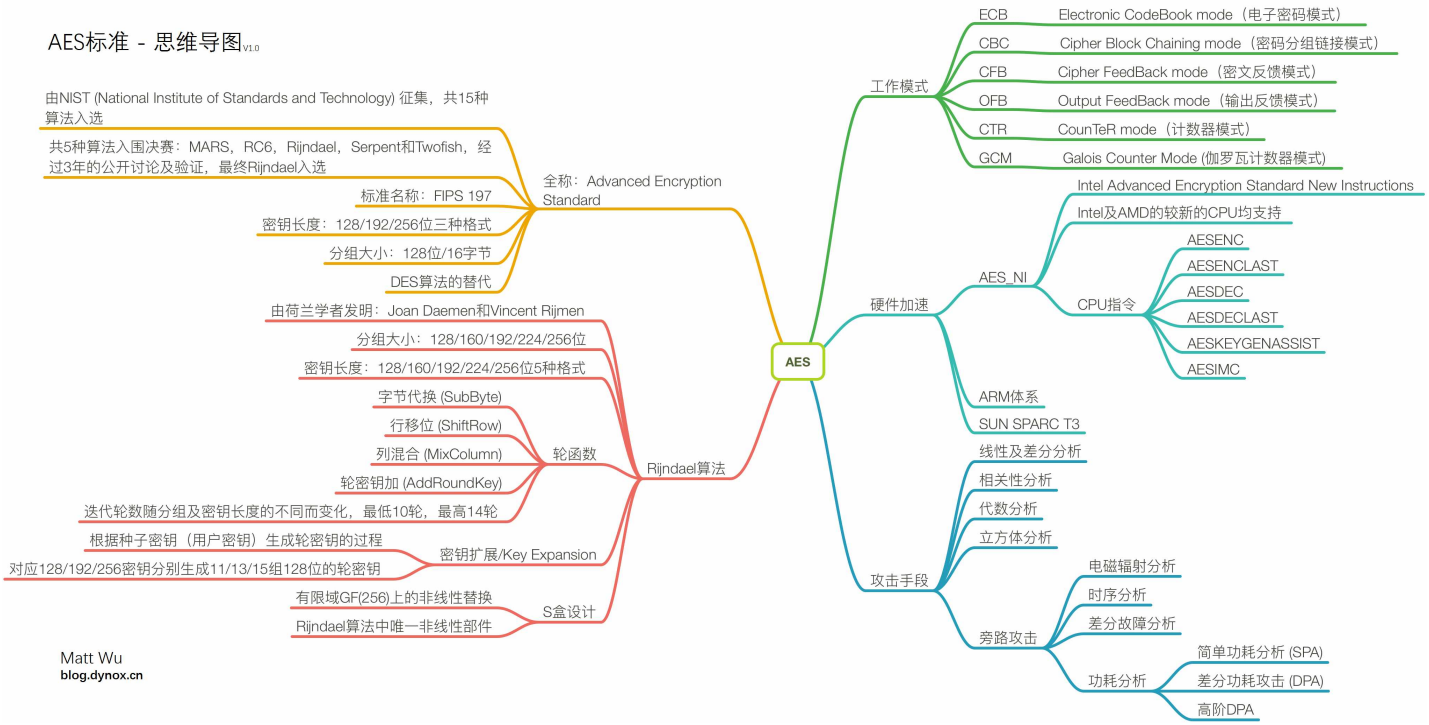
[TOC]

AES简介

AES, Advanced Encryption Standard，其实是一套标准：FIPS 197，而我们所说的AES算法其实是Rijndael算法。

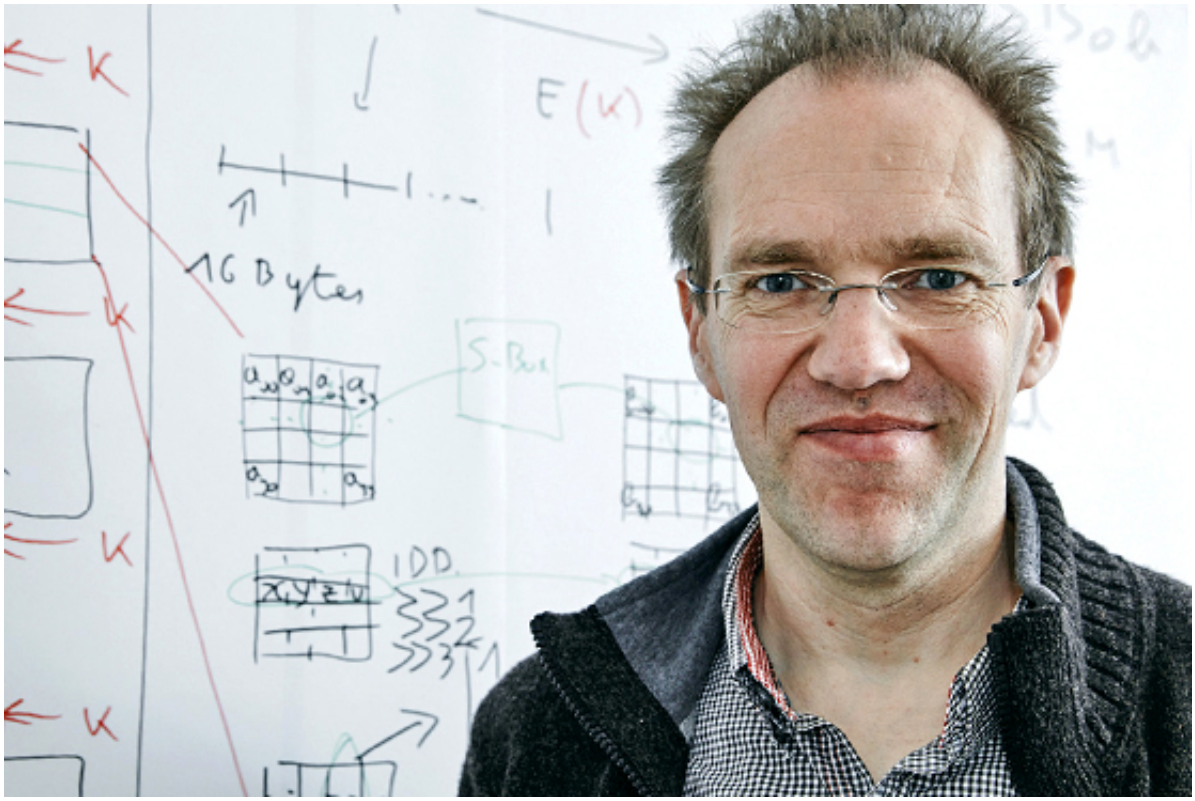
NIST (National INstitute of Standards and Technology) 在1997年9月12日公开征集更高效更安全的替代DES加密算法，第一轮共有15种算法入选，其中5种算法入围了决赛，分别是MARS，RC6，Rijndael，Serpent和Twofish。又经过3年的验证、评测及公众讨论之后Rijndael算法最终入选。



Rijndael算法

Rijndael算法是由比利时学者Joan Daemen和Vincent Rijmen所提出的，算法的名字就由两位作者的名字组合而成。Rijndael的优势在于集安全性、性能、效率、可实现性及灵活性与一体。

Joan Daemen和Vincent Rijmen


K
E(k)
1 ... 1
16 Bytes
S-Box
1DD.
K
K
K

AES vs Rijndael

Rijndael算法支持多种分组及密钥长度，介于128-256之间所有32的倍数均可，最小支持128位，最大256位，共25种组合。而AES标准支持的分组大小固定为128位，密钥长度有3种选择：128位、192位及256位。

加密实例

下面针对16字节的简单明文字串“0011223344....eeff”，分别用AES-128/AES-192及AES-256进行加密运算：

AES-128

密钥选用16字节长的简单字串：“00010203....0e0f” 来，上面的明文经过加密变换后成为"69c4e0d8....6089"。

```
plain :  00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff
key   :  00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
cypher:  69 c4 e0 d8 6a 7b 04 30 d8 cd b7 80 70 b4 c5 5a
```

AES-192

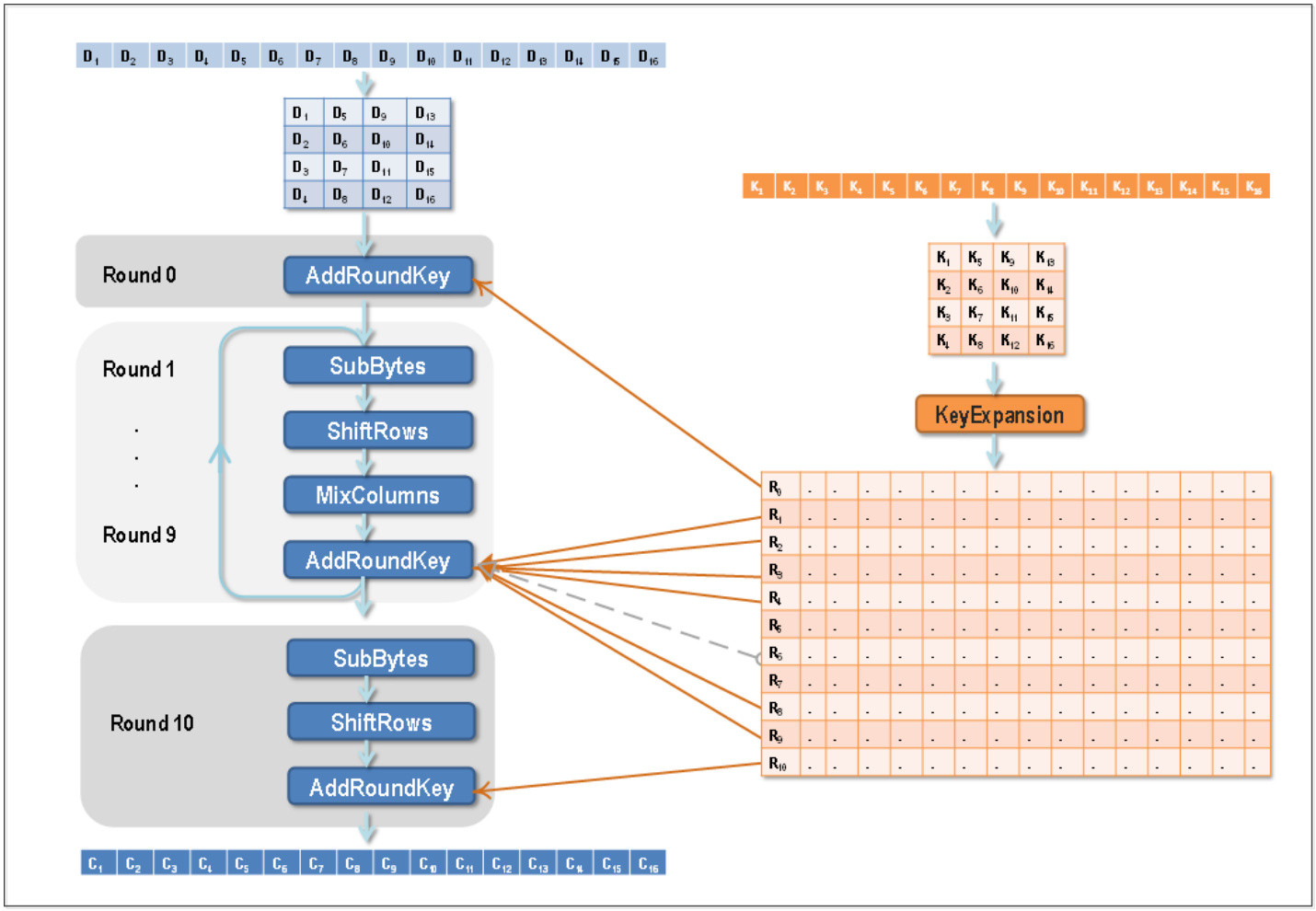
```
plain :  00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff
key   :  00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d .. .. .. 17
cypher:  dd a9 7c a4 86 4c df e0 6e af 70 a0 ec 0d 71 91
```

