Rectangle Cipher

Ruchit Prakash Saxena[1], Anubh Sanoj Gupta[2] and Lavish[3]

[1] IIT Bhilai, Raipur, India, ruchitp@iitbhilai.ac.in (11941040)
[2] IIT Bhilai, Raipur, India, anubhs@iitbhilai.ac.in (11940150)
[3] IIT Bhilai, Raipur, India, lavishg@iitbhilai.ac.in (11940640)

Abstract.
Rectangle Cipher is a bit-slice-styled block cipher, which is lightweight and based on SP-Network, where the bit-slice implementation of the cipher causes its lightweight and fast executions. It attains a very fast and competitive software performance as well as a very low-cost hardware performance. Sixteen 4x4 S-Boxes are present in the substitution layer in parallel and 3 rotations in the permutation layer. Rectangle provides great hardware as well as software performance. So that it offers applications enough flexibility.

Rectangle cipher allows very efficient hardware performance. Because of the cipher's bit-slice implementation, it attains a fast software performance among all the existing lightweight ciphers.

Rectangle attains a better security performance because of the cipher's Substitution box careful selection and the permutation layer's asymmetric design. The deep analysis of the security reveals that out of 25, a maximum of 18-rounds can be attacked.

Keywords: bit-slice implementation, lightweight block cipher, software efficiency, security performance

1 Introduction

Nowadays, many tiny embedded devices like smart cards, RFIDs, sensor nodes, etc are used in many applications on a large scale, which are generally distinguished by strong cost constraints like energy, area, power in the aspect of hardware and small and optimized code, low memory space in the aspect of the software. Cryptographic Protection is also necessary meanwhile. So that many new lightweight ciphers have been introduced for supplying better security at a lower cost, such as Present, SIMON, Hummingbird, KLEIN, DESL/DESX/DESXL, LBlock, Katan, Piccolo, TWINE, LED, SPECK, and so on.

At CHES'2007, the Present cipher was introduced and became popular due to its better hardware performance, simplicity, and security. It utilizes a bit of permutation in the diffusion layer so that it has a very good hardware performance. Some lightweight ciphers attain a low hardware area while remaining unsuccessful in achieving good software performance, which includes Present, Katan, and Hummingbird. Then at CHES'2011, the LED cipher is introduced where their creators claim that their cipher is not only optimal in hardware performance but also decent in software performance.

However, 2nd-generation lightweight ciphers can be improved by analyzing the idea, success, and exclusion of the 1st-generation proposals. So that the Serpent block cipher was introduced with the new bit-slice technique causing an increase in the software speed of DES, which is based on the SP-network. It is a 128-bit block cipher, which contains thirty-two 4x4 S-Boxes in the substitution layer. SHA-3, Trivium, JH are 5 other ciphers, which were also profited by the bit-slice mechanism for their software performance. It is worth seeing that these not only perform well in hardware but also in software.

Also, a bit-slice execution is protected from implementation attacks like- cache and timing attacks in contrast with a table-based execution. Though the blueprint goal of all the above ciphers is not lightweight, and for a committed bit-slice lightweight cipher, there is enough field for improvement.

Therefore Rectangle cipher is introduced, based on SP-network which utilizes the bit-slice mechanism. Hence it attains a very fast and competitive software performance as well as a very low-cost hardware performance. Sixteen 4x4 S-Boxes are present in the substitution layer in parallel and 3 rotations in the permutation layer. The top 3 pros of the cipher include very hardware friendly, very fast software speed, and very good security.

Rectangle cipher allows very efficient hardware performance. Because of the cipher's bit-slice implementation, it attains a very fast software speed among all the lightweight block ciphers.

Rectangle attains a better security performance because of the cipher's Substitution box careful selection and the permutation layer's asymmetric design. The deep analysis of the security reveals that out of 25, the maximum of 18-rounds can be attacked. The remaining 7-rounds are for security purposes.

2 Rectangle Cipher Implementation

Rectangle block cipher is a cipher with the block length equal to sixty-four bits, which can be seen as a 4x16 rectangle array. And the key length is either 80 or 128 bits. It is a twenty-five round substitution-permutation cipher, where each of the twenty-five rounds consists of the three steps- AddRoundkey(ARK), SubColumn(SC), ShiftRow(SR). At final AddRoundKey is applied to get the final cipher block. Its encryption algorithm consists of twenty-five rounds and a final ARK.

The pseudocode for the Rectangle cipher is given below:-

```
    GenerateRoundKeys(state):
.       for i = 0 to 24 do:
.           ARK(state, Ki)
.           SC(state)
.           SR(state)
.       ARK(state, K25)
.
```

State (Cipher and Subkey State)- Cipher state is a cluster of a 64-bit plaintext or intermediate result or ciphertext as shown in the figure below. A cipher state is nothing but a 4x16 rectangular array of bits, due to which the cipher name was given "Rectangle". Let W indicates the cipher state, such that W = w63|| ... ||w1||w0, where wi denotes 1-bit word. The 1st 16 bits are arranged in row 0 (i.e. w15|| . . . ||w1|| w0), the next 16 bits are arranged in row-1 (i.e. w31|| ... || w17 || w16), and so on. Similarly, a 64-bit subkey can also be seen as a 4×16 rectangular array bits.

$$\begin{bmatrix} w_{15} & \cdots & w_2 & w_1 & w_0 \\ w_{31} & \cdots & w_{18} & w_{17} & w_{16} \\ w_{47} & \cdots & w_{34} & w_{33} & w_{32} \\ w_{63} & \cdots & w_{50} & w_{49} & w_{48} \end{bmatrix} \qquad \begin{bmatrix} a_{0,15} & \cdots & a_{0,2} & a_{0,1} & a_{0,0} \\ a_{1,15} & \cdots & a_{1,2} & a_{1,1} & a_{1,0} \\ a_{2,15} & \cdots & a_{2,2} & a_{2,1} & a_{2,0} \\ a_{3,15} & \cdots & a_{3,2} & a_{3,1} & a_{3,0} \end{bmatrix}$$

Cipher State Subkey State

AddRoundkey (ARK)- This operation is nothing but an easy bitwise xor between the round subkey and the state at the intermediate of the algorithm.

