C | SecureAES
- High-Performance AES Encryption in C -

Ji Yong-Hyeon

Department of Information Security, Cryptology, and Mathematics
College of Science and Technology
Kookmin University

December 23, 2023

Acknowledgements

Note (XOR Operation and Modular Reduction in $GF(2^n)$). In the context of Galois Field $GF(2^n)$, particularly in binary polynomial arithmetic, the XOR operation is equivalent to addition and also plays a crucial role in modular reduction. We explore this equivalence through the principles of field theory and polynomial arithmetic.

- **Field Properties:**

  A Galois Field, $GF(p^n)$, is a finite field that contains a finite number of elements, where

  - p is a prime number (base of the field) and
  - n is a positive integer (degree of the field).

  For the binary field $GF(2^n)$, $p = 2$, which implies that every element in this field is either 0 or 1.

- **Addition in $GF(2^n)$:**

  In $GF(2^n)$, the addition of two elements is performed modulo 2. For any two elements $a, b \in GF(2^n)$, the addition is defined as:

  $$a + b = a \oplus b$$

  Since 2 is the base of the field, the addition wraps around upon reaching 2, which is effectively what the XOR operation does.

- **Polynomial Representation:**

  Elements in $GF(2^n)$ can be represented as polynomials where each coefficient is in $GF(2) = \{0, 1\}$. A general element can be written as:

  $$a(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1x + a_0$$

  where $a_i \in \{0, 1\}$ for all i.

- **Modular Reduction:**

  Modular reduction in $GF(2^n)$ involves reducing a polynomial by a fixed irreducible polynomial of degree n, ensuring that the result remains within the field. Let $m(x)$ be the irreducible polynomial. The reduction of a polynomial $f(x)$ is given by: $f(x) \mod m(x)$

- **XOR as Modular Reduction:**

  During modular reduction, the subtraction used in polynomial division becomes XOR, because subtraction and addition are the same in $GF(2)$. Therefore, reducing a polynomial $f(x)$ by $m(x)$ is effectively performed using XOR on the coefficients of corresponding terms.

  For example, if $f(x)$ has a term x^k where $k \geq n$, and $m(x)$ has a term x^k, then reducing $f(x)$ by $m(x)$ involves XORing the coefficients of x^k in $f(x)$ and $m(x)$, effectively eliminating the x^k term in $f(x)$.

In summary, the XOR operation becomes equivalent to both addition and modular reduction in $GF(2^n)$ due to the binary nature of the field. This equivalence simplifies polynomial arithmetic in binary fields, making it a cornerstone of operations in cryptographic algorithms.

Contents

Chapter 1

Block Cipher

Block ciphers are a fundamental component in cryptographic systems. They transform fixed-size blocks of plaintext into ciphertext using a symmetric key. The transformation is designed to be reversible only with knowledge of the key.

1.1 Definition and Structure

- **Secure Pseudo-Random Permutation (PRP) and Substitution Groups:**
  - **Definition:** A block cipher is considered a secure PRP if it is indistinguishable from a random permutation of the input bits, making it resistant to cryptanalysis.
  - **Substitution Groups:** Block ciphers often use substitution-permutation networks (SPNs) that include substitution groups. These groups perform non-linear transformations, crucial for creating cryptographic strength.
- **Confidentiality for Fixed n-bit Data (Blocks):**
  - **Fixed Block Size:** Block ciphers encrypt and decrypt data in fixed-size blocks (commonly 64 or 128 bits). This fixed size is crucial for the algorithm's structure and security.
  - **Padding Schemes:** When the data doesn't fit perfectly into a block, padding schemes are used to fill the remaining space, ensuring consistent block sizes.
- **Block Cipher Operation Modes for Variable-Length Data:**
  - **Mode of Operation:** To handle variable-length data, block ciphers use different modes of operation like CBC (Cipher Block Chaining), CFB (Cipher Feedback), and GCM (Galois/Counter Mode).
  - **Ensuring Security:** Each mode offers distinct features for security and efficiency, often enhancing the cipher's resistance to various attack vectors.
- **Advantages Over Asymmetric Key Cryptography:**
  - **High-Speed Computation:** Block ciphers are generally faster and require less computational power compared to asymmetric key cryptography.
  - **Suitability:** This makes them suitable for encrypting large volumes of data and in environments with limited resources.

- **Deriving Other Cryptographic Functions:**
  - **Versatility:** Block ciphers can be used to design other cryptographic functions like hash functions, message authentication codes (MACs), and random number generators.
  - **Construction Techniques:** Techniques like Cipher Block Chaining-MAC (CBC-MAC) and Counter mode (CTR) are examples of how block ciphers can be adapted for these purposes.

Block ciphers are a critical element in the cryptographic landscape, providing a versatile and efficient means for securing digital data. Their adaptability and robustness make them an indispensable tool in the design of secure communication protocols and cryptographic systems.

1.2 Modes of Operations

Table 1.1: Comparison of Modes

Mode	Integrity	Authentication	EncryptBlk	DecryptBlk	Padding	IV	$\|P\| \overset{?}{=} \|C\|$
ECB	O	X	O	O	O	X	$\|P\| < \|C\|$
CBC	O	X	O	O	O	O	$\|P\| < \|C\|$
OFB	O	X	O	X	X	O	$\|P\| = \|C\|$
CFB	O	X	O	X	X	O	$\|P\| = \|C\|$
CTR	O	X	O	X	X	O	$\|P\| = \|C\|$
CBC – CS	O	X	O	O	X	O	$\|P\| = \|C\|$

1.2.1 Padding

Block ciphers require input lengths to be a multiple of the block size. Padding is used to extend the last block of plaintext to the required length. Without proper padding, the encryption process may be insecure or infeasible.

There are several padding schemes used in practice, such as:

Table 1.2: Padding Standards in Block Ciphers

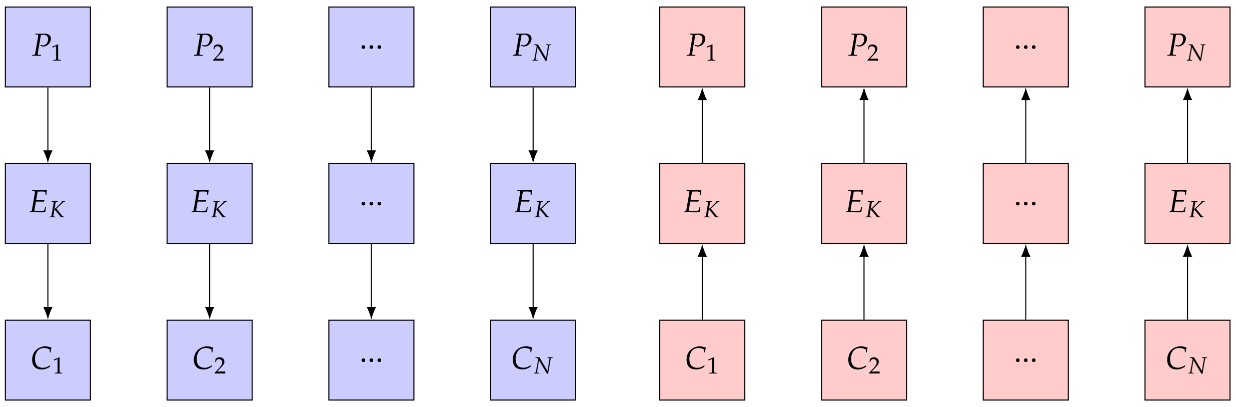
Standard Name	**Padding Method**
PKCS#7	Pad with bytes all the same value as the number of padding bytes `...dd | dd dd dd dd dd dd dd dd dd dd dd dd 04 04 04 04 |`
ANSI X9.23	Pad with zeros, last byte is the number of padding bytes `...dd | dd dd dd dd dd dd dd dd dd dd dd 00 00 00 00 05 |`
ISO/IEC 7816-4	First byte is ’80’ (hex), followed by zeros `...dd | dd dd dd dd dd dd dd dd dd dd 80 00 00 00 00 00 |`
ISO 10126	Pad with random bytes, last byte is the number of padding bytes `...dd | dd dd dd dd dd dd dd dd dd dd 2e 49 1b c1 aa 06 |`

1.2.2 ECB (Electronic CodeBook)

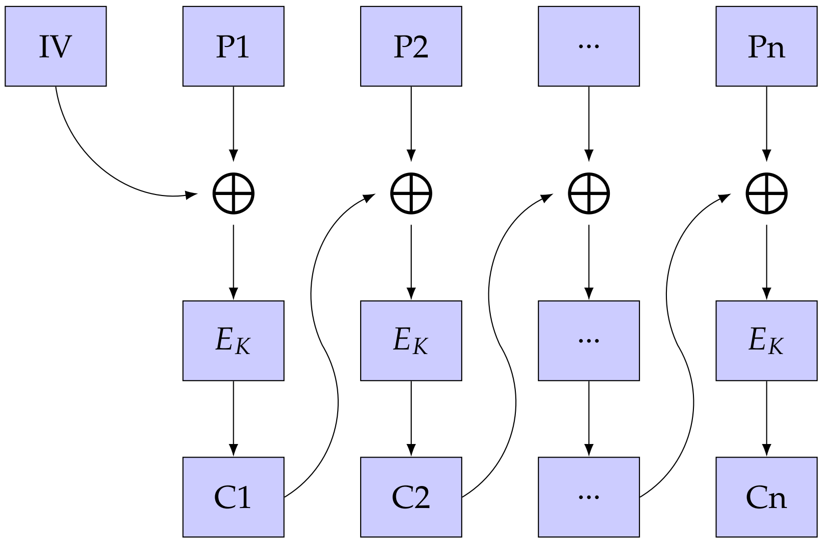
Algorithm 1: Electronic CodeBook

```
Input: $K$ and $P = P_1 \| \cdots \| P_N$ ($P_i \in \{0, 1\}^n$)
Output: $C = C_1 \| \cdots \| C_N$ ($C_i \in \{0, 1\}^n$)
1 for $i \leftarrow 1$ to $N$ do
2     $C_i \leftarrow$ EncryptBlk$(K, P_i)$;
3 end
4 return $C = C_1 \| \cdots \| C_N$;
```

