

RECTANGLE CIPHER

¹ Shreyas Pande, 12041420, shreyassachin@iitbhilai.ac.in

² Niket Srivastav, 12040980, niketsrivastav@iitbhilai.ac.in

³ Prathamesh Gujar, 12041100, prathameshgujar@iitbhilai.ac.in

Abstract. Rectangle Cipher is a lightweight block cipher and a SP Network (a bit slice styled block cipher) where this bit slice implementation of causes its lightweight and fast executions.

Keywords: No keywords given.

1 Introduction

Today, a wide variety of applications use a large number of tiny embedded devices, such as smart cards, RFIDs, sensor nodes, etc. These devices are typically characterised by severe cost constraints, such as energy, area, and power in the case of hardware, and small and optimised code and limited memory space in the case of software. In the meanwhile, cryptographic protection is also essential. In order to provide better security at a cheaper cost, numerous new lightweight cyphers have been developed, including Present, SIMON, Hummingbird, KLEIN, DESL/DESX/DESXL, LBlock, Katan, Piccolo, TWINE, LED, SPECK, and others.

The Present cypher was announced at CHES'2007 and quickly gained popularity because to its improved hardware speed, simplicity, and security. It makes use of some permutation in the diffusion layer to get excellent hardware performance. Present, Katan, and Hummingbird are examples of lightweight cyphers that manage to achieve a low hardware area but fall short of obtaining adequate software performance. The LED cypher is then presented at CHES'2011, where its developers assert that their cypher is not only excellent in hardware performance but also passable in software performance.

However, by studying the concept, effectiveness, and exclusion of the 1st-generation proposals, 2nd-generation lightweight cyphers can be improved. Due to the introduction of the Serpent block cypher employing the new bit-slice technique, the software speed of DES, which is based on the SP-network, increased. With 32 4x4 S-Boxes in the substitution layer, it is a 128-bit block cypher. Five additional ciphers—SHA-3, Trivium, JH, etc.—also benefited from the bit-slice method for their software performance. It is important to note that these work well in both hardware and software.

Additionally, compared to a table-based execution, a bit-slice execution is safeguarded against implementation assaults including timing and caching attacks. There is room for improvement for a committed bit-slice lightweight cypher even though the main objective of all the cyphers mentioned above is not to be lightweight.

As a result, the SP-network-based Rectangle cypher, which makes use of the bit-slice technique, is introduced. As a result, it achieves both highly rapid and competitive software performance and very affordable hardware performance. In the parallel substitution layer, there are sixteen 4x4 S-Boxes, and in the permutation layer, there are three rotations. Very good security, very quick software speed, and hardware friendliness are the cipher's top three advantages.

Hardware performance is highly effective with rectangle cypher. Among all the

lightweight block cyphers, the cypher achieves the fastest software speed thanks to its bit-slice implementation.

2 IMPLEMENTATION

A 4x16 rectangle array can be thought of as the block length in a rectangle block cypher, which has a block length of 64 bits. And either 80 or 128 bits make up the key length. It is a twenty-five round substitution-permutation cypher that uses the three steps of AddRoundKey(ARK), SubColumn(SC), and ShiftRow in each of the twenty-five rounds (SR). At last AddRoundKey is done to obtain final cipher block.

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. A state is the rectangular representation of 4x16 array of bits, due to which the cipher name was given "Rectangle cipher".

Let W denotes the Cipher state then 64 bits of W arranged as given below-

$$\begin{array}{c}
 \left[\begin{array}{ccccc}
 w_{15} & \dots & w_2 & w_1 & w_0 \\
 w_{31} & \dots & w_{18} & w_{17} & w_{16} \\
 w_{47} & \dots & w_{34} & w_{33} & w_{32} \\
 w_{63} & \dots & w_{50} & w_{49} & w_{48}
 \end{array} \right] \text{ Cipher State} \\
 \left[\begin{array}{ccccc}
 a_{0,15} & \dots & a_{0,2} & a_{0,1} & a_{0,0} \\
 a_{1,15} & \dots & a_{1,2} & a_{1,1} & a_{1,0} \\
 a_{2,15} & \dots & a_{2,2} & a_{2,1} & a_{2,0} \\
 a_{3,15} & \dots & a_{3,2} & a_{3,1} & a_{3,0}
 \end{array} \right] \text{ Subkey State}
 \end{array}$$

AddRoundKey(ARK)-

It is an easy bitwise xor between the round subkey and the state at the intermediate of the algorithm.

