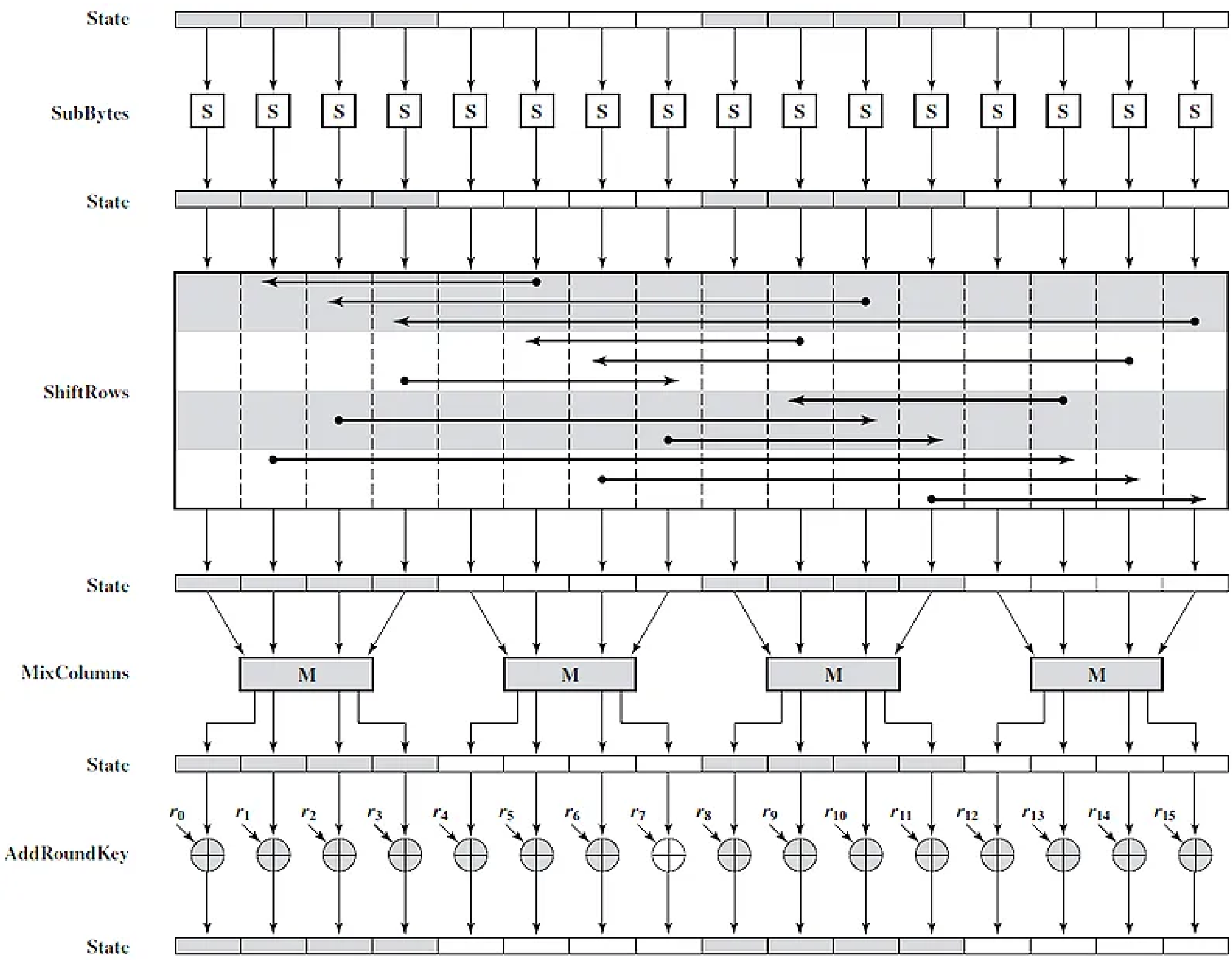
C | SecureAES

- High-Performance AES Encryption in C -

Ji Yong-Hyeon



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

December 10, 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	$\lvert P\rvert \overset{?}{=} \lvert C\rvert$
ECB	O	X	O	O	O	X	$\lvert P\rvert < \lvert C\rvert$
CBC	O	X	O	O	O	O	$\lvert P\rvert < \lvert C\rvert$
OFB	O	X	O	X	X	O	$\lvert P\rvert = \lvert C\rvert$
CFB	O	X	O	X	X	O	$\lvert P\rvert = \lvert C\rvert$
CTR	O	X	O	X	X	O	$\lvert P\rvert = \lvert C\rvert$
CBC – CS	O	X	O	O	X	O	$\lvert P\rvert = \lvert C\rvert$

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)

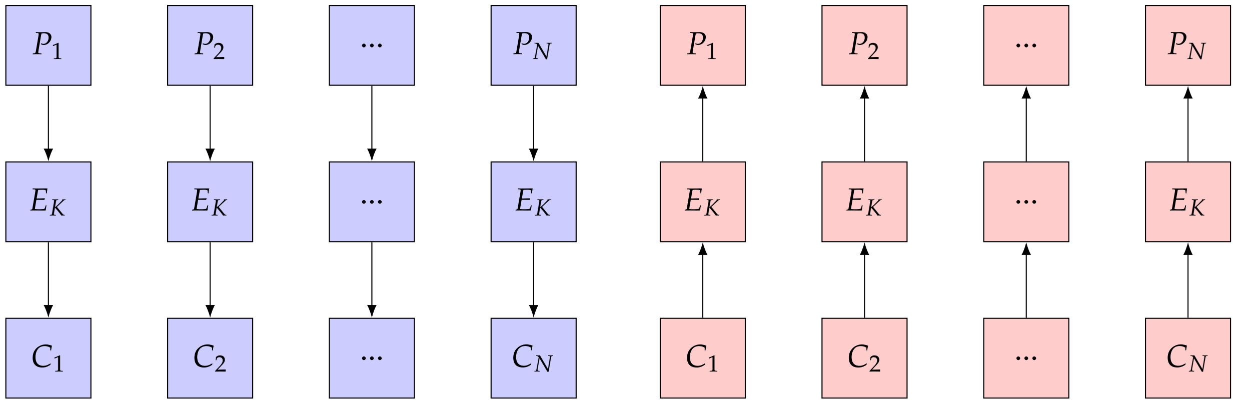
Algorithm 1: Electronic CodeBook

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

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

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

for $i \leftarrow 1$ **to** N **do**
  $P_i \leftarrow \mathsf{DecryptBlk}(K, C_i)$;
end
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)

