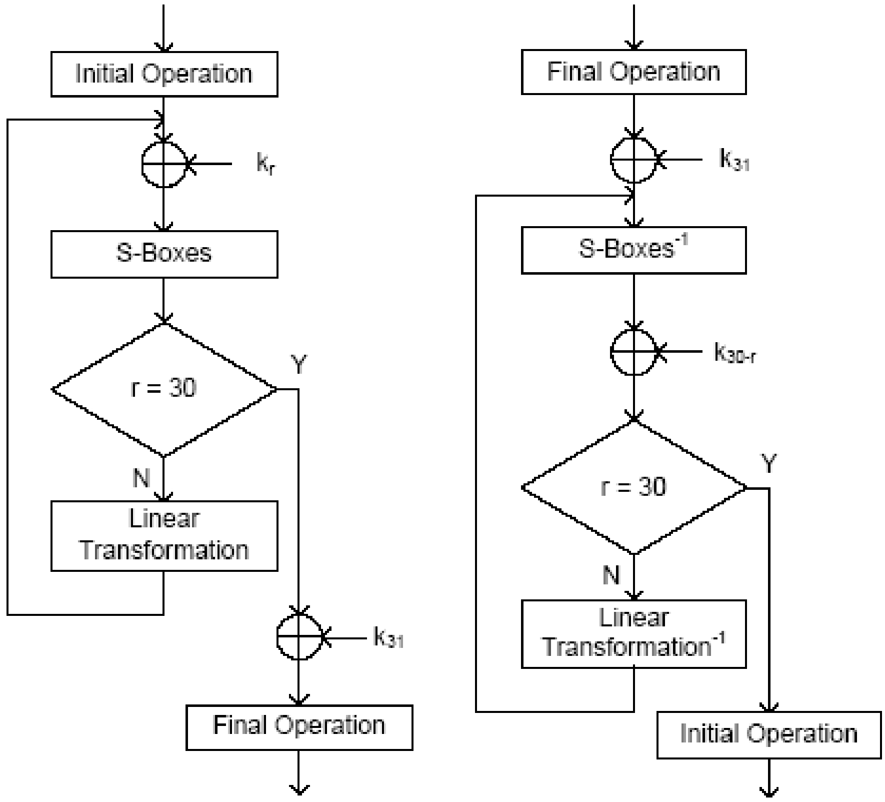
密碼學期末書面報告

題目 ： 實作對稱式加密演算法 Serpent

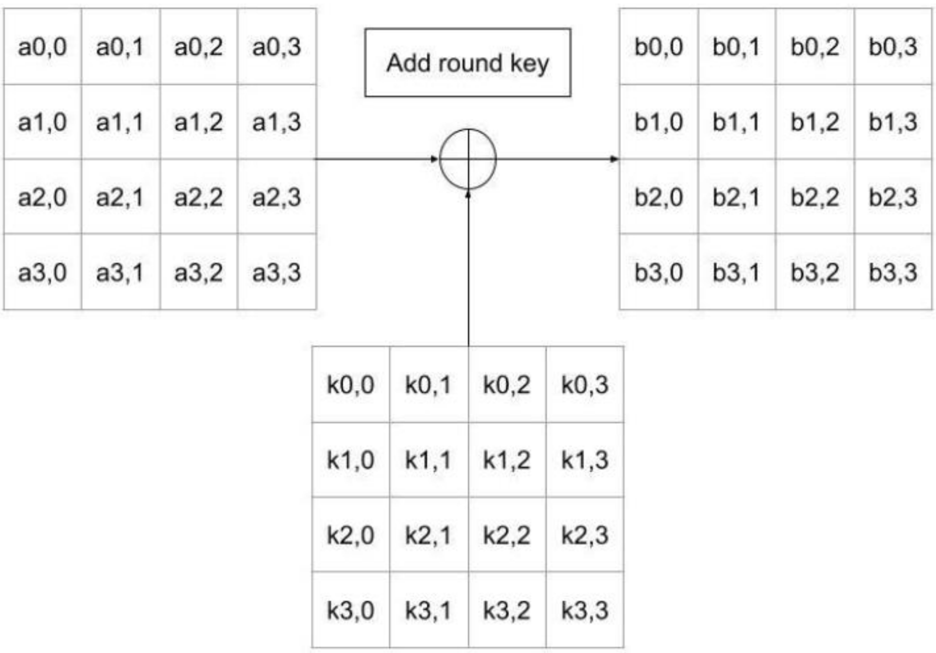
組員 ： 劉俊廷, 鄭佩綺, 林庭毅, 陳建維

設計原理 ：

Serpent 是一個對稱式加密演算法，是進階加密標準(AES)的候選者之一，其被選擇的順序僅次於 Rijndael 演算法，根據這篇論文“Serpent: A Proposal for the Advanced Encryption Standard”，其流程圖如圖(1)，首先，明文會被分成多個 128 bits 的區塊，接下來再對每個區塊逐個做加密，不論使用的金鑰長度是 128 bits、192 bits、256 bits，Serpent 都會進行 32 回合的運算，每個回合包含三個步驟：Add round key、S-boxes substitution、Linear transformation，解密的每個回合同樣包含三個步驟：Add round key、Inverse s-boxes substitution、Inverse linear transformation，順序為解密流程相對應的反運算。由於 Add round key 是區塊和 round key 做 XOR 運算，所以 Add round key 的反運算即為本身，如圖 2 所示；至於 S-boxes substitution，Serpent 採用 