SubColumn(S)-

Similar to the Subbytes operation, SubColumn applies parallel S-box applications on the column's four bits.

The operation is applied as given below-

$$\begin{array}{cccc}
 \begin{pmatrix} a_{0,15} \\ a_{1,15} \\ a_{2,15} \\ a_{3,15} \end{pmatrix} & \dots & \begin{pmatrix} a_{0,2} \\ a_{1,2} \\ a_{2,2} \\ a_{3,2} \end{pmatrix} & \begin{pmatrix} a_{0,1} \\ a_{1,1} \\ a_{2,1} \\ a_{3,1} \end{pmatrix} & \begin{pmatrix} a_{0,0} \\ a_{1,0} \\ a_{2,0} \\ a_{3,0} \end{pmatrix} \\
 \downarrow s & & \downarrow s & \downarrow s & \downarrow s \\
 \begin{pmatrix} b_{0,15} \\ b_{1,15} \\ b_{2,15} \\ b_{3,15} \end{pmatrix} & \dots & \begin{pmatrix} b_{0,2} \\ b_{1,2} \\ b_{2,2} \\ b_{3,2} \end{pmatrix} & \begin{pmatrix} b_{0,1} \\ b_{1,1} \\ b_{2,1} \\ b_{3,1} \end{pmatrix} & \begin{pmatrix} b_{0,0} \\ b_{1,0} \\ b_{2,0} \\ b_{3,0} \end{pmatrix}
 \end{array}$$

HERE THE SBOX IS IMPLEMENTED AS:

sbox:—————

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

The DDT Table is implemented as:

```
sage: from sage.crypto.sbox import SBox
sage: S = SBox(6,5,12,10,1,14,7,9,11,0,3,13,8,15,4,2)
sage: S
(6, 5, 12, 10, 1, 14, 7, 9, 11, 0, 3, 13, 8, 15, 4, 2)
sage: S.difference_distribution_table()
.....
[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]
sage: 
```

ShiftRow(SR)-

It is the left rotation of the rows of the state matrix.
The Rotation offset of each row is given as follows-

Columns	Offset
First	1
Second	2
Third	3
Fourth	4

The LAT is implemented as:

```
sage: from sage.crypto.sbox import SBox
sage: S = SBox(6,5,12,10,1,14,7,9,11,0,3,13,8,15,4,2)
sage: S
(6, 5, 12, 10, 1, 14, 7, 9, 11, 0, 3, 13, 8, 15, 4, 2)
sage: S.linear_approximation_table()
[ 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]
[ 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]
[ 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 0 0 0 0 0]
sage: 
```

we get the values of differential branch number and differential uniformity with the help of sage math as:

```
sage: from sage.crypto.sbox import SBox
sage: S = SBox(6,5,12,10,1,14,7,9,11,0,3,13,8,15,4,2)
sage: S
(6, 5, 12, 10, 1, 14, 7, 9, 11, 0, 3, 13, 8, 15, 4, 2)
sage: S.differential_uniformity()
4
sage: S.differential_branch_number()
2
sage: 
```

let's compare the values of the branch number and differential uniformity of other ciphers with that of ours:

Group Name	Cipher Name	SBox Size	Differential Uniformity	Differential Branch Number
gugu_gaga	Midori	4-bit	4	2
SPL Encrypted	GIFT	4-bit	6	2
Hope we "3" SDVV	Serpent	4-bit	4	3
~../cipher	Prince	4-bit	4	2
TechHeist 3.0	Pride	4-bit	4	2
Rook	Ascon	5-bit	8	3
Three Amigos	Klein	4-bit	4	2
Decryptor	PHOTON-beetle	4-bit	4	3
cryptoducks	LED	4-bit	4	3
Ping 999+	Elephant	4-bit	4	3
Kryptonian	Wage	8-bit	8	2
cipherbytes	Aria	8-bit	4	2
C14	Primates APE	5-bit	2	2
BitBees	Skinny	4-bit	2	2
Bash Ciphers	PRINT	3-bit	2	2
SHA69	Mysterion	4-bit	4	2
Hex Brains	Rectangle	4-bit	4	2

Differential Branch Number Signifies that the greater the value , the greater the diffusion power of the permutation of SBox.

