Cryptography Homework 3

2024 Spring Semester

21 CST H3Art

Exercise 4.2 (optional)

Prove that decryption in a Feistel cipher can be done by applying the encryption algorithm to the ciphertext, with the key schedule reversed.

> **Solution**:
>
> Since Data Encryption Standard (DES) has the core idea about Feistel Cipher, the analysis of DES can be applied to any Feistel cipher.
>
> Therefore, we take DES as an example to prove decryption in Feistel Cipher can be done by applying the encryption algorithm to the ciphertext, DES encryption procedure has 16 rounds as follows:
>
> $$\begin{aligned} L_0R_0 &= \text{Initial Permutation}(x) \\ L_1 &= R_0 \\ R_1 &= L_0 \oplus f(R_0, K_1) \end{aligned}$$
>
> where x is the plaintext, $\mathrm{IP}(x)$ is divided into two halves of equal length, namely L and R, the round function g has the following form $g(L_{i-1}, R_{i-1}, K_i) = (L_i, R_i)$, where:
>
> $$\begin{aligned} L_i &= R_{i-1} \\ R_i &= L_{i-1} \oplus f(R_{i-1}, K_i) \end{aligned}$$
>
> After 16 iterations and invert permutation, we obtain the DES encrypted ciphertext y:
>
> $$\begin{aligned} L_{16} &= R_{15} \\ R_{16} &= L_{15} \oplus f(R_{15}, K_{16}) \\ y &= \text{Initial Permutation}^{-1}(R_{16}L_{16}) \end{aligned}$$
>
> Nextly, we need to decrypt the ciphertext y based on Feistel Cipher idea:
>
> $$\begin{aligned} L'_0R'_0 &= \text{Initial Permutation}(y) = R_{16}L_{16} \\ L'_1 &= R'_0 = L_{16} = R_{15} \\ R'_1 &= L'_0 \oplus f(R'_0, K_{16}) = R_{16} \oplus f(R_{15}, K_{16}) = L_{15} \\ &\vdots \\ L'_{16} &= R'_{15} = L_1 = R_0 \\ R'_{16} &= L'_{15} \oplus f(R'_{15}, K_1) = R_1 \oplus f(R_0, K_1) = L_0 \end{aligned}$$
>
> $$\text{Initial Permutation}^{-1}(R'_{16}L'_{16}) = \text{Initial Permutation}^{-1}(L_0R_0) = x$$
>
> Finally, in 16 rounds of DES we can prove decryption in a Feistel cipher can be done by applying the encryption algorithm to the ciphertext, with the key schedule reversed. If necessary, we can prove by induction that more rounds of Feistel cipher are also true.

Exercise 4.3

Let $DES(x, K)$ represent the encryption of plaintext x with key K using the DES cryptosystem. Suppose $y = DES(x, K)$ and $y' = DES(c(x), c(K))$, where $c(\cdot)$ denotes the bitwise complement of its argument. Prove that $y' = c(y)$ (i.e., if we complement the plaintext and the key, then the ciphertext is also complemented). Note that this can be proved using only the "high-level" description of DES—the actual structure of S-boxes and other components of the system are irrelevant.

Solution:

Suppose $y = DES(x, K)$ and $y' = DES(c(x), c(K))$, according to DES procedure we know that every stage of encryption is as follows:

$$\begin{aligned} L_{i+1} &= R_i \\ R_{i+1} &= L_i \oplus f(R_i, K_i) \end{aligned}$$

We omit the $\text{Initial Permutation}$, for $DES(L_0R_0, K)$, define $L'_0 = c(L_0)$, $R'_0 = c(R_0)$ and $K'_i = c(K_i)$, these definitions lead to $DES(L'_0R'_0, K')$, nextly we will prove that for any stage $L'_i = c(L_i)$ and $R'_i = c(R_i)$ by induction, namely we can prove $y' = c(y)$.

