Bit Serial Implementation of SIMON in software

Moitrayee Chatterjee, Prerit Datta

Department of Computer Science

Texas Tech University

I. Introduction

SIMON and SPECK are categorized as a family of lightweight block ciphers. The ciphers were released by NSA[1] in June 2013 and these are cross platform ciphers as opposed to many other block ciphers [1]. Being lightweight, means the cipher performs well on constrained platforms for which it was designed and thus enabling cryptographic security be attained at quite a much lower cost. SIMON was optimized for hardware and its sister cipher SPECK for software. The ciphers have been investigated by many researchers and have proved to be extremely efficient in very highly resource constrained embedded systems like sensor nodes, RFIDs, FPGAs, etc. SIMON shares several core cryptographic parameters with AES, its lightweight predecessor..

Following sections describes what a *bit-slicing* means, how it differs from a block based algorithm and the problem definition and solution. We tested our implementation on SIMON 128/128, which means the input and key size were chosen to be 128 bits each. This implementation has **68** rounds as shown in fig. 1.

block size $2n$	key size mn	word size n	key words m	const seq	rounds T
32	64	16	4	z_0	32
48	72	24	3	z_0	36
	96		4	z_1	36
64	96	32	3	z_2	42
	128		4	z_3	44
96	96	48	2	z_2	52
	144		3	z_3	54
128	128	64	2	z_2	68
	192		3	z_3	69
	256		4	z_4	72

Fig. 1: Simon Parameters

[1] National Security Agency

II. Bit slice implementation of SIMON

When implementing a block cipher on hardware, there are several parallelism choices (bit level, round level, and encryption level) that affect the area and throughput of the design. We tried bit-serial implementation of SIMON in C language to see if there was any performance difference between Block-based implementation and Bit-sliced version. In bit level parallelism, the input size of the operators range from one bit to n-bits where n is the block size. The whole idea is to perform encryption and other operations bit-by-but instead of performing operation an entire block at once. This implementation is purportedly increases performance and several operations can be performed in parallel. Bit-slice implementation in DES was described as non-conventional yet an efficient way that resulted in tremendous throughput. This implementation in DES involved breaking down of DES algorithm into logical bit operations to facilitate N parallel encryptions on an N-bit microprocessor [2]. The implementation in DES realized speeds of 5 times faster than the standard DES on 64-bit processors on 64 –bit words [3]. Bit-slice was also applied on AES algorithm and the architecture performance of different microprocessors was analyzed. Intel core 2, Pentium 4, and AMD Athlon 64 were considered for evaluation for mainly 64-bit operations and 128-bit SIMD operations. All the AES operations are converted into logical operations and the entire encryption happens at the fastest rate of 170 clock cycles [3][4].

III. Problem Definition

The SIMON algorithm uses Feistel structure and it is essentially a lightweight block cipher aimed to work under a constrained hardware configuration.

A. The Round Function:

SIMON 2n encryption methodologies involves consecutive operations on n-bit words:

- Bitwise AND
- Bitwise XOR
- Left and Right circular shift

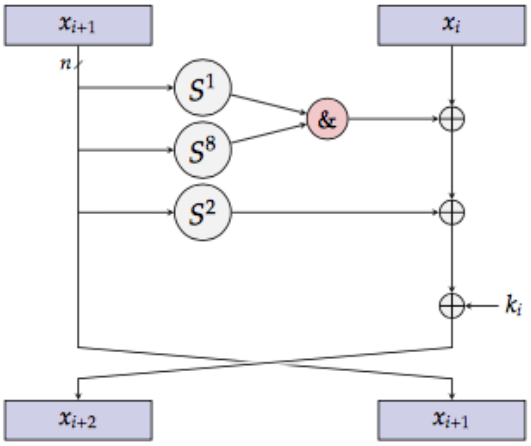


Fig 2: Feistel Structure for SIMON round function [1]

x_i and x_{i+1} are the two n-bit blocks of the 2n-bit inputs as well as the outputs of the previous round. x_{i+1} is operated with shift function and bitwise AND, and then it is bitwise XORed with x_i. The resultant bits are written back to x_{i+2} .