Our Sbox has the branch number as 2 so its not as much powerful as some of the other ciphers like Serpent , LED etc.

Similarly, differential uniformity tells us the spread of the Sbox . the lower the value the more powerful the Cipher is.

Our uniformity value is 4 which is not much powerful but at a mediocre stage than other ciphers.

Key Schedule-

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

(CASE-1: 80-bit key)

If the key is of 80-bit, it is represented as 5x16 rectangular array as shown in the figure below-

$$\begin{bmatrix} k_{0,15} & \dots & k_{0,2} & k_{0,1} & k_{0,0} \\ k_{1,15} & \dots & k_{1,2} & k_{1,1} & k_{1,0} \\ k_{2,15} & \dots & k_{2,2} & k_{2,1} & k_{2,0} \\ k_{3,15} & \dots & k_{3,2} & k_{3,1} & k_{3,0} \\ k_{4,15} & \dots & k_{4,2} & k_{4,1} & k_{4,0} \end{bmatrix}$$

Key's 2D Representation

Suppose $\text{Row}_i = k_{i,15} || \dots || k_{i,1} || k_{i,0}$ indicate the i-th key register's row, such that $0 \leq i \leq 4$, where Row_i is considered to a sixteen bit word. At round i where i belongs to 0 to 24, the sixty-four bit round subkey K_i contains the 1st 4 rows of the key register's contents (as shown in the figure given below), i.e., $K_i = \text{Row}_3 || \text{Row}_2 || \text{Row}_1 || \text{Row}_0$. Once the key register K_i found out, the key register gets updated using the following iterations-

1. We execute Subcolumn to the partial bits that intersect at the four uppermost rows and the four rightmost columns using the S-box S.i.e i.e., $k'_{3,j} || k'_{2,j} || k'_{1,j} || k'_{0,j} := S(k_{3,j} || k_{2,j} || k_{1,j} || k_{0,j})$, such that j belongs to (0, 1, 2, 3).
2. Using 1-round generalized Feistel transformation, i.e.,
 $\text{Row}'_0 := (\text{Row}_0 \ll 8) \oplus \text{Row}_1$
 $\text{Row}'_1 := \text{Row}_2$
 $\text{Row}'_2 := \text{Row}_3$
 $\text{Row}'_3 := (\text{Row}_3 \ll 12) \oplus \text{Row}_4$
 $\text{Row}'_4 := \text{Row}_0$
3. Now on the 5-bit key state (i.e. $k_{0,4} || k_{0,3} || k_{0,2} || k_{0,1} || k_{0,0}$), we perform a XOR operation with $\text{RC}[i]$ (a 5-bit round constant), i.e., $k'_{0,4} || k'_{0,3} || k'_{0,2} || k'_{0,1} || k'_{0,0} := (k_{0,4} || k_{0,3} || k_{0,2} || k_{0,1} || k_{0,0}) \oplus \text{RC}[i]$
 Using the aforementioned iterations on the revised key state, we ultimately obtain K_{25} . The round constants $\text{RC}[i]$ are produced using a 5-bit LFSR for all i between 0 and 24. The five bits ($\text{rc}_4, \text{rc}_3, \text{rc}_2, \text{rc}_1, \text{rc}_0$) are left-shifted by one bit at each round, and the new value of rc_0 is then assessed as $\text{rc}_4 \text{ xor } \text{rc}_2$. At the beginning, $\text{RC}[0] := 0x1$ is taken. Now that we have it, we can use it to 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).