Base case, when $i = 1$:

For $DES(L_0R_0, K)$,

$$\begin{aligned} L_1 &= R_0 \\ R_1 &= L_0 \oplus f(R_0, K_0) \end{aligned}$$

For $DES(L'_0R'_0, K'_0)$,

$$\begin{aligned} L'_1 &= R'_0 \\ R'_1 &= L'_0 \oplus f(R'_0, K'_0) \\ &= c(L_0) \oplus f(c(R_0), c(K_0)) \end{aligned}$$

Since $f(R_i, K_i)$ uses the $\oplus$ operation to combine input bits of R_i (after expansion) and K_i before the permutation in S-boxes, and $\oplus$ is associative and communtantive,

$$c(r) \oplus c(k) = r \oplus k$$

therefore, $R'_1 = c(L_0 \oplus f(R_0, K_0)) = c(R_1)$

Induction step, when $i > 1$:

For $DES(L_0R_0, K)$,

$$\begin{aligned} L_n &= R_{n-1} \\ R_n &= L_{n-1} \oplus f(R_{n-1}, K_{n-1}) \end{aligned}$$

For $DES(L'_0R'_0, K)$,

$$\begin{aligned} L'_n &= R'_{n-1} = c(R_{n-1}) \\ R'_n &= L'_{n-1} \oplus f(R'_{n-1}, K'_{n-1}) \\ &= c(L_{n-1}) \oplus f(c(R_{n-1}), c(K_{n-1})) \\ &= c(L_{n-1} \oplus f(R_{n-1}, K_{n-1})) \end{aligned}$$

Therefore, after 16 stages of Feistel Cipher in DES, we can finally get $L'_{16} = c(L_{16})$ and $R'_{16} = c(R_{16})$, we concatenate L'_{16} and R'_{16}, can obtain:

$$y' = L'_{16}R'_{16} = c(L_{16}R_{16}) = c(y)$$

Exercise 4.4 (optional)

Suppose that we have the following 128-bit AES key, given in hexadecimal notation:

$$2B7E151628AED2A6ABF7158809CF4F3C$$

Construct the complete key schedule arising from this key.

Solution:

To expand a given 128-bit initial key to derive the full-length key for AES encryption, the following steps need to be performed:

- **Initial Key Setup**: The initial 128-bit key is used as the encryption key for the first round.
- **Structure of Key Expansion**: For a 128-bit key, there are a total of 10 rounds, and thus, 10 round keys are needed.
- **Concept of Words**: In AES key expansion, the key is divided into several 32-bit units called "words". For a 128-bit key, there are initially 4 words.
- **Key Expansion Algorithm**:
  - **RotWord Operation**: This involves a simple byte rotation within a word (e.g., $[a0, a1, a2, a3]$ becomes $[a1, a2, a3, a0]$).
  - **SubWord Operation**: This operation applies an S-box substitution to each byte of the word after the RotWord operation.
  - **Rcon Operation**: During the generation of the first word of each new 128-bit block of the key, a round constant (Rcon) is used. This is a predefined series of values in AES, used to add complexity through an XOR operation.
- **Generating Additional Words**: New key words are generated through the operations mentioned above and by XORing with the previous complete key block. This process is repeated until enough round keys are generated. For a 128-bit key, a total of 44 words are needed (initial 4 words plus 40 additional words for the 10 rounds).

