现代密码学作业 AES 实现文档

> 计26 郑睿阳

前言

- 本文档是关于清华大学计算机系 2024-春 现代密码学课程实现 AES 算法的作业。其不仅承担了对于程序的说明，也承担了对于 AES 介绍的作用，可供复习，学习时参考使用。

AES 简介

- AES，全称 **A**dvanced **E**ncryption **S**tandard, 是美国联邦政府采取的一种服从分块密码 (Block cipher) 模式的加密算法。该算法是为了解决曾经被使用的 DES 加密算法的缺陷性所涉及的 (注：对分块密码模式和 DES 加密算法不熟悉的读者请自行查阅相关资料)。

AES 数学基础

- 在介绍 AES 之前, 首先介绍其需使用的基本数学工具。这里所提到的数学工具将在 AES "列混合" (Mix-Column) 一步用到。

Euclid 算法

- 即辗转相除法。

拓展的 Euclid 算法

找到 $x, y, st.\ ax + by = gcd(a, b)$ 关键是要在辗转相除法的每一步记录 $ax_i + by_i = r_i$
举例而言: $a = 42823, b = 6409$

```
42823 = 6 · 6409+4369
6409 = 1 · 4369+2040
4369 = 2 · 2040+289
2040 = 7 · 289+17
289 = 17 · 17
```

其逆过程为

```
17 = 2040 - 7 · 289
17 = 2040 - 7 · (4369 - 2 · 2040)
 = -7 · 4369 + 15 · 2040
17 = -7 · 4369 + 15 · (6409 - 4369)
 = 15 · 6409 - 22 · 4369
17 = 15 ·6409 - 22 ·(42823-6·6409)
  = -22 · 42823 + 147 · 6409
```

一个形象的比喻是, 这是一个逐层向上追溯的过程, 对于每个被减数和减数的 x_i, y_i 的部分, 找到 x_i, y_i 分别作为余数的算式并将其代换为更大的 x_i, y_i。

基本的抽象代数知识

群

Def：设 G 是一个非空集合，在 G 中定义了一个二元运算$\circ$，若$\circ$满足下面条件，则 G 称为一个群。

$1.\forall a,b\in G, a\circ b=G.$ (运算的封闭性) $2.\forall a,b,c, (a\circ b)\circ c=a\circ(b\circ c).$ (结合律) $3.\exists e\in G, st.\forall g\in G, e\circ g=g\circ e=g$ (e为幺元) $4.\forall g\in G, \exists g^{'}, st. g\circ g^{'}=g^{'}\circ g=e$ (逆元)

- 性质可以总结为 “封结幺逆” (谐音：一群凤姐咬你)
- 例如整数集合 $\mathbb{Z}$ 对于一般的加法，以及模 n 剩余类 $\mathbb{Z}/n$ 对于模 n 的加法。
- 如果有： $\forall a,b\in G, a\circ b=b\circ a$, 则 G 为交换群(Abel群)。

其中，如果 G 上定义的运算$\circ$只满足封闭性和结合律，则称$(G,\circ)$ 构成半群。

环

Def：集合 R 非空，在上面定义加法 $+$ 以及乘法 $*$ (注意这里的加法乘法运算是自定义的，不必须是一般意义下实数的加法乘法运算)，使得：

- $(R,+)$ 为 Abel 群
- $(R,*)$ 为半群
- 加法对乘法的**左右**分配律成立，即 $\forall a,b,c\in R, a*(b+c)=a*b+a*c; (b+c)*a=b*a+c*a$, 则称 $(R,+,*)$ 为一个环。
  - 比较常见的环便是 $(\mathbb{Z},+,*)$ (这里的加乘就是一般的运算), 以及 $(\mathbb{Z}/n,+,*)$ (这里的加乘是 $mod(n)$ 意义下的)。
  - 一些特殊的环: (方便的记忆方法：加法已经构成了交换群——性质最多的群，而乘法还是半群，可以尝试一步一步将乘法运算的性质扩展出去)

- 交换环: 若 $\forall a,b \in R, a*b=b*a$ 则 R 构成交换环。
- 单位元: 若 $\exists e \in R, s.t. \forall a \in R, e*a=a*e=a$。
- 逆元: 若对于 $a \in R, \exists b \in R, a*b=b*a=e$, 则 b为a 的逆元。