B. Key Expansion: SIMON encryption uses unique keys for each of the rounds and there can be 2,3 or 4 words(m) in a key. Logical operations are similar for m= 2 and m= 3 and the most significant word is circular shifted to right by 3 and 4 and then it is XORed with the least significant bit and round constant Z_i. However for m=4, there is an additional step where the most significant word(k_{i+3}) is first right circular shifted by 3 and then XORed with k_{i+1}.the resultant bits are then right circular shifted by 1 and XORed with the least significant bit. The final result is then written into most significant word and all these words are shifted to right by one word. This ensures the elimination of slide properties and symmetries caused by circular shifts. Logical operations are similar for m= 2 and m= 3 and the most significant word is circular shifted to right by 3 and 4 and then it is XORed with the least significant bit and round constant Z_i. However for m=4, there is an additional step where the most significant word(k_{i+3}) is first right circular shifted by 3 and then XORed with k_{i+1}.the resultant bits are then right circular shifted by 1 and XORed with the least significant bit. The final result is then written into most significant word and all these words are shifted to right by one word. This ensures the elimination of slide properties and symmetries caused by circular shifts

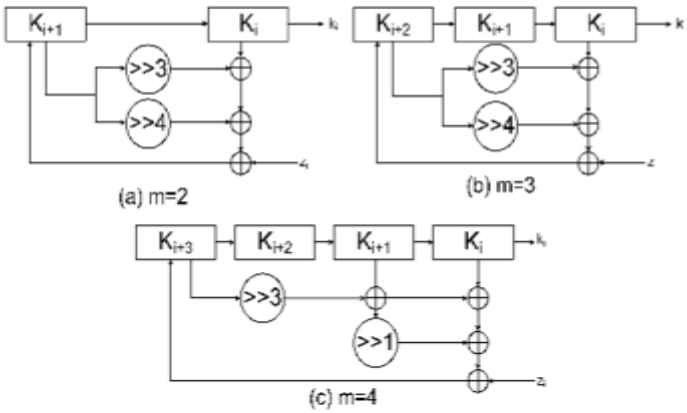


Fig 3: Key Expansion[4]

C. Encryption and Decryption: Plaintexts are converted to cyphertexts through the key expansion and the inverse of the round function is used to decrypt the cipher texts.

D. Bit serial: As per definition given by digital logic, bit serial is processing of data one bit at a time, than processing all bits or a word at a time. SIMON can be bit-serialized to achieve parallelism, that in turn will lead to a more cost-effective hardware configuration, if this bit-serialization is implemented in hardware using logic gates.

Our aim is to implement SIMON in bit-serialized manner using C programming and test its performance using simulator or we can convert out C code to program microcontroller and study the performance.

III. SOLUTION

SIMON algorithm uses Feistel structure as an underlying mechanism. For this project, we are using SIMON **128-bit block input** size and **128-bit key input.** The main step of the encryption as shown below, applies left-circular shifts and various XOR operations:

$$x \leftarrow y \oplus (Sx \,\&\, S^8x) \oplus S^2x \oplus k[i]$$

Since SIMON is essentially a *fiestel structure*, based on our selected parameter for SIMON 128/128. Each of the input would be broken down into two blocks of 64 bits each. **Fig**. 3 shows various transformations of a **64-bit input block (upper block)** as it goes several Left-circular shift operations.

For calculating the ***First bit*** of the next output, we would require AND operation (15^{th} **&** 8^{th}) bit of input block ***x*** followed by XOR with **14^{th} bit** of input block ***x***. This can be clearly demostrated by looking at position 0 (the first bit from right) in **Fig. 4** (b), (c) and (d). So, it can be said that atmost **3-bits** are required for each step to implement bit serial implementation i.e bits : **n, n-1** and **n-7.**

