Cryptography

Advanced Encryption Standard (AES)

M S Vilkhu

26 Oct 2024 (C1/C3/C5)

--07 Sep 24 (C1)

Learning Objectives

- Present an overview of the general structure of Advanced Encryption
- Standard (AES).
- Understand the four transformations used in AES.
- Explain the AES key expansion algorithm..

Introduction

- The Advanced Encryption Standard (AES) was published by the National Institute of Standards and Technology **(NIST) in 2001.**
- AES is a symmetric block cipher that is intended to **replace DES** as the approved standard for a wide range of applications.
- the structure of AES and most symmetric ciphers is quite complex and cannot be explained as easily as many other cryptographic algorithms.

Topics

- Finite Field Arithmetic
- AES Structure
  - AES transformation structure
  - Fix row transformation
  - Mixed row transformation
  - Addround transformation
- AES Key Expansion
- An AES Example
  - Results
  - Avalanche effect
- AES implementation
  - Equivalent Inverse Cipher
  - Implementation Aspects

Introduction

- The Advanced Encryption Standard (AES) was published by the National Institute of Standards and Technology (NIST) in 2001.
- AES is a symmetric block cipher replaced DES as the approved standard for a wide range of applications

Finite Field Arithmetic

- In AES, all operations are performed on **8-bit bytes**.
- In particular, the arithmetic operations of addition, multiplication, and division are performed over the **finite field** GF(2^8). (GF - Galois Field)

- In essence, a **field is a set** in which we can **do addition, subtraction, multiplication, and division without leaving the set**.
- Division is defined with the following rule: **a/b = a(b^{-1}).**
- An example of a **finite field -** one with a **finite number of elements**
- **finite field** is the set Z_p consisting of all the integers {0, 1, …, p - 1}, where **p is a prime number** and in which **arithmetic** is carried out **modulo p**.

Finite Field Arithmetic

- Virtually **all encryption algorithms**, both conventional and public-key, involve **arithmetic operations on integers**.
- If **one of the operations** used in the **algorithm is division**, then we need to work in arithmetic defined over a field; this is because **division** requires that **each nonzero element have a multiplicative inverse**.
- For **convenience and for implementation efficiency**, we would also like to work with **integers that fit exactly into a given number of bits**, with **no wasted bit patterns**.
- That is, we **wish to work with integers** in the range **0 through 2n - 1**, which fit into **an n-bit word**.
- Unfortunately, the set of such integers, $Z_2{}^n$, **using modular arithmetic**, is not a field.
- For example, the integer **2 has no multiplicative inverse** in $Z_2{}^n$, that is, there is **no integer b,** such that **2b mod 2n = 1**.

Multiplicative inverse

- The **multiplicative inverse** of a number is a value that, when multiplied by the original number, gives a product of **1**. In simpler terms, it's the number you multiply by to get 1.
- For example:
- The multiplicative inverse of 5 is $^1/_5$ because 5× $^1/_5$ =1.
- The multiplicative inverse of 2/3 is 3/2 because 2/3×3/2=1.
- In general, for a non-zero number x, the **multiplicative inverse is 1/x** .
- In **modular arithmetic**, particularly in fields like **finite fields** (used in cryptography, such as with **AES),** the multiplicative inverse of a number a under modulo m is a number b such that:

  a×b≡1 (mod m)
- This means that the **product of a and b**, when **divided by m**, leaves a **remainder of 1**.
- For example, in **mod 7**:
- The multiplicative inverse of 3 is 5 because 3×5=15 and 15 mod 7=1

Multiplicative Inverse in Modular Arithmetic:

- **Multiplicative Inverse in Modular Arithmetic:**
- In modular arithmetic, the concept of **multiplicative inverse is more involved**, and it only exists if the two numbers are **coprime** (they have **no common factors other than 1**). The multiplicative inverse of a modulo m is a number b such that:

  $a \times b = 1 \pmod{m}$
- This means when you multiply **a and b,** the result, divided by m, **leaves a remainder of 1**.
- Not all numbers have a multiplicative inverse in modular arithmetic unless they are **coprime** with the modulus m
- Example:
- Consider $a=3$ $m=7$. We want to find b such that: $3 \times b \equiv 1 \pmod{7}$ By testing different values of b, we find that: $3 \times 5 = 15 \equiv 1 \pmod{7}$, the multiplicative inverse of 3 modulo 7 is 5.
- If we try to find the inverse of **2 modulo 8,** it **doesn’t exist** because **2 and 8 are not coprime** (they share a common factor of 2).

Multiplicative Inverse in Finite Fields (GF):

- **Multiplicative Inverse in Finite Fields (GF):**
- In **finite fields**, such as **GF(2^n)** **used in cryptography** (e.g., AES encryption), finding the multiplicative inverse is **essential for operations like decryption**.
- In this context, **finding the multiplicative** inverse is akin to solving an equation in the field where the arithmetic operations are carried out modulo a prime or irreducible polynomial.
- For example, in **GF(2^8)**, the field contains 256 elements.
- Each element (other than zero) has a **unique multiplicative inverse in the field**.
- These operations are crucial in algorithms like AES for steps like the **S-box substitution**, where **elements are replaced with their inverses** in the finite field.

How to Calculate Multiplicative Inverse in Modular Arithmetic:

- **How to Calculate Multiplicative Inverse in Modular Arithmetic:**
- To find a multiplicative inverse in modular arithmetic, we can use the **Extended Euclidean Algorithm**. This algorithm not only finds the greatest common divisor (GCD) of two numbers but also expresses this GCD as a linear combination of the two numbers, which can then be used to find the inverse.
- **Example (using Extended Euclidean Algorithm):**
- Let's find the inverse of 7 mod 26.
- We need to solve: $7 \times x \equiv 1 \pmod{26}$
- Using the Extended Euclidean Algorithm, we get:
- $26 = 3 \times 7 + 5$
- $7 = 1 \times 5 + 2$
- $5 = 2 \times 2 + 1$
- $2 = 2 \times 1 + 0$
- work backwards:
- $1 = 5 - 2 \times 2$
- $1 = 5 - 2 \times (7 - 1 \times 5) = 3 \times 5 - 2 \times 7$
- $1 = 3 \times (26 - 3 \times 7) - 2 \times 7 = 3 \times 26 - 11 \times 7$
- Thus, $-11 \times 7 \equiv 1 \pmod{26} -11 \times 7 \equiv 1 \pmod{26}$, and the inverse of 7 mod 26 is $-11 \equiv 15 \pmod{26} -11 \equiv 15 \pmod{26}$.