域

Def： 若环 R **至少含两个元素**, 且

- R 为交换环
- R 有 **一个** 单位元
- R 的非零元素都有逆元 则 R 为一个域，一般记作 F。
- 注：实际上，可以理解为域是这三类代数结构中需要满足的条件最多的一种——对于加法构成 Abel 群，对于非零元在乘法上也构成 Abel 群。

有限域

- 顾名思义，是指只含有限个元素的域。
- 定义 $GF(p^n)$ 为系数 $\in \mathbb{Z}/p$ 的次数为 $n-1$ 的多项式集合
- 我们考虑一类特殊的 $GF(p^n)$: $GF(2^n)$, 其中的多项式一般可以表达为:

  $b_n x^n + b_{n-1}x^{n-1} + ... + b_1 x + b_0, b_i \in \{0,1\}, i \in \{1,2,...,n\}$

  也可以将其简写为二进制形式 $b_n b_{n-1} ... b_1 b_0$。对于 $n=8$ 的形式(AES中的形式)可以使用十六进制简化。
- 那么, $GF(2^n)$ 中两个元素 a,b 的加利用二进制形式可以表达为 $a \oplus b$。
- 对于乘法, 我们应该事先约定要对一个 n 次多项式取模。在 AES 当中，我们选择的是 $m(x)=x^8+x^4+x^3+x+1$。

出于 AES 的功能考虑，我们考虑一类特殊的乘法: $x*b(x)\ mod(m(x))$, $b(x)$ 二进制表示为 8 位, 简写为:

$$xtime(b) = \text{‘02’} * b = \begin{cases} b(x) << 1, b_7 = 0 \\ (b(x) << 1) \oplus (\text{‘1B’}), b_7 = 1 \end{cases}$$

结合上面模的定义, 这样的操作是显然的。这类乘法的操作特点是简单，快速，可以使用基本的位运算很快完成。因此，与其它元素的乘法多由该乘法算式拓展而来。

使用扩展 Euclid 算法寻找 $GF(2^8)$ 当中的逆元

- 一般的拓展 Euclid 算法是给定 a, b, 寻找 x, y, 使得 $ax + by = gcd(a, b)$。若 $gcd(a, b) = 1$, 则 $ax \equiv 1 \ (mod \ b)$。

系数在 $GF(2^8)$ 当中的多项式：

- 我们考虑 $[a_0, a_1, a_2, a_3]$ 是四个字节，即 a_i 代表的是 8 bit 的信息量，也就是一个上面提到的 $GF(2^8)$ 当中的多项式。
- 考虑多项式 $a(x) = a_3x^3 + a_2x^2 + a_1x + a_0$, 我们使用模多项式 $x^4 + 1$。这也就意味着，每一个作为系数的多项式的
- 对应的多项式运算: $a(x) = a_3x^3 + a_2x^2 + a_1x + a_0$; $b(x) = b_3x^3 + b_2x^2 + b_1x + b_0$; 我们定义加法运算为: $a(x) + b(x) = (a_3 \oplus b_3)x^3 + (a_2 \oplus b_2)x^2 + (a_1 \oplus b_1)x + a_0 \oplus b_0$; 这样的定义是十分朴实无华的，可以理解为对应系数的 "加" (在这里也就是异或)。
- 再定义乘法运算为： $a(x) * b(x) = \sum_{i=0}^{6} c_ix^i$, 其中 c_i 的定义为: $c_0 = a_0 * b_0$;
  $c_1 = a_0 * b_1 \oplus a_1 * b_0$
  $c_2 = a_0 * b_2 \oplus a_1 * b_1 \oplus a_2 * b_0$
  ..... 这里，
- $*$ 运算符合上面我们在 $GF(2^8)$ 当中定义的乘法运算，也就是 $$mod x^8 + x^4 + x^3 + x + 1$ 的多项式乘法运算。
- 需要注意的是, 这种直接的运算方式无法得到一个我们所期待的系数在 $GF(2^8)$ 当中的多项式，因为它的次数是六次，而我们只需要一个三次的多项式。
- 这里的解决方案依然是取模。我们以 $x^4 + 1$ 为模进行运算。根据模运算的特征，我们可以对于具有不同次数的向分别取模再求和。这也就是说，假设通过取模得到的多项式为 $d(x) = d_3x^3 + d_2x^2 + d_1x + d_0$, 那么就有