AES-256

```
plain :  00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff
key   :  00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d .. .. .. 17 .. .. .. 1f
cypher:  8e a2 b7 ca 51 67 45 bf ea fc 49 90 4b 49 60 89
```

总体结构

Rijndael算法是基于代换-置换网络（SPN，Substitution-permutation network）的迭代算法。明文数据经过多轮次的转换后方能生成密文，每个轮次的转换操作由轮函数定义。轮函数任务就是根据密钥编排序列（即轮密码）对数据进行不同的代换及置换等操作。



图左侧为轮函数的流程，主要包含4种主要运算操作：字节代换(SubByte)、行移位(ShiftRow)、列混合(MixColumn)、轮密钥加(AddRoundKey)。图右侧为密钥编排方案，在Rijndael中称为密钥扩展算法（KeyExpansion）。

AES标准算法将128位的明文，以特定次序生成一个4x4的矩阵（每个元素是一个字节，8位），即初始状态（state），经由轮函数的迭代转换之后又将作为下一轮迭代的输入继续参与运算直到迭代结束。

Rijndael算法支持大于128位的明文分组，所以需要列数更多的矩阵来描述。Rijndael轮函数的运算是在特殊定义的有限域GF(256)上进行的。有限域（Finite Field）又名伽罗瓦域（Galois field），简单言之就是一个满足特定规则的集合，集合中的元素可以进行加减乘除运算，且运算结果也是属于此集合。更详细有有关Rijndael算法的数学描述，可以参阅本文最后所罗列的参考资料，在此不做熬述。

轮函数

我们已经得知轮函数主要包含4种运算，但不同的运算轮所做的具体运的算组合并不相同。主要区别是初始轮（Round: 0）和最后一轮（Round: Nr），所有中间轮的运算都是相同的，会依次进行4种运算，即：

1. 字节代换(SubByte)
2. 行移位(ShiftRow)
3. 列混合(MixColumn)
4. 轮密钥加(AddRoundKey)

根据Rinjdael算法的定义，加密轮数会针对不同的分组及不同的密钥长度选择不同的数值：

N_r UNIT: DWORD	$N_b = 4$ (AES)	$N_b = 6$	$N_b = 8$
$N_k = 4$	10 (AES-128)	12	14
$N_k = 6$	12 (AES-192)	12	14
$N_k = 8$	14 (AES-256)	14	14

N_b：分组大小（128/192/256位）
N_k：密钥大小（128/192/256位）
N_r：迭代轮数

AES标准只支持128位分组（Nb = 4）的情况。

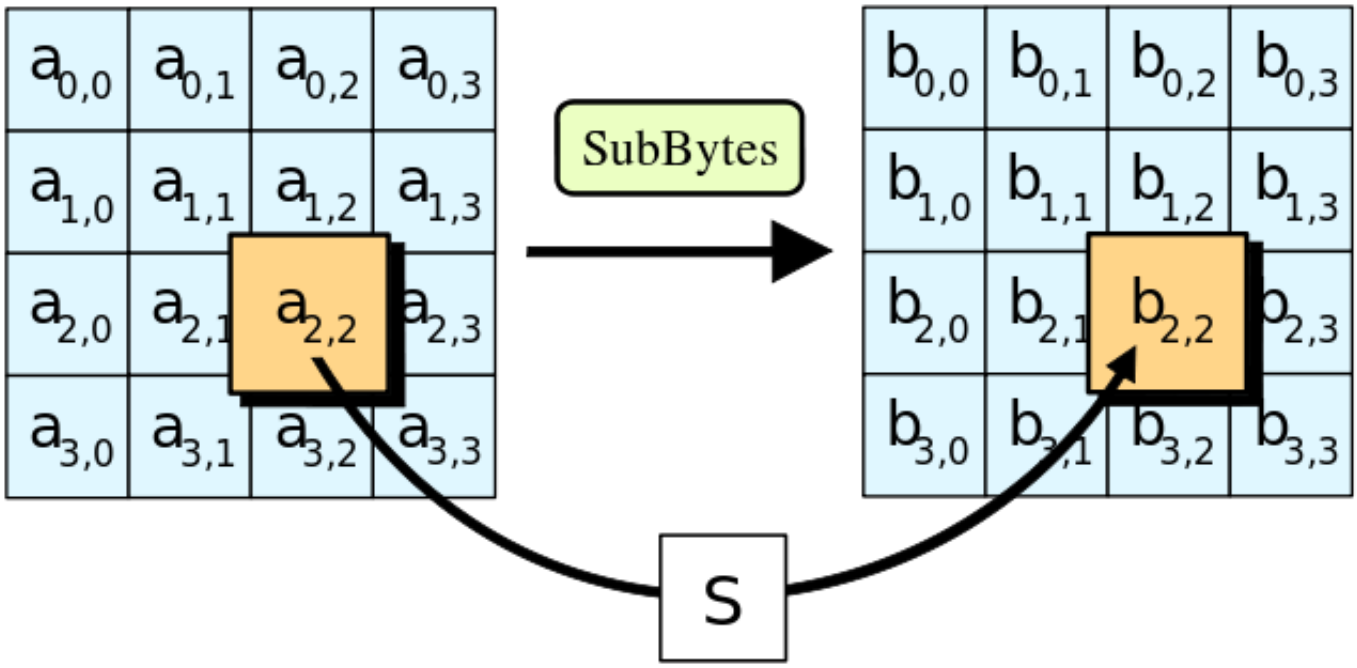
轮函数的实现代码如下，直接实现在加密函数内部循环中：

```
int aes_encrypt(AES_CYPHER_T mode, uint8_t *data, int len, uint8_t *key)
{
    uint8_t w[4 * 4 * 15] = {0}; /* round key */
    uint8_t s[4 * 4] = {0}; /* state */

    int nr, i, j;

    /* key expansion */
    aes_key_expansion(mode, key, w);

    /* start data cypher loop over input buffer */
    for (i = 0; i < len; i += 4 * g_aes_nb[mode]) {

        /* init state from user buffer (plaintext) */
        for (j = 0; j < 4 * g_aes_nb[mode]; j++)
            s[j] = data[i + j];

        /* start AES cypher loop over all AES rounds */
        for (nr = 0; nr <= g_aes_rounds[mode]; nr++) {

            if (nr > 0) {

                /* do SubBytes */
                aes_sub_bytes(mode, s);

                /* do ShiftRows */
                aes_shift_rows(mode, s);

                if (nr < g_aes_rounds[mode]) {
                    /* do MixColumns */
                    aes_mix_columns(mode, s);
                }
            }

            /* do AddRoundKey */
            aes_add_round_key(mode, s, w, nr);
        }

        /* save state (cypher) to user buffer */
        for (j = 0; j < 4 * g_aes_nb[mode]; j++)
            data[i + j] = s[j];
    }

    return 0;
}
```

动画演示加密过程

Enrique Zabala创建了一个AES-128加密算法的动画演示，清楚、直观地介绍了轮函数执行的过程。点击可直接观看。

轮函数拆解：字节代换（Substitute Bytes）



字节代换（SubBytes）是对state矩阵中的每一个独立元素于**置换盒** （Substitution-box，S盒）中进行查找并以此替换输入状态的操作。字节代换是可逆的非线性变换，也是AES运算组中唯一的非线性变换。字节代换逆操作也是通过逆向置换盒的查找及替换来完成的。

S盒是事先设计好的16x16的查询表，即256个元素。其设计不是随意的，要根据设计原则严格计算求得，不然无法保证算法的安全性。既然是S盒是计算得来，所以字节代换的操作完全可以通过计算来完成，不过通过S盒查表操作更方便快捷，图中所示就是通过S盒查找对应元素进行的替换操作。

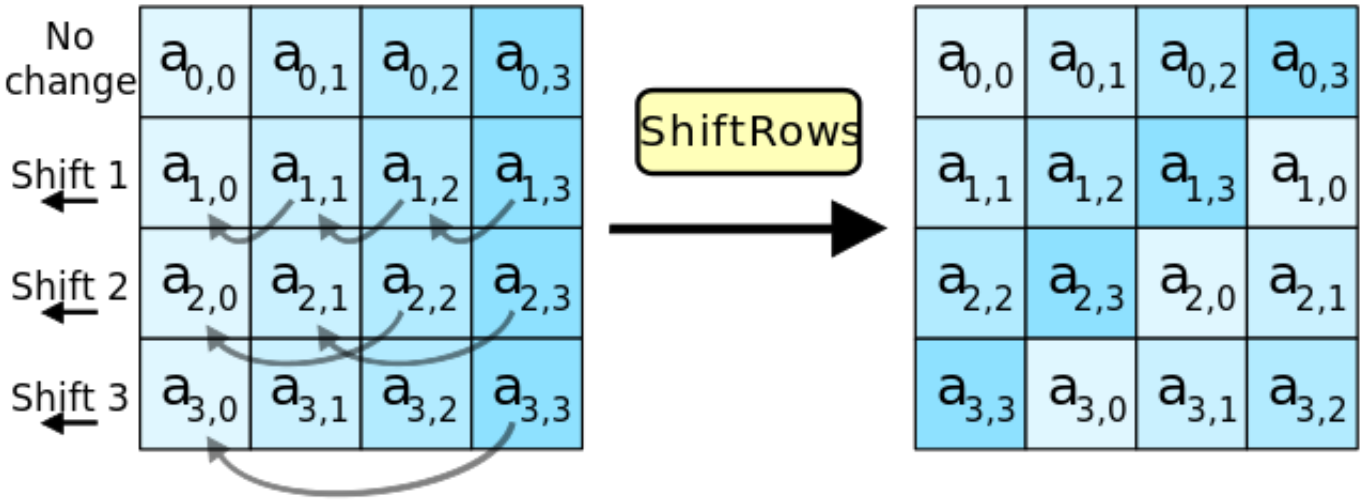
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

```
void aes_sub_bytes(AES_CYPHER_T mode, uint8_t *state)
{
    int i, j;

    for (i = 0; i < g_aes_nb[mode]; i++) {
        for (j = 0; j < 4; j++) {
            state[i * 4 + j] = aes_sub_sbox(state[i * 4 + j]);
        }
    }
}
```

