암호학 HW2

2018037356 - 안동현

(아래 사진들은 코드의 캡쳐를 위해 주석을 지운 상태입니다. 본 파일에는 주석이 존재합니다)

#1 부가적 함수들

```
52
53 static uint32_t droundKey[RNDKEYLEN];
54
55 uint32_t SubWord(const uint32_t);
56 uint32_t RotWord(const uint32_t);
57 void AddRoundKey(uint8_t*, const uint32_t*);
58 void SubBytes(uint8_t*, int);
59 void Shift(uint8_t*, int, int, int);
60 void ShiftRows(uint8_t*, int);
61 uint8_t Mul(uint8_t, uint8_t);
62 void MixColumns(uint8_t*, int);
63
```

이번 과제를 진행하면서 스켈레톤 코드를 제외하고 작성한 추가 함수들의 목록입니다.

KeyExpansion을 위한 SubWord, RotWord가 있고,

Cipher을 위한 AddRoundKey, SubBytes, ShiftRows, MixColumns가 있고,

ShiftRows를 쉽게 도와주기 위한 Shift와

MixColumns를 쉽게 도와주기 위해 이전 과제에서 진행한 GF(2^8)에서의 다항식 계산 함수인 Mul이 있습니다.

1. uint32_t SubWord(const uint32_t rw);

```
106 uint32_t SubWord(const uint32_t rw){
107     return ((uint32_t)sbox[(rw >> 24) & 0xFF] << 24) |
108            ((uint32_t)sbox[(rw >> 16) & 0xFF] << 16) |
109            ((uint32_t)sbox[(rw >> 8) & 0xFF] << 8) |
110            ((uint32_t)sbox[rw & 0xFF]);
111 }
112
```

해당 함수는 KeyExpansion에서 사용되는 함수로 키 확장을 할 때 g연산을 진행하게 되는데 그때 32비트 정수의 각 위치의 8비트 정수들을 sbox에 넣어서 나온 새로운 8비트 정수값으로 다시 32비트 정수를 구성하는 함수다. 각각을 24, 16, 8, 0비트씩 오른쪽으로 민 것을 0xFF와 and 연산을 통해 8비트값을 얻어내고 그것을 32비트로 캐스팅 한 후에 원래 위치로 다시 왼쪽으로 밀어 or 연산으로 합쳐주는 과정이다. 리틀 엔디안, 빅 엔디안 방식에 상관 없이 어차피 원래 위치로 돌려주기만 하면 되기에 보기 편하게 8비트 캐스팅을 통해 4개로 나누어 구현하지 않고, 32비트 정수값에 한번에 접근해서 구현하였다. 8비트 캐스팅을 하면 0, 1, 2, 3 위치에 한번씩 메모리 접근을 해야해서 메모리 접근이 많이 필요하지만 해당 방식으로 구현하면 32비트에 한번에 접근해서 구현하는 것이 가능하다.

2. uint32_t RotWord(const uint32_t w);

```
113 uint32_t RotWord(const uint32_t w){
114     return (w >> 8) | (w << 24);
115 }
116
```

들어온 32비트 정수를 LRot 해주는 함수이다. 하지만 실제 코드 구현은 RRot처럼 보이는데, 그 이유는 현재 인텔 cpu를 사용중이기에 리틀 엔디안 방식이기 때문이다. 들어온 32비트 정수를 8비트 왼쪽으로 민 것과, 24비트 오른쪽으로 민 것을 or 연산을 통해 합쳐주었다.

3. void AddRoundKey(uint8_t *state, const uint32_t *roundKey);

```
166 // 라운드 키를 XOR 연산을 사용하여 state에 더한다.
167 void AddRoundKey(uint8_t *state, const uint32_t *roundKey){
168     uint8_t *p;
169     for (int i = 0; i < Nb; i++){
170         p = (uint8_t*)(roundKey+i);
171         state[0 + i*Nb] = state[0 + i*Nb] ^ p[0];
172         state[1 + i*Nb] = state[1 + i*Nb] ^ p[1];
173         state[2 + i*Nb] = state[2 + i*Nb] ^ p[2];
174         state[3 + i*Nb] = state[3 + i*Nb] ^ p[3];
175     }
176 }
177
```

현재 state의 열 부분과, roundKey를 XOR 연산을 해주는 함수이다. roundKey에 대한 인덱스 접근을 편하게 하기 위해, for문을 Nb만큼만 돌리고, state는 4줄을 써주었다. 또한 보기 좋게 작성함과 동시에 32비트 정수를 shift하고, & 연산을 해주는 것이 보기에 번거로워 8비트 정수의 포인터로 캐스팅해서 받은 후 각각에 접근해서 XOR 연산을 진행했다.

