AES 仿真实现

16337341 朱志儒

实验目的

1. 通过实验加深对 AES 加密算法的基本原理的理解；

2. 了解 AES 算法的详细步骤

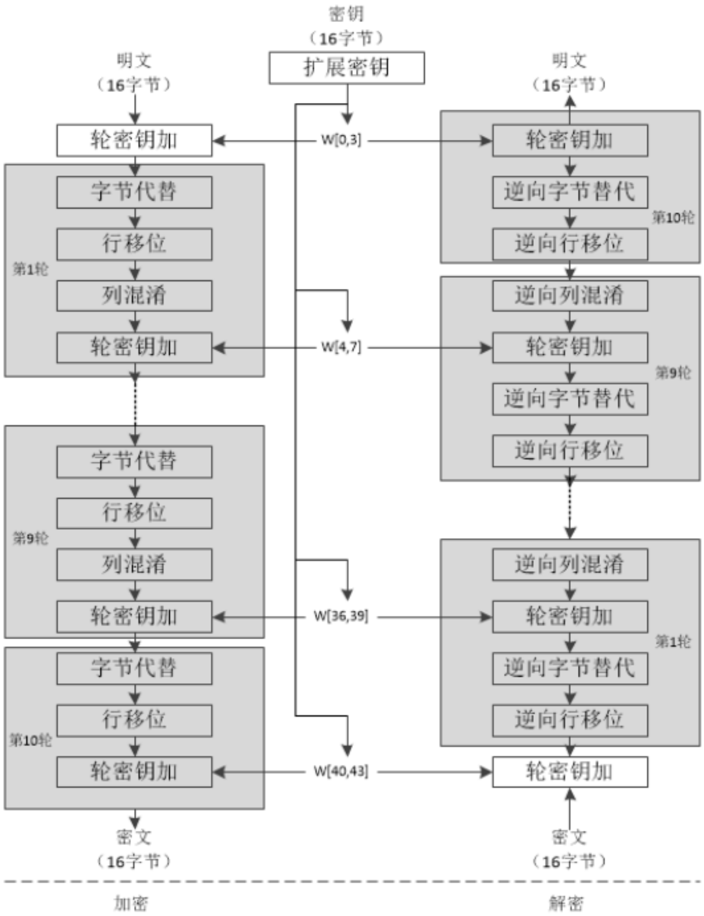
实验原理

AES 的基本结构

AES 为分组密码，分组密码将明文分成多个等长的小组，每次加密一组数据，直到加密完整个明文。在 AES 标准规范中，分组长度只能为 128 位，那么每个分组为 16 个字节。密钥的长度可以使用 128 位、192 位或 256 位，密钥的长度不同，加密轮数也不同，如下表：

AES	密钥长度（32 位比特字）	分组长度（32 位比特字）	加密轮数
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

AES 加密算法涉及：字节代替、行位移、列混淆和轮密钥加。下图为 AES 加解密的流程，从图中可以看出：（1）解密算法的每一步分别对应加密算法的逆操作；（2）加解密所有操作的顺序正好是相反的。加解密中每轮的密钥分别由种子密钥经过密钥扩展算法得到。算法中 16 字节的明文、密文和轮子密钥都以一个 4x4 的矩阵表示。



算法原理

1) 字节替代

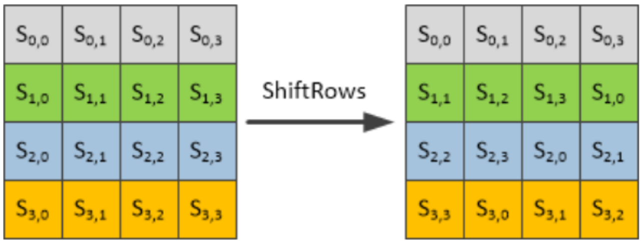
字节替代是通过 S 盒完成一个字节到另外一个字节的映射。这里直接给出构造好的结果，下图为 S 盒。S 盒用于提供密码算法的混淆性。

		y															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
x	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

S 为 16x16 的矩阵，完成一个 8 比特输入到 8 比特输出的映射，输入的高 4-bit 对应的值作为行标，低 4-bit 对应的值作为列标。假设输入字节的值为$A = a_7a_6a_5a_4a_3a_2a_1a_0$，则输出值为$S[a_7a_6a_5a_4][a_3a_2a_1a_0]$。