  $d(x) = \sum_{i=0}^{6}[c_ix^i(mod x^4 + 1)]$

  我们考察 $c_ix^i mod(x^4 + 1)$ 的结果: 因为

  $c_ix^i = c_ix^{i-4}(x^4 + 1) - c_ix^{i-4}$

  所以可以得到

  $c_ix^i \equiv c_ix^{i-4}$

  那么，依次类推，我们可以得到结论

  $c_ix^i mod(x^4 + 1) = c_ix^{imod(x^4+1)}$

那么，$d(x)$ 在 $mod(x^4+1)$ 的意义下可以写成:

$$d(x) = c_3x^3 + (c_6 \oplus c_2)x^2 + (c_5 \oplus c_1)x + c_4 \oplus c_0$$

所以对应的系数应该满足: $d_0 = (a_0 * b_0) \oplus (a_3 * b_1) \oplus (a_2 * b_2) \oplus (a_1 * b_3)$
$d_1 = (a_0 * b_1) \oplus (a_1 * b_0) \oplus (a_3 * b_2) \oplus (a_2 * b_3)$
$d_2 = (a_0 * b_2) \oplus (a_1 * b_1) \oplus (a_2 * b_0) \oplus (a_3 * b_3)$
$d_3 = (a_0 * b_3) \oplus (a_1 * b_2) \oplus (a_2 * b_1) \oplus (a_3 * b_0)$ 如果我们将其视为一个作用在多项式 $b(x)$ 上的变换，那么不难将其写成一个矩阵的形式:

$$\begin{pmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{pmatrix} = \begin{pmatrix} a_0 & a_3 & a_2 & a_1 \\ a_1 & a_0 & a_3 & a_2 \\ a_2 & a_1 & a_0 & a_3 \\ a_3 & a_2 & a_1 & a_0 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

在下面，我们会看到，可以通过这个矩阵变换将 b 向量(明文)变换为 d 向量(密文)。那么，这也就要求这个矩阵必须可逆。但实际上，x^4+1 在 $GF(2^8)$ 上不可约，因此并不是每个多项式 $a(x)$ 形成的矩阵都是可逆的。

- 我们可以认为选取使得矩阵可逆的多项式: $a(x) = 03x^3 + 01x^2 + 01x^2 + 02$
  $a^{-1}(x) = 0bx^3 + 0dx^2 + 09x + 0e$

AES 算法主体

- 注：以下所述的算法实现均依赖于本人实现的代码
- 首先我们先从密码学的角度给出 AES 具体实现的功能。在本次作业当中要求实现 ECB 以及 CTR 模式的 AES 加密。我们首先先从 ECB 模式讲起。

AES-ECB 加密算法

- 在本人的文档当中为 `encrypt()` 函数。
- 作为对称加密算法，AES-ECB 接受长度为 16 bytes 的**倍数**的输入明文，通过一个 16 bytes 的密钥加密，最后输出与输入等长度的密文。ECB 模式下每次会对 16 bytes 的明文进行加密，因此从此也可以看出其满足分块密码的加密模式。每次进行加密的 16 位信息为了方便，将会被以一个名为 `state` 的 $4 * 4$ 的数组存储起来。
- 不过，也需要指出的是，ECB 模式的安全性也因此比较低，因为 16 位的密文只和它对应的 16 位的信息是相关的。
- AES 算法的加密过程可以分为以下几部分:

    - 初始的1轮加密
    - 9轮正常加密
    - 1轮额外加密
- 但是，无论是哪一部分，最终的加密结果都源于以下几个函数:
    - 字节替换 `subBytes()` 函数
    - 行移位 `shiftRows()` 函数
    - 列混合 `mixColumns()` 函数
    - 轮密钥相加 `addRoundKey(const unsigned char rndKey[4][4])` 函数 在这里对这些函数进行逐一介绍:

字节替换

- 输入长度为 16 bytes 的信息，输出的是同等长度的经过字节替换的信息。
- 在 AES 算法当中，我们实现约定好了一个进行替换使用的二维数组 `EnCryp_sbox`, 用法如下： `EnCryp_sbox` 的规模为 $16 * 16$, 行，列序号分别对应 $0 - f$, 指的是某一个需要进行加密的字节的**十六进制表示**的第一位，第二位，在对应的数组当中存储的内容则是该字节替换后的结果。AES 所采用的 `EnCryp_sbox` 如下:

```
const unsigned char EnCryp_sbox[256] = {
      // 0     1     2     3     4     5     6     7     8     9     a    b    c
d    e    f
      0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b,
0xfe, 0xd7, 0xab, 0x76, // 0
      0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf,
0x9c, 0xa4, 0x72, 0xc0, // 1
      0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1,
0x71, 0xd8, 0x31, 0x15, // 2
      0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2,
0xeb, 0x27, 0xb2, 0x75, // 3
      0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3,
0x29, 0xe3, 0x2f, 0x84, // 4
      0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39,
0x4a, 0x4c, 0x58, 0xcf, // 5
      0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f,
0x50, 0x3c, 0x9f, 0xa8, // 6
      0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21,
0x10, 0xff, 0xf3, 0xd2, // 7
      0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d,
0x64, 0x5d, 0x19, 0x73, // 8
      0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14,
0xde, 0x5e, 0x0b, 0xdb, // 9
      0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62,
0x91, 0x95, 0xe4, 0x79, // a
      0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea,
0x65, 0x7a, 0xae, 0x08, // b
      0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f,
0x4b, 0xbd, 0x8b, 0x8a, // c
      0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9,
0x86, 0xc1, 0x1d, 0x9e, // d
      0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9,
0xce, 0x55, 0x28, 0xdf, // e
      0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f,
0xb0, 0x54, 0xbb, 0x16  // f
  };
```

这里需要注意的是，为了优化我们的算法，我们在查询索引的时候并未使用二维数组，而是使用了一维数组，这样就可以直接利用明文的十六进制表示作为索引进行查询。例如，如果我们要对字节 `0x13` 进行加密，就找到横坐标为 1, 纵坐标为 3 的地方，得到加密结果为 `0xc7` 。

- 在我们的代码实现当中，如前所述，我们直接使用明文索引进行查询以提升效率:

```
void subBytes() {
 for (int i = 0; i < 4; i++) {
      for (int j = 0; j < 4; j++) {
          state[i][j] = EnCryp_sbox[(int)state[i][j]];
      }
  }
}
```

行移位

- 输入长度为 16 bytes 的信息，输出的是同等长度的经过行移位的信息。
- 行移位实际上便是采取了循环移位的方法将 16 bytes 的信息状态打乱。`state` 数组的第 0，1，2，3 行分别左移 0，1，2，3位即可得到相应的信息。
- 尽管在 C++ 当中有自带的循环移位 `rotate` 函数，在这里为了追求效率我们还是采用自己的写法:

```
void shiftRows() {
      // The four rows shift to the left respectively 0, 1, 2, 3 bytes
      for (int i = 0; i < 4; i++) {
          char tmp = state[i][0];
          char tmp1 = state[i][0];
          char tmp2 = state[i][1];
          char tmp3 = state[i][3];
          switch (i) {
          case 0:
              break;
          case 1:
              for (int j = 0; j < 3; j++) {
                  state[i][j] = state[i][j + 1];
              }
              state[i][3] = tmp;
              break;
          case 2:
              state[i][0] = state[i][2];
              state[i][1] = state[i][3];
              state[i][2] = tmp1;
              state[i][3] = tmp2;
              break;
          case 3:
              for (int j = 2; j >= 0; j--) {
                  state[i][j + 1] = state[i][j];
              }
              state[i][0] = tmp3;
              break;
          }
      }
  }
```

列混合

- 输入长度为 16 bytes 的信息，输出的是同等长度的经过列混合的信息。
- 列混合是四个函数当中最复杂的一个函数。要了解这个函数的实现，请确保已经理解了上面所介绍的数学基础。
- 在列混合的实现当中，我们实现了一类特殊的 "矩阵乘法", 将 `state` 通过一个映射映射到加密后的 `state` 矩阵。具体的，需要使 `state` 矩阵和下面的矩阵进行上面系数在 $GF(2^8)$ 当中的多项式中定义的矩阵乘法。具体的，变换矩阵为

$$\begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix}$$

字节与 02 相乘满足上面 $xtime$ 函数的定义:

$$xtime(b) = `02' * b = \begin{cases} b(x) << 1, \\ b_7 = 0 \\ (b(x) << 1) \oplus (`1B'), b_7 = 1 \end{cases}$$