(CASE-2: 128-bit key)

If the key is of 80-bit, it is represented as 4x32 rectangular array as shown in the figure below-

$$\begin{bmatrix} k_{0,31} & \dots & k_{0,2} & k_{0,1} & k_{0,0} \\ k_{1,31} & \dots & k_{1,2} & k_{1,1} & k_{1,0} \\ k_{2,31} & \dots & k_{2,2} & k_{2,1} & k_{2,0} \\ k_{3,31} & \dots & k_{3,2} & k_{3,1} & k_{3,0} \end{bmatrix} \text{ Key's 2D Representation}$$

Suppose $\text{Row}_i = k_{i,15} \parallel \dots \parallel k_{i,1} \parallel k_{i,0}$ indicate the i -th key register's row, such that $0 \leq i \leq 4$, where Row_i is considered to a sixteen bit word. At round i where i belongs to 0 to 24, the sixty-four bit round subkey K_i contains the 1st 4 rows of the key register's contents (as shown in the figure given below), i.e., $K_i = \text{Row}_3 \parallel \text{Row}_2 \parallel \text{Row}_1 \parallel \text{Row}_0$. Once the key register K_i found out, the key register gets updated using the following iterations-

1. We execute Subcolumn to the partial 8 rightmost column i.e., $k'_{3,j} \parallel k'_{2,j} \parallel k'_{1,j} \parallel k'_{0,j} := S(k_{3,j} \parallel k_{2,j} \parallel k_{1,j} \parallel k_{0,j})$, such that j belongs to $(0, 1, 2, 3, 4, 5, 6, 7)$.
2. Using 1-round generalized Feistel transformation, i.e.,
 $\text{Row}'0 := (\text{Row}_0 \ll 8) \oplus \text{Row}_1$
 $\text{Row}'1 := \text{Row}_2$
 $\text{Row}'2 := (\text{Row}_2 \ll 12) \oplus \text{Row}_3$
 $\text{Row}'3 := \text{Row}_0$
3. Now on the 5-bit key state $(k_{0,4} \parallel k_{0,3} \parallel k_{0,2} \parallel k_{0,1} \parallel k_{0,0})$, we perform a XOR operation with $\text{RC}[i]$ (a 5-bit round constant), where round constants $\text{RC}[i]$ for i belongs to zero to twenty-four are the same used above in the previous 80-bit key schedule. Then finally, we get K_{25} using the above iterations on the updated key state.

3 Differential Cryptanalysis

It is the way where we observe the differences between input plaintexts and relate it to the output differences. It is a kind of strong attack on the Rectangle cipher.

We know that to attack a n -bit block cipher using differential cryptanalysis there must be a difference propagation with a probability significantly larger than 2^{1-n} .

As the block size is 64 bits, so our cipher must have a difference propagation of at least 2^{-63} to be safe from the differential attack.

We know from the DDT Table that, the max differential probability is: 2^{-6} .

The value of differential branch number was = 2 and differential uniformity = 4.

There was an attack generated on the Rectangle cipher for up to 15 rounds. The probability of the trail of 15-rounds was found to be as: 2^{-66} .

#R	Prob.	#R	Prob.	#R	Prob.
1	2^{-2}	6	2^{-18}	11	2^{-46}
2	2^{-4}	7	2^{-25}	12	2^{-51}
3	2^{-7}	8	2^{-31}	13	2^{-56}
4	2^{-10}	9	2^{-36}	14	2^{-61}
5	2^{-14}	10	2^{-41}	15	2^{-66}

As the ShiftRow Operations of the cipher is simple, the security of Rectangle against cryptanalysis is difficult.

Due to rotational symmetry every trail has up to 16 variants. For a 15 round algorithm, the differential probability lies between 2^{-66} to 2^{-76} .

So after some experimental data, we get that it is impossible to construct an effective 15 round differential attack (distinguisher) . 4 more rounds are required to attain full dependency. Hence 25 rounds were made fixed for the RECTANGLE cipher to resist the attack .

4 Integral Cryptanalysis

we apply a 4 round distinguisher attack on the text with the help of two sets of plaintexts,P1 and P2

we assign 0 to the 32nd and 48th value of P1 set plaintexts and assign a 1 to the 32nd and 48th index of P2 set plaintexts.

