RECTANGLE CIPHER

[1] Kriti Gupta, 12140940, kritig@iitbhilai.ac.in
[2] Siddhi Agarwal, 12141570, siddhiag@iitbhilai.ac.in

Abstract. The Rectangle Cipher is a lightweight block cipher that leverages a bit-slice-style implementation and an SP-network design, enabling efficient and fast execution. Its bit-slice architecture contributes to its lightweight nature and ensures exceptional performance across both software and hardware platforms. The cipher features sixteen parallel 4×4 S-Boxes in its substitution layer and three rotations in its permutation layer, which collectively enhance its flexibility and adaptability for various applications.
The Rectangle Cipher achieves efficient hardware performance while maintaining one of the fastest software execution speeds among existing lightweight ciphers. Its robust security is a result of carefully selected S-Boxes and an asymmetric permutation layer design, providing resilience against cryptanalytic attacks. Comprehensive security analysis shows that a maximum of 18 out of the 25 rounds can be successfully attacked.

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

1 Introduction

Small embedded devices, such as RFIDs, sensor nodes, and smart cards, are increasingly deployed across a variety of applications. These devices are subject to stringent cost constraints, including hardware limitations like area, power, and energy consumption, as well as software requirements such as low memory usage and compact code sizes. Despite these constraints, robust cryptographic protection is essential to ensure their security. Lightweight ciphers have thus emerged as cost-effective solutions, offering strong security with minimal resource consumption. Symmetric-key block ciphers, in particular, play a critical role in securing these devices, making the design of lightweight block ciphers a highly active area of research.

Numerous lightweight block ciphers employing diverse design strategies have been proposed, including PRESENT, SPECK and LED. Among these, PRESENT gained significant attention for its simplicity, strong security, and exceptional hardware performance. It employs a bit permutation in its diffusion layer, ensuring cost-effective hardware implementation. However, PRESENT and other ciphers like KATAN and Hummingbird, while efficient in hardware, often suffer from performance bottlenecks in software implementations, particularly in their diffusion layers. This dual demand for high hardware and software efficiency has driven the search for new approaches in lightweight cipher design.

The bit-slice technique, introduced to improve the software efficiency of DES, has demonstrated significant potential in achieving high software performance for symmetric ciphers. This method aligns software logical instructions with the simultaneous execution of multiple hardware logical gates, as seen in the Serpent block cipher, which uses 32 parallel 4×4 S-Boxes in its substitution layer. Ciphers like SHA-3, Trivium, and JH have also leveraged bit-slice implementations, achieving strong performance across both hardware and software while enhancing resistance to implementation attacks such as timing and cache-based attacks. However, these bit-slice-oriented designs were not explicitly tailored for lightweight applications, leaving room for improvement in crafting dedicated lightweight bit-slice ciphers.

To address these limitations, the RECTANGLE cipher was introduced. RECTANGLE is an SP-network-based lightweight block cipher that uses the bit-slice technique to achieve both high software efficiency and low-cost hardware implementation. It features sixteen parallel 4×4 S-Boxes in its substitution layer and three rotations in its permutation layer, creating an asymmetric design that enhances its security. RECTANGLE excels in hardware compactness, delivering highly competitive software speeds while maintaining robust security. Its carefully selected S-boxes and asymmetric permutation layer ensure resilience against clustering of linear and differential trails, distinguishing it from earlier lightweight designs such as PRESENT. These attributes make RECTANGLE a highly efficient and secure solution for resource-constrained embedded systems.

2 Implementation

The RECTANGLE cipher is structured around a 4×16 rectangular array, representing a block length of 64 bits. It supports key lengths of either 80 or 128 bits, depending on the security requirements. The cipher operates over 25 rounds, with each round consisting of three core operations: AddRoundKey (ARK), SubColumn (SC), and ShiftRow (SR). These operations are designed to ensure strong diffusion and confusion throughout the encryption process.
After completing the 25 rounds, a final AddRoundKey operation is performed to generate the ciphertext block. The cipher's simplicity and efficiency make it highly suitable for lightweight cryptographic applications.
The pseudocode for the RECTANGLE Cipher is given below-

```
def GenerateRoundKeys(state):
    for i in range(25):
        ARK(state, Ki)
        SC(state)
        SR(state)
    ARK(state, K25)
```

State (Cipher and Subkey State)- The cipher state represents a 64-bit plaintext, intermediate result, or ciphertext, and is organized as a 4x16 rectangular array of bits, which is the origin of the cipher's name "Rectangle". Denoted as W, the cipher state is expressed as 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.

$$\begin{bmatrix} w_{15} & \cdots & w_2 & w_1 & w_0 \\ w_{31} & \cdots & w_{18} & w_{17} & w_{16} \\ w_{47} & \cdots & w_{34} & w_{33} & w_{32} \\ w_{63} & \cdots & w_{50} & w_{49} & w_{48} \end{bmatrix}$$

Cipher State

$$\begin{bmatrix} a_{0,15} & \cdots & a_{0,2} & a_{0,1} & a_{0,0} \\ a_{1,15} & \cdots & a_{1,2} & a_{1,1} & a_{1,0} \\ a_{2,15} & \cdots & a_{2,2} & a_{2,1} & a_{2,0} \\ a_{3,15} & \cdots & a_{3,2} & a_{3,1} & a_{3,0} \end{bmatrix}$$

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.