字节与 03 相乘则是可以认为是 $03 * b = 02 * b \oplus b = xtime(b) \oplus b$
$04 * b = 02 * (02 * b) = xtime(xtime(b))$ 这也是我们为加密所指定的几个函数 `mul2`, `mul3`, `mul4` 的实现方法:

```
  unsigned char mul2(unsigned char value) {
      return (value & 0x80) ? (unsigned char)((value << 1) ^ 0x1b) : (unsigned char)
(value << 1);
  }
  unsigned char mul3(unsigned char value) {
      return mul2(value) ^ value;
  }
  unsigned char mul4(unsigned char value) {
      return mul2(mul2(value));
  }
```

而相应的，前面提到在解密的时候，我们需要作用它的逆矩阵: 在这里也就是

$$\begin{pmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{pmatrix}$$

我们需要实现 `mul8`, `mul9`, `mulb`, `muld`, `mule` 的函数。 其中:

- $08 * b = xtime(xtime(xtime(b)))$
- $09 * b = 08 * b + b$
- $0B * b = 09 * b + xtime(b)$
- $0D * b = 0B * b + xtime(b)$
- $0E * b = 0D * b + b$ 代码实现如下:

```
unsigned char mul8(unsigned char val) { return mul2(mul2(mul2(val))); }
unsigned char mule(unsigned char val) { return mul8(val) ^ mul4(val) ^ mul2(val);
}
unsigned char mulb(unsigned char val) { return mul8(val) ^ mul2(val) ^ val; }
unsigned char muld(unsigned char val) { return mul8(val) ^ mul4(val) ^ val; }
unsigned char mul9(unsigned char val) { return mul8(val) ^ val; }
```

轮密钥相加

- 如上所述，在每 16 字节的加密过程当中都需要经过 $1+9+1=11$ 轮的加密。为了提升加密的安全性，我们在每一轮加密当中都需要更换密钥。但实际上，我们在执行 AES 算法的时候，只需要传入初始的 16 位的密钥作为参数即可。此时，我们需要将密钥进行 “扩展”, 另外再扩展出 10 个密钥才能满足加密的需求。

- 由于每一轮使用的密钥都是 16 bytes, 因此我们可以将密钥也视作是一个 $4*4$ 的数组。11 轮密钥则可以视作一个 $44*4$ 规模的数组。将每一列视作是一个单元 (一共是 4 行 44 列), 第 $0-3$ 列用初始的密钥进行填充，则后面的 40 列的填充规则是: 对于第 i 列 $col(i)$, 满足

$$col(i)=\begin{cases}T(col(i-1))\oplus col(i-4), 4\mid i,\\ col(i-1)\oplus col(i-4), 4\nmid i\end{cases}$$

其中，T 函数是对于长度为 4 bytes 的列的变换。具体可以分为以下三部分，按顺序进行:

- 循环移位: 将原本顺序为 $a_0a_1a_2a_3$ 的一列置换为 $a_1a_2a_3a_0$。
- 字节代换: 利用 `subBytes` 函数的原理，将列中的每一个字节替换为相应的代换值。
- 轮常量异或: 列中的第一个元素 $col(i)[0]$ 与事先约定好的 “轮常量” 进行异或。其中，轮常量是依赖于我们当前加密的轮的序号的，该序号从 1 到 10。在我们的具体实现当中，约定轮常量为:

```
const unsigned char rcon[11] = { 0x00, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40,
0x80, 0x1B, 0x36 };
```

上面的 T 函数在代码当中体现为 `key_process(unsigned char* key, int round)` 函数:

```
void key_process(unsigned char* key, int round) {
   // 1. RotWord
   char tmp = key[0];
   for (int i = 0; i < 3; i++) {
       key[i] = key[i + 1];
   }
   key[3] = tmp;
   // 2. Subword
   for (int i = 0; i < 4; i++) {
       key[i] = EnCryp_sbox[(int)key[i]];
   }
   // 3. Round const Xor
   key[0] = key[0] ^ rcon[round];
}
```