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

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

The rest of the blocks are assigned a constant value . i.e a constant property is assigned to the all the rest of the blocks in both of the sets.

so now we can basically see that the difference of the blocks is non-zero in the 0th column while its 0 for all the leftover columns

so therefore we can infer that :

$$P1(\text{col}0) \oplus P2(\text{col}0) = 1100$$

Let's see the encryption part:

as we apply the encryption on plaintexts, we see that after 4 rounds of encryption the only places that are left with 0 output difference is the position number 0 , 17,43 and 60.

which means that the position 0,17,43,60 accept the balanced property.

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

Let's see how we can do the decryption on the given set of texts, we take 2^{48} plaintexts such that the columns 0 , 13 , 14 ,15 contains a fixed constant value and other blocks have a All property.

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

This can be don upto 7 rounds of encryption .

But as 25-round Rectangle provides sufficient security against the Integral Cryptanalysis.

C	A	A	A	...	A	C	C	C
C	A	A	A	...	A	C	C	C
C	A	A	A	...	A	C	C	C
C	A	A	A	...	A	C	C	C

5 Software Implementation

We have implemented Rectangle cipher on 64-bit operating system with a python interpreter.

Cipher Implementation:

We have designed RECTANGLE cipher for vector 4 16-bit words, and a key vector of 5 16-bit word.

We are taking input in hex format and printing it into binary format

If input have less than 64 bits then padding of 0 is added in the end.

Alongwith that a decryptor function that takes input as ciphertext and key is also made which returns us the plaintext back .

Below are the screenshots of the terminal-

```

encryptor.py X
encryptor.py
132 # print("beforeee",state)
133 state = subcolumn(state)
134 print("after sbox",state)
135 # print("afterrrr",state)
136 shiftrows(state)
137 print("after shift",state)
138 # print(state)
139 keyupdate(key , j)
140 # print("-----\n")
141 print(state)
142 # print("key")
143 # print(key)
144 print("-----\n")
145

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
[ '1101110010100010', '000101100110001', '010001001110000', '0011111010100110' ]
-----

Round : 21
after adding key ['0111000101110011', '1011110110000001', '00110100101000101', '0100101011110010']
added key ['1010110111010001', '1010101010110000', '0111000001010101', '0111010001010100', '1110111011000001']
after sbox ['001111000100101', '000011100101000', '1000010010101000', '1011001100100001', '1110111011000001']
after shift ['001111000100101', '000111001010000', '100010001001010', '0011011001100100', '1110111011000001']
['001111000100101', '000111001010000', '100010001001010', '0011011001100100', '1110111011000001']
-----

Round : 22
after adding key ['0100111000011111', '0110101000000001', '0111000111001110', '1101010000101001']
added key ['011100000111010', '0111010001010001', '1111100110000100', '1110001001001101', '001111100000000']
after sbox ['101000011111001', '1110101000101110', '0010000000011111', '1001010000101111']
after shift ['101000011111001', '1110101000101110', '1111001000000001', '111100101000101', '1110101000010101']
['101000011111001', '1110101000101110', '1111001000000001', '111100101000101', '1110101000010101']
-----

Round : 23
after adding key ['0110101000101000', '0011011000010010', '0111001100100101', '1100101101100100']
added key ['1100101111010001', '1110001001001111', '1000000100100100', '0011100000100001', '0100011100110010']
after sbox ['111100000101001', '1101001011100000', '0101010010111010', '1110010101101110', '1110010101101110']
after shift ['111100000101001', '1010010111000001', '101001010001011', '10111001010101110', '1111001010101110']
['111100000101001', '1010010111000001', '101001010001011', '10111001010101110', '1111001010101110']
-----

Round : 24
after adding key ['1111011001110001', '1001110111100100', '1011000111111011', '0001011000110000']
added key ['0000011000011000', '001100000100101', '0001010010110000', '1100101010011110', '0011011110000111']
after sbox ['110001011101010', '0011000101100001', '0101011001010101', '1010110001010101', '1110010110001010']
after shift ['110001011101010', '011000101100010', '0101011001010101', '1010101101100010', '10110010110001010']
['110001011101010', '011000101100010', '0101011001010101', '1010101101100010', '10110010110001010']
-----

FINAL STATE
['110010001010010', '101010011110110', '001010111110111', '0101010000100011']
hexform of cipher is: 4a276f95efb4c42a
shreyas@shreyas-HP-Pavilion-Laptop-15-cclxx:~/Downloads/rectangle_cipher$

