University of Waterloo / Author: Abdurrahman Elbuni

Project Overview:

In this project the 32/64 version of SIMON lightweight block cipher family will be implemented for assessment and evaluation of its security, randomness and implementation complexity. The security will be evaluated by applying cross correlation for both input as a (White Noise) and corresponding output (ciphertext) and analyzing the results.

The security analysis will mostly be applied on the block cipher itself in ECB mode, without forming any specific encryption modes that will increase its security, such as CFB, CTR or CBC.

Lightweight Block Ciphers:

The relatively new field of lightweight cryptography addresses security issues for highly constrained devices. A great deal of excellent work has already been done in this area, much of it aimed specifically at developing block ciphers suitable for lightweight cryptographic applications. The SIMON block cipher built upon that body of work. According to [1], the goal behind the design of SIMON is to provide the security that the cryptographic community expects, while also delivering the flexibility and performance characteristics that developers require.

The term lightweight is used broadly to mean that an algorithm is suitable for use on some constrained platform. But the features that make an algorithm excel on an 8-bit microcontroller, say, do not necessarily imply that it can be realized by an extremely small circuit.

Among the block ciphers intended for use on constrained devices, some have been designed specifically to perform well on dedicated Application-Specific Integrated Circuits (ASICs), and thus can be realized by small circuits with minimal power requirements. Others are meant to perform well on low-cost microcontrollers with limited flash, SRAM, and/or power availability. Unfortunately, design choices meant to optimize performance on one platform often adversely affect performance on another, and SIMON was designed with consideration to mitigate the possible adverse effects.

The SIMON family of block ciphers:

The SIMON block cipher with an n-bit word (and hence a 2n-bit block) is denoted SIMON2n, where n is required to be 16, 24, 32, 48, or 64. SIMON2n with an m-word (mn-bit) key will be referred to as SIMON2n/mn. For our case, SIMON32/64 refers to the version of SIMON acting on 32-bit plaintext blocks and using a 64-bit key.

Each instance of SIMON uses the familiar Feistel structure. The algorithm designers have engineered it to be extremely small in hardware and easy to serialize at various levels, also high measures was taken so as not to sacrifice software performance.

In order to provide this flexibility, SIMON was designed as a family block cipher: It supports block sizes of 32, 48, 64, 96, and 128 bits, with up to three key sizes to go along with each block size. The SIMON family provides ten algorithms in all. Table (1) lists the different block and key sizes, in bits, for SIMON.

block Size	*key Sizes*
32	64
48	72, 96
64	96, 128
96	96, 144
128	128, 192, 256

Table (1)

A. **Round Functions**

SIMON2n encryption and decryption make use of the following operations on n-bit words:

- Bitwise XOR, $\oplus$
- Bitwise AND, &, and
- Left circular shift, S^j by j bits.

For $\in GF(2)^n$, the key-dependent SIMON2n round function is the two-stage Feistel map defined by:

$$(x_i, x_{i+1}) \rightarrow (x_{i+1}, x_{i+2}) \; where \; x_{i+2} = R(x_i, x_{i+1}, k_i) \; where$$
$$f(x, y, k) = \left(Sy \; \& \; S^8 y\right) \oplus S^2 y \oplus x \oplus k \rightarrow (1)$$

Thus, for plaintext $P = (x_0, x_1)$, the ciphertext is

$$C = Enc_{Simon,k}(P) = (x_{32}, x_{33})$$

where k is the round key. The inverse of the round function, used for decryption, is:

$$f^{-1}(y, x, k) = Dec(C) = P$$

In this project the SIMON key schedules take a key and from it generate a sequence of $64bit$ key words k_0,.,k_{T-1}, where T is the number of rounds. The round of SIMON 32/64 is defined as 32.

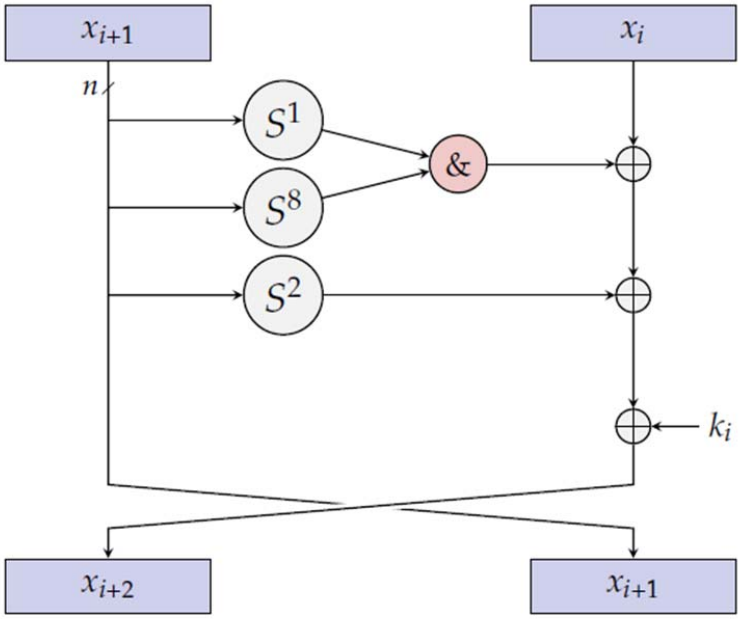

Fig (1) – Feistel structure for SIMON round function

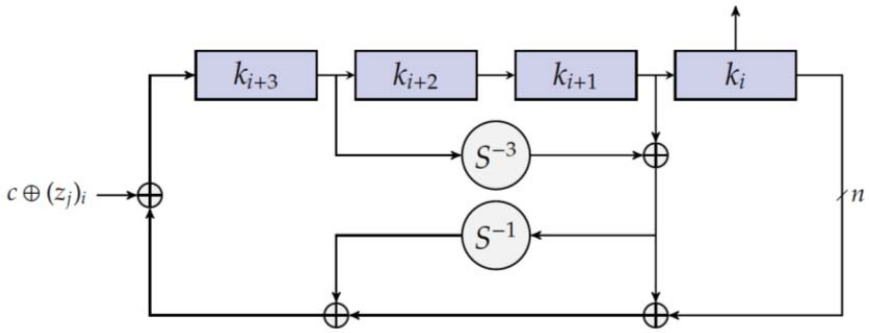

Fig (2) – 4 (16-word) key scheduling

B. **Key Scheduling**

In the key schedule, each register holds an n = 16-bit word and the feedback is given by

$$k_{i+4} = f(k_i, k_{i+1}, k_{i+2}, k_{i+3}) \oplus c \oplus z_i \rightarrow (2)$$

where

$$f(x_0, x_1, x_3, x_4) = x_0 \oplus (I \oplus S^{-1})[S^{-3}(x_3) \oplus x_2], I \text{ is the identity map the constant} \rightarrow (3)$$

$$c = 2^{16} - 4 = 0\text{xfff} \ldots \text{fc or } c = 1 \ldots 100$$

where the most right bit is LSB, and at time i, one bit z_i added to the LSB of c, is the i bit of an LFSR of degree 5, i.e.,

$$z_{i+5} = z_{i+4} \oplus z_{i+2} \oplus z_{i+1} \oplus z_i, \qquad i = 0, 1, \dots, 31$$

where $(z_0, z_1, z_2, z_3, z_4) = (1, 1, 1, 1, 1)$ is an initial state, i.e. $\{z_i\}$ is an m-sequence of period 31 generated by an LFSR with the primitive polynomial $x^5 + x^4 + x^2 + x + 1$.

C. **Implementation phase**

In this project an implementation of Simon 32/64 block cipher using c programming language has been completed to conduct further studies about the SIMON block cipher security. See Appendix 2 for implementation.

After the implementation part, a specific experiment on the SIMON32/64 block cipher will be taken according the description given in the next section.

i. Experiment Procedure:

In this experiment the key need to be initialized to $\mathrm{k} = (1,0, \dots, 0)$ as 64-bit binary vector. And then a counter mode encryption will be applied for $255 \text{ x } 32 = 8160$ bit message, where key stream bits for i$^{\text{th}}$ message block is generated by an input of a 32-bit vector from an LFSR of degree 8 instead of using the counter values: $\mathrm{counter} + \mathrm{i}$.