The key expansion for SIMON algorithm is same for m=2 and 3. For the **SIMON 128/128** implementation, the

specification calls for value of m to be used as two (m=2). **Fig. 5** shows a simplified process of how this expansion takes place for m=2 and 3. According to the specification chosen for this project, the 128-bit Key input would be split into two equal block size of 64-bits each which is shown in the first step of Fig. 2. The rest of the processes in the Fig. 2 are self-explanatory. To have a generalized form of SIMON for any n (word size), we can simplify the original algorithm step-by-step as follows:

$$tmp \leftarrow tmp \oplus S^{-1} tmp \text{ where, } tmp \leftarrow S^{-3} K[i-1]$$

This can be replaced with statement:

$$tmp_l = K_{(l+3) \bmod n}[i-1] \oplus K_{(l+4) \bmod n}[i-1],$$

The values of l and i will vary as follows:
l = 0,1,….n-1,
i= m to T-1

Hence, the final step of algorithm:

```
k[i] ← ~k[i-m] ⊕ tmp ⊕ z[j][(i-m) mod 62] ⊕ 3
```

Now becomes:

$$k[i] \leftarrow k[i-m] \oplus tmp_l \oplus z[j][(i-m\} \bmod 62] \oplus 3$$

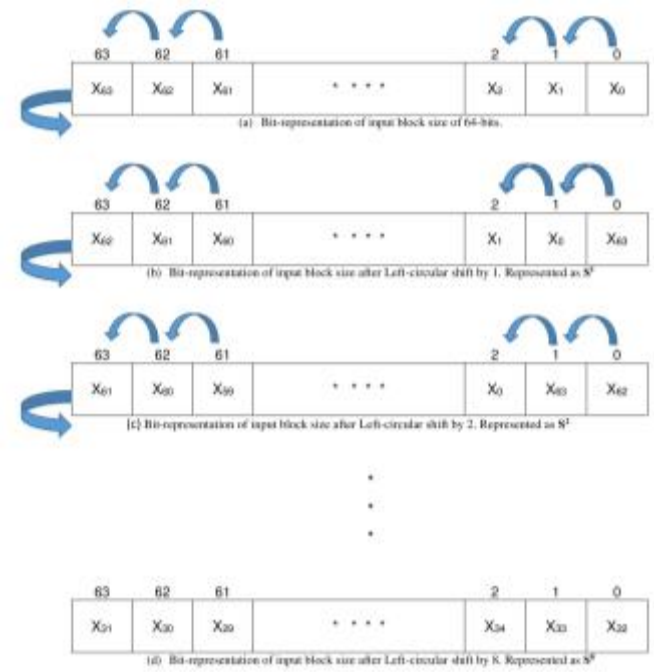


Fig. 4 Representation of various left Left-Circular Shift operations of a input block size of 64 bits (total 128 bits)

IV. Implementation

We have implemented our bit slicing algorithm in C++. Please refer the Annexure 1 and Annexure 2 for the details.

V. RESULTS

We have implemented our bit serialization strategy using C++ code and compared the performance with the existing C++ SIMON code. Our code is not yet been executed in parallel. And it bit serializes only the encryption algorithm. We are using the key expansion and decryption using the existing SIMON code. Figure 1 shows the output from our code along with the time taken by it for the total execution. Figure 2 shows the time taken by the existing SIMON code.