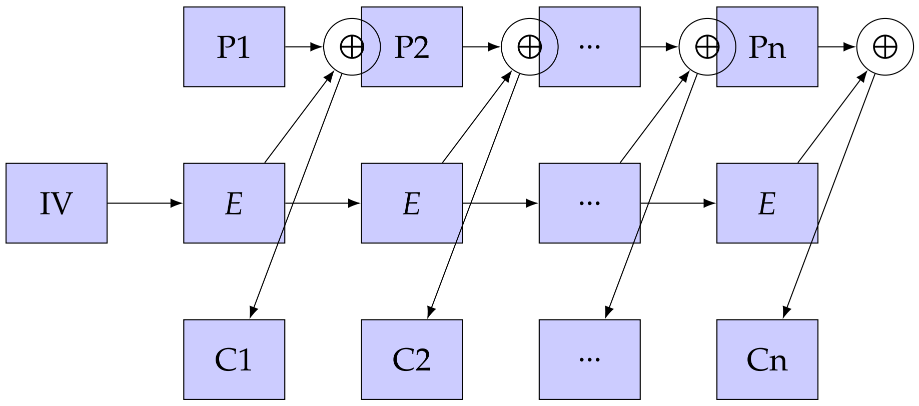
```
Input: $K$ and $C = C_1 \| \cdots \| C_N$ ($C_i \in \{0, 1\}^n$)
Output: $P = P_1 \| \cdots \| P_N$ ($P_i \in \{0, 1\}^n$)
1 for $i \leftarrow 1$ to $N$ do
2     $P_i \leftarrow$ DecryptBlk$(K, C_i)$;
3 end
4 return $C = C_1 \| \cdots \| C_N$;
```



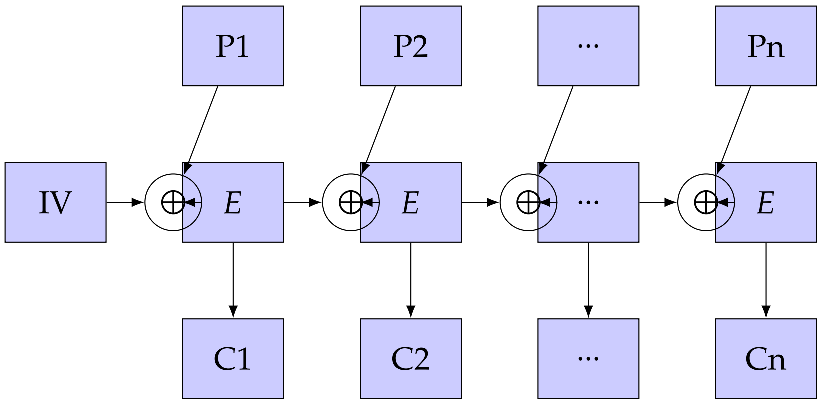
1.2.3 CBC (Cipher Block Chaining)



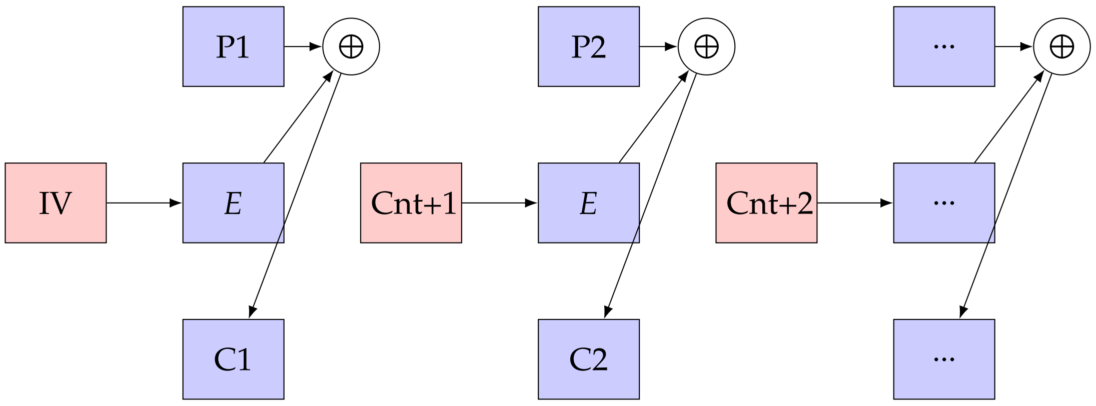
1.2.4 OFB **(Output FeedBack)**



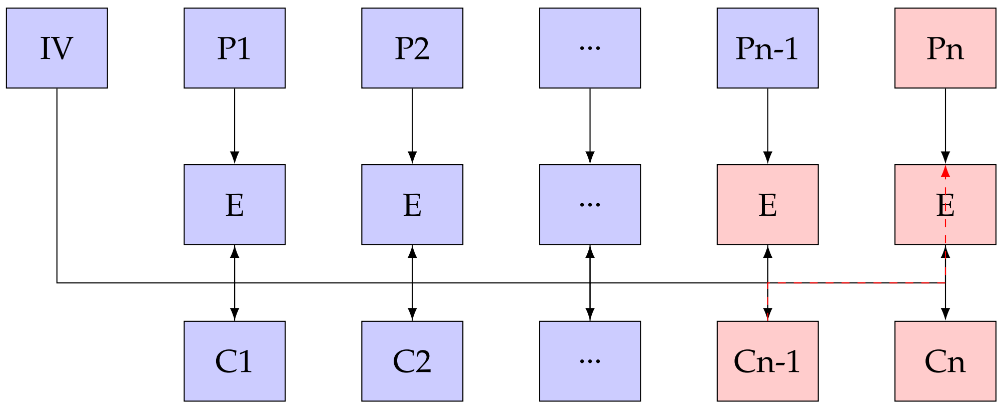
1.2.5 CFB **(Ciphertext FeedBack)**



1.2.6 CTR **(CounTeR)**



1.2.7 CBC – CS (Ciphertext Stealing)



Chapter 2

AES-128

2.1 Overview of AES-128

- KeyExpansion : $\{0, 1\}^{128} \longrightarrow \{0, 1\}^{1408=4\cdot(10+1)\cdot 32}$.
- AddRoundKey : $\{0, 1\}^{128} \times \{0, 1\}^{128} \longrightarrow \{0, 1\}^{128}$.
- SubBytes/ShiftRows/MixColumns : $\{0, 1\}^{128} \longrightarrow \{0, 1\}^{128}$.

Algorithm 2: Encryption of AES-128

```
Input: block src $\in \{0,1\}^{128}$, round-keys $\{rk_i\}_{i=0}^{11}$ ($rk_i \in \{0,1\}^{128}$)
Output: block dst $\in \{0,1\}^{128}$
$t \longleftarrow$ AddRoundKey(src, $rk_0$);
for $i \longleftarrow 1$ to 9 do
    $t \longleftarrow$ (MixColumns $\circ$ ShiftRows $\circ$ SubBytes)($t$);
    $t \longleftarrow$ AddRoundKey($t, rk_i$);
end
$t \longleftarrow$ (ShiftRows $\circ$ SubBytes)($t$);
$t \longleftarrow$ AddRoundKey($t, rk_{10}$);
dst $\longleftarrow t$;
return dst;
```

Algorithm 3: Decryption of AES-128

```
Input: block src $\in \{0,1\}^{128}$, round-keys $\{rk_i\}_{i=0}^{11}$ ($rk_i \in \{0,1\}^{128}$)
Output: block dst $\in \{0,1\}^{128}$
$t \longleftarrow$ AddRoundKey(src, $rk_{10}$);
for $i \longleftarrow 9$ to 1 do
    $t \longleftarrow$ (InvSubBytes $\circ$ InvShiftRows)($t$);
    $t \longleftarrow$ AddRoundKey($t, rk_i$);
    $t \longleftarrow$ InvMixColumns($t$);
end
$t \longleftarrow$ (InvShiftRows $\circ$ InvSubBytes)($t$);
$t \longleftarrow$ AddRoundKey($t, rk_0$);
dst $\longleftarrow t$;
return dst;
```

2.2 Functions and Constants used in AES

2.2.1 Key Expansion

- RotWord : $\{0, 1\}^{32} \to \{0, 1\}^{32}$ is defined by

$$\text{RotWord}\left(X_0 \parallel X_1 \parallel X_2 \parallel X_3\right) := X_1 \parallel X_2 \parallel X_3 \parallel X_0 \quad \text{for} \quad X_i \in \{0, 1\}^8 .$$

Code 2.1: RotWord rotates the input word left by one byte

```c
u32 RotWord(u32 word) {
    return (word << 0x08) | (word >> 0x18);
}
```

- SubWord : $\{0, 1\}^{32} \to \{0, 1\}^{32}$ is defined by

$$\text{SubWord}(X_0 \parallel X_1 \parallel X_2 \parallel X_3) := s(X_0) \parallel s(X_1) \parallel s(X_2) \parallel s(X_3) \quad \text{for} \quad X_i \in \{0, 1\}^8 .$$

Here, $s : \{0, 1\}^8 \to \{0, 1\}^8$ is the S-box.