2) 行移位

行移位是一个 4x4 的矩阵内部字节之间的置换，用于提供算法的扩散性。正向行移位用于加密，原理：第一行保持不变，第二行循环左移 8 比特，第三行循环左移 16 比特，第四行循环左移 24 比特。假设矩阵名字为 state，用公式表示：$state'[i][j] = state[i][(j+1)\%4]$，其中 i、j 属于[0, 3]。



3) 列混淆

列混淆利用 GF(2⁸)域上算术特性的一个代替，同样用于提供算法的扩散性。 列混合变换是通过矩阵相乘来实现的，经行移位后的状态矩阵与固定的矩阵相乘，得到混淆后的状态矩阵，如下图的公式所示：

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} = \begin{bmatrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix}$$

状态矩阵中的第 j 列$(0 \le j \le 3)$的列混合可以表示为下图所示：

$$\begin{aligned} s'_{0,j} &= (2 * s_{0,j}) \oplus (3 * s_{1,j}) \oplus s_{2,j} \oplus s_{3,j} \\ s'_{1,j} &= s_{0,j} \oplus (2 * s_{1,j}) \oplus (3 * s_{2,j}) \oplus s_{3,j} \\ s'_{2,j} &= s_{0,j} \oplus s_{1,j} \oplus (2 * s_{2,j}) \oplus (3 * s_{3,j}) \\ s'_{3,j} &= (3 * s_{0,j}) \oplus s_{1,j} \oplus s_{2,j} \oplus (2 * s_{3,j}) \end{aligned}$$

逆向列混淆的原理图如下:

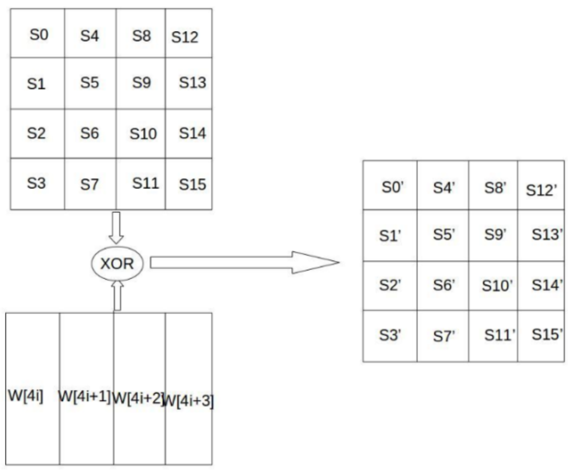
由于:

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} = \begin{bmatrix} 01 & 00 & 00 & 00 \\ 00 & 01 & 00 & 00 \\ 00 & 00 & 01 & 00 \\ 00 & 00 & 00 & 01 \end{bmatrix}$$

说明两个矩阵互逆，经过一次逆向列混淆后即可恢复原文，从而实现解密得到明文。

4) 轮密钥加

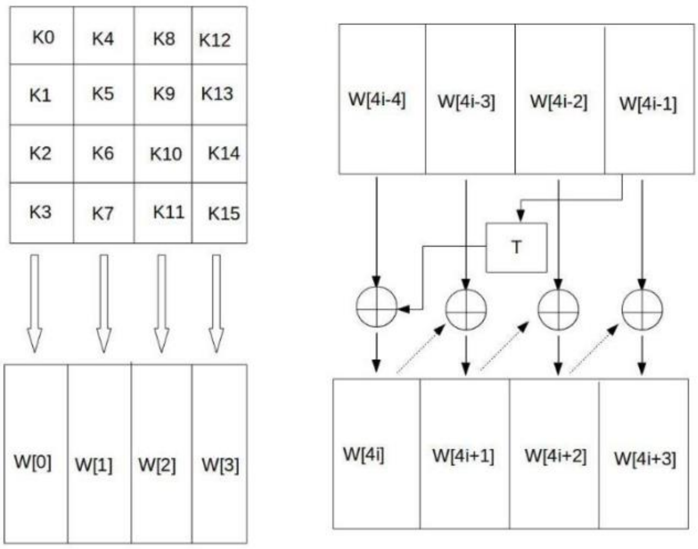
轮密钥加是将 128 位轮密钥 K_i 同状态矩阵中的数据进行逐位异或操作，如下图所示。其中，密钥 K_i 中每个字 W[4i],W[4i+1],W[4i+2],W[4i+3]为 32 位比特字，包含 4 个字节，他们的生成算法将在下面介绍。轮密钥加过程可以看成是字逐位异或的结果，也可以看成字节级别或者位级别的操作。也就是说，可以看成$S_0S_1S_2S_3$组成的 32 位字与 W[4i]的异或运算。