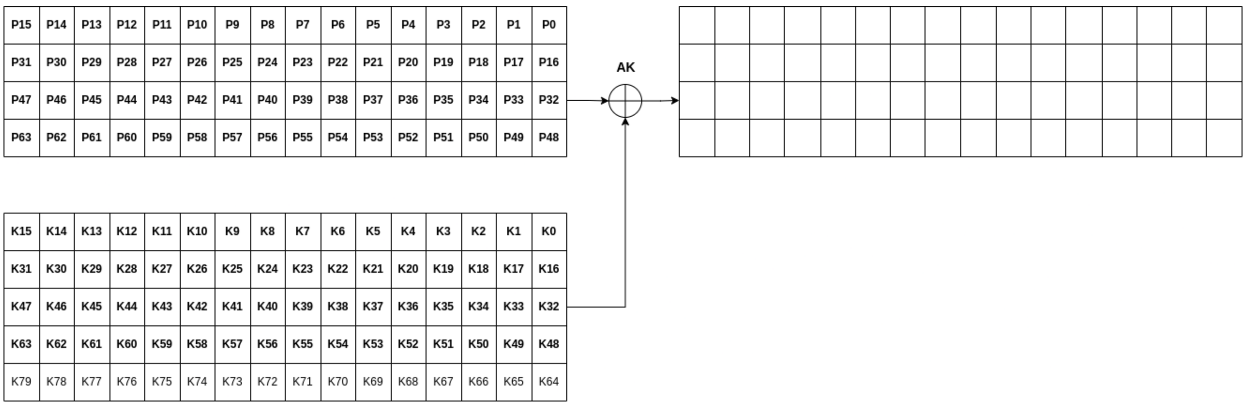

P15 P14 P13 P12 P11 P10 P9 P8 P7 P6 P5 P4 P3 P2 P1 P0
P31 P30 P29 P28 P27 P26 P25 P24 P23 P22 P21 P20 P19 P18 P17 P16
P47 P46 P45 P44 P43 P42 P41 P40 P39 P38 P37 P36 P35 P34 P33 P32
P63 P62 P61 P60 P59 P58 P57 P56 P55 P54 P53 P52 P51 P50 P49 P48
AK
K15 K14 K13 K12 K11 K10 K9 K8 K7 K6 K5 K4 K3 K2 K1 K0
K31 K30 K29 K28 K27 K26 K25 K24 K23 K22 K21 K20 K19 K18 K17 K16
K47 K46 K45 K44 K43 K42 K41 K40 K39 K38 K37 K36 K35 K34 K33 K32
K63 K62 K61 K60 K59 K58 K57 K56 K55 K54 K53 K52 K51 K50 K49 K48
K79 K78 K77 K76 K75 K74 K73 K72 K71 K70 K69 K68 K67 K66 K65 K64

Figure 2: Above diagram show add round-key operation

SubColumn(S)-

SubColumn is similar to the Subbytes operation, where the parallel application of S-boxes gets applied to the 4 bits in the same column. If the input of an S-box is Col(j) = $a_{3,j}||a_{2,j}||a_{1,j}||a_{0,j}$ for $0 \leq j \leq 15$, then the result would be S(Col(j)) = $b_{3,j}||b_{2,j}||b_{1,j}||b_{0,j}$. This operation is applied as follows-

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

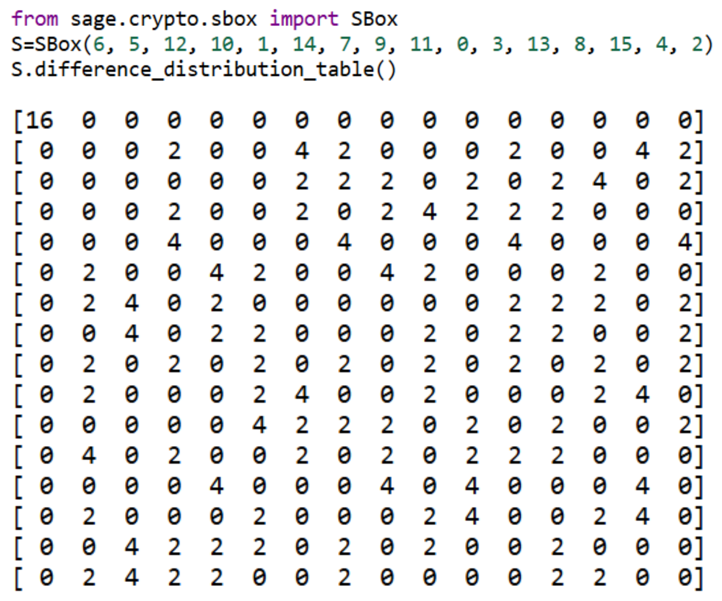
SubColumn Operation on the Columns of the State

The S-Box used in Rectangle is denoted as $F_2^4 \rightarrow F_2^4$ 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

