Term Paper

Saturnin Block Cipher

Ashitabh Misra[1]

Indian Institute of Technology Bhilai,India, ashitabhm@iitbhilai.ac.in

Abstract. This document is the analysis of block cipher called **Saturnin**[1]. This term paper highlights certain key aspects the main **Encryption** block. Claims regarding Active SBOX's and differential analysis has been verified using MILP in this work. Hardware Analysis of Permutation Layer which includes maximum operating frequency and Gate Count on FPGA and ASIC Architecture. A new impossible trail in General Super-Round structure has been proposed. In-depth analysis of Biclique attack on PRESENT-80.

Keywords: Saturnin · Zero-Sum · Impossible Differential · MILP · Biclique · Meet in the middle attack · SBox Properties · Hardware Analysis · Biclique Present-80

Contents

[1]https://project.inria.fr/saturnin/

Contributions

0.1 Contributions on Saturnin

The contributions of this paper are as follows:

1. Prove Zero-Sum property on Saturnin Encryption.
2. New implementation of Encryption, different from the one provided by the authors.
3. Identify Impossible Trails in the Cipher.
4. Two different MILP formulations for the cipher.
5. 1-1 DDT/LAT for cipher SBOX's
6. Cipher Description
7. Calculating Branch Number of Linear Operation.

0.2 Contribution on Biclique attack on PRESENT-80

The contributions of this paper are as follows:

1. Meet-in-the-Middle Attack
2. Basic Bi-Clique attack(drawbacks of **MITM** and Bi-Clique solves them).
3. Sieve in the middle structure.
4. Bi-clique attack on **PRESENT-80**

1 Cipher Specifications

Note We will see two interpretations of the cipher and constantly move back and forth to prove different properties. The first interpretation will be referred to as Cube Representation and the second one as Square representation

Saturnin is an even number of rounds, numbered from 0. There is a **Super-Round** which consists of rounds $(2t, 2t+1)$ (Unless explicitly specified, I will refer to **Normal Rounds** as Rounds).The block cipher has a 256-bit internal state **X** and a 256-bit key state **K**. Each will be represented as $(4x4x4)$ **Cube**, each nibble being 4-bits.

Parameters

The block cipher has two input parameters:

- ***R***: Number of super rounds, the authors suggest $R\epsilon\{10, \cdots, 31\}$

- ***D***: A 4-bit *Domain Separator*.

We will not be looking at the Mode of Operations and focusing are attention on the Encryption Cipher as a standalone.

Initialisation

X and K are internal state and 256-bit Key respectively.
R: $R_4R_3R_2R_1R_0$
D: $D_3D_2D_1D_0$

Two Round Constants RC_0, RC_1 are intialised as the following:

$$\underbrace{1\cdots1}_{\text{7 ones}}\underbrace{R_4\cdots R_0}_{\text{R}}\underbrace{D_3\cdots D_0}_{\text{D}}$$

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\sigma_0(x)$	0	6	14	1	15	4	7	13	9	8	12	5	2	10	3	11
$\sigma_1(x)$	0	9	13	2	15	1	11	7	6	4	5	3	8	12	10	14

Figure 2: Saturnin uses 2 SBOX's

1.1 Basic Terminology

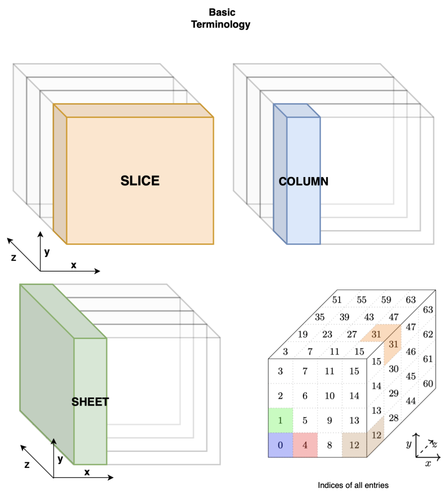


Figure 1: Basic Terminology

The nomenclature mentioned in Figure 1 will come in handy so please do take note of it. The nibble indexing will follow from the Figure 1(Bottom right) as well, unless explicitly mentioned otherwise.

1.2 Substitution Layer

```python
def s_layer(state):
    new_state= [0 for index in range(64)]
    for index in range(64):
        if index%2 ==0:
            new_state[index] = sbox_0[state[index]]
        if index%2 ==1:
            new_state[index] = sbox_1[state[index]]
    return new_state
```

Listing 1: Substitution layer

Saturnin uses two SBOX's σ_0 is used for all even indices and σ_1 is used for all odd indices. The even/odd numbering is decided by Figure 1 (Bottom Right).

1.3 Permutation Layer

Saturnin uses two different kind of Permutations depending upon the Round Number(r). Refer to Algorithm in Figure ??

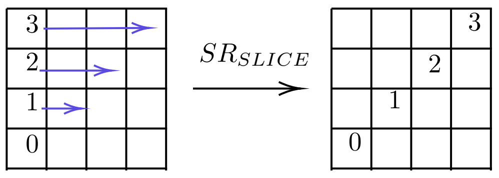
Algorithm 1 Permutation layer of Saturnin

```
Require: $r$
  if $r(mod4) == 0$ : then
    $pass$ {No Shifting occurs}
  else if $r(mod4) == 1)$ : then
    $(x, y, z) \rightarrow (x + y(mod4), y, z)$ {Shifting across a Slice}
  else if $r(mod4) == 3$ : then
    $(x, y, z) \rightarrow (x, y, z + y(mod4))$ {Shifting across a Sheet}
  end if
```

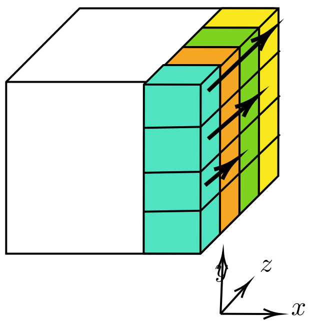
```python
def perm_layer(state,r=1):
    new_state = [0 for i in range(64)]
    if r%4 == 1:
        for i in range(64):
            z = int(i/16)
            x = int(i/4) % 4
            y = int(i%4)

            x_dash = (x+y)%4
            y_dash = y
            z_dash = z

            new_index = z_dash*16 + x_dash*4 +y_dash

            new_state[new_index] = state[i]
    if r%4 == 3:
        for i in range(64):
            z = int(i/16)
            x = int(i/4) % 4
            y = int(i%4)

            x_dash = x
            y_dash = y
            z_dash = (y+z)%4

            new_index = z_dash*16 + x_dash*4 +y_dash

            new_state[new_index] = state[i]
    if r%2 ==0:
        new_state = [state[index] for index in range(64)]

    return new_state
```

Listing 2: Permutation layer

The Square representation of the cipher will have a slightly different permutation layer. We will discuss that in Section 2.