轮密钥加的逆运算同正向的轮密钥加运算完全一致，这是因为异或的逆操作是其自身。轮密钥加非常简单，但却能够影响 S 数组中的每一位。

5) 密钥扩展

AES 首先将初始密钥输入到一个 4*4 的状态矩阵中，如下图所示：



这个 4x4 矩阵的每一列的 4 个字节组成一个字，矩阵 4 列的 4 个字依次命名为 W[0]、W[1]、W[2]和 W[3]，它们构成一个以字为单位的数组 W。

实验内容

密钥设置为“16337341zhuzhiru”，密钥偏移量为“123”：

```
const char original_key[17] = "16337341zhuzhiru";
const char original_iv[17] = "123";
```

AES 加密：

```
string EncryptionAES(const string& strSrc) { //AES 加密
    size_t length = strSrc.length();
    int block_num = length / BLOCK_SIZE + 1;
    //明文
    char* szDataIn = new char[block_num * BLOCK_SIZE + 1];
    memset(szDataIn, 0x00, block_num * BLOCK_SIZE + 1);
```

```

#pragma warning(disable:4996)
    strcpy(szDataIn, strSrc.c_str());

    //进行 PKCS7Padding 填充。
    int k = length % BLOCK_SIZE;
    int j = length / BLOCK_SIZE;
    int padding = BLOCK_SIZE - k;
    for (int i = 0; i < padding; i++)
        szDataIn[j * BLOCK_SIZE + k + i] = padding;
    szDataIn[block_num * BLOCK_SIZE] = '\0';

    //加密后的密文
    char *szDataOut = new char[block_num * BLOCK_SIZE + 1];
    memset(szDataOut, 0, block_num * BLOCK_SIZE + 1);

    //进行进行 AES 的 CBC 模式加密
    AES aes;
    aes.MakeKey(original_key, original_iv, 16, 16);
    aes.Encrypt(szDataIn, szDataOut, block_num * BLOCK_SIZE, AES::CBC);
    string str = base64_encode((unsigned char*)szDataOut, block_num * BLOCK_SIZE);
    delete[] szDataIn;
    delete[] szDataOut;
    return str;
}
```

AES 解密:

```
string DecryptionAES(const string& strSrc) { //AES 解密
    string strData = base64_decode(strSrc);
    size_t length = strData.length();
    //密文
    char *szDataIn = new char[length + 1];
    memcpy(szDataIn, strData.c_str(), length + 1);
    //明文
    char *szDataOut = new char[length + 1];
    memcpy(szDataOut, strData.c_str(), length + 1);

    //进行 AES 的 CBC 模式解密
    AES aes;
    aes.MakeKey(original_key, original_iv, 16, 16);
    aes.Decrypt(szDataIn, szDataOut, length, AES::CBC);

```

```
    //去 PKCS7Padding 填充
    if (0x00 < szDataOut[length - 1] <= 0x16) {
        int tmp = szDataOut[length - 1];
        for (int i = length - 1; i >= (length - tmp); i--) {
            if (szDataOut[i] != tmp) {
                memset(szDataOut, 0, length);
                cout << "去填充失败！解密出错！！" << endl;
                break;
            }
            else
                szDataOut[i] = 0;
        }
    }
    string strDest(szDataOut);
    delete[] szDataIn;
    delete[] szDataOut;
    return strDest;
}
```

XOR 运算:

```
void AES::Xor(char* buff, char const* chain) {
    if (false == m_bKeyInit)
        return;
    for (int i = 0; i < m_blockSize; i++)
        *(buff++) ^= *(chain++);
}
```