SubColumn (SC)- SubColumn is similar to the Subbytes operation, where the parallel application of S-boxes gets applied to the 4 bits in the same column. This operation is shown below in the figure. If the input of an S-box is Col(j) = a3,j || a2,j || a1,j || a0,j for 0 ≤ j ≤ 15, then the result would be S(Col(j)) = b3,j || b2,j || b1,j || b0,j.

$$\begin{tikzcd}
\begin{pmatrix} a_{0,15} \\ a_{1,15} \\ a_{2,15} \\ a_{3,15} \end{pmatrix} \arrow[d, "S"] & \cdots \arrow[d, phantom, "\cdots"] & \begin{pmatrix} a_{0,2} \\ a_{1,2} \\ a_{2,2} \\ a_{3,2} \end{pmatrix} \arrow[d, "S"] & \begin{pmatrix} a_{0,1} \\ a_{1,1} \\ a_{2,1} \\ a_{3,1} \end{pmatrix} \arrow[d, "S"] & \begin{pmatrix} a_{0,0} \\ a_{1,0} \\ a_{2,0} \\ a_{3,0} \end{pmatrix} \arrow[d, "S"] \\
\begin{pmatrix} b_{0,15} \\ b_{1,15} \\ b_{2,15} \\ b_{3,15} \end{pmatrix} & \cdots & \begin{pmatrix} b_{0,2} \\ b_{1,2} \\ b_{2,2} \\ b_{3,2} \end{pmatrix} & \begin{pmatrix} b_{0,1} \\ b_{1,1} \\ b_{2,1} \\ b_{3,1} \end{pmatrix} & \begin{pmatrix} b_{0,0} \\ b_{1,0} \\ b_{2,0} \\ b_{3,0} \end{pmatrix}
\end{tikzcd}$$

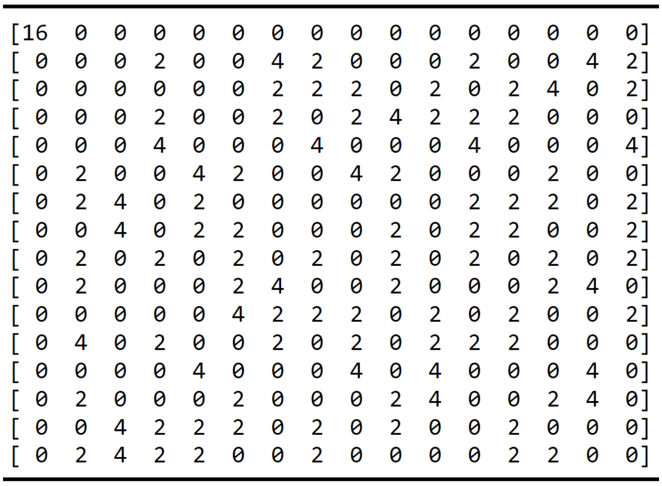
SubColumn Operation on the State Columns

The S-Box used in Rectangle is denoted as S : $F_2^4 \rightarrow F_2^4$, i.e. a 4-bit to 4-bit S-Box in hexadecimal which is given below in the table-

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S(x)$	6	5	C	A	1	E	7	9	B	0	3	D	8	F	4	2

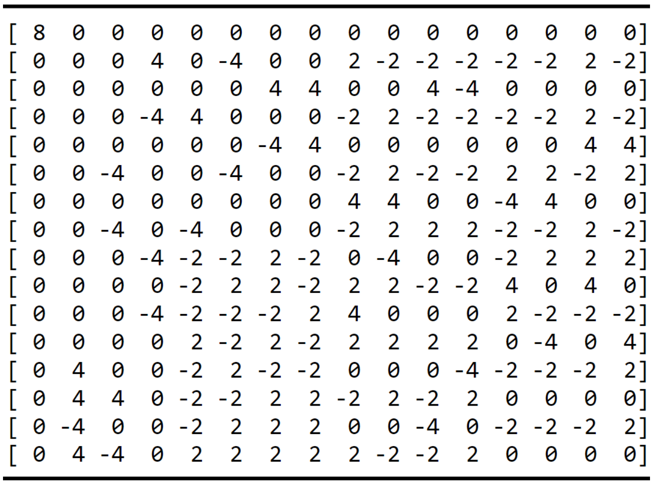
. Rectangle Cipher S-Box

Differentil Distribution Table (DDT) and Linear Approximation Table (LAT) for the above S-Box are shown in the figures below-



```
[16  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  2  0  0  4  2  0  0  0  2  0  0  4  2]
[ 0  0  0  0  0  0  2  2  2  0  2  0  2  4  0  2]
[ 0  0  0  2  0  0  2  0  2  4  2  2  2  0  0  0]
[ 0  0  0  4  0  0  0  4  0  0  0  4  0  0  0  4]
[ 0  2  0  0  4  2  0  0  4  2  0  0  0  2  0  0]
[ 0  2  4  0  2  0  0  0  0  0  0  2  2  2  0  2]
[ 0  0  4  0  2  2  0  0  0  2  0  2  2  0  0  2]
[ 0  2  0  2  0  2  0  2  0  2  0  2  0  2  0  2]
[ 0  2  0  0  0  2  4  0  0  2  0  0  0  2  4  0]
[ 0  0  0  0  0  4  2  2  2  0  2  0  2  0  0  2]
[ 0  4  0  2  0  0  2  0  2  0  2  2  2  0  0  0]
[ 0  0  0  0  4  0  0  0  4  0  4  0  0  0  4  0]
[ 0  2  0  0  0  2  0  0  0  2  4  0  0  2  4  0]
[ 0  0  4  2  2  2  0  2  0  2  0  0  2  0  0  0]
[ 0  2  4  2  2  0  0  2  0  0  0  0  2  2  0  0]
```