State를 보면 i*Nb 방식으로 인덱스에 접근하고 있는데, 이를 이용하면 매 반복마다.

0,1,2,3 | 4,5,6,7 | 8,9,10,11 이런식으로 접근이 가능하다.

4. void SubBytes(uint8_t *state, int mode);

```
178 // mode에 따라 순방향 또는 역방향으로 바이트를 치환한다.
179 void SubBytes(uint8_t *state, int mode){
180
181     if (mode == ENCRYPT){
182         for (int i = 0; i < BLOCKLEN; i++){
183             state[i] = sbox[state[i]];
184         }
185     }
186     else if (mode == DECRYPT){
187         for (int i = 0; i < BLOCKLEN; i++){
188             state[i] = isbox[state[i]];
189         }
190     }
191
192 }
```

단순히 암호화일 경우 sbox에, 복호화일 경우 isbox에 접근해서 해당 값으로 바꿔주는 함수이다. state의 길이는 BLOCKLEN이므로 해당 길이만큼 반복을 시켜줬다.

5. void Shift(uint8_t *state, int start, int term, int n);

```
194 void Shift(uint8_t *state, int start, int term, int n){
195     uint8_t tem[n];
196     for (int i = 0; i < n; i++)
197         tem[i] = state[start + i*term];
198     for (int i = 0; i < term - n; i++)
199         state[start + i*term] = state[start + (i+n) * term];
200     for (int i = 0; i < n; i++)
201         state[start + (term - n + i) * term] = tem[i];
202 }
```

ShiftRows 연산을 편하게 해주기 위해 만든 함수이다. 각 인자는 이를 의미한다.

state -> 기존 state

start -> Shift작업을 해줄 행의 위치

term -> 각 원소의 간격은 얼마인가?

n -> 몇 개의 위치를 변경할 것인가?

문서를 찾아보면 각각의 ShiftRows는 다음과 같은 과정을 거친다.

(기본)

Figure 8 illustrates the **ShiftRows()** transformation.

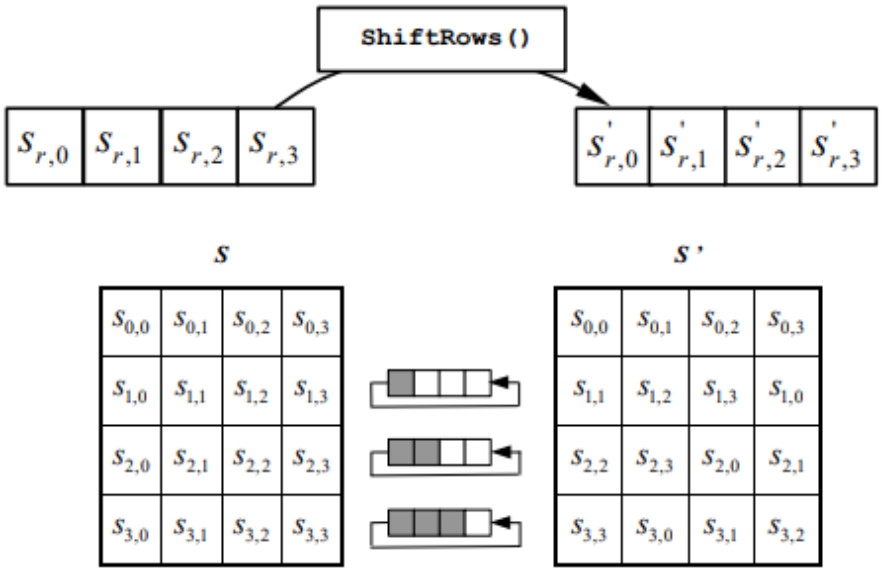


Figure 8. `ShiftRows()` cyclically shifts the last three rows in the State.

(역)