Specifically, i^{th} message block is retrieved by

$$y_i = (a_i, a_{i+1}, \dots, a_{i+31}), \qquad i = 0,1, \dots,254$$

where

$$a = (a_0, a_1, \dots, a_{255})$$

is an m-sequence generated by an LFSR of degree 8 with primitive polynomial $x^8 + x^7 + x^2 + x + 1$, and an initial state $(a_0, a_1, \dots, a_7) = (1,0,\dots,0)$. The 32-bit key stream vector for input y_i, denoted as s_i is given by

$$s_i = Enc_{Simon,k}(y_i);\ i = 0, 1, \dots, 254$$

The requirement is to encrypt the input Y using Simon32/64 with the fixed key stream defined to generate array of 32bit ciphertext with size of *255*:

$$Y = \begin{pmatrix} y_0 \\ \vdots \\ y_{254} \end{pmatrix} Enc_k \rightarrow S = \begin{pmatrix} c_0 \\ \vdots \\ c_{254} \end{pmatrix}$$

Consequently, from the generated input and the ciphertext we shall compute the cross correlation for each column of the first 16 columns of S with each column of Y, where each column is treated as a sequence with period 255. Then, we shall list the upper bound and lower bound of those cross correlation values.

ii. Experiment Execution & Results:

a) A code has been implemented and verified successfully for SIMON 32/64 block cipher. The code-base been designed with high flexibility measures in mind, so that it can be extended easily in the future to emulate any other SIMON version. The following functions are the base library functions of the project code shown in Appendix (2):

- LFSR sequence generation function (***GenerateLFSR***), where a polynomial function, and an initial seed will be passed as an input and a sequence will be return as 8-bit array.
- 16-bit circular left shift function (***CircularLeftShift***) that can calculate and return the result of a j number of shifts.
- 16-bit circular right shift function (***CircularRightShift***) that can calculate and return the result of a j number of shifts.
- Simon cipher function (***SimonEncryption***), which will accept sub-keys and two 16-bit words to calculate and return the ciphertext as an array of two 16-bit words according to the block cipher diagrams shown in figure (1), and figure (2).
- Key generation function (***KeyGeneration***), which will take a variable LFSR sequence, and initial 64-bit key sequence as an input to generate the subsequent 128 set of 64-bit keys according and store them in a global array.
- Finally the cross correlation function (***CrossCorrelation***), which will take x and y as 255-bit vector as an input, and it will calculate their corresponding cross correlation and return it as 255 vector of integer values.

To attain the required results, we need to first generate the keys according to equation (2), or the diagram shown in fig (2). The key scheduling results of 128x64 bit vector is shown at appendix (1), table (1).

b) To calculate the cross correlation between the message (Y) and the ciphertext (S) we need to generate the sequence of the degree 8 LFSR with the primitive polynomial given as

$$x^8 + x^7 + x^2 + x + 1$$

The m-sequence of length $2^8 - 1 = 255$ generated from the program is as follow:

100000001111110010100101101011111000101100100001010001001001010101001100101110100011101100111110110100000010000010111101110111100001001110101100011110011011011111111010101110011100100110101000011011100000110000111000110011000100011010011110100100010101101

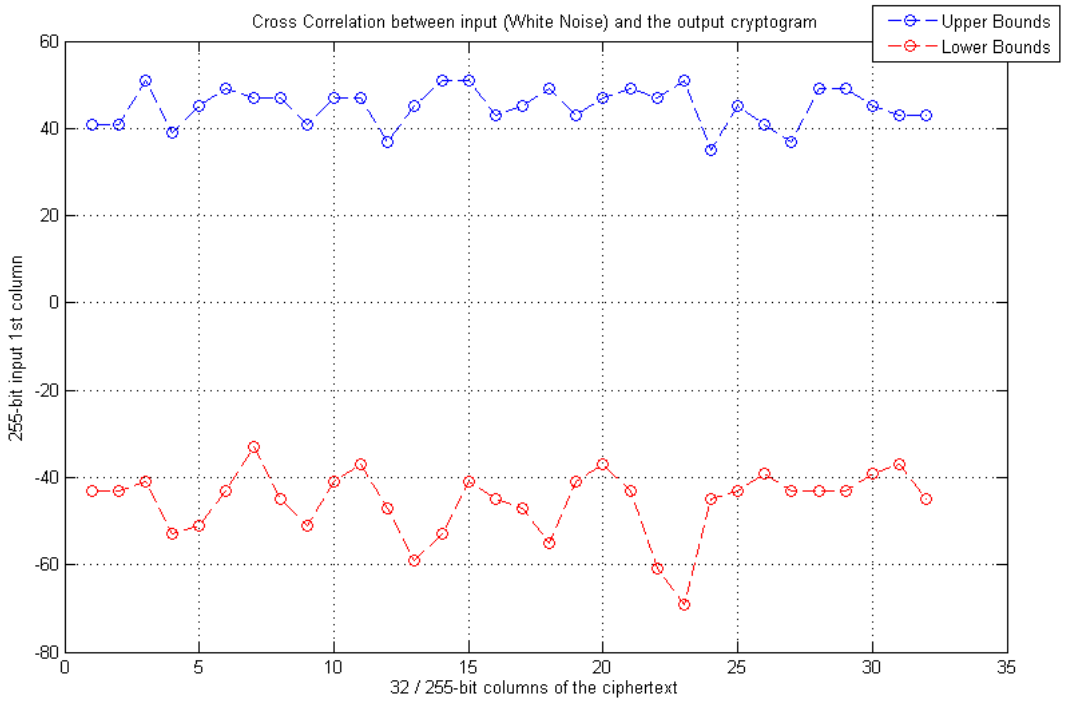

Fig (3) – Cross correlation result between the input and the 32 column of the ciphertext

i. SIMON Security Analysis & Extended Evaluation:

From the practical point of view, an ideal cipher is extremely difficult to implement for most practically relevant block sizes: For a given key, one randomly drawn permutation has to be stored. For the usual block size of b = 128 bit, the storage of such a permutation is practically infeasible. Thus, more structured (and, therefore, less random) assigning of permutations to key values has to be used. As a rule, to build a block cipher, a set of permutation generators is taken and parameterized by the key. From [2] "A block cipher E is called secure, if no attacks on E exist with both expected computational and data complexities lower than the corresponding complexities of any black-box attack", in another words, the adversary has nothing but to brute force the block cipher to decipher any ciphertext.

In our experiment analogically we have generated a white noise signal from 8-stage LFSR and pass it as a pseudorandom input to SIMON32/64 block cipher. By passing a white noise and monitor the resulted cryptogram pattern, ideally we should see a uniformly distributed pattern with single occurrence of each corresponding ciphertext that has a minimum relationship with the given input. However, from the experimental measurement of cross correlation in Figure (3), we can see that the highest cross correlation bound was 51 and the lowest minimum cross correlation bound was -63. These correlation values have no clear interpretation so far, but, theoretically the higher the absolute value of the cross correlation the lower diffusion (security) the block cipher would have.

a. Sub Experiment 1:

To create more sense from the cross correlation experiment, we will extend the original project experiment and create a sub-experiment by selecting 420 (255-bit) random variables that have been generated from highly random PRSG to simulate and identify what we should consider as very high upper bound and very low lower bound correlation values. Presumably, the result of this sub-experiment shall give us at least a soft reference on what is the extreme upper and lower bound cross-correlation that should be expected from a pair of randomly selected 255bit numbers. The combination of 420 random variables pairs with no repetition will be selected; then, each pair's cross correlation will be calculated; the upper bound and lower bound will be normalized by dividing the values by 255; and, finally the upper and lower bounds will be extracted and projected in a graph for a rigorous observation. The 420 random variables will form 87990 combinations of pairs according to the following equation:

$$Combination \binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

$$Cross\ correlation\ pairs = \frac{420!}{2!\,(420-2)!} = 87990\ pair\ of\ 255bit\ random\ variables$$

To generalize the analysis and generate a reference from this case study, a normalization for the cross correlation will be applied for every calculation, where the cross correlation value will be normalized and bounded as follow:

$$C = \frac{C}{\text{variable length} = 255} \rightarrow eq\ (4)$$

$$\mathrm{C} = [-1,1], \quad \forall\, \mathrm{C} \in \mathbb{R}\ .$$

where C is the cross correlation value.