DDT for Rectangle cipher S-Box



```
[ 8  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  4  0 -4  0  0  2 -2 -2 -2 -2 -2  2 -2]
[ 0  0  0  0  0  0  4  4  0  0  4 -4  0  0  0  0]
[ 0  0  0 -4  4  0  0  0 -2  2 -2 -2 -2 -2  2 -2]
[ 0  0  0  0  0  0 -4  4  0  0  0  0  0  0  4  4]
[ 0  0 -4  0  0 -4  0  0 -2  2 -2 -2  2  2 -2  2]
[ 0  0  0  0  0  0  0  0  4  4  0  0 -4  4  0  0]
[ 0  0 -4  0 -4  0  0  0 -2  2  2  2 -2 -2  2 -2]
[ 0  0  0 -4 -2 -2  2 -2  0 -4  0  0 -2  2  2  2]
[ 0  0  0  0 -2  2  2 -2  2  2 -2 -2  4  0  4  0]
[ 0  0  0 -4 -2 -2 -2  2  4  0  0  0  2 -2 -2 -2]
[ 0  0  0  0  2 -2  2 -2  2  2  2  2  0 -4  0  4]
[ 0  4  0  0 -2  2 -2 -2  0  0  0 -4 -2 -2 -2  2]
[ 0  4  4  0 -2 -2  2  2 -2  2 -2  2  0  0  0  0]
[ 0 -4  0  0 -2  2  2  2  0  0 -4  0 -2 -2 -2  2]
[ 0  4 -4  0  2  2  2  2  2 -2 -2  2  0  0  0  0]
```

LAT for Rectangle cipher S-Box

ShiftRow (SR)- It is just a left rotation to each row. These rotations are executed over different offsets, like- 0bit for row-0, 1bit for row-1, 12bits for row-2, and 13 bits for row-3. Assume that the left rotation over x bits be indicated by “<<x”, then the operation ShiftRow is shown in the figure below-

<< **0** — (a0,15 ... a0,2 a0,1 a0,0) → (a0,15 ... a0,2 a0,1 a0,0)
<< **1** — (a1,15 ... a1,2 a1,1 a1,0) → (a1,14 ... a1,1 a1,0 a1,15)
<< **12** — (a2,15 ... a2,2 a2,1 a2,0) → (a2,3 ... a2,6 a2,5 a2,4)
<< **13** — (a3,15 ... a3,2 a3,1 a3,0) → (a3,2 ... a3,5 a3,4 a3,3)

Key Schedule- We can have 2 types of keys with different length i.e. 80 or 128 bits.

CASE-1: 80-bit key

Let the user-provided key, in this case, be V, such that V = v79|| ... || v1 || v0 (Since, 80-bit seed key), the key is initially stored in an 80-bit key register and ordered as a 5×16 rectangle array (shown in the figure below).

$$\begin{bmatrix} v_{15} & \cdots & v_2 & v_1 & v_0 \\ v_{31} & \cdots & v_{18} & v_{17} & v_{16} \\ v_{47} & \cdots & v_{34} & v_{33} & v_{32} \\ v_{63} & \cdots & v_{50} & v_{49} & v_{48} \\ v_{79} & \cdots & v_{66} & v_{65} & v_{64} \end{bmatrix} \qquad \begin{bmatrix} \kappa_{0,15} & \cdots & \kappa_{0,2} & \kappa_{0,1} & \kappa_{0,0} \\ \kappa_{1,15} & \cdots & \kappa_{1,2} & \kappa_{1,1} & \kappa_{1,0} \\ \kappa_{2,15} & \cdots & \kappa_{2,2} & \kappa_{2,1} & \kappa_{2,0} \\ \kappa_{3,15} & \cdots & \kappa_{3,2} & \kappa_{3,1} & \kappa_{3,0} \\ \kappa_{4,15} & \cdots & \kappa_{4,2} & \kappa_{4,1} & \kappa_{4,0} \end{bmatrix}$$

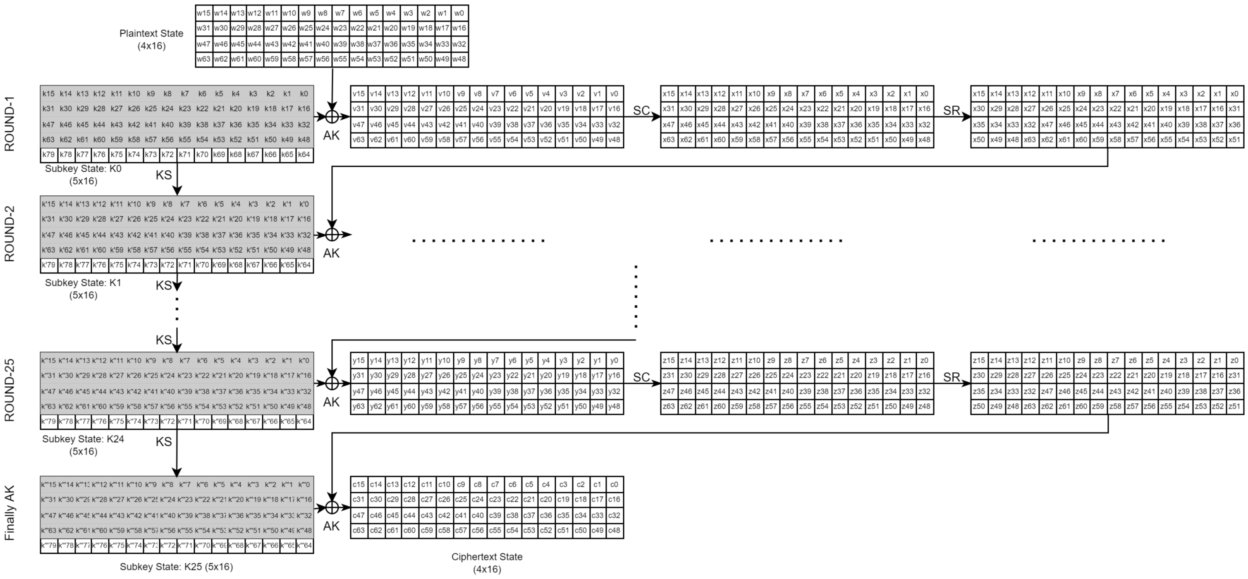
80-bit Key State   Key’s 2D Representation