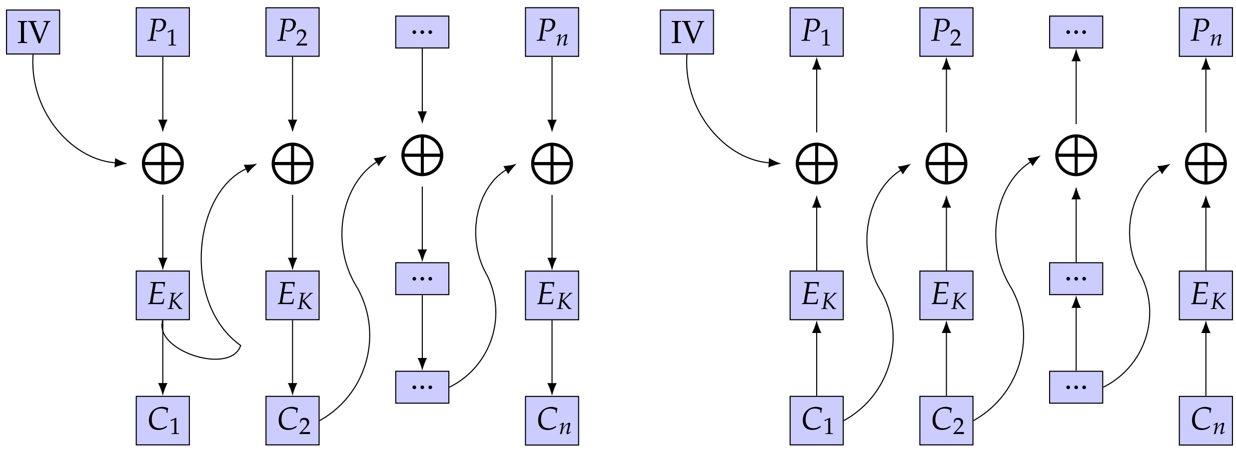
Algorithm 2: Cipher Block Chaining

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

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

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

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



Remark 1.2.

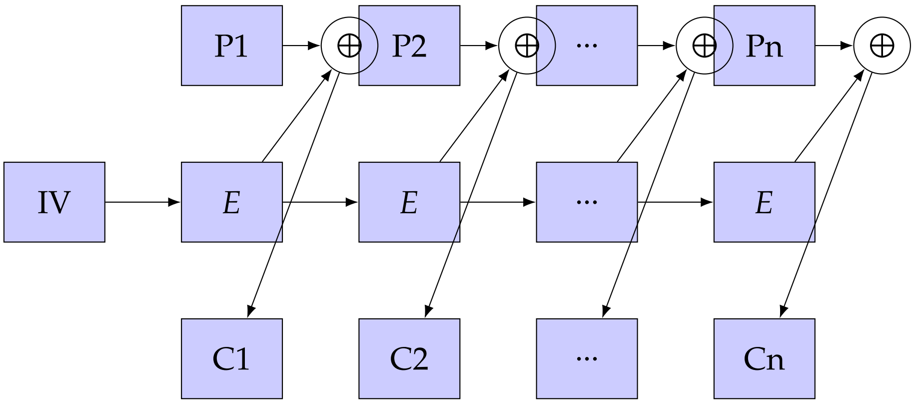
(1) For a set of plaintext/ciphertext pairs (IV, P, C) and (IV', P', C'),

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

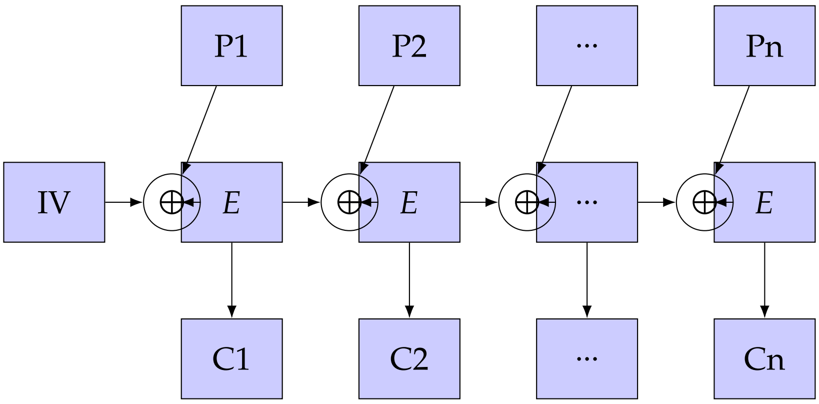
(2) A one-bit error in the ciphertext will affect the corresponding block and the subsequent block in the decryption process, thereby enabling the detection of such errors.

(3) Altering the order of blocks, as well as adding or deleting them, is not possible.

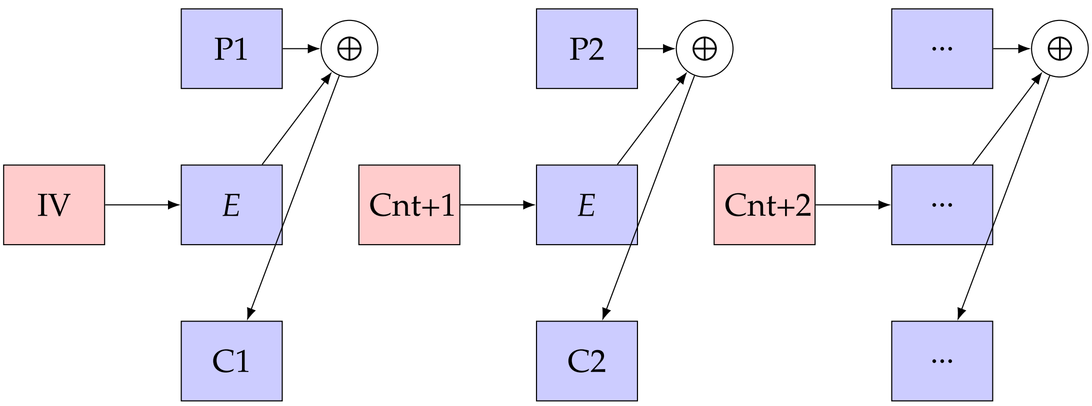
1.2.4 OFB (**Output FeedBack**)