```



```

173
176
177
178 def decrypt(cipher,ke):
179     print("let's start",cipher)
180     k = ke
181     temp=""
182     key=[]
183     for i in range(len(k)):
184         res=int(k[i],16)
185         bn=bin(res).replace("0b","")
186         bn='0'*(4-len(bn))+bn
187         temp+=bn
188     key=temp
189
190     rev_shift ['0101011101011111', '111000001101101', '000000110011001', '0010101101011111']
191     rev_sbox ['1010001110011011', '1110001110010010', '100110010001011', '110010111011011']
192     ['00000110101110', '10000100111000', '110110101100110', '01011001111010']
193     currentkey ['101001001110101', '011001101101010', '01000001101011', '100101100100111', '001011111011110']
194     before doing ['000001101101110', '100001010111000', '110110101100110', '01011001111010']
195     rev_shift ['000001101101110', '01000010111100', '11010111001101', '111001111010010']
196     rev_sbox ['010011011010010', '100101010111111', '011100001000111', '001111101100001']
197     currentkey ['101011011100001', '100000000101010', '000010101000011', '010110011000010']
198     currentkey ['111000110011001', '00010101001101', '011101011001100', '011001101100011', '111001110110001']
199     before doing ['101011011100001', '100000000101010', '000010101000011', '010110011000010']
200     rev_shift ['101011011100001', '010000000101001', '1010010000110000', '1100110000010010']
201     rev_sbox ['111110100001100', '110011000011001', '000010101101010', '100101000011111']
202     ['0101100100001', '011001000010101', '010100000110001', '100000000100100']
203     currentkey ['1010000101001101', '1010100000010100', '010110101101011', '00010101001011', '101010110011110']
204     before doing ['010110001000001', '011001000010101', '0101000000110001', '100000000100100']
205     rev_shift ['010110001000001', '1011001000010110', '0000001100010101', '000000010010100']
206     rev_sbox ['1010000010001111', '101100010000010', '111011000110010', '10110000101010']
207     ['000000000001111', '100110000110001', '110100111000101', '000110001010100']
208     currentkey ['101000010000000', '0010110100110011', '00111111101111', '101010000001110', '0001000001011010']
209     before doing ['000000000001111', '100110000110001', '110100111000101', '000110001010100']
210     rev_shift ['000000000001111', '110011000011000', '00110001001101', '110001010100000']
211     rev_sbox ['110001100100101', '1111011010001000', '0011001100101010', '001100110100000']
212     ['10011010101100', '000001010000010', '110010011010101', '000111001011010']
213     currentkey ['01011011101001', '111100000001010', '111101010011111', '00101100111010', '000011010011100']
214     before doing ['100110110101100', '000001101000010', '110010011010101', '000111001011010']
215     rev_shift ['100110110101100', '000000110100001', '100110110101100', '111100101101000']
216     rev_sbox ['100110110101111', '100100011001001', '01100111110001', '11110000000011']
217     ['010010001010101', '00011110111100', '000000000000000', '000000000000000']
218     currentkey ['1101001010011010', '100111011101101', '011001111100001', '111100000000011', '0101010110011111']
219     decrypted text:
220     ['010100001010101', '000011110111100', '000000000000000', '000000000000000']
221     hexform of plaintext is: a0123ef000000000
222 shreyas@shreyas-HP-Pavilion-Laptop-15-c1xxx:~/Downloads/rectangle ciphers

