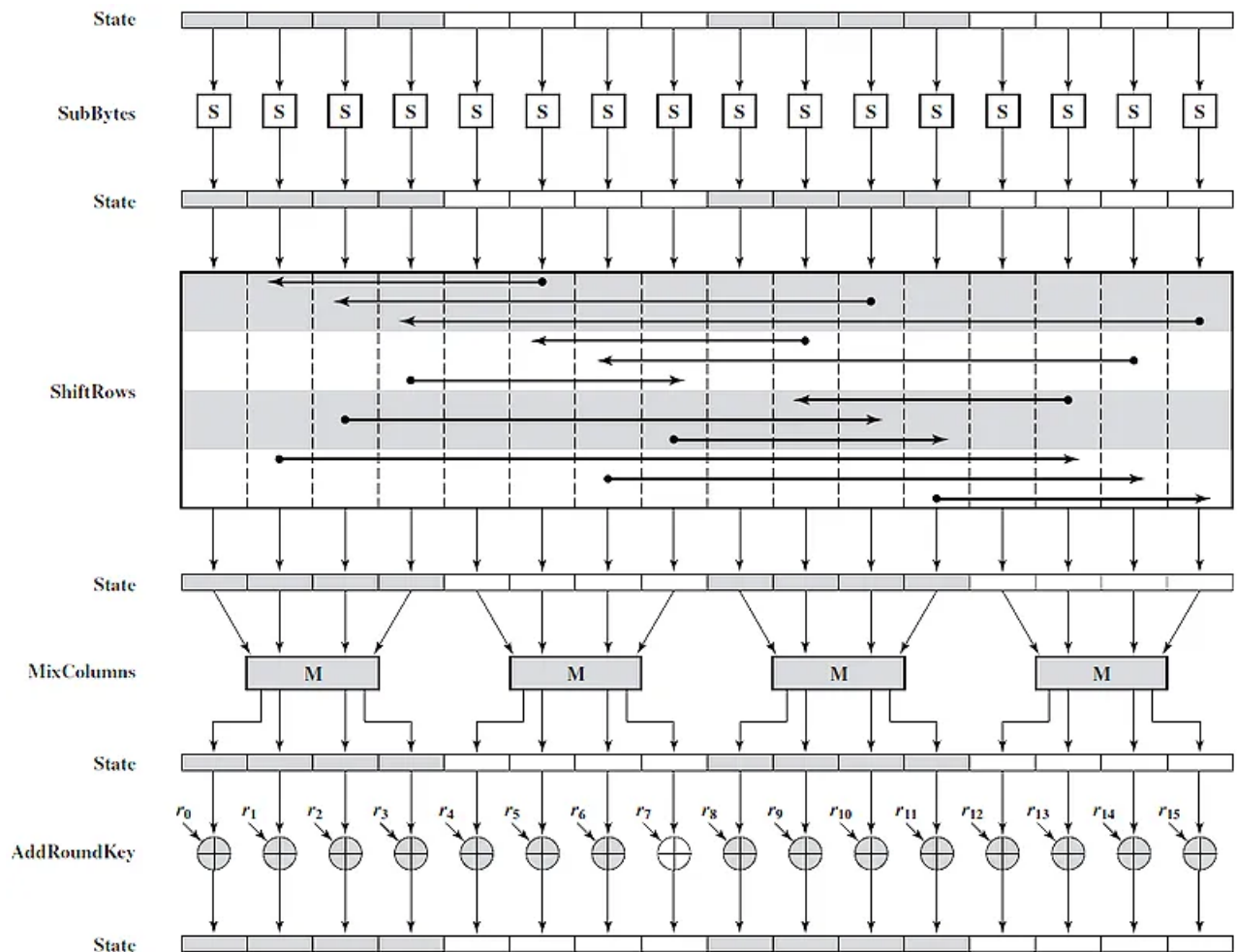# C | SecureAES
- High-Performance AES Encryption in C -

Ji Yong-Hyeon

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

November 18, 2024

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	$\overset{?}{|P| = |C|}$
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 $P = P_1 \parallel \cdots \parallel P_N$ ($P_i \in \{0, 1\}^n$)   **Input:** K and $C = C_1 \parallel \cdots \parallel C_N$ ($C_i \in \{0, 1\}^n$)
Output: $C = C_1 \parallel \cdots \parallel C_N$ ($C_i \in \{0, 1\}^n$)   **Output:** $P = P_1 \parallel \cdots \parallel P_N$ ($P_i \in \{0, 1\}^n$)

1 **for** $i \leftarrow 1$ **to** N **do**   1 **for** $i \leftarrow 1$ **to** N **do**
2     $C_i \leftarrow \mathsf{EncryptBlk}(K, P_i)$;   2 $P_i \leftarrow \mathsf{DecryptBlk}(K, C_i)$;
3 **end**   3 **end**
4 **return** $C = C_1 \parallel \cdots \parallel C_N$;   4 **return** $C = C_1 \parallel \cdots \parallel C_N$;
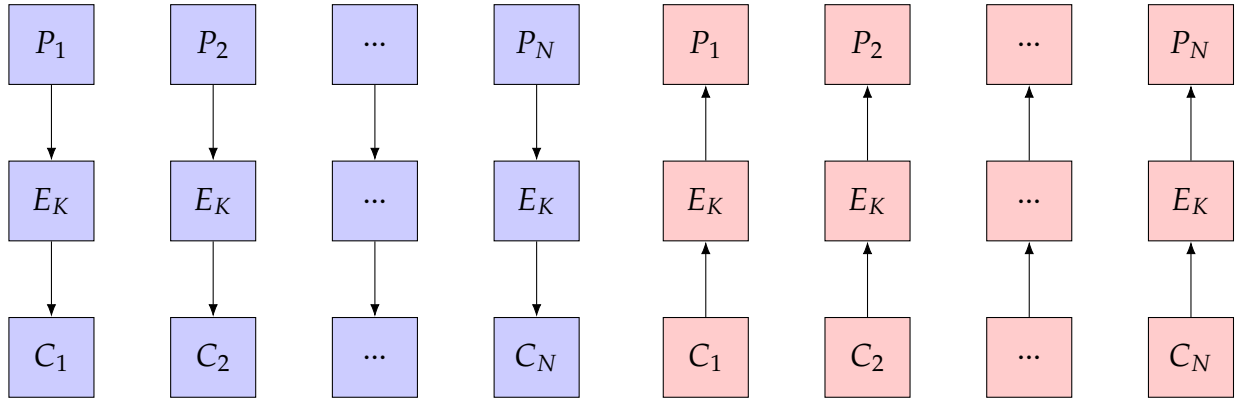
Remark 1.1.

(1) For a pair of plaintext/ciphertext (P, C) and (P', C'),

$$P = P' \implies C = C'.$$

(2) A single-bit error in the ciphertext affects only the corresponding block in the decryption.

(3) Block order changes, as well as additions/deletions, are possible. Therefore, to ensure the integrity of the ciphertext (i.e., detection or prevention of tampering), it should be used in conjunction with a checksum or a Message Authentication Code (MAC).

1.2.3 CBC (Cipher Block Chaining)

Algorithm 2: Cipher Block Chaining

Input: K, IV and $P = P_1 \parallel \cdots \parallel P_N$
Output: $C = C_1 \parallel \cdots \parallel C_N$

1 $C_0 \leftarrow IV$;
2 **for** $i \leftarrow 1$ **to** N **do**
3 $\quad \mid \quad C_i \leftarrow \mathsf{EncryptBlk}(K, P_i \oplus C_{i-1})$;
4 **end**
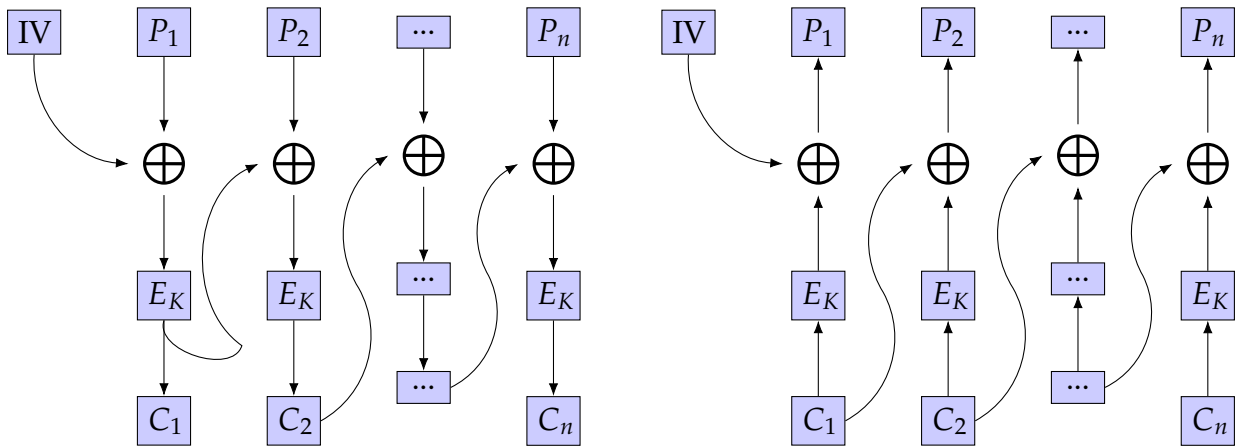5 **return** $C = C_1 \parallel \cdots \parallel C_N$;