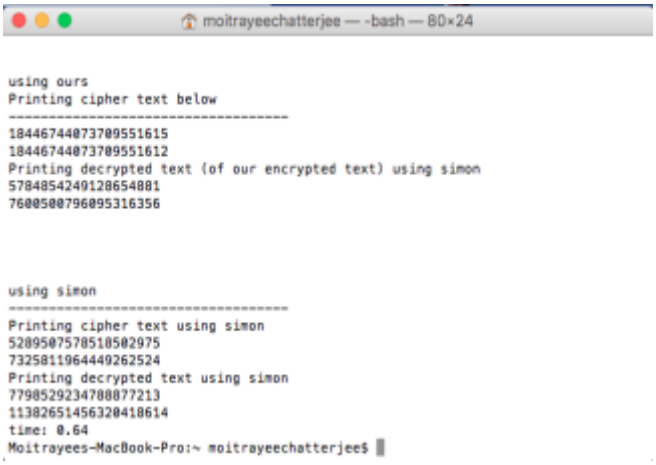


Figure 1: SIMON Bit serial Implementation

Figure 2: SIMON without bit serial approach

VI. Conclusions

As we can see the existing SIMON code already takes competitivly minimal time for a full cycle of Key generation, encryption and decryption, even if we take the bit-serial approach, and split the execution in say 8 different processing units, our approach won't be able to cope up with the philosophy behind the SIMON algorithm: lightweight (i.e. efficient in terms of hardware space and system overhead) and fast. Hence we can conclude that SIMON is already optimized for a resource critical environment.

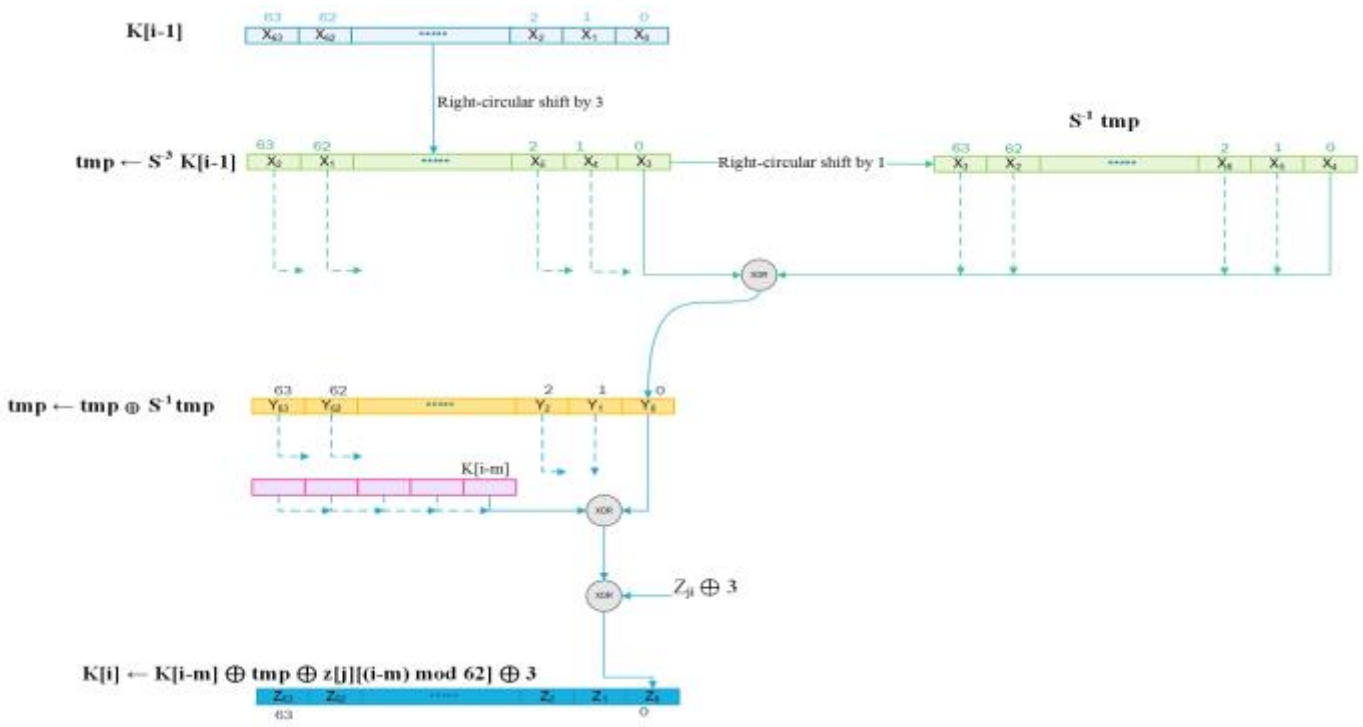


Fig. 5 Key Expansion for m=2 and 3

REFERENCES

[1] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, Louis Wingers, "The simon and speck of lightweight block ciphers," National Security Agency 9800 Savage Road, Fort Meade, MD 20755, USA, June 2013.