(a) Permutation across a Slice



(b) Permutation across a Sheet

Figure 3: Saturnin Permutation Layer

1.4 Linear Layer

M operation is done over each column in the Cube representation. The operation is described in Figure 4 and α operation is described in Figure

$$M : \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} \mapsto \begin{pmatrix} \alpha^2(a) \oplus \alpha^2(b) \oplus \alpha(b) \oplus c \oplus d \\ a \oplus \alpha(b) \oplus b \oplus \alpha^2(c) \oplus c \oplus \alpha^2(d) \oplus \alpha(d) \oplus d \\ a \oplus b \oplus \alpha^2(c) \oplus \alpha^2(d) \oplus \alpha(d) \\ \alpha^2(a) \oplus a \oplus \alpha^2(b) \oplus \alpha(b) \oplus b \oplus c \oplus \alpha(d) \oplus d \end{pmatrix}$$

Figure 4: Linear Layer

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} .$$

Figure 5: α operation

Linear Layer Code

```
def mc_a(a,b,c,d):
    t1  = np.matmul(mds_2,a)%2
    t2 = np.matmul(mds_2,b)%2
    t3 = np.matmul(mds,b)%2
    t4 = c
    t5 = d
    answer_temp = xor([t1,t2,t3,t4,t5])

    return answer_temp%2


def mc_b(a,b,c,d):
    t1  = a
    t2 = np.matmul(mds,b)%2
    t3 = b
    t4 = np.matmul(mds_2,c)%2
    t5 = c
    t6 = np.matmul(mds_2,d)%2
    t7 = np.matmul(mds,d)%2
    t8 = d
    answer_temp = xor([t1,t2,t3,t4,t5,t6,t7,t8])

    return answer_temp%2

def mc_c(a,b,c,d):
    t1  = a
    t2 = b
    t3 = np.matmul(mds_2,c)%2
    t4 = np.matmul(mds_2,d)%2
    t5 = np.matmul(mds,d)%2

    answer_temp = xor([t1,t2,t3,t4,t5])
    return answer_temp%2

def mc_d(a,b,c,d):
    t1  = np.matmul(mds_2,a)%2
    t2 = a
    t3 = np.matmul(mds_2,b)%2
    t4 = np.matmul(mds,b)%2
    t5 = b
    t6 = c
    t7 = np.matmul(mds,d)%2
    t8 = d
    answer_temp = xor([t1,t2,t3,t4,t5,t6,t7,t8])
    return answer_temp%2
```

Listing 3: M Operations

```
def linear_layer(state):
    new_state = [0 for i in range(64)]
    i =0
    for column in range(16):
#          print(i)
        a  = column*4
        b  = column*4+1
        c  = column*4+2
        d  = column*4+3

        a_bin= get_bin(state[a])
        b_bin= get_bin(state[b])
        c_bin= get_bin(state[c])
        d_bin= get_bin(state[d])
#          print(c_bin)

        a_dash = mc_a(a_bin,b_bin,c_bin,d_bin)
        b_dash = mc_b(a_bin,b_bin,c_bin,d_bin)
        c_dash = mc_c(a_bin,b_bin,c_bin,d_bin)
        d_dash = mc_d(a_bin,b_bin,c_bin,d_bin)

        a_num = get_num(a_dash)
        b_num = get_num(b_dash)
        c_num = get_num(c_dash)
        d_num = get_num(d_dash)

        new_state[column*4] = a_num
        new_state[column*4+1] = b_num
        new_state[column*4+2] = c_num
        new_state[column*4+3] = d_num
        i+=1
```

```python

    return new_state
```

Listing 4: Linear layer

1.5 Calculating Branch Number of Linear Layer

The authors did not provide the Branch number for this Linear Layer Operation. However, they did provide a branch number for their Square representation which would apply over 64 bits instead of 16 bits like Cube representation(Section 2). So I have calculated it for Cube representation to do MILP formulation later.

```python


def return_num_active_sbox(states_active):
    num_active_sbox =0
    for state in states_active:
        if '1' in state:
            num_active_sbox+=1
    return num_active_sbox


def min_hamming_weight():
    branch_number = 9999
    branch_value_dict = {'5':0,'6':0,'7':0,'8':0}
    for i in range(1,2**16):
        curr_bin = '{0:016b}'.format(i)
        input_nibbles = textwrap.wrap(curr_bin,4)
        input_wt = return_num_active_sbox(input_nibbles)
        output_nibbles = column_operation(input_nibbles)
        output_wt = return_num_active_sbox(output_nibbles)
        if input_wt+output_wt < branch_number:
            branch_number = input_wt + output_wt
        branch_value_dict[str(input_wt+output_wt)]+=1
    print(branch_number,branch_value_dict)
    return branch_number,branch_value_dict
```

Listing 5: Calculating Branch NUmber of Linear Layer

Branch Number: 5

The code has been omitted for brevity, full code is present in the Codebook. I have used a brute force approach to calculate the minimum hamming weight over all possible inputs.
[2]

Listing 6: Calculating Branch Number of Linear Layer

1.6 Round Constant Addition

We can refer to the original paper[2] for this. It does not play an important role in our analysis hence I have omitted it.It can be also referred to in the Class presentation 2 where I went into detail about it.

[2]I would like to highlight this section

1.7 Round Key Addition

The following code in Listing 8 is fairly intuitive, the pseudocode is mentioned in Algorithm 2.

Algorithm 2 Round Key Addition in Saturnin

```
Require: $K$
  if $r(mod4) == 0$ : then
    $pass$ {No Addition occurs}
  else if $r(mod4) == 1)$ : then
    State $\oplus$ Key
  else if $r(mod4) == 3$ : then
    $State[j] \oplus Key[(j + 20)\%64]$
  end if
```

```python
def add_round_key(state,key,r):
    init_state = [0 for x in range(64)]
    if r%4 == 3:
        for i in range(64):
            init_state = state[i] ^ key[i]
        return init_state
    if r%4 == 1:
        for i in range(64):
            init_state[i] = state[i] ^ key[(i+20)%64]
        return init_state
    if r%4 ==0:
        init_state = [state[x] for x in range(64)]
        return init_state
```

Listing 7: Round Key Addition

1.8 Outline of the cipher

Figure 6 constitutes the first 4 rounds of Saturnin. Please note that the authors provided a bit-sliced version of their cipher. I have implemented it as the Specifications they have provided.