实例说明：

```
   input:  00 10 20 30 40 50 60 70 80 90 a0 b0 c0 d0 e0 f0
     sub:  63 ca b7 04 09 53 d0 51 cd 60 e0 e7 ba 70 e1 8c
```

轮函数拆解：行移位（Shift Rows）



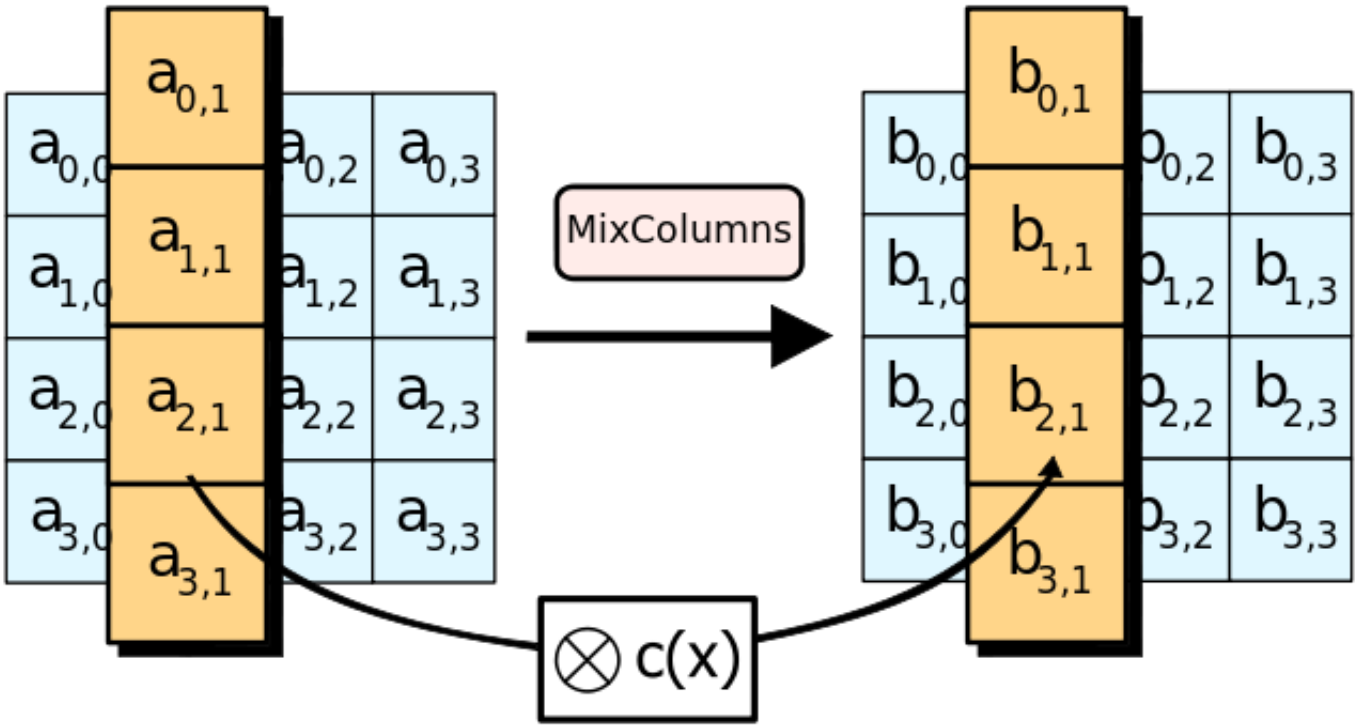
行移位主要目的是实现字节在每一行的扩散，属于线性变换。

```
void aes_shift_rows(AES_CYPHER_T mode, uint8_t *state)
{
    uint8_t *s = (uint8_t *)state;
    int i, j, r;

    for (i = 1; i < g_aes_nb[mode]; i++) {
        for (j = 0; j < i; j++) {
            uint8_t tmp = s[i];
            for (r = 0; r < g_aes_nb[mode]; r++) {
                s[i + r * 4] = s[i + (r + 1) * 4];
            }
            s[i + (g_aes_nb[mode] - 1) * 4] = tmp;
        }
    }
}
```

实例说明：

```
     sub:  63 ca b7 04 09 53 d0 51 cd 60 e0 e7 ba 70 e1 8c
   shift:  63 53 e0 8c 09 60 e1 04 cd 70 b7 51 ba ca d0 e7
```

轮函数拆解：列混合（Mix Columns）



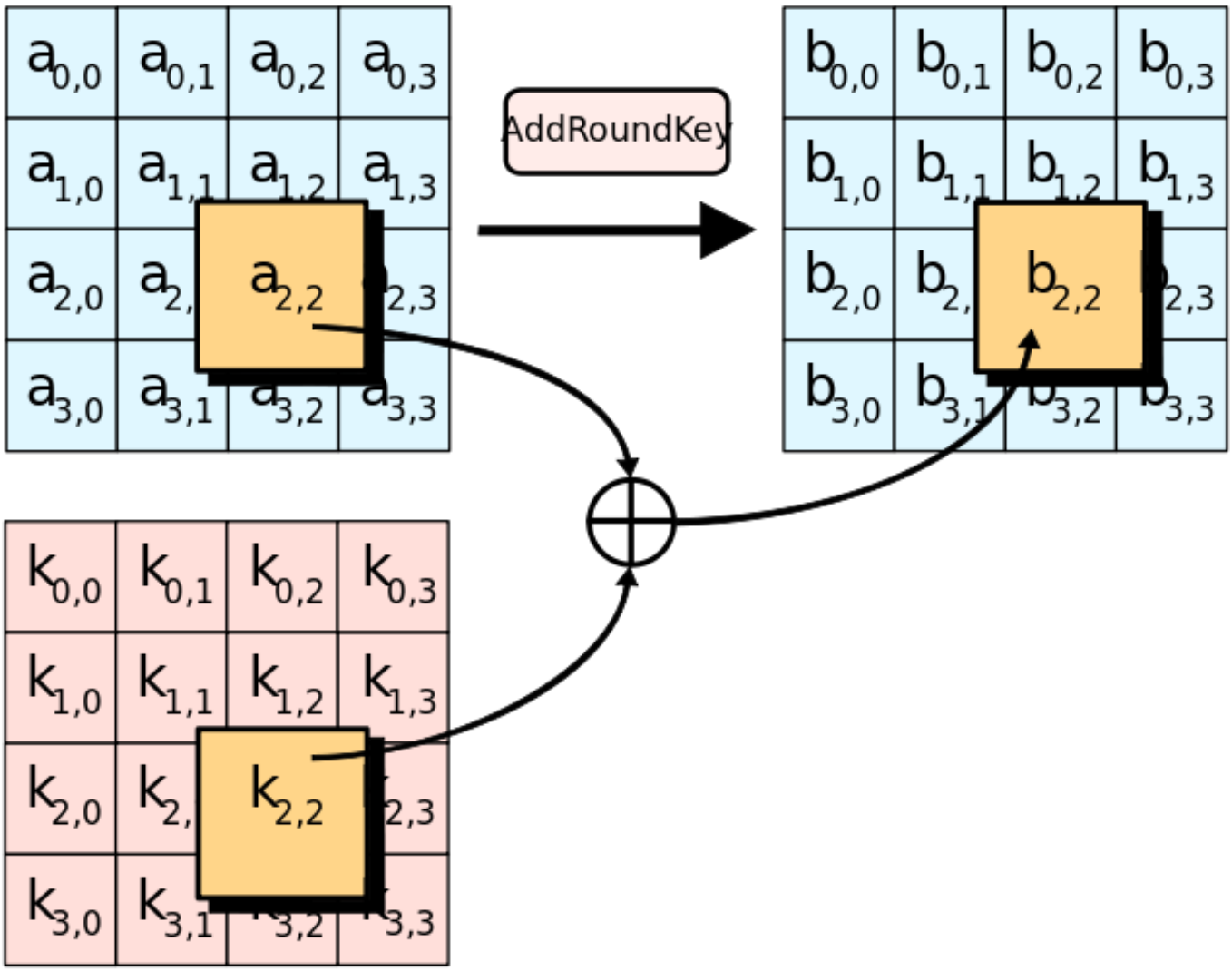
列混合是通过将state矩阵与常矩阵C相乘以达成在列上的扩散，属于代替变换。列混合是Rijndael算法中最复杂的一步，其实质是在有限域GF(256)上的多项式乘法运算。

```
void aes_mix_columns(AES_CYPHER_T mode, uint8_t *state)
{
    uint8_t y[16] = { 2, 3, 1, 1,  1, 2, 3, 1,  1, 1, 2, 3,  3, 1, 1, 2};
    uint8_t s[4];
    int i, j, r;

    for (i = 0; i < g_aes_nb[mode]; i++) {
        for (r = 0; r < 4; r++) {
            s[r] = 0;
            for (j = 0; j < 4; j++) {
                s[r] = s[r] ^ aes_mul(state[i * 4 + j], y[r * 4 + j]);
            }
        }
        for (r = 0; r < 4; r++) {
            state[i * 4 + r] = s[r];
        }
    }
}
```

实例说明：

```
   shift:  63 53 e0 8c 09 60 e1 04 cd 70 b7 51 ba ca d0 e7
     mix:  5f 72 64 15 57 f5 bc 92 f7 be 3b 29 1d b9 f9 1a
```