[2] Rebeiro, C., Selvakumar, D. and Devi, A.S.L., 2006, December. Bitslice implementation of AES. In International Conference on Cryptology and Network Security (pp. 203-212). Springer Berlin Heidelberg.

[3] Biham, E., 1997, January. A fast new DES implementation in software. In International Workshop on Fast Software Encryption (pp. 260-272). Springer Berlin Heidelberg

[4] Aysu, A., Gulcan, E., Schaumont, P.: SIMON says: Break area records of block ciphers on FPGAs. Embedded Systems Letters, IEEE 6(2), 37–40 (June 2014)

ANNEXURE 1: PSEUDOCODE

Input: 1. 128 bit plaintext "P".

2. 128 bit Key:

as of now we are using the SIMON Key expansion algorithm and storing it in a matrix form of K_{ij} : where i^{th} row and j^{th} column.

we have 68 different keys of 64 bits long.

Output:

Encrypted text C

Steps:

1. Break 128 bits plaintext into two parts upper (MSB) and lower (LSB) — P_u and P_l respectively.

$$|P_u| = |P_l| = 64 \text{ bits.}$$

2. As we are working with 128/128 SIMON, we have 68 rounds and each round generating intermediate cipher texts. If we call them X^i for level i

X^1 = Plaintext P_u

then $X^2_{w0} = P_l \wedge (X^1_{w63} \& X^1_{w56}) \wedge X^1_{w62} \wedge K[0]$

In generalised format.

$$X^i_{w_j} = X^{i-1}_L \wedge \left(X^{i-1}_{w_{(k-1)}} \,\&\, X^{i-1}_{w_{(k-5)}} \right) \wedge X^{i-1}_{w_{(k-2)}} \wedge k[i][j]$$

where $2 \leq i \leq 68$ and $0 \leq j \leq 63$.

we assume $k = |X^i_w|$ number of total bits and starts from 0.

i.e $X^i_w = X^i_{w_{63}} X^i_{w_{62}} \ldots X^i_{w_0}$

3. Our contribution is to remove the loop to compute the intermediate steps we have $X^i_L = X^{i-1}_w$.

As we introduce a tree structure and if we call it Cipher Tree to directly generate the final cipher $X^{68}_{w_{67}}$ For this we have to calculate X^{68}_w which is equal to X^{68}_L.

4. From Cipher Tree C_T generate $X^{68}_{w_i}$ and $X^{67}_{w_i}$ in terms of X^1 along with that we need $K[66][0,\ldots,63]$ and $K[67][0,\ldots 63]$.

5. Distribute computation of X_u^{67} among p processing units - where each p_j gets $\frac{64}{|P|}$ work units.

say processor ϕ_i computes $X_{w_i}^{67}$.