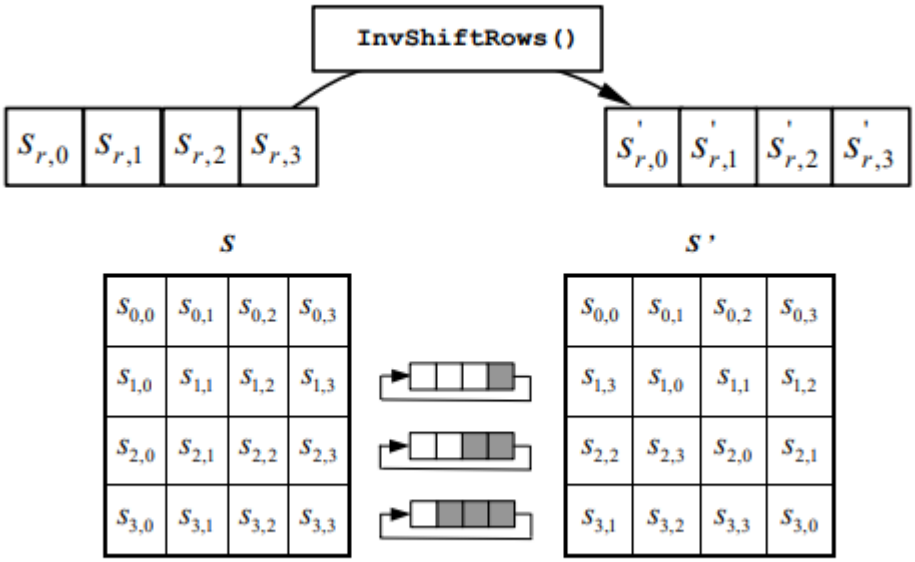


Figure 13. `InvShiftRows()` cyclically shifts the last three rows in the State.

이를 잘 살펴보면 1번은 1개를 뒤로 옮기고, 2번은 2개를 뒤로 옮기고, 3번은 3개를 뒤로 옮긴다. 또 InvShift에서는 1번은 1개를 앞으로 옮기는데, 이건 3개를 뒤로 옮기는 것과 같다. 나머지도 마찬가지인다. 따라서 해당 함수만 있다면 암호화, 복호화에서 그냥 해당 함수만 사용해주면 된다.

원리는 앞쪽의 옮길 원소들을 임시 배열에 저장해두고, 해당 원소를 제외한 나머지를 앞으로 끌고와준다. 그리고 나머지 뒤쪽에 임시배열에 저장해 두었던 걸 넣어주면 된다.

6. void ShiftRows(uint8_t *state, int mode);

```
204 // mode에 따라 순방향 또는 역방향으로 바이트의 위치를 변경한다.
205 void ShiftRows(uint8_t *state, int mode){
206     if (mode == ENCRYPT){
207         Shift(state, 1, Nb, 1);
208         Shift(state, 2, Nb, 2);
209         Shift(state, 3, Nb, 3);
210     }
211     else if (mode == DECRYPT){
212         Shift(state, 1, Nb, 3);
213         Shift(state, 2, Nb, 2);
214         Shift(state, 3, Nb, 1);
215     }
216 }
217
```

그렇게 완성된 ShiftRows 함수이다. 암호화 부분은 1개의 앞쪽 원소를 뒤로 보내기, 2개의 앞쪽 원소들을 뒤로 보내기, 3개의 앞쪽 원소들을 뒤로 보내기를 의미하고,

복호화 부분은 3개의 앞쪽 원소들을 뒤로 보내기, 2개의 앞쪽 원소들을 뒤로 보내기, 1개의 앞쪽 원소를 뒤로 보내기를 의미한다.

7. uint8_t Mul(uint8_t a, uint8_t b);

```
218
219 uint8_t Mul(uint8_t a, uint8_t b){
220     uint8_t r = 0;
221     while (b > 0){
222         if (b & 1) r = r^a;
223         b = b >> 1;
224         a = XTIME(a);
225     }
226     return r;
227 }
228
```

이전에 배운 GF(2^8) 에서의 다항식 곱셈 함수이다. MixColumns의 기약 다항식 x^8 + x^4 + x^3 + x + 1을 사용한 GF(2^8)에서 행렬 곱셈에서 이용된다.

8. void MixColumns(uint8_t *state, int mode);

```
232 void MixColumns(uint8_t *state, int mode){
233     uint8_t tem[BLOCKLEN];
234     memcpy(tem, state, sizeof(uint8_t)*BLOCKLEN);
235     if (mode == ENCRYPT){
236         for (int i = 0; i < Nb; i++){
237             state[0+i*4] = Mul(tem[0+i*4], M[0]) ^
238                            Mul(tem[1+i*4], M[1]) ^
239                            tem[2+i*4] ^
240                            tem[3+i*4];
241
242             state[1+i*4] = tem[0+i*4] ^
243                            Mul(tem[1+i*4], M[5]) ^
244                            Mul(tem[2+i*4], M[6]) ^
245                            tem[3+i*4];
246
247             state[2+i*4] = tem[0+i*4] ^
248                            tem[1+i*4] ^
249                            Mul(tem[2+i*4], M[10]) ^
250                            Mul(tem[3+i*4], M[11]);
251
252             state[3+i*4] = Mul(tem[0+i*4], M[12]) ^
253                            tem[1+i*4] ^
254                            tem[2+i*4] ^
255                            Mul(tem[3+i*4], M[15]);
256         }
257     }
258     else if (mode == DECRYPT){
259         for (int i = 0; i < Nb; i++){
260             state[0+i*4] = Mul(tem[0+i*4], IM[0]) ^
261                            Mul(tem[1+i*4], IM[1]) ^
262                            Mul(tem[2+i*4], IM[2]) ^
263                            Mul(tem[3+i*4], IM[3]);
264
265             state[1+i*4] = Mul(tem[0+i*4], IM[4]) ^
266                            Mul(tem[1+i*4], IM[5]) ^
267                            Mul(tem[2+i*4], IM[6]) ^
268                            Mul(tem[3+i*4], IM[7]);
269
270             state[2+i*4] = Mul(tem[0+i*4], IM[8]) ^
271                            Mul(tem[1+i*4], IM[9]) ^
272                            Mul(tem[2+i*4], IM[10]) ^
273                            Mul(tem[3+i*4], IM[11]);
274
275             state[3+i*4] = Mul(tem[0+i*4], IM[12]) ^
276                            Mul(tem[1+i*4], IM[13]) ^
277                            Mul(tem[2+i*4], IM[14]) ^
278                            Mul(tem[3+i*4], IM[15]);
279         }
280     }
```