Suppose Rowi = ki,15|| ... || ki,1|| ki,0 indicate the i-th key register’s row, such that 0 ≤ i ≤ 4, where Rowi is considered to a sixteen bit word. At round i where i belongs to 0 to24, the sixty-four bit round subkey Ki contains the 1st 4 rows of the key register’s contents (as shown in the figure given below), i.e., Ki = Row3 || Row2 || Row1 || Row0. Once the key register Ki found out, the key register gets updated using the following iterations-

1. Using the S-box S, we perform Subcoumn to the partial bits that are intersected at the 4 uppermost rows and the 4 rightmost columns (shaded as grey in the figure given below), i.e., k’3,j || k’2,j || k’1,j || k’0,j := S(k3,j || k2,j || k1,j || k0,j), such that j belongs to (0, 1, 2, 3).

2. Using 1-round generalized Feistel transformation, i.e.,
Row'0 := (Row0 << 8) $\oplus$ Row1,
Row'1 := Row2,
Row'2 := Row3,
Row'3 := (Row3 << 12) $\oplus$ Row4,
Row'4 := Row0

3. Now on the 5-bit key state (i.e. k0,4|| k0,3 || k0,2 || k0,1|| k0,0), we perform a XOR operation with RC[i] (a 5-bit round constant), i.e., k'0,4 || k'0,3 || k'0,2 || k'0,1 || k'0,0 := (k0,4|| k0,3|| k0,2|| k0,1 || k0,0) $\oplus$ RC[i]
Finally, we get K25 using the above iterations on the updated key state. 5-bit LFSR is used to generate the round constants RC[i] for all i belonging to 0 to 24. At each round, the 5 bits (rc4, rc3, rc2, rc1, rc0) are left shifted over 1 bit, with the new value to rc0 which is evaluated as rc4 $\oplus$ rc2. We take RC[0] := 0x1 at the start. Now, using that we can derive all the round constants(for i=0 to 24)- (0X01, 0X02, 0X04, 0X09, 0X12, 0X05, 0X0B, 0X16, 0X0C, 0X19, 0X13, 0X07, 0X0F, 0X1F, 0X1E, 0X1C, 0X18, 0X11, 0X03, 0X06, 0X0D, 0X1B, 0X17, 0X0E, 0X1D).



. 80-bit key block diagram from 80.png

CASE-2: 128-bit key

Let the user-provided key, in this case, be V, such that V = v127|| ... || v1 || v0 (Since, one hundred and twenty eight bit seed key), the key is initially taken in a key register of 128-bit, which can be ordered as a 4x32 rectangle array. The corresponding 2D representation of the key state is shown below-

$$\begin{bmatrix} \kappa_{0,31} & \cdots & \kappa_{0,2} & \kappa_{0,1} & \kappa_{0,0} \\ \kappa_{1,31} & \cdots & \kappa_{1,2} & \kappa_{1,1} & \kappa_{1,0} \\ \kappa_{2,31} & \cdots & \kappa_{2,2} & \kappa_{2,1} & \kappa_{2,0} \\ \kappa_{3,31} & \cdots & \kappa_{3,2} & \kappa_{3,1} & \kappa_{3,0} \end{bmatrix}$$

128-bit key's 2-D Representation

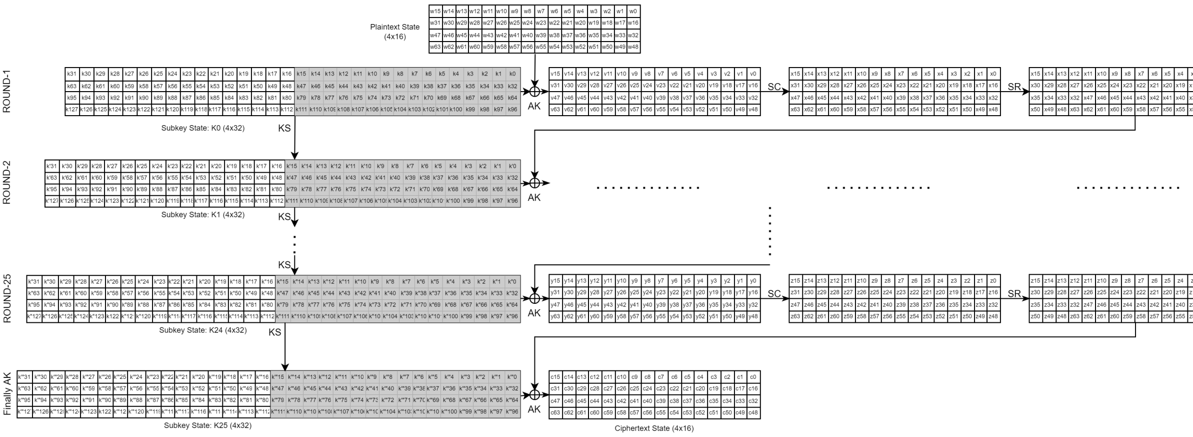
Suppose Rowi = ki,31|| ... || ki,1|| ki,0 indicate the i-th key register's row, such that 0

$\leq$ i $\leq$ 3, where Rowi is considered to a thirty-two bit word. At round i where i belongs to 0 to 24, the 64-bit round subkey Ki contains the rightmost 16 columns of the current contents of the key (shown in the figure below). Once the round subkey Ki is found out , the key register gets updated using the following iterations-

