컴퓨터 보안

-과제2 AES 구현-

12171651 컴퓨터공학과 오윤석

1. 구현 환경 및 구현 언어

- 구현 환경: OS: 윈도우10, Compiler: Visual studio 2019 (16.15ver)

- 구현 언어: c++

2. AES 암호화, 복호화 각 단계 설명 및 코드 설명 기재

a. Key expansion

```
//save key into array
string key_file = "C:/Users/YunSeok/Desktop/INHA/학교/3학년/2학기/컴퓨터보안/과제2/homework2/homework2/key.bin";  //실행 할때 경로 바꿔주세요


ifstream key_input(key_file, ios::in | ios::binary);

key_input.read((char*)key, sizeof(key));

key_input.close();
```

우선 전체적인 코드 실행 순서는 key expansion->Encryption->Decryption으로 진행된다. 처음엔 key.bin file을 read하여 key라는 1차원 배열에 저장을 한다.

(주의. 실행하실 때에는 경로를 변경해 주세요.)

```
for (int i = 0; i < 4; i++)
{
    for (int j = 0; j < 4; j++)
    {
        key_dec_arr[i][j] = key[count];

        count++;
    }
}
```

그 다음에는 1차원의 key 배열을 2차원 (4*4) 배열에 저장을 한 후

```
for (int i = 0; i < 11; i++)
{
    cout << dec;
    cout << "Round " << AES_round << " : ";
    cout << hex;
    for (int j = 0; j < 4; j++) //round key 할당
    {
        for (int k = 0; k < 4; k++)
        {
            round_key[i][j][k] = key_dec_arr[j][k]; //round key arr에서 i는 round
            cout << round_key[i][j][k] << " ";
        }
    }
    cout << endl;
    key_expansion(key_dec_arr);

}
```

각 라운드 별 key를 저장하기 위해 3차원으로 변경 후 key_expansion이라는 함수를 통해 키 변환을 진행한다. (3차원 배열에서 i는 각 라운드를 의미한다.)

```
void key_expansion(int key[][4]) //key expansion
{


    int key_column0[4] = { -1 };
    int key_column1[4] = { -1 };
    int key_column2[4] = { -1 };
    int key_column3[4] = { -1 };

    int past_column0[4] = { -1 };
    int past_column1[4] = { -1 };
    int past_column2[4] = { -1 };
    int past_column3[4] = { -1 };
    int past_past_column3[4] = { -1 };

    for (int i = 0; i < 4; i++)
    {
        key_column0[i] = key[0][i];

    }

    for (int i = 0; i < 4; i++)
    {
        key_column1[i] = key[1][i];

    }

    for (int i = 0; i < 4; i++)
    {
        key_column2[i] = key[2][i];

    }

    for (int i = 0; i < 4; i++)
    {
        key_column3[i] = key[3][i];
        past_past_column3[i] = key[3][i];
    }
```

key_expansion 함수를 처음에 들어오면 처음에는 각 연산의 편의를 위해 각 행으로 array를 나눈다. 이때 마지막 행은 R function후 xor을 위해 past_past_column3에 따로 저장을 해둔다.

```
key_substitue(key_column3); //s_box 치환 완료

key_column3[0] = key_column3[0] ^  rc[0][AES_round]; //RC, 2행 XOR 연산


AES_round++;
```