8 個不同的 s-box，如圖 3(a)，依據當前執行的回合數對 8 取餘數以決定使用哪個 s-box，Inverse s-boxes substitution 則是要使用 Inverse s-boxes 來做替換，如圖 3(b)，替換的進行方式如圖 4，先將區塊分為四個 word，接著一次取四個 word 的 1 bit 組合出一個 16 進位數字，然後依據目前回合使用的 s-box 找到對應的另一個 16 進位數字，最後再把四個 bit 放回對應的 word。



(a) Encryption (b) Decryption



圖(2)、Add round key 示意圖

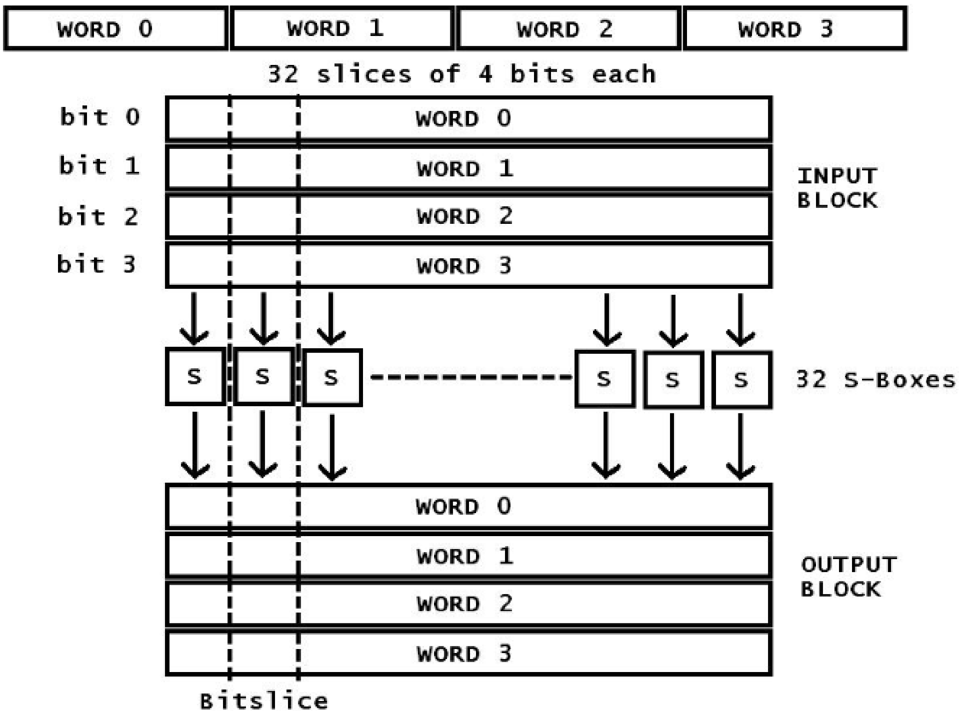
S0:	3	8	15	1	10	6	5	11	14	13	4	2	7	0	9	12
S1:	15	12	2	7	9	0	5	10	1	11	14	8	6	13	3	4
S2:	8	6	7	9	3	12	10	15	13	1	14	4	0	11	5	2
S3:	0	15	11	8	12	9	6	3	13	1	2	4	10	7	5	14
S4:	1	15	8	3	12	0	11	6	2	5	4	10	9	14	7	13
S5:	15	5	2	11	4	10	9	12	0	3	14	8	13	6	7	1
S6:	7	2	12	5	8	4	6	11	14	9	1	15	13	3	10	0
S7:	1	13	15	0	14	8	2	11	7	4	12	10	9	3	5	6