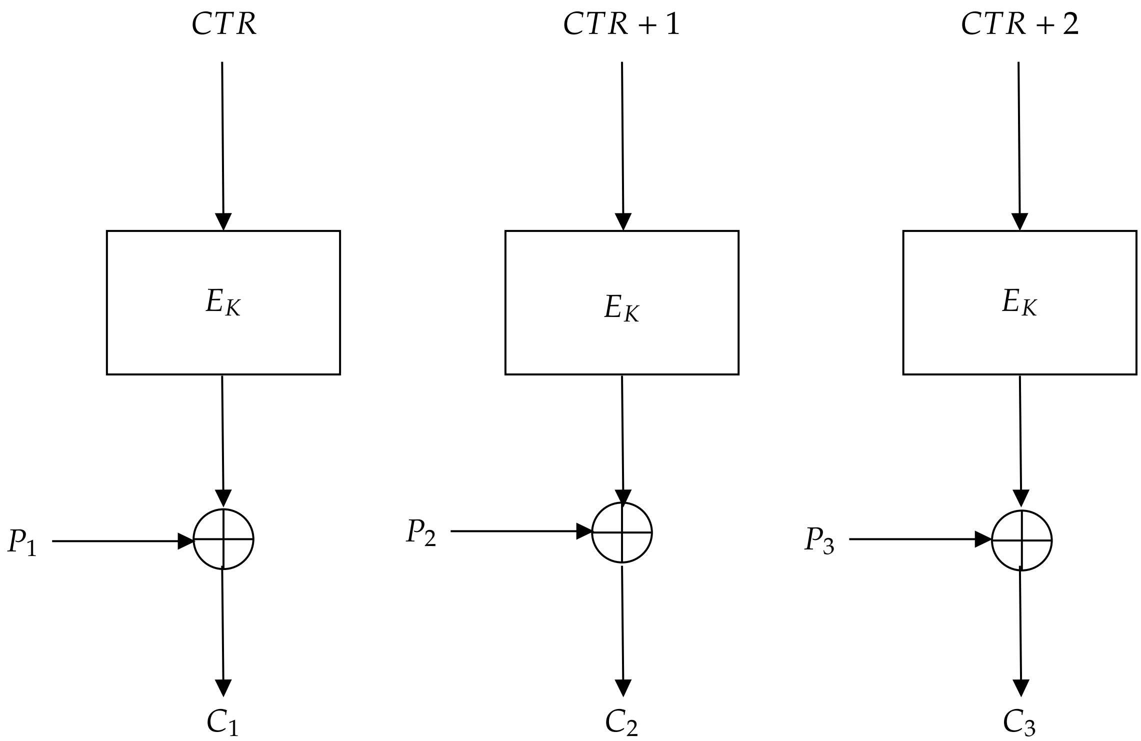


1.2.5 CFB (**Ciphertext FeedBack**)

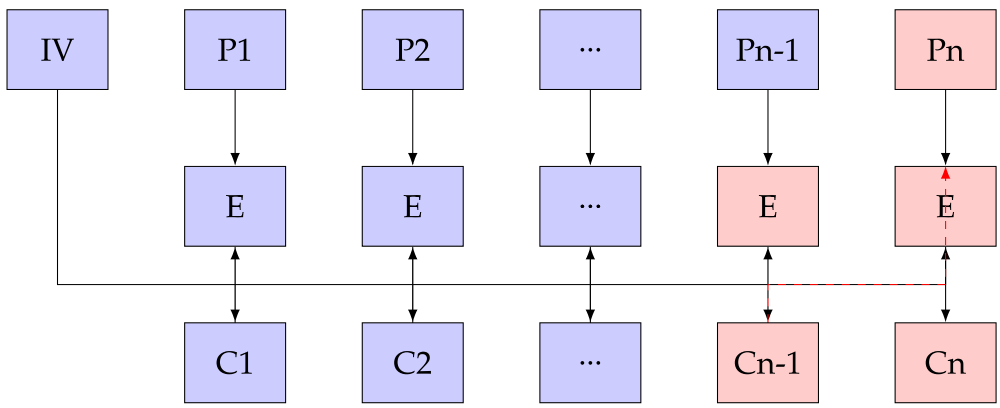


1.2.6 CTR (**CounTeR**)





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. $a = bq + r \quad (0 \le r < b)$. Then

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

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

Sol.

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

□

```
u32 gcd(u32 a, u32 b) {
    u32 r;

    while (b != 0) {
        r = a % b;
        a = b;
        b = r;
    }

    return a;
}

u32 rec_gcd(u32 a, u32 b) {
    return (b == 0) ? a : rec_gcd(b, a % b);
}
```

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,

$$\begin{aligned} 9 = 63 - 27 \cdot 2 &= 63 - (90 + (-1) \cdot 63) \cdot 2 \\ &= (1 + 2) \cdot 63 + (-2) \cdot 90 \\ &= (3) \cdot 63 + (-2) \cdot 90. \end{aligned}$$

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

$$\begin{aligned} a &= b \cdot q + r, \\ a' &= b' \cdot q' + r' \end{aligned}$$

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

$$\begin{aligned} a' &= b = u_b \cdot a + v_b \cdot b = u'_a \cdot a + v'_a \cdot b, \\ b' &= r = a - bq = (u_a \cdot a + v_a \cdot b) - (u_b \cdot a + v_b \cdot b) \cdot q \\ &= (u_a - u_b \cdot q)a + (v_a - v_b \cdot q)b \\ &= u_b \cdot' a + v_b \cdot' b. \end{aligned}$$

Thus,

$$\begin{aligned} u'_a &= u_b, & v'_a &= v_b \\ u'_b &= u_a - u_b q, & v'_b &= v_a - v_b \cdot q. \end{aligned}$$

```
u32 eea(u32 a, u32 b, u32* x, u32* y) {
    u32 an, bn; u32 ua, va, ub, vb; u32 new_an, new_bn;
    u32 n_ua, n_va, n_ub, n_vb; u32 q;

    an = a; bn = b;
    ua = 1; va = 0; ub = 0; vb = 1;
    while (bn != 0) {
        q = an / bn;
        new_an = bn;
        new_bn = an - bn * q;

        n_ua = ub; n_va = vb;
        n_ub = ua - ub * q; n_vb = va - vb * q;

        an = new_an; bn = new_bn;
        ua = n_ua; va = n_va; ub = n_ub; vb = n_vb;
    }
    *x = ua; *y = va;
    return an;
}
```

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 $\deg r \le 7$.

We define

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

Then

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

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

$$
\begin{aligned}
g(x)\cdot f(x) &= \sum_{i=0}^{7} g(x)a_i x^i \\
&= g(x)a_0 + \sum_{i=1}^{7} g(x)a_i x^i \\
&= g(x)a_0 + x\sum_{i=1}^{7} g(x)a_i x^{i-1} \\
&= g(x)a_0 + x\left(g(x)a_1 + \sum_{i=2}^{7} g(x)a_i x^{i-2}\right) \\
&= g(x)a_0 + x\left(g(x)a_1 + x\left(g(x)a_2 + \sum_{i=3}^{7} g(x)a_i x^{i-3}\right)\right) \\
&= \cdots \\
&= g(x)a_0 + x(g(x)a_1 + x(g(x)a_2 + x(g(x)a_3 + x(g(x)a_4 + x(g(x)a_5 + x(g(x)a_6 + x(g(x)a_7)))))))
\end{aligned}
$$

That is,

$$
\begin{array}{lll}
x & \cdot 0 & +(g(x)\cdot a_7) \\
x & \cdot(g(x)\cdot a_7) & +(g(x)\cdot a_6) \\
x & \cdot(x\cdot(g(x)\cdot a_7)+(g(x)\cdot a_6)) & +(g(x)\cdot a_5)
\end{array}
$$

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

```c
u8 GF256_mul(u8 f, u8 g) {
    u8 h = 0x00; // h = f * g
    u8 coef;

    for (i8 i = 7; i >= 0; i--) {
        coef = (f >> i) & 0x01; // Get the coefficient f_i
        h = GF256_xtime(h);     // Multiply h by x
        if (coef == 1) h ^= g;  // If f_i = 1, add g(x) to h(x)
    }
    return h;
}
```

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

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

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

$$GF(2^8)[x]/\langle x^4+1\rangle = \left\{a_0 + a_1x + a_2x^3 + a_3x^3 : a_i \in GF(2^8)\right\}.$$

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

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

where

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

Let $b(x) = b_0 + b_1x + b_2x^2 + b_3x^3$, $c(x) = c_0 + c_1x + c_2x^2 + c_3x^3$, 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

$$
\begin{aligned}
a(x) \cdot b(x) = (a_3 \cdot b_3)\, \boxed{x^6} &+ (a_3 \cdot b_2 + a_2 \cdot b_3)\, \boxed{x^5} \\
&+ (a_3 \cdot b_1 + a_2 \cdot b_2 + a_1 \cdot b_3)\, \boxed{x^4} \\
&+ (a_3 \cdot b_0 + a_2 \cdot b_1 + a_1 \cdot b_2 + a_0 \cdot b_3)x^3 \\
&+ (a_2 \cdot b_0 + a_1 \cdot b_1 + a_0 \cdot b_2)x^2 \\
&+ (a_1 \cdot b_0 + a_0 \cdot b_1)x \\
&+ (a_0 \cdot b_0)
\end{aligned}
$$