Input: K, IV and $C = C_1 \parallel \cdots \parallel C_N$
Output: $P = P_1 \parallel \cdots \parallel P_N$

1 $C_0 \leftarrow IV$;
2 **for** $i \leftarrow 1$ **to** N **do**
3 $\quad \mid \quad P_i \leftarrow C_{i-1} \oplus \mathsf{DecryptBlk}(K, C_i)$;
4 **end**
5 **return** $P = P_1 \parallel \cdots \parallel P_N$;
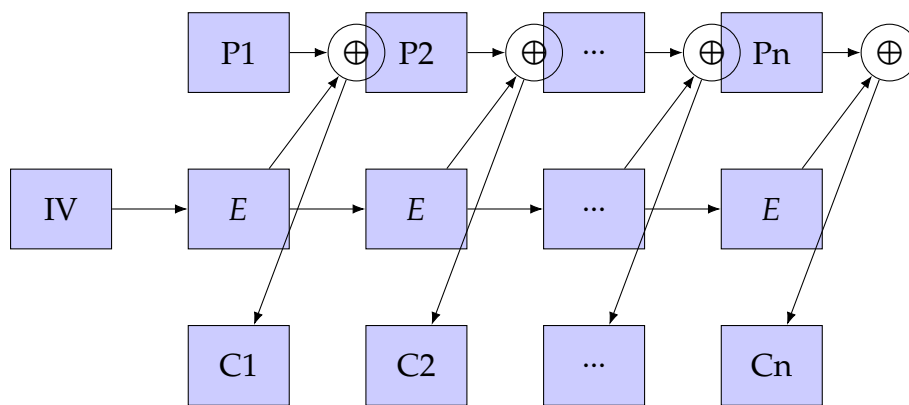

Remark 1.2.

(1) For a set of plaintext/ciphertext pairs (IV, P, C) and (IV', P', C'),
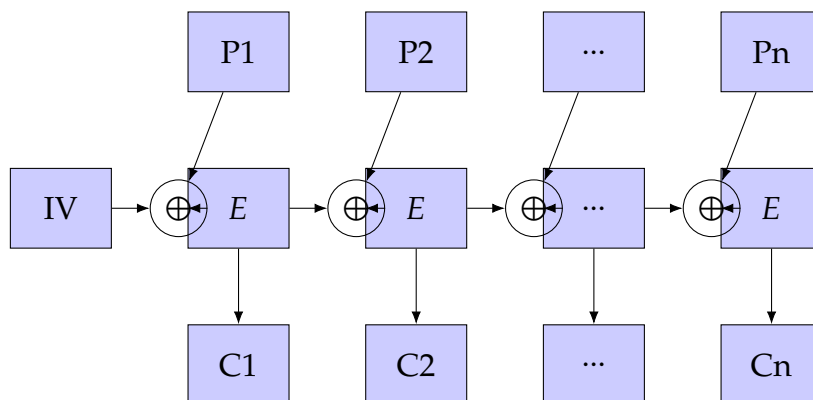
$$IV = IV' \wedge P = P' \implies C = C'.$$

(2) A one-bit error in the ciphertext will affect the corresponding block and the subsequent block in the decryption process, thereby enabling the detection of such errors.

(3) Altering the order of blocks, as well as adding or deleting them, is not possible.
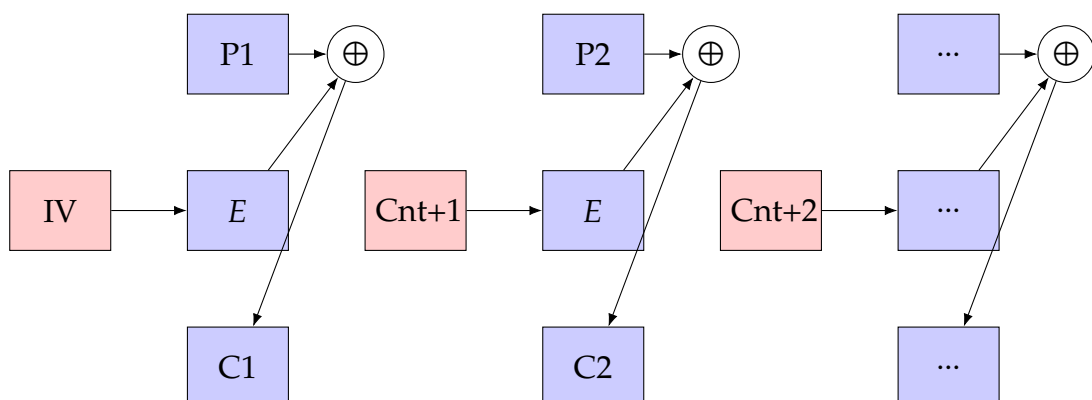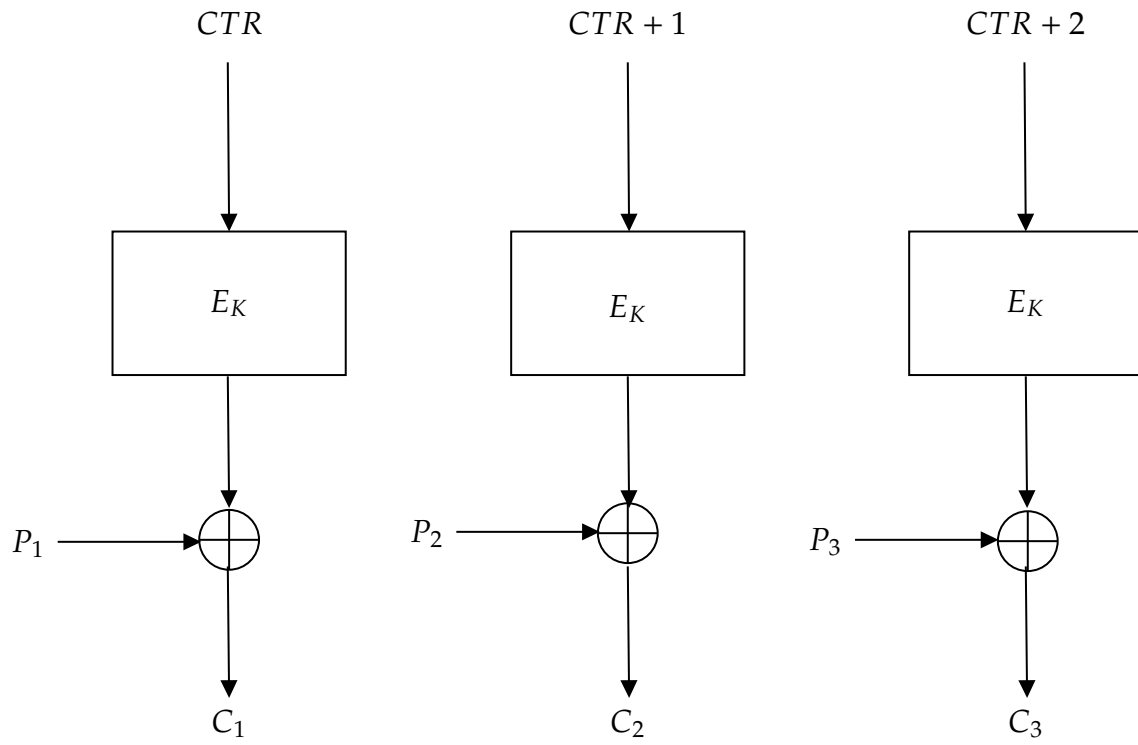
1.2.4 OFB **(Output FeedBack)**

1.2.5 CFB **(Ciphertext FeedBack)**

1.2.6 CTR **(CounTeR)**

1.2.7   CBC − CS (Ciphertext Stealing)
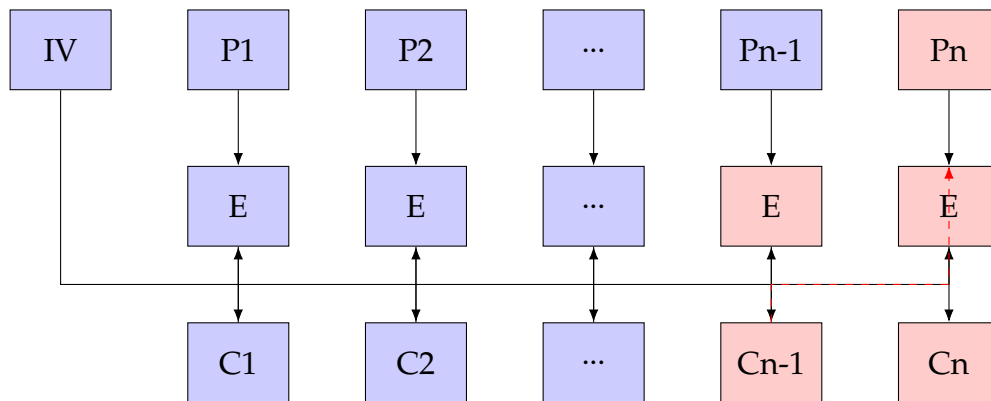
Chapter 2

AES Algorithm

2.1 Number Theory

2.1.1 Euclidean Algorithm

Let $a, b \in \mathbb{N}$. Then $\exists! q, r \in \mathbb{N}$ s.t. $a = bq + r \quad (0 \leq r < b)$. Then

$$\gcd(a, b) = \gcd(b, a \bmod b) = \gcd(b, r).$$

Example 2.1. Find $\gcd(90, 63)$.