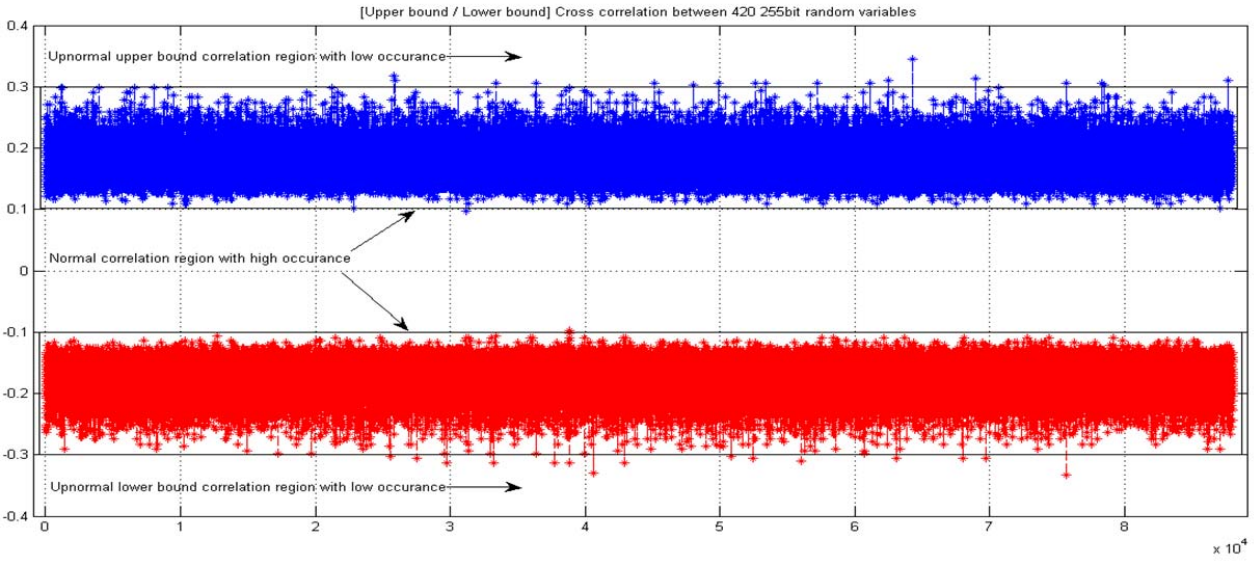

Fig (4) – Cross correlation between 420 of 255bit random variables

From the previous result shown in figure (4), we will establish an assumption. Since the aim of this particular experiment is to identify whether a Gaussian random input would be mapped to a random ciphertext in each encryption cycle and vice versa, which also should prove the strength of the diffusion and the randomness of the SIMON32/64 block cipher, an attempt to formulize a definition for both normal and extreme cases of correlation values using the sub-experiment of 420 random variables will be made.

Assumption 1:

Based on figure (4) observation:

1. Cross correlation range from [-0.29, 0.29] would be considered as a safe correlation region.
2. Cross correlation range from [0.3, 0.1] and [-0.3, -0.1] would be considered as up-normal region, which once many cross correlation values exceeds its minimum and maximum boundaries respectively, the pairs would become more likely to have a predictive pattern or relationship the more they converge toward the unity value (1 or -1).

This sub-experiment will help us look at the strength of diffusion of SIMON32/64 and not the confusion since the key has been fixed. As we will determine and compare the upper bound and lower bound of the cross correlation vector between the input (White Noise) and the ciphertext 32 columns.

Going back to our original experiment, the result in figure (3) will be normalized so that we acquire the next figure (4):

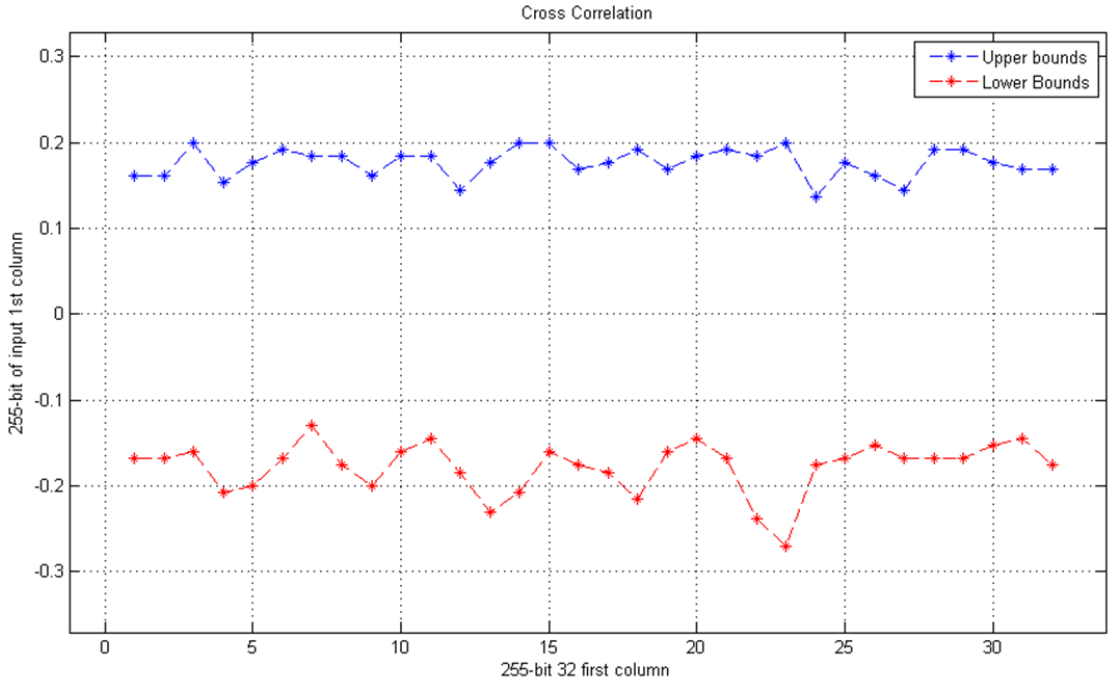

Fig (5) – Normalized cross correlation result between the input and the 32 column of the ciphertext

Relying on the assumption (1), the experiment result has shown normal correlation values as the upper and lower bounds are still within the normal bound region defined in assumption 1.

b. Sub Experiment 2:

In the original experiment the sample set for evaluation was too small; hence, a new approach will be adopted to expand and generate larger sample set with one-time-padding method (single key), but, this time we seek to find weak keys that could lead us in establishing more attacks. So that it can be passed to the same upper bound and lower bound correlation test to attain new results. In this second sub-experiment 100,000 new and random 64bit keys will be fed to SIMON32/64 to encrypt 255 bit of input block using a new key in each iteration, along with using the same original experiment scheme we will calculate the upper bounds and lower bounds cross correlation for each key results and gather a single upper and lower bounds result for each key and plot them in one diagram.

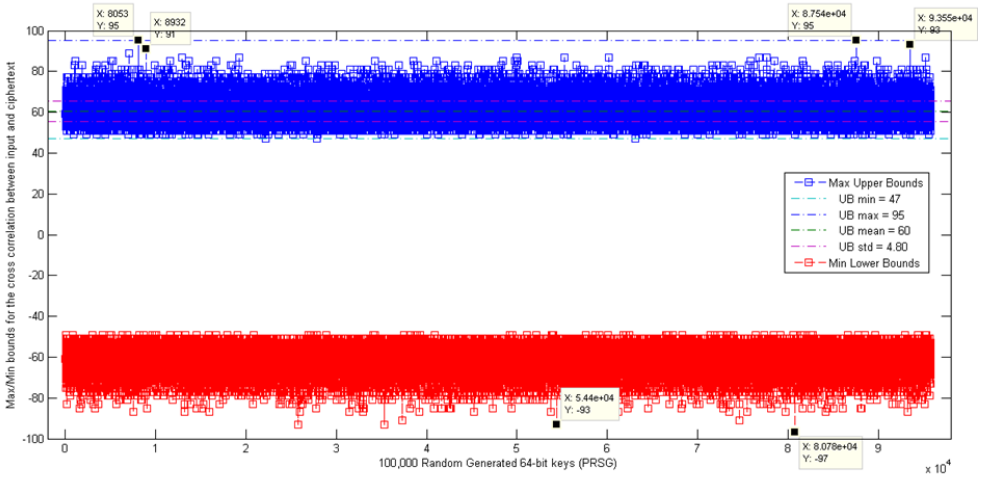

Fig (6) – Cross correlation between input and ciphertext with 100,000 randomly generated 64-bit keys. Calculation time was [5hrs 23min]

After normalizing a subset of 50,000 elements of the cross correlation, the following result has been acquired.