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

$$
\begin{aligned}
&= (a_3 \cdot b_0 + a_2 \cdot b_1 + a_1 \cdot b_2 + a_0 \cdot b_3)x^3 \\
&= (a_2 \cdot b_0 + a_1 \cdot b_1 + a_0 \cdot b_2 + a_3 \cdot b_3)x^2 \\
&= (a_1 \cdot b_0 + a_0 \cdot b_1 + a_3 \cdot b_2 + a_2 \cdot b_3)x \\
&= (a_0 \cdot b_0 + a_3 \cdot b_1 + a_2 \cdot b_2 + a_1 \cdot b_3)
\end{aligned}
$$

$$
= c(x).
$$

Thus, we have

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

where

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

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

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

$$
\begin{aligned}
\mathtt{02} \otimes a &= \mathtt{xtime}(a), \\
\mathtt{03} \otimes a &= (\mathtt{02} \oplus \mathtt{01}) \otimes a \\
&= (\mathtt{02} \otimes a) \oplus (\mathtt{01} \otimes a) \\
&= \mathtt{xtime}(a) \oplus a.
\end{aligned}
$$

Thus,

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

and so

$$
\begin{aligned}
c_0 &= (b_0 \oplus b_1 \oplus b_2 \oplus b_3) \oplus \mathtt{temp0} \oplus b_0 && \text{with } \mathtt{temp0} = \mathtt{xtime}\,(b_0 \oplus b_1), \\
c_1 &= (b_0 \oplus b_1 \oplus b_2 \oplus b_3) \oplus \mathtt{temp1} \oplus b_1 && \text{with } \mathtt{temp1} = \mathtt{xtime}\,(b_1 \oplus b_2), \\
c_2 &= (b_0 \oplus b_1 \oplus b_2 \oplus b_3) \oplus \mathtt{temp2} \oplus b_2 && \text{with } \mathtt{temp2} = \mathtt{xtime}\,(b_2 \oplus b_3), \\
c_3 &= (b_0 \oplus b_1 \oplus b_2 \oplus b_3) \oplus \mathtt{temp3} \oplus b_3 && \text{with } \mathtt{temp3} = \mathtt{xtime}\,(b_3 \oplus b_0).
\end{aligned}
$$

That is,

$\mathtt{sum} \leftarrow b_0 \oplus b_1 \oplus b_2 \oplus b_3$;
$c_0 \leftarrow \mathtt{sum} \oplus \mathtt{temp0} \oplus b_0$;
$c_0 \leftarrow \mathtt{sum} \oplus \mathtt{temp1} \oplus b_1$;
$c_0 \leftarrow \mathtt{sum} \oplus \mathtt{temp2} \oplus b_2$;
$c_0 \leftarrow \mathtt{sum} \oplus \mathtt{temp3} \oplus b_3$;

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

$$
\begin{aligned}
[a(x)]^1 &= a_3x^3 + a_2x^2 + a_1x + a_0, \\
[a(x)]^2 &= (a_3^2 + a_1^2)x^2 + (a_2^2 + a_0^2), \\
[a(x)]^4 &= a_3^4 + a_2^4 + a_1^4 + a_0^4.
\end{aligned}
$$

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

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

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

2.5 Little and Big Endian

2.5.1 Introduction

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

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

$[\mathrm{u32} \longrightarrow (\mathrm{u8} \times 4)]:$

$$
\begin{aligned}
\mathtt{b[0]} &= (\mathtt{x} \gg \mathtt{0x18}) \\
\mathtt{b[1]} &= (\mathtt{x} \gg \mathtt{0x10}) \wedge \mathtt{0xff} \\
\mathtt{b[2]} &= (\mathtt{x} \gg \mathtt{0x08}) \wedge \mathtt{0xff} \\
\mathtt{b[3]} &= (\mathtt{x} \qquad\qquad) \wedge \mathtt{0xff}
\end{aligned}
$$

$[(\mathrm{u8} \times 4) \longrightarrow \mathrm{u32}]:$

$$
\begin{aligned}
\mathtt{x} = {} & (\mathtt{b[0]} \ll \mathtt{0x18}) \wedge \\
& (\mathtt{b[1]} \ll \mathtt{0x10}) \wedge \\
& (\mathtt{b[2]} \ll \mathtt{0x08}) \wedge \\
& (\mathtt{b[3]} \qquad\qquad)
\end{aligned}
$$

	b[0]	b[1]	b[2]	b[3]
pInt ⟶	00	01	02	03
	*(b+0)	*(b+1)	*(b+2)	*(b+3)

$$\mathtt{b} \Longleftrightarrow \mathtt{\&(b[0])} \Longleftrightarrow \mathtt{(u8*)\&b[0]}$$

```c
u8 b[4] = { 0x00, 0x01, 0x02, 0x03 };
u32* pInt;
u32 x;

pInt = (u32*)b;
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
```

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

```
u8 b[4] = { 0x00, 0x01, 0x02, 0x03 };
u32* px = (u32*)b;
u32 x, y;

x = *px;
y = ((u32)b[0] << 0x18) ^
    ((u32)b[1] << 0x10) ^
    ((u32)b[2] << 0x08) ^
    ((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

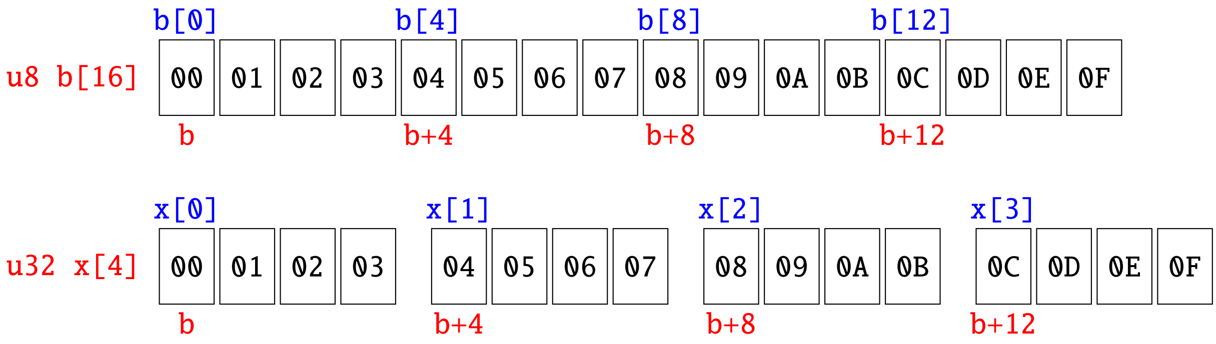
```
#define GETU32(pt) ((u32)(pt)[0] << 0x18) ^ \
                   ((u32)(pt)[1] << 0x10) ^ \
                   ((u32)(pt)[2] << 0x08) ^ \
                   ((u32)(pt)[3]        )

u8 b[4] = { 0x00, 0x01, 0x02, 0x03 };
u32 y;

y = GETU32(b);
```

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

Byte to Word

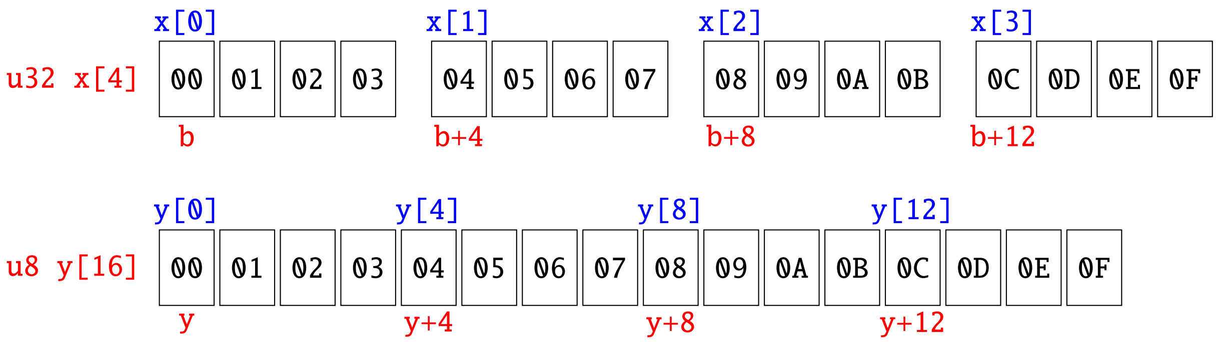