轮函数拆解：轮密钥加（Add Round Key）



密钥加是将轮密钥简单地与状态进行逐比特异或。实现代码如下：

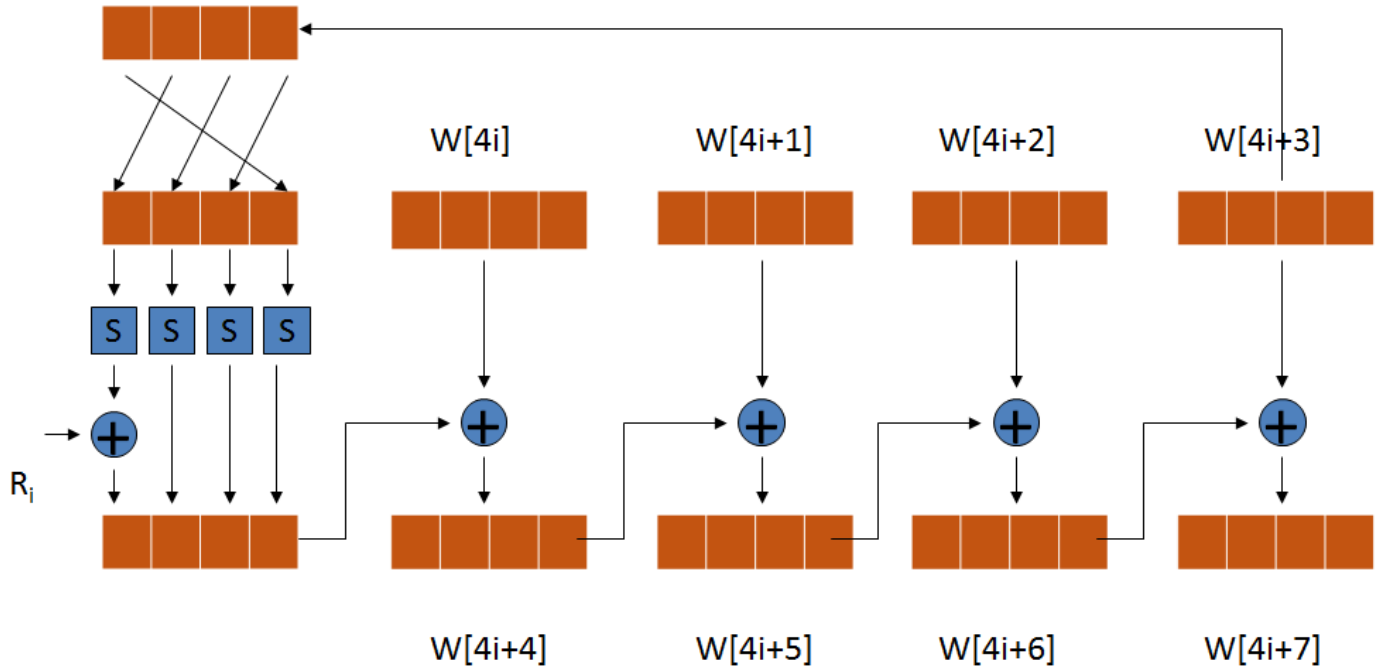
```
void aes_add_round_key(AES_CYPHER_T mode, uint8_t *state,
                       uint8_t *round, int nr)
{
    uint32_t *w = (uint32_t *)round;
    uint32_t *s = (uint32_t *)state;
    int i;

    for (i = 0; i < g_aes_nb[mode]; i++) {
        s[i] ^= w[nr * g_aes_nb[mode] + i];
    }
}
```

实例说明：

```
    mix:  5f 72 64 15 57 f5 bc 92 f7 be 3b 29 1d b9 f9 1a
  round:  d6 aa 74 fd d2 af 72 fa da a6 78 f1 d6 ab 76 fe
  state:  89 d8 10 e8 85 5a ce 68 2d 18 43 d8 cb 12 8f e4
```

密钥扩展算法（Key Expansion）

密钥扩展算法是Rijndael的密钥编排实现算法，其目的是根据种子密钥（用户密钥）生成多组轮密钥。轮密钥为多组128位密钥，对应不同密钥长度，分别是11，13，15组。



实现代码：

```
/*
 * nr: number of rounds
 * nb: number of columns comprising the state, nb = 4 dwords (16 bytes)
 * nk: number of 32-bit words comprising cipher key, nk = 4, 6, 8 (KeyLength/(4*8)
)
 */

void aes_key_expansion(AES_CYPHER_T mode, uint8_t *key, uint8_t *round)
{
    uint32_t *w = (uint32_t *)round;
    uint32_t  t;
    int      i = 0;

    do {
        w[i] = *((uint32_t *)&key[i * 4 + 0]);
    } while (++i < g_aes_nk[mode]);

    do {
        if ((i % g_aes_nk[mode]) == 0) {
            t = aes_rot_dword(w[i - 1]);
            t = aes_sub_dword(t);
            t = t ^ aes_swap_dword(g_aes_rcon[i/g_aes_nk[mode] - 1]);
        } else if (g_aes_nk[mode] > 6 && (i % g_aes_nk[mode]) == 4) {
            t = aes_sub_dword(w[i - 1]);
        } else {
            t = w[i - 1];
        }
        w[i] = w[i - g_aes_nk[mode]] ^ t;

    } while (++i < g_aes_nb[mode] * (g_aes_rounds[mode] + 1));

    /* key can be discarded (or zeroed) from memory */
}
```

以AES-128为例，从128位种子密钥生成11组轮密钥（每组128位）：

```
Input:
    key :  00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
Key Expansion:
    00:  rs: 00010203
    01:  rs: 04050607
    02:  rs: 08090a0b
    03:  rs: 0c0d0e0f
    04:  rot: 0d0e0f0c sub: d7ab76fe rcon: 01000000 xor: fe76abd6 rs: d6aa74fd
    05:  equ: d6aa74fd rs: d2af72fa
    06:  equ: d2af72fa rs: daa678f1
    07:  equ: daa678f1 rs: d6ab76fe
    08:  rot: ab76fed6 sub: 6238bbf6 rcon: 02000000 xor: f6bb3860 rs: b692cf0b
    09:  equ: b692cf0b rs: 643dbdf1
    10:  equ: 643dbdf1 rs: be9bc500
    11:  equ: be9bc500 rs: 6830b3fe
    12:  rot: 30b3fe68 sub: 046dbb45 rcon: 04000000 xor: 45bb6d00 rs: b6ff744e
    13:  equ: b6ff744e rs: d2c2c9bf
    14:  equ: d2c2c9bf rs: 6c590cbf
    15:  equ: 6c590cbf rs: 0469bf41
    16:  rot: 69bf4104 sub: f90883f2 rcon: 08000000 xor: f28308f1 rs: 47f7f7bc
    17:  equ: 47f7f7bc rs: 95353e03
    18:  equ: 95353e03 rs: f96c32bc
    19:  equ: f96c32bc rs: fd058dfd
    20:  rot: 058dfdfd sub: 6b5d5454 rcon: 10000000 xor: 54545d7b rs: 3caaa3e8
    21:  equ: 3caaa3e8 rs: a99f9deb
    22:  equ: a99f9deb rs: 50f3af57
    23:  equ: 50f3af57 rs: adf622aa
    24:  rot: f622aaad sub: 4293ac95 rcon: 20000000 xor: 95ac9362 rs: 5e390f7d
    25:  equ: 5e390f7d rs: f7a69296
    26:  equ: f7a69296 rs: a7553dc1
    27:  equ: a7553dc1 rs: 0aa31f6b
    28:  rot: a31f6b0a sub: 0ac07f67 rcon: 40000000 xor: 677fc04a rs: 14f9701a
    29:  equ: 14f9701a rs: e35fe28c
    30:  equ: e35fe28c rs: 440adf4d
    31:  equ: 440adf4d rs: 4ea9c026
    32:  rot: a9c0264e sub: d3baf72f rcon: 80000000 xor: 2ff7ba53 rs: 47438735
    33:  equ: 47438735 rs: a41c65b9
    34:  equ: a41c65b9 rs: e016baf4
    35:  equ: e016baf4 rs: aebf7ad2
    36:  rot: bf7ad2ae sub: 08dab5e4 rcon: 1b000000 xor: e4b5da13 rs: 549932d1
    37:  equ: 549932d1 rs: f0855768
    38:  equ: f0855768 rs: 1093ed9c
    39:  equ: 1093ed9c rs: be2c974e
    40:  rot: 2c974ebe sub: 71882fae rcon: 36000000 xor: ae2f8847 rs: 13111d7f
    41:  equ: 13111d7f rs: e3944a17
    42:  equ: e3944a17 rs: f307a78b
    43:  equ: f307a78b rs: 4d2b30c5
```

加密过程实例

```
Encrypting block ...
 Round 0:
   input:  00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff
   round:  00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
   state:  00 10 20 30 40 50 60 70 80 90 a0 b0 c0 d0 e0 f0
 Round 1:
   input:  00 10 20 30 40 50 60 70 80 90 a0 b0 c0 d0 e0 f0
     sub:  63 ca b7 04 09 53 d0 51 cd 60 e0 e7 ba 70 e1 8c
   shift:  63 53 e0 8c 09 60 e1 04 cd 70 b7 51 ba ca d0 e7
     mix:  5f 72 64 15 57 f5 bc 92 f7 be 3b 29 1d b9 f9 1a
   round:  d6 aa 74 fd d2 af 72 fa da a6 78 f1 d6 ab 76 fe
   state:  89 d8 10 e8 85 5a ce 68 2d 18 43 d8 cb 12 8f e4
 Round 2:
   input:  89 d8 10 e8 85 5a ce 68 2d 18 43 d8 cb 12 8f e4
     sub:  a7 61 ca 9b 97 be 8b 45 d8 ad 1a 61 1f c9 73 69
   shift:  a7 be 1a 69 97 ad 73 9b d8 c9 ca 45 1f 61 8b 61
     mix:  ff 87 96 84 31 d8 6a 51 64 51 51 fa 77 3a d0 09
   round:  b6 92 cf 0b 64 3d bd f1 be 9b c5 00 68 30 b3 fe
   state:  49 15 59 8f 55 e5 d7 a0 da ca 94 fa 1f 0a 63 f7
 Round 3:
   input:  49 15 59 8f 55 e5 d7 a0 da ca 94 fa 1f 0a 63 f7
     sub:  3b 59 cb 73 fc d9 0e e0 57 74 22 2d c0 67 fb 68
   shift:  3b d9 22 68 fc 74 fb 73 57 67 cb e0 c0 59 0e 2d
     mix:  4c 9c 1e 66 f7 71 f0 76 2c 3f 86 8e 53 4d f2 56
   round:  b6 ff 74 4e d2 c2 c9 bf 6c 59 0c bf 04 69 bf 41
   state:  fa 63 6a 28 25 b3 39 c9 40 66 8a 31 57 24 4d 17
 Round 4:
   input:  fa 63 6a 28 25 b3 39 c9 40 66 8a 31 57 24 4d 17
     sub:  2d fb 02 34 3f 6d 12 dd 09 33 7e c7 5b 36 e3 f0
   shift:  2d 6d 7e f0 3f 33 e3 34 09 36 02 dd 5b fb 12 c7
     mix:  63 85 b7 9f fc 53 8d f9 97 be 47 8e 75 47 d6 91
   round:  47 f7 f7 bc 95 35 3e 03 f9 6c 32 bc fd 05 8d fd
   state:  24 72 40 23 69 66 b3 fa 6e d2 75 32 88 42 5b 6c
 Round 5:
   input:  24 72 40 23 69 66 b3 fa 6e d2 75 32 88 42 5b 6c
     sub:  36 40 09 26 f9 33 6d 2d 9f b5 9d 23 c4 2c 39 50
   shift:  36 33 9d 50 f9 b5 39 26 9f 2c 09 2d c4 40 6d 23
     mix:  f4 bc d4 54 32 e5 54 d0 75 f1 d6 c5 1d d0 3b 3c
   round:  3c aa a3 e8 a9 9f 9d eb 50 f3 af 57 ad f6 22 aa
   state:  c8 16 77 bc 9b 7a c9 3b 25 02 79 92 b0 26 19 96
 Round 6:
   input:  c8 16 77 bc 9b 7a c9 3b 25 02 79 92 b0 26 19 96
     sub:  e8 47 f5 65 14 da dd e2 3f 77 b6 4f e7 f7 d4 90
   shift:  e8 da b6 90 14 77 d4 65 3f f7 f5 e2 e7 47 dd 4f
     mix:  98 16 ee 74 00 f8 7f 55 6b 2c 04 9c 8e 5a d0 36
```

```
    round:  5e 39 0f 7d f7 a6 92 96 a7 55 3d c1 0a a3 1f 6b
    state:  c6 2f e1 09 f7 5e ed c3 cc 79 39 5d 84 f9 cf 5d
 Round 7:
    input:  c6 2f e1 09 f7 5e ed c3 cc 79 39 5d 84 f9 cf 5d
      sub:  b4 15 f8 01 68 58 55 2e 4b b6 12 4c 5f 99 8a 4c
    shift:  b4 58 12 4c 68 b6 8a 01 4b 99 f8 2e 5f 15 55 4c
      mix:  c5 7e 1c 15 9a 9b d2 86 f0 5f 4b e0 98 c6 34 39
    round:  14 f9 70 1a e3 5f e2 8c 44 0a df 4d 4e a9 c0 26
    state:  d1 87 6c 0f 79 c4 30 0a b4 55 94 ad d6 6f f4 1f
 Round 8:
    input:  d1 87 6c 0f 79 c4 30 0a b4 55 94 ad d6 6f f4 1f
      sub:  3e 17 50 76 b6 1c 04 67 8d fc 22 95 f6 a8 bf c0
    shift:  3e 1c 22 c0 b6 fc bf 76 8d a8 50 67 f6 17 04 95
      mix:  ba a0 3d e7 a1 f9 b5 6e d5 51 2c ba 5f 41 4d 23
    round:  47 43 87 35 a4 1c 65 b9 e0 16 ba f4 ae bf 7a d2
    state:  fd e3 ba d2 05 e5 d0 d7 35 47 96 4e f1 fe 37 f1
 Round 9:
    input:  fd e3 ba d2 05 e5 d0 d7 35 47 96 4e f1 fe 37 f1
      sub:  54 11 f4 b5 6b d9 70 0e 96 a0 90 2f a1 bb 9a a1
    shift:  54 d9 90 a1 6b a0 9a b5 96 bb f4 0e a1 11 70 2f
      mix:  e9 f7 4e ec 02 30 20 f6 1b f2 cc f2 35 3c 21 c7
    round:  54 99 32 d1 f0 85 57 68 10 93 ed 9c be 2c 97 4e
    state:  bd 6e 7c 3d f2 b5 77 9e 0b 61 21 6e 8b 10 b6 89
 Round 10:
    input:  bd 6e 7c 3d f2 b5 77 9e 0b 61 21 6e 8b 10 b6 89
      sub:  7a 9f 10 27 89 d5 f5 0b 2b ef fd 9f 3d ca 4e a7
    shift:  7a d5 fd a7 89 ef 4e 27 2b ca 10 0b 3d 9f f5 9f
    round:  13 11 1d 7f e3 94 4a 17 f3 07 a7 8b 4d 2b 30 c5
    state:  69 c4 e0 d8 6a 7b 04 30 d8 cd b7 80 70 b4 c5 5a
Output:
  cypher:  69 c4 e0 d8 6a 7b 04 30 d8 cd b7 80 70 b4 c5 5a
```