Code 2.2: SubWord applies the S-box to each byte of the input word

```c
u32 SubWord(u32 word) {
    return (u32)s_box[word >> 0x18] << 0x18 |
        (u32)s_box[(word >> 0x10) & 0xFF] << 0x10 |
        (u32)s_box[(word >> 0x08) & 0xFF] << 0x08 |
        (u32)s_box[word & 0xFF];
}
```

- Round Constant `rCon`:

The constant $\texttt{rCon}_i \in \mathbb{F}_{2^8}$ used in generating the i-th round key corresponds to the value of x^{i-1} in the binary finite field $\mathbb{F}_{2^8}$ and is as follows:

i	1	2	3	4	5	6	7	8	9	10
$\texttt{Rcon}_i$	0x01	0x02	0x04	0x08	0x10	0x20	0x40	0x80	0x1b	0x36

Code 2.3: rCon Array Declaration

```c
static const u32 rCon[10] = {
    0x01000000, 0x02000000, 0x04000000, 0x08000000,
    0x10000000, 0x20000000, 0x40000000, 0x80000000,
    0x1b000000, 0x36000000
};
```

Algorithm 4: Key Schedule (AES-128)

```
Input: User key $uk = (uk_0, \ldots, uk_{15})$ ($uk_i \in \{0,1\}^8$);        // $uk \in \{0,1\}^{128}$ is 16-byte
Output: round-keys $\{rk_i\}_{i=0}^{43}$ ($rk_i \in \{0,1\}^{32}$);        // $\{rk_i\}_{i=0}^{43} \in \{0,1\}^{1408}$ is 176-byte
1  $rk_0 \leftarrow uk_0 \| uk_1 \| uk_2 \| uk_3$;
2  $rk_1 \leftarrow uk_4 \| uk_5 \| uk_6 \| uk_7$;
3  $rk_2 \leftarrow uk_8 \| uk_9 \| uk_{10} \| uk_{11}$;
4  $rk_3 \leftarrow uk_{12} \| uk_{13} \| uk_{14} \| uk_{15}$;
5  for $i = 4$ to $43$ do
6      $t \leftarrow rk_{i-1}$;
7      if $i \bmod 4 = 0$ then
           /* SubWord ∘ RotWord : $\{0,1\}^{32} \rightarrow \{0,1\}^{32}$ */
8          $t \leftarrow \mathrm{RotWord}(t)$;
9          $t \leftarrow \mathrm{SubWord}(t)$;
10         $t \leftarrow t \oplus (\mathrm{rCon}_{i/4} \| 0x00 \| 0x00 \| 0x00)$;
11     end
12     $rk_i \leftarrow rk_{i-4} \oplus_{32} t$;
13 end
```

Code 2.4: AES-128 Key Expansion

```c
void KeyExpansion(const u8* uKey, u32* rKey) {
    u32 temp;
    int i = 0;

    // Copy the input key to the first round key
    while (i < 4) {
        rKey[i] = (u32)uKey[4*i] << 0x18 |
        (u32)uKey[4*i+1] << 0x10 |
        (u32)uKey[4*i+2] << 0x08 |
        (u32)uKey[4*i+3];
        i++;
    }

    i = 4;

    // Generate the remaining round keys
    while (i < 44) {
        temp = rKey[i-1];
        if (i % 4 == 0) {
            temp = SubWord(RotWord(temp)) ^ rCon[i/4-1];
        }
        rKey[i] = rKey[i-4] ^ temp;
        i++;
    }
}
```

2.2.2 AddRoundKey

- AddRoundKey : $\{0,1\}^{128} \times \{0,1\}^{128} \longrightarrow \{0,1\}^{128}$ is defined by

$$\text{AddRoundKey}\left(\{X_i\}_{i=0}^{15}, \{rk_i\}_{i=0}^{3}\right) := \{X_i \oplus_8 uk_i\}_{i=0}^{15}.$$

Code 2.5: AES AddRoundKey

```c
void AddRoundKey(u8* state, const u32* rKey) {
    for (int i = 0; i < AES_BLOCK_SIZE; i++) {
        // i =  0,  1,  2,  3 => wordIndex = 0
        // i =  4,  5,  6,  7 => wordIndex = 1
        // i =  8,  9, 10, 11 => wordIndex = 2
        // i = 12, 13, 14, 15 => wordIndex = 3
        int wordIndex = i / 4;

        // i =  0,  1,  2,  3 => bytePosition = 0,  1,  2,  3
        // i =  4,  5,  6,  7 => bytePosition = 0,  1,  2,  3
        // i =  8,  9, 10, 11 => bytePosition = 0,  1,  2,  3
        // i = 12, 13, 14, 15 => bytePosition = 0,  1,  2,  3
        int bytePosition = i % 4;
/*
 * +-------+------------+--------------+----------------------+
 * | i     | wordIndex  | bytePosition | shiftedWord          |
 * +-------+------------+--------------+----------------------+
 * | 0-3   | 0          | 0            | rKey[0] >> 0x18      |
 * |       |            | 1            | rKey[0] >> 0x10      |
 * |       |            | 2            | rKey[0] >> 0x08      |
 * |       |            | 3            | rKey[0]              |
 * ------------------------------------------------------------
 * | 4-7   | 1          | 0            | rKey[1] >> 24        |
 * |       |            | 1            | rKey[1] >> 16        |
 * |       |            | 2            | rKey[1] >> 8         |
 * |       |            | 3            | rKey[1]              |
 * ------------------------------------------------------------
 * | ...   | ...        | ...          | ...                  |
 * ------------------------------------------------------------
 * | 15    | 3          | 3            | rKey[3]              |
 * +-------+------------+--------------+----------------------+
*/
        u32 shiftedWord =
            rKey[wordIndex] >> (8 * (3 - bytePosition));

        u8 keyByte = shiftedWord & 0xFF;
        state[i] ^= keyByte;

/* Extract the corresponding byte from the round key word */
// state[i] ^= (rKey[i / 4] >> (8 * (3 - (i % 4)))) & 0xFF;
    }
}
```

2.2.3 SubBytes / InvSubBytes

- SubBytes : $\{0, 1\}^{128} \rightarrow \{0, 1\}^{128}$ is defined by

$$\text{SubBytes}(\{X_i\}_{i=0}^{15}) = \left\{s(X_i)\right\}_{i=0}^{15}.$$

- InvSubBytes : $\{0, 1\}^{128} \rightarrow \{0, 1\}^{128}$ is defined by

$$\text{SubBytes}(\{X_i\}_{i=0}^{15}) = \left\{s^{-1}(X_i)\right\}_{i=0}^{15}.$$

Table 2.1: Substitution Box

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
10	ca	82	...	...	...	...	...	...	...	...	...	...	...	...	...	...
30	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
40	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
50	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
60	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
70	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
80	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
90	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
a0	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
b0	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
c0	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
d0	...	...	...	...	...	...	...	...	...	...	...	...	...	c1	...	...
e0	...	...	...	...	...	...	...	...	...	...	...	...	...	...	28	...
f0	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	16

Code 2.6: Byte Substitution

```c
void SubBytes(u8* state) {
    for (int i = 0; i < AES_BLOCK_SIZE; i++) {
        state[i] = s_box[state[i]];
    }
}
```

Code 2.7: Inverse Byte Substitution

```c
void SubBytes(u8* state) {
    for (int i = 0; i < AES_BLOCK_SIZE; i++) {
        state[i] = inv_s_box[state[i]];
    }
}
```

2.2.4 ShiftRows / InvShiftRows

- ShiftRows : $\{0,1\}^{128} \longrightarrow \{0,1\}^{128}$ is defined by

$$\begin{array}{|c|c|c|c|}\hline X_0 & X_4 & X_8 & X_{12} \\ \hline X_1 & X_5 & X_9 & X_{13} \\ \hline X_2 & X_6 & X_{10} & X_{14} \\ \hline X_3 & X_7 & X_{11} & X_{15} \\ \hline \end{array} \quad \Longrightarrow \quad \begin{array}{|c|c|c|c|}\hline X_0 & X_4 & X_8 & X_{12} \\ \hline X_5 & X_9 & X_{13} & X_1 \\ \hline X_{10} & X_{14} & X_2 & X_6 \\ \hline X_{15} & X_3 & X_7 & X_{11} \\ \hline \end{array}$$

- InvShiftRows : $\{0,1\}^{128} \longrightarrow \{0,1\}^{128}$ is defined by

$$\begin{array}{|c|c|c|c|}\hline X_0 & X_4 & X_8 & X_{12} \\ \hline X_1 & X_5 & X_9 & X_{13} \\ \hline X_2 & X_6 & X_{10} & X_{14} \\ \hline X_3 & X_7 & X_{11} & X_{15} \\ \hline \end{array} \quad \Longrightarrow \quad \begin{array}{|c|c|c|c|}\hline X_0 & X_4 & X_8 & X_{12} \\ \hline X_{13} & X_1 & X_5 & X_9 \\ \hline X_{10} & X_{14} & X_2 & X_6 \\ \hline X_7 & X_{11} & X_{15} & X_3 \\ \hline \end{array}$$

Code 2.8: ShiftRows

```c
void ShiftRows(u8* state) {
    u8 temp;

    // Row 1: shift left by 1
    temp = state[1];
    state[1] = state[5];
    state[5] = state[9];
    state[9] = state[13];
    state[13] = temp;

    // Row 2: shift left by 2
    temp = state[2];
    state[2] = state[10];
    state[10] = temp;
    temp = state[6];
    state[6] = state[14];
    state[14] = temp;

    // Row 3: shift left by 3 (or right by 1)
    temp = state[15];
    state[15] = state[11];
    state[11] = state[7];
    state[7] = state[3];
    state[3] = temp;
}
```

Code 2.9: Inverse ShiftRows

```
void InvShiftRows(u8* state) {
    u8 temp;

    // Row 1: shift left by 3 (or right by 1)
    temp = state[13];
    state[13] = state[9];
    state[9] = state[5];
    state[5] = state[1];
    state[1] = temp;

    // Row 2: shift left by 2
    temp = state[2];
    state[2] = state[10];
    state[10] = temp;
    temp = state[6];
    state[6] = state[14];
    state[14] = temp;

    // Row 3: shift left by 1
    temp = state[3];
    state[3] = state[7];
    state[7] = state[11];
    state[11] = state[15];
    state[15] = temp;
}
```

2.2.5 MixColumns / InvMixColumns

- Multiplication in the finite filed GF(2^8).

  $$\mathrm{MUL}_{GF256} : \{0, 1\}^8 \times \{0, 1\}^8 \to \{0, 1\}^8 .$$

  Here,

  $$\{0, 1\}^8 \simeq GF(2^8) = \mathbb{F}_{2^8} := \mathbb{F}_2[z]/(z^8 + z^4 + z^3 + z + 1) = \left\{a_7 z^7 + \cdots + a_1 z + a_0 : a_i \in \mathbb{F}_2\right\} .$$

  Note that
  $$a(z) \times b(z) := a(z) \times b(z) \bmod (z^8 + z^4 + z^3 + z + 1)$$

Note. Given two polynomials $a(x)$ and $b(x)$ in $GF(2^8)$:

$$a(x) = a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0,$$
$$b(x) = b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0.$$

The algorithm performs polynomial multiplication in the finite field $GF(2^8)$. It uses a shift-and-add method, with an additional reduction step modulo an irreducible polynomial $m(x) = x^8 + x^4 + x^3 + x + 1$.