```
u8  b[16] = { 0x00, 0x01, 0x02, 0x03,
              0x04, 0x05, 0x06, 0x07,
              0x08, 0x09, 0x0A, 0x0B,
              0x0C, 0x0D, 0x0E, 0x0F};

u32  x[4] = { GETU32(b       ),
              GETU32(b + 0x04),
              GETU32(b + 0x08),
              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



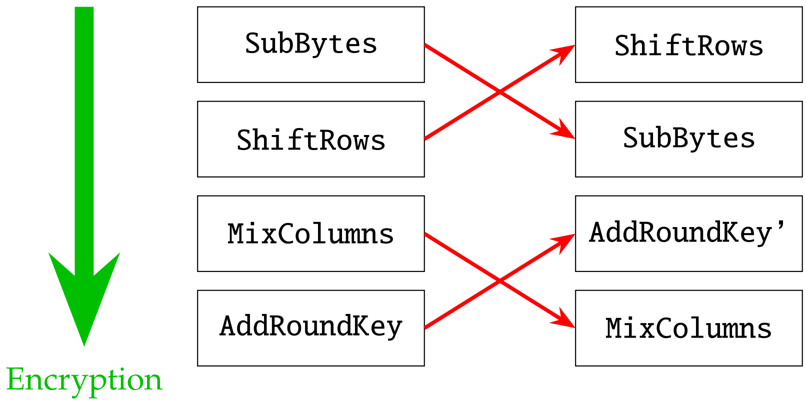
```
#define PUTU32(ct, st) { (ct)[0] = (u8)((st) >> 0x18); \
                         (ct)[1] = (u8)((st) >> 0x10); \
                         (ct)[2] = (u8)((st) >> 0x08); \
                         (ct)[3] = (u8)((st)       ); }

u32 x[4] = { 0x00010203, 0x04050607, 0x08090A0B, 0x0C0D0E0F };
u8 y[16];

PUTU32(y       , x[0]);
PUTU32(y + 0x04, x[1]);
PUTU32(y + 0x08, x[2]);
PUTU32(y + 0x0C, x[3]);
```

```
>>> x/16xb x
0x7fffffffd260: 0x03 0x02 0x01 0x00 0x07 0x06 0x05 0x04
0x7fffffffd268: 0x0b 0x0a 0x09 0x08 0x0f 0x0e 0x0d 0x0c
>>> x/16xb y
0x7fffffffd280: 0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07
0x7fffffffd288: 0x08 0x09 0x0a 0x0b 0x0c 0x0d 0x0e 0x0f
```

2.6 Implementation of 32-bit AES



$$
\begin{aligned}
y &= \texttt{MixCol(x)} \oplus \texttt{rk} \\
&= \texttt{MixCol(x} \oplus \texttt{InvMixCol(rk))} \\
&= \texttt{MixCol(x} \oplus \texttt{rk')}
\end{aligned}
$$

	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

```
Function Cipher (in, Nr, w):
    state ← in
    state ← AddRoundKey(state, w[0 .. 3])      // w[0..3] = w[0], w[1], w[2], w[3]
    for round ← 1 to Nr − 1 do
        state ← SubBytes(state)
        state ← ShiftRows(state)
        state ← MixColumns(state)
        state ← AddRoundKey(state, w[4*round .. 4*round + 3])
    end
    state ← SubBytes(state)
    state ← ShiftRows(state)
    state ← AddRoundKey(state, w[4*Nr .. 4*Nr + 3])
    return state
end
```

Algorithm 4: AES Inverse Encryption

```
Function InvCipher (in, Nr, w):
    state ← in
    state ← AddRoundKey(state, w[4*Nr .. 4*Nr + 3])
    for round ← Nr − 1 downto 1 do
        state ← InvShiftRows(state)
        state ← InvSubBytes(state)
        state ← AddRoundKey(state, w[4*round .. 4*round + 3])
        state ← InvMixColumns(state)
    end
    state ← InvShiftRows(state)
    state ← InvSubBytes(state)
    state ← AddRoundKey(state, w[0 .. 3])
    return state
end
```

Algorithm 5: AES Inverse Encryption 2

```
Function EqInvCipher (in, Nr, w):
    state ← in
    state ← AddRoundKey(state, w[4*Nr .. 4*Nr + 3])
    for round ← Nr − 1 downto 1 do
        state ← InvSubBytes(state)
        state ← InvShiftRows(state)
        state ← InvMixColumns(state)
        state ← AddRoundKey(state, dw[4*round .. 4*round + 3])
        /* y = MixCol(x ⊕ rk') ⟹ rk' = x ⊕ InvMixCol(y)                 */
        /* for round ← 1 to Nr − 1 do
               i ← 4 * round
               dw[i .. i + 3] ← InvMixColumns(dw[i .. i + 3])
           end
                                                                        */
    end
    state ← InvSubBytes(state)
    state ← InvShiftRows(state)
    state ← AddRoundKey(state, w[0 .. 3])
    return state
end
```

$$\begin{array}{|c|c|c|c|}\hline a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ \hline a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ \hline a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ \hline a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \\ \hline \end{array} \xrightarrow{\texttt{SubBytes}} \begin{array}{|c|c|c|c|}\hline b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ \hline b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ \hline b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ \hline b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \\ \hline \end{array} \xrightarrow{\texttt{ShiftRows}} \begin{array}{|c|c|c|c|}\hline c_{0,0} & c_{0,1} & c_{0,2} & c_{0,3} \\ \hline c_{1,0} & c_{1,1} & c_{1,2} & c_{1,3} \\ \hline c_{2,0} & c_{2,1} & c_{2,2} & c_{2,3} \\ \hline c_{3,0} & c_{3,1} & c_{3,2} & c_{3,3} \\ \hline \end{array} \xrightarrow{\texttt{MixColumns}} \begin{array}{|c|c|c|c|}\hline d_{0,0} & d_{0,1} & d_{0,2} & d_{0,3} \\ \hline d_{1,0} & d_{1,1} & d_{1,2} & d_{1,3} \\ \hline d_{2,0} & d_{2,1} & d_{2,2} & d_{2,3} \\ \hline d_{3,0} & d_{3,1} & d_{3,2} & d_{3,3} \\ \hline \end{array}$$