扩展密钥:

```
void AES::MakeKey(char const* key, char const* chain, int keylength, int blockSize) {
    if (NULL == key)
        return;
    if (!(16 == keylength || 24 == keylength || 32 == keylength))
        return;
    if (!(16 == blockSize || 24 == blockSize || 32 == blockSize))
        return;
    m_keylength = keylength;
    m_blockSize = blockSize;
    //初始化
    memcpy(m_chain0, chain, m_blockSize);
    memcpy(m_chain, chain, m_blockSize);
```

```
    //计算轮数
    switch (m_keylength) {
        case 16:
            m_iROUNDS = (m_blockSize == 16)?10:(m_blockSize == 24?12:14);
            break;

        case 24:
            m_iROUNDS = (m_blockSize != 32) ? 12 : 14;
            break;

        default: // 32 bytes = 256 bits
            m_iROUNDS = 14;
    }
    int BC = m_blockSize / 4;
    int i, j;
    for (i = 0; i <= m_iROUNDS; i++) {
        for (j = 0; j < BC; j++)
            m_Ke[i][j] = 0;
    }
    for (i = 0; i <= m_iROUNDS; i++) {
        for (j = 0; j < BC; j++)
            m_Kd[i][j] = 0;
    }
    int ROUND_KEY_COUNT = (m_iROUNDS + 1) * BC;
    int KC = m_keylength / 4;
    //将 bytes 转成 int
    int* pi = tk;
    char const* pc = key;
    for (i = 0; i < KC; i++) {
        *pi = (unsigned char) *(pc++) << 24;
        *pi |= (unsigned char) *(pc++) << 16;
        *pi |= (unsigned char) *(pc++) << 8;
        *(pi++) |= (unsigned char) *(pc++);
    }
    //轮密钥数组赋值
    int t = 0;
    for (j = 0; (j < KC) && (t < ROUND_KEY_COUNT); j++, t++) {
        m_Ke[t / BC][t % BC] = tk[j];
        m_Kd[m_iROUNDS - (t / BC)][t % BC] = tk[j];
    }
    int tt, rconpointer = 0;
    while (t < ROUND_KEY_COUNT) {
        //使用 phi 计算
        tt = tk[KC - 1];
```

```
        tk[0] ^= (sm_S[(tt >> 16) & 0xFF] & 0xFF) << 24
            ^ (sm_S[(tt >> 8) & 0xFF] & 0xFF) << 16
            ^ (sm_S[tt & 0xFF] & 0xFF) << 8
            ^ (sm_S[(tt >> 24) & 0xFF] & 0xFF)
            ^ (sm_rcon[rconpointer++] & 0xFF) << 24;
        if (KC != 8)
            for (i = 1, j = 0; i < KC;)
                tk[i++] ^= tk[j++];
        else {
            for (i = 1, j = 0; i < KC / 2;)
                tk[i++] ^= tk[j++];
            tt = tk[KC / 2 - 1];
            tk[KC / 2] ^= (sm_S[tt & 0xFF] & 0xFF)
                ^ (sm_S[(tt >> 8) & 0xFF] & 0xFF) << 8
                ^ (sm_S[(tt >> 16) & 0xFF] & 0xFF) << 16
                ^ (sm_S[(tt >> 24) & 0xFF] & 0xFF) << 24;
            for (j = KC / 2, i = j + 1; i < KC;)
                tk[i++] ^= tk[j++];
        }
        //轮密钥数组赋值
        for (j = 0; (j < KC) && (t < ROUND_KEY_COUNT); j++, t++) {
            m_Ke[t / BC][t % BC] = tk[j];
            m_Kd[m_iROUNDS - (t / BC)][t % BC] = tk[j];
        }
    }
    //逆行混淆
    for (int r = 1; r < m_iROUNDS; r++)
        for (j = 0; j < BC; j++) {
            tt = m_Kd[r][j];
            m_Kd[r][j] = sm_U1[(tt >> 24) & 0xFF] ^ sm_U2[(tt >> 16) & 0xFF]
                ^ sm_U3[(tt >> 8) & 0xFF] ^ sm_U4[tt & 0xFF];
        }
    m_bKeyInit = true;
}
```