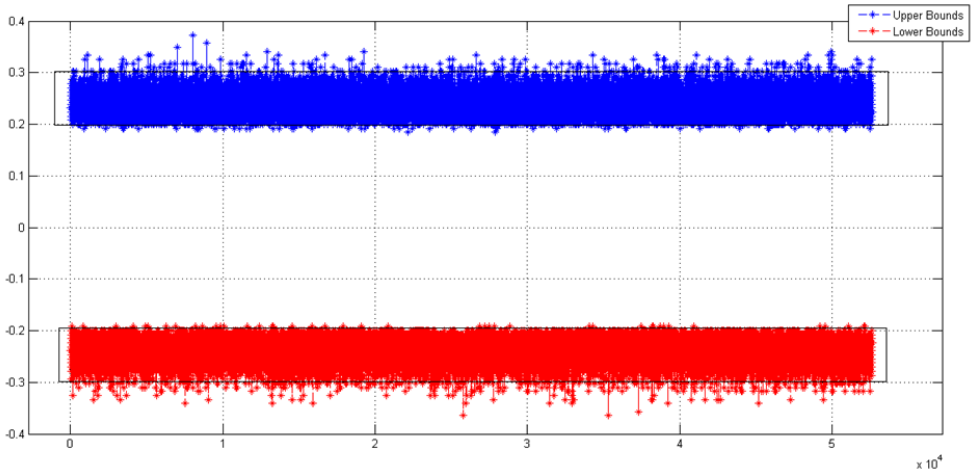

Fig (7) – Normalized subset of 52,000 samples

Comparing the results from figure 7 & 4, it is clear that the SIMON block cipher [P, K, C] has less diffusion and more correlation values in the up-normal region [0.3,0.4] & [-0.3, -0.4]; this effect might be caused by bit bias using specific weak keys; however, nothing can be proved without further research and study. The worst cross correlation found was 95. See Table 2 & 3 for more details about the maximum correlation values and their corresponding keys.

- The 4 keys that generated Maximum Upper Bounds > 90

Keys (Hex)	(Key) Binary Format	Upper bound
0x162E3DACFD30D0D5	0001011000101110001111011010110011111101001100001101000011010101	95
0x328EAB317F66B9E8	0011001010001110101010110011000101111111011001101011100111101000	95
0x4B812C5E1DD35E2A	0100101110000001001011000101111000011101110100110101111000101010	93
0x5547BDBBFBF0266F	0101010101000111101111011011011111110111111000000100110011011111	91

Table 2

- The 4 keys that generated Minimum Lower Bounds < -92

Keys (Hex)	(Key) Binary Format	Upper bound
0xA5DDA13B1BD6295C	1010010111011101101000010011101100011011110101100010100101011100	-93
0xDCCA36B9CE83C845	1101110011001010001101101011100111001110100000111100100001000101	-93
0x0749F1585BAB7E99	0000011101001001111100010101100001011011101010110111111010011001	-93
0x2EC5D2558354DE6D	0100000000101110110001011101001001010101100000110101010011011110	-97

Table 3

ii. Conclusion & Future Work:

In conclusion, for a light weight block cipher like SIMON the cross correlation evaluation conducted has shown that it has given shifted correlation results that were below the expected optimal diffusion region derived from sub-experiment (1), by using a certain set of keys. However, there was no clear reason been elicited! Furthermore, alleged set of weak keys has been obtained for future examination, where it may help in unfolding differential attack on the Fiestel round function?

Because SIMON is a lightweight block cipher, possibly it would be more susceptible to side channel attacks. Especially if it was implemented in constrained devices, such as RFID and EPC tags.

It is hard to rely on correlation solely to find a weakness in a specific block cipher, nevertheless, this case study arises many questions and variations of possible exploits, and one of the outstanding speculative questions was: what if we can contrive a formula that can determine an approximation for the self-cross-correlation distribution for variable length binary sequence X ? For instance if we have a binary sequence X with length of 2, we can easily contrive the self-cross-correlation distribution by calculating the non-identical and non-repeated pair permutation of X to be cross correlated in the following manner:

$$l(X) = 2 \rightarrow Number\ of\ bits$$
$$X = \{0,1,2,3\} = \{00,01,10,11\} \rightarrow Sample\ space$$

$$Combination\binom{2^n}{k} = \frac{2^n!}{k!\,(2^n-k)!}, \forall\, k = 2\ for\ non\ repeated\ combination\ of\ pairs$$

$$s = Combination\binom{4}{2} = \frac{4!}{2!\,(2)!} = 6$$

$$So\ we\ will\ have\ 6\ nonrepeated\ \&\ 7\ nonidentical\ pair\ combinations\ of\ X$$

$$X_{pairs} = \{[0,1],[0,2],[0,3],[1,2],[1,3],[2,3]\}$$

$$C_k = Correlation\left(X_{i,1}, X_{i,2}\right), \forall\, i = 1, \dots, s$$

$$C = \{-1, 0, -2, 0, 2, 0, 0\}$$

$$C_{Normalized} = \frac{C}{l(X)} = \frac{C}{2} = \{-\frac{1}{2}, 0, -1, 0, 1, 0, 0\}$$

So according to the proposed method we can determine the self-cross-correlation distribution for X with length equal to 2-bits from $C_{Normalized}$ data set and so forth for (3, 4, 5, 6,….., n) number of bit length. Now what if we can approximate the cross correlation distribution probability model for binary variable X with particular number of bits, for instance 255 bits? Well, first of all the combination equation has exponential time complexity, which make it computationally expensive to generate even a 14 bits self-cross-correlation distribution, as it will generate almost 8 million cross correlation values to be stored and represented in a histogram. However, a quick proof of concept test has been conducted to find the distribution of $C_{Normalized}$ for the first 10 binary variables $X^{1,\dots,10}$ [1 to 10] bits has been calculated and projected in figure (8).

Notice by observing figure (8), we can clearly see that all of the normalized correlation band frequencies are exponentially increasing in proportion of the bit length of the binary sequence X; further notice is that we should divide the distribution of binary variables into even and odd length values, so that they can be modeled and studied independently. For instance if the band [0.8, 1] is been selected for even number of bits in the right side, its frequency value will be increased as follows:

$$Band[0.8, 1]_{even} = [1, 16, 161, 949, 4970, \dots..]$$
$$Using\ exponential\ curve\ fitting\ application, we\ find\ that\ f(x)\ is\ equal\ to$$
$$f(x) = 1.205 * e^{1.665x}$$

Assuming that this exponential fitting can be generalized for n-bit even length of binary variables, we can predict how many cross correlation values would be in this specific band for variable X with length of 254-bit as follow:

$$Band(0.8,1) \rightarrow f(254) = 5.6036 * 10^{183} \rightarrow eq(5)$$

Note that if we want to extract crucial and sensitive information from this particular band of $[0.8, 1]$ for a block cipher with 254 bit plaintext size, we have tightened the search space from 2^{254} to almost $2^{188} \approx 5.6036 * 10^{183}$.

Prospect extended attacks:

1. Differential cryptanalysis
    a. Impossible differential cryptanalysis
    b. Boomerang attack
2. Weak keys attack
3. Linear span attack

Further investigation has been conducted using 8 weak keys extracted above to check whether the reason beyond the high correlation resulted from these weak keys is because of repeated differential words inside the 32 cycle round function. The experiment done by taking the left 16-bit word starting from the second round until the 32rd round, to find that the 8 keys selected have shared 21 identical words. Although by brutes forcing the two words 2^{32} we might find a possible differential vulnerability, but, this experiment could not be validated because of the confined time. Moreover, the 2^{32} iterations could be minimized to 2^{16} using birthday paradox to find a possible exploit by 25% of success rate based on the information sensed so far.

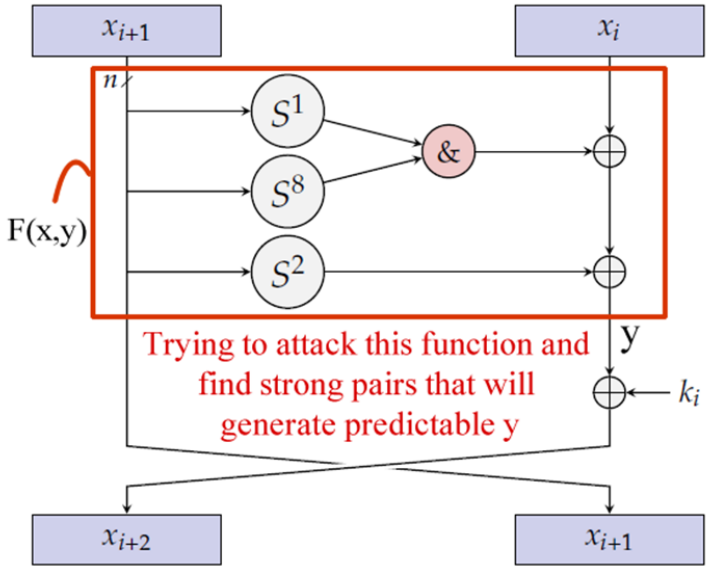

Fig (9) – Unfinished experiment

References:

[1]. Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., & Wingers, L. (2013). The SIMON and SPECK Families of Lightweight Block Ciphers.IACR Cryptology ePrint Archive, 2013, 404.