Here is the executable Python code for arising the 128-bit AES key:

```python
def sub_word(word: int):
    """
    Applies an S-box substitution on each byte of the input 32-bit word
    """
    sbox = [
        0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5, 0x30, 0x01, 0x67, 0x2B, 0xFE, 0xD7, 0xAB, 0x76,
        0xCA, 0x82, 0xC9, 0x7D, 0xFA, 0x59, 0x47, 0xF0, 0xAD, 0xD4, 0xA2, 0xAF, 0x9C, 0xA4, 0x72, 0xC0,
        0xB7, 0xFD, 0x93, 0x26, 0x36, 0x3F, 0xF7, 0xCC, 0x34, 0xA5, 0xE5, 0xF1, 0x71, 0xD8, 0x31, 0x15,
        0x04, 0xC7, 0x23, 0xC3, 0x18, 0x96, 0x05, 0x9A, 0x07, 0x12, 0x80, 0xE2, 0xEB, 0x27, 0xB2, 0x75,
        0x09, 0x83, 0x2C, 0x1A, 0x1B, 0x6E, 0x5A, 0xA0, 0x52, 0x3B, 0xD6, 0xB3, 0x29, 0xE3, 0x2F, 0x84,
        0x53, 0xD1, 0x00, 0xED, 0x20, 0xFC, 0xB1, 0x5B, 0x6A, 0xCB, 0xBE, 0x39, 0x4A, 0x4C, 0x58, 0xCF,
        0xD0, 0xEF, 0xAA, 0xFB, 0x43, 0x4D, 0x33, 0x85, 0x45, 0xF9, 0x02, 0x7F, 0x50, 0x3C, 0x9F, 0xA8,
        0x51, 0xA3, 0x40, 0x8F, 0x92, 0x9D, 0x38, 0xF5, 0xBC, 0xB6, 0xDA, 0x21, 0x10, 0xFF, 0xF3, 0xD2,
        0xCD, 0x0C, 0x13, 0xEC, 0x5F, 0x97, 0x44, 0x17, 0xC4, 0xA7, 0x7E, 0x3D, 0x64, 0x5D, 0x19, 0x73,
        0x60, 0x81, 0x4F, 0xDC, 0x22, 0x2A, 0x90, 0x88, 0x46, 0xEE, 0xB8, 0x14, 0xDE, 0x5E, 0x0B, 0xDB,
        0xE0, 0x32, 0x3A, 0x0A, 0x49, 0x06, 0x24, 0x5C, 0xC2, 0xD3, 0xAC, 0x62, 0x91, 0x95, 0xE4, 0x79,
        0xE7, 0xC8, 0x37, 0x6D, 0x8D, 0xD5, 0x4E, 0xA9, 0x6C, 0x56, 0xF4, 0xEA, 0x65, 0x7A, 0xAE, 0x08,
        0xBA, 0x78, 0x25, 0x2E, 0x1C, 0xA6, 0xB4, 0xC6, 0xE8, 0xDD, 0x74, 0x1F, 0x4B, 0xBD, 0x8B, 0x8A,
        0x70, 0x3E, 0xB5, 0x66, 0x48, 0x03, 0xF6, 0x0E, 0x61, 0x35, 0x57, 0xB9, 0x86, 0xC1, 0x1D, 0x9E,
        0xE1, 0xF8, 0x98, 0x11, 0x69, 0xD9, 0x8E, 0x94, 0x9B, 0x1E, 0x87, 0xE9, 0xCE, 0x55, 0x28, 0xDF,
        0x8C, 0xA1, 0x89, 0x0D, 0xBF, 0xE6, 0x42, 0x68, 0x41, 0x99, 0x2D, 0x0F, 0xB0, 0x54, 0xBB, 0x16
    ]

    return (
        (sbox[(word >> 24) & 0xFF] << 24) |
        (sbox[(word >> 16) & 0xFF] << 16) |
        (sbox[(word >> 8) & 0xFF] << 8) |
        (sbox[word & 0xFF])
    )


def rot_word(word: int):
    """
    Performs a left rotation of 8 bits on the 32-bit word
    """
    return ((word << 8) & 0xFFFFFFFF) | (word >> 24)
```

```python
def key_expansion(key: str):
    """
    Expands and generates a list of key words from the initial key
    """
    Nk = 4  # Number of 32-bit words in the key
    Nr = 10  # Number of rounds
    Nb = 4  # Number of columns in the state

    key_words = [0] * (Nb * (Nr + 1))
    initial_key = bytes.fromhex(key)

    # Loading key into the first 4 words
    for i in range(Nk):
        key_words[i] = int.from_bytes(initial_key[4*i:4*i+4], 'big')

    # Variables for the algorithm
    rcon = [
        0x01000000, 0x02000000, 0x04000000, 0x08000000, 0x10000000,
        0x20000000, 0x40000000, 0x80000000, 0x1b000000, 0x36000000
    ]

    # Expanding the keys
    for i in range(Nk, Nb * (Nr + 1)):
        temp = key_words[i - 1]
        if i % Nk == 0:
            temp = sub_word(rot_word(temp)) ^ rcon[i // Nk - 1]
        key_words[i] = key_words[i - Nk] ^ temp

    return key_words

if __name__ == '__main__':
    key = "2B7E151628AED2A6ABF7158809CF4F3C"
    expanded_keys = key_expansion(key)
    for index, word in enumerate(expanded_keys):
        print(f"The {index+1} word in expanded key is: {hex(word)}")
```