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

$$\begin{aligned} \begin{pmatrix} d_{0,0} \\ d_{1,0} \\ d_{2,0} \\ d_{3,0} \end{pmatrix} = \begin{pmatrix} \texttt{02} & \texttt{03} & \texttt{01} & \texttt{01} \\ \texttt{01} & \texttt{02} & \texttt{03} & \texttt{01} \\ \texttt{01} & \texttt{01} & \texttt{02} & \texttt{03} \\ \texttt{03} & \texttt{01} & \texttt{01} & \texttt{02} \end{pmatrix} \begin{pmatrix} S(a_{0,0}) \\ S(a_{1,1}) \\ S(a_{2,2}) \\ S(a_{3,3}) \end{pmatrix} &= S(a_{0,0}) \begin{pmatrix} \texttt{02} \\ \texttt{01} \\ \texttt{01} \\ \texttt{03} \end{pmatrix} \oplus S(a_{1,1}) \begin{pmatrix} \texttt{03} \\ \texttt{02} \\ \texttt{01} \\ \texttt{01} \end{pmatrix} \oplus S(a_{2,2}) \begin{pmatrix} \texttt{01} \\ \texttt{03} \\ \texttt{02} \\ \texttt{01} \end{pmatrix} \oplus S(a_{3,3}) \begin{pmatrix} \texttt{01} \\ \texttt{01} \\ \texttt{03} \\ \texttt{02} \end{pmatrix} \\ &= Te0(a_{0,0}) \oplus Te1(a_{1,1}) \oplus Te2(a_{2,2}) \oplus Te3(a_{3,3}) \end{aligned}$$

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

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

$a_{0,0}$	$S(a_{0,0}) \begin{pmatrix} \texttt{02} & \texttt{01} & \texttt{01} & \texttt{03} \end{pmatrix}^T$
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).

$$\begin{array}{|c|c|c|c|}\hline d_{0,0} & d_{0,1} & d_{0,2} & d_{0,3} \\ \hline d_{1,0} & d_{1,1} & d_{1,2} & d_{1,3} \\ \hline d_{2,0} & d_{2,1} & d_{2,2} & d_{2,3} \\ \hline d_{3,0} & d_{3,1} & d_{3,2} & d_{3,3} \\ \hline \end{array} \xrightarrow{\texttt{AddRoundKey}} \begin{array}{|c|c|c|c|}\hline e_{0,0} & e_{0,1} & e_{0,2} & e_{0,3} \\ \hline e_{1,0} & e_{1,1} & e_{1,2} & e_{1,3} \\ \hline e_{2,0} & e_{2,1} & e_{2,2} & e_{2,3} \\ \hline e_{3,0} & e_{3,1} & e_{3,2} & e_{3,3} \\ \hline \end{array}$$

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

```
Function KeyExpansion (key):
    i ← 0
    while i ≤ (Nk − 1) do
        w[i] ← key[4 ∗ i .. 4 ∗ i + 3]
        i ← i + 1
    end
    while i ≤ (4 ∗ Nr + 3) do
        temp ← w[i − 1]
        if (i mod Nk) = 0 then
            /* RotWord([a0, a1, a2, a3]) = [a1, a2, a3, a0]                  */
            /* SubWord([a0, a1, a2, a3]) = [S(a0), S(a1), S(a2), S(a3)]      */
            temp ← SubWord(RotWord(temp)) ⊕ Rcon[i/Nk]
        end
        else if Nk > 6 and (i mod Nk) = 4 then
            temp ← SubWord(temp)
        end
        w[i] ← w[i − Nk] ⊕ temp
        i ← i + 1
    end
    return w
end
```

Chapter 3

AES-GCM

3.1 Multiplication in GF(2^{128})

3.1.1 Basic Multiplication

Let $m(x) := 1 + x + x^2 + x^7 + x^{128}$.

$$
\begin{aligned}
GF(2^{128}) &= GF(2)[x]/\langle m(x)\rangle \\
&= GF(2)[x]/\langle 1 + x + x^2 + x^7 + x^{128}\rangle \\
&= \left\{ \sum_{i=0}^{127} b_i : b_i \in \mathrm{GF}(2) \text{ for } i = 1, 2, \ldots, 127 \right\}.
\end{aligned}
$$

$$
\begin{aligned}
B &= B[0] \parallel B[1] \parallel \cdots \parallel B[15] & B[i] &\in \mathrm{GF}(2^8) \\
&= b_0 b_1 \cdots b_7 \parallel b_8 \cdots b_{15} \parallel \cdots \parallel b_{120} \cdots b_{127} & b_i &\in \mathrm{GF}(2)
\end{aligned}
$$

Let $f(x), g(x) \in GF(2^{128})$, say,

$$
\begin{aligned}
p(x) &= p_0 + p_1 x + p_2 x^2 + \cdots + p_{127} x^{127} \\
q(x) &= q_0 + q_1 x + q_2 x^2 + \cdots + q_{127} x^{127}.
\end{aligned}
$$

Then

$$
\begin{aligned}
p(x) \cdot q(x) = p(x) \cdot \left(q_0 + q_1 x + q_2 x^2 + \cdots + q_{127} x^{127}\right) \\
= q_0 \cdot p(x) + q_1 \cdot x p(x) + q_2 \cdot x^2 p(x) + \cdots + q_{127} \cdot x^{127} p(x).
\end{aligned}
$$

The function `xtime`() : GF(2^{128}) $\to$ GF(2^{128}) is define by

$$\begin{aligned}\texttt{xtime}(p(x)) &= xp(x)\\ &= x(p_0 + p_1x + p_2x^2 + \cdots p_{127}x^{127})\\ &= p_0x + p_1x^2 + p_2x^3 + \cdots + p_{126}x^{127} + p_{127}x^{128}\\ &= p_0x + p_1x^2 + p_2x^3 + \cdots + p_{126}x^{127} + p_{127}(1 + x + x^2 + x^7)\end{aligned}$$

for $p(x) \in \mathrm{GF}(2^{128})$.

$$\begin{aligned}p(x) &= P[0] \parallel P[1] \parallel \cdots P[15]\\ &= p_0p_1\cdots p_7 \parallel p_8\cdots p_{15} \parallel \cdots \parallel p_{120}\cdots p_{127}\end{aligned}$$

$$xp(x) = \left((0p_0\cdots p_6) \oplus \begin{cases} \texttt{0000:0000} = \texttt{0x00} & : p_{127} = 0\\ \texttt{1110:0001} = \texttt{0xE1} & : p_{127} = 1\end{cases}\right) \parallel (p_7\cdots p_{14}) \parallel \cdots \parallel (p_{119}\cdots p_{126}).$$

```c
// p(x) <- x*p(x)
void GF128_xtime(u8 p[16]) {
    u8 msb = (u8)(p[15] & 0x01);  // p[15] = p120 p121 ... p127
    for (int i = 15; i > 0; i--) {
        p[i] = (p[i] >> 1) | ((p[i-1] & 0x01) << 7);
    } p[0] >>= 1;
    if (msb) p[0] ^= 0xE1;
}
```