圖 3(a)、s-box 內容

InvS0:	13	3	11	0	10	6	5	12	1	14	4	7	15	9	8	2
InvS1:	5	8	2	14	15	6	12	3	11	4	7	9	1	13	10	0
InvS2:	12	9	15	4	11	14	1	2	0	3	6	13	5	8	10	7
InvS3:	0	9	10	7	11	14	6	13	3	5	12	2	4	8	15	1
InvS4:	5	0	8	3	10	9	7	14	2	12	11	6	4	15	13	1
InvS5:	8	15	2	9	4	1	13	14	11	6	5	3	7	12	10	0
InvS6:	15	10	1	13	5	3	6	0	4	9	14	7	2	12	8	11
InvS7:	3	0	6	13	9	14	15	8	5	12	11	7	10	1	4	2

圖 3(b)、inverse s-box 內容

而 Linear transformation 則是對區塊進行一系列的位移，如圖(5)，其中<<<代表逆時鐘 方向位移，Inverse linear transformation 則是將原本的 Linear transformation 以完全相反 的順序執行，並將逆時鐘方向位移改為順時鐘方向。



圖(4)、S-boxes substitution 示意圖

$$
\begin{aligned}
X_0, X_1, X_2, X_3 &:= S_i(B_i \oplus K_i) \\
X_0 &:= X_0 <<< 13 \\
X_2 &:= X_2 <<< 3 \\
X_1 &:= X_1 \oplus X_0 \oplus X_2 \\
X_3 &:= X_3 \oplus X_2 \oplus (X_0 << 3) \\
X_1 &:= X_1 <<< 1
\end{aligned}
\qquad
\begin{aligned}
X_3 &:= X_3 <<< 7 \\
X_0 &:= X_0 \oplus X_1 \oplus X_3 \\
X_2 &:= X_2 \oplus X_3 \oplus (X_1 << 7) \\
X_0 &:= X_0 <<< 5 \\
X_2 &:= X_2 <<< 22 \\
B_{i+1} &:= X_0, X_1, X_2, X_3
\end{aligned}
$$

圖(5)、Linear transformation 進行之運算

函式實作

加密 ：

1. key scheduling

若金鑰長度不滿 256 bits，補 0

將金鑰分割成 8 個 word(32 bits)，以下式計算出 prekey

$$w_i := (w_{i-8} \oplus w_{i-5} \oplus w_{i-3} \oplus w_{i-1} \oplus \Phi \oplus i) <<< 11$$

$\Phi = 0x9e3779b9$、 <<<表示旋轉

prekey 放入 S 盒替換成 word k，然後依序取出成 33 個 subkey K

$$
\begin{aligned}
\{k_0, k_1, k_2, k_3\} &:= S_3(w_0, w_1, w_2, w_3) \\
\{k_4, k_5, k_6, k_7\} &:= S_2(w_4, w_5, w_6, w_7) \\
\{k_8, k_9, k_{10}, k_{11}\} &:= S_1(w_8, w_9, w_{10}, w_{11}) \\
\{k_{12}, k_{13}, k_{14}, k_{15}\} &:= S_0(w_{12}, w_{13}, w_{14}, w_{15}) \\
\{k_{16}, k_{17}, k_{18}, k_{19}\} &:= S_7(w_{16}, w_{17}, w_{18}, w_{19}) \\
&\dots \\
\{k_{124}, k_{125}, k_{126}, k_{127}\} &:= S_4(w_{124}, w_{125}, w_{126}, w_{127}) \\
\{k_{128}, k_{129}, k_{130}, k_{131}\} &:= S_3(w_{128}, w_{129}, w_{130}, w_{131})
\end{aligned}
$$