解密轮函数

对Rijndael算法来说解密过程就是加密过程的逆向过程，其解密轮函数实现如下：

```
int aes_decrypt(AES_CYPHER_T mode, uint8_t *data, int len, uint8_t *key)
{
    uint8_t w[4 * 4 * 15] = {0}; /* round key */
    uint8_t s[4 * 4] = {0}; /* state */

    int nr, i, j;

    /* key expansion */
    aes_key_expansion(mode, key, w);

    /* start data cypher loop over input buffer */
    for (i = 0; i < len; i += 4 * g_aes_nb[mode]) {

        /* init state from user buffer (cyphertext) */
        for (j = 0; j < 4 * g_aes_nb[mode]; j++)
            s[j] = data[i + j];

        /* start AES cypher loop over all AES rounds */
        for (nr = g_aes_rounds[mode]; nr >= 0; nr--) {

            /* do AddRoundKey */
            aes_add_round_key(mode, s, w, nr);

            if (nr > 0) {
                if (nr < g_aes_rounds[mode]) {
                    /* do MixColumns */
                    inv_mix_columns(mode, s);
                }

                /* do ShiftRows */
                inv_shift_rows(mode, s);

                /* do SubBytes */
                inv_sub_bytes(mode, s);
            }
        }

        /* save state (cypher) to user buffer */
        for (j = 0; j < 4 * g_aes_nb[mode]; j++)
            data[i + j] = s[j];
    }

    return 0;
}
```

解密过程实例

```
Decrypting block ...
 Round 10:
   input:  69 c4 e0 d8 6a 7b 04 30 d8 cd b7 80 70 b4 c5 5a
   round:  13 11 1d 7f e3 94 4a 17 f3 07 a7 8b 4d 2b 30 c5
   shift:  7a d5 fd a7 89 ef 4e 27 2b ca 10 0b 3d 9f f5 9f
     sub:  7a 9f 10 27 89 d5 f5 0b 2b ef fd 9f 3d ca 4e a7
   state:  bd 6e 7c 3d f2 b5 77 9e 0b 61 21 6e 8b 10 b6 89
 Round 9:
   input:  bd 6e 7c 3d f2 b5 77 9e 0b 61 21 6e 8b 10 b6 89
   round:  54 99 32 d1 f0 85 57 68 10 93 ed 9c be 2c 97 4e
     mix:  e9 f7 4e ec 02 30 20 f6 1b f2 cc f2 35 3c 21 c7
   shift:  54 d9 90 a1 6b a0 9a b5 96 bb f4 0e a1 11 70 2f
     sub:  54 11 f4 b5 6b d9 70 0e 96 a0 90 2f a1 bb 9a a1
   state:  fd e3 ba d2 05 e5 d0 d7 35 47 96 4e f1 fe 37 f1
 Round 8:
   input:  fd e3 ba d2 05 e5 d0 d7 35 47 96 4e f1 fe 37 f1
   round:  47 43 87 35 a4 1c 65 b9 e0 16 ba f4 ae bf 7a d2
     mix:  ba a0 3d e7 a1 f9 b5 6e d5 51 2c ba 5f 41 4d 23
   shift:  3e 1c 22 c0 b6 fc bf 76 8d a8 50 67 f6 17 04 95
     sub:  3e 17 50 76 b6 1c 04 67 8d fc 22 95 f6 a8 bf c0
   state:  d1 87 6c 0f 79 c4 30 0a b4 55 94 ad d6 6f f4 1f
 Round 7:
   input:  d1 87 6c 0f 79 c4 30 0a b4 55 94 ad d6 6f f4 1f
   round:  14 f9 70 1a e3 5f e2 8c 44 0a df 4d 4e a9 c0 26
     mix:  c5 7e 1c 15 9a 9b d2 86 f0 5f 4b e0 98 c6 34 39
   shift:  b4 58 12 4c 68 b6 8a 01 4b 99 f8 2e 5f 15 55 4c
     sub:  b4 15 f8 01 68 58 55 2e 4b b6 12 4c 5f 99 8a 4c
   state:  c6 2f e1 09 f7 5e ed c3 cc 79 39 5d 84 f9 cf 5d
 Round 6:
   input:  c6 2f e1 09 f7 5e ed c3 cc 79 39 5d 84 f9 cf 5d
   round:  5e 39 0f 7d f7 a6 92 96 a7 55 3d c1 0a a3 1f 6b
     mix:  98 16 ee 74 00 f8 7f 55 6b 2c 04 9c 8e 5a d0 36
   shift:  e8 da b6 90 14 77 d4 65 3f f7 f5 e2 e7 47 dd 4f
     sub:  e8 47 f5 65 14 da dd e2 3f 77 b6 4f e7 f7 d4 90
   state:  c8 16 77 bc 9b 7a c9 3b 25 02 79 92 b0 26 19 96
 Round 5:
   input:  c8 16 77 bc 9b 7a c9 3b 25 02 79 92 b0 26 19 96
   round:  3c aa a3 e8 a9 9f 9d eb 50 f3 af 57 ad f6 22 aa
     mix:  f4 bc d4 54 32 e5 54 d0 75 f1 d6 c5 1d d0 3b 3c
   shift:  36 33 9d 50 f9 b5 39 26 9f 2c 09 2d c4 40 6d 23
     sub:  36 40 09 26 f9 33 6d 2d 9f b5 9d 23 c4 2c 39 50
   state:  24 72 40 23 69 66 b3 fa 6e d2 75 32 88 42 5b 6c
 Round 4:
   input:  24 72 40 23 69 66 b3 fa 6e d2 75 32 88 42 5b 6c
   round:  47 f7 f7 bc 95 35 3e 03 f9 6c 32 bc fd 05 8d fd
     mix:  63 85 b7 9f fc 53 8d f9 97 be 47 8e 75 47 d6 91
   shift:  2d 6d 7e f0 3f 33 e3 34 09 36 02 dd 5b fb 12 c7
     sub:  2d fb 02 34 3f 6d 12 dd 09 33 7e c7 5b 36 e3 f0
```

```
   state:  fa 63 6a 28 25 b3 39 c9 40 66 8a 31 57 24 4d 17
 Round 3:
   input:  fa 63 6a 28 25 b3 39 c9 40 66 8a 31 57 24 4d 17
   round:  b6 ff 74 4e d2 c2 c9 bf 6c 59 0c bf 04 69 bf 41
     mix:  4c 9c 1e 66 f7 71 f0 76 2c 3f 86 8e 53 4d f2 56
   shift:  3b d9 22 68 fc 74 fb 73 57 67 cb e0 c0 59 0e 2d
     sub:  3b 59 cb 73 fc d9 0e e0 57 74 22 2d c0 67 fb 68
   state:  49 15 59 8f 55 e5 d7 a0 da ca 94 fa 1f 0a 63 f7
Round 2:
   input:  49 15 59 8f 55 e5 d7 a0 da ca 94 fa 1f 0a 63 f7
   round:  b6 92 cf 0b 64 3d bd f1 be 9b c5 00 68 30 b3 fe
     mix:  ff 87 96 84 31 d8 6a 51 64 51 51 fa 77 3a d0 09
   shift:  a7 be 1a 69 97 ad 73 9b d8 c9 ca 45 1f 61 8b 61
     sub:  a7 61 ca 9b 97 be 8b 45 d8 ad 1a 61 1f c9 73 69
   state:  89 d8 10 e8 85 5a ce 68 2d 18 43 d8 cb 12 8f e4
 Round 1:
   input:  89 d8 10 e8 85 5a ce 68 2d 18 43 d8 cb 12 8f e4
   round:  d6 aa 74 fd d2 af 72 fa da a6 78 f1 d6 ab 76 fe
     mix:  5f 72 64 15 57 f5 bc 92 f7 be 3b 29 1d b9 f9 1a
   shift:  63 53 e0 8c 09 60 e1 04 cd 70 b7 51 ba ca d0 e7
     sub:  63 ca b7 04 09 53 d0 51 cd 60 e0 e7 ba 70 e1 8c
   state:  00 10 20 30 40 50 60 70 80 90 a0 b0 c0 d0 e0 f0
 Round 0:
   input:  00 10 20 30 40 50 60 70 80 90 a0 b0 c0 d0 e0 f0
   round:  00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
   state:  00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff
Output:
   plain:  00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff
```