1. Initialization: Set $p(x) = 0$ to initialize the product polynomial.
2. Iterate over each bit of $b(x)$, from LSB to MSB.
   (i) If the current bit b_i of $b(x)$ is 1, update $p(x)$ as $p(x) \oplus a(x)$. In $GF(2^8)$, addition is equivalent to the XOR operation:
   $$p(x) = p(x) \oplus a(x).$$
   (ii) Shift $a(x)$ left by 1 (multiply by x), increasing its degree by 1:
   $$a(x) = a(x) \cdot x.$$
   (iii) If the coefficient of x^8 in $a(x)$ is 1, reduce $a(x)$ by $m(x)$ to keep the degree under 8:
   $$a(x) = a(x) \oplus m(x).$$
   (iv) Shift $b(x)$ right by 1 (divide by x) for the next iteration:
   $$b(x) = b(x) \ / \ x.$$
3. After all bits of $b(x)$ are processed, $p(x)$ be the product of $a(x)$ and $b(x)$ modulo $m(x)$.

Note (Modular Reduction in $GF(2^8)$ using XOR). In the context of multiplication in the binary finite field $GF(2^8)$, modular reduction ensures that results of operations remain within the field. The use of XOR for modular reduction is due to the properties of polynomial arithmetic over GF(2) and the representation of elements in $GF(2^8)$.

– **Polynomial Representation in $GF(2^8)$:**

1. **Elements as Polynomials**: Each element in $GF(2^8)$ can be represented as a polynomial of degree less than 8, where each coefficient is either 0 or 1, i.e.,

$$GF(2^8) = \mathbb{F}_2[x]/(x^8 + x^4 + x^3 + x + 1) = \left\{a_7x^7 + \cdots + a_1x + a_0 : a_i \in \mathbb{F}_2\right\}.$$

This corresponds to an 8-bit binary number, with each bit representing a coefficient of the polynomial, i.e.,

$$a_7x^7 + \cdots + a_1x + a_0 \iff (a_7 \ldots a_1a_0)_2.$$

2. **Binary Operations**: In $GF(2)$, addition and subtraction are equivalent to the XOR operation, since $1 + 1 = 0$ in this field, the same as $1 \oplus 1$.

– **Modular Reduction with an Irreducible Polynomial**

1. **Irreducible Polynomial**: In $GF(2^8)$, an irreducible polynomial of degree 8, typically $p(x) = x^8 + x^4 + x^3 + x + 1$ (represented as `0x11b` in binary), is used for modular reduction.
2. **Modular Reduction Process**: After multiplying two polynomials, if the resulting polynomial's degree is 8 or higher, it must be reduced modulo the irreducible polynomial to ensure the result remains a polynomial of degree less than 8, thus staying within $GF(2^8)$.
3. **XOR for Reduction**: XOR is used for modular reduction in $GF(2^8)$ because polynomial subtraction in GF(2) is performed by XORing coefficients.

– Given two elements in $GF(2^8)$, $a(x)$ and $b(x)$, their product is $c(x) = a(x) \cdot b(x)$. If $\deg(c(x)) \geq 8$, then $c(x)$ must be reduced modulo the irreducible polynomial $p(x)$. This is achieved by XORing the coefficients of $c(x)$ and $p(x)$:

$$c(x) = a(x) \cdot b(x) \mod p(x)$$

If $c(x)$ has a term x^8 or higher, we subtract $p(x)$ from $c(x)$ to reduce its degree. In GF(2), subtraction is equivalent to addition, performed by XORing coefficients:

$$c'(x) = c(x) \oplus p(x)$$

This operation effectively eliminates the term x^8 (or higher) in $c(x)$, ensuring that the result remains within $GF(2^8)$. Consider the product of two polynomials $a(x)$ and $b(x)$ in $GF(2^8)$:

$$a(x) = x^6 + x^4 + x^2 + x + 1 \quad \text{and} \quad b(x) = x^7 + x + 1$$

The product $c(x) = a(x) \cdot b(x)$ might yield a polynomial of degree 8 or higher. To reduce $c(x)$ modulo $p(x) = x^8 + x^4 + x^3 + x + 1$, we perform XOR between the coefficients of $c(x)$ and $p(x)$, ensuring the result stays within $GF(2^8)$.

Code 2.10: Multiplication in GF(2^8)

```
u8 MUL_GF256(u8 a, u8 b) {
    u8 res = 0;
    // Mask for detecting the MSB (0x80 = 0b10000000)
    u8 MSB_mask = 0x80;
    u8 MSB;
    /*
     * The reduction polynomial
     * (x^8 + x^4 + x^3 + x + 1) = 0b100011011
     * for AES, represented in hexadecimal
    */
    u8 modulo = 0x1B;

    for (int i = 0; i < 8; i++) {
        // Add a to result if LSB(b)=1
        if (b & 1)
            res ^= a;

        MSB = a & MSB_mask; // Store the MSB of a
        a <<= 1; // Multiplying it by x effectively

        // Reduce the result modulo the reduction polynomial
        if (MSB)
            a ^= modulo;

        b >>= 1; // Moving to the next bit
    }

    return res;
}

#define MUL_GF256(a, b) ({ \
    u8 res = 0; \
    u8 MSB_mask = 0x80; \
    u8 MSB; \
    u8 modulo = 0x1B; \
    u8 temp_a = (a); \
    u8 temp_b = (b); \
    for (int i = 0; i < 8; i++) { \
        if (temp_b & 1) \
        res ^= temp_a; \
        MSB = temp_a & MSB_mask; \
        temp_a <<= 1; \
        if (MSB) \
        temp_a ^= modulo; \
        temp_b >>= 1; \
    } \
    res; \
})
```

- MixColumns : $\{0,1\}^{128} \to \{0,1\}^{128}$ is defined by

$$\text{MixColumns}\left(\begin{pmatrix} X_0 & X_4 & X_8 & X_{12} \\ X_1 & X_5 & X_9 & X_{13} \\ X_2 & X_6 & X_{10} & X_{14} \\ X_3 & X_7 & X_{11} & X_{15} \end{pmatrix}\right) := \begin{pmatrix} \texttt{0x02} & \texttt{0x03} & \texttt{0x01} & \texttt{0x01} \\ \texttt{0x01} & \texttt{0x02} & \texttt{0x03} & \texttt{0x01} \\ \texttt{0x01} & \texttt{0x01} & \texttt{0x02} & \texttt{0x03} \\ \texttt{0x03} & \texttt{0x01} & \texttt{0x01} & \texttt{0x02} \end{pmatrix} \begin{pmatrix} X_0 & X_4 & X_8 & X_{12} \\ X_1 & X_5 & X_9 & X_{13} \\ X_2 & X_6 & X_{10} & X_{14} \\ X_3 & X_7 & X_{11} & X_{15} \end{pmatrix}.$$

- InvMixColums : $\{0,1\}^{128} \to \{0,1\}^{128}$ is defined by

$$\text{MixColums}\left(\begin{pmatrix} X_0 & X_4 & X_8 & X_{12} \\ X_1 & X_5 & X_9 & X_{13} \\ X_2 & X_6 & X_{10} & X_{14} \\ X_3 & X_7 & X_{11} & X_{15} \end{pmatrix}\right) := \begin{pmatrix} \texttt{0x0e} & \texttt{0x0b} & \texttt{0x0d} & \texttt{0x09} \\ \texttt{0x09} & \texttt{0x0e} & \texttt{0x0b} & \texttt{0x0d} \\ \texttt{0x0d} & \texttt{0x09} & \texttt{0x0e} & \texttt{0x0b} \\ \texttt{0x0b} & \texttt{0x0d} & \texttt{0x09} & \texttt{0x0e} \end{pmatrix} \begin{pmatrix} X_0 & X_4 & X_8 & X_{12} \\ X_1 & X_5 & X_9 & X_{13} \\ X_2 & X_6 & X_{10} & X_{14} \\ X_3 & X_7 & X_{11} & X_{15} \end{pmatrix}.$$

Code 2.11: MixColumns

```c
void MixColumns(u8* state) {
    u8 temp[4];
    // Multiply and add the elements in the column
    // by the fixed polynomial
    for (int i = 0; i < 4; i++) {
        temp[0] =
            MUL_GF256(0x02, state[i * 4]) ^
            MUL_GF256(0x03, state[i * 4 + 1]) ^
            state[i * 4 + 2] ^
            state[i * 4 + 3];

        temp[1] =
            state[i * 4] ^
            MUL_GF256(0x02, state[i * 4 + 1]) ^
            MUL_GF256(0x03, state[i * 4 + 2]) ^
            state[i * 4 + 3];

        temp[2] =
            state[i * 4] ^
            state[i * 4 + 1] ^
            MUL_GF256(0x02, state[i * 4 + 2]) ^
            MUL_GF256(0x03, state[i * 4 + 3]);

        temp[3] =
            MUL_GF256(0x03, state[i * 4]) ^
            state[i * 4 + 1] ^
            state[i * 4 + 2] ^
            MUL_GF256(0x02, state[i * 4 + 3]);

        // Copy the mixed column back to the state
        for (int j = 0; j < 4; j++)
            state[i * 4 + j] = temp[j];
    }
}
```

Code 2.12: Inverse MixColumns

```
void InvMixColumns(u8* state) {
    u8 temp[4];

    for (int i = 0; i < 4; i++) {
        temp[0] =
            MUL_GF256(0x0e, state[i * 4]) ^
            MUL_GF256(0x0b, state[i * 4 + 1]) ^
            MUL_GF256(0x0d, state[i * 4 + 2]) ^
            MUL_GF256(0x09, state[i * 4 + 3]);

        temp[1] =
            MUL_GF256(0x09, state[i * 4]) ^
            MUL_GF256(0x0e, state[i * 4 + 1]) ^
            MUL_GF256(0x0b, state[i * 4 + 2]) ^
            MUL_GF256(0x0d, state[i * 4 + 3]);

        temp[2] =
            MUL_GF256(0x0d, state[i * 4]) ^
            MUL_GF256(0x09, state[i * 4 + 1]) ^
            MUL_GF256(0x0e, state[i * 4 + 2]) ^
            MUL_GF256(0x0b, state[i * 4 + 3]);

        temp[3] =
            MUL_GF256(0x0b, state[i * 4]) ^
            MUL_GF256(0x0d, state[i * 4 + 1]) ^
            MUL_GF256(0x09, state[i * 4 + 2]) ^
            MUL_GF256(0x0e, state[i * 4 + 3]);

        for (int j = 0; j < 4; j++)
            state[i * 4 + j] = temp[j];
    }
}
```