整体的轮密钥相加过程如下:

```
void keyExpansion(const unsigned char* key) {
   // Fill in the array roundKeys with the expanded key
   // The round key for the initial round is key itself
   for (int i = 0; i < 4; i++) {
       for (int j = 0; j < 4; j++) {
           roundKeys[0][j][i] = key[(i << 2) + j];
       }
   }
   // 10 more rounds
   for (int i = 1; i < 11; i++) {
       unsigned char newkey[4][4];
       for (int j = 0; j < 4; j++) {
           if (j == 0) {
               // The first column needs to be dealt in a special way
               unsigned char tmp[4];
               for (int k = 0; k < 4; k++) {
                   tmp[k] = roundKeys[i - 1][k][3];
               }

               // process tmp
               key_process(tmp, i);

               for (int k = 0; k < 4; k++) {
#ifdef MYDEBUG
                   int value = static_cast<int>(roundKeys[i - 1][k][j]);
                   std::cout << std::hex << std::setw(2) << std::setfill('0') <<
value << std::endl;
#endif
                   newkey[k][j] = tmp[k] ^ roundKeys[i - 1][k][j];
               }
           }
           else {
               for (int k = 0; k < 4; k++) {
                   newkey[k][j] = newkey[k][j - 1] ^ roundKeys[i - 1][k][j];
               }
           }
       }
       // Copy it to roundKeys[i]
       for (int m = 0; m < 4; m++) {
           for (int n = 0; n < 4; n++) {
               roundKeys[i][m][n] = newkey[m][n];
           }
       }
   }
}
```

主体步骤

- 以上，我们介绍了四个在 AES-ECB 过程当中所需要用到的函数。下面，我们对于 AES-ECB 模式当中每个部分所需要用到的函数及其顺序进行介绍。
- 初始的 1 轮加密: 需要使用一轮 `addRoundKey(roundKeys[0])` 函数，使用初始密钥作为参数。
- 9 轮正常加密: 每轮加密会按照顺序经过上面的四个函数:

```
for (int i = 1; i < 10; i++) {
  subBytes();
 shiftRows();
  mixColumns();
  addRoundKey(roundKeys[i]);
}
```

- 1轮额外加密: 不经过列混合，其余和上面 9 轮一样:

```
subBytes();
shiftRows();
addRoundKey(roundKeys[10]);
```

AES-CTR 算法

- 为了解决上面所说的不安全的问题，AES-CTR 模式则是引入了一个规模为 16 bytes 的计数器 `counter`。在代码实现当中，其具体的填充内容来自数组 `iv`(可以理解为 `counter` 和 `iv` 在初始条件下是相同的)。每次进行 16 bytes 的加密时，将此时的 `counter` 作为明文，传入的 `key` 作为密钥，按照 ECB 模式进行加密得到 “异或密钥”。这一轮的加密结果便是明文与 “异或密钥” 进行异或之后的结果。在此之后, 将 `counter` 当中存储的信息视作是一个数，则将其加 1，并进入下一轮，重复刚才的过程。
- 值得一提，在 AES-CTR 当中不要求输入的明文是 16 bytes 的倍数，因为执行异或时可以将 “异或密钥” 截断，使之长度和明文相当。
- 以下是代码实现:

```
void aes_ctr(const unsigned char* plaintext, int len, const unsigned char* iv,
unsigned char* ciphertext, const unsigned char* key = nullptr) {
      // Notice that this function is used both for encryption and decryption.
While used for decryption, simply swap plaintext and ciphertext.

      // Under such circumstance, the key is already set
      unsigned char counter[16]; // A 16 byte array used to store the key
      memcpy(counter, iv, 16);

      // len stands for the length of pt in bytes
      for (int i = 0; i < len; i += 16) {
          unsigned char my_key[16];
          // In the first round, just use the initial key
          if (i != 0) {
              // Else, we increase the key by 1
              for (int j = 15; j >= 0; j--) {
                  if ((counter[j] & (unsigned char)(0xFF)) == (unsigned char)
(0xFF)) {
                      counter[j] = 0; // Carry to the previous number
                  }
                  else {
                      counter[j]++; // Just add 1 and stop
                      break;
                  }
              }
          }
          encrypt(counter, my_key);
          for (int j = i; j < i + 16; j++) {
              if (j >= len) {
                  // exceed length
                  break;
              }
              ciphertext[j] = my_key[j - i] ^ plaintext[j];
          }
      }

  }
```

效率

- 为了提升加密速度，在生成可执行文件时，对于 `AES.cpp` 进行了 `O2` 优化:

```
g++ -O2 AES.cpp -o AES.exe
```