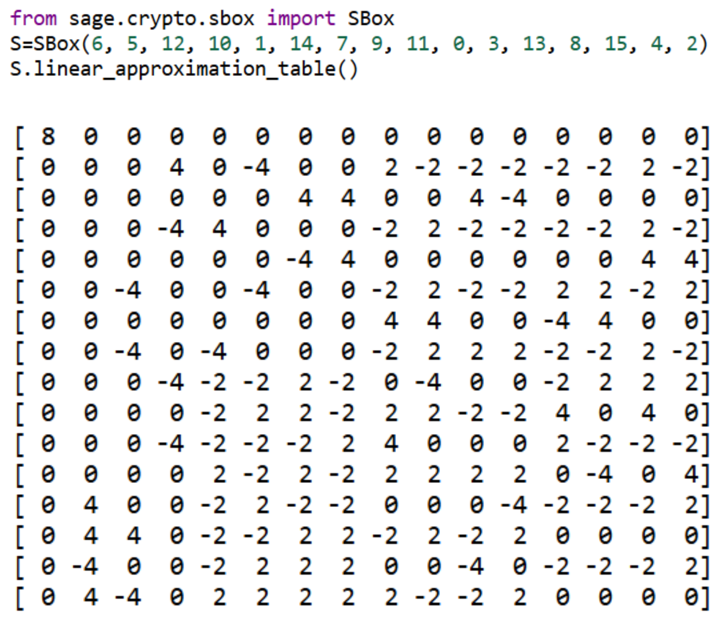
Rectangle Cipher S-Box

The DDT is implemented as:



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

The LAT is implemented as:



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

On calculating, Differential Uniformity = 4 and Differential Branch Number = 2 for the s-box of Rectangle Cipher

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

Group No.	Cipher Name	SBox size	Differential Uniformity	Differential Branch NUmber
4	PRINCE	4-bit	4	2
5	KLEIN	4-bit	4	2
6	LED	4-bit	4	3
8	Midori	4-bit	4	2
10	GIFT	4-bit	6	2
11	PRINT	3-bit	2	2
12	Rectangle	4-bit	4	2

Our uniformity value is 4 which is mostly like other ciphers.

ShiftRow(SR)-

The ShiftRow operation involves performing left rotations on the rows of a state matrix, with varying offsets for each row. The specifics of the rotations are as follows:

Let the notation $\ll x$ represent a left rotation by x bits. The transformation is illustrated below:

$$\begin{aligned}
\text{Row 0:} &\quad (a_{0,15}, \ldots, a_{0,1}, a_{0,0}) \xrightarrow{\ll 0} (a_{0,15}, \ldots, a_{0,1}, a_{0,0}) \\
\text{Row 1:} &\quad (a_{1,15}, \ldots, a_{1,1}, a_{1,0}) \xrightarrow{\ll 1} (a_{1,14}, \ldots, a_{1,0}, a_{1,15}) \\
\text{Row 2:} &\quad (a_{2,15}, \ldots, a_{2,1}, a_{2,0}) \xrightarrow{\ll 12} (a_{2,3}, \ldots, a_{2,6}, a_{2,5}, a_{2,4}) \\
\text{Row 3:} &\quad (a_{3,15}, \ldots, a_{3,1}, a_{3,0}) \xrightarrow{\ll 13} (a_{3,2}, \ldots, a_{3,5}, a_{3,4}, a_{3,3})
\end{aligned}$$

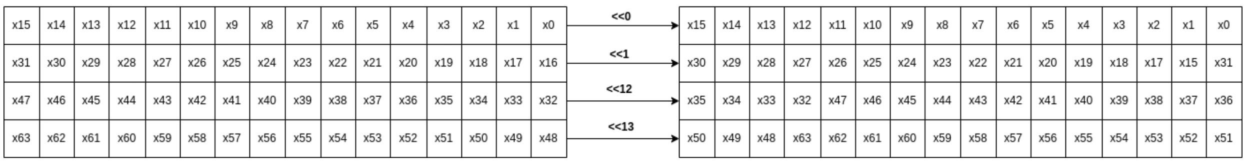

x15 x14 x13 x12 x11 x10 x9 x8 x7 x6 x5 x4 x3 x2 x1 x0
x31 x30 x29 x28 x27 x26 x25 x24 x23 x22 x21 x20 x19 x18 x17 x16
x47 x46 x45 x44 x43 x42 x41 x40 x39 x38 x37 x36 x35 x34 x33 x32
x63 x62 x61 x60 x59 x58 x57 x56 x55 x54 x53 x52 x51 x50 x49 x48
<<0
<<1
<<12
<<13
x15 x14 x13 x12 x11 x10 x9 x8 x7 x6 x5 x4 x3 x2 x1 x0
x30 x29 x28 x27 x26 x25 x24 x23 x22 x21 x20 x19 x18 x17 x15 x31
x35 x34 x33 x32 x47 x46 x45 x44 x43 x42 x41 x40 x39 x38 x37 x36
x50 x49 x48 x63 x62 x61 x60 x59 x58 x57 x56 x55 x54 x53 x52 x51

Figure 3: Above diagram show shift row operation

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-

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

Suppose $Row_i = k_{i,15}||$ ... $||k_{i,1}||\ k_{i,0}$ 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., $K_i = Row_3||Row_2||Row_1||Row_0$.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(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) ⊕ Row1
   Row'1 := Row2
   Row'2 := Row3
   Row'3 := (Row3 « 12) ⊕ 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) ⊕ RC[i]
   Using the aforementioned iterations on the revised key state, we ultimately obtain K25. The round constants RC[i] are produced using a 5-bit LFSR for all i between 0 and 24. The five bits (rc4, rc3, rc2, rc1, rc0) are left-shifted by one bit at each round, and the new value of rc0 is then assessed as rc4 xor rc2. At the beginning, 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

1. **Key Representation:**
   - The 128-bit seed key is organized into a 4×32 array, where each row contains 32 bits:
$$\begin{bmatrix} \kappa_{0,31} & \cdots & \kappa_{0,1} & \kappa_{0,0} \\ \kappa_{1,31} & \cdots & \kappa_{1,1} & \kappa_{1,0} \\ \kappa_{2,31} & \cdots & \kappa_{2,1} & \kappa_{2,0} \\ \kappa_{3,31} & \cdots & \kappa_{3,1} & \kappa_{3,0} \end{bmatrix}$$
   - Each row is represented as a 32-bit word: $\text{Row}_i = \kappa_{i,31}\|\cdots\|\kappa_{i,0}$.