Sol.

$$
\begin{aligned}
a &= b \cdot q + r \\
90 &= 63 \cdot 1 + 27 & \gcd(90, 63) &= \gcd(63, 27) \\
63 &= 27 \cdot 2 + 9 & \gcd(63, 27) &= \gcd(27, 9) \\
27 &= 9 \cdot 3 + 0 & \gcd(27, 9) &= 9.
\end{aligned}
$$

$\square$

```
1  u32 gcd(u32 a, u32 b) {
2      u32 r;
3
4      while (b != 0) {
5          r = a % b;
6          a = b;
7          b = r;
8      }
9
10     return a;
11 }
12
13 u32 rec_gcd(u32 a, u32 b) {
14     return (b == 0) ? a : rec_gcd(b, a % b);
15 }
```

2.1.2   Extended Euclidean Algorithm (EEA)

There exists $x, y \in \mathbb{Z}$ such that $ax + by = \gcd(x, y)$.

$a = bq + r$	a	b
$90 = 63 \cdot 1 + 27$	$90 = 1 \cdot 90 + 0 \cdot 63$	$63 = 0 \cdot 90 + 1 \cdot 63$
$63 = 27 \cdot 2 + 9$	$63 = 0 \cdot 90 + 1 \cdot 63$	$27 = 90 - 63 = 1 \cdot 90 + (-1) \cdot 63$
$27 = 9 \cdot 3 + 0$	$27 = 1 \cdot 90 + (-1) \cdot 63$	$9 = 63 - 27 \cdot 2$

Here,

$$9 = 63 - 27 \cdot 2 = 63 - (90 + (-1) \cdot 63) \cdot 2$$
$$= (1 + 2) \cdot 63 + (-2) \cdot 90$$
$$= (3) \cdot 63 + (-2) \cdot 90.$$

Let $\gcd(a, b) = xa + yb$ with $a = u_a \cdot a + v_a \cdot b$ and $b = u_b \cdot a + v_b \cdot b$. Consider

$$a = b \cdot q + r,$$
$$a' = b' \cdot q' + r'$$

with $\gcd(a, b) = \gcd(b, r) = \gcd(a', b')$. Then

$$a' = b = u_b \cdot a + v_b \cdot b = u_a' \cdot a + v_a' \cdot b,$$
$$b' = r = a - bq = (u_a \cdot a + v_a \cdot b) - (u_b \cdot a + v_b \cdot b) \cdot q$$
$$= (u_a - u_b \cdot q)a + (v_a - v_b \cdot q)b$$
$$= u_b \cdot' a + v_b \cdot' b.$$

Thus,

$$u_a' = u_b, \qquad\qquad\qquad v_a' = v_b$$
$$u_b' = u_a - u_b q, \qquad\qquad\qquad v_b' = v_a - v_b \cdot q.$$

```
1  u32 eea(u32 a, u32 b, u32* x, u32* y) {
2      u32 an, bn; u32 ua, va, ub, vb; u32 new_an, new_bn;
3      u32 n_ua, n_va, n_ub, n_vb; u32 q;
4
5      an = a; bn = b;
6      ua = 1; va = 0; ub = 0; vb = 1;
7      while (bn != 0) {
8          q = an / bn;
9          new_an = bn;
10         new_bn = an - bn * q;
11
12         n_ua = ub; n_va = vb;
13         n_ub = ua - ub * q; n_vb = va - vb * q;
14
15         an = new_an; bn = new_bn;
16         ua = n_ua; va = n_va; ub = n_ub; vb = n_vb;
17     }
18     *x = ua; *y = va;
19     return an;
20 }
```

2.2 Arrays

```
u8 a[4] = { 0xaa, 0xbb, 0xcc, 0xdd };
```

```
--- Memory -----------------------------------------------------
0x00007fffffffd294   aa bb cc dd 00 d1 92 71 77 09 82 87 01 00 00 00
0x00007fffffffd2a4   00 00 00 00 90 cd d9 f7 ff 7f 00 00 00 00 00 00
----------------------------------------------------------------
>>> x/4xb a
0x7fffffffd294: 0xaa      0xbb      0xcc      0xdd
>>> x/4xb a + 1
0x7fffffffd295: 0xbb      0xcc      0xdd      0x00
>>> x/4xb a + 2
0x7fffffffd296: 0xcc      0xdd      0x00      0xed
>>> x/4xb a + 3
0x7fffffffd297: 0xdd      0x00      0xed      0x81
>>> print /x *a
$1 = 0xaa
>>> print /x *(a + 1)
$2 = 0xbb
>>> print /x *(a + 2)
$3 = 0xcc
>>> print /x *(a + 3)
$4 = 0xdd
```

Variables	a[0]	a[1]	a[2]	a[3]
Data	0xaa	0xbb	0xcc	0xdd
Real Address	0x7fffffffd294	0x7fffffffd295	0x7fffffffd296	0x7fffffffd297
Symbolic Address	a	a + 1	a + 2	a + 3
Variables2	*a	*(a + 1)	*(a + 2)	*(a + 3)
Address2	&a[0]	&a[1]	&a[2]	&a[3]

```
u8 a[2][3] = { 0x00, 0x11, 0x22, 0x33, 0x44, 0x55 };
u8 a[2][3] = { {0x00, 0x11, 0x22}, {0x33, 0x44, 0x55} };
```

```
--- Memory ------------------------------------------------------
0x00007fffffffd292   00 11 22 33 44 55 00 0d d5 00 1a 96 86 93 01 00
0x00007fffffffd2a2   00 00 00 00 00 00 90 cd d9 f7 ff 7f 00 00 00 00
-----------------------------------------------------------------
>>> x/6xb a
0x7fffffffd292: 0x00      0x11      0x22      0x33      0x44      0x55
```

Variables	a[0][0]	a[0][1]	a[0][2]	a[1][0]	a[1][1]	a[1][2]
Data	0x00	0x11	0x22	0x33	0x44	0x55
Real Address	0x7fff--d292	0x7fff--d293	0x7fff--d294	0x7fff--d295	0x7fff--d296	0x7fff--d297
Symbolic Address	*a	*a + 1	*a + 2	*(a + 1)	*(a + 1) + 1	*(a + 1) + 2
Variables2	*(*(a + 0) + 0)	*(*(a + 0) + 1)	*(*(a + 0) + 2)	*(*(a + 1) + 0)	*(*(a + 1) + 1)	*(*(a + 1) + 2)
Address2	&a[0][0]	&a[0][1]	&a[0][2]	&a[1][0]	&a[1][1]	&a[1][2]

```
--- Memory ------------------------------------------------------
0x00007fffffffd292   00 11 22 33 44 55 00 0d d5 00 1a 96 86 93 01 00
0x00007fffffffd2a2   00 00 00 00 00 00 90 cd d9 f7 ff 7f 00 00 00 00
-----------------------------------------------------------------
>>> x/6xb a
0x7fffffffd292: 0x00      0x11      0x22      0x33      0x44      0x55
>>> x/6xb *a
0x7fffffffd292: 0x00      0x11      0x22      0x33      0x44      0x55
>>> x/6xb *a + 1
0x7fffffffd293: 0x11      0x22      0x33      0x44      0x55      0x00
>>> x/6xb *a + 2
0x7fffffffd294: 0x22      0x33      0x44      0x55      0x00      0x0d
>>> x/6xb *(a + 1)
0x7fffffffd295: 0x33      0x44      0x55      0x00      0x0d      0xd5
>>> x/6xb *(a + 1) + 1
0x7fffffffd296: 0x44      0x55      0x00      0x0d      0xd5      0x00
>>> x/6xb *(a + 1) + 2
0x7fffffffd297: 0x55      0x00      0x0d      0xd5      0x00      0x1a
>>> print /x **a
$1 = 0x0
>>> print /x *(*a + 1)
$2 = 0x11
>>> print /x *(*a + 2)
$3 = 0x22
>>> print /x *(*(a + 1))
$4 = 0x33
>>> print /x *(*(a + 1) + 1)
$5 = 0x44
>>> print /x *(*(a + 1) + 2)
$6 = 0x55
```

2.3 S-Box ($GF(2^8)$)

Let $m(x) := x^8 + x^4 + x^3 + x + 1$.

$$
\begin{aligned}
GF(2^8) &= GF(2)[x]/\langle x^8 + x^4 + x^3 + x + 1 \rangle \\
&= GF(2)[x]/\langle m(x) \rangle \\
&= \left\{ b_0 + b_1 x + \cdots + b_6 x^6 + b_7 x^7 : b_0, b_1, \cdots, b_6, b_7 \in GF(2) \right\} \\
&= \left\{ \sum_{i=0}^{7} b_i x^i : b_i \in \{0, 1\} \right\}.
\end{aligned}
$$

Let $f(x), g(x) \in GF(2^8)$, say,

$$
f(x) = \sum_{i=0}^{7} a_i x^i + \langle m(x) \rangle, \quad g(x) = \sum_{j=0}^{7} b_j x^j + \langle m(x) \rangle.
$$

Then

$$
f(x) + g(x) = \sum_{i=0}^{7} (a_i + b_i) x^i + \langle m(x) \rangle = \sum_{i=0}^{7} (a_i \oplus b_i) x^i + \langle m(x) \rangle.
$$

Here, $\oplus : GF(2) \times GF(2) \to GF(2)$. And

$$
f(x) \cdot g(x) = m(x) \cdot q(x) + r(x).
$$

Here $\deg r \leq 7$.