定义加密块:

```
void AES::DefEncryptBlock(char const* in, char* result) {
    if (false == m_bKeyInit)
        return;
    int* Ker = m_Ke[0];
    int t0 = ((unsigned char) *(in++) << 24);
    t0 |= ((unsigned char) *(in++) << 16);
```

```
    t0 |= ((unsigned char) *(in++) << 8);
    (t0 |= (unsigned char) *(in++)) ^= Ker[0];
    int t1 = ((unsigned char) *(in++) << 24);
    t1 |= ((unsigned char) *(in++) << 16);
    t1 |= ((unsigned char) *(in++) << 8);
    (t1 |= (unsigned char) *(in++)) ^= Ker[1];
    int t2 = ((unsigned char) *(in++) << 24);
    t2 |= ((unsigned char) *(in++) << 16);
    t2 |= ((unsigned char) *(in++) << 8);
    (t2 |= (unsigned char) *(in++)) ^= Ker[2];
    int t3 = ((unsigned char) *(in++) << 24);
    t3 |= ((unsigned char) *(in++) << 16);
    t3 |= ((unsigned char) *(in++) << 8);
    (t3 |= (unsigned char) *(in++)) ^= Ker[3];
    int a0, a1, a2, a3;
    //轮加密
    for (int r = 1; r < m_iROUNDS; r++) {
        Ker = m_Ke[r];
        a0 = (sm_T1[(t0 >> 24) & 0xFF] ^ sm_T2[(t1 >> 16) & 0xFF]
            ^ sm_T3[(t2 >> 8) & 0xFF] ^ sm_T4[t3 & 0xFF]) ^ Ker[0];
        a1 = (sm_T1[(t1 >> 24) & 0xFF] ^ sm_T2[(t2 >> 16) & 0xFF]
            ^ sm_T3[(t3 >> 8) & 0xFF] ^ sm_T4[t0 & 0xFF]) ^ Ker[1];
        a2 = (sm_T1[(t2 >> 24) & 0xFF] ^ sm_T2[(t3 >> 16) & 0xFF]
            ^ sm_T3[(t0 >> 8) & 0xFF] ^ sm_T4[t1 & 0xFF]) ^ Ker[2];
        a3 = (sm_T1[(t3 >> 24) & 0xFF] ^ sm_T2[(t0 >> 16) & 0xFF]
            ^ sm_T3[(t1 >> 8) & 0xFF] ^ sm_T4[t2 & 0xFF]) ^ Ker[3];
        t0 = a0;
        t1 = a1;
        t2 = a2;
        t3 = a3;
    }
    //最后一轮加密不同
    Ker = m_Ke[m_iROUNDS];
    int tt = Ker[0];
    result[0] = sm_S[(t0 >> 24) & 0xFF] ^ (tt >> 24);
    result[1] = sm_S[(t1 >> 16) & 0xFF] ^ (tt >> 16);
    result[2] = sm_S[(t2 >> 8) & 0xFF] ^ (tt >> 8);
    result[3] = sm_S[t3 & 0xFF] ^ tt;
    tt = Ker[1];
    result[4] = sm_S[(t1 >> 24) & 0xFF] ^ (tt >> 24);
    result[5] = sm_S[(t2 >> 16) & 0xFF] ^ (tt >> 16);
    result[6] = sm_S[(t3 >> 8) & 0xFF] ^ (tt >> 8);
    result[7] = sm_S[t0 & 0xFF] ^ tt;
    tt = Ker[2];
```

```
    result[8] = sm_S[(t2 >> 24) & 0xFF] ^ (tt >> 24);
    result[9] = sm_S[(t3 >> 16) & 0xFF] ^ (tt >> 16);
    result[10] = sm_S[(t0 >> 8) & 0xFF] ^ (tt >> 8);
    result[11] = sm_S[t1 & 0xFF] ^ tt;
    tt = Ker[3];
    result[12] = sm_S[(t3 >> 24) & 0xFF] ^ (tt >> 24);
    result[13] = sm_S[(t0 >> 16) & 0xFF] ^ (tt >> 16);
    result[14] = sm_S[(t1 >> 8) & 0xFF] ^ (tt >> 8);
    result[15] = sm_S[t2 & 0xFF] ^ tt;
}
```

