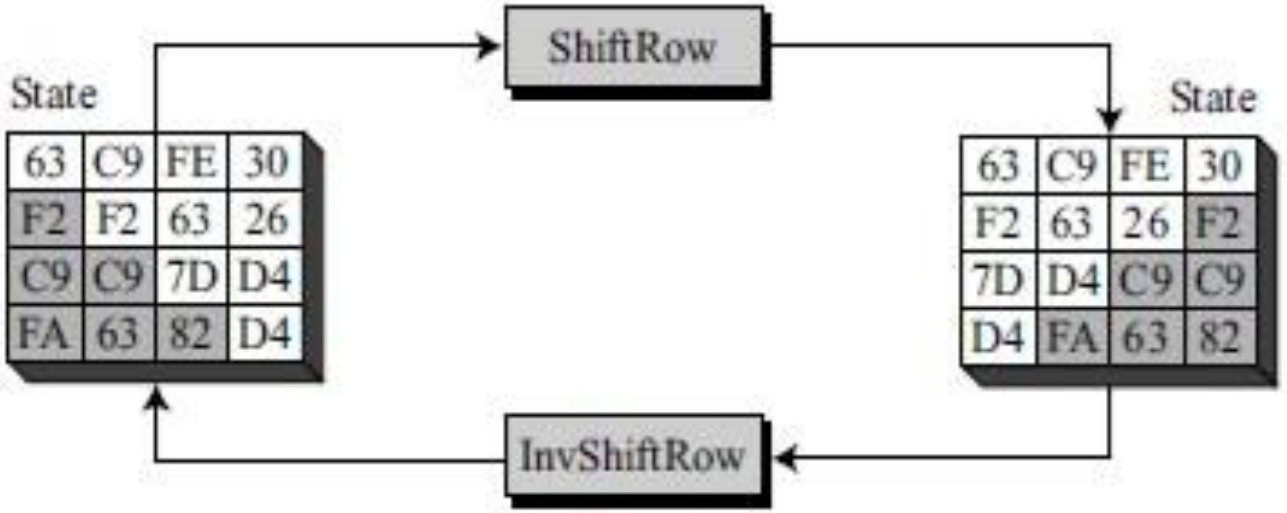
Algorithm

Pseudocode for ShiftRows transformation

```
ShiftRows (S)
{
    for (r = 1 to 3)
        shiftrow (s_r, r)                    // s_r is the rth row
}
```

```
shiftrow (row, n)                            // n is the number of bytes to be shifted
{
   CopyRow (row, t)                          // t is a temporary row
   for (c = 0 to 3)
        row_((c – n) mod 4) ← t_c
}
```

Example


State
63 C9 FE 30
F2 F2 63 26
C9 C9 7D D4
FA 63 82 D4
ShiftRow
State
63 C9 FE 30
F2 63 26 F2
7D D4 C9 C9
D4 FA 63 82
InvShiftRow

Mixing

- The substitution provided by the SubBytes transformation changes the value of the byte based only on original value and an entry in the table; the process does not include the neighboring bytes.

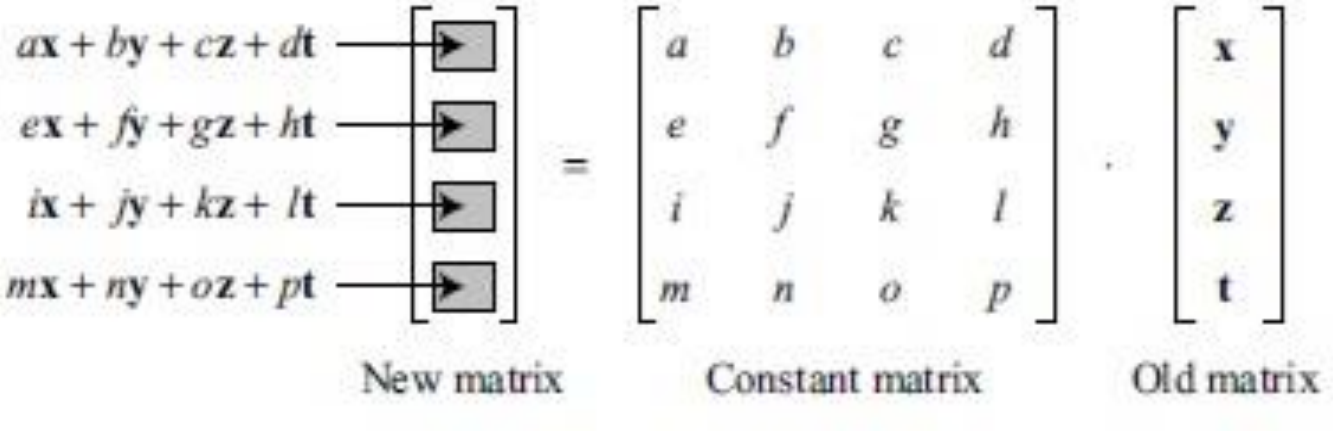
- The permutation provided by the ShiftRows transformation exchanges bytes without permuting the bits inside the bytes, i.e. ShiftRows is a byte-exchange transformation.
  - Need an interbyte transformation that changes the bits inside a byte, based on the bits inside the neighboring bytes.
  - Need to mix bytes to provide diffusion at the bit level.