We define

$$
\begin{aligned}
\texttt{xtime} \ : \ GF(2^8) &\longrightarrow GF(2^8) \\
f(x) &\longmapsto x \cdot f(x)
\end{aligned}.
$$

Then

1. $f(x) = a_0 + a_1 x + \cdots + a_6 x^6 + a_7 x^7$

2.

$$
\begin{aligned}
x \cdot f(x) &= a_0 x + a_1 x^2 + \cdots + a_6 x^7 + a_7 x^8 \\
&= a_0 x + a_1 x^2 + \cdots + a_6 x^7 + a_7 (x^4 + x^3 + x + 1) \quad \because x^8 \equiv x^4 + x^3 + x + 1 \pmod{m(x)} \\
&= \begin{cases} \sum_{i=0}^{6} a_i x^{i+1} + 0 & : a_7 = 0 \\ \sum_{i=0}^{6} a_i x^{i+1} + (x^4 + x^3 + x + 1) & : a_7 = 1 \end{cases} \\
&= \begin{cases} (f(x) \ll 1) & : a_7 = 0 \\ (f(x) \ll 1) \oplus \texttt{0x1b} & : a_7 = 1 \end{cases}
\end{aligned}
$$

```
1  u8 xtime(u8 f) {
2      return (f & 0x80) ? (f << 1) ^ 0x1b : f << 1;
3  }
```

$$g(x) \cdot f(x) = \sum_{i=0}^{7} g(x)a_i x^i$$

$$= g(x)a_0 + \sum_{i=1}^{7} g(x)a_i x^i$$

$$= g(x)a_0 + x \sum_{i=1}^{7} g(x)a_i x^{i-1}$$

$$= g(x)a_0 + x \left(g(x)a_1 + \sum_{i=2}^{7} g(x)a_i x^{i-2} \right)$$

$$= g(x)a_0 + x \left(g(x)a_1 + x \left(g(x)a_2 + \sum_{i=3}^{7} g(x)a_i x^{i-3} \right) \right)$$

$$= \cdots$$

$$= g(x)a_0 + x(g(x)a_1 + x(g(x)a_2 + x(g(x)a_3 + x(g(x)a_4 + x(g(x)a_5 + x(g(x)a_6 + x(g(x)a_7)))))))$$

That is,

x	$\cdot 0$	$+(g(x) \cdot a_7)$
x	$\cdot (g(x) \cdot a_7)$	$+(g(x) \cdot a_6)$
x	$\cdot (x \cdot (g(x) \cdot a_7) + (g(x) \cdot a_6))$	$+(g(x) \cdot a_5)$

Step 1. $x \cdot 0 + g(x) \cdot a_7$

Step 2. $x \cdot (g(x) \cdot a_7) + (g(x) \cdot a_6)$

Step 3. $x \cdot (x \cdot (g(x) \cdot a_7) + (g(x) \cdot a_6)) + (g(x) \cdot a_5)$

$\cdots$

Final. $g \cdot a_0 + x(g \cdot a_1 + \cdots + x(g \cdot a_5 + x(g \cdot a_6 + x(g \cdot a_7))))$

```
1  u8 GF256_mul(u8 f, u8 g) {
2      u8 h = 0x00; // h = f * g
3      u8 coef;
4
5      for (i8 i = 7; i >= 0; i--) {
6          coef = (f >> i) & 0x01; // Get the coefficient f_i
7          h = GF256_xtime(h);     // Multiply h by x
8          if (coef == 1) h ^= g;  // If f_i = 1, add g(x) to h(x)
9      }
10     return h;
11 }
```

2.4 MixColumns $(GF(2^8)[x]/\langle x^4+1 \rangle)$

$$
\text{MixColumns} \quad : \quad \begin{aligned} \{0,1\}^{128} &\longrightarrow \{0,1\}^{128} \\ \sum_{i=0}^{127} a_i x^i &\longmapsto \sum_{j=0}^{127} b_j x^j \end{aligned}
$$

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}

$$\Downarrow$$

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}'

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}

$\Longrightarrow$

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}'

Consider

$$
GF(2^8)[x]/\langle x^4+1 \rangle = \left\{ a_0 + a_1 x + a_2 x^3 + a_3 x^3 : a_i \in GF(2^8) \right\}.
$$

Note that $x^5 = (x^4+1)x + x \equiv x \pmod{\langle x^4+1 \rangle}$. We choose

$$
a(x) = (\texttt{0x03}) \cdot x^3 + (\texttt{0x01}) \cdot x^2 + (\texttt{0x01}) \cdot x + (\texttt{0x02}) \in GF(2^8)[x]/\langle x^4+1 \rangle,
$$

where

$$
\begin{aligned}
\texttt{0x01} = \texttt{0b 0000 0001} &= 1, \\
\texttt{0x02} = \texttt{0b 0000 0010} &= x, \\
\texttt{0x03} = \texttt{0b 0000 0011} &= x+1.
\end{aligned}
$$

Let $b(x) = b_0 + b_1 x + b_2 x^2 + b_3 x^3$, $c(x) = c_0 + c_1 x + c_2 x^2 + c_3 x^3$, and let

$$
\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} \mapsto \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} a_0 \cdot b_0 \bmod x^4+1 \\ a_1 \cdot b_1 \bmod x^4+1 \\ a_2 \cdot b_2 \bmod x^4+1 \\ a_3 \cdot b_3 \bmod x^4+1 \end{pmatrix}.
$$

Then

$$a(x) \cdot b(x) = (a_3 \cdot b_3)\,x^6 + (a_3 \cdot b_2 + a_2 \cdot b_3)\,x^5$$
$$+ (a_3 \cdot b_1 + a_2 \cdot b_2 + a_1 \cdot b_3)\,x^4$$
$$+ (a_3 \cdot b_0 + a_2 \cdot b_1 + a_1 \cdot b_2 + a_0 \cdot b_3)x^3$$
$$+ (a_2 \cdot b_0 + a_1 \cdot b_1 + a_0 \cdot b_2)x^2$$
$$+ (a_1 \cdot b_0 + a_0 \cdot b_1)x$$
$$+ (a_0 \cdot b_0)$$

$$= (a_3 \cdot b_3)\,x^2 + (a_3 \cdot b_2 + a_2 \cdot b_3)\,x$$
$$+ (a_3 \cdot b_1 + a_2 \cdot b_2 + a_1 \cdot b_3)\,1$$
$$+ (a_3 \cdot b_0 + a_2 \cdot b_1 + a_1 \cdot b_2 + a_0 \cdot b_3)x^3$$
$$+ (a_2 \cdot b_0 + a_1 \cdot b_1 + a_0 \cdot b_2)x^2$$
$$+ (a_1 \cdot b_0 + a_0 \cdot b_1)x$$
$$+ (a_0 \cdot b_0) \quad \text{over} \quad GF(2^8)[x]/\langle x^4 + 1\rangle$$

$$= (a_3 \cdot b_0 + a_2 \cdot b_1 + a_1 \cdot b_2 + a_0 \cdot b_3)x^3$$
$$= (a_2 \cdot b_0 + a_1 \cdot b_1 + a_0 \cdot b_2 + a_3 \cdot b_3)x^2$$
$$= (a_1 \cdot b_0 + a_0 \cdot b_1 + a_3 \cdot b_2 + a_2 \cdot b_3)x$$
$$= (a_0 \cdot b_0 + a_3 \cdot b_1 + a_2 \cdot b_2 + a_1 \cdot b_3)$$

$$=c(x).$$

Thus, we have

$$T \;:\; [GF(2^8)]^4 \;\longrightarrow\; [GF(2^8)]^4$$
$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} \longmapsto \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} a_3 & a_2 & a_1 & a_0 \\ a_2 & a_1 & a_0 & a_3 \\ a_1 & a_0 & a_3 & a_2 \\ a_0 & a_3 & a_2 & a_1 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix},$$

where

$$\begin{pmatrix} a_3 & a_2 & a_1 & a_0 \\ a_2 & a_1 & a_0 & a_3 \\ a_1 & a_0 & a_3 & a_2 \\ a_0 & a_3 & a_2 & a_1 \end{pmatrix} = \begin{pmatrix} \texttt{0x02} & \texttt{0x03} & \texttt{0x01} & \texttt{0x01} \\ \texttt{0x01} & \texttt{0x02} & \texttt{0x03} & \texttt{0x01} \\ \texttt{0x01} & \texttt{0x01} & \texttt{0x02} & \texttt{0x03} \\ \texttt{0x03} & \texttt{0x01} & \texttt{0x01} & \texttt{0x02} \end{pmatrix}.$$

Here, T has GF-MUL $\times$ 16.