- So, the multiplicative inverse of **7 modulo 26** is **15.**

Coprime

- **Coprime** (or **relatively prime**) refers to two numbers that have **no common factors other than 1**. In other words, two numbers are coprime if their **greatest common divisor (GCD)** is 1.
- **Example of Coprime Numbers:**
- **8 and 15** are coprime. The factors of 8 are 1,2,4,8 and the factors of 15 are 1,3,5,15. The only common factor is 1, so 8 and 15 are coprime.
- **9 and 28** are coprime because the factors of 9 are 1,3,9 and the factors of 28 are 1,2,4,7,14,28. The only common factor is 1.
- **Non-Coprime Example:**
- **12 and 18** are not coprime. The factors of 12 are 1,2,3,4,6,12 and the factors of 18 are 1,2,3,6,9,18 They share common factors of 2, 3, and 6, so their GCD is greater than 1 (it's 6).
- **How to Check if Two Numbers are Coprime:**
- To check if two numbers are coprime, find the **greatest common divisor (GCD)** of the two numbers:
- If GCD(a,b)=1 , the numbers a and b are coprime.
- If GCD(a,b)>1, they are **not** coprime.

Coprime in Modular Arithmetic and Cryptography:

- **Coprime in Modular Arithmetic and Cryptography:**
- Coprime numbers are particularly important in areas like **cryptography** (e.g., RSA encryption) and **modular arithmetic**. For example:
- In RSA, two numbers (usually large primes) are chosen such that they are coprime to a given modulus, ensuring secure encryption and decryption.
- For a number to have a **multiplicative inverse** in modular arithmetic, it **must be coprime** with the modulus.

- **Coprime in a Set:**
- You can also have multiple numbers be coprime with respect to each other. For example, the numbers 8,15,21 are pairwise coprime because any pair of numbers from this set has a GCD of 1.
- **Understanding coprime numbers is essential in number theory, encryption algorithms, and many applications in mathematics!**

Advanced Encryption Standard

- The more popular and widely adopted symmetric encryption algorithm likely to be encountered nowadays is the Advanced Encryption Standard (AES). It is found at least six time faster than triple DES.
- A replacement for DES was needed as its key size was too small. With increasing computing power, it was considered vulnerable against exhaustive key search attack. Triple DES was designed to overcome this drawback but it was found slow.

The features of AES are as follows −

- Symmetric key symmetric block cipher
- 128-bit data, 128/192/256-bit keys
- Stronger and faster than Triple-DES
- Provide full specification and design details
- Software implementable in C and Java

AES vs Feistel

AES (Advanced Encryption Standard) is an **iterative cipher** rather than a **Feistel cipher**.

- **Key Differences Between AES and Feistel Cipher:**

1. **Iterative Cipher (like AES):**
   1. In an **iterative cipher** like AES, each round of encryption involves applying a series of transformations (like substitution, permutation, and mixing) **to the entire block of data**.
   2. **AES** works on the entire block of data (128 bits) at each stage, applying several rounds (10, 12, or 14 rounds depending on the key size).
   3. Each round of AES includes **SubBytes (substitution), ShiftRows (permutation), MixColumns (mixing),** and **AddRoundKey (XOR with round key)** operations. Only the **last round** omits the **MixColumns** step.
   4. In AES, there is no distinction between the left and right halves of the data block. The entire block is processed in each round.

AES vs Feistel

AES (Advanced Encryption Standard) is an **iterative cipher** rather than a **Feistel cipher**.

- **Key Differences Between AES and Feistel Cipher:**

2. Feistel Cipher:

1. A **Feistel cipher** (used in encryption algorithms like DES) splits the input block into two halves (usually left and right), and then only one half of the block is processed during each round. The result is XORed with the other half.
2. A key characteristic of a Feistel cipher is that it only requires **half of the data to be modified in each round**. The unmodified half is swapped, and this process is repeated across multiple rounds.
3. Feistel networks can use the **same operations for encryption and decryption**, which is **not the case for non-Feistel ciphers like AES**.

AES vs Feistel

- **AES Structure:**
- **Block Size:** AES operates on a block size of 128 bits.
- **Key Size:** AES supports key sizes of 128, 192, or 256 bits.
- **Rounds:** AES consists of 10, 12, or 14 rounds depending on the key size.
- Each round is composed of:
  - **SubBytes:** A substitution step using an S-box.
  - **ShiftRows:** A permutation step where the rows of the block are shifted.
  - **MixColumns:** A mixing operation that mixes the bytes within each column.
  - **AddRoundKey:** A step where the block is XORed with the round key.

AES vs Feistel

- **Feistel Cipher Structure (like DES):**
- **Block Splitting:** Feistel ciphers split the block into two halves.
- **Rounds:** Each round typically involves:
  - Processing one half with a round function.
  - XORing the output of the round function with the other half.
  - Swapping the two halves for the next round.
- In Feistel ciphers, decryption is similar to encryption, but the order of the round keys is reversed. This property allows Feistel ciphers to be efficient for hardware and software implementations. However, **AES, being an iterative cipher,** uses different operations for encryption and decryption.

•AES is **iterative**, meaning each round processes the entire data block and transforms it using a set of defined operations.

•Feistel ciphers, like DES, work by processing half the data block at a time and swapping the halves between rounds.