[2]. Bogdanov, A. (2010). Analysis and Design of Block Cipher Constructions. Europäischer Univ.-Verlag.

Appendix 1: (Experiment Results**)**

Key[0] = [0x0] Key[1] = [0x0] Key[2] = [0x0] Key[3] = [0x8000] Key[4] = [0xe7fd] Key[5] = [0x9d7d] Key[6] = [0x5585] Key[7] = [0x9416] Key[8] = [0xf000] Key[9] = [0x8cc6] Key[10] = [0xcd31] Key[11] = [0xc69f] Key[12] = [0xc1e3] Key[13] = [0x1cb1] Key[14] = [0x2541] Key[15] = [0x2e8d] Key[16] = [0xdbcf] Key[17] = [0x5268] Key[18] = [0xec1d] Key[19] = [0x841a] Key[20] = [0xa7ad] Key[21] = [0xd909] Key[22] = [0x7346] Key[23] = [0xa6c1] Key[24] = [0x4368] Key[25] = [0xe04b] Key[26] = [0x3b16] Key[27] = [0x9f33] Key[28] = [0xe6ee] Key[29] = [0xb98] Key[30] = [0x158a] Key[31] = [0x16bf] Key[32] = [0x4fb] Key[33] = [0x3bfa] Key[34] = [0x9356] Key[35] = [0xd49b] Key[36] = [0x1adb] Key[37] = [0xcc0c] Key[38] = [0x73c] Key[39] = [0xfc44] Key[40] = [0x9f61] Key[41] = [0x1d49] Key[42] = [0xc8db] Key[43] = [0x96ff] Key[44] = [0xf8c0] Key[45] = [0xde96] Key[46] = [0xdc9c] Key[47] = [0xbbf9] Key[48] = [0x1a20] Key[49] = [0x915e] Key[50] = [0x7e5a] Key[51] = [0xbb1b] Key[52] = [0xf0fe] Key[53] = [0x1ec4] Key[54] = [0x2504] Key[55] = [0xa96] Key[56] = [0xbf5f]	Key[57] = [0xdaa0] Key[58] = [0xc3da] Key[59] = [0x61dd] Key[60] = [0x8d74] Key[61] = [0x5e12] Key[62] = [0x8336] Key[63] = [0xedbb] Key[64] = [0xc0fe] Key[65] = [0x5753] Key[66] = [0x3832] Key[67] = [0xd642] Key[68] = [0xb455] Key[69] = [0x614a] Key[70] = [0x9091] Key[71] = [0x6cda] Key[72] = [0xf111] Key[73] = [0xe75c] Key[74] = [0x6744] Key[75] = [0x5023] Key[76] = [0xc518] Key[77] = [0xd8b4] Key[78] = [0xb617] Key[79] = [0x85e9] Key[80] = [0x26e9] Key[81] = [0xfce6] Key[82] = [0xbea4] Key[83] = [0x13b7] Key[84] = [0xc8cc] Key[85] = [0xb779] Key[86] = [0x76ac] Key[87] = [0x95e] Key[88] = [0x7a4b] Key[89] = [0xdd93] Key[90] = [0xc2cb] Key[91] = [0xf5b8] Key[92] = [0x2701] Key[93] = [0x3751] Key[94] = [0x87cd] Key[95] = [0xd641] Key[96] = [0x53a8] Key[97] = [0x3c8] Key[98] = [0xc514] Key[99] = [0x8733] Key[100] = [0xe6ec] Key[101] = [0x918] Key[102] = [0xfff0] Key[103] = [0xfd55] Key[104] = [0xf4fb] Key[105] = [0xb73c] Key[106] = [0x5e67] Key[107] = [0x9204] Key[108] = [0x3cc5] Key[109] = [0x4dc0] Key[110] = [0x77f8] Key[111] = [0x46df] Key[112] = [0xb4ae] Key[113] = [0xc3e7]	Key[114] = [0xe9f7] Key[115] = [0xd47a] Key[116] = [0x9e8e] Key[117] = [0x1b2e] Key[118] = [0x8a9b] Key[119] = [0x33b5] Key[120] = [0x8287] Key[121] = [0x237d] Key[122] = [0xa951] Key[123] = [0xa032] Key[124] = [0xb1bd] Key[125] = [0xcc55] Key[126] = [0x43c8] Key[127] = [0xbae8]		