Chapter 3

AES - 128 / 192 / 256 (Byte Version)

3.1 Specification

Table 3.1: Parameters of the Block Cipher AES (1-word = 32-bit)

Algorithms	Block Size (N_b-word)	Key Length (N_k-word)	Number of Rounds (N_r)	Round-Key Length (word)	Number of Round-Keys ($N_r + 1$)	Total Size of Round-Keys ($N_b(N_r + 1)$)
AES-128	4	4 (4·32-bit)	10	4	11	44 (176-byte)
AES-192	4	6 (6·32-bit)	12	4	13	52 (208-byte)
AES-256	4	8 (8·32-bit)	14	4	15	60 (240-byte)

Code 3.1: Configuration

```c
// Define macros for AES key length
#define AES_VERSION 128 // Can be 128, 192, or 256
// Define macro for AES block size
#define AES_BLOCK_SIZE 16

// Define Nk and Nr based on AES key length
#if AES_VERSION == 128
    #define Nk 4
    #define Nr (Nk + 6) // 10
    #define ROUND_KEYS_SIZE (16 * (Nr + 1)) // 176
#elif AES_VERSION == 192
    #define Nk 6
    #define Nr (Nk + 6) // 12
    #define ROUND_KEYS_SIZE (16 * (Nr + 1)) // 208
#elif AES_VERSION == 256
    #define Nk 8
    #define Nr (Nk + 6) // 14
    #define ROUND_KEYS_SIZE (16 * (Nr + 1)) // 240
#else
    #error "Invalid AES ky length"
#endif
```

3.2 Key Expansion (General Version)

Algorithm 5: Key Schedule (General Version)

```
Input: User-key $uk = (uk_0, \ldots, uk_{N_k-1})$ $(uk_i \in \{0,1\}^8)$;      // $uk$ is 16/24/32-byte
Output: Round-key $\{rk_i\}_{i=0}^{4(N_r+1)-1}$ $(rk_i \in \{0,1\}^{32})$
/* $\{rk_i\}_{i=0}^{4(N_r+1)-1}$ is 176/208/240-byte */
1  $l \leftarrow N_k/4$;                                                       // $l = 4, 6, 8$
2  for $i = 0$ to $l - 1$ do
3      $rk_i \leftarrow uk_{4i} \| uk_{4i+1} \| uk_{4i+2} \| uk_{4i+3}$;
4  end
5  for $i = l$ to $4(N_r + 1) - 1$ do
6      $t \leftarrow rk_{i-1}$;
7      if $i \bmod l = 0$ then
8          $t \leftarrow (\text{SubWord} \circ \text{RotWord})(t)$;
9          $t \leftarrow t \oplus_{32} (\text{rCon}_{i/l} \| \text{0x00} \| \text{0x00} \| \text{0x00})$;
10     else if $l > 6$ && $i \bmod l = 4$ then
11         $t \leftarrow \text{SubWord}(t)$;
12     end
13     $rk_i \leftarrow rk_{i-l} \oplus_{32} t$;
14 end
```

Code 3.2: Key Expansion (General ver.)

```c
void KeyExpansion(const u8* uKey, u32* rKey) {
    u32 temp;

    for (int i = 0; i < Nk; i++) {
        rKey[i] = (u32)uKey[4*i] << 0x18 |
                  (u32)uKey[4*i+1] << 0x10 |
                  (u32)uKey[4*i+2] << 0x08 |
                  (u32)uKey[4*i+3];
    }

    for (int i = Nk; i < (Nr + 1) * 4; i++) {
        temp = rKey[i - 1];
        if (i % Nk == 0) {
            temp = SubWord(RotWord(temp)) ^ rCon[i / Nk - 1];
        } else if (Nk > 6 && i % Nk == 4) {
            // Additional S-box transformation for AES-256
            temp = SubWord(temp);
        }
        rKey[i] = rKey[i - Nk] ^ temp;
    }
}
```

Code 3.3: Key Expansion Test

```c
void RANDOM_KEY_GENERATION(u8* key) {
    srand((u32)time(NULL));

    // Initialize pointer to the start of the key array
    u8* p = key;

    // Set the counter to 16 bytes
    int cnt = 0;

    // Loop until all 16 bytes are filled
    while (cnt < AES_BLOCK_SIZE) {
        *p = rand() & 0xff; // Assign a random byte (0 to 255)
        p++;                // Move to the next byte
        cnt++;              // Decrement the byte count
    }
}

void KeyExpansionTest() {
    u8 uKey[AES_BLOCK_SIZE] = { 0x00, };
    RANDOM_KEY_GENERATION(uKey);
    // u8 uKey[AES_BLOCK_SIZE] = {
    //   0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6,
    //   0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c
    // };
    for (int i = 0; i < AES_BLOCK_SIZE; i++) {
        printf("%02x", uKey[i]);
    } printf("\n");

    u32 rKeys[ROUND_KEYS_SIZE / sizeof(u32)];
    KeyExpansion(uKey, rKeys);
    for (int i = 0; i < ROUND_KEYS_SIZE / sizeof(u32); i++) {
        printf("%08x\n", rKeys[i]);
    }
}

int main() {
    KeyExpansionTest();
    return 0;
}
```

3.3 8-bit AES - 128 / 192 / 256

Algorithm 6: Encryption of 8-bit AES

```
**Input:** block src $\in \{0,1\}^{128}$, round-keys $\{rk_i\}_{i=0}^{N_r+1}$ $(rk_i \in \{0,1\}^{32*4})$
**Output:** block dst $\in \{0,1\}^{128}$
1  $t \leftarrow$ AddRoundKey(src, $rk_0$);      // AddRoundKey: $\{0,1\}^{8*16} \times \{0,1\}^{32*4} \rightarrow \{0,1\}^{8*16}$
2  for $i \leftarrow 1$ to $N_r - 1$ do
3      $t \leftarrow$ SubBytes($t$);                 // SubBytes: $\{0,1\}^{8*16} \rightarrow \{0,1\}^{8*16}$
4      $t \leftarrow$ ShiftRows($t$);                // ShiftRows: $\{0,1\}^{8*16} \rightarrow \{0,1\}^{8*16}$
5      $t \leftarrow$ MixColumns($t$);               // MixColumns: $\{0,1\}^{8*16} \rightarrow \{0,1\}^{8*16}$
6      $t \leftarrow$ AddRoundKey($t, rk_i$);
7  end
8  $t \leftarrow$ SubBytes($t$);
9  $t \leftarrow$ ShiftRows($t$);
10 $t \leftarrow$ AddRoundKey($t, rk_{N_r}$);
11 dst $\leftarrow t$;
12 return dst;
```

Code 3.4: 8-bit AES Encryption

```c
void AES_Encrypt(const u8* plaintext, const u8* key,
    u8* ciphertext) {
    // AES-128/192/256: roundKey[44]/roundKey[52]/roundKey[60]
    u32 roundKey[ROUND_KEYS_SIZE / sizeof(u32)];
    u8 state[AES_BLOCK_SIZE]; // state[16]

    // Copy plaintext to state
    for (int i = 0; i < AES_BLOCK_SIZE; i++)
        state[i] = plaintext[i];

    KeyExpansion(key, roundKey);

// 0: roundKey[  0] | roundKey[1] | roundKey[ 2] | roundKey[    3]
    AddRoundKey(state, roundKey); // Initial round

    for (int round = 1; round <= Nr; round++) { // Main rounds
        SubBytes(state); ShiftRows(state);
        if (round != Nr) MixColumns(state);
// 1: roundKey[  4] | roundKey[5] | roundKey[ 6] | roundKey[    7]
// 2: roundKey[  8] | roundKey[9] | roundKey[10] | roundKey[   11]
// i: roundKey[4*i] |     ...     |     ...      | roundKey[4*i+3]
        AddRoundKey(state, roundKey + 4 * round);
    }

    // Copy state to ciphertext
    for (int i = 0; i < AES_BLOCK_SIZE; i++)
        ciphertext[i] = state[i];
}
```

Algorithm 7: Decryption of 8-bit AES

```
Input: block src $\in \{0, 1\}^{128}$, round-keys $\{rk_i\}_{i=0}^{N_r+1}$ $(rk_i \in \{0, 1\}^{32*4})$
Output: block dst $\in \{0, 1\}^{128}$
1  $t \leftarrow$ AddRoundKey(src, $rk_{N_r}$);
2  for $i \leftarrow N_r - 1$ to 1 do
3      $t \leftarrow$ InvShiftRows($t$);
4      $t \leftarrow$ InvSubBytes($t$);
5      $t \leftarrow$ AddRoundKey($t, rk_i$);
6      $t \leftarrow$ InvMixColumns($t$);
7  end
8  $t \leftarrow$ InvShiftRows($t$);
9  $t \leftarrow$ InvSubBytes($t$);
10 $t \leftarrow$ AddRoundKey($t, rk_0$);
11 dst $\leftarrow t$;
12 return dst;
```

Code 3.5: 8-bit AES Decryption

```c
void AES_Decrypt(const u8* ciphertext, const u8* key,
    u8* plaintext) {
    u32 roundKey[ROUND_KEYS_SIZE / sizeof(u32)];
    u8 state[AES_BLOCK_SIZE];

    KeyExpansion(key, roundKey);

    for (int i = 0; i < AES_BLOCK_SIZE; i++)
        state[i] = ciphertext[i];

    // Initial round with the last round key
    AddRoundKey(state, roundKey + 4 * Nr);

    // Main rounds in reverse order
    for (int round = Nr - 1; round >= 0; round--) {
        InvShiftRows(state);
        InvSubBytes(state);
// i: roundKey[4*i]  |     ...      |     ...      | roundKey[4*i+3]
// 1: roundKey[  4]  | roundKey[5]  | roundKey[ 6] | roundKey[    7]
// 0: roundKey[  0]  | roundKey[1]  | roundKey[ 2] | roundKey[    3]
        AddRoundKey(state, roundKey + 4 * round);
        if (round != 0)
            InvMixColumns(state);
    }

    for (int i = 0; i < AES_BLOCK_SIZE; i++)
        plaintext[i] = state[i];
}
```