AES Operation

- AES is an **iterative** rather than Feistel cipher.
- It is based on 'substitution–permutation network' (SPN).
- It comprises of a **series of linked operations**, some of which involve replacing inputs by specific outputs (substitutions) and others involve shuffling bits around (permutations).
- AES performs all its **computations on bytes** rather than bits.
- Hence, AES treats the **128 bits of a plaintext block as 16 bytes**. These **16 bytes** are arranged in **four columns** and **four rows (4 x 4)** for processing as a matrix
- Unlike DES, the **number of rounds in AES is variable** and **depends on the length** of the key.
  - AES uses 10 rounds for 128-bit keys,
  - 12 rounds for 192-bit keys and
  - 14 rounds for 256-bit keys.
- Each of these rounds **uses a different 128-bit round key**, which is **calculated from the original AES key.**

AES Operation

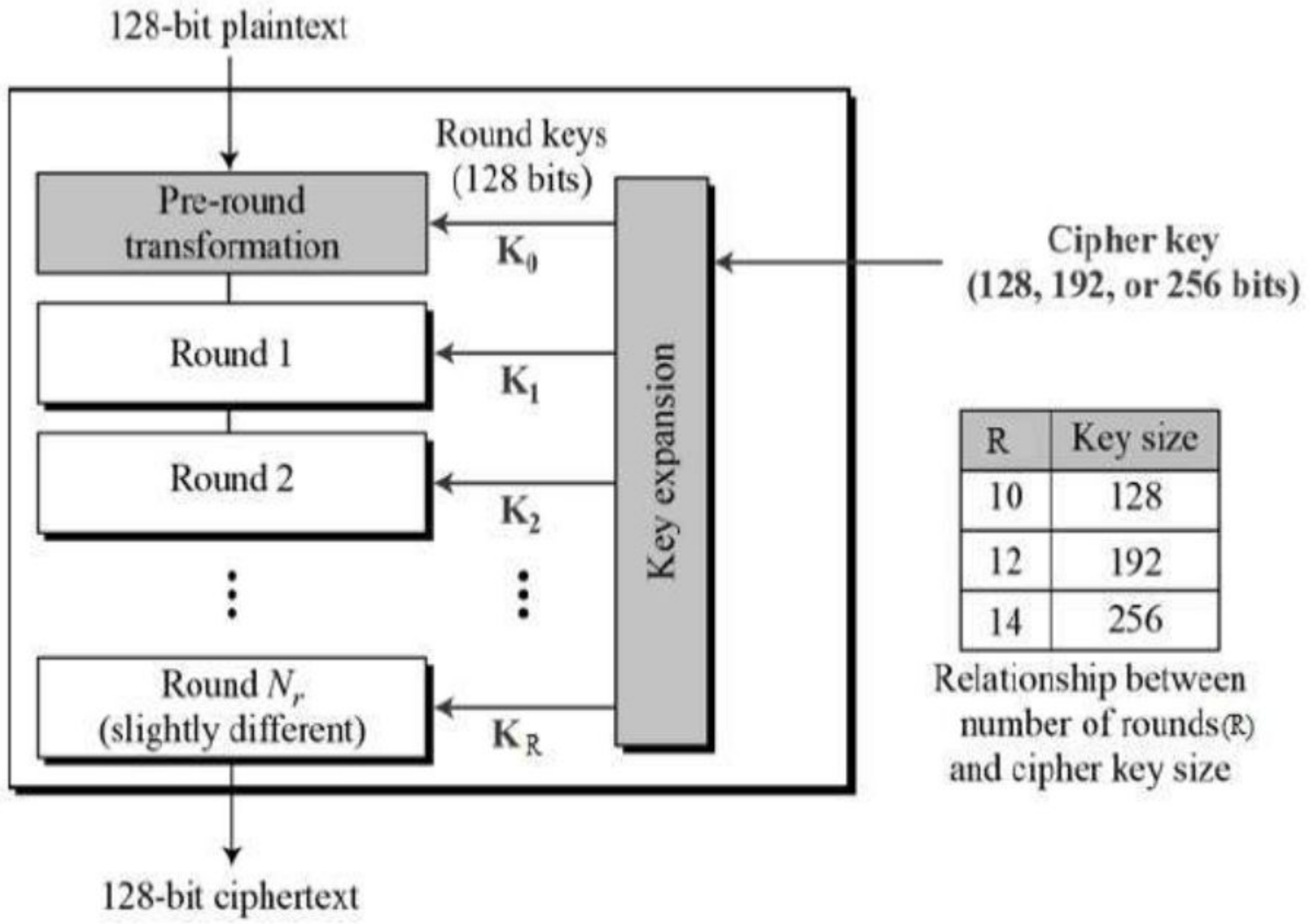
- Explanation
- AES treats the **128 bits of a plaintext block as 16 bytes**.
- [0x32, 0x88, 0x31, 0xe0, 0x43, 0x5a, 0x31, 0x37, 0xf6, 0x30, 0x98, 0x07, 0xa8, 0x8d, 0xa2, 0x34]

- This sequence is then organized into a **state matrix, Columns in the matrix are** formed from consecutive 4 bytes of the block. as follows

$$\begin{bmatrix} 0x32 & 0x43 & 0xf6 & 0xa8 \\ 0x88 & 0x5a & 0x30 & 0x8d \\ 0x31 & 0x31 & 0x98 & 0xa2 \\ 0xe0 & 0x37 & 0x07 & 0x34 \end{bmatrix}$$