이를 이용하면 MixColumns 연산을 쉽게 할 수 있다.

(암호화 연산)

$$s'_{0,c} = (\{02\} \bullet s_{0,c}) \oplus (\{03\} \bullet s_{1,c}) \oplus s_{2,c} \oplus s_{3,c}$$

$$s'_{1,c} = s_{0,c} \oplus (\{02\} \bullet s_{1,c}) \oplus (\{03\} \bullet s_{2,c}) \oplus s_{3,c}$$

$$s'_{2,c} = s_{0,c} \oplus s_{1,c} \oplus (\{02\} \bullet s_{2,c}) \oplus (\{03\} \bullet s_{3,c})$$

$$s'_{3,c} = (\{03\} \bullet s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \bullet s_{3,c}).$$

(복호화 연산)

$$s'_{0,c} = (\{0e\} \bullet s_{0,c}) \oplus (\{0b\} \bullet s_{1,c}) \oplus (\{0d\} \bullet s_{2,c}) \oplus (\{09\} \bullet s_{3,c})$$

$$s'_{1,c} = (\{09\} \bullet s_{0,c}) \oplus (\{0e\} \bullet s_{1,c}) \oplus (\{0b\} \bullet s_{2,c}) \oplus (\{0d\} \bullet s_{3,c})$$

$$s'_{2,c} = (\{0d\} \bullet s_{0,c}) \oplus (\{09\} \bullet s_{1,c}) \oplus (\{0e\} \bullet s_{2,c}) \oplus (\{0b\} \bullet s_{3,c})$$

$$s'_{3,c} = (\{0b\} \bullet s_{0,c}) \oplus (\{0d\} \bullet s_{1,c}) \oplus (\{09\} \bullet s_{2,c}) \oplus (\{0e\} \bullet s_{3,c})$$

문서의 연산에 맞게 다항식 곱셈을 이용하면 쉽게 해결이 가능하다.

코드는 먼저 uint8_t tem[BLOCKLEN];으로 임시 배열을 만들어주고

memcpy(tem, state, sizeof(uint8_t)*BLOCKLEN); 로 기존 state를 복사해준다. 그리고

아까처럼 Nb만큼 반복을 통해 state의 0,1,2,3 | 4,5,6,7 | 8,9,10,11 에 접근해서 연산을 진행해주면 된다.

#2 메인 함수들

이제 메인 함수들인 KeyExpansion과, Cipher을 보자.

1. void KeyExpansion(const uint8_t *key, uint32_t *roundKey);

```
67 void KeyExpansion(const uint8_t *key, uint32_t *roundKey)
68 {
69     int i = 0;
70     uint32_t temp;
71     uint8_t *p;
72     while (i < Nk){
73         p = (uint8_t*)(roundKey+i);
74         p[0] = key[4*i+0];
75         p[1] = key[4*i+1];
76         p[2] = key[4*i+2];
77         p[3] = key[4*i+3];
78         i++;
79     }
80
81     i = Nk;
82     while (i < Nb * (Nr+1)){
83         temp = roundKey[i-1];
84         if (i % Nk == 0){
85             temp = SubWord(RotWord(temp)) ^ Rcon[i/Nk];
86         }
87         else if (Nk > 6 && (i % Nk == 4))
88             temp = SubWord(temp);
89
90         roundKey[i] = roundKey[i-Nk] ^ temp;
91         i++;
92     }
93
94     for (int i = 0; i < Nb * (Nr+1); i++){
95         droundKey[i] = roundKey[i];
96     }
97
98     for (int i = 1; i < Nr; i++){
99         uint8_t tem[BLOCKLEN];
00         memcpy(tem, droundKey + i*Nb, sizeof(uint8_t) * BLOCKLEN);
01         MixColumns(tem, DECRYPT);
02         memcpy(droundKey + i*Nb, tem, sizeof(uint8_t) * BLOCKLEN);
03     }
04 }
```