```

Software Implementation-

Software Implementation contains 3 files-

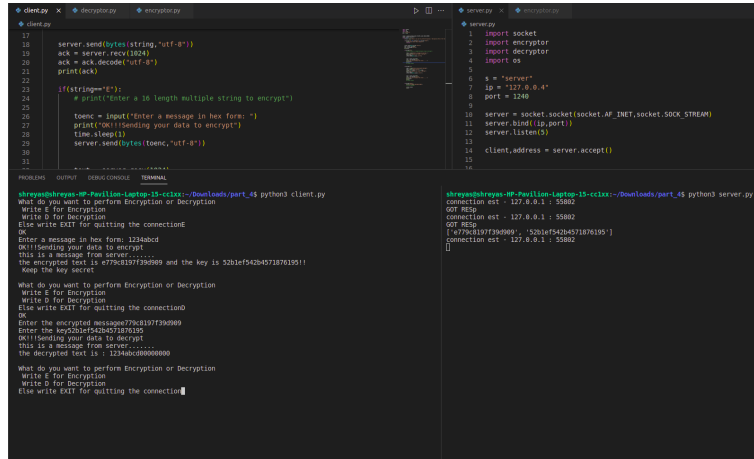
1. server.py-opens socket and waits for the client to connect and share message.
2. client.py-establishes connection i.e binds with socket of server.py
3. encryptor.py-contains the encryption of the RECTANGLE cipher
4. decryptor.py- contains the decryption of the RECTANGLE cipher

We have implemented a client server based application in which client can send a message to server in the form of UDP packets and server in return sends the encrypted message which also is in the form of UDP packet.

key is kept secured with the server which no client can access it.

Server here acts as a black box or oracle for each and every client which establishes a connection with server.

Below are the screenshots of the terminal-



```

client.py
17
18 server.sendBytes(string,"utf-8")
19 ack = server.recv(1024)
20 ack = ack.decode("utf-8")
21 print(ack)
22
23 if(string=="E"):
24     # print("Enter a 16 length multiple string to encrypt")
25
26     toenc = input("Enter a message in hex form: ")
27     print("Sending your data to encrypt")
28     time.sleep(1)
29     server.sendBytes(toenc,"utf-8")
30
31
server.py
1
2 import socket
3 import encryptor
4 import decryptor
5 import os
6
7 s = "server"
8 ip = "127.0.0.1"
9 port = 1240
10
11 server = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
12 server.bind((ip,port))
13 server.listen(5)
14
15 client,address = server.accept()
16
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
$python3 client.py
What do you want to perform Encryption or Decryption
Write E for Encryption
Write D for Decryption
Else write EXIT for quitting the connection
E
Enter a message in hex form: 1234abcd
Sending your data to encrypt
This is a message from server: 1234abcd
The encrypted test is 675a0b37f39d09 and the key is 5201ef542b4571876195!!
Keep the key secret
What do you want to perform Encryption or Decryption
Write E for Encryption
Write D for Decryption
Else write EXIT for quitting the connection
D
Enter the encrypted message: 675a0b37f39d09
Sending your data to decrypt
This is a message from server: 675a0b37f39d09
The decrypted test is : 1234abcd00000000
What do you want to perform Encryption or Decryption
Write E for Encryption
Write D for Decryption
Else write EXIT for quitting the connection
EXIT
$python3 server.py
connection est - 127.0.0.1 : 55802
GET RESP
connection est - 127.0.0.1 : 55802
GET RESP
[675a0b37f39d09, '5201ef542b4571876195']
connection est - 127.0.0.1 : 55802

```

6 Conclusion

A bit-slice approach based on SP-Network has been presented as part of RECTANGLE, a lightweight block cypher that is quick to execute. Both a highly quick and competitive software performance and a very affordable hardware performance are attained. In the parallel substitution layer, there are 16 44 S-Boxes, and in the permutation layer, there are 3 rotations. Excellent hardware and software performance is offered by rectangle. in order to provide applications with sufficient flexibility. Because to the careful selection of the cipher's Substitution box and the asymmetric architecture of the permutation layer, Rectangle achieves a higher security performance. The Rectangle cypher is said to be a straightforward and intriguing design with the potential to lead to numerous new cryptographic issues. Rectangle cypher security is thereafter advocated.

7 References

1. <https://eprint.iacr.org/2014/084.pdf>
2. <https://eprint.iacr.org/>