AES Operation

- The schematic of AES structure is given in the illustration

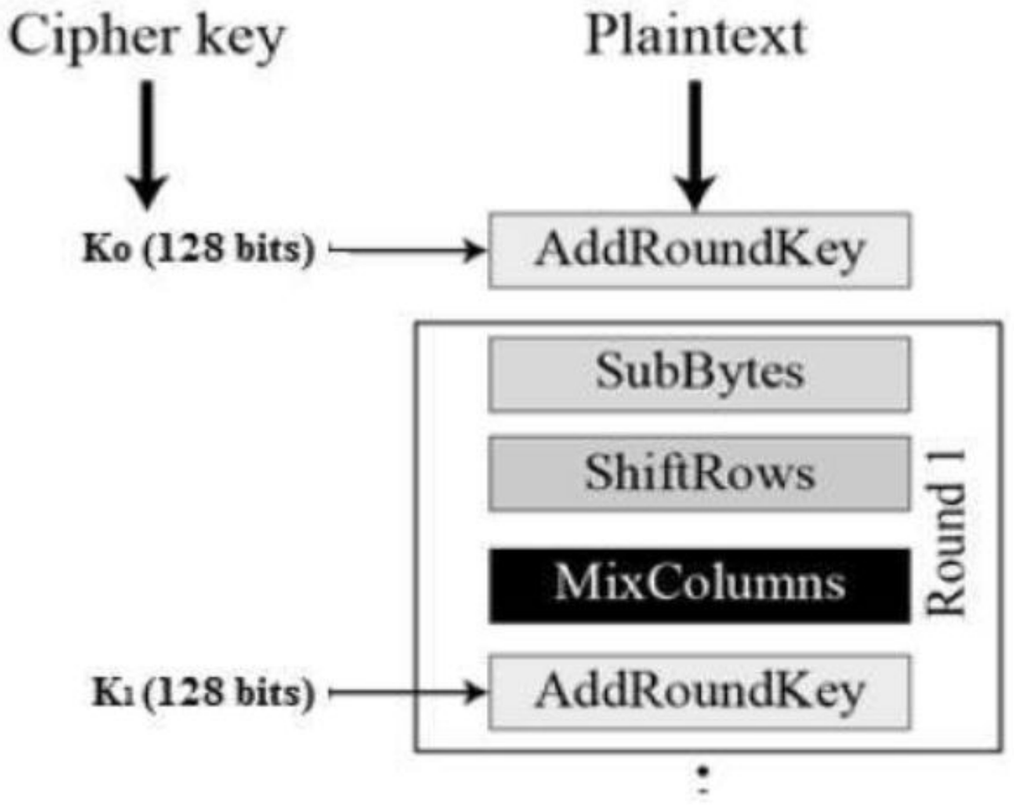


R	Key size
10	128
12	192
14	256

Relationship between number of rounds (R) and cipher key size

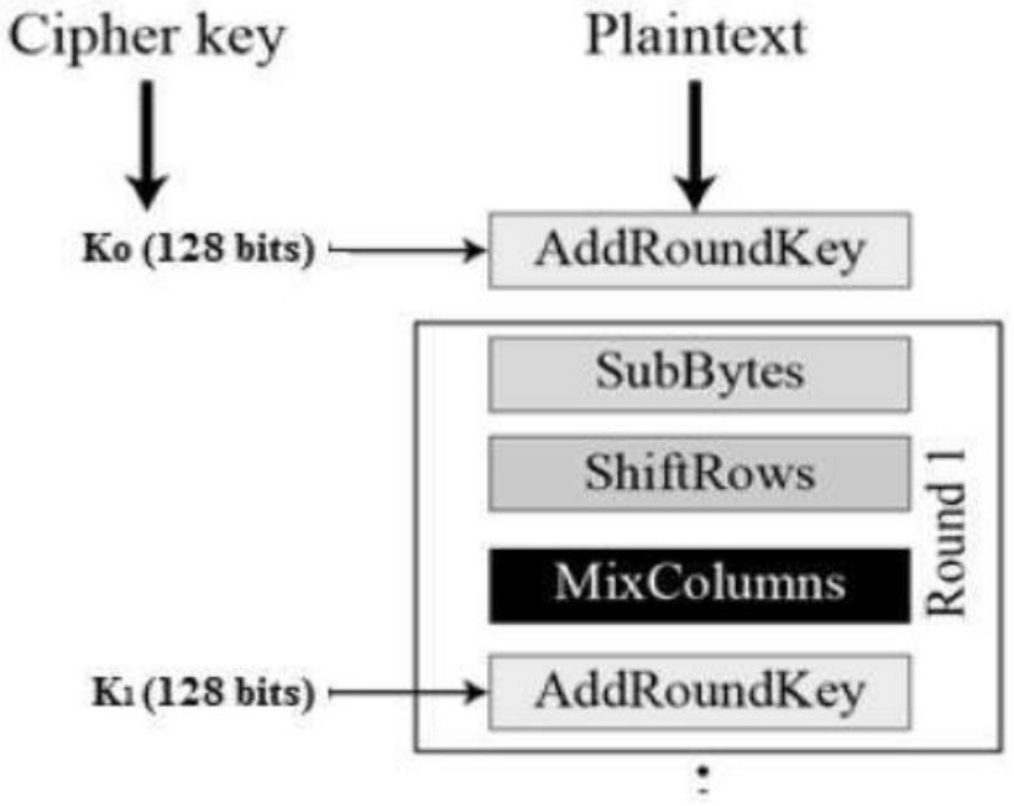
Encryption Process

- **Encryption Process**
- we restrict to description of a **typical round** of AES encryption.
- Each round comprise of **four sub-processes.**
  - **SubBytes**
  - **ShiftBytes**
  - **MixColumns**
  - **AddRoundKey**
- The first-round process is depicted in figure



Encryption Process

- **Encryption Process**
- How **SubByte** is implemented



Encryption Process

- **How SubBytes Works:**
- The **SubBytes** step in **AES (Advanced Encryption Standard)** is a **non-linear substitution** step where each byte in the **state matrix** is replaced by its corresponding value from a fixed **S-Box** (substitution box).
- This operation enhances the security of AES by introducing **confusion**, which makes the relationship between the key and the ciphertext complex.

- **SubBytes** uses a precomputed **S-Box**, which contains a mapping of each byte value (0x00 to 0xFF) to another unique byte.
- Each byte of the **state matrix** is **substituted independently** based on this S-Box lookup.

Encryption Process

- **How SubBytes Works:**
- **AES State Matrix:**
- AES works on blocks of 128 bits, which are arranged into a 4x4 **state matrix** of bytes. Let's assume a sample **state matrix** (with hexadecimal values):
- **State Matrix** (Before SubBytes):

[19 a0 9a e9]