Table (1) – 128 elements of keys

A[0] = [0xf5a53f01]	A[53] = [0xe2e99549]	A[106] = [0x721eef41]	A[159] = [0x2b273abf]	A[212] = [0x97962331]
A[1] = [0xfad29f80]	A[54] = [0x7174caa4]	A[107] = [0xb90f77a0]	A[160] = [0x15939d5f]	A[213] = [0x4bcb1198]
A[2] = [0x7d694fc0]	A[55] = [0xb8ba6552]	A[108] = [0x5c87bbd0]	A[161] = [0xac9ceaf]	A[214] = [0x25e588cc]
A[3] = [0x3eb4a7e0]	A[56] = [0xdc5d32a9]	A[109] = [0xae43dde8]	A[162] = [0x8564e757]	A[215] = [0x12f2c466]
A[4] = [0x1f5a53f0]	A[57] = [0x6e2e9954]	A[110] = [0xd721eef4]	A[163] = [0xc2b273ab]	A[216] = [0x89796233]
A[5] = [0x8fad29f8]	A[58] = [0x37174caa]	A[111] = [0x6b90f77a]	A[164] = [0x615939d5]	A[217] = [0x44bcb119]
A[6] = [0x47d694fc]	A[59] = [0x9b8ba655]	A[112] = [0x35c87bbd]	A[165] = [0xb0ac9cea]	A[218] = [0xa25e588c]
A[7] = [0xa3eb4a7e]	A[60] = [0xcdc5d32a]	A[113] = [0x1ae43dde]	A[166] = [0xd8564e75]	A[219] = [0x512f2c46]
A[8] = [0xd1f5a53f]	A[61] = [0xe6e2e995]	A[114] = [0x8d721eef]	A[167] = [0xec2b273a]	A[220] = [0xa8979623]
A[9] = [0x68fad29f]	A[62] = [0xf37174ca]	A[115] = [0xc6b90f77]	A[168] = [0x7615939d]	A[221] = [0xd44bcb11]
A[10] = [0x347d694f]	A[63] = [0xf9b8ba65]	A[116] = [0xe35c87bb]	A[169] = [0x3b0ac9ce]	A[222] = [0x6a25e588]
A[11] = [0x9a3eb4a7]	A[64] = [0x7cdc5d32]	A[117] = [0xf1ae43dd]	A[170] = [0x1d8564e7]	A[223] = [0xb512f2c4]
A[12] = [0x4d1f5a53]	A[65] = [0xbe6e2e99]	A[118] = [0x78d721ee]	A[171] = [0xec2b273]	A[224] = [0xda897962]
A[13] = [0x268fad29]	A[66] = [0xdf37174c]	A[119] = [0x3c6b90f7]	A[172] = [0x7615939]	A[225] = [0x6d44bcb1]
A[14] = [0x1347d694]	A[67] = [0x6f9b8ba6]	A[120] = [0x9e35c87b]	A[173] = [0x83b0ac9c]	A[226] = [0x36a25e58]
A[15] = [0x9a3eb4a]	A[68] = [0xb7cdc5d3]	A[121] = [0xcf1ae43d]	A[174] = [0xc1d8564e]	A[227] = [0x1b512f2c]
A[16] = [0x84d1f5a5]	A[69] = [0x5be6e2e9]	A[122] = [0x678d721e]	A[175] = [0x60ec2b27]	A[228] = [0xda89796]
A[17] = [0x4268fad2]	A[70] = [0x2df37174]	A[123] = [0xb3c6b90f]	A[176] = [0x30761593]	A[229] = [0x6d44bcb]
A[18] = [0xa1347d69]	A[71] = [0x16f9b8ba]	A[124] = [0xd9e35c87]	A[177] = [0x183b0ac9]	A[230] = [0x36a25e5]
A[19] = [0x509a3eb4]	A[72] = [0xb7cdc5d]	A[125] = [0x6cf1ae43]	A[178] = [0xc1d8564]	A[231] = [0x1b512f2]
A[20] = [0x284d1f5a]	A[73] = [0x5be6e2e]	A[126] = [0xb678d721]	A[179] = [0x860ec2b2]	A[232] = [0x80da8979]
A[21] = [0x14268fad]	A[74] = [0x2df3717]	A[127] = [0xdb3c6b90]	A[180] = [0xc3076159]	A[233] = [0xc06d44bc]
A[22] = [0x8a1347d6]	A[75] = [0x816f9b8b]	A[128] = [0xed9e35c8]	A[181] = [0xe183b0ac]	A[234] = [0xe036a25e]
A[23] = [0x4509a3eb]	A[76] = [0x40b7cdc5]	A[129] = [0xf6cf1ae4]	A[182] = [0x70c1d856]	A[235] = [0xf01b512f]
A[24] = [0x2284d1f5]	A[77] = [0x205be6e2]	A[130] = [0xfb678d72]	A[183] = [0x3860ec2b]	A[236] = [0xf80da897]
A[25] = [0x914268fa]	A[78] = [0x102df371]	A[131] = [0xfdb3c6b9]	A[184] = [0x1c307615]	A[237] = [0xfc06d44b]
A[26] = [0x48a1347d]	A[79] = [0x816f9b8]	A[132] = [0xfed9e35c]	A[185] = [0x8e183b0a]	A[238] = [0x7e036a25]
A[27] = [0x24509a3e]	A[80] = [0x40b7cdc]	A[133] = [0xff6cf1ae]	A[186] = [0xc70c1d85]	A[239] = [0x3f01b512]
A[28] = [0x92284d1f]	A[81] = [0x8205be6e]	A[134] = [0x7fb678d7]	A[187] = [0x63860ec2]	A[240] = [0x9f80da89]
A[29] = [0x4914268f]	A[82] = [0x4102df37]	A[135] = [0xbfdb3c6b]	A[188] = [0x31c30761]	A[241] = [0x4fc06d44]
A[30] = [0xa48a1347]	A[83] = [0xa0816f9b]	A[136] = [0x5fed9e35]	A[189] = [0x98e183b0]	A[242] = [0xa7e036a2]
A[31] = [0x524509a3]	A[84] = [0xd040b7cd]	A[137] = [0xaff6cf1a]	A[190] = [0xcc70c1d8]	A[243] = [0x53f01b51]
A[32] = [0xa92284d1]	A[85] = [0xe8205be6]	A[138] = [0x57fb678d]	A[191] = [0x663860ec]	A[244] = [0x29f80da8]
A[33] = [0x54914268]	A[86] = [0xf4102df3]	A[139] = [0xabfdb3c6]	A[192] = [0x331c3076]	A[245] = [0x94fc06d4]
A[34] = [0xaa48a134]	A[87] = [0x7a0816f9]	A[140] = [0xd5fed9e3]	A[193] = [0x198e183b]	A[246] = [0x4a7e036a]
A[35] = [0x5524509a]	A[88] = [0xbd040b7c]	A[141] = [0xeaff6cf1]	A[194] = [0x8cc70c1d]	A[247] = [0xa53f01b5]
A[36] = [0x2a92284d]	A[89] = [0xde8205be]	A[142] = [0x757fb678]	A[195] = [0x4663860e]	A[248] = [0xd29f80da]
A[37] = [0x95491426]	A[90] = [0xef4102df]	A[143] = [0x3abfdb3c]	A[196] = [0x2331c307]	A[249] = [0x694fc06d]
A[38] = [0xcaa48a13]	A[91] = [0x77a0816f]	A[144] = [0x9d5fed9e]	A[197] = [0x1198e183]	A[250] = [0xb4a7e036]
A[39] = [0x65524509]	A[92] = [0xbbd040b7]	A[145] = [0xceaff6cf]	A[198] = [0x88cc70c1]	A[251] = [0x5a53f01b]
A[40] = [0x32a92284]	A[93] = [0xdde8205b]	A[146] = [0xe757fb67]	A[199] = [0xc4663860]	A[252] = [0xad29f80d]
A[41] = [0x99549142]	A[94] = [0xeef4102d]	A[147] = [0x73abfdb3]	A[200] = [0x62331c30]	A[253] = [0xd694fc06]
A[42] = [0x4caa48a1]	A[95] = [0xf77a0816]	A[148] = [0x39d5fed9]	A[201] = [0xb1198e18]	A[254] = [0xeb4a7e03]
A[43] = [0xa6552450]	A[96] = [0x7bbd040b]	A[149] = [0x9ceaff6c]	A[202] = [0x588cc70c]	
A[44] = [0xd32a9228]	A[97] = [0x3dde8205]	A[150] = [0x4e757fb6]	A[203] = [0x2c466386]	
A[45] = [0xe9954914]	A[98] = [0x1eef4102]	A[151] = [0x273abfdb]	A[204] = [0x962331c3]	
A[46] = [0x74caa48a]	A[99] = [0xf77a081]	A[152] = [0x939d5fed]	A[205] = [0xcb1198e1]	
A[47] = [0xba655245]	A[100] = [0x87bbd040]	A[153] = [0xc9ceaff6]	A[206] = [0xe588cc70]	
A[48] = [0x5d32a922]	A[101] = [0x43dde820]	A[154] = [0x64e757fb]	A[207] = [0xf2c46638]	
A[49] = [0x2e995491]	A[102] = [0x21eef410]	A[155] = [0xb273abfd]	A[208] = [0x7962331c]	
A[50] = [0x174caa48]	A[103] = [0x90f77a08]	A[156] = [0x5939d5fe]	A[209] = [0xbcb1198e]	
A[51] = [0x8ba65524]	A[104] = [0xc87bbd04]	A[157] = [0xac9ceaff]	A[210] = [0x5e588cc7]	
A[52] = [0xc5d32a92]	A[105] = [0xe43dde82]	A[158] = [0x564e757f]	A[211] = [0x2f2c4663]	