2. **Subkey Extraction:**
   - At each round i $(0 \leq i \leq 24)$, the round subkey K_i is formed by extracting the 16 rightmost columns of the current key register.

3. **Key Register Update:**
   (a) **S-Box Transformation:**
$$\kappa'_{3,j}\|\kappa'_{2,j}\|\kappa'_{1,j}\|\kappa'_{0,j} := S(\kappa_{3,j}\|\kappa_{2,j}\|\kappa_{1,j}\|\kappa_{0,j}), \quad 0 \leq j \leq 7$$
   (b) **Generalized Feistel Transformation:**
$$\text{Row}'_0 := (\text{Row}_0 \ll 8) \oplus \text{Row}_1$$
$$\text{Row}'_1 := \text{Row}_2$$
$$\text{Row}'_2 := (\text{Row}_2 \ll 16) \oplus \text{Row}_3$$
$$\text{Row}'_3 := \text{Row}_0$$
   (c) **Round Constant XOR:**
   - A 5-bit round constant RC[i] is XORed with the rightmost 5 bits of Row 0:
$$(\kappa_{0,4}\|\kappa_{0,3}\|\kappa_{0,2}\|\kappa_{0,1}\|\kappa_{0,0}) \oplus RC[i]$$

- The constants RC[i] (i=0,...,24) are the same as those used in the 80-bit key schedule.

4. **Final Subkey Extraction:**
   - After the 24th round, the final subkey K_{25} is extracted from the updated key register.

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 probablity 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 probaility is : 2^{-6}
There was a attack generated on the Rectangle cipher for upto 15 rounds . The probability of the trail of 15-rounds was found to be as : 2^{-66}.

$\sharp R$	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 Operation of the cipher is simple, the security of Rectangle against cryptanalysis is difficult .
Due to rotational symmetry every trail has upto 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 .
. The value of differential branch number was = 2 and differntial uniformity = 4.

4 Integral Cryptanalysis

Integral cryptanalysis relies on analyzing how sums (or XORs) of many values propagate through the rounds of a cipher.

We design **4-round intergral distinguisher** in following way:

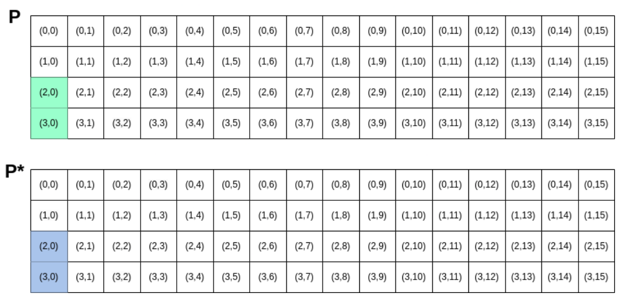
- Choose two plaintexts P and P^* such that:

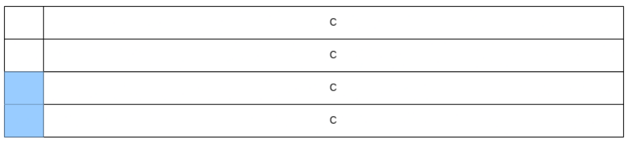
$$P_{2,0} = P_{3,0} = 0 \quad \text{and} \quad P^*_{2,0} = P^*_{3,0} = 1.$$

  The other 62 bits of P and P^* are randomly fixed to a constant value, i.e., they follow constant property.

  The difference between P and P^* is non-zero only in column 0:

$$P_{\text{Col}(0)} \oplus P^*_{\text{Col}(0)} = 1100.$$





- Encrypt P and P^* through the cipher for 4 rounds:

  After 4 rounds, the XOR difference of the outputs in certain positions equals 0 deterministically. These positions are:

$$(0,0),(1,1),(2,12),(3,13)$$

Conclusion:
At the end of 4 rounds, the XOR sum of the intermediate outputs at these bit positions is zero,i.e., they follow balanced property. This forms a 4-round integral distinguisher.

Extending to 7 Rounds Using Decryption

- We construct a set of 2^{48} plaintexts where:
  - Certain values in columns 0, 13, 14, and 15 are fixed.
  - The remaining 12 columns take all possible 2^{48} combinations,i.e., they follow All property.

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

- After encrypting the 2^{48} plaintexts through 3 rounds, the intermediate values form 2^{47} subsets. Each subset contains two values that satisfy the data condition of the 4-round integral distinguisher.

- After encrypting for 7 rounds, the XOR sum of the outputs in the same 4 bit positions remains 0:

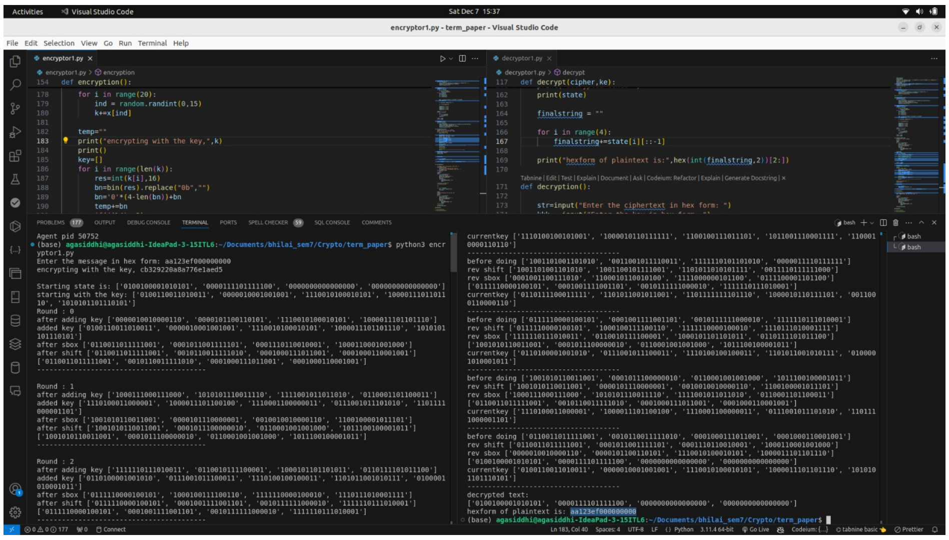
$$(0,0),(1,1),(2,12),(3,13)$$

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)	(0,8)	(0,9)	(0,10)	(0,11)	(0,12)	(0,13)	(0,14)	(0,15)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)	(1,8)	(1,9)	(1,10)	(1,11)	(1,12)	(1,13)	(1,14)	(1,15)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)	(2,8)	(2,9)	(2,10)	(2,11)	(2,12)	(2,13)	(2,14)	(2,15)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)	(3,8)	(3,9)	(3,10)	(3,11)	(3,12)	(3,13)	(3,14)	(3,15)

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.
Along with 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-



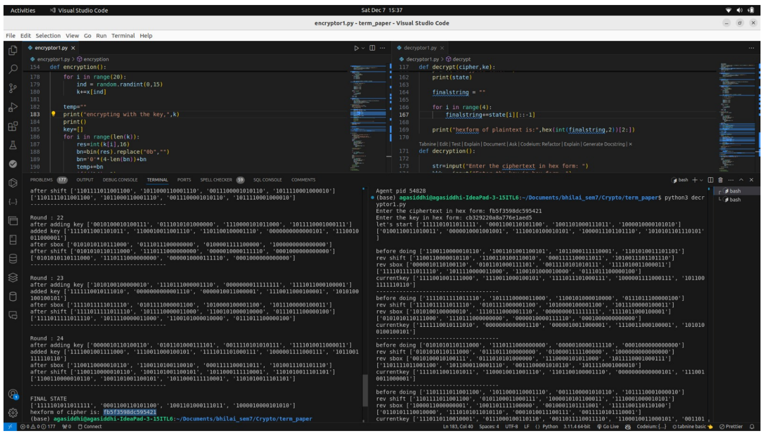
Activities
Visual Studio Code
Sat Dec 7 15:37
encryptor1.py - term_paper - Visual Studio Code
File Edit Selection View Go Run Terminal Help
encryptor1.py
decryptor1.py
PROBLEMS 177 OUTPUT DEBUG CONSOLE TERMINAL PORTS SPELL CHECKER 59 SQL CONSOLE COMMENTS
bash
Ln 183, Col 40 Spaces: 4 UTF-8 LF Python 3.11.4 64-bit Go Live Codeium: {...} tabnine basic Prettier

```python
def encryption():
    for i in range(20):
        ind = random.randint(0,15)
        k+=x[ind]

    temp=""
    print("encrypting with the key,",k)
    print()
    key=[]
    for i in range(len(k)):
        res=int(k[i],16)
        bn=bin(res).replace("0b","")
        bn='0'*(4-len(bn))+bn
        temp+=bn
```

```python
def decrypt(cipher,ke):
    print(state)

    finalstring = ""

    for i in range(4):
        finalstring+=state[i][::-1]

    print("hexform of plaintext is:",hex(int(finalstring,2))[2:])

Tabnine | Edit | Test | Explain | Document | Ask | Codeium: Refactor | Explain | Generate Docstring | ×
def decryption():

    str=input("Enter the ciphertext in hex form: ")
```

```
Agent pid 50752
(base) agasiddhi@agasiddhi-IdeaPad-3-15ITL6:~/Documents/bhilai_sem7/Crypto/term_paper$ python3 encr
yptor1.py
Enter the message in hex form: aa123ef000000000
encrypting with the key, cb329220a8a776e1aed5

Starting state is: ['0100100001010101', '0000111101111100', '0000000000000000', '0000000000000000']
starting with the key: ['0100110011010011', '0000010001001001', '1110010100010101', '10000111011011
10', '1010101101110101']
Round : 0
after adding key ['0000010010000110', '0000101100110101', '1110010100010101', '1000011101101110']
added key ['0100110011010011', '0000010001001001', '1110010100010101', '1000011101101110', '1010101
101110101']
after sbox ['0110011011111001', '0001011001111101', '0001110110010001', '1000110001001000']
after shift ['0110011011111001', '0010110011111010', '0001000111011001', '0001000110001001']
['0110011011111001', '0010110011111010', '0001000111011001', '0001000110001001']
----------------------------------------

Round : 1
after adding key ['1000111000111000', '1010101110011110', '1111001011011010', '0110001101100011']
added key ['1110100011000001', '1000011101100100', '1110001100000011', '0111001011101010', '1101111
000001101']
after sbox ['1001010110011001', '0000101110000001', '0010010010000110', '1100100001011101']
after shift ['1001010110011001', '0001011100000010', '0110001001001000', '1011100100001011']