1. Using the S-box S, we perform Subcoumn to the partial 8 rightmost columns (shaded in the figure given below), i.e., k'3,j || k'2,j|| k'1,j || k'0,j := S(k3,j || k2,j || k1,j || k0,j), such that 0 $\leq$ j $\leq$ 7

2. Using 1-round generalized Feistel transformation, i.e.,
Row'0 := (Row0 << 8) $\oplus$ Row1,
Row'1 := Row2,
Row'2 := (Row2 << 16) $\oplus$ Row3,
Row'3 := Row0

3. Now on the 5-bit key state (k0,4|| k0,3|| k0,2|| k0,1|| k0,0), we perform a XOR operation with RC[i] (a 5-bit round constant), where round constants RC[i] for i belongs to zero to twenty-four are the same used above in the previous 80-bit key schedule. Then finally, we get K25 using the above iterations on the updated key state.



. 128-bit key block diagram from 128.png

3 Security Analysis

3.1 Integral Cryptanalysis

We implemented the Square attac and used a 4-round integral distinguisher which holds with with probability 1.

Consider two set of plaintexts P and P* such that $P_{32} = P_{48} = 0$ and $P^*_{32} = P^*_{48} = 0$

P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15
P16	P17	P18	P19	P20	P21	P22	P23	P24	P25	P26	P27	P28	P29	P30	P31
P32	P33	P34	P35	P36	P37	P38	P39	P40	P41	P42	P43	P44	P45	P46	P47
P48	P49	P50	P51	P52	P53	P54	P55	P56	P57	P58	P59	P60	P61	P62	P63

P*0	P*1	P*2	P*3	P*4	P*5	P*6	P*7	P*8	P*9	P*10	P*11	P*12	P*13	P*14	P*15
P*16	P*17	P*18	P*19	P*20	P*21	P*22	P*23	P*24	P*25	P*26	P*27	P*28	P*29	P*30	P*31
P*32	P*33	P*34	P*35	P*36	P*37	P*38	P*39	P*40	P*41	P*42	P*43	P*44	P*45	P*46	P*47
P*48	P*49	P*50	P*51	P*52	P*53	P*54	P*55	P*56	P*57	P*58	P*59	P*60	P*61	P*62	P*63

whereas P_i and P_i^* contains a random fixed value s.t. i $\neq$ 32,48 - Constant Property (represented by **C** in figure)

	C
	C
▨	C
▨	C

It is to observe that the above 2 sets of plaintext have non-zero difference only in col 0, with $P_{Col(0)} \oplus P^*_{Col(0)} = 1100$

Encryption direction :-

After encrypting for 4-rounds, the round outputs have zero difference in round 3 at four bit positions: (0), (17), (43), (60), i.e. Data condition for integral distinguisher which holds with probability 1 is :-

The XOR sum = 0 in any 4 bit positions, i.e. $\oplus\ S_4[0] = \oplus\ S_4[17] = \oplus\ S_4[43] = \oplus\ S_4[60] = 0$ (Balanced property)

P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15
P16	P17	P18	P19	P20	P21	P22	P23	P24	P25	P26	P27	P28	P29	P30	P31
P32	P33	P34	P35	P36	P37	P38	P39	P40	P41	P42	P43	P44	P45	P46	P47
P48	P49	P50	P51	P52	P53	P54	P55	P56	P57	P58	P59	P60	P61	P62	P63

where S_4 denotes output after completion of 3 rounds i.e at the starting of 4th round.

Decryption direction :-

Choose 2^{48} plaintexts s.t. cols - 0, 13, 14, 15 conatin certain fixed values and other 12 cols having 2^{48} [12 x 4 = 48] possible values (ALL property)

C	A	C	C	C
C	A	C	C	C
C	A	C	C	C
C	A	C	C	C

After encrypting 3-rounds the 2^{48} intermediate values can be divided into 2^{47} subsets, where each subset conatins two values satisfying the above 4-round distinguisher.

This can be done upto 7 rounds having the above same integral distinguisher.

And we can similarly show that a 25-round Rectangle provides the sufficient security against IC.

3.2 Differential Cryptanalysis

Differential Cryptanalysis is the process in which we compare the way differences in input(plaintext) related to the differences in encrypted output(ciphertext). It is one of the strongest techniques for the cryptanalysis of block ciphers. The propagation of difference for some rounds should be larger than 2^{1-n} for an n-bit block cipher attack.

Therefore for Rectangle to behold out against this differential attack, the difference propagation probability must be higher than 2^{-63}. The maximum differential probability is found out to be 0.25 and the differential uniformity as 4 (as shown in the figure below).

A search algorithm based on the branch and bound mechanism was introduced by M.Matsui, which finds the best differential trail of Data Encryption Standard (DES). Based on this algorithm, the best differential trail of Rectangle cipher is from one round to 15 rounds was found, which has been shown in the table given below-

♯ R	Cor. Pot.	♯ R	Cor. Pot.	♯ R	Cor. Pot.
1	2^{-2}	6	2^{-20}	11	2^{-50}
2	2^{-4}	7	2^{-26}	12	2^{-56}
3	2^{-8}	8	2^{-32}	13	2^{-62}
4	2^{-12}	9	2^{-38}	14	2^{-68}
5	2^{-16}	10	2^{-44}	15	2^{-74}

. Probabilities at each round

On the experimental basis, we found that 15-rounds are effective to find the differential distinguisher. Then after 4 rounds, full dependency is reached and a 25-round Rectangle is enough to behold out against this differential cryptanalysis attack.

4 Software Implementation

We have implemented RECTANGLE cipher on a 2.5GHz Intel(R) Core i7 CPU running a 64-bit operating system with a C++ compiler (Visual Studio, Ubuntu)

Cipher Implementation :
We have designed a C++ code "rectangle.cpp" for the software implementation of our Rectangle cipher. We are taking a vector of 4 16-bit words, for our message or cipher state and a vector of 5 16-bit words for our key. We are taking input and printing output in binary format as given in the test vectors of paper.