$$K_i := \{k_{4i}, k_{4i+1}, k_{4i+2}, k_{4i+3}\}$$

```
for (i = 0; i < Nb * PNr; ++i) {
    if(i < Nk){
        s->words[i] = (k[Nb*i]<<24) | (k[Nb*i+1]<<16) |
                (k[Nb*i+2]<<8) | (k[Nb*i+3]);
    }else if(i < 8){
        s->words[i] = 0x0000;
    }else{
        s->words[i] = rotword(s->words[i-8] ^ s->words[i-5] ^ s->words[i-3] ^ s->words[i-1] ^ FRAC ^ i, 11);
    }
```

```
if(i % 4 == 3){
    which_sbox--;
    if(which_sbox < 0)
        which_sbox += 8;

    temp[0] = s->words[i-3];
    temp[1] = s->words[i-2];
    temp[2] = s->words[i-1];
    temp[3] = s->words[i];
    s->words[i-3] = 0x0000;
    s->words[i-2] = 0x0000;
    s->words[i-1] = 0x0000;
    s->words[i] = 0x0000;
    for(j=0;j<32;j++){
        sub_temp = sub_4bit(which_sbox, (temp[3]&0x01) << 3 | (temp[2]&0x01) << 2 |
            (temp[1]&0x01) << 1 | (temp[0]&0x01));
        s->words[i-3] |= (sub_temp&0x01) << j;
        s->words[i-2] |= (sub_temp&0x02) << j;
        s->words[i-1] |= (sub_temp&0x04) << j;
        s->words[i] |= (sub_temp&0x08) << j;
        temp[0] >> 1;
        temp[1] >> 1;
        temp[2] >> 1;
        temp[3] >> 1;
    }
}
```

2. Initial Permutation(IP)

將初始的明文做排列

```
static inline void initial_permutation(uint8_t *s)
{
    short i;
    for(i = 0;i < 128; ++i){
        uint8_t b_position = (i*32%127);
        uint8_t bit_a = (1 << (i%8)) & s[i/8];
        uint8_t bit_b = (1 << (b_position%8)) & s[b_position/8];
        if(bit_a > 0 && bit_b == 0){
            s[i/8] -= bit_a;
            s[b_position/8] += (1 << (b_position%8));
        }
        else if(bit_a == 0 && bit_b > 0){
            s[i/8] += (1 << (i%8));
            s[b_position/8] -= bit_b;
        }
    }
}
```