['1001010110011001', '0001011100000010', '0110001001001000', '1011100100001011']
----------------------------------------

Round : 2
after adding key ['1111110111010011', '0110010111100001', '1000101101101011', '0110111101011100']
added key ['0110100001001010', '0111001011100011', '1110100100100011', '1101011001010111', '0100001
010001011']
after sbox ['0111110000100101', '1000100111100110', '1111110000100010', '1110111010001111']
after shift ['0111110000100101', '0001001111001101', '0010111111000010', '1111110111010001']
['0111110000100101', '0001001111001101', '0010111111000010', '1111110111010001']
----------------------------------------
```

```
currentkey ['1110100100101001', '1000010110111111', '1100100111011101', '1011001110001111', '110001
0000110110']
------------------------------------
before doing ['1001101001101010', '0011001011110011', '1111110101101010', '0000011110111111']
rev shift ['1001101001101010', '1001100101111001', '1101011010101111', '0011110111111000']
rev sbox ['0001001100111010', '1100010110010100', '1111000000101100', '0111100001101100']
['0111110000100101', '0001001111001101', '0010111111000010', '1111110111010001']
currentkey ['0110111100011111', '1101011001011001', '1101111111101110', '1000010110111101', '001100
0110000110']
------------------------------------
before doing ['0111110000100101', '0001001111001101', '0010111111000010', '1111110111010001']
rev shift ['0111110000100101', '1000100111100110', '1111110000100010', '1110111010001111']
rev sbox ['1111110111010011', '0110010111100001', '1000101101101011', '0110111101011100']
['1001010110011001', '0001011100000010', '0110001001001000', '1011100100001011']
currentkey ['0110100001001010', '0111001011100011', '1110100100100011', '1101011001010111', '010000
1010001011']
------------------------------------
before doing ['1001010110011001', '0001011100000010', '0110001001001000', '1011100100001011']
rev shift ['1001010110011001', '0000101110000001', '0010010010000110', '1100100001011101']
rev sbox ['1000111000111000', '1010101110011110', '1111001011011010', '0110001101100011']
['0110011011111001', '0010110011111010', '0001000111011001', '0001000110001001']
currentkey ['1110100011000001', '1000011101100100', '1110001100000011', '0111001011101010', '110111
1000001101']
------------------------------------
before doing ['0110011011111001', '0010110011111010', '0001000111011001', '0001000110001001']
rev shift ['0110011011111001', '0001011001111101', '0001110110010001', '1000110001001000']
rev sbox ['0000010010000110', '0000101100110101', '1110010100010101', '1000011101101110']
['0100100001010101', '0000111101111100', '0000000000000000', '0000000000000000']
currentkey ['0100110011010011', '0000010001001001', '1110010100010101', '1000011101101110', '101010
1101110101']
------------------------------------
decrypted text:
['0100100001010101', '0000111101111100', '0000000000000000', '0000000000000000']
hexform of plaintext is: aa123ef000000000
(base) agasiddhi@agasiddhi-IdeaPad-3-15ITL6:~/Documents/bhilai_sem7/Crypto/term_paper$
```