定义解密块:

```
void AES::DefDecryptBlock(char const* in, char* result) {
    if (false == m_bKeyInit)
        return;
    int* Kdr = m_Kd[0];
    int t0 = ((unsigned char) *(in++) << 24);
    t0 = t0 | ((unsigned char) *(in++) << 16);
    t0 |= ((unsigned char) *(in++) << 8);
    (t0 |= (unsigned char) *(in++)) ^= Kdr[0];
    int t1 = ((unsigned char) *(in++) << 24);
    t1 |= ((unsigned char) *(in++) << 16);
    t1 |= ((unsigned char) *(in++) << 8);
    (t1 |= (unsigned char) *(in++)) ^= Kdr[1];
    int t2 = ((unsigned char) *(in++) << 24);
    t2 |= ((unsigned char) *(in++) << 16);
    t2 |= ((unsigned char) *(in++) << 8);
    (t2 |= (unsigned char) *(in++)) ^= Kdr[2];
    int t3 = ((unsigned char) *(in++) << 24);
    t3 |= ((unsigned char) *(in++) << 16);
    t3 |= ((unsigned char) *(in++) << 8);
    (t3 |= (unsigned char) *(in++)) ^= Kdr[3];
    int a0, a1, a2, a3;
    //轮解密
    for (int r = 1; r < m_iROUNDS; r++) {
        Kdr = m_Kd[r];
        a0 = (sm_T5[(t0 >> 24) & 0xFF] ^ sm_T6[(t3 >> 16) & 0xFF]
            ^ sm_T7[(t2 >> 8) & 0xFF] ^ sm_T8[t1 & 0xFF]) ^ Kdr[0];
        a1 = (sm_T5[(t1 >> 24) & 0xFF] ^ sm_T6[(t0 >> 16) & 0xFF]
            ^ sm_T7[(t3 >> 8) & 0xFF] ^ sm_T8[t2 & 0xFF]) ^ Kdr[1];
        a2 = (sm_T5[(t2 >> 24) & 0xFF] ^ sm_T6[(t1 >> 16) & 0xFF]
            ^ sm_T7[(t0 >> 8) & 0xFF] ^ sm_T8[t3 & 0xFF]) ^ Kdr[2];
        a3 = (sm_T5[(t3 >> 24) & 0xFF] ^ sm_T6[(t2 >> 16) & 0xFF]
```

```
                ^ sm_T7[(t1 >> 8) & 0xFF] ^ sm_T8[t0 & 0xFF]) ^ Kdr[3];
            t0 = a0;
            t1 = a1;
            t2 = a2;
            t3 = a3;
        }
        //最后一轮解密不同
        Kdr = m_Kd[m_iROUNDS];
        int tt = Kdr[0];
        result[0] = sm_Si[(t0 >> 24) & 0xFF] ^ (tt >> 24);
        result[1] = sm_Si[(t3 >> 16) & 0xFF] ^ (tt >> 16);
        result[2] = sm_Si[(t2 >> 8) & 0xFF] ^ (tt >> 8);
        result[3] = sm_Si[t1 & 0xFF] ^ tt;
        tt = Kdr[1];
        result[4] = sm_Si[(t1 >> 24) & 0xFF] ^ (tt >> 24);
        result[5] = sm_Si[(t0 >> 16) & 0xFF] ^ (tt >> 16);
        result[6] = sm_Si[(t3 >> 8) & 0xFF] ^ (tt >> 8);
        result[7] = sm_Si[t2 & 0xFF] ^ tt;
        tt = Kdr[2];
        result[8] = sm_Si[(t2 >> 24) & 0xFF] ^ (tt >> 24);
        result[9] = sm_Si[(t1 >> 16) & 0xFF] ^ (tt >> 16);
        result[10] = sm_Si[(t0 >> 8) & 0xFF] ^ (tt >> 8);
        result[11] = sm_Si[t3 & 0xFF] ^ tt;
        tt = Kdr[3];
        result[12] = sm_Si[(t3 >> 24) & 0xFF] ^ (tt >> 24);
        result[13] = sm_Si[(t2 >> 16) & 0xFF] ^ (tt >> 16);
        result[14] = sm_Si[(t1 >> 8) & 0xFF] ^ (tt >> 8);
        result[15] = sm_Si[t0 & 0xFF] ^ tt;
    }
```