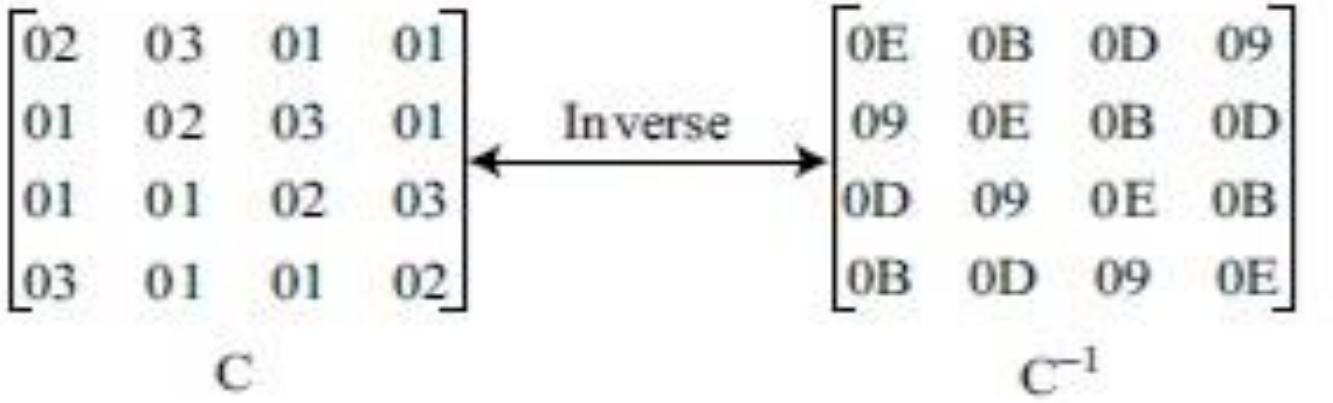
- The mixing transformation changes the contents of each byte by taking four bytes at a time and combining them to recreate four new bytes.

- To guarantee that each new byte is different (even if all four bytes are the same), the combination process first multiplies each byte with a different constant and then mixes them.