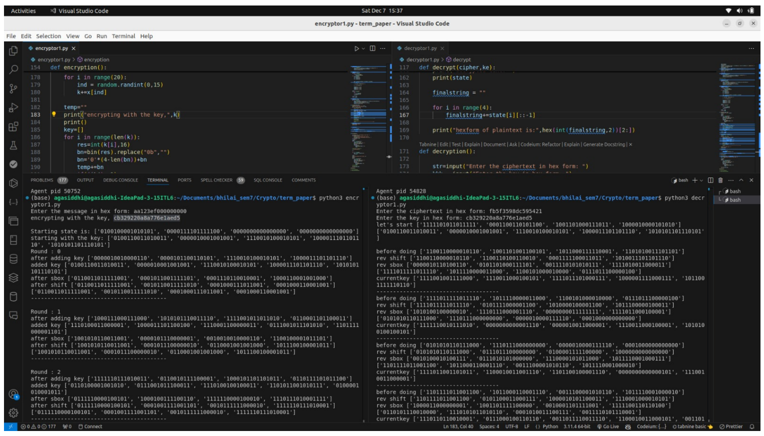
Activities
Visual Studio Code
Sat Dec 7 15:37
encryptor1.py - term_paper - Visual Studio Code
File Edit Selection View Go Run Terminal Help
encryptor1.py
decryptor1.py
PROBLEMS 177 OUTPUT DEBUG CONSOLE TERMINAL PORTS SPELL CHECKER 59 SQL CONSOLE COMMENTS
bash
Ln 183, Col 40 Spaces: 4 UTF-8 LF Python 3.11.4 64-bit Go Live Codeium: {...} tabnine basic Prettier

```python
def encryption():
    for i in range(20):
        ind = random.randint(0,15)
        k+=x[ind]

    temp=""
    print("encrypting with the key,",k)
    print()
    key=[]
    for i in range(len(k)):
        res=int(k[i],16)
        bn=bin(res).replace("0b","")
        bn='0'*(4-len(bn))+bn
        temp+=bn
```

```python
def decrypt(cipher,ke):
    print(state)

    finalstring = ""

    for i in range(4):
        finalstring+=state[i][::-1]

    print("hexform of plaintext is:",hex(int(finalstring,2))[2:])

Tabnine | Edit | Test | Explain | Document | Ask | Codeium: Refactor | Explain | Generate Docstring | ×
def decryption():

    str=input("Enter the ciphertext in hex form: ")
```

```
after shift ['1101111011001100', '1011000110001110', '0011100001010110', '1011110001000010']
['1101111011001100', '1011000110001110', '0011100001010110', '1011110001000010']
----------------------------------------

Round : 22
after adding key ['0010100010100111', '0111010101000000', '1110000101011000', '1011110001000111']
added key ['1111011001101011', '1100010011001110', '1101100100001110', '0000000000000101', '1110010
011000001']
after sbox ['0101010110111000', '0111011100000000', '0100001111100000', '1000000000000000']
after shift ['0101010110111000', '1110111000000000', '0000010000111110', '0001000000000000']
['0101010110111000', '1110111000000000', '0000010000111110', '0001000000000000']
----------------------------------------

Round : 23
after adding key ['1010100100000010', '1110111000001110', '0000000011111111', '1111011000100001']
added key ['1111110010111010', '0000000000001110', '0000010011000001', '1110011000100001', '1010100
100100101']
after sbox ['1111011111011110', '0101111000001100', '1010000100001100', '1011100000100011']
after shift ['1111011111011110', '1011110000011000', '1100101000010000', '0111011100000100']
['1111011111011110', '1011110000011000', '1100101000010000', '0111011100000100']
----------------------------------------

Round : 24
after adding key ['0000010110100110', '0101101000111101', '0011110101010111', '1111010011000011']
added key ['1111001001111000', '1110011000100101', '1111011101000111', '1000001111000111', '1011001
111110110']
after sbox ['1100110000010110', '1100110100110010', '0001111100011011', '1010011101101110']
after shift ['1100110000010110', '1001101001100101', '1011000111110001', '1101010011101101']
['1100110000010110', '1001101001100101', '1011000111110001', '1101010011101101']
----------------------------------------

FINAL STATE
['1111101011011111', '0001100110101100', '1001101000111011', '1000010000101010']
hexform of cipher is: fb5f3598dc595421
(base) agasiddhi@agasiddhi-IdeaPad-3-15ITL6:~/Documents/bhilai_sem7/Crypto/term_paper
```

```
Agent pid 54828
(base) agasiddhi@agasiddhi-IdeaPad-3-15ITL6:~/Documents/bhilai_sem7/Crypto/term_paper$ python3 decr
yptor1.py
Enter the ciphertext in hex form: fb5f3598dc595421
Enter the key in hex form: cb329220a8a776e1aed5
let's start ['1111101011011111', '0001100110101100', '1001101000111011', '1000010000101010']
['0100110011010011', '0000010001001001', '1110010100010101', '1000011101101110', '1010101101110101'
]

before doing ['1100110000010110', '1001101001100101', '1011000111110001', '1101010011101101']
rev shift ['1100110000010110', '1100110100110010', '0001111100011011', '1010011101101110']
rev sbox ['0000010110100110', '0101101000111101', '0011110101010111', '1111010011000011']
['1111011111011110', '1011110000011000', '1100101000010000', '0111011100000100']
currentkey ['1111001001111000', '1110011000100101', '1111011101000111', '1000001111000111', '101100
1111110110']
------------------------------------
before doing ['1111011111011110', '1011110000011000', '1100101000010000', '0111011100000100']
rev shift ['1111011111011110', '0101111000001100', '1010000100001100', '1011100000100011']
rev sbox ['1010100100000010', '1110111000001110', '0000000011111111', '1111011000100001']
['0101010110111000', '1110111000000000', '0000010000111110', '0001000000000000']
currentkey ['1111110010111010', '0000000000001110', '0000010011000001', '1110011000100001', '101010
0100100101']
------------------------------------
before doing ['0101010110111000', '1110111000000000', '0000010000111110', '0001000000000000']
rev shift ['0101010110111000', '0111011100000000', '0100001111100000', '1000000000000000']
rev sbox ['0010100010100111', '0111010101000000', '1110000101011000', '1011110001000111']
['1101111011001100', '1011000110001110', '0011100001010110', '1011110001000010']
currentkey ['1111011001101011', '1100010011001110', '1101100100001110', '0000000000000101', '111001
0011000001']
------------------------------------
before doing ['1101111011001100', '1011000110001110', '0011100001010110', '1011110001000010']
rev shift ['1101111011001100', '0101100011000111', '1000010101100011', '1110001000010101']
rev sbox ['1000011000000001', '1001101111100000', '0010001101111001', '1111100110110100']
['0110101110010000', '1110101011010110', '0001010011100111', '0011110101110001']
currentkey ['1110110110010001', '0111000100110110', '0011011110011110', '1100010011000101', '001101
```



```
def encryption():
    for i in range(20):
        ind = random.randint(0,15)
        k+=x[ind]

    temp=""
    print("encrypting with the key,",k)
    print()
    key=[]
    for i in range(len(k)):
        res=int(k[i],16)
        bn=bin(res).replace("0b","")
        bn='0'*(4-len(bn))+bn
        temp+=bn
```

```
def decrypt(cipher,ke):
    print(state)

    finalstring = ""

    for i in range(4):
        finalstring+=state[i][::-1]