Chapter 4

Pre-Computation using SubMix

4.1 SubMix and InvSubInvMix

- SubMix : $\{0, 1\}^{128} \to \{0, 1\}^{128}$ and InvSubInvMix : $\{0, 1\}^{128} \to \{0, 1\}^{128}$ are defined by

$$\begin{aligned} \text{SubMix} &:= \text{MixColumns} \circ \text{SubBytes} \\ \text{InvSubInvMix} &:= \text{InvMixColumns} \circ \text{InvSubBytes}. \end{aligned}$$

Algorithm 8: Encryption and Decryption of AES using SubMix and InvSubInvMix

Input: block src $\in \{0, 1\}^{128}$, round-keys $\{rk_i\}_{i=0}^{N_r+1}$ $(rk_i \in \{0, 1\}^{128})$

Output: block dst $\in \{0, 1\}^{128}$

```
Function AES_ENC:
    $t \leftarrow$ src;
    $t \leftarrow$ AddRoundKey($t, rk_0$);
    for $i \leftarrow 1$ to $N_r - 1$ do
        $t \leftarrow$ ShiftRows($t$);
        $t \leftarrow$ SubMix($t$);
        $t \leftarrow$ AddRoundKey($t, rk_i$);
    end
    $t \leftarrow$ SubBytes($t$);
    $t \leftarrow$ ShiftRows($t$);
    $t \leftarrow$ AddRoundKey($t, rk_{N_r}$);
    dst $\leftarrow t$;
    return dst;
end
```

```
Function AES_DEC:
    $t \leftarrow$ src;
    $t \leftarrow$ AddRoundKey($t, rk_{N_r}$);
    for $i \leftarrow N_r - 1$ downto 1 do
        $t \leftarrow$ InvShiftRows($t$);
        $t \leftarrow$ InvSubInvMix($t$);
        $t' \leftarrow$ InvMixColumns($rk_i$);
        $t \leftarrow$ AddRoundKey($t, t'$);
    end
    $t \leftarrow$ InvShiftRows($t$);
    $t \leftarrow$ InvSubBytes($t$);
    $t \leftarrow$ AddRoundKey($t, rk_0$);
    dst $\leftarrow t$;
    return dst;
end
```

4.2 8×32 Table Look Up

Convert the 8-bit string $\{X_i\}_{i=0}^{15}\,(X_i \in \{0,1\}^8)$ into a 32-bit string $\{W_i\}_{i=0}^{3}\,(W_i \in \{0,1\}^{32})$ as follows:

$$W_i := (X_{4i} \ll 24) \parallel (X_{4i+1} \ll 16) \parallel (X_{4i+2} \ll 8) \parallel (X_{4i+3}) \in \{0,1\}^{32}$$

for $i = 0,1,2,3$. In other words,

X_0	$(W_0 \gg$ `0x18`$)$ `& 0xff`	X_4	$(W_1 \gg$ `0x18`$)$ `& 0xff`
X_1	$(W_0 \gg$ `0x10`$)$ `& 0xff`	X_5	$(W_1 \gg$ `0x10`$)$ `& 0xff`
X_2	$(W_0 \gg$ `0x08`$)$ `& 0xff`	X_6	$(W_1 \gg$ `0x08`$)$ `& 0xff`
X_3	(W_0) `& 0xff`	X_7	(W_1) `& 0xff`
X_8	$(W_2 \gg$ `0x18`$)$ `& 0xff`	X_{12}	$(W_3 \gg$ `0x18`$)$ `& 0xff`
X_9	$(W_2 \gg$ `0x10`$)$ `& 0xff`	X_{13}	$(W_3 \gg$ `0x10`$)$ `& 0xff`
X_{10}	$(W_2 \gg$ `0x08`$)$ `& 0xff`	X_{14}	$(W_3 \gg$ `0x08`$)$ `& 0xff`
X_{11}	(W_2) `& 0xff`	X_{15}	(W_3) `& 0xff`

Note that

X_0	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	X_{10}	X_{11}	X_{12}	X_{13}	X_{14}	X_{15}
W_0				W_1				W_2				W_3			

Algorithm 9: SubMix and InvSubInvMix

```
Input: $(W_0, W_1, W_2, W_3) \in \{0,1\}^{128=32\times 4}$
Output: $(V_0, V_1, V_2, V_3) \in \{0,1\}^{128=32\times 4}$

Function SubMix:
    for $i \leftarrow 0$ to 3 do
        $V_i \leftarrow Te_0\left((W_i \gg$ 0x18) & 0xff$\right)$
                $\oplus_{32}\, Te_1\left((W_i \gg$ 0x10) & 0xff$\right)$
                $\oplus_{32}\, Te_2\left((W_i \gg$ 0x08) & 0xff$\right)$
                $\oplus_{32}\, Te_3\,(W_i$ & 0xff);
    end
    return $(V_0, V_1, V_2, V_3)$;
end
```

```
Input: $(W_0, W_1, W_2, W_3) \in \{0,1\}^{128=32\times 4}$
Output: $(U_0, U_1, U_2, U_3) \in \{0,1\}^{128=32\times 4}$

Function InvSubInvMix:
    for $i \leftarrow 0$ to 3 do
        $U_i \leftarrow Td_0\left((W_i \gg$ 0x18) & 0xff$\right)$
                $\oplus_{32}\, Td_1\left((W_i \gg$ 0x10) & 0xff$\right)$
                $\oplus_{32}\, Td_2\left((W_i \gg$ 0x08) & 0xff$\right)$
                $\oplus_{32}\, Td_3\,(W_i$ & 0xff);
    end
    return $(U_0, U_1, U_2, U_3)$;
end
```

4.3 Generation of 8×32 Tables

Note that

$$\begin{pmatrix} V_0 & V_4 & V_8 & V_{12} \\ V_1 & V_5 & V_9 & V_{13} \\ V_2 & V_6 & V_{10} & V_{14} \\ V_3 & V_7 & V_{11} & V_{15} \end{pmatrix} = \mathrm{SubMix}\left(\begin{pmatrix} X_0 & X_4 & X_8 & X_{12} \\ X_1 & X_5 & X_9 & X_{13} \\ X_2 & X_6 & X_{10} & X_{14} \\ X_3 & X_7 & X_{11} & X_{15} \end{pmatrix}\right) :=$$

$$\begin{pmatrix} \texttt{0x02} & \texttt{0x03} & \texttt{0x01} & \texttt{0x01} \\ \texttt{0x01} & \texttt{0x02} & \texttt{0x03} & \texttt{0x01} \\ \texttt{0x01} & \texttt{0x01} & \texttt{0x02} & \texttt{0x03} \\ \texttt{0x03} & \texttt{0x01} & \texttt{0x01} & \texttt{0x02} \end{pmatrix} \begin{pmatrix} s(X_0) & s(X_4) & s(X_8) & s(X_{12}) \\ s(X_1) & s(X_5) & s(X_9) & s(X_{13}) \\ s(X_2) & s(X_6) & s(X_{10}) & s(X_{14}) \\ s(X_3) & s(X_7) & s(X_{11}) & s(X_{15}) \end{pmatrix},$$

$$\begin{pmatrix} U_0 & V_4 & U_8 & U_{12} \\ U_1 & V_5 & U_9 & U_{13} \\ U_2 & V_6 & U_{10} & U_{14} \\ U_3 & V_7 & U_{11} & U_{15} \end{pmatrix} = \mathrm{InvSubInvMix}\left(\begin{pmatrix} X_0 & X_4 & X_8 & X_{12} \\ X_1 & X_5 & X_9 & X_{13} \\ X_2 & X_6 & X_{10} & X_{14} \\ X_3 & X_7 & X_{11} & X_{15} \end{pmatrix}\right) :=$$

$$\begin{pmatrix} \texttt{0x0e} & \texttt{0x0b} & \texttt{0x0d} & \texttt{0x09} \\ \texttt{0x09} & \texttt{0x0e} & \texttt{0x0b} & \texttt{0x0d} \\ \texttt{0x0d} & \texttt{0x09} & \texttt{0x0e} & \texttt{0x0b} \\ \texttt{0x0b} & \texttt{0x0d} & \texttt{0x09} & \texttt{0x0e} \end{pmatrix} \begin{pmatrix} s^{-1}(X_0) & s^{-1}(X_4) & s^{-1}(X_8) & s^{-1}(X_{12}) \\ s^{-1}(X_1) & s^{-1}(X_5) & s^{-1}(X_9) & s^{-1}(X_{13}) \\ s^{-1}(X_2) & s^{-1}(X_6) & s^{-1}(X_{10}) & s^{-1}(X_{14}) \\ s^{-1}(X_3) & s^{-1}(X_7) & s^{-1}(X_{11}) & s^{-1}(X_{15}) \end{pmatrix}.$$

Then

$$\begin{pmatrix} V_{4i} \\ V_{4i+1} \\ V_{4i+2} \\ V_{4i+3} \end{pmatrix} = \begin{pmatrix} \texttt{0x02} & \texttt{0x03} & \texttt{0x01} & \texttt{0x01} \\ \texttt{0x01} & \texttt{0x02} & \texttt{0x03} & \texttt{0x01} \\ \texttt{0x01} & \texttt{0x01} & \texttt{0x02} & \texttt{0x03} \\ \texttt{0x03} & \texttt{0x01} & \texttt{0x01} & \texttt{0x02} \end{pmatrix} \begin{pmatrix} s(X_{4i}) \\ s(X_{4i+1}) \\ s(X_{4i+2}) \\ s(X_{4i+3}) \end{pmatrix}$$

$$= s(X_{4i}) \begin{pmatrix} \texttt{0x02} \\ \texttt{0x01} \\ \texttt{0x01} \\ \texttt{0x03} \end{pmatrix} \oplus_{32} s(X_{4i+1}) \begin{pmatrix} \texttt{0x03} \\ \texttt{0x02} \\ \texttt{0x01} \\ \texttt{0x01} \end{pmatrix} \oplus_{32} s(X_{4i+2}) \begin{pmatrix} \texttt{0x01} \\ \texttt{0x03} \\ \texttt{0x02} \\ \texttt{0x01} \end{pmatrix} \oplus_{32} s(X_{4i+3}) \begin{pmatrix} \texttt{0x01} \\ \texttt{0x01} \\ \texttt{0x03} \\ \texttt{0x02} \end{pmatrix},$$