$p_0 \longrightarrow X_{u_0}^{67}$

$p_1 \longrightarrow X_{w_1}^{67}$

⋮

6. Repeat same step for X_w^{62}

Recursive func^n to calculate X_{wj}^i

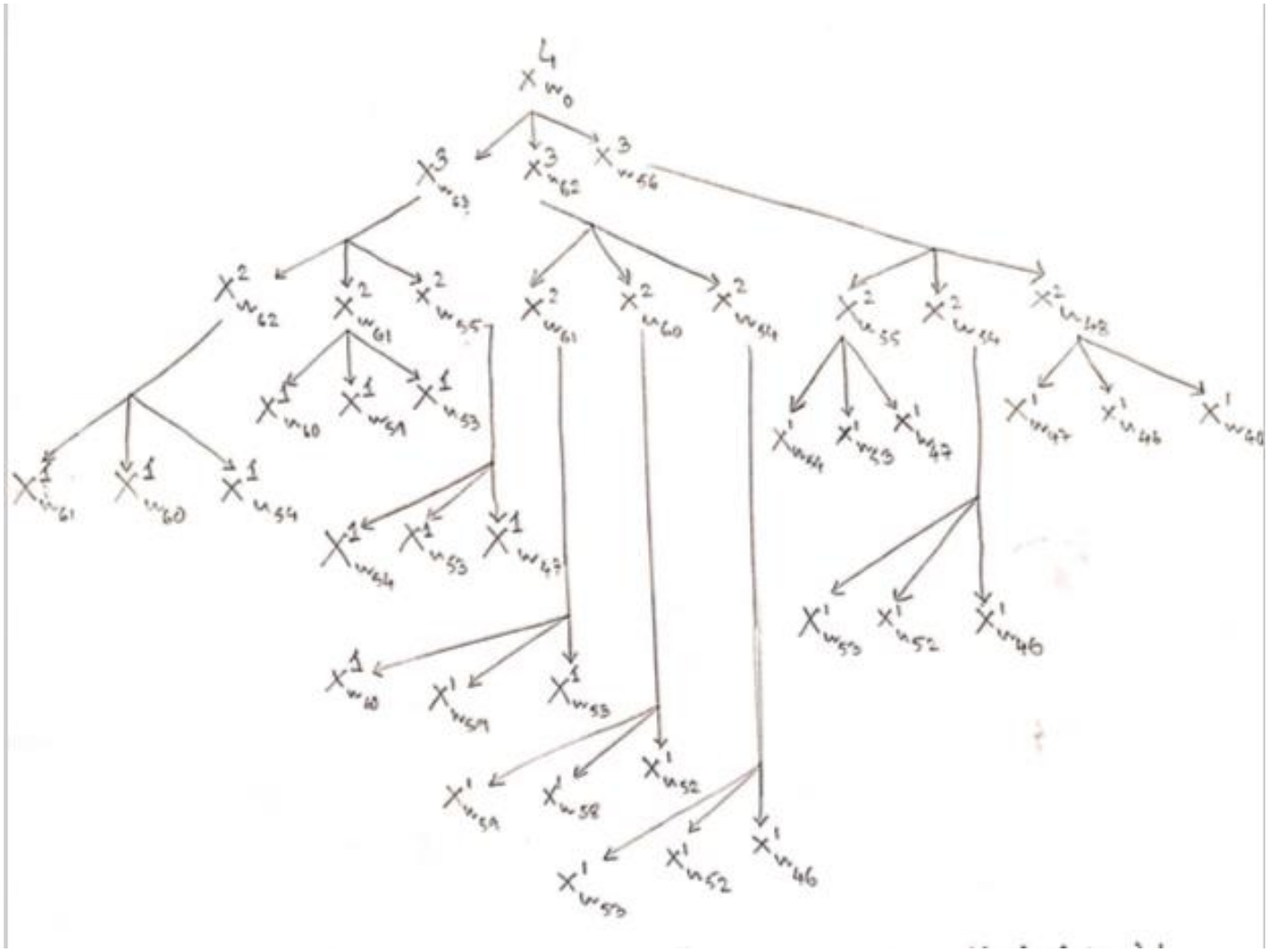
Input: Bit slicing table T of size 64×3 where each row T_i is a 4-tuple $\langle j, j_1, j_2, j_3 \rangle$

Output: X_{wj}^i in terms of X_{wj}^1 for $0 \le j \le 63$.

Algo: if $i == 1$

return $T[j]$ // which is $(X_{j_1} \& X_{j_3}) \wedge X_{j_2}$

else $X_{wj}^i = f(T^{i-1}[j_1] \& T^{i-1}[j_3]) \wedge f(T^{i-1}[j_2])$



Representation of bit zero of the intermediate cipher at level 4 using the upper bits of input text.