Let $a \in \{0, 1\}^8$. Then

$$02 \otimes a = \texttt{xtime}\,(a),$$
$$03 \otimes a = (02 \oplus 01) \otimes a$$
$$= (02 \otimes a) \oplus (01 \otimes a)$$
$$= \texttt{xtime}\,(a) \oplus a.$$

Thus,

$$
\begin{aligned}
c_0 &= (\texttt{02} \otimes b_0) \oplus (\texttt{03} \otimes b_1) \oplus b_2 \oplus b_3 \\
&= \texttt{xtime}\,(b_0) \oplus \texttt{xtime}\,(b_1) \oplus b_1 \oplus b_2 \oplus b_3 \\
&= \texttt{xtime}\,(b_0 \oplus b_1) \oplus b_1 \oplus b_2 \oplus b_3 \quad \because ax + ab = x(a+b) \\
&= \texttt{temp0} \oplus b_1 \oplus b_2 \oplus b_3.
\end{aligned}
$$

and so

$$c_0 = (b_0 \oplus b_1 \oplus b_2 \oplus b_3) \oplus \texttt{temp0} \oplus b_0 \qquad \text{with } \texttt{temp0} = \texttt{xtime}\,(b_0 \oplus b_1),$$

$$c_1 = (b_0 \oplus b_1 \oplus b_2 \oplus b_3) \oplus \texttt{temp1} \oplus b_1 \qquad \text{with } \texttt{temp1} = \texttt{xtime}\,(b_1 \oplus b_2),$$

$$c_2 = (b_0 \oplus b_1 \oplus b_2 \oplus b_3) \oplus \texttt{temp2} \oplus b_2 \qquad \text{with } \texttt{temp2} = \texttt{xtime}\,(b_2 \oplus b_3),$$

$$c_3 = (b_0 \oplus b_1 \oplus b_2 \oplus b_3) \oplus \texttt{temp3} \oplus b_3 \qquad \text{with } \texttt{temp3} = \texttt{xtime}\,(b_3 \oplus b_0).$$

That is,

```
1  sum ← b₀ ⊕ b₁ ⊕ b₂ ⊕ b₃;
2  c₀ ← sum ⊕ temp0 ⊕ b₀;
3  c₀ ← sum ⊕ temp1 ⊕ b₁;
4  c₀ ← sum ⊕ temp2 ⊕ b₂;
5  c₀ ← sum ⊕ temp3 ⊕ b₃;
```

It has $\texttt{xtime}\,() \times 4$. Consider

$$
\begin{aligned}
[a(x)]^1 &= a_3 x^3 + a_2 x^2 + a_1 x + a_0, \\
[a(x)]^2 &= (a_3^2 + a_1^2)x^2 + (a_2^2 + a_0^2), \\
[a(x)]^4 &= a_3^4 + a_2^4 + a_1^4 + a_0^4.
\end{aligned}
$$

Let $a(x) = (\texttt{03})x^3 + x^2 + x + (\texttt{02})$. Then

$$
\begin{aligned}
[a(x)]^2 &= (\texttt{04})x^2 + (\texttt{05}), \\
[a(x)]^3 &= (\texttt{0b})x^3 + (\texttt{0d})x^2 + (\texttt{09})x + (\texttt{0e}),
\end{aligned}
$$

and so $[a(x)]^{-1} = [a(x)]^3 = (\texttt{0b})x^3 + (\texttt{0d})x^2 + (\texttt{09})x + (\texttt{0e})$.

2.5   Little and Big Endian

2.5.1   Introduction

```
1 u8 b[4] = { 0x00, 0x01, 0x02, 0x03 };
2 u32 x = 0x00010203;
```

```
>>> x/4xb b
0x7fffffffd294:  0x00     0x01     0x02     0x03
>>> x/4xb &x
0x7fffffffd290:  0x03     0x02     0x01     0x00
```

$[u32 \rightarrow (u8 \times 4)]:$ $[(u8 \times 4) \rightarrow u32]:$

b[0] = (x ≫ 0x18)   x = (b[0] ≪ 0x18) ^

b[1] = (x ≫ 0x10) ^ 0xff   (b[1] ≪ 0x10) ^

b[2] = (x ≫ 0x08) ^ 0xff   (b[2] ≪ 0x08) ^

b[3] = (x   ) ^ 0xff   (b[3]   )

b ⟺ &(b[0]) ⟺ (u8*)&b[0]

```
1 u8 b[4] = { 0x00, 0x01, 0x02, 0x03 };
2 u32* pInt;
3 u32 x;
4
5 pInt = (u32*)b;
6 x = *pInt;
```

```
>>> x/4xb b
0x7fffffffd294:  0x00     0x01     0x02     0x03
>>> x/4xb pInt
0x7fffffffd294:  0x00     0x01     0x02     0x03
>>> x/4xb &x
0x7fffffffd284:  0x00     0x01     0x02     0x03
>>> print /x x
$1 = 0x3020100
```

$$u8\ b[4]\ =\ \{\ 0x00,\ 0x01,\ 0x02,\ 0x03\ \};\ \Longrightarrow\ \begin{cases}u32\ x\ =\ 0x03020100; \\ u32\ y\ =\ 0x00010203;\end{cases}$$

```
1  u8 b[4] = { 0x00, 0x01, 0x02, 0x03 };
2  u32* px = (u32*)b;
3  u32 x, y;
4
5  x = *px;
6  y = ((u32)b[0] << 0x18) ^
7      ((u32)b[1] << 0x10) ^
8      ((u32)b[2] << 0x08) ^
9      ((u32)b[3]         );
```
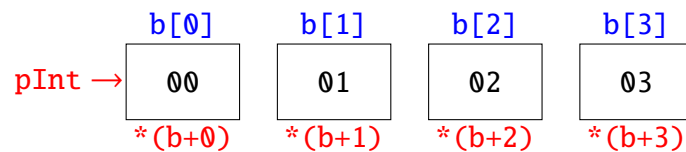
```
>>> x/4xb b
0x7fffffffd294: 0x00    0x01    0x02    0x03
>>> x/4xb px
0x7fffffffd294: 0x00    0x01    0x02    0x03
>>> x /4xb &x
0x7fffffffd280: 0x00    0x01    0x02    0x03
>>> x /4xb &y
0x7fffffffd284: 0x03    0x02    0x01    0x00
>>> x /wx 0x7fffffffd280
0x7fffffffd280: 0x03020100
>>> x /wx 0x7fffffffd284
0x7fffffffd284: 0x00010203
```

2.5.2 Marco

```
1  #define GETU32(pt) ((u32)(pt)[0] << 0x18) ^ \
2                     ((u32)(pt)[1] << 0x10) ^ \
3                     ((u32)(pt)[2] << 0x08) ^ \
4                     ((u32)(pt)[3]         )
5
6  u8 b[4] = { 0x00, 0x01, 0x02, 0x03 };
7  u32 y;
8
9  y = GETU32(b);
```

```
>>> x/4xb b
0x7fffffffd294: 0x00    0x01    0x02    0x03
>>> x/4xb &y
0x7fffffffd290: 0x03    0x02    0x01    0x00
```

Byte to Word
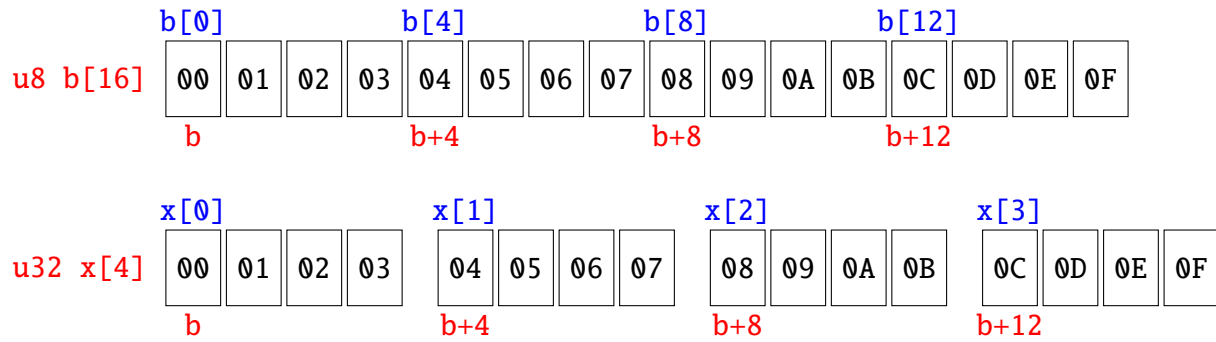

```
1 u8 b[16] = { 0x00, 0x01, 0x02, 0x03,
2             0x04, 0x05, 0x06, 0x07,
3             0x08, 0x09, 0x0A, 0x0B,
4             0x0C, 0x0D, 0x0E, 0x0F};
5
6 u32 x[4] = { GETU32(b        ),
7             GETU32(b + 0x04),
8             GETU32(b + 0x08),
9             GETU32(b + 0x0C)};
```

```
>>> x/16xb b
0x7fffffffd280: 0x00   0x01   0x02   0x03   0x04   0x05   0x06   0x07
0x7fffffffd288: 0x08   0x09   0x0a   0x0b   0x0c   0x0d   0x0e   0x0f
>>> x/16xb x
0x7fffffffd270: 0x03   0x02   0x01   0x00   0x07   0x06   0x05   0x04
0x7fffffffd278: 0x0b   0x0a   0x09   0x08   0x0f   0x0e   0x0d   0x0c
```