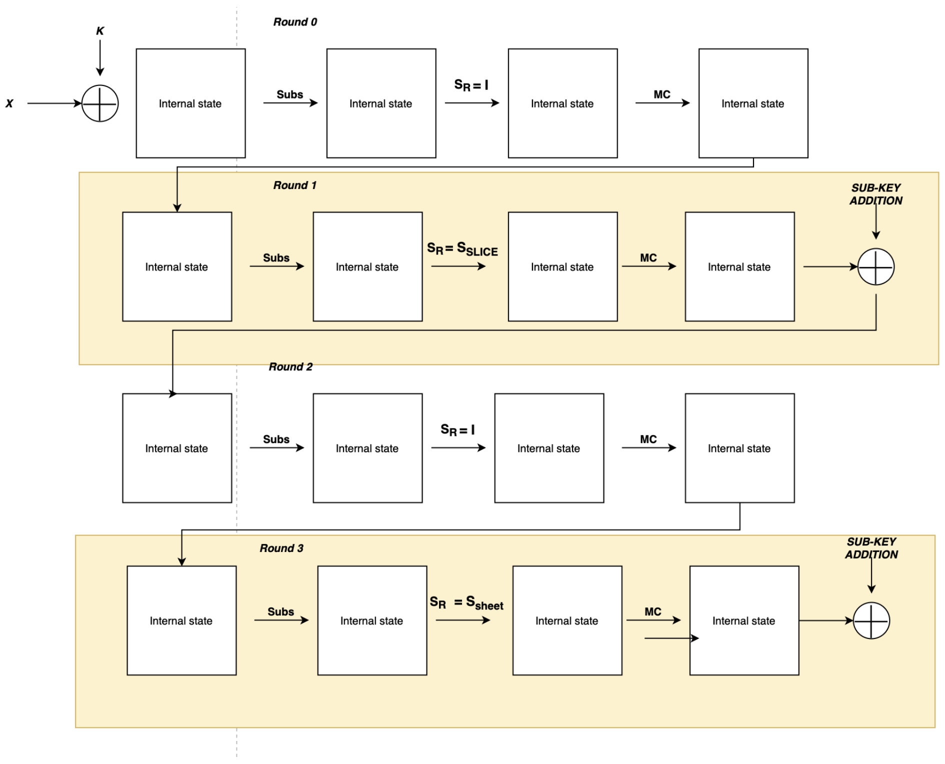


Figure 6: Outline of Saturnin

2 Re-interpretation of Saturnin

The authors collapsed the design from 3D to 2D(register representation) and changed the operations accordingly. Let x be the index within each nibble and N be index of each nibble the mapping is done in the following manner.

$$\text{Bits(in 2D):}(4x, N), (4x + 1, N), (4x + N), (4x + 3, N) \text{ are mapped to}$$

$$\text{Nibbles(in 3D): } (N mod\, 4, x, \lfloor \tfrac{N}{4} \rfloor)$$

The Substitution layer and Round Key addition only have minor changes, just change of corresponding indices.I have provided a further mapping to 1-D which is somewhat trivial and shall not consider an original contribution. However, it was a necessary operation for implementation in Icarus Verilog(open source alternative to **Vivado**) since the latest version does not support Multi-Dimensional Arrays.

```

    new_state = [0 for i in range(64)]
    if r%4 == 1:
        for i in range(64):
            z = int(i/16)
            x = int(i/4) % 4
            y = int(i%4)

            x_dash = (x+y)%4
            y_dash = y
            z_dash = z

            new_index = z_dash*16 + x_dash*4 +y_dash

            new_state[new_index] = state[i]
    if r%4 == 3:
        for i in range(64):
            z = int(i/16)
            x = int(i/4) % 4
            y = int(i%4)

            x_dash = x
            y_dash = y
            z_dash = (y+z)%4

            new_index = z_dash*16 + x_dash*4 +y_dash

            new_state[new_index] = state[i]
    if r%2 ==0:
        new_state = [state[index] for index in range(64)]

    return new_state
```

Listing 8: Permutation operation in 1D

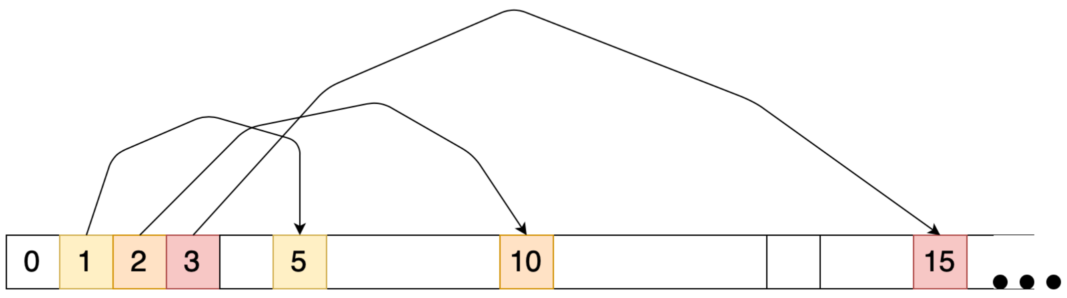


Figure 7: 1-D permutation operation

2.1 Structure using Super-SBox

The Super-SBox structure constitutes of the following things:

- 2 rounds $(2r, 2r + 1)$ are composed into 1 Super-Round.
- The SBox is applied to each column of the internal structure.
- Linear Layer(L_{64}) in terms of Original Operations would look like, $SR^{-1}_{2t+1} \circ MC \circ SR_{2t+1}$
- The Linear Layer, if t *is even*, would look like 4 independent applications on each **slice**.
- The Linear Layer, if t *is odd*, would look like 4 independent applications on each **sheet**.
- The **Branch number** of L_{64} is 5.

The mapping is made more clear in the Figure **ADD**.Please note, **Register representation** is different from **Square representation** although both are 2D.

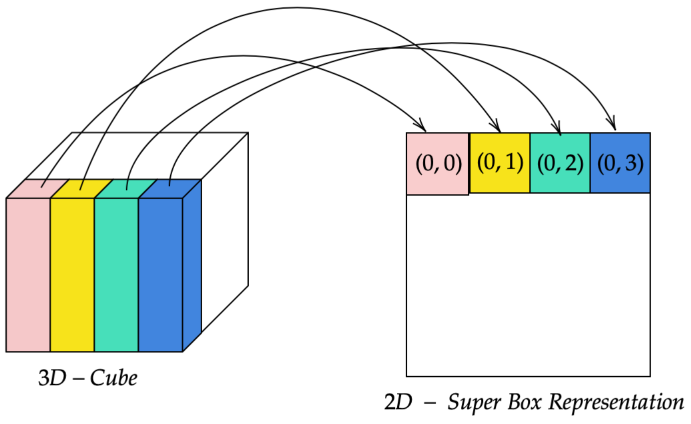


Figure 8: Mapping Cube representation to Square representation

2.2 Outline Square representation Saturnin

Taking into account that when for Super-Round $(2t, 2t + 1)$:

1. When t is even, L_{64} is applied along the slices.
2. When t is odd, L_{64} is applied along sheets.

The following outline (mentioned in Figure **??**) can be generated.