$$\begin{pmatrix} U_{4i} \\ U_{4i+1} \\ U_{4i+2} \\ U_{4i+3} \end{pmatrix} = \begin{pmatrix} \texttt{0x0e} & \texttt{0x0b} & \texttt{0x0d} & \texttt{0x09} \\ \texttt{0x09} & \texttt{0x0e} & \texttt{0x0b} & \texttt{0x0d} \\ \texttt{0x0d} & \texttt{0x09} & \texttt{0x0e} & \texttt{0x0b} \\ \texttt{0x0b} & \texttt{0x0d} & \texttt{0x09} & \texttt{0x0e} \end{pmatrix} \begin{pmatrix} s^{-1}(X_{4i}) \\ s^{-1}(X_{4i+1}) \\ s^{-1}(X_{4i+2}) \\ s^{-1}(X_{4i+3}) \end{pmatrix}$$

$$= s^{-1}(X_{4i}) \begin{pmatrix} \texttt{0x0e} \\ \texttt{0x09} \\ \texttt{0x0d} \\ \texttt{0x0b} \end{pmatrix} \oplus s^{-1}(X_{4i+1}) \begin{pmatrix} \texttt{0x0b} \\ \texttt{0x0e} \\ \texttt{0x09} \\ \texttt{0x0d} \end{pmatrix} \oplus s^{-1}(X_{4i+2}) \begin{pmatrix} \texttt{0x0d} \\ \texttt{0x0b} \\ \texttt{0x0e} \\ \texttt{0x09} \end{pmatrix} \oplus s^{-1}(X_{4i+3}) \begin{pmatrix} \texttt{0x09} \\ \texttt{0x0d} \\ \texttt{0x0b} \\ \texttt{0x0e} \end{pmatrix}.$$

We define $Te_i/Td_i : \{0,1\}^8 \to \{0,1\}^{32}$ ($i \in \{0,1,2,3\}$) as follows:

$$\begin{aligned}
Te_0(X) &:= \left(\texttt{0x02} \otimes_8 s(X),\ s(X),\ s(X),\ \texttt{0x03} \otimes_8 s(X)\right),\\
Te_1(X) &:= \left(\texttt{0x03} \otimes_8 s(X),\ \texttt{0x02} \otimes_8 s(X),\ s(X),\ s(X)\right),\\
Te_2(X) &:= \left(s(X),\ \texttt{0x03} \otimes_8 s(X),\ \texttt{0x02} \otimes_8 s(X),\ s(X)\right),\\
Te_3(X) &:= \left(s(X),\ s(X),\ \texttt{0x03} \otimes_8 s(X),\ \texttt{0x02} \otimes_8 s(X)\right),
\end{aligned}$$

and

$$\begin{aligned}
Td_0(X) &:= \left(\texttt{0x0e} \otimes_8 s^{-1}(X),\ \texttt{0x09} \otimes_8 s^{-1}(X),\ \texttt{0x0d} \otimes_8 s^{-1}(X),\ \texttt{0x0b} \otimes_8 s^{-1}(X)\right),\\
Td_1(X) &:= \left(\texttt{0x0b} \otimes_8 s^{-1}(X),\ \texttt{0x0e} \otimes_8 s^{-1}(X),\ \texttt{0x09} \otimes_8 s^{-1}(X),\ \texttt{0x0d} \otimes_8 s^{-1}(X)\right),\\
Td_2(X) &:= \left(\texttt{0x0d} \otimes_8 s^{-1}(X),\ \texttt{0x0b} \otimes_8 s^{-1}(X),\ \texttt{0x0e} \otimes_8 s^{-1}(X),\ \texttt{0x09} \otimes_8 s^{-1}(X)\right),\\
Td_3(X) &:= \left(\texttt{0x09} \otimes_8 s^{-1}(X),\ \texttt{0x0d} \otimes_8 s^{-1}(X),\ \texttt{0x0b} \otimes_8 s^{-1}(X),\ \texttt{0x0e} \otimes_8 s^{-1}(X)\right).
\end{aligned}$$

SubMix/InvSubInvMix : $\{0,1\}^{128} \to \{0,1\}^{128}$ are computed as follows:

$$\begin{aligned}
\text{SubMix}(W_0, W_1, W_2, W_3) &:= (V_0, V_1, V_2, V_3),\\
\text{InvSubInvMix}(W_0, W_1, W_2, W_3) &:= (U_0, U_1, U_2, U_3)
\end{aligned}$$

where $\begin{cases} W_0 := X_0 \parallel X_1 \parallel X_2 \parallel X_3 \\ W_1 := X_4 \parallel X_5 \parallel X_6 \parallel X_7 \\ W_2 := X_8 \parallel X_9 \parallel X_{10} \parallel X_{11} \\ W_3 := X_{12} \parallel X_{13} \parallel X_{14} \parallel X_{15} \end{cases}$ ($W_i \in \{0,1\}^{32}$) and

$$\begin{array}{lcccccccc}
V_0 & := & Te_0(X_0) & \oplus_{32} & Te_1(X_1) & \oplus_{32} & Te_2(X_2) & \oplus_{32} & Te_3(X_3),\\
V_1 & := & Te_0(X_4) & \oplus_{32} & Te_1(X_5) & \oplus_{32} & Te_2(X_6) & \oplus_{32} & Te_3(X_7),\\
V_2 & := & Te_0(X_8) & \oplus_{32} & Te_1(X_9) & \oplus_{32} & Te_2(X_{10}) & \oplus_{32} & Te_3(X_{11}),\\
V_3 & := & Te_0(X_{12}) & \oplus_{32} & Te_1(X_{13}) & \oplus_{32} & Te_2(X_{14}) & \oplus_{32} & Te_3(X_{15}),\\
\\
U_0 & := & Td_0(X_0) & \oplus_{32} & Td_1(X_1) & \oplus_{32} & Td_2(X_2) & \oplus_{32} & Td_3(X_3),\\
U_1 & := & Td_0(X_4) & \oplus_{32} & Td_1(X_5) & \oplus_{32} & Td_2(X_6) & \oplus_{32} & Td_3(X_7),\\
U_2 & := & Td_0(X_8) & \oplus_{32} & Td_1(X_9) & \oplus_{32} & Td_2(X_{10}) & \oplus_{32} & Td_3(X_{11}),\\
U_3 & := & Td_0(X_{12}) & \oplus_{32} & Td_1(X_{13}) & \oplus_{32} & Td_2(X_{14}) & \oplus_{32} & Td_3(X_{15}).
\end{array}$$

4.4 Implementation of SubMix and InvSubInvMix

Code 4.1: SubMix

```
void SubMix(u8* state) {
    u32 temp[4];

    // temp[0] = Te0[state[ 0]] ^ Te1[state[ 1]] ^
    //           Te2[state[ 2]] ^ Te3[state[ 3]];
    // temp[1] = Te0[state[ 4]] ^ Te1[state[ 5]] ^
    //           Te2[state[ 6]] ^ Te3[state[ 7]];
    // temp[2] = Te0[state[ 8]] ^ Te1[state[ 9]] ^
    //           Te2[state[10]] ^ Te3[state[11]];
    // temp[3] = Te0[state[11]] ^ Te1[state[12]] ^
    //           Te2[state[13]] ^ Te3[state[14]];

    for (int i = 0; i < 4; i++) {
        temp[i] = Te0[state[4*i + 0]] ^
                  Te1[state[4*i + 1]] ^
                  Te2[state[4*i + 2]] ^
                  Te3[state[4*i + 3]];
    }

    for (int i = 0; i < 4; i++) {
        state[4*i + 0] = (temp[i] >> 0x18) & 0xff;
        state[4*i + 1] = (temp[i] >> 0x10) & 0xff;
        state[4*i + 2] = (temp[i] >> 0x08) & 0xff;
        state[4*i + 3] = temp[i] & 0xff;
    }
}
```

Code 4.2: InvSubInvMix

```
void InvSubInvMix(u8* state) {
    u32 temp[4];

    for (int i = 0; i < 4; i++) {
        temp[i] = Td0[state[4*i + 0]] ^
                  Td1[state[4*i + 1]] ^
                  Td2[state[4*i + 2]] ^
                  Td3[state[4*i + 3]];
    }

    for (int i = 0; i < 4; i++) {
        state[4*i + 0] = (temp[i] >> 0x18) & 0xff;
        state[4*i + 1] = (temp[i] >> 0x10) & 0xff;
        state[4*i + 2] = (temp[i] >> 0x08) & 0xff;
        state[4*i + 3] = temp[i] & 0xff;
    }
}
```

Chapter 5

Base64 Encoding and Decoding

5.1 Introduction

Base64 encoding is a method of converting binary data into a set of 64 printable characters from the ASCII standard. This encoding is commonly used to encode data when there is a requirement to transmit binary data over media that are designed to deal with textual data. The Base64 index table consists of the characters `A-Z`, `a-z`, `0-9`, plus two additional characters, commonly `+` and `/`, for a total of 64 characters.

5.2 Principles of Base64 Encoding

Base64 encoding processes data in blocks of 3 bytes (24 bits) at a time. Each block of 24 bits is then divided into four 6-bit groups. Each 6-bit group is used as an index into the Base64 character table, resulting in a 4-character encoding of the original 24-bit block.

Let the input stream be a sequence of bits represented by $\mathbf{b}$. This stream is divided into blocks of 24 bits:

$$\mathbf{b} = b_1 b_2 b_3 \dots b_{24} b_{25} b_{26} \dots$$

Each block of 24 bits $(b_1 b_2 \dots b_{24})$ is then split into four 6-bit groups:

$$\mathbf{g}_1 = b_1 b_2 \dots b_6, \ \mathbf{g}_2 = b_7 b_8 \dots b_{12}, \ \mathbf{g}_3 = b_{13} b_{14} \dots b_{18}, \ \mathbf{g}_4 = b_{19} b_{20} \dots b_{24}$$

Each group $\mathbf{g}_i$ is then converted into a decimal index n_i:

$$n_i = \sum_{k=0}^{5} g_{i,k} \cdot 2^{5-k}$$

where $g_{i,k}$ is the k-th bit of group $\mathbf{g}_i$.

Each index n_i is used to select a character from the Base64 table, forming the encoded string.