ANNEXURE 2: C++ CODE

```
#include <iostream>
#include <vector>
#include <climits>
#include <random>
#include <tuple>
#include <functional>
#include <iterator>
#include <bitset>


using namespace std;

#define WORD_SIZE 64
#define ITERATION_SIZE 1

#define MASK 0xFFFFFFFFFFFFFFFF
#define RNDCONST 0xFFFFFFFFFFFFFFFC
#define RNDNUM 68
#define M 2
#define J 2
#define security_input 8

#define u64 unsigned long long

u64 z[68] = {
        1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0,
1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1,
1};
u64 k[RNDNUM]; // Key words


unsigned long long circular_shift_left(unsigned long long number, int shift_amount)
{
    unsigned long long value;


    value = ((number << shift_amount) | (number >> (WORD_SIZE - shift_amount)));
    value = value & MASK;

    return value;
}

unsigned long long circular_shift_right(unsigned long long number, int shift_amount)
{
    unsigned long long value;
```

```
    value = ((number >> shift_amount) | (number << (WORD_SIZE - shift_amount)));
    value = value & MASK;

    return value;
}

void key_expansion(u64 k_initl, u64 k_init2)
{
    u64 temp,templ,temp2;
    int z_bit;

    for (int i = 0; i < RNDNUM; i++)
    {
        if (i==0)
            k[i]=k_initl;
        else if (i==1)
            k[i]=k_init2;
        else{
            temp = circular_shift_right(k[i - 1], 3);
            temp2 = temp ^ circular_shift_right(temp, 1);
            k[i] = k[i - 2] ^ z[(i - 2)] ^ temp2 ^ RNDCONST;
        }
    }
}

void encrypt_round(u64 x, u64 y, u64 &cipl, u64 &cip2)
{
    u64 templ, temp2, temp3;

    cipl = x;
    cip2 = y;

    for (int i = 0; i < RNDNUM; i++)
    {
        templ = cipl;
        temp2 = (circular_shift_left(cipl, 1) & circular_shift_left(cipl, 8)) ^
circular_shift_left(cipl, 2);
        cipl = cip2 ^ temp2 ^ k[i];
        cip2 = templ;
    }

}

void decrypt_round(u64 x, u64 y,u64 &x_out,u64 &y_out)
{
    u64 templ, temp2, temp3;

    x_out = y;
    y_out = x;

    for (int i = RNDNUM - 1; i >= 0; i--)
```

```
    {
        templ = y_out;
        temp2 = (circular_shift_left(y_out, 1) & circular_shift_left(y_out, 8)) ^
circular_shift_left(y_out, 2);
        y_out = x_out ^ temp2 ^ k[i];
        x_out = templ;

    }

}


void generateBitPositioningTable(vector<vector<int64_t>>& T){
    vector<int64_t> entry;
    entry.push_back(0);
    entry.push_back(1);
    entry.push_back(7);
    T.push_back(entry);
    for(int i=0; i< 63; i++){
        vector<int64_t> prev = T[i];
        vector<int64_t> entry;
        for(int j=0; j< prev.size(); j++){
            int curr = (prev[j]-1)<0?63:(prev[j]-1);
            entry.push_back(curr);
        }
        //cout<<entry[0]<<", "<<entry[1]<<", "<<entry[2]<<endl;
        T.push_back(entry);
    }
}

void generateRandomKeys(vector<vector<int64_t>>& K){
    int rMax = 20;
    for(int i=0; i< 68; i++){
        vector<int64_t> entry;
        for(int j=0; j< 64; j++){
            entry.push_back(rand()%rMax+1);
            //cout<<rand()%rMax+1<<endl;
        }
        K.push_back(entry);
        //copy(entry.begin(), entry.end(), std::ostream_iterator<int>(cout, "Wn"));
    }
}

//global declaration of dynamic programming table
vector<pair<uint64_t, uint64_t>> dp(100);
bool done[100];


//recursive routine to generate cipher text
bool generateX(u64& k_initl, vector<vector<int64_t>>& T, int64_t level, uint64_t& upper, uint64_t&
lower, int64_t bit=0){
```

```
    //cout<<"level = "<<level<<endl;
    if (level == 0) return (bool)((upper >> bit) & 1);