算法设计思想

加密算法的一般设计准则

- 混淆 (Confusion) 最大限度地复杂化密文、明文与密钥之间的关系，通常用非线性变换算法达到最大化的混淆。
- 扩散 (Diffusion) 明文或密钥每变动一位将最大化地影响密文中的位数，通常采用线性变换算法达到最大化的扩散。

AES评判要求

NIST在征集算法的时候就提出了几项硬性要求：

- 分组加密算法：支持128位分组大小，128/192/256位密钥
- 安全性不低于3DES，但实施与执行要比3DES的更高效

- 优化过的ANSI C的实现代码
- KAT(Known-Answer tests)及MCT(Monte Carlo Tests)测试及验证
- 软件及硬件实现的便捷
- 可抵御已知攻击

Rijndael设计思想

1. 安全性（Security） 算法足够强，抗攻击
2. 经济性（Efficiency） 算法运算效率高
3. 密钥捷变（Key Agility） 更改密钥所引入的损失尽量小，即最小消耗的密钥扩展算法
4. 适应性 （Versatility） 适用于不同的CPU架构，软件或硬件平台的实现
5. 设计简单（Simplicity） 轮函数的设计精简，只是多轮迭代

S盒设计

S盒是由一个有限域GF(256)上的乘法求逆并串联线性仿射变换所构造出来的，不是一个随意构造的简单查询表。因其运算复杂，众多的AES 软件及硬件实现直接使用了查找表(LUP, Look-up table)，但查询表的方式并不适合所有场景，针对特定的硬件最小化面积设计需求，则要采用优化的组合逻辑以得到同价的S盒替换。

工作模式

分组加密算法是按分组大小来进行加解密操作的，如DES算法的分组是64位，而AES是128位，但实际明文的长度一般要远大于分组大小，这样的情况如何处理呢?

这正是"mode of operation"即工作模式要解决的问题：明文数据流怎样按分组大小切分，数据不对齐的情况怎么处理等等。

早在1981年，DES算法公布之后，NIST在标准文献FIPS 81中公布了4种工作模式：

- 电子密码本：Electronic Code Book Mode (ECB)
- 密码分组链接：Cipher Block Chaining Mode (CBC)
- 密文反馈：Cipher Feedback Mode (CFB)
- 输出反馈：Output Feedback Mode (OFB)

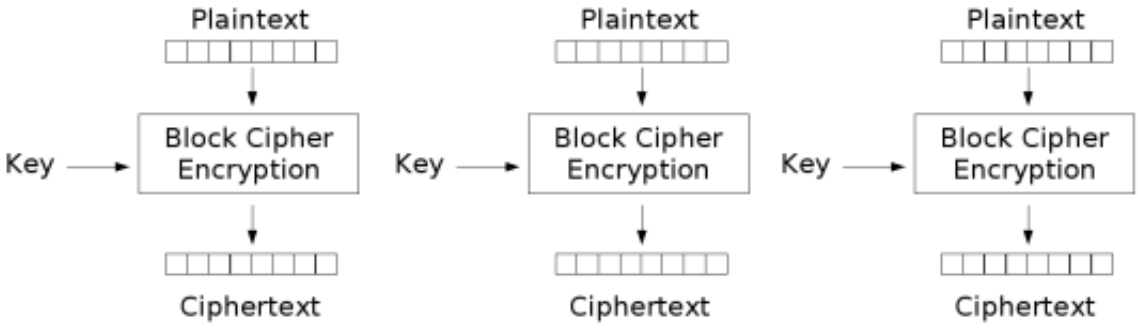
2001年又针对AES加入了新的工作模式：

- 计数器模式：Counter Mode (CTR)

后来又陆续引入其它新的工作模式。在此仅介绍几种常用的：

ECB：电子密码本模式

ECB模式只是将明文按分组大小切分，然后用同样的密钥正常加密切分好的明文分组。



Electronic Codebook (ECB) mode encryption

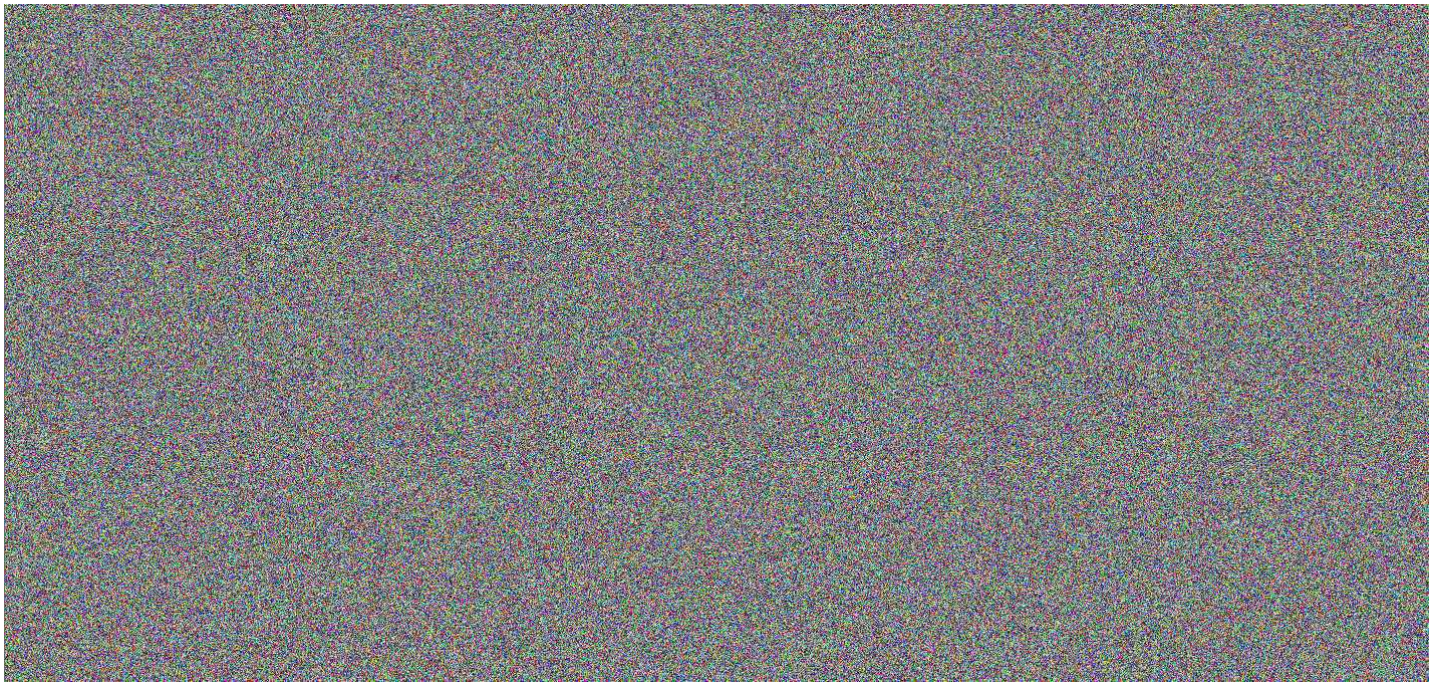
ECB的理想应用场景是短数据（如加密密钥）的加密。此模式的问题是无法隐藏原明文数据的模式，因为同样的明文分组加密得到的密文也是一样的。

举例来说明，下图为明文图片：

经ECB模式加密的图片：

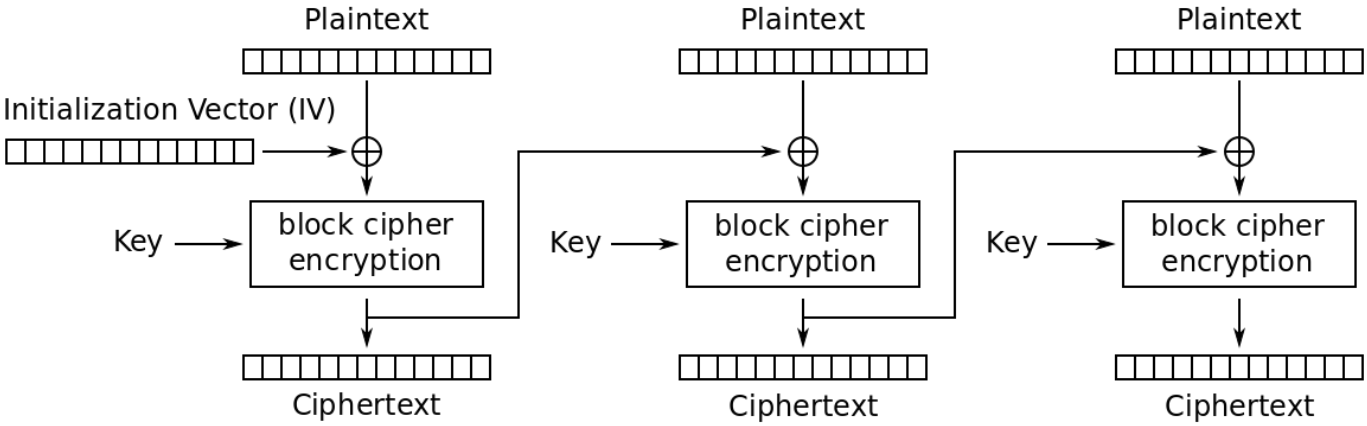
图中也正好验证了AES的扩散效果：作为局部图案的叶子，其红颜色在加密后扩散到了整张图片上。

经CBC模式加密的图片：



CBC：密码分组链接模式

此模式是1976年由IBM所发明，引入了IV（初始化向量：Initialization Vector）的概念。IV是长度为分组大小的一组随机，通常情况下不用保密，不过在大多数情况下，针对同一密钥不应多次使用同一组IV。 CBC要求第一个分组的明文在加密运算前先与IV进行异或；从第二组开始，所有的明文先与前一分组加密后的密文进行异或。[区块链(blockchain)的鼻祖！]