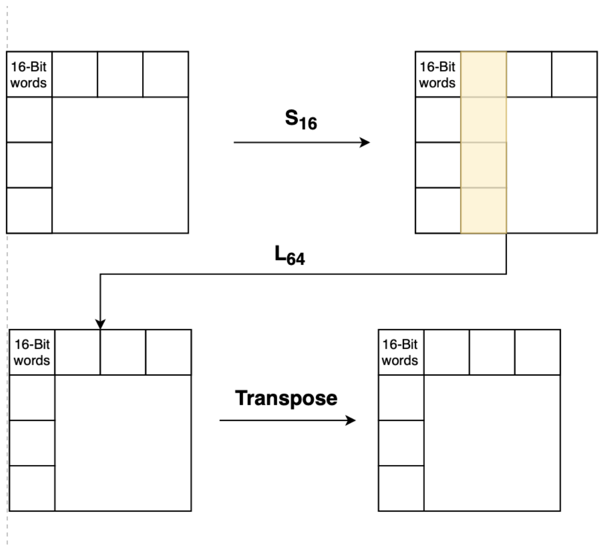


Figure 9: Super-SBox representation of Saturnin

3 Zero-Sum Property

The maximum degree of component function of either SBox's is 3. So using that to prove the Zero-Sum property. So the number of plaintext-ciphertext pairs needed are 2^{3^r+1}

```


key= [3, 6, 12, 1, 11, 9, 1, 10, 3, 7, 3, 11, 0, 6, 3, 0, 6, 14, 9, 11, 13,
    2, 7, 10, 10, 2, 6, 14, 2, 9, 13, 11, 11, 2, 11, 5, 5, 10, 14, 5, 5, 11,
       10, 5, 15, 12, 15, 14, 15, 15, 1, 9, 14, 9, 8, 5, 11, 1, 0, 5, 11, 3,
     14, 12]


init_list = [x for x in range(0,16)]

pt_list = [[0 for i in range(64)] for j in range(16)]

for i in range(64):
    random.shuffle(init_list)
    for j in range(16):
        pt_list[j][i] = init_list[j]


ct_list = []
for i in range(16):
    ct_list.append(round_fn(pt_list[i],key,0))


for i in range(64):
    curr = 0
    for j in range(16):
        curr ^= ct_list[j][i]
    print(curr, end=' ')
```

Listing 9: Zero-Sum property

After running we can see that the result comes out to be 0 for Number of rounds 1 and 2. For higher rounds it is not computationally feasible to prove it.

4 Impossible Differential

The authors have provided two trails one of which I will be showing though both work on the same intuition. I have also added a second Impossible trial using the **Square representation** which is different from the ones mentioned in the paper.

The authors use the fact that a Linear Layer of Saturnin looks like $SR^{-1}_{2t+1} \circ MC \circ SR_{2t+1}$. **MD** is $SR^{slice} \circ MC \circ SR_{slice}$, **MC** is $SR^{sheet} \circ MC \circ SR_{sheet}$. Recall that,$L_{64}$ only mixing bits along a column. Hence there can be no active bits outside the Sheet.
Using the above logic the authors proposed the trail shown in Figure 10.

4.1 Impossible trail using Super-Box representation 3.5 Super-Round

Since we have already seen that each super Round is 2 rounds in the normal cipher this impossible trail can be expanded into **7 round** Impossible trail for Saturnin. The trail is show in Figure 11.

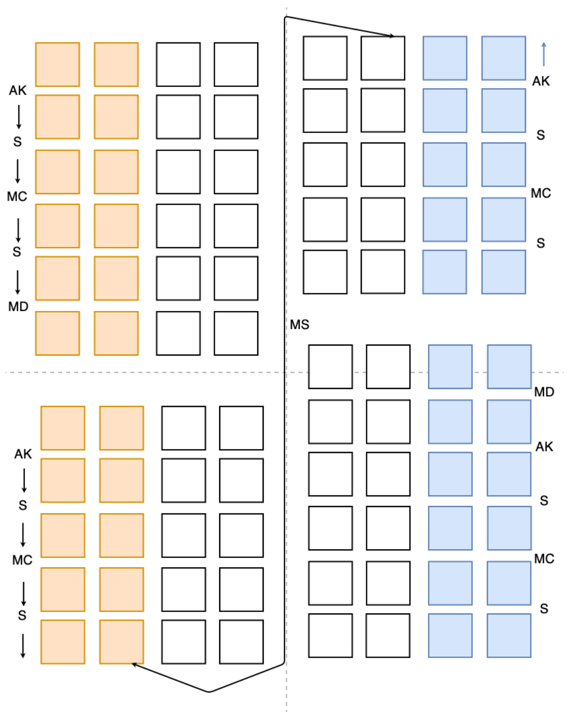


Figure 10: Impossible Differential Provided by authors

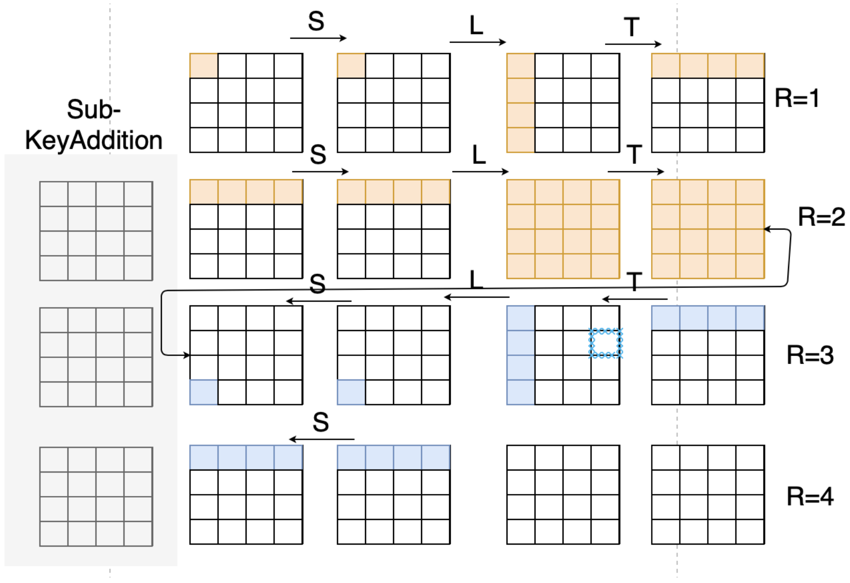


Figure 11: 3.5 Super Round Trail

2 [1267010382]	4 [329513419]	6 [56414502]	8 [10192982]	10 [1143106]
12 [283372]	14 [20740]	16 [347147]	18 [22466]	20 [24586]
22 [927]	24 [6530]	26 [164]	28 [276]	32 [25871]
34 [1331]	36 [2018]	38 [40]	40 [476]	42 [4]
44 [26]	46 [2]	48 [84]	50 [8]	52 [8]
64 [858]	66 [32]	68 [68]	72 [4]	80 [6]

Figure 12: Different Entries in S_{16} and their frequencies

[3]

5 Differential Analysis of Saturnin

The authors claim that there will be at least 25 active SBOX's in 4 Super-Rounds(**We will prove this claim in Section 6.3**). For now we will use it to perform some basic Differential analysis. The original paper consists of a table with some value frequencies of entries in S_{16}, Super SBox.