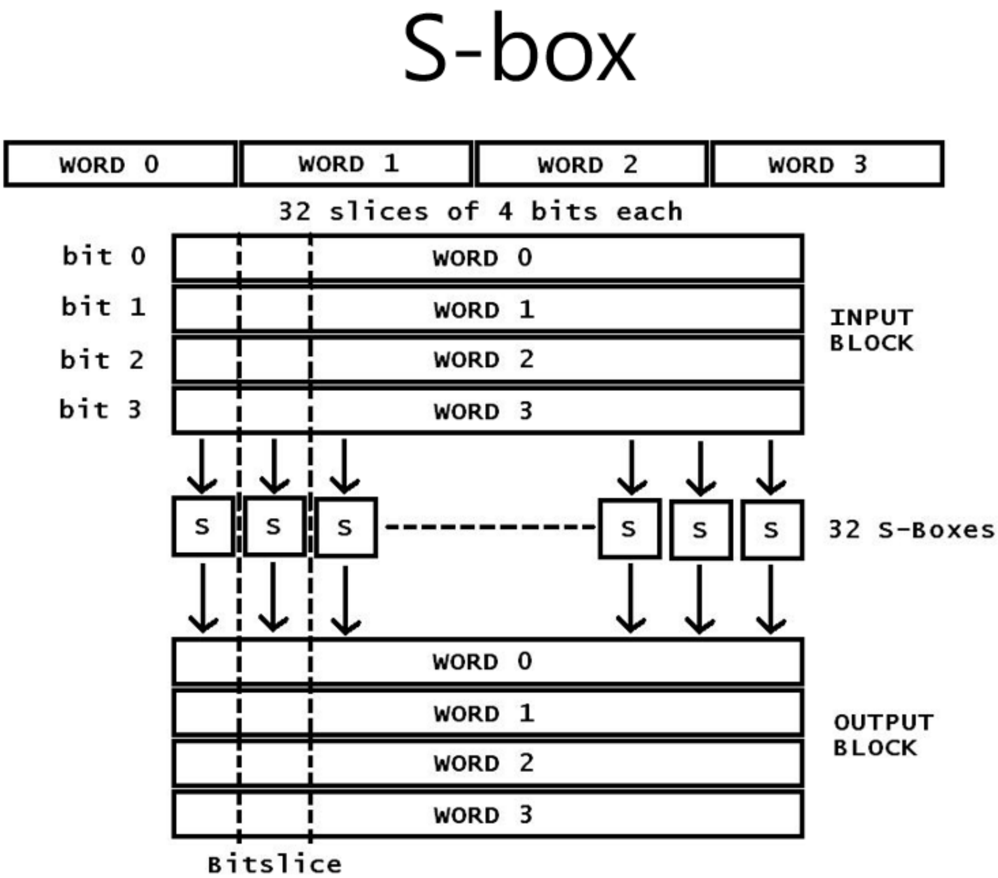
3. Add round key

將步驟一所生成出的回合金鑰(round key)和明文做 XOR

```
static inline void add_round_key(uint8_t *s, const unsigned int *k)
{
    s[0]  ^= (uint8_t)(k[0] >> 24); s[1]  ^= (uint8_t)(k[0] >> 16);
    s[2]  ^= (uint8_t)(k[0] >>  8); s[3]  ^= (uint8_t)(k[0]);
    s[4]  ^= (uint8_t)(k[1] >> 24); s[5]  ^= (uint8_t)(k[1] >> 16);
    s[6]  ^= (uint8_t)(k[1] >>  8); s[7]  ^= (uint8_t)(k[1]);
    s[8]  ^= (uint8_t)(k[2] >> 24); s[9]  ^= (uint8_t)(k[2] >> 16);
    s[10] ^= (uint8_t)(k[2] >>  8); s[11] ^= (uint8_t)(k[2]);
    s[12] ^= (uint8_t)(k[3] >> 24); s[13] ^= (uint8_t)(k[3] >> 16);
    s[14] ^= (uint8_t)(k[3] >>  8); s[15] ^= (uint8_t)(k[3]);
}
```

4. S-BOX 替換

將已經和回合金鑰 XOR 後的資料進入 S-BOX 置換



以下是加密所使用的 S-BOX

```
static const uint8_t sbox[8][16]={
    { 0x03, 0x08, 0x0f, 0x01, 0x0a, 0x06, 0x05, 0x0b, 0x0e, 0x0d, 0x04, 0x02, 0x07, 0x00, 0x09, 0x0c},
    { 0x0f, 0x0c, 0x02, 0x07, 0x09, 0x00, 0x05, 0x0a, 0x01, 0x0b, 0x0e, 0x08, 0x06, 0x0d, 0x03, 0x04},
    { 0x08, 0x06, 0x07, 0x09, 0x03, 0x0c, 0x0a, 0x0f, 0x0d, 0x01, 0x0e, 0x04, 0x00, 0x0b, 0x05, 0x02},
    { 0x00, 0x0f, 0x0b, 0x08, 0x0c, 0x09, 0x06, 0x03, 0x0d, 0x01, 0x02, 0x04, 0x0a, 0x07, 0x05, 0x0e},
    { 0x01, 0x0f, 0x08, 0x03, 0x0c, 0x00, 0x0b, 0x06, 0x02, 0x05, 0x04, 0x0a, 0x09, 0x0e, 0x07, 0x0d},
    { 0x0f, 0x05, 0x02, 0x0b, 0x04, 0x0a, 0x09, 0x0c, 0x00, 0x03, 0x0e, 0x08, 0x0d, 0x06, 0x07, 0x01},
    { 0x07, 0x02, 0x0c, 0x05, 0x08, 0x04, 0x06, 0x0b, 0x0e, 0x09, 0x01, 0x0f, 0x0d, 0x03, 0x0a, 0x00},
    { 0x01, 0x0d, 0x0f, 0x00, 0x0e, 0x08, 0x02, 0x0b, 0x07, 0x04, 0x0c, 0x0a, 0x09, 0x03, 0x05, 0x06},
};
```