Word to Byte
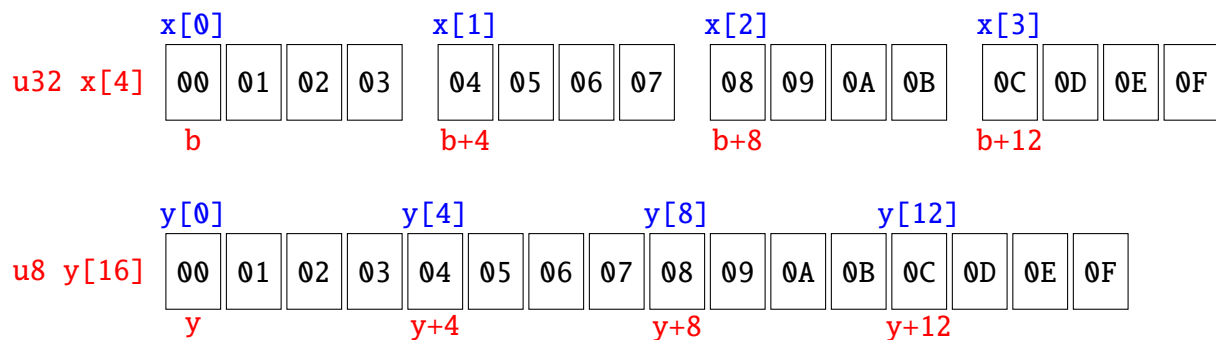
```
1  #define PUTU32(ct, st) { (ct)[0] = (u8)((st) >> 0x18); \
2                            (ct)[1] = (u8)((st) >> 0x10); \
3                            (ct)[2] = (u8)((st) >> 0x08); \
4                            (ct)[3] = (u8)((st)         ); }
5
6  u32 x[4] = { 0x00010203, 0x04050607, 0x08090A0B, 0x0C0D0E0F };
7  u8 y[16];
8
9  PUTU32(y       , x[0]);
10 PUTU32(y + 0x04, x[1]);
11 PUTU32(y + 0x08, x[2]);
12 PUTU32(y + 0x0C, x[3]);
```

```
>>> x/16xb x
0x7ffffffffd260: 0x03  0x02  0x01  0x00  0x07  0x06  0x05  0x04
0x7ffffffffd268: 0x0b  0x0a  0x09  0x08  0x0f  0x0e  0x0d  0x0c
>>> x/16xb y
0x7ffffffffd280: 0x00  0x01  0x02  0x03  0x04  0x05  0x06  0x07
0x7ffffffffd288: 0x08  0x09  0x0a  0x0b  0x0c  0x0d  0x0e  0x0f
```

2.6   Implementation of 32-bit AES
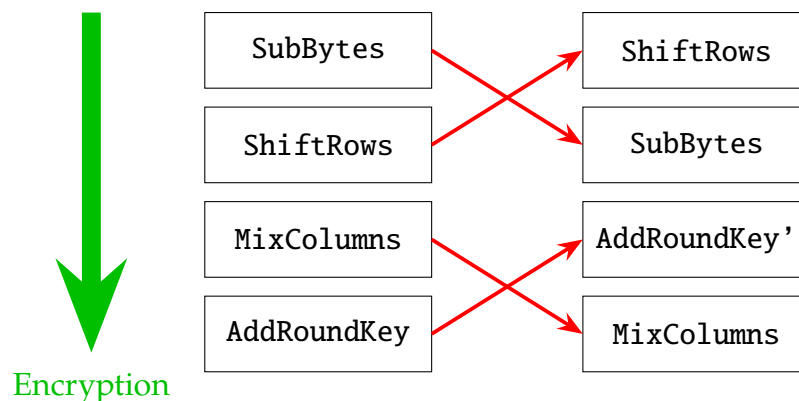
$$y = \texttt{MixCol}(x) \oplus \texttt{rk}$$
$$= \texttt{MixCol}(x \oplus \texttt{InvMixCol}(rk))$$
$$= \texttt{MixCol}(x \oplus \texttt{rk'})$$

	Key Length		Block Size		Number of Rounds
	Nk-word	(bit)	Nb-word	(bit)	Nr
AES-128	4-word	(128)	4-word	(128)	10
AES-192	6-word	(192)	4-word	(128)	10
AES-256	4-word	(256)	4-word	(128)	10

Algorithm 3: AES Encryption

---

1 Function Cipher *(in, Nr, w)***:**
2 $state \leftarrow in$
3 $state \leftarrow$ AddRoundKey(*state, w[0 .. 3]*)   `// w[0..3] = w[0], w[1], w[2], w[3]`
4   **for** *round* $\leftarrow 1$ **to** $Nr - 1$ **do**
5 $state \leftarrow$ SubBytes(*state*)
6 $state \leftarrow$ ShiftRows(*state*)
7 $state \leftarrow$ MixColumns(*state*)
8          $state \leftarrow$ AddRoundKey(*state, w[4*round .. 4*round + 3]*)
9   **end**
10 $state \leftarrow$ SubBytes(*state*)
11 $state \leftarrow$ ShiftRows(*state*)
12     $state \leftarrow$ AddRoundKey(*state, w[4*Nr .. 4*Nr + 3]*)
13   **return** $state$
14 end

Algorithm 4: AES Inverse Encryption

---

1 Function InvCipher *(in, Nr, w)***:**
2 $state \leftarrow in$
3      $state \leftarrow$ AddRoundKey(*state, w[4*Nr .. 4*Nr + 3]*)
4   **for** *round* $\leftarrow Nr - 1$ **downto** 1 **do**
5 $state \leftarrow$ InvShiftRows(*state*)
6 $state \leftarrow$ InvSubBytes(*state*)
7          $state \leftarrow$ AddRoundKey(*state, w[4*round .. 4*round + 3]*)
8 $state \leftarrow$ InvMixColumns(*state*)
9   **end**
10 $state \leftarrow$ InvShiftRows(*state*)
11 $state \leftarrow$ InvSubBytes(*state*)
12 $state \leftarrow$ AddRoundKey(*state, w[0 .. 3]*)
13   **return** $state$
14 end

Algorithm 5: AES Inverse Encryption 2

---

1 Function EqInvCipher *(in, Nr, w)***:**
2 $state \leftarrow in$
3      $state \leftarrow$ AddRoundKey(*state, w[4*Nr .. 4*Nr + 3]*)
4   **for** *round* $\leftarrow Nr - 1$ **downto** 1 **do**
5 $state \leftarrow$ InvSubBytes(*state*)
6 $state \leftarrow$ InvShiftRows(*state*)
7 $state \leftarrow$ InvMixColumns(*state*)
8          $state \leftarrow$ AddRoundKey(*state, dw[4*round .. 4*round + 3]*)
             `/* y = MixCol(x` $\oplus$ `rk') ` $\implies$ ` rk' = x ` $\oplus$ ` InvMixCol(y)`   `*/`
             `/* for` *round* $\leftarrow 1$ **to** $Nr - 1$ **do**
                 $i \leftarrow 4 * round$
                 $dw[i \,..\, i+3] \leftarrow$ InvMixColumns($dw[i \,..\, i+3]$)
             end

                                                          `*/`
9   **end**
10 $state \leftarrow$ InvSubBytes(*state*)
11 $state \leftarrow$ InvShiftRows(*state*)
12 $state \leftarrow$ AddRoundKey(*state, w[0 .. 3]*)
13   **return** $state$
14 end
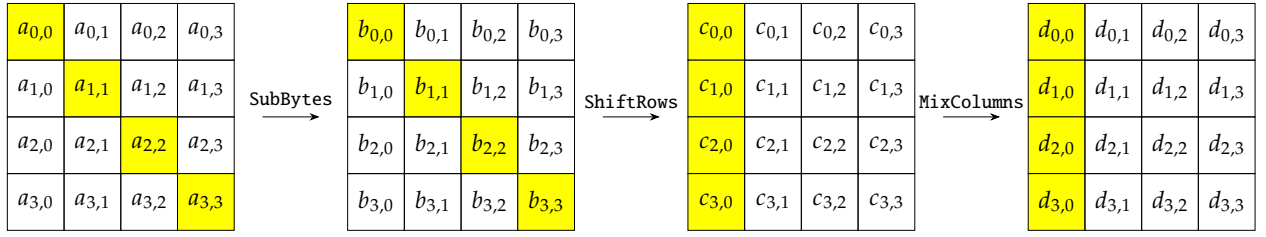
$$
\begin{pmatrix} c_{0,0} \\ c_{1,0} \\ c_{2,0} \\ c_{3,0} \end{pmatrix} = \begin{pmatrix} b_{0,0} \\ b_{1,1} \\ b_{2,2} \\ b_{3,3} \end{pmatrix} = \begin{pmatrix} S(a_{0,0}) \\ S(a_{1,1}) \\ S(a_{2,2}) \\ S(a_{3,3}) \end{pmatrix}
\qquad
\begin{pmatrix} d_{0,0} \\ d_{1,0} \\ d_{2,0} \\ d_{3,0} \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \begin{pmatrix} c_{0,0} \\ c_{1,0} \\ c_{2,0} \\ c_{3,0} \end{pmatrix}
$$

$$
\begin{pmatrix} d_{0,0} \\ d_{1,0} \\ d_{2,0} \\ d_{3,0} \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \begin{pmatrix} S(a_{0,0}) \\ S(a_{1,1}) \\ S(a_{2,2}) \\ S(a_{3,3}) \end{pmatrix} = S(a_{0,0}) \begin{pmatrix} 02 \\ 01 \\ 01 \\ 03 \end{pmatrix} \oplus S(a_{1,1}) \begin{pmatrix} 03 \\ 02 \\ 01 \\ 01 \end{pmatrix} \oplus S(a_{2,2}) \begin{pmatrix} 01 \\ 03 \\ 02 \\ 01 \end{pmatrix} \oplus S(a_{3,3}) \begin{pmatrix} 01 \\ 01 \\ 03 \\ 02 \end{pmatrix}
$$

$$
= Te0(a_{0,0}) \oplus Te1(a_{1,1}) \oplus Te2(a_{2,2}) \oplus Te3(a_{3,3})
$$

Let $Te0 : \mathbb{F}_2^8 \to \mathbb{F}_2^{32}$. Then,

$$
(2^{32})^{2^8} = 2^{32 \times 256} = 2^{8192} = 2^{8 \times 1024} = 1024\text{-byte.}
$$