키 확장의 과정은 AddRoundKey 연산에서 사용될 w 즉 roundKey를 만드는 과정을 의미한다. 연산을 간단히 설명하자면, 먼저 key는 8비트 정수의 배열이지만 roundKey는 32비트 정수의 배열임을 인지하고 있자. 즉 key의 4개의 원소를 합친 것이 roundKey 원소 하나라는 것이다.

1번째로 w[0,3]까지는 기존의 키값들을 32비트 정수로 만들어 넣어준다.

그 과정이

```
    uint8_t *p;
    while (i < Nk){
        p = (uint8_t*)(roundKey+i);
        p[0] = key[4*i+0];
        p[1] = key[4*i+1];
        p[2] = key[4*i+2];
        p[3] = key[4*i+3];
        i++;
    }
```

해당 부분으로, 리틀 엔디안 방식을 고려해서 보기 편하게 8비트 정수 포인터 타입으로 캐스팅해서 각각의 p[0], p[1] ....에 넣어줬다. key는 각각의 열이 들어가야 하므로 4*i+(0....3) 을 사용해준다.

이제 이후로 나오는 w[4.....] 에서는 이전의 값들을 이용해서 만든다. 방금 4개를 만들었으니 40개만 더 만들면 된다. 그 과정이 이부분이다.

```
i = Nk;
while (i < Nb * (Nr+1)){
    temp = roundKey[i-1];
    if (i % Nk == 0){
        temp = SubWord(RotWord(temp)) ^ Rcon[i/Nk];
    }
    else if (Nk > 6 && (i % Nk == 4))
        temp = SubWord(temp);

    roundKey[i] = roundKey[i-Nk] ^ temp;
    i++;
}
```

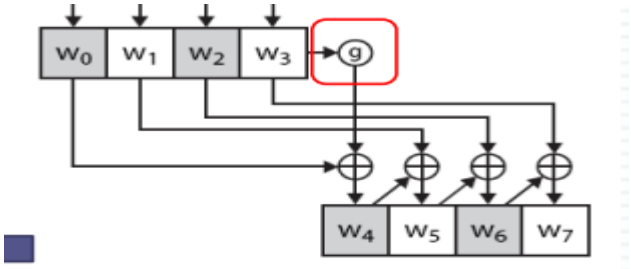
이전에 4개를 만들었으니 i = Nk로 초기화해준다. 그리고 Nb * (Nr + 1) == 44 만큼 반복문을 돌려준다. 바로 이전 w인 temp = roundKey[i-1]; 를 받아주고, 이 과정에서 i가 Nk로 나누어 떨어지는 부분은 g연산을 진행해준다.

$$g(w_3) = \text{S-Box}(\text{LRotWord}(w_3)) \oplus \text{RCon}_i$$

문서에 맞게 RotWord와 SubWord 함수를 이용해주고 Rcon값과 XOR 연산을 해준다.