AES-ECB

- 在执行 AES-ECB 模式的时候，无论使用 8k bits 还是 8M bits 的输入，加密速度均较快，无法测量出准确的结果。

```
(base) PS D:\Cryptography\Cryptography\AES> .\AES.exe aes-ecb 8kbit_data.txt output.txt
workmode: aes-ecb inputfile: 8kbit_data.txt outputfile: output.txt
Key is:         2b 7e 15 16 28 ae d2 a6 ab f7 15 88 9 cf 4f 3c
AES-128 CTR Use: 0ms
The Speed is: nanMbps
(base) PS D:\Cryptography\Cryptography\AES> .\AES.exe aes-ecb 8mbit_data.txt output.txt
workmode: aes-ecb inputfile: 8mbit_data.txt outputfile: output.txt
Key is:         2b 7e 15 16 28 ae d2 a6 ab f7 15 88 9 cf 4f 3c
AES-128 CTR Use: 0ms
The Speed is: infMbps
```

AES-CTR

- 在执行 AES-CTR 模式的时候，使用 8k bits 的时候加密速度较快，无法准确的测量出具体加密速度；

```
(base) PS D:\Cryptography\Cryptography\AES> .\AES.exe aes-ctr 8kbit_data.txt output.txt
workmode: aes-ctr inputfile: 8kbit_data.txt outputfile: output.txt
Key is:         2b 7e 15 16 28 ae d2 a6 ab f7 15 88 9 cf 4f 3c
AES-128 CTR Use: 0ms
The Speed is: nanMbps
```

- 在使用 8M bits 进行加密时，经过多次对于效率的测量:

```
(base) PS D:\Cryptography\Cryptography\AES> .\AES.exe aes-ctr 8mbit_data.txt output.txt
workmode: aes-ctr inputfile: 8mbit_data.txt outputfile: output.txt
Key is:         2b 7e 15 16 28 ae d2 a6 ab f7 15 88 9 cf 4f 3c
AES-128 CTR Use: 23ms
The Speed is: 347.826Mbps
(base) PS D:\Cryptography\Cryptography\AES> .\AES.exe aes-ctr 8mbit_data.txt output.txt
workmode: aes-ctr inputfile: 8mbit_data.txt outputfile: output.txt
Key is:         2b 7e 15 16 28 ae d2 a6 ab f7 15 88 9 cf 4f 3c
AES-128 CTR Use: 22ms
The Speed is: 363.636Mbps
(base) PS D:\Cryptography\Cryptography\AES> .\AES.exe aes-ctr 8mbit_data.txt output.txt
workmode: aes-ctr inputfile: 8mbit_data.txt outputfile: output.txt
Key is:         2b 7e 15 16 28 ae d2 a6 ab f7 15 88 9 cf 4f 3c
AES-128 CTR Use: 24ms
The Speed is: 333.333Mbps
(base) PS D:\Cryptography\Cryptography\AES> .\AES.exe aes-ctr 8mbit_data.txt output.txt
workmode: aes-ctr inputfile: 8mbit_data.txt outputfile: output.txt
Key is:         2b 7e 15 16 28 ae d2 a6 ab f7 15 88 9 cf 4f 3c
AES-128 CTR Use: 21ms
The Speed is: 380.952Mbps
```

可以看到效率平均下来大概为 350 MBps。

并行优化

- 事实上，AES-CTR 加密算法是可以通过并发实现的。尽管原本在不使用并发的情况下在使用 `-O2` 优化编译的情况下已经可以达到效率的要求，但是我们还是实现了简单的并发执行程序。

- 首先我们来分析 AES-CTR 为什么能够并行实现。在该模式下，对密文的加密实际上是首先通过对于计数器的加密得到中间密钥，再利用中间密钥和明文进行异或得到密文。而实际上，不同的计数器之间完全不必要是串行关系，可以预先计算出每个计数器的值，然后并行处理它们的异或操作。

- 在程序当中，设计了宏变量 `CORE` 来表示 CPU 核数，`MIN_LENGTH` 表示并行处理的最小单元（如果长度小于这个最小单元则用一个线程处理即可）。然后按照上面的思路进行改进即可。