5. Linear Transformation

經過 S-BOX 置換後的資料進行線性轉換，進而增加加密後的安全性

```
static inline void linear_transformation(uint8_t *s, unsigned int i, uint8_t shift_key)
{
    unsigned int X[4];
    for (i = 0; i < Nb; ++i) {
        X[i] = (s[Nb*i]<<24) | (s[Nb*i+1]<<16) | (s[Nb*i+2]<<8) | (s[Nb*i+3]);
    }

    X[0]  = rotword(X[0], shift_option[shift_key][0]);
    X[2]  = rotword(X[2], shift_option[shift_key][1]);
    X[1] ^= X[0] ^ X[2];
    X[3] ^= X[2] ^ (X[0] << 3);
    X[1]  = rotword(X[1], shift_option[shift_key][2]);
    X[3]  = rotword(X[3], shift_option[shift_key][3]);
    X[0] ^= X[1] ^ X[3];
    X[2] ^= X[3] ^ (X[1] << 7);
    X[0]  = rotword(X[0], 5);
    X[2]  = rotword(X[2], 22);

    for (i = 0; i < Nb; ++i) {
        s[Nb*i+3] = (uint8_t)(X[i] & 0xff);
        s[Nb*i+2] = (uint8_t)(X[i] >> 8 & 0xff);
        s[Nb*i+1] = (uint8_t)(X[i] >> 16 & 0xff);
        s[Nb*i] = (uint8_t)(X[i] >> 24 & 0xff);
    }
}
```

6. Final Permutation

上述的步驟若以進行 32 回合的重複加密後，最後再將資料進行最終排列

```c
static inline void final_permutation(uint8_t *s)
{
    short i;
    for(i = 0;i < 128; ++i){
        uint8_t b_position = (i*2%127);
        uint8_t bit_a = (1 << (i%8)) & s[i/8];
        uint8_t bit_b = (1 << (b_position%8)) & s[b_position/8];
        if(bit_a > 0 && bit_b == 0){
            s[i/8] -= bit_a;
            s[b_position/8] += (1 << (b_position%8));
        }
        else if(bit_a == 0 && bit_b > 0){
            s[i/8] += (1 << (i%8));
            s[b_position/8] -= bit_b;
        }
    }
}
```

解密 ：

解密過程和加密有些類似，差別在於反方向的計算，先將密文進行反最終排列，可得未經由 Final Permutation 的密文，如函式 inv_final_permutation()

```c
void inv_final_permutation(uint8_t *s) {
    short i;
    for (i = 127; i >= 0; --i) {
        uint8_t b_position = ((i << 1) % 127);
        uint8_t bit_a = (1 << (i & 7)) & s[i >> 3];
        uint8_t bit_b = (1 << (b_position & 7)) & s[b_position >> 3];

        if (bit_a > 0 && bit_b == 0) {
            s[i >> 3] -= bit_a;
            s[b_position >> 3] += (1 << (b_position & 7));
        }
        else if (bit_a == 0 && bit_b > 0) {
            s[i >> 3] += (1 << (i & 7));
            s[b_position >> 3] -= bit_b;
        }
    }
}
```