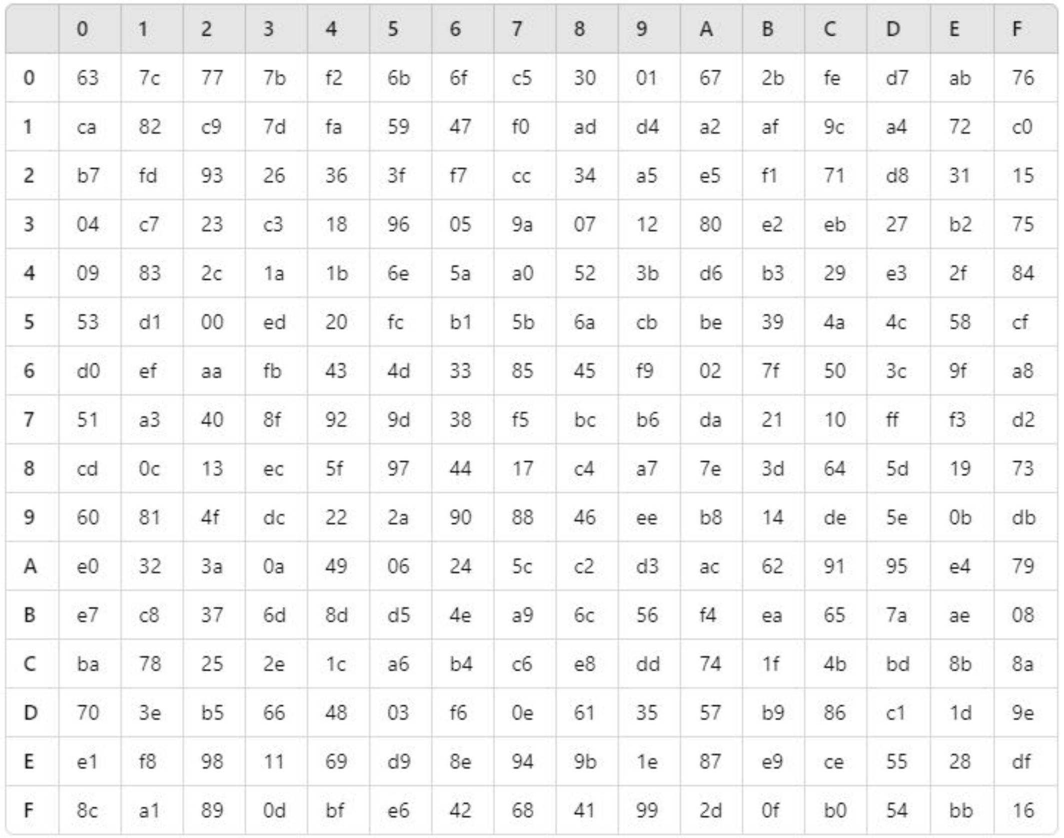
[3d f4 c6 f8]

[e3 e2 8d 48]

[be 2b 2a 08]

Encryption Process

- **How SubBytes Works:**
- **The AES S-Box:**
- The **S-Box** is a 16x16 table of byte substitutions.
- small sample of the S-Box (values in hexadecimal):



	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

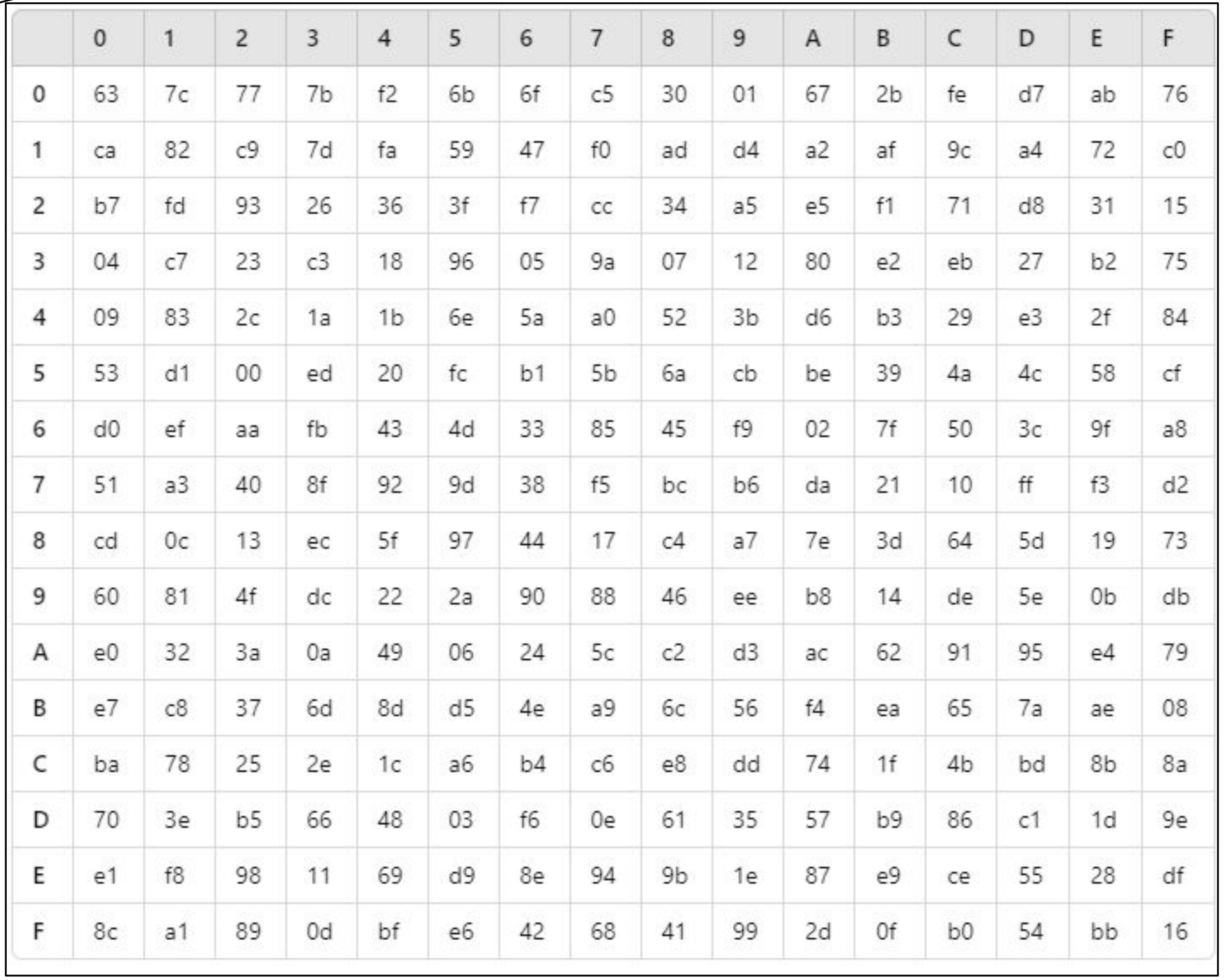
Encryption Process

Full S-Box

19

- **How SubBytes Works:**
- **SubBytes Substitution Example:**
- Using the **S-Box**, we will replace each byte of the state matrix with its corresponding value from the S-Box. Let’s walk through the substitution of one byte as an example:

1. **Substitute byte 19:**
    1. The byte "19" is in hexadecimal. Look up row 1 (first hex digit) and column 9 (second hex digit) in the S-Box.
    2. In the S-Box, the value at (1, 9) is "d4".
    3. So, **19** ⟶ **d4**.