- 代码如下：

```
if (workmode == "aes-enc-ctr" || workmode == "AES-ENC-CTR"|| workmode == "aes-dec-
ctr" || workmode == "AES-DEC-CTR") {

        vector<thread> my_threads;
        // First, calculate how many blocks to divide
        int blocks;
        int block_length;
        int last_length;

        if ((count / CORE) < MIN_LENGTH) {
            // Each block is MIN_LENGTH
            blocks = (count % MIN_LENGTH == 0) ? (count / MIN_LENGTH) : ((count +
MIN_LENGTH) / MIN_LENGTH);
            // Beware that the message length could be shorter than block_length
itself
            block_length = (MIN_LENGTH < count) ? MIN_LENGTH : count;
            last_length = (count % block_length == 0) ? (block_length) : (count %
block_length);
        }
        else {
            // We want each block's length to be mtp of 16 bytes (which means
ending with 0000 byte-wise)
            //cout << "count: " << count << endl;
            int init_length = (count / CORE);
            //cout << "init_length: " << (int)init_length << endl; // in bytes
            int length = (init_length) & 0xFFFFFFF0;
            //cout << "length: " << length << endl;
            blocks = (count % length == 0) ? (count / length) : ((count + length)
/ length);
            last_length = (count % length == 0) ? (length) : (count % length);
            //cout << "blocks: " << blocks << endl;
            block_length = length;
        }

        //cout << "Blocks: " << blocks << " block length: " << block_length <<
endl;

        for (int i = 0; i < blocks; i++) {
            Param my_params;
            my_params.plain_data = &plain_data[i * block_length];
            if (i == blocks - 1) {
                my_params.count = last_length;
            }
            else {
                my_params.count = block_length;
            }
            my_params.offset = i * (block_length >> 4);
            my_params.output_data = &output_data[i * block_length];
            //cout << "Create thread " << i << endl;
            my_threads.push_back(thread(aes_ctr_multi_thread, my_params));
```

```
        }

        for (int i = 0; i < blocks; i++) {
            my_threads[i].join();
        }
    }
```

在实际测量当中，编译选项开 `-O2` 优化的情况下，有如下结果：

```
(base) PS D:\Cryptography\Cryptography\AES> AES.exe aes-enc-ctr .\8mbit_data.txt .\output.txt
workmode: aes-enc-ctr inputfile: .\8mbit_data.txt outputfile: .\output.txt
Key is:                 1 23 45 67 89 ab cd ef fe dc ba 98 76 54 32 10
Initial Vector is:      f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 fa fb fc fd fe ff
aes-enc-ctr Use: 8ms
The Speed is: 1000Mbps
(base) PS D:\Cryptography\Cryptography\AES> AES.exe aes-enc-ctr .\8mbit_data.txt .\output.txt
workmode: aes-enc-ctr inputfile: .\8mbit_data.txt outputfile: .\output.txt
Key is:                 1 23 45 67 89 ab cd ef fe dc ba 98 76 54 32 10
Initial Vector is:      f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 fa fb fc fd fe ff
aes-enc-ctr Use: 11ms
The Speed is: 727.273Mbps
(base) PS D:\Cryptography\Cryptography\AES> AES.exe aes-enc-ctr .\8mbit_data.txt .\output.txt
workmode: aes-enc-ctr inputfile: .\8mbit_data.txt outputfile: .\output.txt
Key is:                 1 23 45 67 89 ab cd ef fe dc ba 98 76 54 32 10
Initial Vector is:      f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 fa fb fc fd fe ff
aes-enc-ctr Use: 8ms
The Speed is: 1000Mbps
(base) PS D:\Cryptography\Cryptography\AES> AES.exe aes-enc-ctr .\8mbit_data.txt .\output.txt
workmode: aes-enc-ctr inputfile: .\8mbit_data.txt outputfile: .\output.txt
Key is:                 1 23 45 67 89 ab cd ef fe dc ba 98 76 54 32 10
Initial Vector is:      f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 fa fb fc fd fe ff
aes-enc-ctr Use: 16ms
The Speed is: 500Mbps
(base) PS D:\Cryptography\Cryptography\AES> AES.exe aes-enc-ctr .\8mbit_data.txt .\output.txt
workmode: aes-enc-ctr inputfile: .\8mbit_data.txt outputfile: .\output.txt
Key is:                 1 23 45 67 89 ab cd ef fe dc ba 98 76 54 32 10
Initial Vector is:      f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 fa fb fc fd fe ff
aes-enc-ctr Use: 8ms
The Speed is: 1000Mbps
```

平均时间 $800-900Mbps$。

参考资料

- 2024年春季学期清华大学《现代密码学》课程讲义
- AES 官方文档: Announcing the ADVANCED ENCRYPTION STANDARD (AES)
- B站-AES 加密过程详解 作者: 可厉害的土豆 https://www.bilibili.com/video/BV1i341187fK/?spm_id_from=333.337.search-card.all.click