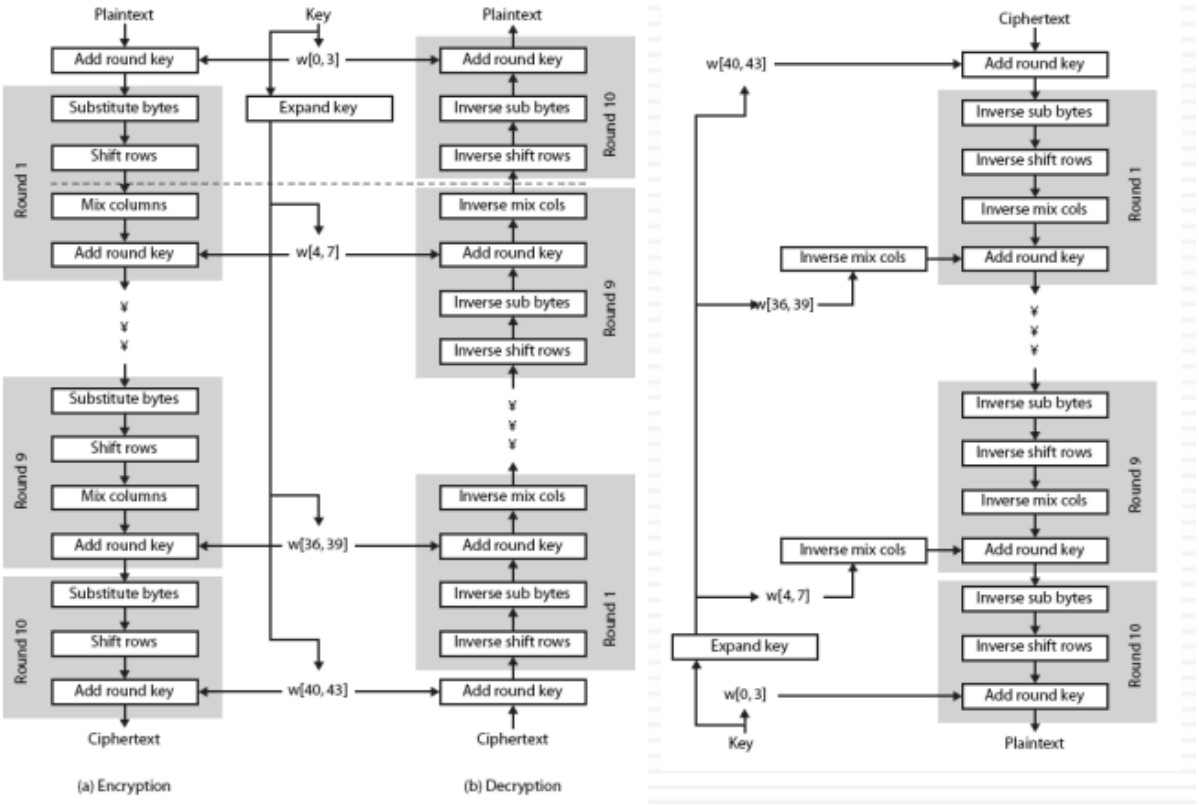
만약 key의 비트수가 늘어난다면 else if 문이 돌아간다.

그리고 Nk로 나누어 떨어지지 않는다면 Nk만큼 이전의 w와 temp를 XOR 해준다.

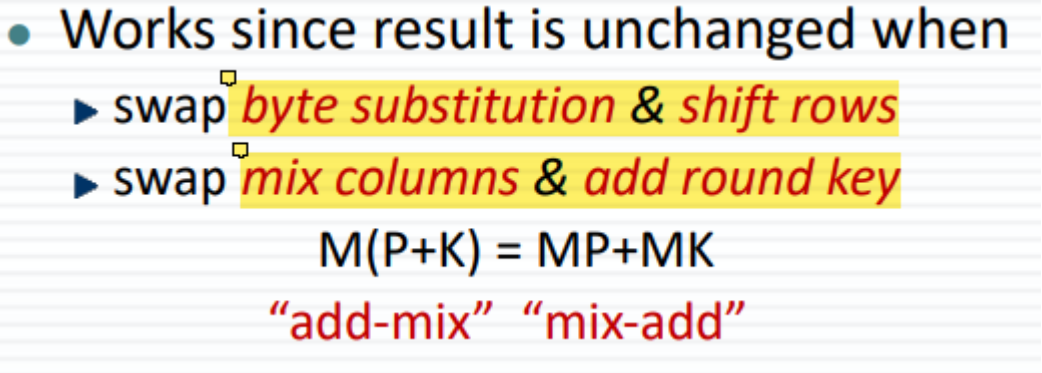


해당 그림을 보면 쉽게 이해가 간다.

이렇게 한다면 roundKey값을 만드는 건 끝난다. 근데 이때 추가적인 작업이 하나 들어갈 수 있는데,



위 그림을 보면 암호화와 복호화 과정은 비슷하지만 연산의 순서가 약간 다르다. 하지만



해당 구절을 이용하고 코드를 조금만 변형해주면 오른쪽과 같이 복호화지만, 암호화와 똑같은 연산 순서를 가지는 것이 가능하다.

이를 위해서 Inverse Mix Cols를 통해 특수한 roundKey를 하나 더 만들 것이다.

그 과정이 여기 있다.

```
for (int i = 0; i < Nb * (Nr+1); i++){
    droundKey[i] = roundKey[i];
}

for (int i = 1; i < Nr; i++){
    uint8_t tem[BLOCKLEN];
    memcpy(tem, droundKey + i*Nb, sizeof(uint8_t) * BLOCKLEN);
    MixColumns(tem, DECRYPT);
    memcpy(droundKey + i*Nb, tem, sizeof(uint8_t) * BLOCKLEN);
}
```

droundKey라는 배열을 만들어서 roundKey를 복사해준 다음, BLOCKLEN * 8비트정수사이즈 만큼 잘라서 Inverse Mix Cols을 해주는 것이다.

memcpy로 잘라서 Inverse Mix Cols를 진행해주고, 다시 memcpy 로droundKey 값을 바뀐 값으로 바꿔주는 연산이다. 이렇게 하면 droundKey의 모든 부분에 연산이 진행되어서 새로운 w가 만들어진다. 이제 복호화 과정에서는 roundKey대신 droundKey를 사용하고, 연산 순서를 암호화와 같이 해주면 된다.