These are the test vectors given in the paper :

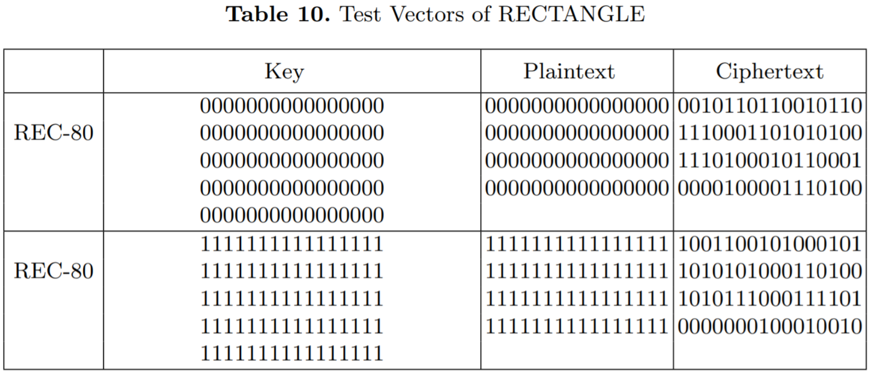
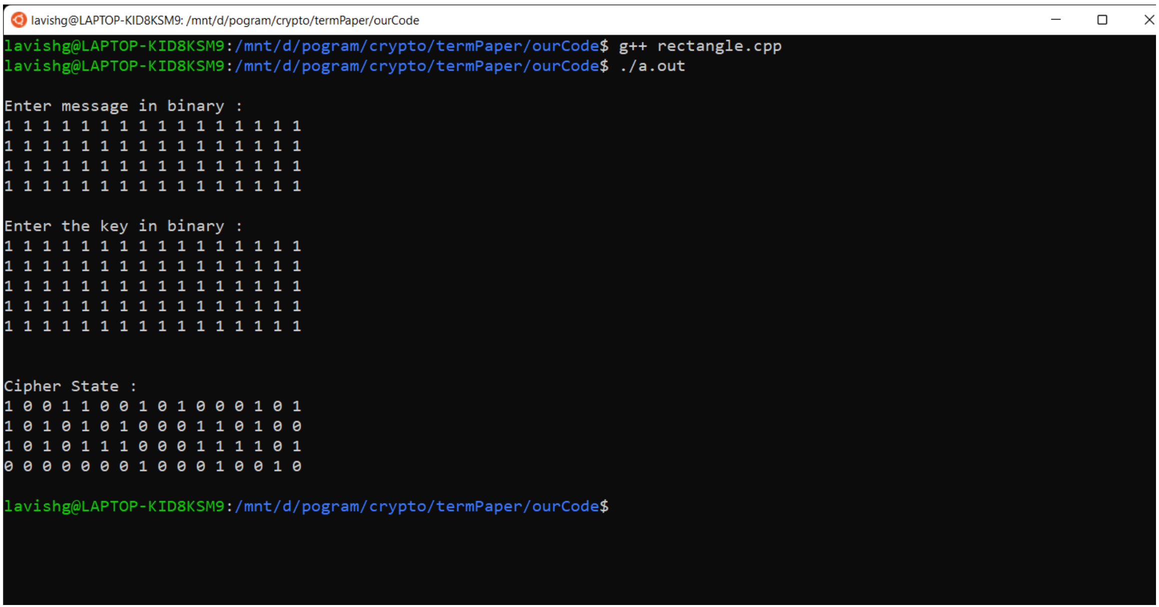


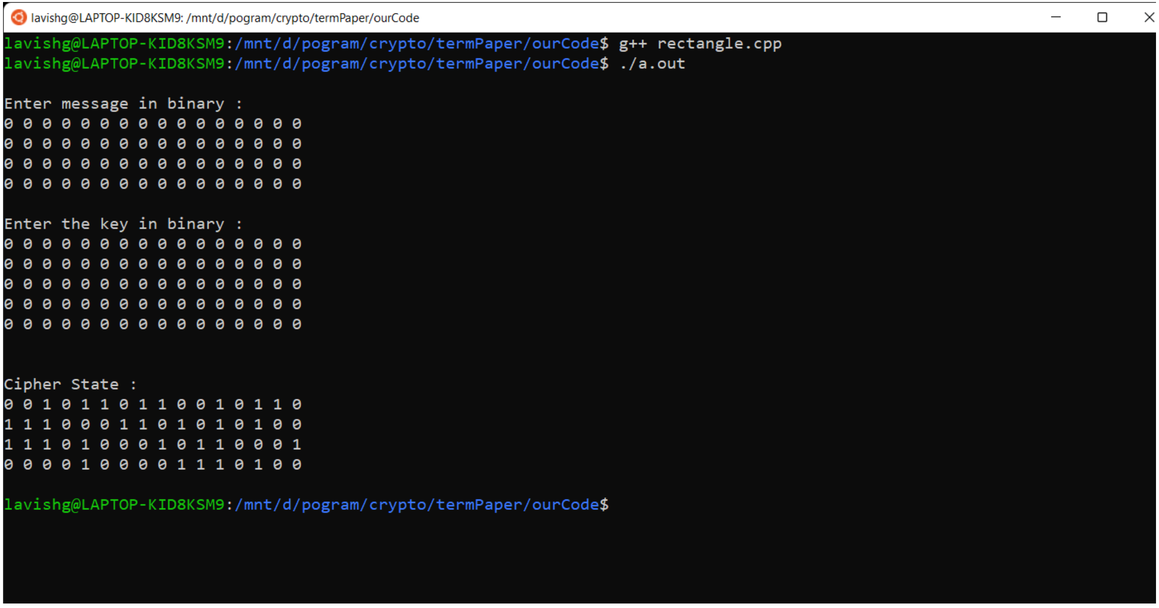
Table 10. Test Vectors of RECTANGLE

	Key	Plaintext	Ciphertext
REC-80	0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000	0000000000000000 0000000000000000 0000000000000000 0000000000000000	0010110110010110 1110001101010100 1110100010110001 0000100001110100
REC-80	1111111111111111 1111111111111111 1111111111111111 1111111111111111 1111111111111111	1111111111111111 1111111111111111 1111111111111111 1111111111111111	1001100101000101 1010101000110100 1010111000111101 0000000100010010