    if (done[level]) {

        return (bool)((dp[level].first >> bit) & 1);
    }

    uint64_t tmp = upper, out = 0ULL;
    for(int i=0; i< 64; i++){
        //cout<<"i = "<<i<<endl;
        out |= (((generateX(k_initl, T, level-1, upper, lower, T[i][0]) & generateX(k_initl, T, level-1,
upper, lower, T[i][1])) ^ generateX(k_initl, T, level-1, upper, lower, T[i][2])) ^ k_initl ^ lower) <<
i;
        //out |= (((generateX(K, T, level-1, upper, lower, T[i][0]) & generateX(K, T, level-1, upper,
lower, T[i][1])) ^ generateX(K, T, level-1, upper, lower, T[i][2]))) << i;
    }

    upper = out, lower = tmp;
    dp[level] = {upper, lower};
    done[level]=true;
    return (bool)((upper >> bit) & 1);
}


int main() {

    int start_s=clock(); // calculate time taken in execution: start timer

    /*
     key expansion
     */
    u64 cipl, cip2, x_out, y_out;

    u64 x,y;
    u64 k_initl,k_init2;
    u64 randoml=1,random2=1,random3=1,random4=1;
    u64 temp;
    u64 random;

    cout<<"Generating keys"<<endl;
    if(security_input==8)
    {
        x = 0x6373656420737265;
        y = 0x6c6c657661727420;
        k[0] = 0x0706050403020100;
        k[1] = 0x0f0e0d0c0b0a0908;
    }
    // You have to define the version of SIMON you want to use at the beginning of this file.
    // To override the random inputs generated, you can uncomment these lines and use them instead.
    // x : plaintext upper word
    // y : plaintext lower word
```

```
//k[i] : Key words, k[0] lowest key word

x = 0x6373656420737265;
y = 0x6c6c657661727420;
k_initl = 0x0706050403020100;
k_init2 = 0x0f0e0d0c0b0a0908;

key_expansion(k_initl,k_init2);

/*
 key expansion
 */

//plain text is a 128 bit integer, we take an array of 64bit integers
//lest assume plainText[0] is upper and plainText[1] is lower
vector<uint64_t> plainText;
plainText.push_back(70);
plainText.push_back(56);
cout<<"Printing plain text below"<<endl;
//cout<<plainText[0]<<plainText[1]<<endl;
cout<<x<<","<<y<<endl;
uint64_t x_bck = x, y_bck = y;
uint64_t x_our = x, y_our = y;
u64 xour_out, your_out;
//generate the bit positioning table
//cout<<"Generating bit positioning table"<<endl;
vector<vector<int64_t>> T;
generateBitPositioningTable(T);


cout<<endl<<endl<<endl<<endl;
cout<<"using ours"<<endl;
//calculate X66-upper for all 64 bits - rounds go from 0 to 67, hence X67 is the cipher text
vector<int64_t> cipherText;

generateX(k_initl, T, 66, x_our, y_our);
uint64_t yCipher = x_our;

generateX(k_initl, T, 67, x_our, yCipher);
//cipherText.push_back(plainText[0]); cipherText.push_back(lowerCipher);

cout<<"Printing cipher text below"<<endl;
cout<<"-----------------------------------"<<endl;
//cout<<bitset<64>(plainText[0])<<"Wn"<<bitset<64>(plainText[1])<<endl;
cout<<(x_our)<<"Wn"<<(yCipher)<<endl;
cout<<"Printing decrypted text (of our encrypted text) using simon"<<endl;
decrypt_round(x_our, yCipher, xour_out, your_out);
cout<<xour_out<<"Wn"<<your_out<<endl;

cout<<endl<<endl<<endl<<endl;
cout<<"using simon"<<endl;
```

```
    cout<<"-----------------------------------"<<endl;
    cout<<"Printing cipher text using simon"<<endl;
    encrypt_round(x, y, cipl, cip2);
    cout<<cipl<<"Wn"<<cip2<<endl;

    cout<<"Printing decrypted text using simon"<<endl;
    decrypt_round(cipl, cip2, x_out, y_out);
    cout<<x_out<<"Wn"<<y_out<<endl;

    int stop_s=clock(); // stop the timer
    cout << "time: " << (stop_s-start_s)/double(CLOCKS_PER_SEC)*1000 << endl; //print the total time
taken

    return 0;
}
```