이는 문서 상에서

```
EqInvCipher(byte in[4*Nb], byte out[4*Nb], word dw[Nb*(Nr+1)])
begin
   byte  state[4,Nb]

   state = in

   AddRoundKey(state, dw[Nr*Nb, (Nr+1)*Nb-1])

   for round = Nr-1 step -1 downto 1
      InvSubBytes(state)
      InvShiftRows(state)
      InvMixColumns(state)
      AddRoundKey(state, dw[round*Nb, (round+1)*Nb-1])
   end for

   InvSubBytes(state)
   InvShiftRows(state)
   AddRoundKey(state, dw[0, Nb-1])

   out = state
end


For the Equivalent Inverse Cipher, the following pseudo code is added at
the end of the Key Expansion routine (Sec. 5.2):
   for i = 0 step 1 to (Nr+1)*Nb-1
      dw[i] = w[i]
   end for

   for round = 1 step 1 to Nr-1
      InvMixColumns(dw[round*Nb, (round+1)*Nb-1])     //  note  change  of
type
   end for

Note that, since InvMixColumns operates on a two-dimensional array of bytes
while the Round Keys are held in an array of words, the call to
InvMixColumns in this code sequence involves a change of type (i.e. the
input to InvMixColumns() is normally the State array, which is considered
to be a two-dimensional array of bytes, whereas the input here is a Round
Key computed as a one-dimensional array of words).
```

Figure 15. Pseudo Code for the Equivalent Inverse Cipher.

방식을 이용한 것이다.

2. void Cipher(uint8_t *state, const uint32_t *roundKey, int mode);

```
121 void Cipher(uint8_t *state, const uint32_t *roundKey, int mode)
122 {
123     uint32_t temKey[Nb];
124 
125     if (mode == ENCRYPT){
126         memcpy(temKey, roundKey, sizeof(uint32_t) * Nb);
127         AddRoundKey(state, temKey);
128 
129         for (int i = 1; i < Nr; i++){
130             SubBytes(state, mode);
131             ShiftRows(state, mode);
132             MixColumns(state, mode);
133 
134             memcpy(temKey, roundKey + i*Nb, sizeof(uint32_t) * Nb);
135             AddRoundKey(state, temKey);
136         }
137 
138         SubBytes(state, mode);
139         ShiftRows(state, mode);
140 
141         memcpy(temKey, roundKey + Nr*Nb, sizeof(uint32_t) * Nb);
142         AddRoundKey(state, temKey);
143 
144     }
145     else if (mode == DECRYPT){
146         memcpy(temKey, droundKey + Nr*Nb, sizeof(uint32_t) * Nb);
147         AddRoundKey(state, temKey);
148 
149         for (int i = Nr - 1; i > 0; i--){
150             SubBytes(state, mode);
151             ShiftRows(state, mode);
152             MixColumns(state, mode);
153             memcpy(temKey, droundKey + i*Nb, sizeof(uint32_t) * Nb);
154             AddRoundKey(state, temKey);
155         }
156 
157         SubBytes(state, mode);
158         ShiftRows(state, mode);
159 
160         memcpy(temKey, droundKey, sizeof(uint32_t) * Nb);
161         AddRoundKey(state, temKey);
162 
163     }
164 }
165 
```

마지막으로 Cipher 함수를 보자. 암호화든, 복호화든 연산을 위한 roundKey와, 반복문의 진행 방향만 다를 뿐 연산의 순서는 동일하게 진행하였음을 볼 수 있다.

라운드에 들어가기 전

AddRoundKey

1 ~ 9라운드 까지는

SubBytes -> ShiftRows -> MixColumns -> AddRoundKey

10라운드는

SubBytes -> ShiftRows -> AddRoundKey

과정을 사용한다.

이 과정에서 w[0,3], w[4,7], ... w[40,43]을 이용하기 위해 uint32_t temKey[Nb];를 선언하여

roundKey의 특정 위치를 복사해준다. 첫 w[0,3]은 주소상 roundKey부터 Nb개의 32비트 정수를 복사, 나머지는 roundKey + 라운드 * Nb 부터 Nb개의 32비트 정수를 복사해준다.

그렇게 나온 temKey를 w로서 사용해서 AddRoundKey에 넣어준다.