Table (2) – [255x32] bit input vector

S[0] = [0xaed087a9]	S[53] = [0x7d895fae]	S[106] = [0x5306735d]	S[159] = [0x208f8a9a]	S[212] = [0xb7cd5990]
S[1] = [0x8a992db3]	S[54] = [0x3ef88bd6]	S[107] = [0xeaf352ed]	S[160] = [0xd125913]	S[213] = [0x7605eeb2]
S[2] = [0x9ff21273]	S[55] = [0x391b61a3]	S[108] = [0xf858781e]	S[161] = [0x25cacdbf]	S[214] = [0x8499ab39]
S[3] = [0x27d0551d]	S[56] = [0xf51b0d26]	S[109] = [0x172879f2]	S[162] = [0x859faa09]	S[215] = [0x4257e75b]
S[4] = [0x9c9d0cc7]	S[57] = [0x2707cf65]	S[110] = [0xb6e4fc67]	S[163] = [0x91ecd8c3]	S[216] = [0x584d47d4]
S[5] = [0xc0192016]	S[58] = [0x3e7e586]	S[111] = [0xdb31f94e]	S[164] = [0x30ba6d6e]	S[217] = [0x5f1437ed]
S[6] = [0x2b2dcb31]	S[59] = [0xc2c28e6c]	S[112] = [0x828aa077]	S[165] = [0x56d8882b]	S[218] = [0x6b392ec5]
S[7] = [0xe302d4ad]	S[60] = [0x52baec5b]	S[113] = [0xc2e1d738]	S[166] = [0xa47d4c73]	S[219] = [0x39843aff]
S[8] = [0x9b9fbed9]	S[61] = [0x388573b7]	S[114] = [0x511b8214]	S[167] = [0xf79ff84f]	S[220] = [0x959c9f69]
S[9] = [0x1729592f]	S[62] = [0x2a41a28b]	S[115] = [0x9291b116]	S[168] = [0x3b62ffd1]	S[221] = [0x6c2cecdf]
S[10] = [0x1f0173af]	S[63] = [0x2cdce6]	S[116] = [0x926a5731]	S[169] = [0x477af55]	S[222] = [0x1feff62]
S[11] = [0x851ab670]	S[64] = [0x1c4ea5dc]	S[117] = [0xe8173715]	S[170] = [0xdcf4695]	S[223] = [0xe61fbda]
S[12] = [0x3f6efd95]	S[65] = [0x830b84af]	S[118] = [0xf7ee3ee1]	S[171] = [0xe2fb85ff]	S[224] = [0x4f75a7f6]
S[13] = [0x6ed8efeb]	S[66] = [0x1dc7f4d8]	S[119] = [0x643c0da9]	S[172] = [0xec73795c]	S[225] = [0xfa0ab6f]
S[14] = [0xfdb8282a]	S[67] = [0x7e0a1248]	S[120] = [0xa97a9624]	S[173] = [0x9a450bec]	S[226] = [0xa1a0d647]
S[15] = [0x2d9a8728]	S[68] = [0x3f33d0f9]	S[121] = [0xb45f5ab5]	S[174] = [0x9279c92f]	S[227] = [0x591b95bc]
S[16] = [0x97c06452]	S[69] = [0x99eb5dfa]	S[122] = [0xf426db92]	S[175] = [0x958ecda2]	S[228] = [0x88eeba1b]
S[17] = [0x4a29bf1c]	S[70] = [0x4134fa28]	S[123] = [0xa0935ca1]	S[176] = [0x3b5c5094]	S[229] = [0x87e251fb]
S[18] = [0x80cc7675]	S[71] = [0x205741f]	S[124] = [0x69b7bfce]	S[177] = [0xe702843f]	S[230] = [0x98228b06]
S[19] = [0x72fb898b]	S[72] = [0x5c232f58]	S[125] = [0x2eaeab2f]	S[178] = [0x4f5b1c55]	S[231] = [0x50e833b1]
S[20] = [0x4af5319b]	S[73] = [0x37bbeb73]	S[126] = [0xe30702fc]	S[179] = [0xba7a3f0c]	S[232] = [0xa99ddf8d]
S[21] = [0x8c3c5b5f]	S[74] = [0x214fdf2b]	S[127] = [0xfc2a2563]	S[180] = [0x66a6cfd8]	S[233] = [0xc3a5c592]
S[22] = [0xafd7c49e]	S[75] = [0x8015d70d]	S[128] = [0xbbdf5b0e]	S[181] = [0x7dbf47af]	S[234] = [0xe6f2e6f4]
S[23] = [0xc09a293e]	S[76] = [0x7349681d]	S[129] = [0xc53cd99f]	S[182] = [0x3fde7ec4]	S[235] = [0xf9bb2ea7]
S[24] = [0x7594b578]	S[77] = [0x36528065]	S[130] = [0xd442d2af]	S[183] = [0x9758297f]	S[236] = [0xa931950e]
S[25] = [0xfac74c93]	S[78] = [0x634719f7]	S[131] = [0x7bf9f268]	S[184] = [0x6b1e0ff5]	S[237] = [0xb46c9ed4]
S[26] = [0x876674b6]	S[79] = [0xbd0e2ba1]	S[132] = [0x8b69ffec]	S[185] = [0x958b6a8b]	S[238] = [0x4d5e3fa3]
S[27] = [0x1d057a88]	S[80] = [0x3fa67f9b]	S[133] = [0xc62c812b]	S[186] = [0x241ba6ca]	S[239] = [0x62bfe427]
S[28] = [0xbf1878e7]	S[81] = [0x4817c080]	S[134] = [0x47de32d8]	S[187] = [0x56697c3b]	S[240] = [0xdeb313f9]
S[29] = [0x1bff8cab]	S[82] = [0xd91543d4]	S[135] = [0xf9d4d5b0]	S[188] = [0x5bf8a7af]	S[241] = [0xba4a4d0d]
S[30] = [0xb3c561f9]	S[83] = [0x51800a19]	S[136] = [0xe98bf682]	S[189] = [0xe24bc96c]	S[242] = [0xf6e21cda]
S[31] = [0x2171f9f9]	S[84] = [0x602c98b0]	S[137] = [0x98c68ea8]	S[190] = [0xe82d3ede]	S[243] = [0x1ca0a004]
S[32] = [0x7459af64]	S[85] = [0x4432158c]	S[138] = [0xd9a13ff5]	S[191] = [0x84c4fb15]	S[244] = [0xf6dc8448]
S[33] = [0xda03f7d7]	S[86] = [0x1d55be61]	S[139] = [0x440b2bd]	S[192] = [0x9658f6ba]	S[245] = [0xfb7de31f]
S[34] = [0xebaaac55]	S[87] = [0x837fbc5b]	S[140] = [0xbb7c4037]	S[193] = [0xcf92d369]	S[246] = [0xc662f104]
S[35] = [0x86744536]	S[88] = [0x1cbeac6e]	S[141] = [0x5a168dd1]	S[194] = [0x9da586d2]	S[247] = [0x3baaa255]
S[36] = [0x25445403]	S[89] = [0x9c9a827b]	S[142] = [0x759797e7]	S[195] = [0xf97b2d72]	S[248] = [0x37b362f8]
S[37] = [0xd9acf44c]	S[90] = [0x4d77e810]	S[143] = [0x399e8cfe]	S[196] = [0x99ea0814]	S[249] = [0x90138379]
S[38] = [0x1526c106]	S[91] = [0x4f603dda]	S[144] = [0x6c496c1b]	S[197] = [0xb95c5144]	S[250] = [0xb68e5145]
S[39] = [0x6045f736]	S[92] = [0xda2c0c3]	S[145] = [0x6e1c3027]	S[198] = [0x1acf0a26]	S[251] = [0x8d0dc747]
S[40] = [0x4259f44e]	S[93] = [0xda3045f2]	S[146] = [0x2651ac5a]	S[199] = [0xab721c80]	S[252] = [0x5a86903c]
S[41] = [0x971b92a2]	S[94] = [0x9645033b]	S[147] = [0x1893a981]	S[200] = [0x84faaac2]	S[253] = [0x5ef62654]
S[42] = [0x933a261b]	S[95] = [0xaf07af0f]	S[148] = [0x94422f07]	S[201] = [0x8ba21706]	S[254] = [0x2826cbe8]
S[43] = [0x9f501128]	S[96] = [0x5df9220d]	S[149] = [0x8ca905c2]	S[202] = [0x3a50b8f9]	
S[44] = [0x4058520c]	S[97] = [0xe698c251]	S[150] = [0x42271ef0]	S[203] = [0xa9e95799]	
S[45] = [0x6c2f0ae1]	S[98] = [0x9801bc1c]	S[151] = [0xee2683ea]	S[204] = [0x6580dd59]	
S[46] = [0x75cd6d38]	S[99] = [0xa88514f2]	S[152] = [0xb971bc04]	S[205] = [0xb183090c]	
S[47] = [0x3e2db655]	S[100] = [0x69d342ee]	S[153] = [0x51234c26]	S[206] = [0xb577cd9d]	
S[48] = [0x88cc1ef0]	S[101] = [0x64ae8051]	S[154] = [0x1cc1d4a2]	S[207] = [0x38698bff]	
S[49] = [0xc0eae7ad]	S[102] = [0x88d9ed9b]	S[155] = [0x3a8d0fd]	S[208] = [0x6ff33f50]	
S[50] = [0xe2aa8e56]	S[103] = [0x7071dffc]	S[156] = [0xb5c3bc18]	S[209] = [0x2487cdf7]	
S[51] = [0xd774c45d]	S[104] = [0x5de267cb]	S[157] = [0xbd7121cf]	S[210] = [0xe0c93364]	
S[52] = [0xace12b31]	S[105] = [0xf7c150bf]	S[158] = [0xa0e00f62]	S[211] = [0x20df033a]	