The highest entry in S_{16} is 80, which implies the highest differential probability in S_{16} is

$$(\frac{80}{2^{16}}) = 2^{-9.68} \tag{1}$$

Given the fact that in 4 rounds we will have atleast 25 active SBOX's is that the probability of best differential characteristic is

$$(\frac{80}{2^{16}})^{25} = 2^{-241.9} \tag{2}$$

6 MILP Formulation of Saturnin

I have performed MILP formulation in 2 ways. First formulating the Cube representation and Second formulating the Square representation.

6.1 Independent Analysis of MILP equations

MILP: Even Rounds

$$\begin{bmatrix} x_0 & x_4 & x_8 & x_{12} \\ x_1 & x_5 & x_9 & x_{13} \\ x_2 & x_6 & x_{10} & x_{14} \\ x_3 & x_7 & x_{11} & x_{15} \end{bmatrix} \rightarrow \begin{bmatrix} y_0 & y_4 & y_8 & y_{12} \\ y_1 & y_5 & y_9 & y_{13} \\ y_2 & y_6 & y_{10} & y_{14} \\ y_3 & y_7 & y_{11} & y_{15} \end{bmatrix}$$

Consider first column,

$$x_0 + x_1 + x_2 + x_3 + y_0 + y_1 + y_2 + y_3 \geq B * d_0 \tag{3}$$

[3] I would like to highlight this section

Table 1: MILP results for Cube representation

Number of Rounds	Number of Active SBOX's
1	1
2	5
3	9
4	10
5	14
6	15
7	19
8	20
9	24
10	25
11	29
12	30

MILP: Round r s.t r $\equiv$ 1mod 4(Slice permutation)

$$\begin{bmatrix} x_0 & x_4 & x_8 & x_{12} \\ x_1 & x_5 & x_9 & x_{13} \\ x_2 & x_6 & x_{10} & x_{14} \\ x_3 & x_7 & x_{11} & x_{15} \end{bmatrix} \rightarrow \begin{bmatrix} x_4 & x_8 & x_{12} & x_0 \\ x_9 & x_{13} & x_1 & x_5 \\ x_{14} & x_2 & x_6 & x_{10} \\ x_3 & x_7 & x_{11} & x_{15} \end{bmatrix} \rightarrow \begin{bmatrix} y_0 & y_4 & y_8 & y_{12} \\ y_1 & y_5 & y_9 & y_{13} \\ y_2 & y_6 & y_{10} & y_{14} \\ y_3 & y_7 & y_{11} & y_{15} \end{bmatrix}$$

Consider first column,

$$x_4 + x_9 + x_{14} + x_3 + y_0 + y_1 + y_2 + y_3 \geq B * d_i \tag{4}$$

MILP: Round r s.t r $\equiv$ 3mod 4(Sheet permutation)

$$\begin{bmatrix} x_{15} & x_{31} & x_{47} & x_{63} \\ x_{14} & x_{30} & x_{46} & x_{62} \\ x_{13} & x_{29} & x_{45} & x_{61} \\ x_{12} & x_{28} & x_{44} & x_{60} \end{bmatrix} \rightarrow \begin{bmatrix} x_{31} & x_{47} & x_{63} & x_{15} \\ x_{46} & x_{62} & x_{14} & x_{30} \\ x_{61} & x_{13} & x_{29} & x_{45} \\ x_{12} & x_{28} & x_{44} & x_{60} \end{bmatrix} \rightarrow \begin{bmatrix} y_{15} & y_{31} & y_{47} & y_{63} \\ y_{14} & y_{30} & y_{46} & y_{62} \\ y_{13} & y_{29} & y_{45} & y_{61} \\ y_{12} & y_{28} & y_{44} & y_{60} \end{bmatrix}$$

Consider first column,

$$x_{31} + x_{46} + x_{61} + x_{12} + y_{15} + y_{14} + y_{13} + y_{12} \geq B * d_i \tag{5}$$

6.2 MILP of Cube representation

[4] The L.P files are given in the folder, to generate them please refer to *milp_saturnin.py*. Please note the authors have **not** provided any details on this formulation, so this can be considered an original contribution. The results are mentioned in Table 1.

6.3 MILP of SQUARE representation

[5]

Please run file square_milp_saturnin.py to reproduce these results.

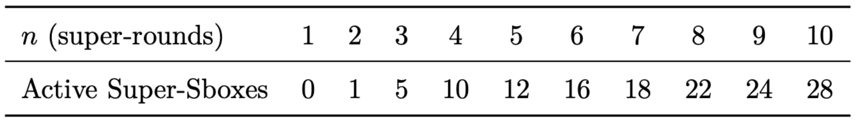
The results of this MILP formulation are mentioned in Table 6.3. If you see the corresponding entry for 4 Super-Rounds. You will see that indeed minimum number of SBOX's possible is 25.

[4] I would like to highlight this section

[5] I would like to highlight this section

Table 2: MILP results for Super-Round Interpretation of Saturnin

Number of Super-Rounds	Number of Active SBOX's
1	1
2	5
3	9
4	25
5	26
6	30
7	34
8	50
9	51
10	55
11	59
12	75



n (super-rounds)	1	2	3	4	5	6	7	8	9	10
Active Super-Sboxes	0	1	5	10	12	16	18	22	24	28

Figure 13: MILP formulation in Original Paper of Super-Rounds

Please Note that the authors MILP formulation(Figure 13)of Super-Rounds is consistently showing looser bounds than the results I am getting. However, my results are in line with the Differential Crypt-analysis in Section 5 of their paper. Whereas according to their results for Round 4, their analysis will not work. Authors do mention that these are very loose lower bounds.

So further analysis might be interesting.

7 Hardware Analysis of Permutation Layer

For Vivado I have used Artix-7 AC701 Evaluation Platform, it was the only one with enough number of I/O ports to accommodate this cipher. The results are given in Table 3.

Table 3: VIVADO Results

Parameters	Results
Clock Frequency	462.963 MHz
Worse Pulse Width Slack	0.005ns

For ASIC Implementation the results are given in Table 4.