물론 복호화는 droundKey를 사용해주어야 한다. 이렇게 구현을 완료했을 때 결과물을 확인해보자

#3 결과물

```
wollong@wollong-virtual-machine:~/proj#2$ ./test
<키>
0f 15 71 c9 47 d9 e8 59 0c b7 ad d6 af 7f 67 98
<라운드 키>
0f 15 71 c9 47 d9 e8 59 0c b7 ad d6 af 7f 67 98
dc 90 37 b0 9b 49 df e9 97 fe 72 3f 38 81 15 a7
d2 c9 6b b7 49 80 b4 5e de 7e c6 61 e6 ff d3 c6
c0 af df 39 89 2f 6b 67 57 51 ad 06 b1 ae 7e c0
2c 5c 65 f1 a5 73 0e 96 f2 22 a3 90 43 8c dd 50
58 9d 36 eb fd ee 38 7d 0f cc 9b ed 4c 40 46 bd
71 c7 4c c2 8c 29 74 bf 83 e5 ef 52 cf a5 a9 ef
37 14 93 48 bb 3d e7 f7 38 d8 08 a5 f7 7d a1 4a
48 26 45 20 f3 1b a2 d7 cb c3 aa 72 3c be 0b 38
fd 0d 42 cb 0e 16 e0 1c c5 d5 4a 6e f9 6b 41 56
b4 8e f3 52 ba 98 13 4e 7f 4d 59 20 86 26 18 76
---
<평문>
01 23 45 67 89 ab cd ef fe dc ba 98 76 54 32 10
<암호문>
ff 0b 84 4a 08 53 bf 7c 69 34 ab 43 64 14 8f b9
<복호문>
01 23 45 67 89 ab cd ef fe dc ba 98 76 54 32 10
<역암호문>
1f e0 22 1f 19 67 12 c4 be cd 5c 1c 60 71 ba a6
<복호문>
01 23 45 67 89 ab cd ef fe dc ba 98 76 54 32 10 .....PASSED
---
AES 성능시험.....................PASSED
CPU 사용시간 = 3.5440초
wollong@wollong-virtual-machine:~/proj#2$
```

예상 결과와 동일함을 알 수 있다! 그리고, 성능은 3.2초 ~ 4.1초 사이를 왔다 갔다 하였다.

평문, 암호문, 복호문 등등이 동일하게 나오는 것을 보아 구현이 문제 없이 된 것 같다.

#4 느낀점

개인적으로 과제는 이전 과제보다 어려웠다. 많은 비트연산들과 메모리 복사, 접근 등이 복잡하게 느껴졌지만 꼼수 없이 차근차근 문서와 강의자료를 살펴보니 해답이 나올 수 있었다. 특히나 구현이 어려웠던 것은

KeyExpansion과 MixColumns인데, 먼저 KeyExpansion이 어려웠던 이유는 내가 사용하고 있는 컴퓨터의 엔디안이 리틀 엔디안이라는 것을 간과하고 있었다는 것이다. 금방 깨달아서 연산의 순서나, 표현법을 다르게 했더니 금방 PASS를 받을 수 있었다. 그리고 MixColumns이 어려웠던 이유는 문서에서

$$
\begin{aligned}
\{57\} \bullet \{02\} &= xtime(\{57\}) = \{ae\} \\
\{57\} \bullet \{04\} &= xtime(\{ae\}) = \{47\} \\
\{57\} \bullet \{08\} &= xtime(\{47\}) = \{8e\} \\
\{57\} \bullet \{10\} &= xtime(\{8e\}) = \{07\},
\end{aligned}
$$

$$
\begin{aligned}
\{57\} \bullet \{13\} &= \{57\} \bullet (\{01\} \oplus \{02\} \oplus \{10\}) \\
&= \{57\} \oplus \{ae\} \oplus \{07\} \\
&= \{fe\}.
\end{aligned}
$$

이런 구절이 존재하였는데, 제대로 읽지 않아서 해당 기호가 XTIME을 하는 기호인줄 착각했기 때문이다. 생각해보면 02는 당연히 x를 곱해주는 연산이라서 다항식 곱이나, XTIME이나 똑같은데, 고민을 하고 있었다. 따라서 모든 연산을 다항식 곱을 통해 해결할 수 있었다.

최적화 부분에서 원래 처음에는 복호화의 연산 순서를 다르게 했었다. 즉 droundKey변형을 사용하지 않았었는데, 이때는 실행 속도가 4.2 ~ 4.9까지 나왔었다. 변형을 이용해 연산 순서를 같게 했더니 속도가 약 1초나 빨라진 것이다.

재밌었던 점은 이전의 과제와 연계되어 이용을 했다는 것이 기분이 좋았고, 디버깅을 해가며 하나하나씩 PASS를 받는 과정이 매우 재미있고 보람찼었다.