	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

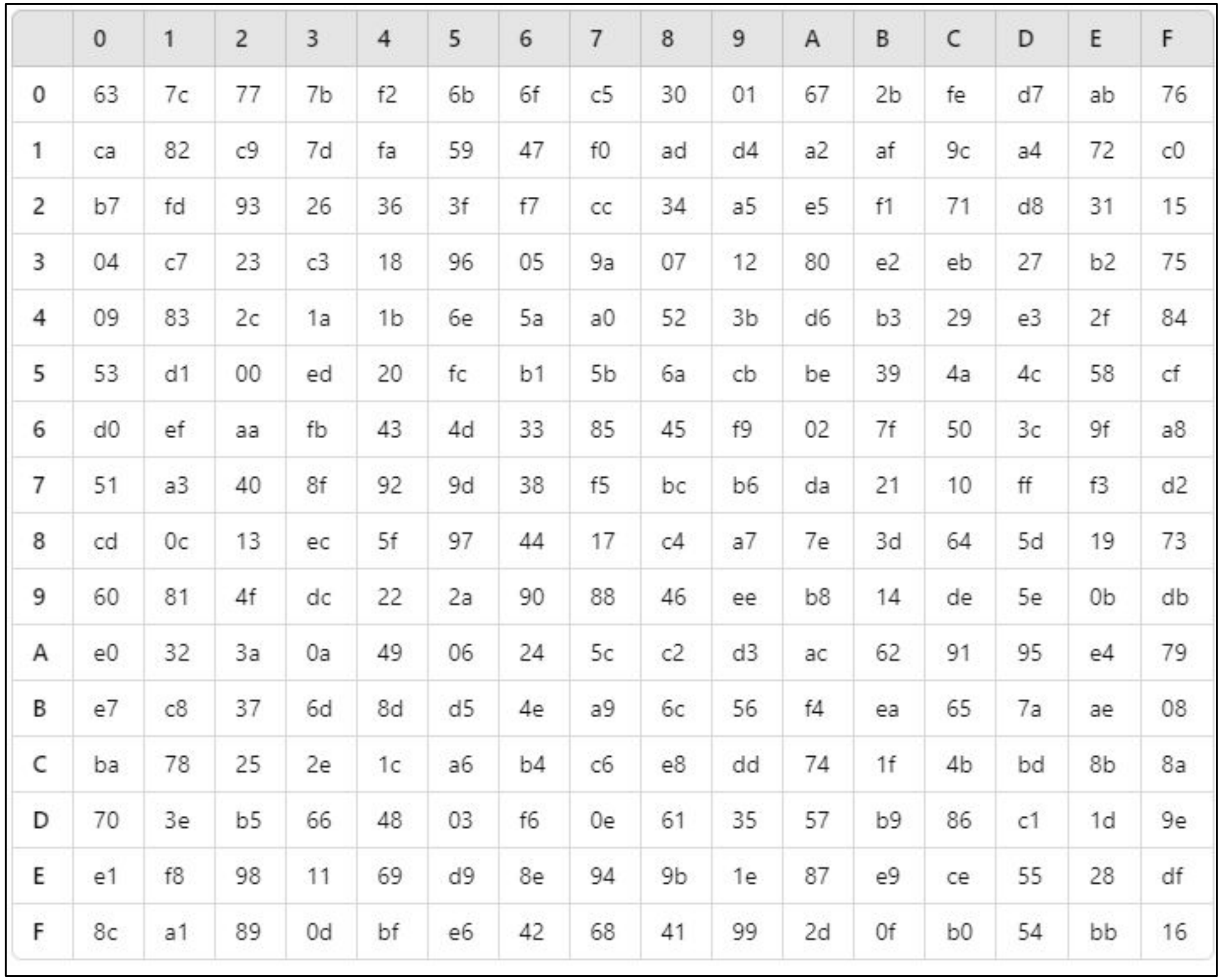
Encryption Process

19

- **How SubBytes Works:**
- **SubBytes Substitution Example:**
- Using the **S-Box**, we will replace each byte of the state matrix with its corresponding value from the S-Box. Let's walk through the substitution of one byte as an example:

1. **Substitute byte 19:**
    1. The byte "19" is in hexadecimal. Look up row 1 (first hex digit) and column 9 (second hex digit) in the S-Box.
    2. In the S-Box, the value at (1, 9) is "d4".
    3. So, **19** → **d4**.
    4. Now applying substitution to each byte in **state matrix**

Full S-Box



	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

```
[ 19  a0  9a  e9 ]
[ 3d  f4  c6  f8 ]
[ e3  e2  8d  48 ]
[ be  2b  2a  08 ]
```