Table 4: Leonardo Results:ASIC Implementation

Parameters	Results
Gate Count	1638
GE(Gate Equivalence)	1,297
Clock Frequency	46678.8 MHz

8 Properties of the SBOX

Saturnin Consists of 2 SBOX's(Figure 2). So the analysis is done on both of them.

1-1 DDT

Table 5: 1-1 DDT of σ_0

input\ output	0001	0010	0100	1000
0001	2	0	0	4
0010	4	0	0	2
0100	0	4	0	0
1000	0	2	4	0

Table 6: 1-1 DDT of σ_1

input\ output	0001	0010	0100	1000
0001	0	2	4	0
0010	0	4	2	0
0100	0	0	0	4
1000	4	0	0	2

As is evident from 1-1 DDT Tables of Both SBox's, there are no Good inputs or Outputs.

1-1 LAT

Table 7: 1-1 LAT of σ_0

input/output	0001	0010	0100	1000	
0001	0	0	0	-1/8	
0010	1/8	1/4	0	0	
0100	0	1/8	0	1/4	
1000	-1/4	0	1/8	0	

Table 8: 1-1 LAT of σ_1

input/output	0001	0010	0100	1000	
0001	0	0	0	-1/8	
0010	1/8	1/4	0	0	
0100	0	1/8	0	1/4	
1000	-1/4	0	1/8	0	

8.1 Boomerang Connectivity Table

BCT for σ_0

Δ_0/∇_0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16
1	16	6	0	0	0	0	2	0	4	6	2	2	0	0	0	2
2	16	4	0	0	0	2	0	2	6	6	0	0	0	2	2	0
3	16	6	2	2	2	0	0	0	6	4	0	0	2	0	0	0
4	16	0	4	4	0	0	0	0	0	2	0	2	10	0	6	4
5	16	0	4	4	10	0	4	6	0	2	2	0	0	0	0	0
6	16	2	0	2	0	2	0	2	0	0	2	2	0	0	2	2
7	16	2	2	0	0	0	2	2	0	0	2	2	0	2	2	0
8	16	0	10	0	4	0	6	0	0	2	0	0	4	2	4	0
9	16	0	0	0	0	2	2	0	2	0	0	2	2	2	2	2
10	16	0	0	10	4	2	0	4	0	2	0	0	4	0	0	6
11	16	0	0	0	2	2	2	2	2	0	2	0	0	2	0	2
12	16	0	6	0	0	0	2	0	2	0	2	2	4	2	4	0
13	16	2	4	0	6	2	4	2	0	0	2	0	0	2	0	0
14	16	2	0	4	0	2	0	0	0	0	0	2	6	2	2	4
15	16	0	0	6	4	2	0	4	2	0	2	2	0	0	0	2

BCT for σ_1

Δ_0/∇_0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16
1	16	0	6	0	4	0	6	0	0	2	0	0	2	0	2	2
2	16	0	4	2	6	0	6	2	0	0	0	2	0	2	0	0
3	16	2	6	0	6	2	4	0	2	0	2	0	0	0	0	0
4	16	0	0	0	0	10	2	0	4	0	4	0	0	6	2	4
5	16	10	0	0	0	0	2	0	4	4	4	6	2	0	0	0
6	16	0	2	2	0	0	0	0	0	0	2	2	2	2	2	2
7	16	0	2	0	0	0	0	2	2	2	0	2	2	2	2	0
8	16	4	0	0	0	4	2	2	10	6	0	0	0	4	0	0
9	16	0	0	2	2	2	0	2	0	2	0	0	0	2	2	2
10	16	4	0	2	0	4	2	0	0	0	10	4	0	0	0	6
11	16	2	0	2	2	0	0	2	0	2	0	2	2	0	0	2
12	16	0	0	0	2	4	0	2	6	2	0	0	2	4	2	0
13	16	6	2	2	0	0	0	2	4	4	0	2	2	0	0	0
14	16	0	2	2	0	6	0	2	0	0	4	0	0	2	2	4
15	16	4	0	2	2	0	0	0	0	0	6	4	2	0	2	2

BCT Code

```python
sbox = sbox_0
inv_sbox = inv_sbox_0

bct = [[0 for i in range(16)] for i in range(16)]



def get_val_bct(x,d0,d1):
    return inv_sbox[sbox[x] ^ d0] ^ inv_sbox[sbox[x ^ d1] ^ d0]



```

```python
def do_assignment(d1,d0):
    bct[d1][d0]+=1




[[[ do_assignment(d1,d0)  for x in range(16) if get_val_bct(x,d0,d1) == d1]
    for d0 in range(16)] for d1 in range(16)]

for i in bct:
    print(i)
```

Listing 10: BCT Python Code

8.2 Boomerang Difference Table

The table would be too large to print over here but, you can generate it in the Codebook.

```python

bdt = {}

for j in range(16):
    bdt[j] = [[0 for i in range(16)] for i in range(16)]
def get_c_d0(x,d0,invd0):
    return inv_sbox_0[sbox_0[x] ^ invd0] ^ inv_sbox_0[sbox_0[x ^ d0] ^ invd0
    ]

def get_c_d1(x,d0):
    return  sbox_0[x] ^ sbox_0[x ^ d0]

def do_assignment(d1,d0,invd0):
    bdt[d1][d0][invd0] += 1


[[[[do_assignment(d1,d0,invd0) for x in range(16) if get_c_d0(x,d0,invd0) ==
     d0 and get_c_d1(x,d0) == d1] for invd0 in range(16)] for d0 in range
    (16)] for d1 in range(16)]



for i in bdt:
    print("*"*25, f" d1 = {i}", "*"*25)
    for i in bdt[i]:
        print(i)
    print("\n")
```

Listing 11: BDT Python Code

Other properties like Component Functions, Maximum DDT Entry etc. can be generated via sage file *sbox_prop_1.sage*

9 Biclique Attack on PRESENT-80

In this section we will talk discuss the Bi-clique Attack on Present -80. The flow of discussion on this paper will go in the following manner.

1. Discussion on Meet-in-the-Middle Attack.
2. Drawbacks of MITM and Motivation for a Biclique attack.
3. Sieve-in-the-middle structure introduction.
4. Biclique Attack on PRESENT-80

9.1 Meet in the middle attack

In this attack we divide the Cipher into 2 two parts. Let E_K be encryption under key K.It is ideal for this attack that Cipher be able to break in 2 parts E_K1,E_K2.

$$E_K = E_{K1} \circ E_{K2} \tag{6}$$

Refer to Figure14 more clarity.

