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 18, 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:

  $$a(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1 x + a_0$$

  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 AES-128

1.1 Overview of Advanced Encryption Standard

- $\mathrm{KeyExpansion} : \{\mathtt{0}, \mathtt{1}\}^{128} \longrightarrow \{\mathtt{0}, \mathtt{1}\}^{1408}$.

- $\mathrm{AddRoundKey} : \{\mathtt{0}, \mathtt{1}\}^{128} \times \{\mathtt{0}, \mathtt{1}\}^{128} \longrightarrow \{\mathtt{0}, \mathtt{1}\}^{128}$.

- $\mathrm{SubBytes} : \{\mathtt{0}, \mathtt{1}\}^{128} \longrightarrow \{\mathtt{0}, \mathtt{1}\}^{128}$.

- $\mathrm{ShiftRows} : \{\mathtt{0}, \mathtt{1}\}^{128} \longrightarrow \{\mathtt{0}, \mathtt{1}\}^{128}$.

- $\mathrm{MixColumns} : \{\mathtt{0}, \mathtt{1}\}^{128} \longrightarrow \{\mathtt{0}, \mathtt{1}\}^{128}$.

Algorithm 1: Encryption of AES-128

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

Algorithm 2: 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 $\in \{0, 1\}^{128}$
$t \leftarrow$ src;
$t \leftarrow$ AddRoundKey($t, rk_{10}$);
for $i \leftarrow 9$ to 1 do
    $t \leftarrow$ InvShiftRows($t$);
    $t \leftarrow$ InvSubBytes($t$);
    $t \leftarrow$ AddRoundKey($t, rk_i$);
    $t \leftarrow$ InvMixColumns($t$);
end
$t \leftarrow$ SubBytes($t$);
$t \leftarrow$ ShiftRows($t$);
$t \leftarrow$ AddRoundKey($t, rk_0$);
dst $\leftarrow t$;
return dst;
```

1.2 Functions and Constants used in AES

1.2.1 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 1.1: 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 1.2: 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 1.3: rCon Array Declaration

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

Algorithm 3: 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
$rk_0 \leftarrow uk_0 \| uk_1 \| uk_2 \| uk_3$;
$rk_1 \leftarrow uk_4 \| uk_5 \| uk_6 \| uk_7$;
$rk_2 \leftarrow uk_8 \| uk_9 \| uk_{10} \| uk_{11}$;
$rk_3 \leftarrow uk_{12} \| uk_{13} \| uk_{14} \| uk_{15}$;
for $i = 4$ to $43$ do
    $t \leftarrow rk_{i-1}$;
    if $i \bmod 4 = 0$ then
        /* SubWord ∘ RotWord : $\{0,1\}^{32} \rightarrow \{0,1\}^{32}$                */
        $t \leftarrow \mathrm{RotWord}(t)$;
        $t \leftarrow \mathrm{SubWord}(t)$;
        $t \leftarrow t \oplus (\mathrm{rCon}_{i/4} \| \texttt{0x00} \| \texttt{0x00} \| \texttt{0x00})$;
    end
    $rk_i \leftarrow rk_{i-4} \oplus_{32} t$;
end
```

Code 1.4: AES 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++;
    }
}
```

1.2.2 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 1.5: AES AddRoundKey

```
void AddRoundKey(u8* state, const u32* rKey) {
    for (int i = 0; i < AES_KEY_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;
    }
}
```

1.2.3 SubBytes / InvSubBytes

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

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

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

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

Table 1.1: 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 1.6: Byte Substitution

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

Code 1.7: Inverse Byte Substitution

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

1.2.4 ShiftRows / InvShiftRows

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

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

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

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

Code 1.8: 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 1.9: ShiftRows

```c
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;
}
```

1.2.5 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,

$$\{0,1\}^8 \simeq GF(2^8) = \mathbb{F}_{2^8} := \mathbb{F}_2[z]/(z^8+z^4+z^3+z+1) = \left\{a_7z^7+\cdots+a_1z+a_0 : a_i \in \mathbb{F}_2\right\}.$$

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 $GF(2^8)$:

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

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 1.10: Multiplication in GF(2^8)

```c
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 1.11: 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 1.12: Inverse MixColumns

```c
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 2

AES - 128 / 192 / 256

2.1 Specification

Table 2.1: Parameters of the Block Cipher AES

Algorithms	Block	Key	Number of	Round-Key	Number of	Total Size of
	Size	Length	Rounds	Length	Round-Keys	Round-Keys
	(N_b-byte)	(N_k-byte)	(N_r)	(byte)	($N_r + 1$)	($N_b(N_r + 1)$)
AES-128	16	16 (128-bit)	10	16	11	176
AES-192	16	24 (192-bit)	12	16	13	208
AES-256	16	32 (256-bit)	14	16	15	240

Code 2.1: Configuration

```
// 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
```

2.2 Key Expansion (General Version)

Algorithm 4: 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 */
$l \leftarrow N_k/4$;   // $l = 4, 6, 8$
for $i = 0$ to $l - 1$ do
    $rk_i \leftarrow uk_{4i} \| uk_{4i+1} \| uk_{4i+2} \| uk_{4i+3}$;
end
for $i = l$ to $4(N_r + 1) - 1$ do
    $t \leftarrow rk_{i-1}$;
    if $i \bmod l = 0$ then
        $t \leftarrow \mathrm{SubWord} \circ \mathrm{RotWord}(t)$;
        $t \leftarrow t \oplus (\mathrm{rCon}_{i/l} \| \texttt{0x00} \| \texttt{0x00} \| \texttt{0x00})$;
    else if $l > 6$ && $i \bmod l = 4$ then
        $t \leftarrow \mathrm{SubWord}(t)$;
    end
    $rk_i \leftarrow rk_{i-l} \oplus_{32} t$;
end
```

Code 2.2: 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;
    }
}
```

2.3 Advanced Encryption Standard - 128 / 192 / 256

Algorithm 5: Encryption of AES

```
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 $\in \{0, 1\}^{128}$
$t \leftarrow$ src;
$t \leftarrow \mathrm{AddRoundKey}(t, rk_0)$;
for $i \leftarrow 1$ to $N_r - 1$ do
    $t \leftarrow \mathrm{SubBytes}(t)$;
    $t \leftarrow \mathrm{ShiftRows}(t)$;
    $t \leftarrow \mathrm{MixColumns}(t)$;
    $t \leftarrow \mathrm{AddRoundKey}(t, rk_i)$;
end
$t \leftarrow \mathrm{SubBytes}(t)$;
$t \leftarrow \mathrm{ShiftRows}(t)$;
$t \leftarrow \mathrm{AddRoundKey}(t, rk_{N_r})$;
dst $\leftarrow t$;
return dst;
```

Algorithm 6: Decryption of AES

```
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 $\in \{0, 1\}^{128}$
$t \leftarrow$ src;
$t \leftarrow \mathrm{AddRoundKey}(t, rk_{N_r})$;
for $i \leftarrow N_r - 1$ to $1$ do
    $t \leftarrow \mathrm{InvShiftRows}(t)$;
    $t \leftarrow \mathrm{InvSubBytes}(t)$;
    $t \leftarrow \mathrm{AddRoundKey}(t, rk_i)$;
    $t \leftarrow \mathrm{InvMixColumns}(t)$;
end
$t \leftarrow \mathrm{InvSubBytes}(t)$;
$t \leftarrow \mathrm{InvShiftRows}(t)$;
$t \leftarrow \mathrm{AddRoundKey}(t, rk_0)$;
dst $\leftarrow t$;
return dst;
```

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
};
```

```
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
};
```