그 다음은 R function인데 우선 마지막 열만 key_substitute 함수에 보낸 후 rc와 연산을 하면 끝나고, round를 1개 증가시킨다.

```
void key_substitue(int* key_arr) //key값을 s_box에 치환
{

    int dmp0 = key_arr[0];
    int dmp1 = key_arr[1];
    int dmp2 = key_arr[2];
    int dmp3 = key_arr[3];

    key_arr[0] = dmp1; //1행->0행

    key_arr[1] = dmp2; //2행->1행

    key_arr[2] = dmp3; //3행->2행

    key_arr[3] = dmp0; //0행->3행

    int key_column; //열, 가로
    int key_row; //행, 세로



    for (int i = 0; i < 4; i++)
    {

        key_column = key_arr[i] / 16;
        key_row = key_arr[i] % 16;

        key_arr[i] = s_box[key_column][key_row];

    }


}
```

```
//S-Box 선언후 시작
int s_box[16][16] = { {21,10,59,33,2,199,15,188,63,114,221,138,24,153,193,91},
                      {161,47,166,79,208,1,218,80,50,38,83,9,127,96,147,58},
                      {238,206,169,187,204,194,56,248,247,205,190,251,242,32,22,34},
                      {134,145,45,3,151,35,186,117,129,23,175,68,86,4,130,25},
                      {232,17,89,5,234,214,227,229,88,196,254,213,131,27,66,159},
                      {100,136,216,105,192,217,195,253,71,53,143,252,148,155,142,7},
                      {125,84,87,163,168,78,191,113,245,177,14,210,99,55,132,167},
                      {95,198,20,202,72,16,189,6,180,231,157,162,222,176,178,172},
                      {74,67,182,215,146,244,29,118,75,49,85,133,207,135,109,237},
                      {18,112,92,154,224,158,212,120,94,37,179,122,31,51,241,233},
                      {12,137,219,42,243,70,43,62,255,8,115,110,126,220,97,226},
                      {60,123,164,200,249,141,225,11,116,13,82,183,121,150,28,57},
                      {128,104,181,40,52,36,239,44,106,69,184,139,64,223,39,171},
                      {101,173,230,235,152,30,246,209,46,98,165,90,124,170,76,61},
                      {48,228,93,119,149,211,250,144,26,236,54,203,65,140,156,73},
                      {197,185,108,160,240,0,111,107,81,77,102,19,103,41,201,174} };
```

치환 과정은 비교적 간략하게 작성하였다. 우선 1행을 0행으로, 2행을 1행으로, 3행을 2행으로, 0행을 3행으로 보낸 뒤 시작된다. 그 다음 for문을 통해 해당 값의 16으로 나눈 몫은 s box의 세로축이 되고, 나머지는 가로축이 되어 미리 선언된 s_box arr를 통해 치환된다. 그 후 이전 코드로 돌아가서 rc와 2행과 xor연산을 한다.

```
for (int i = 0; i < 4; i++)
{
    past_column0[i] = key_column0[i];

}

for (int i = 0; i < 4; i++)
{
    past_column1[i] = key_column1[i];

}

for (int i = 0; i < 4; i++)
{
    past_column2[i] = key_column2[i];

}

for (int i = 0; i < 4; i++)
{
    past_column3[i] = key_column3[i];

}
```

```
for (int i = 0; i < 4; i++)
{
    key_column0[i] = past_column0[i] ^ key_column3[i];

}

for (int i = 0; i < 4; i++)
{
    key_column1[i] = key_column0[i]^past_column1[i] ;

}

for (int i = 0; i < 4; i++)
{
    key_column2[i] = key_column1[i]^ past_column2[i];

}


for (int i = 0; i < 4; i++)
{
    key_column3[i] = key_column2[i]^ past_past_column3[i];

}
```

```
//다음 round key 할당
for (int i = 0; i < 4; i++)
{
    key[0][i] = key_column0[i];

}

for (int i = 0; i < 4; i++)
{
key[1][i]= key_column1[i];

}

for (int i = 0; i < 4; i++)
{
    key[2][i]= key_column2[i];

}

for (int i = 0; i < 4; i++)
{
    key[3][i] = key_column3[i];
}
```

이후 각 행끼리 xor연산을 하여 다음 round key를 생성하기 위해 우선 past_column arr를 각 행마다 할당한다. 그리고 다음 round key를 생성하게 되는데, 다음 라운드 0번째 행은 이전 0행과 r function을 통과한 3행과 xor연산을 하고, 1번째 행은 이전 1번째 행과 다음 라운드 0번째 행과 xor연산을 한다. 2번째, 3번째 행 또한 다음 라운드 이전 행과, 이전 라운드 같은 행과 xor연산을 하여 최종적으로 다음 round key를 생성하게 되고, 0라운드부터 10라운드까지 round key가 생성이 된다.

```
int num = 0;

for (int i = 0; i < 11; i++)        //key를 2차원 배열로 변환
{
    for (int j = 0; j < 4; j++)
    {
        if (num == 16)
        {
            num = 0;
        }

        for (int k = 0; k < 4; k++)
        {
            real_round_key[i][num] = round_key[i][j][k];
            num++;
        }

    }
}

cout << endl;
```