5.3 Principles of Base64 Decoding

Base64 decoding is the reverse process. Each character in the encoded string is mapped back to its corresponding 6-bit group. These groups are then concatenated to form the original byte sequence.

Given a Base64 encoded string, each character is converted back into a 6-bit group according to the Base64 index table. Let these groups be $\mathbf{d}_1, \mathbf{d}_2, \mathbf{d}_3, \mathbf{d}_4$. The original 24-bit block is reconstructed by concatenating these groups:

$$\mathbf{b}' = \mathbf{d}_1\mathbf{d}_2\mathbf{d}_3\mathbf{d}_4$$

The binary sequence $\mathbf{b}'$ is then divided back into the original bytes to retrieve the original data.

5.4 Padding in Base64 Encoding

In cases where the input byte sequence is not a multiple of 3, padding is used. The input is padded with zeros to form a complete 24-bit block. The corresponding Base64 encoded output is then padded with the ′=′ character to indicate that padding was used. The number of ′=′ characters used indicates the number of bytes that were added as padding (one byte of padding results in two ′=′ characters, and two bytes of padding results in one ′=′ character).

Chapter 6

Cryptanalysis

6.1 Linear Cryptanalysis

Linear cryptanalysis is an advanced cryptanalytic technique primarily used to attack symmetric key ciphers. It exploits the linear relations (approximations) between input bits, output bits, and key bits of a cipher. The technique was first introduced by Mitsuru Matsui and has since been instrumental in the analysis of various cryptographic algorithms.

6.1.1 Linear Approximation Example

Consider a block cipher with an input block X and an output block Y. The essence of linear cryptanalysis lies in finding a linear approximation that correlates certain bits of X, Y, and the key K. A typical linear approximation can be expressed as follows:

$$P(X) \oplus Q(Y) = 0$$

where $P(X)$ and $Q(Y)$ are linear functions of the input and output blocks, respectively, $\oplus$ denotes the bitwise XOR operation, and K' is a subkey or a combination of bits from the key K. Ideally, this equation holds with a probability significantly different from 0.5.

6.1.2 Linear Approximation Table (LAT)

A crucial tool in linear cryptanalysis is the Linear Approximation Table (LAT), particularly for analyzing S-boxes in block ciphers. An S-box is a fundamental component in many block ciphers that performs substitution, and LAT is used to measure the correlation between input and output bits of this S-box.

The LAT is a matrix where each cell corresponds to a particular input-output bit combination and contains a value representing the correlation between these bits. These values are critical in identifying which linear approximations are strong (i.e., have a high correlation) and can be exploited in an attack.

6.1.3 Attack Process

The general process of conducting a linear cryptanalytic attack involves several steps:

1. **Identify Strong Linear Approximations:** Using the LAT, identify the linear approximations with the highest bias. The bias is the deviation of the approximation's probability from 0.5.

    - *Mathematical Foundation:* Analyze the Linear Approximation Table (LAT) for strong linear correlations. LAT is a 2D matrix with biases of linear approximations as entries.
    - *Calculation:* For an S-box with n input and output bits, LAT is a $2^n \times 2^n$ matrix. Bias is calculated as $\frac{\#matches}{2^n} - \frac{1}{2}$.
    - *Selection Criterion:* Choose approximations with the highest absolute bias values.

2. **Estimate Bias:** Accurately estimate the bias of the chosen approximation. This estimation is crucial for the success of the attack.

    - *Theory:* Bias is the deviation of the approximation's probability from 0.5.
    - *Mathematical Approach:* Use statistical methods like maximum likelihood estimation or Bayesian inference for bias refinement.
    - *Accuracy Considerations:* More data leads to more accurate bias estimation.

3. **Collect Plaintext-Ciphertext Pairs:** Amass a large number of plaintext-ciphertext pairs that correspond to the chosen approximation. The number of required pairs depends on the bias; weaker biases require more pairs.

    - *Data Requirement Analysis:* Required pairs $N \approx \frac{1}{\epsilon^2}$, where ϵ is the bias.
    - *Collection Methodology:* Data collection can vary based on the cipher and scenario.
    - *Computational Aspects:* Consider the complexity of storing and processing large amounts of data.

4. **Key Recovery:** Apply statistical methods to analyze the collected data. Exploit the bias to make informed guesses about the key bits or subkey, thereby incrementally recovering parts of the encryption key.

    - *Statistical Analysis:* Use statistical tests like chi-squared tests to analyze data.
    - *Exploiting Bias:* Use bias to infer key bits probabilistically.
    - *Incremental Key Recovery:* Recover different segments of the key gradually.
    - *Complexity Considerations:* Analyze the time complexity and employ optimization techniques if necessary.

In practice, the effectiveness of linear cryptanalysis depends on the cipher's structure, the quality of the linear approximations, and the amount of available data. It has been notably applied to attack DES (Data Encryption Standard) and has influenced the design of new cryptographic algorithms to be resistant to this form of analysis.

6.2 Differential Cryptanalysis

Differential Cryptanalysis is a method used in cryptography to analyze the effect of specific differences in input pairs on the differences in the resulting output pairs. This technique is especially powerful in the analysis of block ciphers.

6.2.1 Differential Example

Given a pair of inputs (X, X') and their corresponding outputs (Y, Y') under a cryptographic transformation. The differentials are defined as follows:

$$\Delta X = X \oplus X', \quad \Delta Y = Y \oplus Y'$$

where $\oplus$ denotes the bitwise XOR operation. This operation highlights the changes from X to X' and from Y to Y' at the bit level.

6.2.2 Differential Characteristic

A differential characteristic is a crucial concept in this analysis. It is a sequence of expected input and output differentials for each round or stage of the cipher. These characteristics are probabilistic in nature and provide insights into how certain input differentials propagate to output differentials through the cipher's structure.

Probability of a Differential Characteristic

The probability of a differential characteristic is a key metric in differential cryptanalysis. It measures the likelihood that a specific input differential will result in the expected output differential after passing through the rounds of the cipher

6.2.3 Attack Process

1. **Select Pairs of Plaintexts with a Certain Difference:** Choose plaintext pairs (P, P') such that the difference $\Delta P = P \oplus P'$ is specific and strategically chosen based on the cipher's structure.

    - *Concept:* Choose plaintext pairs (P_1, P_2) such that $\Delta P = P_1 \oplus P_2$, where $\oplus$ is the bitwise XOR operation.
    - *Method:* Selection of ΔP is based on cipher's structure analysis. Aim to identify a ΔP that can cause predictable effects in ciphertexts.
    - *Mathematical Formulation:* This involves a heuristic approach to identify exploitable differences.

2. **Analyze How This Difference Propagates Through the Cipher:** Study the propagation of ΔP through the cipher. This involves analyzing the impact of the differential on the cipher's components like S-boxes, permutation layers, etc.

    - *Theory:* Study the propagation of ΔP through the cipher's rounds.
    - *Mathematical Tools:* Use differential characteristics and probabilistic analysis of the cipher components.

- *Computational Analysis:* Combine analytical techniques and computational experiments to understand difference propagation.

3. **Gather Plaintext-Ciphertext Pairs Following the Differential:** Collect a significant number of plaintext-ciphertext pairs that conform to the chosen differential. This data is used to analyze the behavior of the cipher under the differential attack.

   - *Data Collection:* Accumulate pairs (P, C) that conform to ΔP.
   - *Statistical Analysis:* Analyze the distribution of output difference $\Delta C = C_1 \oplus C_2$.
   - *Data Requirements:* The number of pairs depends on the differential characteristic's probability and cipher's strength.

4. **Make Key Hypotheses Based on Output Differences:** Based on the differences observed in the ciphertext pairs, hypothesize about the possible key or subkey values. These hypotheses are then tested and refined to recover the key or reduce the key space.

   - *Hypothesis Generation:* Formulate hypotheses about the key based on ΔC.
   - *Mathematical Techniques:* Use linear algebra, probability theory, and optimization algorithms for hypothesis refinement.
   - *Verification:* Test hypotheses against multiple plaintext-ciphertext pairs.

Appendix A

Additional Data A

A.1 Substitution-BOX

```
static const u8 s_box[256] = {
    0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5,
    0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76,
    0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0,
    0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0,
    0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc,
    0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15,
    0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a,
    0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75,
    0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0,
    0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84,
    0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b,
    0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf,
    0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85,
    0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8,
    0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5,
    0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2,
    0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17,
    0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73,
    0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88,
    0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb,
    0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c,
    0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79,
    0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9,
    0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08,
    0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6,
    0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a,
    0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e,
    0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e,
    0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94,
    0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf,
    0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68,
    0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16
};
```

```c
static const u8 inv_s_box[256] = {
    0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38,
    0xbf, 0x40, 0xa3, 0x9e, 0x81, 0xf3, 0xd7, 0xfb,
    0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87,
    0x34, 0x8e, 0x43, 0x44, 0xc4, 0xde, 0xe9, 0xcb,
    0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d,
    0xee, 0x4c, 0x95, 0x0b, 0x42, 0xfa, 0xc3, 0x4e,
    0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2,
    0x76, 0x5b, 0xa2, 0x49, 0x6d, 0x8b, 0xd1, 0x25,
    0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16,
    0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0x65, 0xb6, 0x92,
    0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda,
    0x5e, 0x15, 0x46, 0x57, 0xa7, 0x8d, 0x9d, 0x84,
    0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a,
    0xf7, 0xe4, 0x58, 0x05, 0xb8, 0xb3, 0x45, 0x06,
    0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02,
    0xc1, 0xaf, 0xbd, 0x03, 0x01, 0x13, 0x8a, 0x6b,
    0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea,
    0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4, 0xe6, 0x73,
    0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85,
    0xe2, 0xf9, 0x37, 0xe8, 0x1c, 0x75, 0xdf, 0x6e,
    0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89,
    0x6f, 0xb7, 0x62, 0x0e, 0xaa, 0x18, 0xbe, 0x1b,
    0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20,
    0x9a, 0xdb, 0xc0, 0xfe, 0x78, 0xcd, 0x5a, 0xf4,
    0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31,
    0xb1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xec, 0x5f,
    0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d,
    0x2d, 0xe5, 0x7a, 0x9f, 0x93, 0xc9, 0x9c, 0xef,
    0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0,
    0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61,
    0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26,
    0xe1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0c, 0x7d
};
```