加密:

```
    void AES::Encrypt(char const* in, char* result, size_t n, int iMode) {
        if (false == m_bKeyInit)
            return;
        // n 应该大于 0 且 m_blockSize 不等于 0
        if (0 == n || n % m_blockSize != 0)
            return;
        int i;
        char const* pin;
        char* presult;
        if (CBC == iMode) { //CBC 模式
            for (i = 0, pin = in, presult = result; i < n / m_blockSize; i++) {
                Xor(m_chain, pin);
```

```
            EncryptBlock(m_chain, presult);
            memcpy(m_chain, presult, m_blockSize);
            pin += m_blockSize;
            presult += m_blockSize;
        }
    }
    else if (CFB == iMode) { //CFB 模式
        for (i = 0, pin = in, presult = result; i < n / m_blockSize; i++) {
            EncryptBlock(m_chain, presult);
            Xor(presult, pin);
            memcpy(m_chain, presult, m_blockSize);
            pin += m_blockSize;
            presult += m_blockSize;
        }
    }
    else { //ECB 模式
        for (i = 0, pin = in, presult = result; i < n / m_blockSize; i++) {
            EncryptBlock(pin, presult);
            pin += m_blockSize;
            presult += m_blockSize;
        }
    }
}
```

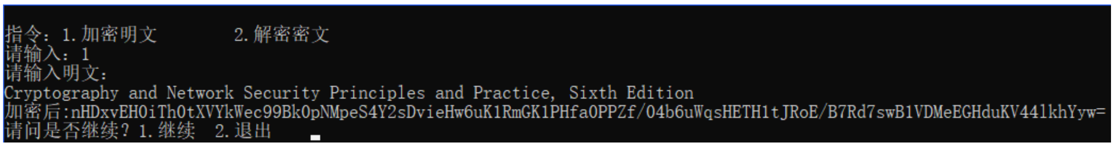
解密:

```
void AES::Decrypt(char const* in, char* result, size_t n, int iMode) {
    if (false == m_bKeyInit)
        return;
    // n 应该大于 0 且 m_blockSize 不等于 0
    if (0 == n || n % m_blockSize != 0)
        return;
    int i;
    char const* pin;
    char* presult;
    if (CBC == iMode) { //CBC 模式
        for (i = 0, pin = in, presult = result; i < n / m_blockSize; i++) {
            DecryptBlock(pin, presult);
            Xor(presult, m_chain);
            memcpy(m_chain, pin, m_blockSize);
            pin += m_blockSize;
            presult += m_blockSize;
        }
    }
```

```
    else if (CFB == iMode) {//CFB 模式
        for (i = 0, pin = in, presult = result; i < n / m_blockSize; i++) {
            EncryptBlock(m_chain, presult);
            Xor(presult, pin);
            memcpy(m_chain, pin, m_blockSize);
            pin += m_blockSize;
            presult += m_blockSize;
        }
    }
    else { //ECB 模式
        for (i = 0, pin = in, presult = result; i < n / m_blockSize; i++) {
            DecryptBlock(pin, presult);
            pin += m_blockSize;
            presult += m_blockSize;
        }
    }
}
```

实验结果

运行程序 AES.exe，对“Cryptography and Network Security Principles and Practice, Sixth Edition”加密后得到的密文为“nHDxvEH0iTh0tXVYkWec99Bk0pNMpeS4Y2sDvieHw6uK1RmGK1PHfa0PPZf/04b6uWqsHETH1tJRoE/B7Rd7swB1VDMeEGHduKV44lkhYyw=”, 如图所示：



对密文“nHDxvEH0iTh0tXVYkWec99Bk0pNMpeS4Y2sDvieHw6uK1RmGK1PHfa0PPZf/04b6uWqsHETH1tJRoE/B7Rd7swB1VDMeEGHduKV44lkhYyw=”解密可得明文“Cryptography and Network Security Principles and Practice, Sixth Edition”，如图所示：

```
指令：1.加密明文        2.解密密文
请输入：1
请输入明文：
Cryptography and Network Security Principles and Practice, Sixth Edition
加密后:nHDxvEH0iTh0tXVYkWec99Bk0pNMpeS4Y2sDvieHw6uK1RmGK1PHfa0PPZf/04b6uWqsHETH1tJRoE/B7Rd7swB1VDMeEGHduKV44lkhYyw=
请问是否继续？1.继续  2.退出    1

指令：1.加密明文        2.解密密文
请输入：2
请输入密文：
nHDxvEH0iTh0tXVYkWec99Bk0pNMpeS4Y2sDvieHw6uK1RmGK1PHfa0PPZf/04b6uWqsHETH1tJRoE/B7Rd7swB1VDMeEGHduKV44lkhYyw=
解密后:Cryptography and Network Security Principles and Practice, Sixth Edition
请问是否继续？1.继续  2.退出
```

由此可以看出 AES 加密和解密过程正确无误。