그 다음에는 연산의 편의성을 위해 3차원의 round key를 2차원 real_round-key로 변환시킨다.

```
KEY EXPANSION
Round 0 : 0 10 20 30 40 50 60 70 80 90 a0 b0 c0 d0 e0 f0
Round 1 : 64 20 e5 b0 24 70 85 c0 a4 e0 25 70 64 30 c5 80
Round 2 : e0 4 af 18 c4 74 2a d8 60 94 f a8 4 a4 ca 28
Round 3 : 17 bc 58 1a d3 c8 72 c2 b3 5c 7d 6a b7 f8 b7 42
Round 4 : 4e b7 1 11 9d 7f 73 d3 2e 23 e b9 99 db b9 fb
Round 5 : 4 ba 12 34 99 c5 61 e7 b7 e6 6f 5e 2e 3d d6 a5
Round 6 : 20 4c 54 22 b9 89 35 c5 e 6f 5a 9b 20 52 8c 3e
Round 7 : b8 83 d6 cc 1 a e3 9 f 65 b9 92 2f 37 35 ac
Round 8 : 4d a0 a8 ee 4c aa 4b e7 43 cf f2 75 6c f8 c7 d9
Round 9 : 51 8c ca 8d 1d 26 81 6a 5e e9 73 1f 32 11 b4 c6
Round 10 : e4 75 25 a0 f9 53 a4 ca a7 ba d7 d5 95 ab 63 13
```

각 round key는 이렇게 생성이 되었다.

b. Encrytion

```
//ENCRYPTION start!
for (int i = 0; i < 11; i++)
{
    cout << dec;
    cout << "ROUND " << i << endl;
    cout << hex;
    if (i == 0)
    {

            AR(plain_text, real_round_key[i]);



    }

    else if (i == 10)
    {
        SB(plain_text);
        SR(plain_text);
        AR(plain_text, real_round_key[i]);
    }

    else
    {
        SB(plain_text);
        SR(plain_text);
        MC(plain_text);
        AR(plain_text, real_round_key[i]);


    }

    cout << endl;
}

cout << endl;
```

Encryption 과정의 전체적인 틀은 첫 번째 라운드는 Add round key만 해주고 마지막 라운드는 SB, SR, AR과정만 해주고, 나머지 라운드는 SB, SR, MC, AR과정을 모두 해준다.

우선 SB 과정에 대해 설명하겠다.

```
void SB(int* plain) //key값을 s_box에 치환
{

    int key_column; //열, 가로
    int key_row; //행, 세로


    cout << "SB: ";

    for (int i = 0; i < 16; i++)
    {

        key_column = plain[i] / 16;
        key_row = plain[i] % 16;

        plain[i] = s_box[key_column][key_row];

        cout << plain[i] << " ";

    }
    cout << endl;


}
```

SB 과정은 아까 key expansion에서 s box와 치환한 것과 같이 똑 같은 방식으로 진행된다. 이 과정 후 plain arr는 치환이 완료된다.

```
void SR(int* plain)
{
    int row1[4] = { -1 };
    int row2[4] = { -1 };
    int row3[4] = { -1 };
    int row4[4] = { -1 };

    int a = 0;
    int b = 0;
    int c = 0;
    int d = 0;

    for (int i = 0; i < 16; i++) //연산 편의를 위해 각 row로 나누기
    {
        if (i % 4 == 0)
        {
            row1[a] = plain[i];
            a++;
        }

        else if (i % 4 == 1)
        {
            row2[b] = plain[i];
            b++;
        }
        else if (i % 4 == 2)
        {
            row3[c] = plain[i];
            c++;

        }

        else if (i % 4 == 3)
        {
            row4[d] = plain[i];
            d++;
        }
    }
```

다음은 SR이다. 이 함수 또한 시작은 연산의 편의를 위해 각 row를 배열에 따로 할당을 한다.

```
int dmp = row2[0];

for (int i = 0; i < 3; i++) //row2 1번 left shift
{
    row2[i] = row2[i + 1];
}
row2[3] = dmp;


for (int i = 0; i < 2; i++) //row3 2번 left shift
{
    dmp = row3[0];

    for (int j = 0; j < 3; j++)
    {
        row3[j] = row3[j + 1];
    }

    row3[3] = dmp;
}

for (int i = 0; i < 3; i++) //row4 3번 left shift
{
    dmp = row4[0];

    for (int j = 0; j < 3; j++)
    {
        row4[j] = row4[j + 1];
    }

    row4[3] = dmp;
}
```