Recall that

$$\begin{aligned}p(x)\cdot q(x) &= p(x)\cdot\left(q_0 + q_1x + q_2x^2 + \cdots + q_{127}x^{127}\right)\\ &= q_0\cdot p(x) + q_1\cdot xp(x) + q_2\cdot x^2p(x) + \cdots + q_{127}\cdot x^{127}p(x).\end{aligned}$$

for $p(x), q(x) \in \mathrm{GF}(2^{128})$.

```c
// p(x) <- p(x)*q(x)
void GF128_mul(u8 p[16], u8 q[16]) {
    u8 buffer[16] = { 0x00, };
    u8 qi;  // q0, q1, ..., q127
    for (int i = 0; i < 16; i++) {        // Q[0], Q[1], ..., Q[15]
        for (int j = 0; j < 8; j++) {    // q0, q1, ..., q127
            qi = q[i] & (1 << (7-j));    // Q[0] = q0q1...q7
            if (qi) {
                for (int k = 0; k < 16; k++) buffer[k] ^= p[k];
            }
            GF128_xtime(p);      // xp(x), x^2p(x), ..., x^127p(x)
        }
    }
    for (int i = 0; i < 16; i++) p[i] = buffer[i];
}
```

3.1.2 GHASH and Efficient Multiplication

```
void GHASH_v1(u8 msg[],
              int msg_blks,
              u8 H[16],
              u8 tag[16]) {
    u8 x[16];
    u8 out[16] = { 0x00, };

    for (int i = 0; i < msg_blks, i++) {
        for (int j = 0; j < 16; j++)
            x[j] = msg[i * 16 + j];
        xor_b_array(out, 16, x)      // out <- out ^ x
        GF128_mul(out, H);           // out <- out * H
    }
    for (int i = 0; i < 16; i++)
        tag[i] = out[i];
}
```

Consider $q(x) \in \mathrm{GF}(2^{128})$, where

$$\begin{aligned} q(x) &= \boxed{(q_0 + q_1x + \cdots + q_7x^7)} + \boxed{(q_8x^8 + \cdots + q_{15}x^{15})} + \cdots + \boxed{(q_{120}x^{120} + \cdots + q_{127}x^{127})} \\ &= \boxed{(q_0 + q_1x + \cdots + q_7x^7)} + \boxed{(q_8 + \cdots + q_{15}x^7)}\,x^8 + \cdots + \boxed{(q_{120} + \cdots + q_{127}x^7)}\,x^{120} \\ &= \boxed{B_0(x)} + \boxed{B_1(x)}\,x^8 + \cdots + \boxed{B_{15}(x)}\,x^{120} \quad \text{with} \quad B_i(x) \in \mathrm{GF}(2^8). \end{aligned}$$

Then

$$\begin{aligned} H(x) \cdot q(x) &= H(x) \cdot \left\{ B_0(x) + B_1(x)x^8 + \cdots + B_{15}(x)x^{120} \right\} \\ &= H(x)B_0(x) + H(x)B_1(x)x^8 + \cdots + H(x)B_{15}(x)x^{120}. \end{aligned}$$

Since $B_i(x) \in \mathrm{GF}(2^8)$, we have

$$
\text{256 times}\left\{
\begin{array}{lcl}
H(x)\cdot(\mathtt{0000{:}0000}) & \dashrightarrow & \mathtt{HT}[0] = \mathtt{HT}[0][0] \parallel \mathtt{HT}[0][1] \parallel \cdots \parallel \mathtt{HT}[0][15] \\
H(x)\cdot(\mathtt{1000{:}0000}) & \dashrightarrow & \mathtt{HT}[1] = \mathtt{HT}[1][0] \parallel \mathtt{HT}[1][1] \parallel \cdots \parallel \mathtt{HT}[1][15] \\
H(x)\cdot(\mathtt{0100{:}0000}) & \dashrightarrow & \mathtt{HT}[2] = \mathtt{HT}[2][0] \parallel \mathtt{HT}[2][1] \parallel \cdots \parallel \mathtt{HT}[2][15] \\
\vdots & \dashrightarrow & \vdots \\
H(x)\cdot(\mathtt{1111{:}1111}) & \dashrightarrow & \mathtt{HT}[255] = \mathtt{HT}[255][0] \parallel \mathtt{HT}[255][1] \parallel \cdots \parallel \mathtt{HT}[255][15]
\end{array}
\right.
$$

We use 256×16 bytes (equivalent to 4 KB) of memory to store the fixed function $H(x)$, ensuring efficient handling of precomputed values.

```
// HT[q(x)][] <- H(x)*q(x)
void Make_GHASH_H_table(u8 H[16], u8 HT[256][16]) {
    u8 buf[16];
    u8 H_mul[16];    // H(x), H(x)*x, H(x)*x^2, ... , H(x)*x^7
    u8 qj;           // q0, q1, ... , q7
    for (int i = 0; i < 256; i++) {      // 0x00 - 0xFF
        // H(x)*[0x00 - 0xFF] ==> HT[i][0]...HT[i][16]
        for (int j = 0; j < 16; j++) { // Initialize buffer
            buf[j] = 0x00;
            H_mul[j] = H[j];  // Initialize as H(x)
        }
        for (int j = 0; j < 8; j++) {   // q0, q1, ... , q7
            // [i polynomial] =  [q0 q1 q2 ... q7] (8-bit)
            qj = (u8)((i >> (7 - j)) & 0x01);
            if (qj == 1) {
                // buf <- buf + qj*H(x)*x^j
                xor_b_array(buf, 16, H_mul);
            } GF128_xtime(H_mul); // Hmul <-- H_mul * x
        } copy_b_array(buf, 16, HT[i]); // buf[] --> HT[i][]
    }
}
```

The function $\mathtt{x}^8\mathtt{time}() : \mathrm{GF}(2^{128}) \longrightarrow \mathrm{GF}(2^{128})$ is define by

$$
\begin{aligned}
\mathtt{x}^8\mathtt{time}(p(x)) &= x^8 p(x) \\
&= x^8(p_0 + p_1 x + p_2 x^2 + \cdots p_{127}x^{127}) \\
&= (p_0 x^8 + p_1 x^9 + p_2 x^{10} + \cdots p_{119}x^{127}) + (p_{120}x^{128} + \cdots + p_{127}x^{135}) \\
&= 0 + P_0(x)x^8 + P_1(x)x^{16} + \cdots + P_{14}x^{12} + P_{15}(x)x^{128} \\
&= (\mathtt{0}^8 \parallel P_1(x) \parallel \cdots \parallel P_{14}(x)) \oplus P_{15}(x)(1 + x + x^2 + x^7).
\end{aligned}
$$

for $p(x) \in \mathrm{GF}(2^{128})$. We use 256×2 bytes (equivalent to 0.5 KB) of memory to store the $P_{15}(x)x^{128}$:

$$\begin{aligned}
&P_{15}(x)x^{128} \\
&= (p_{120} + p_{121}x + \cdots + p_{127}x^7)(1 + x + x^2 + x^7) \\
&= p_{120} + p_{121}x + p_{122}x^2 + p_{123}x^3 + \cdots + p_{127}x^7 \\
&\qquad + p_{120}x + p_{121}x^2 + p_{122}x^3 + \cdots + p_{126}x^7 + p_{127}x^8 \\
&\qquad\qquad + p_{120}x^2 + p_{121}x^3 + \cdots + p_{125}x^7 + p_{126}x^8 + p_{127}x^9 \\
&\qquad\qquad\qquad + p_{120}x^7 + p_{121}x^8 + p_{122}x^9 + \cdots + p_{127}x^{14} \\
&= (p_{120}) \,\|\, (p_{120} + p_{121}) \,\|\, (p_{120} + p_{121} + p_{122}) \,\|\, (p_{121} + p_{122} + p_{123}) \\
&\quad \|\, (p_{122} + p_{123} + p_{124}) \,\|\, (p_{123} + p_{124} + p_{125}) \,\|\, (p_{124} + p_{125} + p_{126}) \\
&\quad \|\, (p_{120} + p_{125} + p_{126} + p_{127}) \,\|\, (p_{121} + p_{126} + p_{127}) \,\|\, (p_{122} + p_{127}) \\
&\quad \|\, (p_{123}) \,\|\, (p_{124}) \,\|\, (p_{125}) \,\|\, (p_{126}) \,\|\, (p_{127}) \,\|\, (0) \\
&= R_0(x) \,\|\, R_1(x) \quad \text{with} \quad R_i(x) \in GF(2^8).
\end{aligned}$$