    print("hexform of plaintext is:",hex(int(finalstring,2))[2:])

def decryption():

    str=input("Enter the ciphertext in hex form: ")
```

```
Agent pid 50752
(base) agasiddhi@agasiddhi-IdeaPad-3-15ITL6:~/Documents/bhilai_sem7/Crypto/term_paper$ python3 encr
yptor1.py
Enter the message in hex form: aa123ef000000000
encrypting with the key, cb329220a8a776e1aed5

Starting state is: ['0100100001010101', '0000111101111100', '0000000000000000', '0000000000000000']
starting with the key: ['0100110011010011', '0000010001001001', '1110010100010101', '10000111011011
10', '1010101101110101']
Round : 0
after adding key ['0000010010000110', '0000101100110101', '1110010100010101', '1000011101101110']
added key ['0100110011010011', '0000010001001001', '1110010100010101', '1000011101101110', '1010101
101110101']
after sbox ['0110011011111001', '0001011001111101', '0001110110010001', '1000110001001000']
after shift ['0110011011111001', '0010110011111010', '0001000111011001', '0001000110001001']
['0110011011111001', '0010110011111010', '0001000111011001', '0001000110001001']
----------------------------------------

Round : 1
after adding key ['1000111000111000', '1010101110011110', '1111001011011010', '0110001101100011']
added key ['1110100011000001', '1000011101100100', '1110001100000011', '0111001011101010', '1101111
000001101']
after sbox ['1001010110011001', '0000101110000001', '0010010010000110', '1100100001011101']
after shift ['1001010110011001', '0001011100000010', '0110001001001000', '1011100100001011']
['1001010110011001', '0001011100000010', '0110001001001000', '1011100100001011']
----------------------------------------

Round : 2
after adding key ['1111110111010011', '0110010111100001', '1000101101101011', '0110111101011100']
added key ['0110100001001010', '0111001011100011', '1110100100100011', '1101011001010111', '0100001
010001011']
after sbox ['0111110000100101', '1000100111100110', '1111110000100010', '1110111010001111']
after shift ['0111110000100101', '0001001111001101', '0010111111000010', '1111110111010001']
['0111110000100101', '0001001111001101', '0010111111000010', '1111110111010001']
----------------------------------------
```

```
Agent pid 54828
(base) agasiddhi@agasiddhi-IdeaPad-3-15ITL6:~/Documents/bhilai_sem7/Crypto/term_paper$ python3 decr
yptor1.py
Enter the ciphertext in hex form: fb5f3598dc595421
Enter the key in hex form: cb329220a8a776e1aed5
let's start ['1111101011011111', '0001100110101100', '1001101000111011', '1000010000101010']
['0100110011010011', '0000010001001001', '1110010100010101', '1000011101101110', '1010101101110101'
]

before doing ['1100110000010110', '1001101001100101', '1011000111110001', '1101010011101101']
rev shift ['1100110000010110', '1100110100110010', '0001111100011011', '1010011101101110']
rev sbox ['0000010110100110', '0101101000111101', '0011110101010111', '1111010011000011']
['1111011111011110', '1011110000011000', '1100101000010000', '0111011100000100']
currentkey ['1111001001111000', '1110011000100101', '1111011101000111', '1000001111000111', '101100
1111110110']
-----------------------------------
before doing ['1111011111011110', '1011110000011000', '1100101000010000', '0111011100000100']
rev shift ['1111011111011110', '0101111000001100', '1010000100001100', '1011100000100011']
rev sbox ['1010100100000010', '1110111000001110', '0000000011111111', '1111011000100001']
['0101010110111000', '1110111000000000', '0000010000111110', '0001000000000000']
currentkey ['1111110010111010', '0000000000001110', '0000010011000001', '1110011000100001', '101010
0100100101']
-----------------------------------
before doing ['0101010110111000', '1110111000000000', '0000010000111110', '0001000000000000']
rev shift ['0101010110111000', '0111011100000000', '0100001111100000', '1000000000000000']
rev sbox ['0010100010100111', '0111010101000000', '1110000101011000', '1011110001000111']
['1101111011001100', '1011000110001110', '0011100001010110', '1011110001000010']
currentkey ['1111011001101011', '1100010011001110', '1101100100001110', '0000000000000101', '111001
0011000001']
-----------------------------------
before doing ['1101111011001100', '1011000110001110', '0011100001010110', '1011110001000010']
rev shift ['1101111011001100', '0101100011000111', '1000010101100011', '1110001000010101']
rev sbox ['1000011000000001', '1001101111100000', '0010001101111001', '1111100110110100']
['0110101110010000', '1110101011010110', '0001010011100111', '0011110101110001']
currentkey ['1110110110010001', '0111000100110110', '0011011110011110', '1100010011000101', '001101
```

6 Application

Encrypt message in QR codes using RECTANGLE

- encryptor.py - contains the encryption of RECTANGLE cipher
- decryptor.py - contains the decryption of RECTANGLE cipher
- generate_qr.py - encrypts the message and embeds it in QR code
- decrypt_qr.py - scans and retrieves the plaintext from the QR-code

Figure 8: Above QR Codes contain encrypted messages

7 Conclusion

The RECTANGLE cipher is a lightweight block cipher designed with a bit-slice technique based on an SP-network, enabling efficient and fast execution. Its design achieves exceptional performance in both software and hardware environments, making it highly adaptable for various applications. The cipher incorporates 16 parallel 4×4 S-Boxes in its substitution layer and employs three rotations in the permutation layer. This structure ensures low-cost implementation in hardware and competitive speed in software. RECTANGLE's strong security performance stems from the deliberate selection of its S-Boxes and the asymmetric design of its permutation layer. Its simple yet innovative design has the potential to inspire new challenges and advancements in cryptography. Overall, the RECTANGLE cipher offers a robust combination of flexibility, efficiency, and security, making it a noteworthy contribution to the field.

8 References

- Research Paper source- https://eprint.iacr.org/2014/084.pdf