而由於加密的最後一回合有跟金鑰做 XOR，所以解密同樣需要先跟回合金鑰 XOR 再進行 inverse_SBOX、回合金鑰 XOR、 inverse_linear_transformation 等操作，並執行 32 回合，最後再 inverse_initial_permutation，就能解出原本的明文。

```c
void inv_sub_bytes(uint8_t *s, unsigned int round) {
    unsigned int i, j, sub_temp;
    unsigned int temp[4];
    for (i = 0; i < Nb; i++) {
        temp[i] = (s[Nb * i] << 24) | (s[Nb * i + 1] << 16) | (s[Nb * i + 2] << 8) | (s[Nb * i + 3]);
        s[Nb * i] = 0x00;
        s[Nb * i + 1] = 0x00;
        s[Nb * i + 2] = 0x00;
        s[Nb * i + 3] = 0x00;
    }

    for (j = 0; j < 32; j++) {
        sub_temp = inv_sub_4bit(round % 8, (temp[3] & 0x80000000) >> 28 |
                                            (temp[2] & 0x80000000) >> 29 | (temp[1] & 0x80000000) >> 30 | (temp[0] & 0x80000000) >> 31);

        // printf("before sub %x %x %x %x\n", (temp[3]&0x80000000) >> 28 , (temp[2]&0x80000000) >> 29 , (temp[1]&0x80000000) >> 30 , (temp[0]&0x80000000) >> 31);
        // printf("sub_temp %x\n", sub_temp);

        int f;
        for (f = 0; f < 4; ++f) {
            s[(j >> 3) + (f << 2)] |= (sub_temp & 0x01) << (7 - (j & 7));
            sub_temp >>= 1;
        }
        temp[0] <<= 1;
        temp[1] <<= 1;
        temp[2] <<= 1;
        temp[3] <<= 1;
    }
}

void add_round_key(uint8_t *s, const unsigned int *k) {
    s[0] ^= (uint8_t)(k[0] >> 24);
    s[1] ^= (uint8_t)(k[0] >> 16);
    s[2] ^= (uint8_t)(k[0] >> 8);
    s[3] ^= (uint8_t)(k[0]);

    s[4] ^= (uint8_t)(k[1] >> 24);
    s[5] ^= (uint8_t)(k[1] >> 16);
    s[6] ^= (uint8_t)(k[1] >> 8);
    s[7] ^= (uint8_t)(k[1]);

    s[8] ^= (uint8_t)(k[2] >> 24);
    s[9] ^= (uint8_t)(k[2] >> 16);
    s[10] ^= (uint8_t)(k[2] >> 8);
    s[11] ^= (uint8_t)(k[2]);

    s[12] ^= (uint8_t)(k[3] >> 24);
    s[13] ^= (uint8_t)(k[3] >> 16);
    s[14] ^= (uint8_t)(k[3] >> 8);
    s[15] ^= (uint8_t)(k[3]);
}

void inv_linear_transformation(uint8_t *s) {
    unsigned int X[4], i;
    for (i = 0; i < Nb; ++i) {
        X[i] = (s[Nb * i] << 24) | (s[Nb * i + 1] << 16) | (s[Nb * i + 2] << 8) | (s[Nb * i + 3]);
    }

    X[2] = inv_rotword(X[2], 22);
    X[0] = inv_rotword(X[0], 5);
    X[2] ^= X[3] ^ (X[1] << 7);
    X[0] ^= X[1] ^ X[3];
    X[3] = inv_rotword(X[3], 7);
    X[1] = inv_rotword(X[1], 1);
    X[3] ^= X[2] ^ (X[0] << 3);
    X[1] ^= X[0] ^ X[2];
    X[2] = inv_rotword(X[2], 3);
    X[0] = inv_rotword(X[0], 13);

    for (i = 0; i < Nb; ++i) {
        s[Nb * i + 3] = (uint8_t)(X[i] & 0xff);
        s[Nb * i + 2] = (uint8_t)(X[i] >> 8 & 0xff);
        s[Nb * i + 1] = (uint8_t)(X[i] >> 16 & 0xff);
        s[Nb * i] = (uint8_t)(X[i] >> 24 & 0xff);
    }
}
```

```c
void inv_initial_permutation(uint8_t *s) {
    short i;
    for (i = 127; i >= 0; --i) {
        uint8_t b_position = ((i << 5) % 127);
        uint8_t bit_a = (1 << (i & 7)) & s[i >> 3];
        uint8_t bit_b = (1 << (b_position & 7)) & s[b_position >> 3];
        if (bit_a > 0 && bit_b == 0) {
            s[i >> 3] -= bit_a;
            s[b_position >> 3] += (1 << (b_position & 7));
        }
        else if (bit_a == 0 && bit_b > 0) {
            s[i >> 3] += (1 << (i & 7));
            s[b_position >> 3] -= bit_b;
        }
    }
}
```

安全性與效能評估 ：

在安全性的驗證，我們選擇判斷 Serpent 是否符合雪崩效應。雪崩效應是一個加密演算法的理想屬性，指的是當輸入的明文只要發生一點點變化，加密後的密文就會發生巨大的改變，具體來說，反轉輸入之明文的其中一個 bit，加密後之密文的每個 bit 會有 50%的機率發生反轉，也就是說只要 Serpent 具有以上的特性，我們就可以說這個加密演算法具有一定程度的安全性。