Thus, $\texttt{x}^8\texttt{time()} : GF(2^{128}) \longrightarrow GF(2^{128})$ is defined by

$$\texttt{x}^8\texttt{time}(P(x)) = x^8(P_0(x), \cdots, P_{15}(x)) = (0, P_0(x), \cdots, P_{14}(x)) \oplus R_0(P_{15}(x)) \,\|\, R_1(P_{15}(x))$$

```c
void Make_GHASH_const_R0R1(u8 R0[256], u8 R1[256]) {
    u8 a[8];     // a0 a1 ... a7
    for (int i = 0; i < 256; i++) {
        R0[i] = 0; R1[i] = 0;
    }

    for (int i = 0; i < 256; i++) {          // 0x00 - 0xFF
        for (int j = 0; j < 8; j++)
            a[j] = (i >> (7-j)) & 0x01;      // a0, a1, ..., a7
        R0[i] = a[0] << 7;
        R0[i] ^= (a[0] ^ a[1]) << 6;
        R0[i] ^= (a[0] ^ a[1] ^ a[2]) << 5;
        R0[i] ^= (a[1] ^ a[2] ^ a[3]) << 4;
        R0[i] ^= (a[2] ^ a[3] ^ a[4]) << 3;
        R0[i] ^= (a[3] ^ a[4] ^ a[5]) << 2;
        R0[i] ^= (a[4] ^ a[5] ^ a[6]) << 1;
        R0[i] ^= a[5] ^ a[6] ^ a[7] ^ a[0];

        R1[i] = (a[7] ^ a[6] ^ a[1]) << 7;
        R1[i] ^= (a[7] ^ a[2]) << 6;
        R1[i] ^= a[3] << 5;
        R1[i] ^= a[4] << 4;
        R1[i] ^= a[5] << 3;
        R1[i] ^= a[6] << 2;
        R1[i] ^= a[7] << 1;
    }
}
```

Chapter 4

AES-128

4.1 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 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 $\in \{0,1\}^{128}$

$t \leftarrow$ AddRoundKey(src, rk_0);
for $i \leftarrow 1$ **to** 9 **do**
  $t \leftarrow$ (MixColumns $\circ$ ShiftRows $\circ$ SubBytes)(t);
  $t \leftarrow$ AddRoundKey(t, rk_i);
end
$t \leftarrow$ (ShiftRows $\circ$ SubBytes)(t);
$t \leftarrow$ AddRoundKey(t, rk_{10});
dst $\leftarrow t$;
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 $\in \{0,1\}^{128}$

$t \leftarrow$ AddRoundKey(src, rk_{10});
for $i \leftarrow 9$ **to** 1 **do**
  $t \leftarrow$ (InvSubBytes $\circ$ InvShiftRows)(t);
  $t \leftarrow$ AddRoundKey(t, rk_i);
  $t \leftarrow$ InvMixColumns(t);
end
$t \leftarrow$ (InvShiftRows $\circ$ InvSubBytes)(t);
$t \leftarrow$ AddRoundKey(t, rk_0);
dst $\leftarrow t$;
return dst;

4.2 Functions and Constants used in AES

4.2.1 Key Expansion

- $\text{RotWord} : \{0, 1\}^{32} \to \{0, 1\}^{32}$ 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 4.1: RotWord rotates the input word left by one byte

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

- $\text{SubWord} : \{0, 1\}^{32} \to \{0, 1\}^{32}$ 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 4.2: 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 4.3: 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 $uk = (uk_0, \ldots, uk_{15})$ ($uk_i \in \{0, 1\}^8$); // $uk \in \{0, 1\}^{128}$ 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 : $\{0, 1\}^{32} \rightarrow \{0, 1\}^{32}$ */
    $t \leftarrow \mathrm{RotWord}(t)$;
    $t \leftarrow \mathrm{SubWord}(t)$;
    $t \leftarrow t \oplus (\mathtt{rCon}_{i/4} \,\|\, \mathtt{0x00} \,\|\, \mathtt{0x00} \,\|\, \mathtt{0x00})$;
  end
  $rk_i \leftarrow rk_{i-4} \oplus_{32} t$;
end

Code 4.4: 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++;
    }
}
```

4.2.2 AddRoundKey

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

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

Code 4.5: 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;
    }
}
```

4.2.3 SubBytes / InvSubBytes

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

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

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

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

Table 4.1: 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 4.6: Byte Substitution

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

Code 4.7: Inverse Byte Substitution

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

4.2.4 ShiftRows / InvShiftRows

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

$\Longrightarrow$

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 : $\{0, 1\}^{128} \longrightarrow \{0, 1\}^{128}$ 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}

$\Longrightarrow$

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 4.8: 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 4.9: 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;
}
```

4.2.5 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 4.10: 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 4.11: 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 4.12: 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 5

AES - 128 / 192 / 256 (Byte Version)

5.1 Specification

Table 5.1: 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 5.1: 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 5.2: Configuration in Rust

```
// Define a constant for the AES key length.
pub const AES_VERSION: u32 = 128; // Can be 128, 192, or 256

// Define constant for AES block size
pub const AES_BLOCK_SIZE: usize = 16;

// Define constants Nk and Nr based on AES key length
pub const NK: usize = match AES_VERSION {
    128 => 4,
    192 => 6,
    256 => 8,
    _ => panic!("Invalid AES key length"),
};

pub const NR: usize = NK + 6;
pub const ROUND_KEYS_SIZE: usize = 16 * (NR + 1);
```

5.2 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 */

$l \leftarrow N_k/4$; // $l = 4, 6, 8$

for $i = 0$ **to** $l - 1$ **do**

  $rk_i \leftarrow uk_{4i} \parallel uk_{4i+1} \parallel uk_{4i+2} \parallel uk_{4i+3}$;

end

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

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

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

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

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

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

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

  end

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

end

Code 5.3: 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 5.4: Key Expansion Test

```
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;
}
```

5.3 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 $\in \{0,1\}^{128}$

$t \leftarrow \text{AddRoundKey}(\text{src}, rk_0)$; // AddRoundKey: $\{0,1\}^{8*16} \times \{0,1\}^{32*4} \rightarrow \{0,1\}^{8*16}$

for $i \leftarrow 1$ **to** $N_r - 1$ **do**

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

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

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

  $t \leftarrow \text{AddRoundKey}(t, rk_i)$;

end

$t \leftarrow \text{SubBytes}(t)$;

$t \leftarrow \text{ShiftRows}(t)$;

$t \leftarrow \text{AddRoundKey}(t, rk_{N_r})$;

dst $\leftarrow t$;

return dst;

Code 5.5: 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 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 $\in \{0, 1\}^{128}$

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

Code 5.6: 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];
}
```