그 후 각 row는 순서대로 shift를 하게 되는데, 1번째 row는 0번 left shift, 2번째 row는 1번 left shift, 3번째 row는 2번 left shift, 4번째 row는 3번 left shift된다.

```
for (int i = 0; i < 4; i++)
{
    plain[i * 4] = row1[i];
}

for (int i = 0; i < 4; i++)
{
    plain[(i * 4)+1] = row2[i];
}

for (int i = 0; i < 4; i++)
{
    plain[(i * 4) + 2] = row3[i];
}

for (int i = 0; i < 4; i++)
{
    plain[(i * 4) + 3]= row4[i];
}

cout << "SR: ";

for (int i = 0; i < 16; i++)
{
    cout << plain[i] << " ";
}
cout << endl;
```

이후 나눠진 row를 다시 1차원 plain arr에 합침으로써 SR은 종료된다.

```
//Mix column 과정 연산자
#define xtime(x)   ((x<<1) ^ (((x>>7) & 1) * 0x4d))
#define Multiply(x, y)                                \
      (  ((y & 1) * x) ^                              \
      ((y>>1 & 1) * xtime(x)) ^                       \
      ((y>>2 & 1) * xtime(xtime(x))) ^                \
      ((y>>3 & 1) * xtime(xtime(xtime(x)))) ^         \
      ((y>>4 & 1) * xtime(xtime(xtime(xtime(x))))))   \
```

그 다음 MC과정인데 그 전에 MC와 Inverse MC과정을 위해 xtime과 Multiply 함수를 미리 정의해준다. 여기서 0x4d는 우리의 과제의 Irreducible Polynomials 가 $x^8 + x^6 + x^3 + x^2 + 1$ 이기 때문에 0x4d로 설정했다. 이때 덧셈은 xor, 곱셈은 shift로 구현하였다.

```
void MC(int* plain)
{
    int plain_column[4][4] = { -1 };
    int num1 = 0;
    int num2 = 0;
    uint8_t i;
    uint8_t Tmp, Tm, t;

    for (int i = 0; i < 4; i++)
    {
        for (int j = 0; j < 4; j++)
        {
            plain_column[i][j] = plain[num1];
            num1++;
        }
    }
```

우선 8비트 크기의 부호 없는 정수가 MC 과정에서 쓰이기 때문에 i, Tmp, Tm, t를 선언을 한 후 연산의 편의를 위해 2차원(4*4) 배열로 변환을 한다.

```
for (i = 0; i < 4; ++i)
{
    t = plain_column[i][0];
    Tmp = plain_column[i][0] ^ plain_column[i][1] ^ plain_column[i][2] ^ plain_column[i][3];
    Tm = plain_column[i][0] ^ plain_column[i][1]; Tm = xtime(Tm);  plain_column[i][0] ^= Tm ^ Tmp;
    Tm = plain_column[i][1] ^ plain_column[i][2]; Tm = xtime(Tm); plain_column[i][1] ^= Tm ^ Tmp;
    Tm = plain_column[i][2] ^ plain_column[i][3]; Tm = xtime(Tm); plain_column[i][2] ^= Tm ^ Tmp;
    Tm = plain_column[i][3] ^ t;   Tm = xtime(Tm); plain_column[i][3] ^= Tm ^ Tmp;
}


    for (int i = 0; i < 4; i++)
    {
        for (int j = 0; j < 4; j++)
        {
            plain[num2] = plain_column[i][j];
            num2++;
        }
    }

    cout << "MC: ";

    for (int i = 0; i < 16; i++)
    {
        cout << plain[i] << " ";
    }

    cout << endl;
```

이후 행렬 곱 연산이 시작된다. 행렬 곱 연산은 계산의 편의를 위해 위에서 정의한 xtime을 사용하여 연산을 한다. 각 행의 첫번쨰 원소는 수업 시간에 배운 것처럼 (2*첫번쨰 원소)+(3*두번째 원소)+세번째 원소+네번째 원소가 되고(여기서 *는 점곱), 2번째 원소, 3번째, 4번째 원소는 각 점곱 2와 3이 1만큼 right shift하게 된다. 이 과정을 통해 MC는 끝나게 된다.

```
void AR(int* plain, int* key)
{
    for (int i = 0; i < 16; i++)
    {
        plain[i] = plain[i] ^ key[i];
    }

    cout << "AR: ";

    for (int j = 0; j < 16; j++)
    {
        cout << plain[j] << " ";
    }
    cout << endl;
}
```