- The mixing can be provided by matrix multiplication

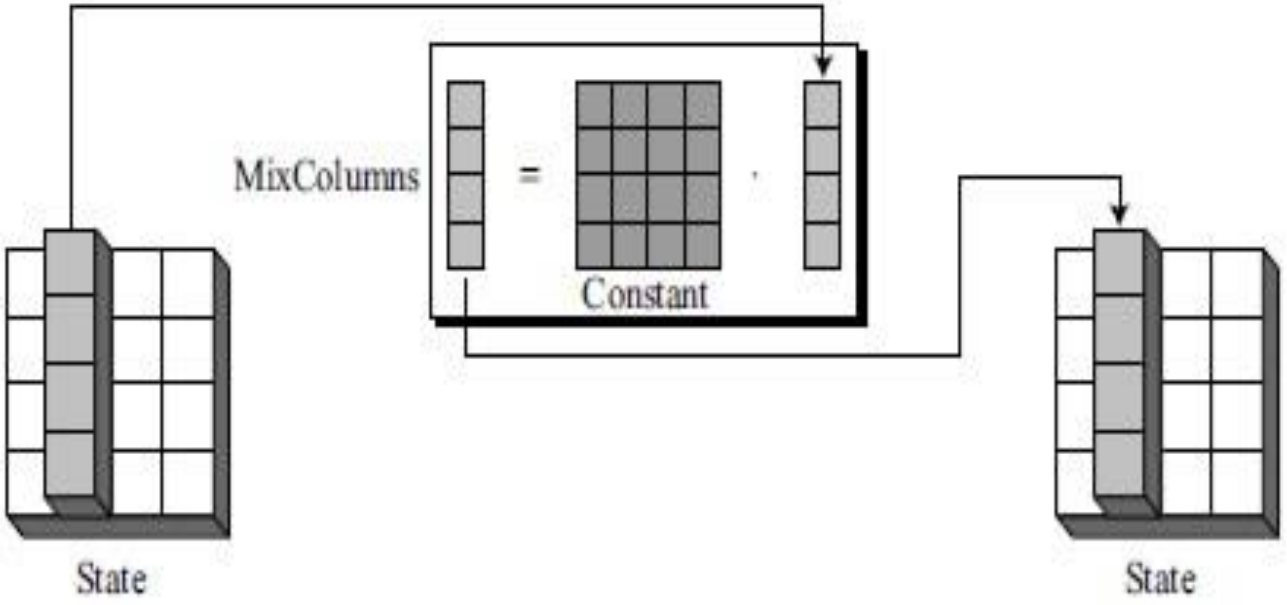
Mixing bytes using multiplication





MixColumns

- The MixColumns transformation operates at the column level; it transforms each column of the state to a new column.
- The transformation is actually the matrix multiplication of a state column by a constant square matrix.
- The bytes in the state column and constants matrix are interpreted as 8-bit words (or polynomials) with coefficients in GF(2).
- Multiplication of bytes is done in $GF(2^8)$ with modulus (10001101) or $(x^8 + x^4 + x^3 + x + 1)$.
- Addition is the same as XORing of 8-bit words.


MixColumns
=
Constant
State
State

InvMixColumns

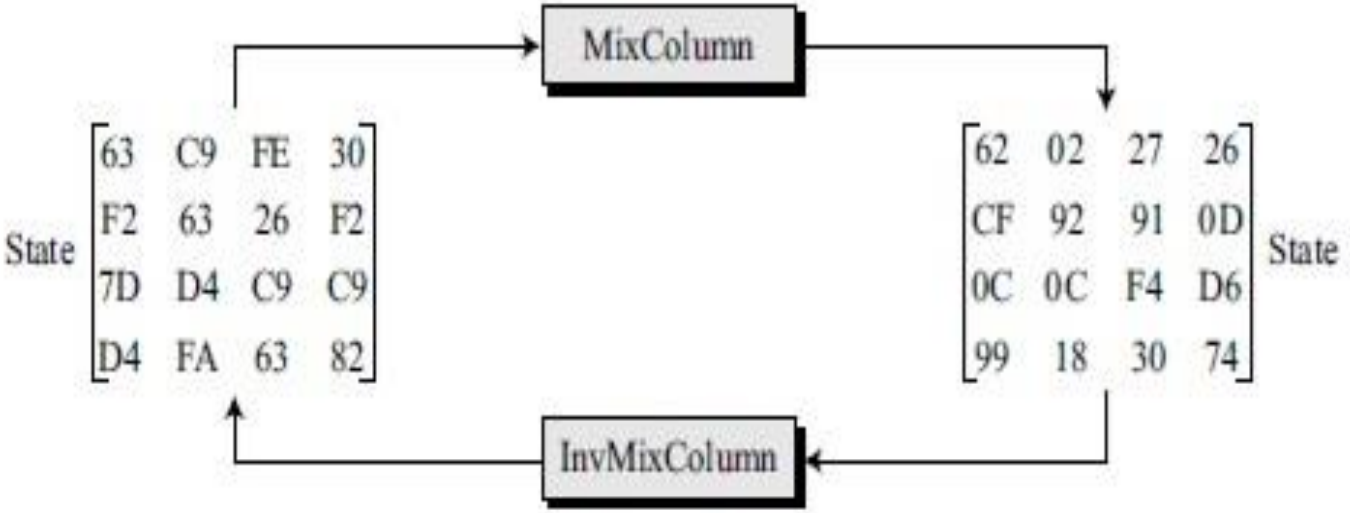
- The InvMixColumns transformation is basically the same as the MixColumns transformation. If the two constant matrices are inverses of each other, it is easy to prove that the two transformations are inverses of each other.

Algorithm

Pseudocode for MixColumns transformation

```
MixColumns (S)
{
    for (c = 0 to 3)
        mixcolumn (s_c)
}
mixcolumn (col)
{
   CopyColumn (col, t)                    // t is a temporary column

   col_0 ← (0x02) • t_0 ⊕ (0x03 • t_1) ⊕ t_2 ⊕ t_3

   col_1 ← t_0 ⊕ (0x02) • t_1 ⊕ (0x03) • t_2 ⊕ t_3

   col_2 ← t_0 ⊕ t_1 ⊕ (0x02) • t_2 ⊕ (0x03) • t_3

   col_3 ← (0x03 • t_0) ⊕ t_1 ⊕ t_2 ⊕ (0x02) • t_3
}
```

Example


MixColumn
State
63 C9 FE 30
F2 63 26 F2
7D D4 C9 C9
D4 FA 63 82
62 02 27 26
CF 92 91 0D
0C 0C F4 D6
99 18 30 74
State
InvMixColumn