After executing the test vectors on our code, we get the following output on the console :


lavishg@LAPTOP-KID8KSM9: /mnt/d/pogram/crypto/termPaper/ourCode

```
lavishg@LAPTOP-KID8KSM9:/mnt/d/pogram/crypto/termPaper/ourCode$ g++ rectangle.cpp
lavishg@LAPTOP-KID8KSM9:/mnt/d/pogram/crypto/termPaper/ourCode$ ./a.out

Enter message in binary :
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

Enter the key in binary :
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1


Cipher State :
1 0 0 1 1 0 0 1 0 1 0 0 0 1 0 1
1 0 1 0 1 0 1 0 0 0 1 1 0 1 0 0
1 0 1 0 1 1 1 0 0 0 1 1 1 1 0 1
0 0 0 0 0 0 0 1 0 0 0 1 0 0 1 0

lavishg@LAPTOP-KID8KSM9:/mnt/d/pogram/crypto/termPaper/ourCode$
```


lavishg@LAPTOP-KID8KSM9: /mnt/d/pogram/crypto/termPaper/ourCode

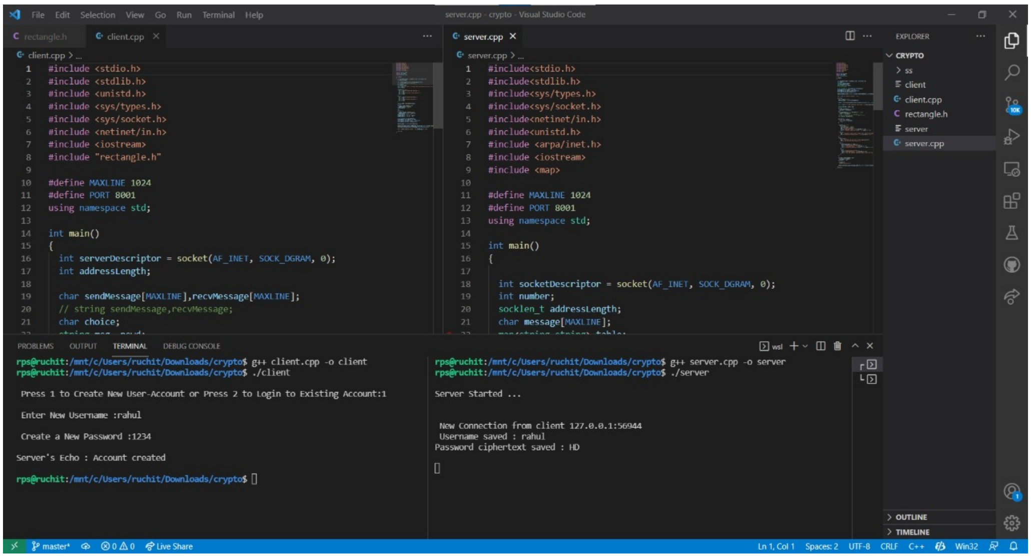
```
lavishg@LAPTOP-KID8KSM9:/mnt/d/pogram/crypto/termPaper/ourCode$ g++ rectangle.cpp
lavishg@LAPTOP-KID8KSM9:/mnt/d/pogram/crypto/termPaper/ourCode$ ./a.out

Enter message in binary :
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Enter the key in binary :
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0


Cipher State :
0 0 1 0 1 1 0 1 1 0 0 1 0 1 1 0
1 1 1 0 0 0 1 1 0 1 0 1 0 1 0 0
1 1 1 0 1 0 0 0 1 0 1 1 0 0 0 1
0 0 0 0 1 0 0 0 0 1 1 1 0 1 0 0

lavishg@LAPTOP-KID8KSM9:/mnt/d/pogram/crypto/termPaper/ourCode$
```

Software Application :
We have made a Login-Authentication system of client and server. The user can choose between one of the two scenarios.

- If the user wants to create the account in the system, the user can enter its username and password. We are sending the username of the user and the ciphered text of the given password to the server. The server would then save the values in a table for later authentication.

- If the user wants to authenticate into the server, then he/she would enter its username and password made during the login time into the client and then the Server would check in its table, whether the given username and password have an entry in the saved table. If the ciphered password would match to the ciphered password present in the table for the particular user then it would successfully authenticate the user otherwise the authentication would fail.

Here are some screenshots of the running application :


File Edit Selection View Go Run Terminal Help
server.cpp - crypto - Visual Studio Code
rectangle.h
client.cpp
server.cpp
EXPLORER
CRYPTO
ss
client
client.cpp
rectangle.h
server
server.cpp
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE
OUTLINE
TIMELINE
master*
Live Share
Ln 1, Col 1 Spaces: 2 UTF-8 CRLF C++ Win32