$\longrightarrow$

Updated State Matrix:

```
d4 09 2f 74
27 36 d0 bb
e5 b5 66 2c
2c 42 6a 0d
```

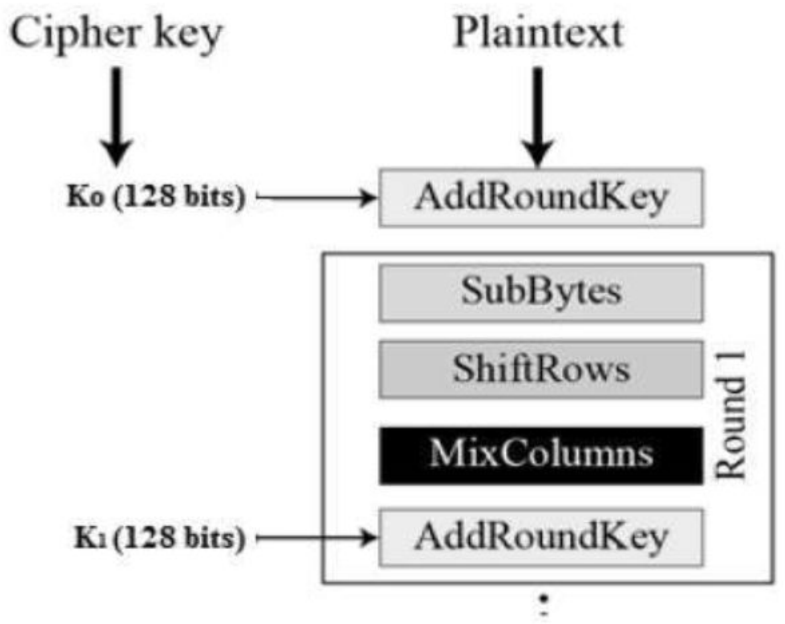
Encryption Process

- **How SubBytes Works:**
- **Why SubBytes Is Important:**
- The **S-Box** substitution introduces **non-linearity** and **confusion** into the encryption process, which helps to **thwart linear and differential cryptanalysis attacks**.
- Every byte of the state is transformed independently, ensuring that even a small change in the input leads to substantial changes in the output.

- **Byte Substitution (SubBytes)**
- The **16 input bytes** are **substituted by looking up** a fixed **table (S-box)** given in design.
- The result is in a matrix of **four rows and four columns (4x4)**.

Encryption Process

- **How Shiftrows is implemented**
- **Each of the four rows** of the **matrix is shifted to the left**.
- Any **entries that 'fall off'** are **re-inserted on the right side** of row. Shift is carried out as follows
    - First row is not shifted.
    - Second row is shifted one (byte) position to the left.
    - Third row is shifted two positions to the left.
    - Fourth row is shifted three positions to the left.
    - The **result is a new matrix** consisting of the same 16 bytes but shifted with respect to each other.



Encryption Process

- **ShiftRows**
- The **ShiftRows** step in **AES (Advanced Encryption Standard)** is a **permutation** operation that shifts the rows of the state matrix (which is a 4x4 array of bytes) by a certain number of positions. This operation is applied to the **state matrix** after the **SubBytes** step and before the **MixColumns** step in each round of AES.
- Let's assume a sample 4x4 state matrix filled with hexadecimal byte values:
- State Matrix (Before ShiftRows):

[a0 a1 a2 a3]

[b0 b1 b2 b3]

[c0 c1 c2 c3]

[d0 d1 d2 d3]

Encryption Process

- **Shiftrows**

How ShiftRows Works:

- **Row 0**: No shift (remains unchanged).
- **Row 1**: Each byte is shifted **1 position to the left**.
- **Row 2**: Each byte is shifted **2 positions to the left**.
- **Row 3**: Each byte is shifted **3 positions to the left**.

ShiftRows Example:

- Let's apply the **ShiftRows** transformation to the above state matrix.

1. **Row 0**: No shift
    1. [a0 a1 a2 a3] → [a0 a1 a2 a3]
2. **Row 1**: Shift 1 byte to the left
    1. [b0 b1 b2 b3] → [b1 b2 b3 b0]
3. **Row 2**: Shift 2 bytes to the left
    1. [c0 c1 c2 c3] → [c2 c3 c0 c1]
4. **Row 3**: Shift 3 bytes to the left
    1. [d0 d1 d2 d3] → [d3 d0 d1 d2]

[a0 a1 a2 a3]
[b0 b1 b2 b3]
[c0 c1 c2 c3]
[d0 d1 d2 d3]

$\longrightarrow$

State Matrix (After ShiftRows):
[a0 a1 a2 a3]
[b1 b2 b3 b0]
[c2 c3 c0 c1]
[d3 d0 d1 d2]

Encryption Process

- **Shiftrows**
- **Why ShiftRows Matters:**

Why ShiftRows Matters:

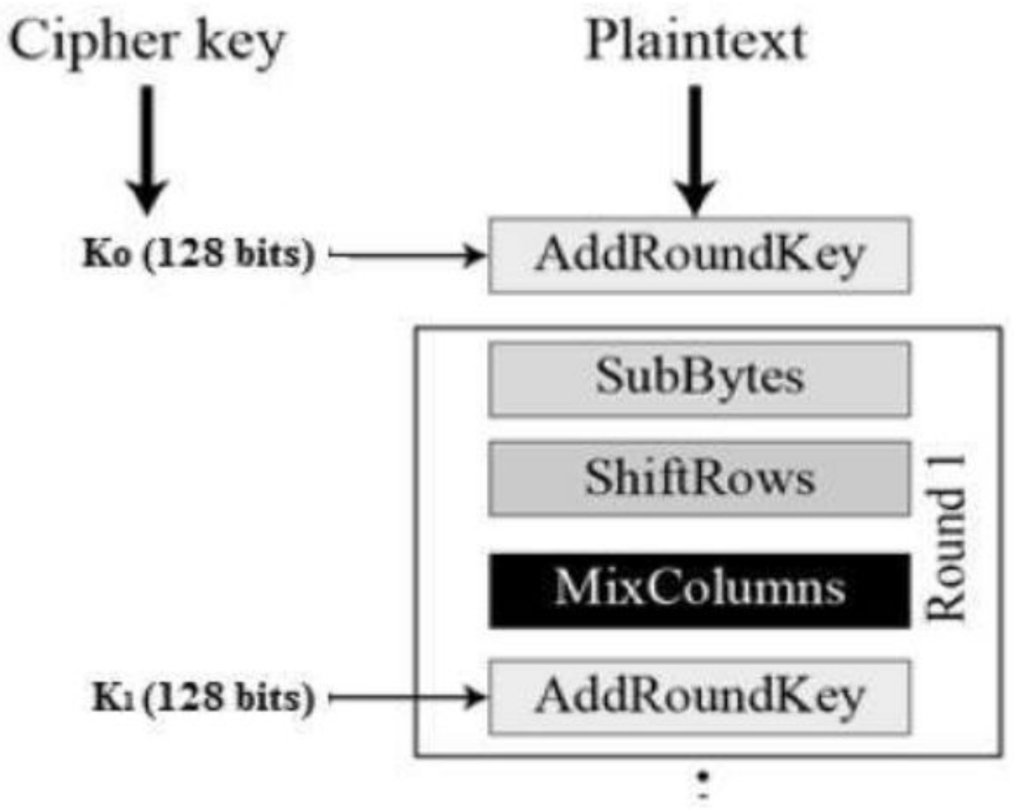
The **ShiftRows** step ensures that the **columns of the state matrix are mixed** up in a way that **contributes to diffusion**.