$a_{0,0}$	$S(a_{0,0}) \begin{pmatrix} 02 & 01 & 01 & 03 \end{pmatrix}^T$
00	-- -- -- -- -- -- -- --
01	-- -- -- -- -- -- -- --
02	-- -- -- -- -- -- -- --
⋮	⋮
FF	-- -- -- -- -- -- -- --

Table 2.1: Transformation $S(a_{0,0})$ Matrix with Corresponding Values

Thus, $T0$ has a size of 1 KB (= 1024 bytes).

$d_{0,0}$	$d_{0,1}$	$d_{0,2}$	$d_{0,3}$
$d_{1,0}$	$d_{1,1}$	$d_{1,2}$	$d_{1,3}$
$d_{2,0}$	$d_{2,1}$	$d_{2,2}$	$d_{2,3}$
$d_{3,0}$	$d_{3,1}$	$d_{3,2}$	$d_{3,3}$

$\xrightarrow{\texttt{AddRoundKey}}$

$e_{0,0}$	$e_{0,1}$	$e_{0,2}$	$e_{0,3}$
$e_{1,0}$	$e_{1,1}$	$e_{1,2}$	$e_{1,3}$
$e_{2,0}$	$e_{2,1}$	$e_{2,2}$	$e_{2,3}$
$e_{3,0}$	$e_{3,1}$	$e_{3,2}$	$e_{3,3}$
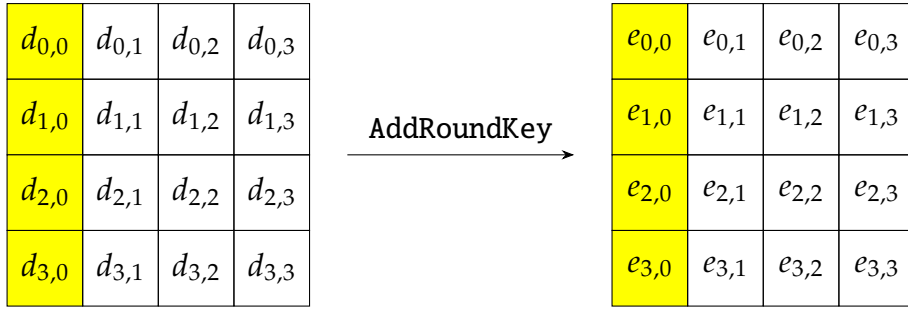
Here,

$$\begin{pmatrix} e_{0,0} \\ e_{1,0} \\ e_{2,0} \\ e_{3,0} \end{pmatrix} = Te0(a_{0,0}) \oplus Te1(a_{1,1}) \oplus Te2(a_{2,2}) \oplus Te3(a_{3,3}) \oplus \begin{pmatrix} rk_{0,0} \\ rk_{1,0} \\ rk_{2,0} \\ rk_{3,0} \end{pmatrix},$$

where

$$Te0(x) = \begin{pmatrix} \texttt{02} * S(x) \\ S(x) \\ S(x) \\ \texttt{03} * S(x) \end{pmatrix} \quad Te1(x) = \begin{pmatrix} \texttt{03} * S(x) \\ \texttt{02} * S(x) \\ S(x) \\ S(x) \end{pmatrix} \quad Te2(x) = \begin{pmatrix} S(x) \\ \texttt{03} * S(x) \\ \texttt{02} * S(x) \\ S(x) \end{pmatrix} \quad Te3(x) = \begin{pmatrix} S(x) \\ S(x) \\ \texttt{03} * S(x) \\ \texttt{02} * S(x) \end{pmatrix}.$$

2.7 Key Schedule

	Round Key Length	Number of Round Keys	Size of Round Key
AES-128	4-word (128-bit)	11 round	44-word (176-byte, 1408-bit)
AES-192	4-word (128-bit)	13 round	52-word (208-byte, 1664-bit)
AES-256	4-word (128-bit)	15 round	60-word (240-byte, 1920-bit)

Algorithm 6: Pseudocode for KeyExpansion

---

1 **Function** KeyExpansion *(key)*:
2     $i \leftarrow 0$
3   **while** $i \leq (Nk - 1)$ **do**
4         $w[i] \leftarrow key[4 * i .. 4 * i + 3]$
5         $i \leftarrow i + 1$
6   **end**
7   **while** $i \leq (4 * Nr + 3)$ **do**
8         $temp \leftarrow w[i - 1]$
9   **if** $(i \bmod Nk) = 0$ **then**
            /* RotWord($[a_0, a_1, a_2, a_3]$) = $[a_1, a_2, a_3, a_0]$ */
            /* SubWord($[a_0, a_1, a_2, a_3]$) = $[S(a_0), S(a_1), S(a_2), S(a_3)]$ */
10             $temp \leftarrow$ SubWord(RotWord($temp$)) $\oplus$ Rcon$[i/Nk]$
11   **end**
12   **else if** $Nk > 6$ *and* $(i \bmod Nk) = 4$ **then**
13             $temp \leftarrow$ SubWord($temp$)
14   **end**
15         $w[i] \leftarrow w[i - Nk] \oplus temp$
16         $i \leftarrow i + 1$
17   **end**
18   **return** w
19 **end**

Chapter 3

AES-128

3.1 Overview of AES-128

- KeyExpansion : $\{0, 1\}^{128} \rightarrow \{0, 1\}^{1408=4 \cdot (10+1) \cdot 32}$.

- AddRoundKey : $\{0, 1\}^{128} \times \{0, 1\}^{128} \rightarrow \{0, 1\}^{128}$.

- SubBytes/ShiftRows/MixColumns : $\{0, 1\}^{128} \rightarrow \{0, 1\}^{128}$.

Algorithm 7: 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 $\in \{0, 1\}^{128}$

1 $t \leftarrow$ AddRoundKey(src, rk_0);
2 **for** $i \leftarrow 1$ **to** 9 **do**
3 $\quad$ $t \leftarrow$ (MixColumns $\circ$ ShiftRows $\circ$ SubBytes)(t);
4 $\quad$ $t \leftarrow$ AddRoundKey(t, rk_i);
5 **end**
6 $t \leftarrow$ (ShiftRows $\circ$ SubBytes)(t);
7 $t \leftarrow$ AddRoundKey(t, rk_{10});
8 dst $\leftarrow t$;
9 **return** dst;

Algorithm 8: 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 $\in \{0, 1\}^{128}$

1 $t \leftarrow$ AddRoundKey(src, rk_{10});
2 **for** $i \leftarrow 9$ **to** 1 **do**
3 $\quad$ $t \leftarrow$ (InvSubBytes $\circ$ InvShiftRows)(t);
4 $\quad$ $t \leftarrow$ AddRoundKey(t, rk_i);
5 $\quad$ $t \leftarrow$ InvMixColumns(t);
6 **end**
7 $t \leftarrow$ (InvShiftRows $\circ$ InvSubBytes)(t);
8 $t \leftarrow$ AddRoundKey(t, rk_0);
9 dst $\leftarrow t$;
10 **return** dst;

3.2 Functions and Constants used in AES

3.2.1 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 3.1: RotWord rotates the input word left by one byte

```
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 3.2: SubWord applies the S-box to each byte of the input word

```
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 $\text{rCon}_i \in \mathbb{F}_{2^8}$ 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
rCon_i	0x01	0x02	0x04	0x08	0x10	0x20	0x40	0x80	0x1b	0x36

Code 3.3: rCon Array Declaration

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

Algorithm 9: Key Schedule (AES-128)

Input: User key $uk = (uk_0, \ldots, uk_{15})$ $(uk_i \in \{0, 1\}^8)$;   `// uk ∈ {0, 1}`^{128} `is 16-byte`

Output: round-keys $\{rk_i\}_{i=0}^{43}$ $(rk_i \in \{0, 1\}^{32})$;   `// {rk`$_i$`}`$_{i=0}^{43}$ `∈ {0, 1}`^{1408} `is 176-byte`

1 $rk_0 \leftarrow uk_0 \parallel uk_1 \parallel uk_2 \parallel uk_3$;

2 $rk_1 \leftarrow uk_4 \parallel uk_5 \parallel uk_6 \parallel uk_7$;

3 $rk_2 \leftarrow uk_8 \parallel uk_9 \parallel uk_{10} \parallel uk_{11}$;

4 $rk_3 \leftarrow uk_{12} \parallel uk_{13} \parallel uk_{14} \parallel uk_{15}$;