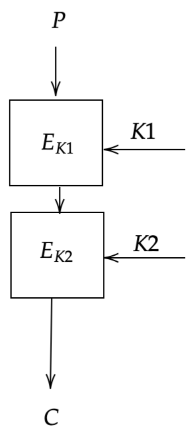


Figure 14: Structure of MITM

9.1.1 Attack Procedure

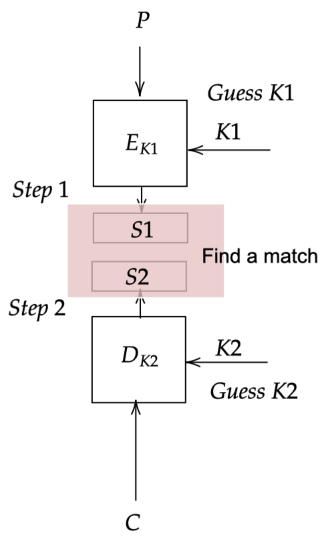


Figure 15: MITM Attack Procedure

The attack is orchestrated in the following manner:

1. Guess $2^{k/2}$ values of K_1. Calculate $E_{K1}(P)$ for each guess and store in $S1$.

2. Guess $2^{k/2}$ values of K_2. Calculate $D_{K2}(P)$ for each guess and store in $S2$.

3. Find a match in $S1$ and $S2$ return the corresponding $K = K_1 \cup K_2$.

Done.
Step 3 is also referred to as **Matching** step. Note that the data complexity of this attack is a single pair of Plaintext-Ciphertext. !

9.2 Drawbacks of MITM and motivation for Biclique Attack

Drawbacks MITM depends on the Key bits being independent which is decreasingly the case as the Cipher grows larger.
Motivation for Biclique: This attack does not assume any *Key dependency.* Sometime Biclique is used in conjunction to MITM, since MITM cannot handle Larger Rounds very well.

9.3 Biclique Structure

Biclique structure is most intuitive to understand using a picture. Refer to Figure 16

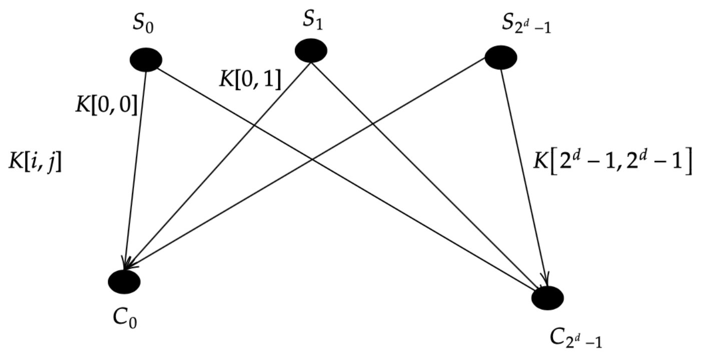


Figure 16: d-dimensional Biclique

9.4 Sieve-in-the-middle structure

The motivation behind a Sieve structure is to cut down the number of Matches that need to be compared. Now **u** and **v**(refer Figure 17) do not correspond to bits in some intermediate state. Now they are the input outputs of a predefined function **S**.There input outputs can be predetermined(or calculated on the fly).

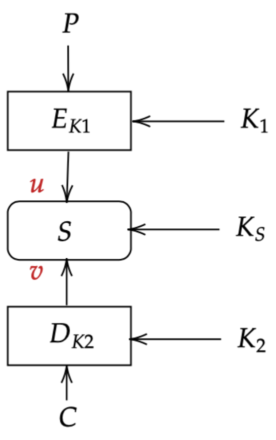


Figure 17: Sieve Structure

9.5 Sieve used for Attack

Recall, the properties of PRESENT-80 SBox:

1. The inputs of an SBox go to 4 distinct SBox's

2. The four output bits from a particular SBOX got 4-distinct SBox's of distinct groups.

These properties of the permutation layer help somewhat justify the Sieve that they have chosen.(Figure 18). The sieve can be seen as a **Super Box** who's inputs outputs are known to us from the beginning. So values of $K_{[i,j]}$ that do not conform to this sieve for respective values of S_j and C_i will be automatically filtered out.

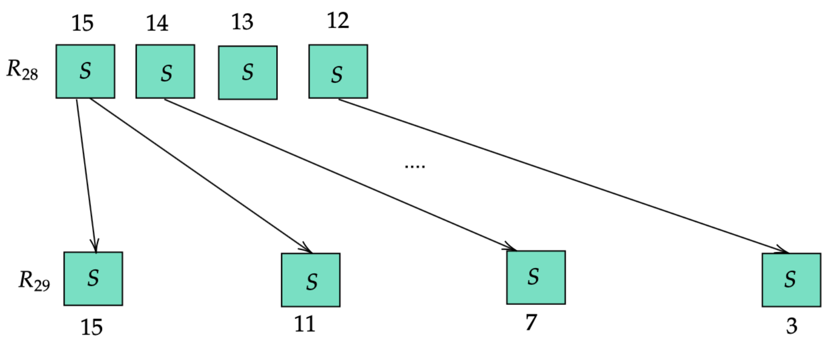


Figure 18: Sieve Used in Round 28

9.6 Attack structure

Combining all the pieces together we get the net structure of the attack.(See Figure 19)

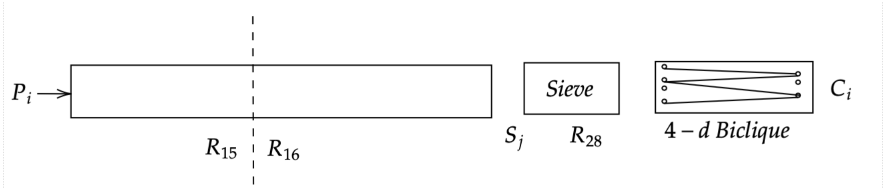


Figure 19: Structure of Attack

Matching takes place at the boundary of round 15 and 16.

9.7 Attack procedure

Note: The Key state of 80 bits is divided into 2^72 sets of 2^8 Keys.Also, $\Delta K[i,j] = \Delta K[i] \oplus \Delta K[j]$

Constructing Biclique

1. Select a series of 8 bits(in this case, [44, 43, 42, 41, 11, 10, 9, 8]). Assume these bits to be 0. Pick a random values for remaining 72 bits, this will be your $K[0,0]$.

2. Assume $C_0 = 0_{(64)}$ and decrypt it till round 28 using $K[0,0]$ and obtain S_0

3. Decrypt C_0 with keys $K[0,0] \oplus \Delta K[0,j] \, \forall 0 \geq j \leq 15$ to obtain S_j.(Refer to Figure 20

4. Encrypt S_0 with $K[0,0] \oplus \Delta K[i,0] \, \forall 0 \geq i \leq 15$ to obtain C_i and get respective P_i.

Matching

- Compute the value of 12 bits at ([63,62,61,59,58,57,55,54,53,51,50,49]) of input round 15 by encrypting P_i's and decrypting all S_j's

- Find values at 16 bit positions ([63,62,61,47,46,45,37, 36,35,15,14,13]). using $\Delta K[i,j]$, $\forall 0 \geq i \leq 15$.

