Crypto Engineering (GBX9SY03)

TP — Square attack on 31/2 rounds of AES

TACHTOUKT chayma

BELFKIRA Amine

Exercice 1: Warming up

Q.1

The function **xtime** makes the multiplication of a polynome P by X in the field $F_2[X]/X8 + X4 + X3 + X + 1$ such that :

- if the factor of X power 7 is 0, the multiplication is just P shifted by 1 bit to the right.
- if the factor of X power 7 is 1, after the multiplication we will have a factor of X power 8 so we need to compute the modulo of X8 then we add it to P shifted by 1 bit to the right.

a second variant of the function xtime using the polynome X8 + X6 + X5 + X4 + X3 + X + 1:

```
uint8_t xtime2(uint8_t p){
    uint8_t m = p >> 7;
    if(m&1){
    m = 0x7B;// representation in hexadecimal of X6+X5+X4+X3+X+1
    }
    return ((p << 1) ^ m);

}
```

Q.2 prev_aes128_round_key is simply the opposite steps of next_aes128_round_key, knowing that the inverse of a xor is a xor so:

```
void prev_aes128_round_key(const uint8_t next_key[16], uint8_t
prev_key[16], int round)
{
   int i;
   for (i = 4; i < 16; i++)
   {
       prev_key[i] = next_key[i] ^ next_key[i - 4];
```

```
    }
    prev_key[0] = next_key[0] ^ S[prev_key[13]] ^ RC[round -
1];
    prev_key[1] = next_key[1] ^ S[prev_key[14]];
    prev_key[2] = next_key[2] ^ S[prev_key[15]];
    prev_key[3] = next_key[3] ^ S[prev_key[12]];
}
```

Q.3 We need to take k1≠ k2 for F not to be trivial, because if so, the result will be zero; xor of two equal elements is 0. The function test implemented in /test/Exercice1.c shows that a block ciphered with the function F will be different.

Implementation:

F is keyedFunction:

```
void keyedFunction(uint8_t block[AES_BLOCK_SIZE], uint8_t
*key)
{
    uint8_t key1[AES_128_KEY_SIZE], key2[AES_128_KEY_SIZE];
    uint8_t block1[AES_BLOCK_SIZE], block2[AES_BLOCK_SIZE];
    for (int i = 0; i < AES_BLOCK_SIZE; i++)
    {
        key1[i] = key[i];
        block1[i] = block[i];
        block2[i] = block[i];
    }
    for (int i = 0; i < AES_BLOCK_SIZE; i++)
    {
        key1[i + AES_BLOCK_SIZE] = key[i + AES_BLOCK_SIZE];
    }
    aes128_enc(block1, key1, 3, 1);
    aes128_enc(block2, key2, 3, 1);
    for (int i = 0; i < AES_BLOCK_SIZE; i++)
    {
        block[i] = block1[i] ^ block2[i];
    }
}
```

Exercice 2: Key-recovery attack for 31/2-round AES

Q.1 In order to guess the key of the last round, we will guess the value of each byte separately and eliminate wrong guesses by partially decrypting and checking the 3-round distinguisher. For this, we will first generate a random key that we will be our target to find. Next, we will need to initialize a block that will contain the Λ-set. We will also initialize our table where we will store the candidates for each key byte. The attack consists in encrypting for 4 rounds a Λ-set, so that one first guesses the candidates for each key byte using the function ***porentielkEY(uint8_t **block, uint8_t **keyGuess).*** We are going to create a loop so that for each loop we eliminate bad guesses, generating a new Λ-set which will give us each time an indication of the validity of our guess. And to validate that we have only one candidate for each byte of the key we will look for false positives. Now that we have found the key extension, we can retrieve the original master key using the prev_aes128_round_key function which we will run 4 times as it is 4 round encryption. Our program tests the detection of 50 keys.

Q.2 We changed the representation as in (Exercise 1, Q.1), and we also changed the S-box for that we used a new function which has the same construction as the other function but with the new parameters.