다음은 AR함수이다. AR은 단순히 각 round key와 plain의 xor연산으로 이루어 진다.

```
ofstream output("cipher.bin", ios::out | ios::binary);

output.write((char*)plain_text, sizeof(plain_text) - 1);

output.close();
```

총 0~10 라운드로 구성된 라운드를 거치고 나면 cipher text가 생성이된다. 이 cipher를 bin파일로 출력을 하게되면

```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000   FF 00 00 00 12 00 00 00 A8 00 00 00 96 00 00 00  ÿ.......¨...-...
00000010   D7 00 00 00 FA 00 00 00 D3 00 00 00 FC 00 00 00  ×...ú...Ó...ü...
00000020   19 00 00 00 B2 00 00 00 CA 00 00 00 97 00 00 00  ....²...Ê...—...
00000030   3F 00 00 00 02 00 00 00 C1 00 00 00 6E 00 00     ?.......Á...n..
```

```
CIPHER: ff 12 a8 96 d7 fa d3 fc 19 b2 ca 97 3f 2 c1 6e
```

이렇게 cipher.bin이 생성된다.

c. Decryption

```
cout << "<----- DECRYPTION ----->" << endl << endl;
int round = 0;
for (int i = 10; i > -1; i--)
{


    cout << dec;
    cout << "ROUND " << round << endl;
    cout << hex;
    if (i == 10)
    {
        AR(plain_text, real_round_key[i]);
        round++;
    }

    else if (i == 0)
    {
        Inverse_SR(plain_text);
        Inverse_SB(plain_text);
        AR(plain_text, real_round_key[i]);
        round++;

        cout << endl;
        cout << "DECRYPTED: ";

        for (int j = 0; j < 16; j++)
        {
            cout << plain_text[j] << " ";
        }
        cout << endl;

    }

    else
    {
        Inverse_SR(plain_text);
        Inverse_SB(plain_text);
        AR(plain_text, real_round_key[i]);
        inverse_MC(plain_text);
        round++;
    }
```

Decryption과정도 전체적인 틀은 Encryption과정과 같다. 하지만 반대로 첫번째 cipher는 AR만 진행이 되고, 마지막은 Inverse-SR, Inverse-SB, AR, 나머지는 Inverse-SR, Inverse-SB, AR, Inverse-MC 과정으로 진행된다.

우선 Inverse_SR을 보자

```
void Inverse_SR(int* plain)
{
    int row1[4] = { -1 };
    int row2[4] = { -1 };
    int row3[4] = { -1 };
    int row4[4] = { -1 };

    int a = 0;
    int b = 0;
    int c = 0;
    int d = 0;

    for (int i = 0; i < 16; i++) //연산 편의를 위해 각 row로 나누기
    {
        if (i % 4 == 0)
        {
            row1[a] = plain[i];
            a++;
        }

        else if (i % 4 == 1)
        {
            row2[b] = plain[i];
            b++;
        }
        else if (i % 4 == 2)
        {
            row3[c] = plain[i];
            c++;

        }

        else if (i % 4 == 3)
        {
            row4[d] = plain[i];
            d++;
        }
    }
```

이 함수 또한 연산의 편의를 위해 각 row로 나누었다.

```
shiftRight(row2, 1, 4);
shiftRight(row3, 2, 4);
shiftRight(row4, 3, 4);



for (int i = 0; i < 4; i++)
{
    plain[i * 4] = row1[i];
}

for (int i = 0; i < 4; i++)
{
    plain[(i * 4) + 1] = row2[i];
}

for (int i = 0; i < 4; i++)
{
    plain[(i * 4) + 2] = row3[i];
}

for (int i = 0; i < 4; i++)
{
    plain[(i * 4) + 3] = row4[i];
}

cout << "SR: ";

for (int i = 0; i < 16; i++)
{
    cout << plain[i] << " ";
}
cout << endl;
```

그 다음은 SR과는 반대로 오른쪽으로 해당열(0열은 0번, 1열은 1번...)만큼 오른쪽으로 SHIFT해준다.

```
void shiftRight(int arr[], int d, int n) {
    reverse(arr, 0, n - d);
    reverse(arr, n - d, n);
    reverse(arr, 0, n);
}
```

shiftRight함수는 reverse를 통해 간단하게 구현하였다.