This helps to spread the **influence of each byte** of the input across multiple bytes in the output, which enhances the security of AES.

Encryption Process

- **Encryption Process**
- Here, we restrict to description of a typical round of AES encryption. Each round comprise of four sub-processes. The first-round process is depicted in figure

- SubBytes
- ShiftRows
- **MixColumns**
- AddRoundKey



Encryption Process

- **MixColumns**
- Each column of four bytes is now **transformed using a special mathematical function**.
- This function **takes as input the four bytes of one column** and **outputs four completely new bytes**, which replace the original column.
- The result is another **new matrix consisting of 16 new bytes**.
- It should be noted that **this step is not performed in the last round**.
- The **purpose** of MixColumns is to provide **diffusion**, which spreads the influence of each input byte over the four output bytes, making the relationship between the plaintext and ciphertext more complex.

Encryption Process

- **MixColumns**
- **How MixColumns Works**
- **State matrix**: AES operates on a 4x4 matrix of bytes (the "state"). After the **SubBytes** and **ShiftRows** transformations, the **MixColumns** operation is applied to each **column** of this matrix.
- **Multiplication in GF(2^8)**: Each **byte in the column** is treated as a polynomial, and the column is **multiplied by a fixed polynomial matrix** in the **finite field GF(2^8)** (Galois Field).
- This operation ensures that each **output byte depends on all four input bytes in the column**.

Encryption Process

- **MixColumns**
- **How MixColumns Works**
- MixColumns Transformation
- The standard matrix used for MixColumns is:

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix}$$

- Each column of the state matrix is multiplied by this constant matrix to get a new column in the state.

Encryption Process

- MixColumns
- **How MixColumns Works**
- **Example of MixColumns Operation**
- Suppose we have a single column from the state matrix:

$$\text{Column} = \begin{bmatrix} b0 \\ b1 \\ b2 \\ b3 \end{bmatrix} \qquad \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix}$$

- The resulting column after MixColumns is computed as follows:

$$\begin{bmatrix} b'0 \\ b'1 \\ b'2 \\ b'3 \end{bmatrix} = \begin{bmatrix} (2 \cdot b0) \oplus (3 \cdot b1) \oplus (1 \cdot b2) \oplus (1 \cdot b3) \\ (1 \cdot b0) \oplus (2 \cdot b1) \oplus (3 \cdot b2) \oplus (1 \cdot b3) \\ (1 \cdot b0) \oplus (1 \cdot b1) \oplus (2 \cdot b2) \oplus (3 \cdot b3) \\ (3 \cdot b0) \oplus (1 \cdot b1) \oplus (1 \cdot b2) \oplus (2 \cdot b3) \end{bmatrix}$$

Encryption Process

- MixColumns
- **How MixColumns Works**

- **Multiplication Rules in GF(2^8)**
- **Multiplication by 1**: The byte remains unchanged.
- **Multiplication by 2**: This is done by performing a left shift and, if the most significant bit is 1, XORing the result with **0x1B** (which represents the irreducible polynomial for AES).
- **Multiplication by 3**: This is equivalent to multiplying by 2 and then XORing the result with the original byte.

Encryption Process

- MixColumns
- **MixColumns**

- **Summary**
- **MixColumns** takes each column of the state matrix and performs a **matrix multiplication with a fixed polynomial matrix.**
- This operation ensures that the transformation spreads the influence of each byte across all four bytes in the column, achieving diffusion.
- The operation is performed over **GF(2^8)**, using predefined rules for multiplication by 1, 2, and 3.
- MixColumns is critical in AES for strengthening the cipher's security by mixing the data from different bytes in each column.

Encryption Process

- **AddRoundkey**
- The 16 bytes of the matrix are now considered as 128 bits and are XORed to the 128 bits of the round key.
- the 128 bits of **State** are **bitwise XORed** with the 128 bits of the round key
- If this is the **last round** then the **output is the ciphertext**. Otherwise, the resulting 128 bits are **interpreted as 16 bytes and we begin another similar round**.

State Matrix **Round Key**

$$\begin{bmatrix} 19 & a0 & 9a & e9 \\ 3d & f4 & c6 & f8 \\ e3 & e2 & 8d & 48 \\ be & 2b & 2a & 08 \end{bmatrix} \oplus \begin{bmatrix} a0 & 88 & 23 & 2a \\ fa & 54 & a3 & 6c \\ 19 & f8 & 44 & 95 \\ 12 & ba & d9 & 16 \end{bmatrix}$$

- perform the **XOR operation** between the state matrix and the round key. XOR is applied element-wise:

19 ⊕ a0 = b9

a0 ⊕ 88 = 28

9a ⊕ 23 = b9

e9 ⊕ 2a = c3

$$\longrightarrow \begin{bmatrix} b9 & 28 & b9 & c3 \\ c7 & a0 & 65 & 94 \\ fa & 1a & c9 & dd \\ ac & 91 & f3 & 1e \end{bmatrix}$$

Encryption Process

- **AddRoundkey**
- **Purpose of AddRoundKey:**
- **Security:** The XOR operation ensures that the data is mixed with the round key, providing a layer of security for each round.
- **Key Dependency:** It makes AES heavily dependent on the encryption key, ensuring that even small changes in the key or plaintext will result in completely different ciphertext due to the avalanche effect.

- AddRoundKey is an XOR operation between the state matrix and the round key.
- It is done at the beginning of the AES encryption process and at the end of each round.
- The operation ensures the key's influence in the transformation of the data, contributing to AES's overall strength against cryptanalysis.

Thank You