Cipher Block Chaining (CBC) mode encryption

CBC模式相比ECB实现了更好的模式隐藏，但因为其将密文引入运算，加解密操作无法并行操作。同时引入的IV向量，还需要加、解密双方共同知晓方可。

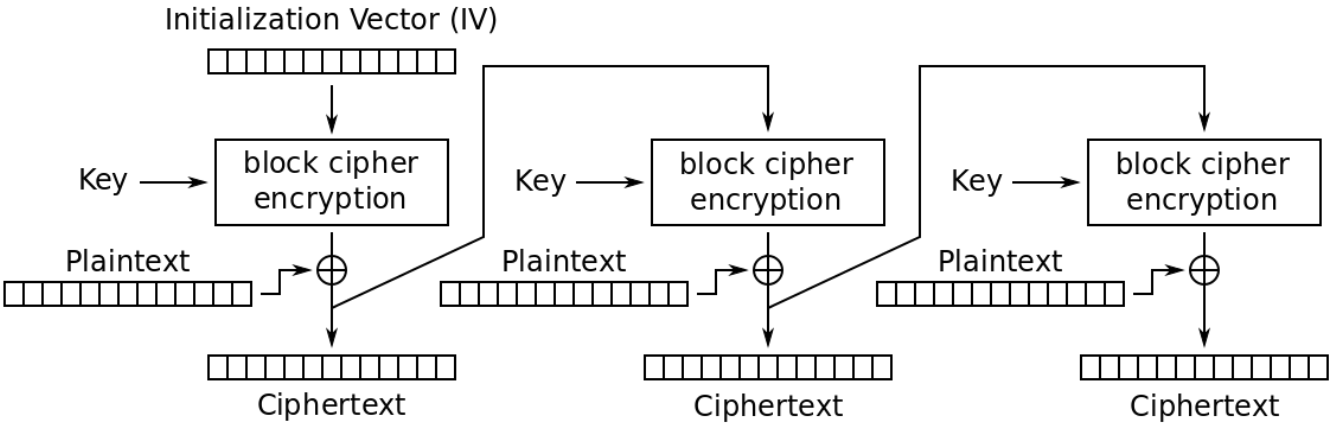
实现代码： ```c int aes*encrypt*cbc(AES*CYPHER*T mode, uint8*t *data, int len, uint8*t *key, uint8*t *iv) { *uint8*t w[4 * 4 * 15] = {0}; /* round key */ uint8*t s[4 * 4] = {0}; /* state */ uint8*t v[4 * 4] = {0}; /* iv */

```
int nr, i, j;

/* key expansion */
aes_key_expansion(mode, key, w);
memcpy(v, iv, sizeof(v));

/* start data cypher loop over input buffer */
for (i = 0; i < len; i += 4 * g_aes_nb[mode]) {
    /* init state from user buffer (plaintext) */
    for (j = 0; j < 4 * g_aes_nb[mode]; j++)
        s[j] = data[i + j] ^ v[j];

    /* start AES cypher loop over all AES rounds */
    for (nr = 0; nr <= g_aes_rounds[mode]; nr++) {

        if (nr > 0) {

            /* do SubBytes */
            aes_sub_bytes(mode, s);

            /* do ShiftRows */
            aes_shift_rows(mode, s);

            if (nr < g_aes_rounds[mode]) {
                /* do MixColumns */
                aes_mix_columns(mode, s);
            }
        }

        /* do AddRoundKey */
        aes_add_round_key(mode, s, w, nr);
    }

    /* save state (cypher) to user buffer */
    for (j = 0; j < 4 * g_aes_nb[mode]; j++)
        data[i + j] = v[j] = s[j];
}

return 0;
```

}```

CFB：密文反馈模式

与CBC模式类似，但不同的地方在于，CFB模式先生成密码流字典，然后用密码字典与明文进行异或操作并最终生成密文。后一分组的密码字典的生成需要前一分组的密文参与运算。

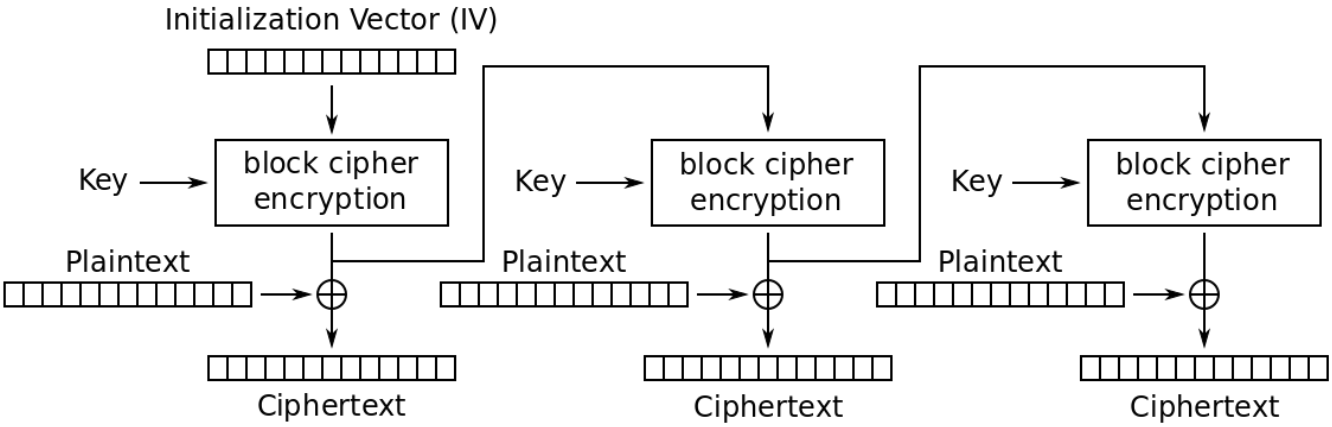


Cipher Feedback (CFB) mode encryption

CFB模式是用分组算法实现流算法，明文数据不需要按分组大小对齐。

OFB：输出反馈模式

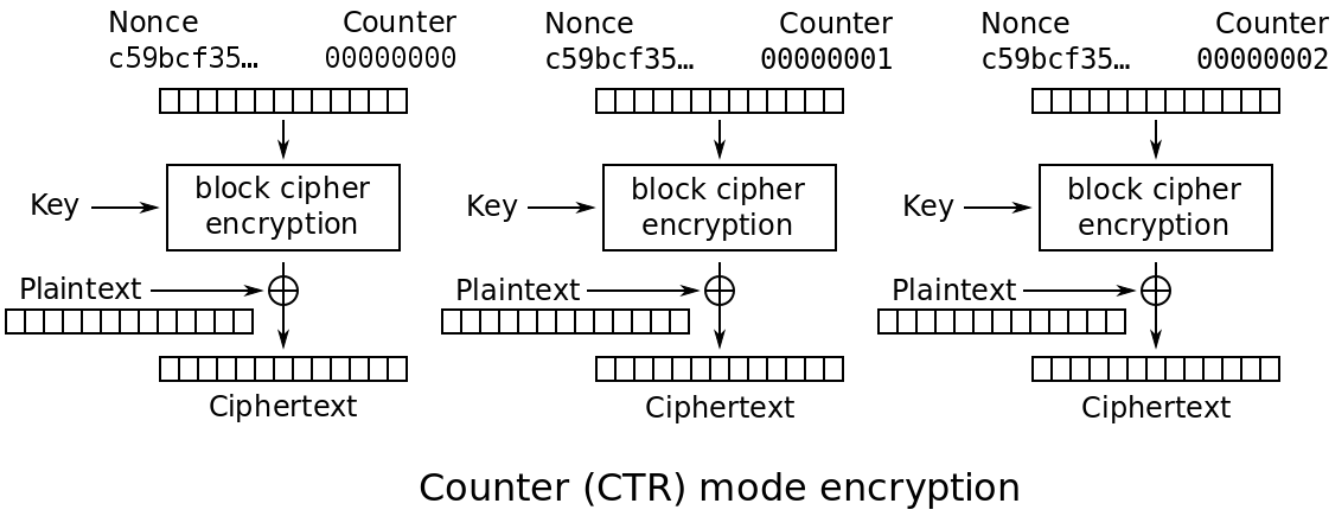
OFB模式与CFB模式不同的地方是：生成字典的时候会采用明文参与运算，CFB采用的是密文。



Output Feedback (OFB) mode encryption

CTR：计数器模式模式

CTR模式同样会产生流密码字典，但同是会引入一个计数，以保证任意长时间均不会产生重复输出。



Counter (CTR) mode encryption

CTR模式只需要实现加密算法以生成字典，明文数据与之异或后得到密文，反之便是解密过程。CTR模式可以采用并行算法处理以提升吞量，另外加密数据块的访问可以是随机的，与前后上下文无关。

CCM：Counter with CBC-MAC

CCM模式，全称是Counter with Cipher Block Chaining-Message Authentication Code，是CTR工作模式和CMAC认证算法的组合体，可以同时数据加密和鉴别服务。

明文数据通过CTR模式加密成密文，然后在密文后面再附加上认证数据，所以最终的密文会比明文要长。具体的加密流程如下描述：先对明文数据认证并产生一个tag，在后续加密过程中使用此tag和IV生成校验值U。然后用CTR模式来加密原输入明文数据，在密文的后面附上校验码U。

GCM：伽罗瓦计数器模式

类型CCM模式，GCM模式是CTR和GHASH的组合，GHASH操作定义为密文结果与密钥以及消息长度在GF（2^128）域上相乘。GCM比CCM的优势是在于更高并行度及更好的性能。TLS 1.2标准使用的就是AES-GCM算法，并且Intel CPU提供了GHASH的硬件加速功能。

硬件加速

AES作为主导的加密标准，其应用越来越广泛，特别是针对网络数据的加密需求，越来越多的硬件都集成AES 128/192/256位算法及不同的工作模式的硬件加速的实现。

AES_NI: X86架构

Intel于2010发发布了支持AES加速的CPU，实现了高阶的AES加解密指令即AES*NI：AES New Instructions。AES*NI包含6指令：其中4条用于加解密，2条用于密钥扩展。根据AES_NI白皮书中所说，

AES_NI可以带来2-3倍的性能提升。

Instruction	Description
AESENC	Perform one round of an AES encryption flow
AESENCLAST	Perform the last round of an AES encryption flow
AESDEC	Perform one round of an AES decryption flow
AESDECLAST	Perform the last round of an AES decryption flow
AESKEYGENASSIST	Assist in AES round key generation
AESIMC	Assist in AES Inverse Mix Columns