Table (3) – [255 x 32] bit ciphertext vectors

Appendix 2 (Source Code):

```
/*
 * File:   main.c
 * Author: Abdurrahman Elbuni
 * Supervised by: Professor Guang Gong
 * Created on February 20, 2015, 11:39 AM
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define N 16
#define T 4

#define u32 unsigned int
#define u16 unsigned short
#define u8 unsigned char

// x^5+x^4+x^2+x+1
#define LFSR_5_POLY 0x17
// x^8+x^7+x^2+x+1
#define LFSR_8_POLY 0x87

#define CROSSCORRELATIOM_SIZE 255

// Constants
// C=> 2^16-4 (1111111111111100b))
#define C 0xFFFC

// Function prototypes section
void GenerateLFSR(u8 poly,u8 stages, u8 seed, u8* seqArray);
u32 NumberOfSetBits(int i);
u16 CircularLeftShift(u16 word, u8 j);
u16 CircularRightShift(u16 word, u8 j);
void SimonEncryption(u16 *plainText, u16 *cipherText);
void KeyGeneration(u8 *LFSRSeq, u16 *KeySeq);
void CrossCorrelation(u8* x,u8* y, int* correlationResult);

u16 KeySeq[128] = {0x00, 0x00, 0x00,0x8000 };

// Main entry
int main(int argc, char** argv) {

    u32 i, j, k, testMask;
    u16 registers[2];
    u32 initialSeed = 0x1F; // Initial seed declared as 11111
    u32 initialSeed8 = 0x01; // Initial seed declared as 00000001

    // {LSB, 0, 0, MSB}
    u16 plainText[2] = {0};
    u8 LFSRSeq5[32] = {0};
    u8 LFSRSeq8[255] = {0};

    u32 Yi[255] = {0};
    u32 Si[255] = {0};
    u8 SiTestVector [32][255] = {0};
    u8 YiTestVector [32][255] = {0};
    u8 *SiPointer;
    u8 *YiPointer;
```

```
// Store cross correlation result for 32x16 (Message x Cipher).
int crossCorrelation[32][16][CROSSCORRELATIOM_SIZE] = {0};
int crossUpperLower[32][16][2] = {0};
int *correlationPointer;

// Initialize key array
printf("Initial 5 stages m-sequence:\n");
GenerateLFSR(LFSR_5_POLY, 5, initialSeed, LFSRSeq5);
printf("\n");

// Circular shift test
// printf("%x\n\n",CircularRightShift(0x8200,1));

// ************ Part (1) *************
// ****** Key Scheduling ******
KeyGeneration(LFSRSeq5, KeySeq);

// ****** Encryption ******
SimonEncryption(plainText, registers);
printf("%x %x",registers[0],registers[1]);

// ************ Part (2) *************
// ***********************************
printf("\nInitial 8 stages m-sequence:\n");
GenerateLFSR(LFSR_8_POLY, 8, initialSeed8, LFSRSeq8);
printf("\n");

for(i=0;i<255;i++)
{
    for(j=32;j>0;j--)
    {
        Yi[i] = Yi[i] << 1;
        Yi[i] = Yi[i] | LFSRSeq8[(i+j-1)%255];
    }
    // temporary variable to hold current 32 bit input (16bit Concatenation)
    plainText[0]=Yi[i]>>16;
    plainText[1]=Yi[i]&0x0000ffff;
    // Apply SIMON Encryption
    SimonEncryption(plainText, registers);
    // Cipher text stored inside 32bit cipher array (16bit Concatenation))
    Si[i] = (registers[0]<<16);
    Si[i] = (registers[1]|Si[i]);
}

// Extract the first 16bit of cipher and 32bit of input for cross correlation
for(i=0;i<255;i++)
{
    testMask = 0x00000001;
    for(j=0;j<1;j++)
    {
        YiTestVector[j][i] = ((Yi[i]&testMask)>0)?1:0;
        testMask = testMask >> 1;
    }
}
for(i=0;i<255;i++)
{
    testMask = 0x80000000;
    for(j=0;j<32;j++)
    {
        SiTestVector[j][i] = ((Si[i]&testMask)>0)?1:0;
        testMask = testMask >> 1;
    }
}
```

```
    for(i=0;i<32;i++)
    {
        for(j=0;j<1;j++)
        {
            YiPointer = &YiTestVector[j][0];
            SiPointer = &SiTestVector[i][0];
            correlationPointer = &crossCorrelation[j][i][0];
            CrossCorrelation(SiPointer,YiPointer,correlationPointer);
        }
    }

    for(i=0;i<32;i++)
    {
        for(j=0;j<1;j++)
        {
            for(k=0;k<CROSSCORRELATIOM_SIZE;k++){
                //printf("%d,",crossCorrelation[i][j]);
                if(crossUpperLower[j][i][0]<crossCorrelation[j][i][k])
                {
                    crossUpperLower[j][i][0] = crossCorrelation[j][i][k];
                }
                if(crossUpperLower[j][i][1]>crossCorrelation[j][i][k])
                {
                    crossUpperLower[j][i][1] = crossCorrelation[j][i][k];
                }
            }

            //Print results
            printf("[MessageIndex,CipherIndex]=[%d,%d]\n",j,i);
            printf("[Message,Cipher]=[");
            for(k=0;k<CROSSCORRELATIOM_SIZE;k++){
                printf("%d",YiTestVector[j][k]);
            }
            printf(",\n");
            for(k=0;k<CROSSCORRELATIOM_SIZE;k++){
                printf("%d",SiTestVector[i][k]);
            }
            printf("]\n");

printf("[upper,lower]=[%d,%d]",crossUpperLower[j][i][0],crossUpperLower[j][i][1]);
            printf(")\n");
        }
    }
    for(i=0;i<255;i++)
    {
        printf("S[%d] = [%x]\n",i,Si[i]);
    }
    for(i=0;i<255;i++)
    {
        printf("A[%d] = [%x]\n",i,Yi[i]);
    }

    return (EXIT_SUCCESS);
}
```

```
// Cross correlation for (255 bit) (x) with (255 bit) (y)). correlationResult will
have the result
// passed by reference (VALIDATED))
void CrossCorrelation(u8* x,u8* y, int* correlationResult)
{
    int i,j, sum=0;
    for(i=0;i<CROSSCORRELATIOM_SIZE;i++)
    {
        sum=0;
        for(j=0;j<CROSSCORRELATIOM_SIZE;j++)
        {
            if(x[j]==y[(j+i)%CROSSCORRELATIOM_SIZE])
            {
                sum = sum + 1;
            }else{
                sum = sum - 1;
            }

        }
        correlationResult[i] = sum;
        //printf("%d,",sum);
    }
}

// This function will generate and assign the m-sequence to a global array according
to the initial seed
void GenerateLFSR(u8 poly,u8 stages, u8 seed, u8* seqArray)
{
    int i=0;
    // This variable will be added to the sequence in case there is a carry bit
    u8 carryBit = (1 << stages - 1);
    // This variable will generate the bit mask of (00011111) in case LFSR dimension
was 5 and so forth.
    u8 initiBitSeq = seed;
    u32 tempR; // Hold in previous state value of an LFSR

    for(i;i<((int)floor((pow(2,stages))-1));i++)
    {
        tempR = initiBitSeq & poly;
        tempR = NumberOfSetBits(tempR);
        // Shift the sequence 1 bit to the right
        seqArray[i] = initiBitSeq & 0x01;
        initiBitSeq = initiBitSeq >> 1;
        // Check if we need to add carry bit to the left of initiBitSeq
        // If least significant bit is on, this mean we have an odd tab result
        // which lead to add a carry
        if(tempR & 0x00000001)
        {
            initiBitSeq = initiBitSeq | carryBit;
        }
        printf("%d", seqArray[i]);
    }
    printf("\n%d",i);
}

// Calculate number of 1's in a sequence
u32 NumberOfSetBits(int i)
{
     i = i - ((i >> 1) & 0x55555555);
     i = (i & 0x33333333) + ((i >> 2) & 0x33333333);
     return (((i + (i >> 4)) & 0x0F0F0F0F) * 0x01010101) >> 24;
}
```

```
// Left circular 16bit shift function by j bits
u16 CircularLeftShift(u16 word, u8 j)
{
    return (word << j) | (word >> (16 - j));
}

// Right circular 16bit shift function by j bits
u16 CircularRightShift(u16 word, u8 j)
{
    return (word >> j) | (word << (16 - j));
}

// SIMON32/64 - Pass the 16bit width plain text and key for encryption
void SimonEncryption(u16 *plainText, u16 *cipherText)
{
    int i;
    u16 tmp;
    // Initialize registers values
    for(i=0;i<2;i++)
    {
        cipherText[i%2] = plainText[i];
    }

    // Start the 32 rounds of Simon encryption
    for(i=0;i<32;i++)
    {
        tmp = cipherText[0]; // ==> 1
        cipherText[0] =
((CircularLeftShift(tmp,1)&CircularLeftShift(tmp,8))^cipherText[1])^CircularLeftShift(
tmp,2)^KeySeq[i]; // x[1]
        cipherText[1] = tmp; // x[0]
    }
}

// Key Scheduling:
// Function to generate keys from initial seeds and m-sequence
void KeyGeneration(u8 *LFSRSeq, u16 *KeySeq)
{
    u16 tmp, i, I;

    // Key initialization
    for(i=4;i<128;i++)
    {
        tmp = (C ^ (u16)LFSRSeq[(i-4)%31]);
        I = CircularRightShift(KeySeq[(i-1)],3) ^ KeySeq[(i-3)]; // Identity map
        KeySeq[i] =  (KeySeq[i-4] ^ I) ^ (CircularRightShift(I,1)) ^ (C ^
(u16)LFSRSeq[(i-4)%31]);
    }
}
```