The results of executing the above code are as follows:

```
The 1 word in expanded key is: 0x2b7e1516
The 2 word in expanded key is: 0x28aed2a6
The 3 word in expanded key is: 0xabf71588
The 4 word in expanded key is: 0x9cf4f3c
The 5 word in expanded key is: 0xa0fafe17
The 6 word in expanded key is: 0x88542cb1
The 7 word in expanded key is: 0x23a33939
The 8 word in expanded key is: 0x2a6c7605
The 9 word in expanded key is: 0xf2c295f2
The 10 word in expanded key is: 0x7a96b943
The 11 word in expanded key is: 0x5935807a
The 12 word in expanded key is: 0x7359f67f
The 13 word in expanded key is: 0x3d80477d
The 14 word in expanded key is: 0x4716fe3e
The 15 word in expanded key is: 0x1e237e44
The 16 word in expanded key is: 0x6d7a883b
The 17 word in expanded key is: 0xef44a541
The 18 word in expanded key is: 0xa8525b7f
The 19 word in expanded key is: 0xb671253b
The 20 word in expanded key is: 0xdb0bad00
The 21 word in expanded key is: 0xd4d1c6f8
The 22 word in expanded key is: 0x7c839d87
The 23 word in expanded key is: 0xcaf2b8bc
The 24 word in expanded key is: 0x11f915bc
The 25 word in expanded key is: 0x6d88a37a
The 26 word in expanded key is: 0x110b3efd
The 27 word in expanded key is: 0xdbf98641
The 28 word in expanded key is: 0xca0093fd
The 29 word in expanded key is: 0x4e54f70e
The 30 word in expanded key is: 0x5f5fc9f3
The 31 word in expanded key is: 0x84a64fb2
The 32 word in expanded key is: 0x4ea6dc4f
The 33 word in expanded key is: 0xead27321
The 34 word in expanded key is: 0xb58dbad2
The 35 word in expanded key is: 0x312bf560
The 36 word in expanded key is: 0x7f8d292f
The 37 word in expanded key is: 0xac7766f3
The 38 word in expanded key is: 0x19fadc21
The 39 word in expanded key is: 0x28d12941
The 40 word in expanded key is: 0x575c006e
The 41 word in expanded key is: 0xd014f9a8
The 42 word in expanded key is: 0xc9ee2589
The 43 word in expanded key is: 0xe13f0cc8
The 44 word in expanded key is: 0xb6630ca6
```

Exercise 4.6 (for the case of CBC mode)

Prove that decryption in CBC mode or CFB mode can be parallelized efficiently. More precisely, suppose we have n ciphertext blocks and n processors. Show that it is possible to decrypt all n ciphertext blocks in constant time.

Solution:

Suppose we are given n ciphertext blocks $y_1, \ldots, y_n$, and we have n processors $P_1, \ldots, P_n$.

For CBC mode, in the first step, each P_i decrypts y_i, obtaining the intermediate state obtained by XORing the plaintext block with the initial vector or the previous ciphertext block, we denote it as z_i.

In the next step, each P_i computes $z_i \oplus y_i$ can finally get the plaintext blocks x_i.

Exercise 4.8 (for the case of CFB mode)

Suppose that $X = (x_1, \ldots, x_n)$ and $X' = (x'_1, \ldots, x'_n)$ are two sequences of n plaintext blocks. Define

$$\mathbf{same}(X, X') = \max\{j : x_i = x'_i \text{ for all } i \leq j\}$$

Suppose X and X' are encrypted in CBC or CFB mode using the same key and the same IV. Show that it is easy for an adversary to compute $\mathbf{same}(X, X')$.

Solution:

In CFB mode, consider two ciphertexts, $Y = (y_1, \ldots, y_n)$ and $Y' = (y'_1, \ldots, y'_n)$, corresponding to plaintexts X and X'. Assume that $x_i = x'_i$ for $1 \leq i \leq j$, but $x_{j+1} \neq x'_{j+1}$. This implies that $y_i = y'_i$ for all $1 \leq i \leq j$, and $y_{j+1} \neq$

y'_{j+1}. Therefore, the adversary can determine:

$$\mathbf{same}(X, X') = \max\{j : y_i = y'_i \text{ for all } i \le j\}$$

Exercise 4.9 (for the case of OFB mode)

Suppose that $X = (x_1, \ldots, x_n)$ and $X' = (x_1, \ldots, x_n)$ are two sequences of n plaintext blocks. Suppose X and X' are encrypted in OFB mode using the same key and the same IV. Show that it is easy for an adversary to compute $X \oplus X'$ .Show that a similar result holds for CTR mode if ctr is reused.

Solution:

Consider Y and Y' as the ciphertexts of X and X', respectively, both encrypted using the same keystream. In encryption, the plaintexts are combined with the keystream using the XOR operation to produce the ciphertexts. Therefore, the result of XOR-ed Y and Y' directly gives the XOR of X and X', expressed as:

$$\begin{aligned} Y \oplus Y' &= (X \oplus K) \oplus (X' \oplus K) \\ &= X \oplus X' \oplus (K \oplus K) \\ &= X \oplus X' \oplus 0 \\ &= X \oplus X' \end{aligned}$$