Note, here you will see the benefit of using a sieve rather than only using a Biclique. The Sieve helped filter out values of S_j's that will result in an impossible trail.Another interesting thing to note is that we only calculated values of 12 bits on one side and not 16 to match the two sets.

9.7.1 Analysis

We have a total of 2^4 output values($0 \geq i \leq 15$). The possibility that Key matches all 12 bits is $2^4 * 2^{-12} = 2^{-8}$.
So for a set of 2^8 Keys we will find the correct key on average.

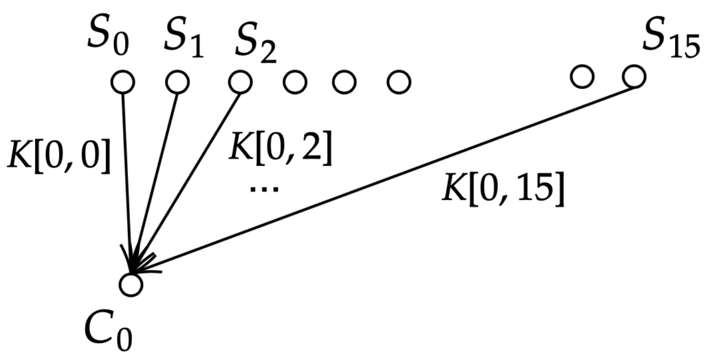


Figure 20: Corresponding to Point 2 in 9.7

10 Appendix

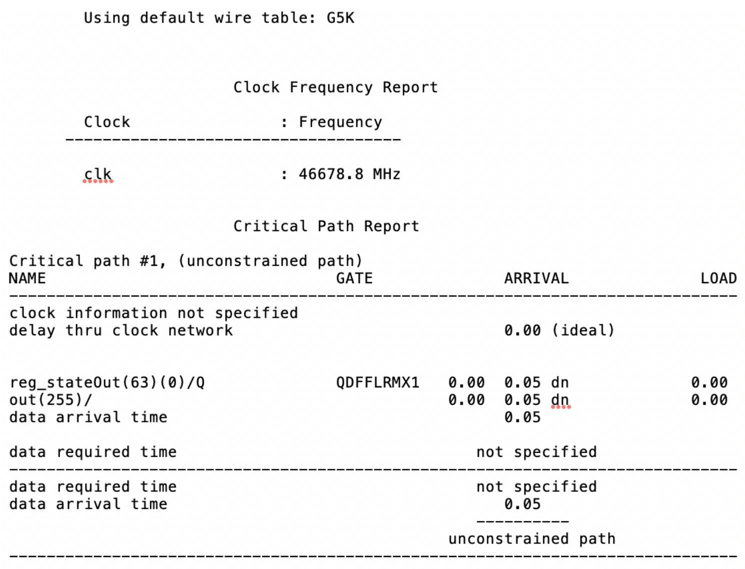
10.1 Verilog Code

```


module dff(clk, in, out);
  input       clk;
  input[0:255] in;
  output [0:255] out;

  wire [3:0] stateIn[0:63];
  reg [3:0] stateOut[0:63];

  genvar j;

  generate
    for(j=0; j<64;j=j+1)
    begin: assignment1
      assign stateIn[j] = in[j*4:(j+1)*4-1];
      assign out[j*4:(j+1)*4-1] = stateOut[j];
    end
  endgenerate

  integer x;
  integer y;
  integer z;

  integer x_dash;
  integer y_dash;
  integer z_dash;
  integer i;
  integer new_index;

  always @(posedge clk)
  begin
    for(i=0;i<64;i=i+1)
    begin
      z = i/16;
      x = (i/4)%4;
      y = i%4;

      x_dash = x;
      y_dash = y;
      z_dash = (y+z)%4;

      new_index = z_dash*16 + x_dash*4 + y_dash;
      stateOut[new_index] = stateIn[i];
    end
  end

endmodule
```

Listing 12: Verilog Code for Sheet Permutation

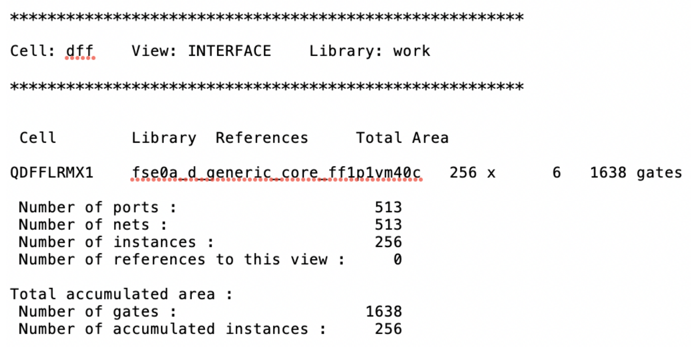
Slice permutation is almost exactly the same operation. Hence I omitted the code out for that one.

10.2 Leonardo Output Files



```
        Using default wire table: G5K


                        Clock Frequency Report

        Clock                : Frequency
      ------------------------------------

        clk                  : 46678.8 MHz

                        Critical Path Report

Critical path #1, (unconstrained path)
NAME                                GATE              ARRIVAL              LOAD
-------------------------------------------------------------------------------
clock information not specified
delay thru clock network                              0.00 (ideal)


reg_stateOut(63)(0)/Q               QDFFLRMX1   0.00  0.05 dn              0.00
out(255)/                                       0.00  0.05 dn              0.00
data arrival time                                     0.05

data required time                                 not specified
-------------------------------------------------------------------------------
data required time                                 not specified
data arrival time                                     0.05
                                                   ----------
                                                unconstrained path
-------------------------------------------------------------------------------
```

Figure 21: Frequency Information from Leonardo Tool



```
*******************************************************

Cell: dff    View: INTERFACE    Library: work

*******************************************************

 Cell        Library  References     Total Area

QDFFLRMX1    fse0a_d_generic_core_ff1p1vm40c   256 x      6   1638 gates

 Number of ports :                      513
 Number of nets :                       513
 Number of instances :                  256
 Number of references to this view :      0

Total accumulated area :
 Number of gates :                     1638
 Number of accumulated instances :      256
```

Figure 22: Gate Count Information from Leonardo

References

[1] Biclique cryptanalysis of MIBS-80 and PRESENT-80. Mohammad Hossein Faghihi Sereshgi, Mohammad Dakhilalian, and Mohsen Shakiba

[2] Saturnin: a suite of lightweight symmetric algorithms for post-quantum security Anne Canteaut, S ebastien Duval, Ga etan Leurent, Mar ıa Naya-Plasencia1, L eo Perrin, Thomas Pornin and Andr e Schrottenloher.