Shift가 끝나면 다시 plain arr로 합치면서 Inverse_SR은 마무리된다.

```
for (int i = 0; i < 16; i++)
{
    count1++;
    for (int j = 0; j < 16; j++)
    {
        for (int k = 0; k < 16; k++)
        {
            if (plain[i] == s_box[j][k])
            {
                str_column = convert2hex(j);
                str_row = convert2hex(k);

                original = str_column + str_row;

                ch_original = original.c_str();

                plain[i] = (int)strtol(ch_original, NULL, 16);

                cout << plain[i] << " ";


                    goto here;
            }
        }


    }

here:
    ;
```

Inverse_SB 함수부터 보자

Inverse_sb는 for문을 통해 s-box에서 일치하는 element를 찾으면, 각 element의 왼쪽, 오른쪽부분의 count를 convert2hex함수로 16진수로 str_column, str_row로 할당한다. 그후 이 둘을 다시 합치고 다시 int 형으로 변환 후 plain arr에 할당한다.

그 다음은 AR과정인데, 이전 AR과 같은 함수라 생략하겠다.

```
void inverse_MC(int* plain)
{
    int plain_column[4][4] = { -1 };
    int num1 = 0;
    int num2 = 0;

    for (int i = 0; i < 4; i++)
    {
        for (int j = 0; j < 4; j++)
        {
            plain_column[i][j] = plain[num1];
            num1++;
        }
    }

    int i;
    uint8_t a, b, c, d;
    for (i = 0; i < 4; ++i)
    {
        a = plain_column[i][0];
        b = plain_column[i][1];
        c = plain_column[i][2];
        d = plain_column[i][3];

        plain_column[i][0] = Multiply(a, 0x0e) ^ Multiply(b, 0x0b) ^ Multiply(c, 0x0d) ^ Multiply(d, 0x09);

        plain_column[i][1] = Multiply(a, 0x09) ^ Multiply(b, 0x0e) ^ Multiply(c, 0x0b) ^ Multiply(d, 0x0d);

        plain_column[i][2] = Multiply(a, 0x0d) ^ Multiply(b, 0x09) ^ Multiply(c, 0x0e) ^ Multiply(d, 0x0b);

        plain_column[i][3] = Multiply(a, 0x0b) ^ Multiply(b, 0x0d) ^ Multiply(c, 0x09) ^ Multiply(d, 0x0e);

    }
```

다음은 Inverse_MC이다. 이 과정역시 MC와 비슷하지만 반대가 되는 연산이다. 여기서는 연산의 편의를 위해 Multuply가 쓰였다.

```
#define Multiply(x, y)                                \
      (  ((y & 1) * x) ^                              \
      ((y>>1 & 1) * xtime(x)) ^                       \
      ((y>>2 & 1) * xtime(xtime(x))) ^                \
      ((y>>3 & 1) * xtime(xtime(xtime(x)))) ^         \
      ((y>>4 & 1) * xtime(xtime(xtime(xtime(x))))))   \
```

Multuply는 x와 y가 주어지면 xtime을 이용하여 역 연산을 구해주는 역할을 한다.

다시 Inverse_MC로 돌아와 연산 과정을 보면 아까 MC와는 반대로 행렬 곱을 통해 구현이 된 것을 볼 수 있다.

이 과정들을 모두 거치게 되면 해독된 plain text가 나오게 된다.

```
DECRYPTED: 0 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff
```

```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000   00 00 00 00 11 00 00 00 22 00 00 00 33 00 00 00  ........"...3...
00000010   44 00 00 00 55 00 00 00 66 00 00 00 77 00 00 00  D...U...f...w...
00000020   88 00 00 00 99 00 00 00 AA 00 00 00 BB 00 00 00  ˆ...™...ª...»...
00000030   CC 00 00 00 DD 00 00 00 EE 00 00 00 FF 00 00     Ì...Ý...î...ÿ..
```