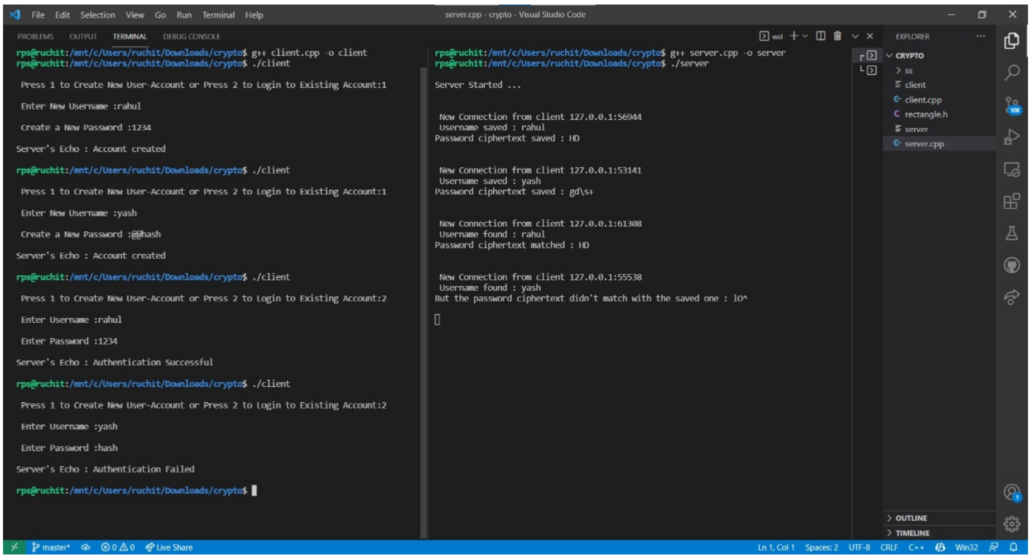
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <iostream>
#include "rectangle.h"

#define MAXLINE 1024
#define PORT 8001
using namespace std;

int main()
{
  int serverDescriptor = socket(AF_INET, SOCK_DGRAM, 0);
  int addressLength;

  char sendMessage[MAXLINE],recvMessage[MAXLINE];
  // string sendMessage,recvMessage;
  char choice;
```

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<unistd.h>
#include <arpa/inet.h>
#include <iostream>
#include <map>

#define MAXLINE 1024
#define PORT 8001
using namespace std;

int main()
{

  int socketDescriptor = socket(AF_INET, SOCK_DGRAM, 0);
  int number;
  socklen_t addressLength;
  char message[MAXLINE];
```

```
rps@ruchit:/mnt/c/Users/ruchit/Downloads/crypto$ g++ client.cpp -o client
rps@ruchit:/mnt/c/Users/ruchit/Downloads/crypto$ ./client

 Press 1 to Create New User-Account or Press 2 to Login to Existing Account:1

 Enter New Username :rahul

 Create a New Password :1234

Server's Echo : Account created

rps@ruchit:/mnt/c/Users/ruchit/Downloads/crypto$ 
```

```
rps@ruchit:/mnt/c/Users/ruchit/Downloads/crypto$ g++ server.cpp -o server
rps@ruchit:/mnt/c/Users/ruchit/Downloads/crypto$ ./server

Server Started ...


 New Connection from client 127.0.0.1:56944
 Username saved : rahul
Password ciphertext saved : HD
```

Connection Estalished


File Edit Selection View Go Run Terminal Help
server.cpp - crypto - Visual Studio Code
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE
EXPLORER
CRYPTO
ss
client
client.cpp
rectangle.h
server
server.cpp
OUTLINE
TIMELINE
master*
Live Share
Ln 1, Col 1 Spaces: 2 UTF-8 CRLF C++ Win32

```
rps@ruchit:/mnt/c/Users/ruchit/Downloads/crypto$ g++ client.cpp -o client
rps@ruchit:/mnt/c/Users/ruchit/Downloads/crypto$ ./client

 Press 1 to Create New User-Account or Press 2 to Login to Existing Account:1

 Enter New Username :rahul

 Create a New Password :1234

Server's Echo : Account created

rps@ruchit:/mnt/c/Users/ruchit/Downloads/crypto$ ./client

 Press 1 to Create New User-Account or Press 2 to Login to Existing Account:1

 Enter New Username :yash

 Create a New Password :@@hash

Server's Echo : Account created

rps@ruchit:/mnt/c/Users/ruchit/Downloads/crypto$ ./client

 Press 1 to Create New User-Account or Press 2 to Login to Existing Account:2

 Enter Username :rahul

 Enter Password :1234

Server's Echo : Authentication Successful

rps@ruchit:/mnt/c/Users/ruchit/Downloads/crypto$ ./client

 Press 1 to Create New User-Account or Press 2 to Login to Existing Account:2

 Enter Username :yash

 Enter Password :hash

Server's Echo : Authentication Failed

rps@ruchit:/mnt/c/Users/ruchit/Downloads/crypto$ 
```

```
rps@ruchit:/mnt/c/Users/ruchit/Downloads/crypto$ g++ server.cpp -o server
rps@ruchit:/mnt/c/Users/ruchit/Downloads/crypto$ ./server

Server Started ...


 New Connection from client 127.0.0.1:56944
 Username saved : rahul
Password ciphertext saved : HD


 New Connection from client 127.0.0.1:53141
 Username saved : yash
Password ciphertext saved : gd\s+


 New Connection from client 127.0.0.1:61308
 Username found : rahul
Password ciphertext matched : HD


 New Connection from client 127.0.0.1:55538
 Username found : yash
But the password ciphertext didn't match with the saved one : lO^
```

Some Input-Outputs

5 Conclusion

RECTANGLE, a lightweight block cipher has been proposed, with a bit-slice technique based on SP-Network, which causes its lightweight and fast executions. It attains a very fast and competitive software performance as well as a very low-cost hardware performance. 16 4×4 S-Boxes are present in the substitution layer in parallel and 3 rotations in the permutation layer. Rectangle provides great hardware as well as software performance. So that it offers applications enough flexibility. Rectangle attains a better security performance because of the cipher's Substitution box careful selection and the permutation layer's asymmetric design. It is believed that the Rectangle cipher is a simple and interesting design and it has the ability to trigger various new cryptographic problems. And at the end, Rectangle cipher security is encouraged further.

6 References

1. Research Paper sources- https://eprint.iacr.org/2014/084.pdf
2. Image (80 and 128-bit key)- https://app.diagrams.net/
3. Image sources- http://eprint.iacr.org/