目前OpenSSL，Linux's Crypto API以及Windows Cryptography API中均已加入对AES_NI的支持。

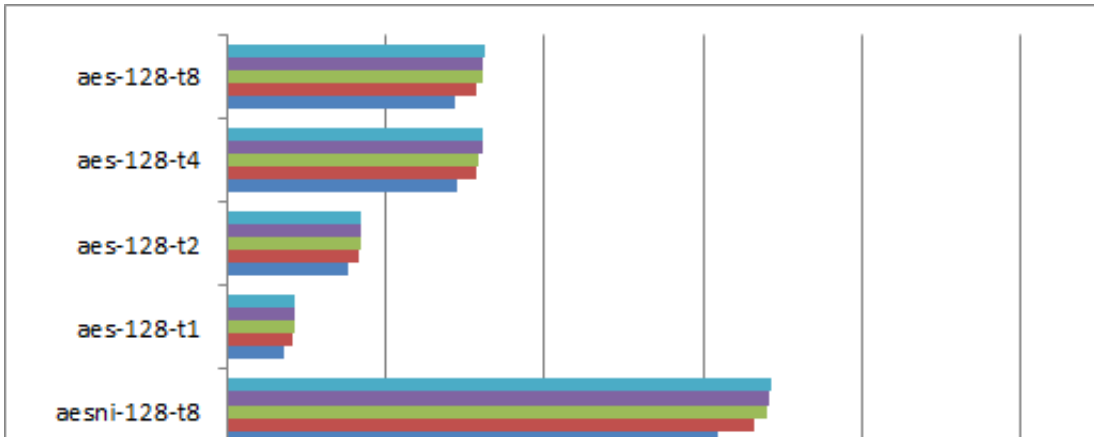
AES_NI: 测试

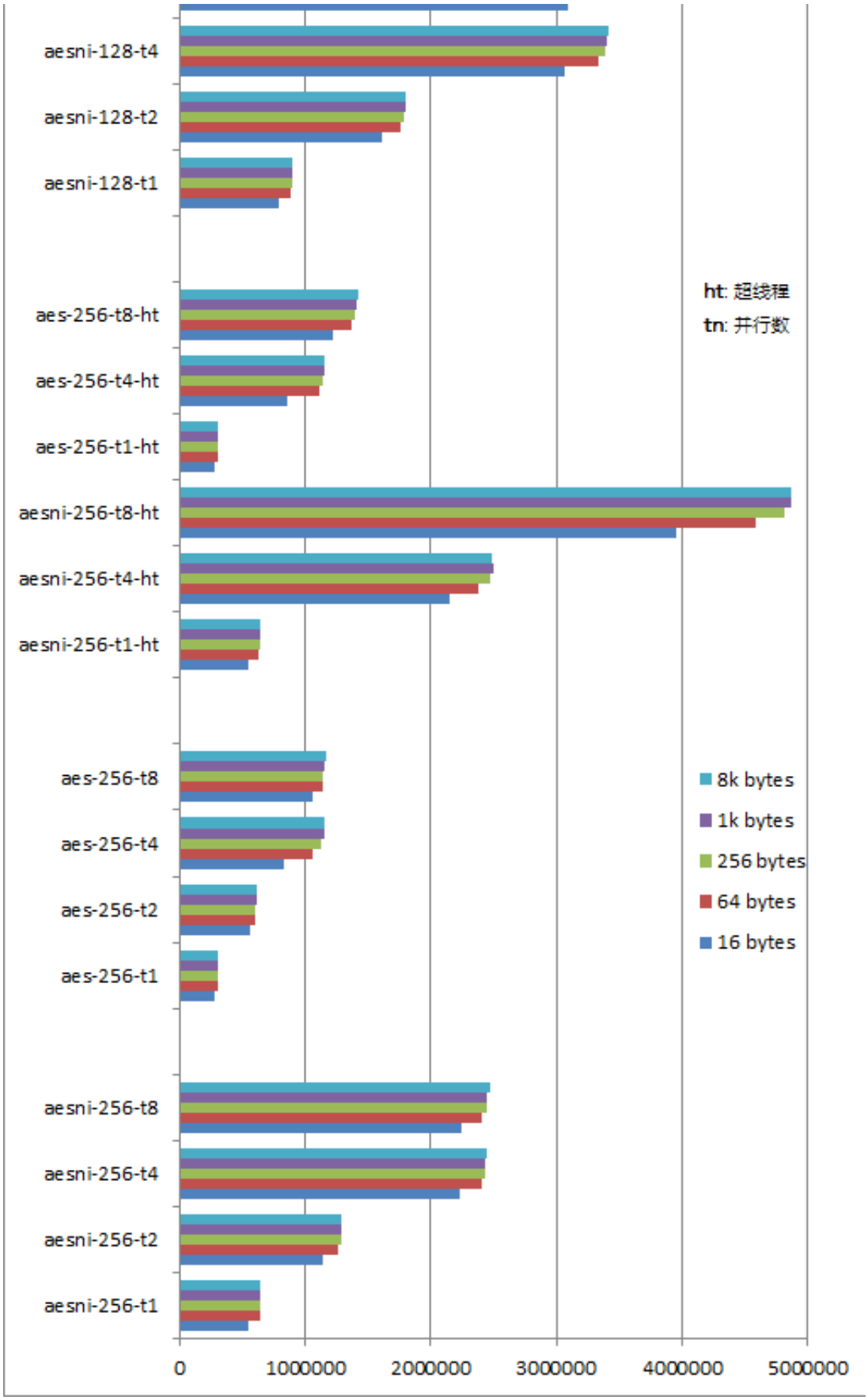
测试环境：

```
Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz 4 Cores with HyperThread (Enabled or Disab
led)
Ubuntu 16.04 AMD64, OpenSSL 1.0.2g-fips  1 Mar 2016
```

测试方法：

```
关闭硬件加速1/2/4/8线程AES-256/128-CBC：
OPENSSL_ia32cap="~0x200000200000000" openssl speed -multi {1/2/4/8} -elapsed -evp
{aes-256/128-cbc}

开启硬件加速1/2/4/8线程AES-256/128-CBC：
openssl speed -multi {1/2/4/8} -elapsed -evp {aes-256/128-cbc}

超线程的开户与关闭只能通过UEFI/BIOS来设置，测试命令同上。
```





从图中可以得到如下结论：

1. AES_NI加速可以提升性能1倍多，AESNI-128基本上都是AES-128的2.2倍左右。
2. AES-128与AES-256的性能比基本在1.36左右（15/11，忽略密钥编排用时的情况下）

3. 比较有趣的一点发现是，超线程所带来的影响比预想的要大得多。针对高并行的情形，在开启AES*NI时超线程可以带来接近1倍的性能提升；但在关闭AES*NI的情况下对性能提升的贡献要小的多。超线程虽然逻辑上让我们觉得一核变成了两核，其实质只是同一物理核上的队列管理机制，关闭AES*NI的情况下的测试数据基本验证了这一点。另一方面AES*NI硬件加速是基于物理核的，不可能是针对超线程的，所以超线程与AES*NI组合所带来的巨大的性能提升让人有些费解，比较可能的解释是AES*NI硬件加速引擎的潜力足够强大以至于一个物理核心不能完全发挥其效能，所以在超线程开启的情况下能有更好的表现。

ARM及其它体系

2011年发布的ARMv8-A处理器架构开始支持AES加速指令，其指令集与AES_NI不兼容但实现了类似的功能。除ARM外，SUN SPARC(T4, T5, M5以后)及IBM Power7+架构的CPU均已支持AES加速。

实现上的安全性考虑

内存与交换

程序如果将密钥存储在可交换内存页中，在内存吃紧的情况下有可能会交换出来并写入磁盘。如辅以代码逆向等，密钥很有可能会泄露。

应用层最好用mlock(Linux)或VirtualLock(Windows)来防止内存页被交换至磁盘。

但因为密钥在内存中，所以任何能访问内存的方式均有可能导致密钥的泄漏。曾流行的一种攻击是通过1394 DMA方式来访问目标机内存，Linux/Windows Login bypass，Windows bitlock等漏洞均由起引起。较新的CPU为硬件虚拟化所引入的IO MMU （Intel VT-d or AMD-Vi）可以有效地限制硬件对内存的访问权限。

传统攻击

AES从产生至今依然是最安全的加密算法，传统攻击手段依然无法撼动其安全性。虽然已有攻击手段显示可以将AES-256的暴力搜索次数从2^256次降至2^119次，但依然没有实际操作价值。

不过随着计算力的提升，特别是量子计算机的发展，AES将不再是安全的。当然可以肯定的是：一定会出现更安全的加密算法。

旁路攻击

旁路攻击（Side-channel attack, SCA）是指绕过对加密算法的正面对抗及分析，利用硬件实现加密算法的逻辑电路在运算中所泄露的信息，如执行时间、功耗、电磁辐射等，并结合统计理论来实现对密码系统攻击的手段。

旁路攻击条件

旁路攻击成功的必要条件：

1. 在泄漏的物理信号与处理的数据之间建立关联
2. 在信息泄漏模型中处理的数据与芯片中处理的数据之间建立关联

智能卡CPU的实现逻辑相对比较简单，并且都是单线程处理机制，因此可以很好的建立起密码-时序或密码-功耗之间的关联。

时序攻击

不同的数值及不同的运算所需时间是不同的，在算法(运算逻辑)固定的前提下完全可以根据运行时间反推出具体的操作数。举个简单的例子：

```
if (strelen(passwd) != sizeof(fixed_passwd))
  return 0;

for (i = 0; i < sizeof(fixed_passwd); i++)
  if (passwd[i] != fixed_passwd[i])
    return 0;
```

这段代码在密码的判断上就存在时序攻击的漏洞，如果第一个字符不匹配则直接退出，只有在当前字符匹配的情况下才会继续下一个字符的比较。

所以如果实际密码长度为8位且只能用字母及数字，则理论上暴力搜索次数为 (26 * 2 + 10) ^ 8。但因为算法的实现没有考虑到时序攻击，如果将执行时间加入考量，则搜索次数将降低至(26 * 2 + 10) * 8。

本文示例代码中aes_mul()的实现也有时序攻击的漏洞，并且实现效率也比较低，当然主要目的是为了算法演示。

功耗攻击

当信号发生0-1跳变时，需要电源对电容进行充电；而在其它三种情况(0-0, 1-1, 1-0)下则不会进行充电操作，因此可以很容易区分出前者来，这就是功耗攻击原理的简单解释。

功耗攻击一般分为简单功耗攻击(Simple Power Analysis，SPA)，差分功耗攻击(Differential Power Analysis, DPA)，高阶DPA等。SPA可以揭示出执行操作和能耗泄露间的关系，而DPA则能够揭示出处理数据和能耗泄露间的关系。

DPA利用不同数据对应的条件功耗分布的差异进行统计分析以找出数值与功耗的微弱关联性，并利用此关联性极大的降低密钥的搜索空间，进而完成高效且低成本的攻击。

上海交大的教授郁昱就通过功耗攻击成功破解了来自多家手机制造商以及服务供应商的SIM卡的密钥。更详细信息可见于他在Blackhat 2015年的演示稿: Cloning 3G/4G SIM Cards with a PC and an Oscilloscope:

Lessons Learned in Physical Security。

以色列特拉维夫大学的研究人员利用旁路攻击，成功从Android和iOS设备上窃取到用于加密比特币钱包、Apple Pay账号和其他高价值资产的密钥，详细请参阅论文: ECDSA Key Extraction from Mobile Devices via Nonintrusive Physical Side Channels。

参考资料

1. 密码学原理与实践(第二版)，Douglas R. Stinson，**冯登国**译
2. AES Proposal: Rijndael by Joan Daemen and Vincent Rijmen
3. FIPS 197: **Announcing the AES**
4. Advanced Encryption Standard - **Wikipedia**
5. The Design of Rijndael by Joan Daemen & Vincent Rijmen
6. The Block Cipher Companion, **L. Knudsen & M. Robshaw**, 2011
7. 加密芯片的旁道攻击防御对策研究(博士学位论文), **李海军**, 2008
8. 旁路之能量分析攻击总结
9. AES算法介绍: **万天添**，2015/3/23
10. AES_NI - **Wikipedia**
11. AES_NI v3.01 - **Intel**

相关代码

1. https://github.com/matt-wu/AES/

<最早的手工计算AES-128的想法来源于2016年底读过的一本书《How Software Works: The Magic Behind Encryption ...》，在阅读过程中发现AES一节中的数据全对不上，然后于17年初开始翻阅AES及Rijndael算法标准等资料，等看完所有文档后才发现此书对AES的介绍真是简化得没边了，后来又做了大量的延伸阅读，春节期间根据FIPS 197及《The Design of Rijndael》实现了AES 128/192/256 ECB/CBC的计算过程，之后开始本blog的书写，中间断断续续直至今日才完工，本文估计用时约40小时。学习从来不是容易的事！但越是不容易的事情做起来才更有乐趣！>