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          $t \leftarrow t \oplus (\mathrm{rCon}_{i/4} \parallel$ `0x00` $\parallel$ `0x00` $\parallel$ `0x00`$)$;

11   **end**

12      $rk_i \leftarrow rk_{i-4} \oplus_{32} t$;

13 **end**

Code 3.4: AES-128 Key Expansion

```
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++;
    }
}
```

3.2.2 AddRoundKey

- AddRoundKey : $\{0,1\}^{128} \times \{0,1\}^{128} \to \{0,1\}^{128}$ 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 3.5: AES AddRoundKey

```
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;
    }
}
```

3.2.3   SubBytes / InvSubBytes

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

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

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

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

Table 3.1: 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 3.6: Byte Substitution

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

Code 3.7: Inverse Byte Substitution

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

3.2.4 ShiftRows / InvShiftRows

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

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}

$\implies$

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

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

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}

$\implies$

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

Code 3.8: 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 3.9: Inverse 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;
}
```

3.2.5 MixColumns / InvMixColumns

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

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

Here,

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

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 $GF(2^8)$:

$$a(x) = a_7 x^7 + a_6 x^6 + a_5 x^5 + a_4 x^4 + a_3 x^3 + a_2 x^2 + a_1 x + a_0,$$
$$b(x) = b_7 x^7 + b_6 x^6 + b_5 x^5 + b_4 x^4 + b_3 x^3 + b_2 x^2 + b_1 x + b_0.$$

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.,

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

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

$$a_7 x^7 + \cdots + a_1 x + a_0 \iff (a_7 \ldots a_1 a_0)_2.$$

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 3.10: Multiplication in GF(2^8)

```
1  u8 MUL_GF256(u8 a, u8 b) {
2      u8 res = 0;
3      // Mask for detecting the MSB (0x80 = 0b10000000)
4      u8 MSB_mask = 0x80;
5      u8 MSB;
6      /*
7       * The reduction polynomial
8       * (x^8 + x^4 + x^3 + x + 1) = 0b100011011
9       * for AES, represented in hexadecimal
10      */
11     u8 modulo = 0x1B;
12
13     for (int i = 0; i < 8; i++) {
14         // Add a to result if LSB(b)=1
15         if (b & 1)
16             res ^= a;
17
18         MSB = a & MSB_mask; // Store the MSB of a
19         a <<= 1; // Multiplying it by x effectively
20
21         // Reduce the result modulo the reduction polynomial
22         if (MSB)
23             a ^= modulo;
24
25         b >>= 1; // Moving to the next bit
26     }
27
28     return res;
29 }
30
31 #define MUL_GF256(a, b) ({ \
32     u8 res = 0; \
33     u8 MSB_mask = 0x80; \
34     u8 MSB; \
35     u8 modulo = 0x1B; \
36     u8 temp_a = (a); \
37     u8 temp_b = (b); \
38     for (int i = 0; i < 8; i++) { \
39         if (temp_b & 1) \
40         res ^= temp_a; \
41         MSB = temp_a & MSB_mask; \
42         temp_a <<= 1; \
43         if (MSB) \
44         temp_a ^= modulo; \
45         temp_b >>= 1; \
46     } \
47     res; \
48 })
```

- MixColumns : $\{0,1\}^{128} \rightarrow \{0,1\}^{128}$ 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 : $\{0,1\}^{128} \rightarrow \{0,1\}^{128}$ 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 3.11: 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 3.12: Inverse MixColumns

```
1   void InvMixColumns(u8* state) {
2       u8 temp[4];
3
4       for (int i = 0; i < 4; i++) {
5           temp[0] =
6               MUL_GF256(0x0e, state[i * 4]) ^
7               MUL_GF256(0x0b, state[i * 4 + 1]) ^
8               MUL_GF256(0x0d, state[i * 4 + 2]) ^
9               MUL_GF256(0x09, state[i * 4 + 3]);
10
11          temp[1] =
12              MUL_GF256(0x09, state[i * 4]) ^
13              MUL_GF256(0x0e, state[i * 4 + 1]) ^
14              MUL_GF256(0x0b, state[i * 4 + 2]) ^
15              MUL_GF256(0x0d, state[i * 4 + 3]);
16
17          temp[2] =
18              MUL_GF256(0x0d, state[i * 4]) ^
19              MUL_GF256(0x09, state[i * 4 + 1]) ^
20              MUL_GF256(0x0e, state[i * 4 + 2]) ^
21              MUL_GF256(0x0b, state[i * 4 + 3]);
22
23          temp[3] =
24              MUL_GF256(0x0b, state[i * 4]) ^
25              MUL_GF256(0x0d, state[i * 4 + 1]) ^
26              MUL_GF256(0x09, state[i * 4 + 2]) ^
27              MUL_GF256(0x0e, state[i * 4 + 3]);
28
29          for (int j = 0; j < 4; j++)
30              state[i * 4 + j] = temp[j];
31      }
32  }
```

Chapter 4

AES - 128 / 192 / 256 (Byte Version)

4.1 Specification

Table 4.1: 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 4.1: Configuration in C

```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
#elif AES_VERSION == 192
    #define Nk 6
#elif AES_VERSION == 256
    #define Nk 8
#else
    #error "Invalid AES ky length"
#endif

#define Nr (Nk + 6) // 10 / 12 / 14
#define ROUND_KEYS_SIZE (16 * (Nr + 1)) // 176 / 208 / 240
```

Code 4.2: Configuration in Rust

```rust
1  // Define a constant for the AES key length.
2  pub const AES_VERSION: u32 = 128; // Can be 128, 192, or 256
3
4  // Define constant for AES block size
5  pub const AES_BLOCK_SIZE: usize = 16;
6
7  // Define constants Nk and Nr based on AES key length
8  pub const NK: usize = match AES_VERSION {
9      128 => 4,
10     192 => 6,
11     256 => 8,
12     _ => panic!("Invalid AES key length"),
13 };
14
15 pub const NR: usize = NK + 6;
16 pub const ROUND_KEYS_SIZE: usize = 16 * (NR + 1);
```

4.2   Key Expansion (General Version)

Algorithm 10: 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 $\quad$ $rk_i \leftarrow uk_{4i} \parallel uk_{4i+1} \parallel uk_{4i+2} \parallel uk_{4i+3}$;

4 **end**

5 **for** $i = l$ **to** $4(N_r + 1) - 1$ **do**

6 $\quad$ $t \leftarrow rk_{i-1}$;

7 $\quad$ **if** $i \bmod l = 0$ **then**

8 $\quad\quad$ $t \leftarrow (\text{SubWord} \circ \text{RotWord})(t)$;

9 $\quad\quad$ $t \leftarrow t \oplus_{32} (\text{rCon}_{i/l} \parallel \texttt{0x00} \parallel \texttt{0x00} \parallel \texttt{0x00})$;

10 $\quad$ **else if** $l > 6$ `&&` $i \bmod l = 4$ **then**

11 $\quad\quad$ $t \leftarrow \text{SubWord}(t)$;

12 $\quad$ **end**

13 $\quad$ $rk_i \leftarrow rk_{i-l} \oplus_{32} t$;

14 **end**

Code 4.3: Key Expansion in C (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 4.4: Key Expansion Test

```
1  void RANDOM_KEY_GENERATION(u8* key) {
2      srand((u32)time(NULL));
3
4      // Initialize pointer to the start of the key array
5      u8* p = key;
6
7      // Set the counter to 16 bytes
8      int cnt = 0;
9
10     // Loop until all 16 bytes are filled
11     while (cnt < AES_BLOCK_SIZE) {
12         *p = rand() & 0xff; // Assign a random byte (0 to 255)
13         p++;                    // Move to the next byte
14         cnt++;                  // Decrement the byte count
15     }
16 }
17
18 void KeyExpansionTest() {
19     u8 uKey[AES_BLOCK_SIZE] = { 0x00, };
20     RANDOM_KEY_GENERATION(uKey);
21     // u8 uKey[AES_BLOCK_SIZE] = {
22     //   0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6,
23     //   0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c
24     // };
25     for (int i = 0; i < AES_BLOCK_SIZE; i++) {
26         printf("%02x", uKey[i]);
27     } printf("\n");
28
29     u32 rKeys[ROUND_KEYS_SIZE / sizeof(u32)];
30     KeyExpansion(uKey, rKeys);
31     for (int i = 0; i < ROUND_KEYS_SIZE / sizeof(u32); i++) {
32         printf("%08x\n", rKeys[i]);
33     }
34 }
35
36 int main() {
37     KeyExpansionTest();
38     return 0;
39 }
```

4.3   8-bit AES - 128 / 192 / 256

Algorithm 11: 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 $\quad$ $t \leftarrow$ SubBytes(t);   // SubBytes: $\{0,1\}^{8*16} \rightarrow \{0,1\}^{8*16}$

4 $\quad$ $t \leftarrow$ ShiftRows(t);   // ShiftRows: $\{0,1\}^{8*16} \rightarrow \{0,1\}^{8*16}$

5 $\quad$ $t \leftarrow$ MixColumns(t);   // MixColumns: $\{0,1\}^{8*16} \rightarrow \{0,1\}^{8*16}$

6 $\quad$ $t \leftarrow$ 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 4.5: 8-bit AES Encryption

```
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 12: 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 4.6: 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];
}
```