若我們想測試 Serpent 是否具有雪崩效應的特性，我們必須統計改變輸入的一個 bit，輸出的每個 bit 發生反轉的機率，發生反轉的機率統計方式如下：

1. 以明文 P 經由固定金鑰 K 加密，得 C1

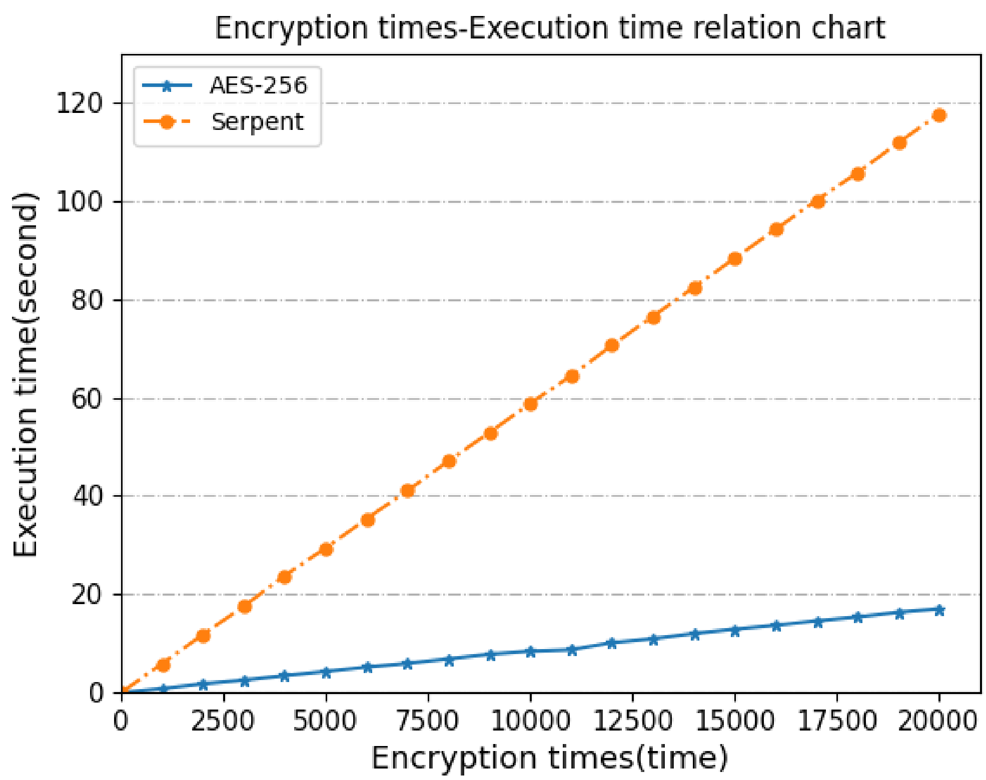
2. 改變明文的一個位元得 P’，經由 K 加密得 C2

3. 統計 C1 與 C2 相異的位元個數，除以明文位元數得出發生反轉的機率

我們設計了一個 C 語言函式進行計算，其部分的統計數據與最後的結果如下圖，可以發現結果非常近似於 50%，因此我們可以說這個加密演算法具有一定程度的安全性。

```
ciphertext 122 :        61 81 29 6a af 8f 80 9c 29 2b 45 5c a8 c9 f9 61
change bit num:64
change prob:0.500000

ciphertext 123 :        5e f2 ab c5 bd a5 5a 50 b7  5 15 98 e3 9e ff e0
change bit num:62
change prob:0.484375

ciphertext 124 :        f6 f6 33 be c0 7f 49 27 79 b7 30 63 53 68 89 3e
change bit num:72
change prob:0.562500

ciphertext 125 :        a5 cd f4 41 a7 55 9f 83 c5 7f b0  8 24 f9 c4 89
change bit num:63
change prob:0.492188

ciphertext 126 :        89 23 ec  7 28 5f 78 f2 35 cd 58 a2  c c5 c0 4f
change bit num:55
change prob:0.429688

ciphertext 127 :         4 9c 6d 40 d3 b8 32 a8 88 4a f4 22  3 e4 e3 85
change bit num:60
change prob:0.468750

ciphertext 128 :        6a 1e  6 d3 4c 1f 3e ac e8 cf a2 5f 7f a4 6d f3
change bit num:66
change prob:0.515625

change prob for 128 times test using total_change:      0.507202
change prob for 128 times test using total_prob:        0.507202
```

而在效能方面的驗證，由於 Serpent 不論金鑰長度是 128 bits、192 bits 還是 256 bits，都會執行 32 回合，所以我們把 Serpent 和 AES-256 拿來做對比，讓它們分別執行固定次數的加密並計算執行時間，結果如下：



可以發現 AES-256 較 Serpent 快上許多，若以最小平方法近似直線之斜率比較，約快了六倍，這與 Serpent 高達 32 的加密回合數有關，不過考量暴力破解所需的時間是將所有金鑰的可能帶入進行加密，高回合數帶來的執行時間也代表 Serpent 具有更高的安全性。

文獻參考 ：

- Ross Anderson, Eli Biham, and Lars Knudsen, “Serpent: A Proposal for the Advanced Encryption Standard”
- https://crypto.stackexchange.com/questions/67983/how-do-the-serpent-s-boxes-work