Date : 18/04/2022

CSPC63: **Principles of Cryptography**

# Assignment - 2

Roll no. : **106119100**

Name : **Rajneesh Pandey**

Section : **CSE-B**

# Write a program to implement differential cryptanalysis of a Substitution- Permutation network.

## Explanation:

Differential Cryptanalyst is similar to linear cryptanalysis in many respects. The main difference from linear cryptanalysis is that differential cryptanalysis involves comparing the **x-or** of two inputs to the **x-or** of the corresponding two outputs. In general, we will be looking at inputs **x** and **x∗** (which are assumed to be binary strings) having a specified (fixed) **x-or** value denoted by **x= x ⊕ x∗**.

Prime markings () to indicate the **x-or** of two bitstrings.

Differential cryptanalysis is a chosen-plaintext attack. We assume that an attacker has many tuples (x,x∗,y,y∗), where the x-or value x= x ⊕ x∗ is fixed. The plaintext elements (i.e., x and x∗) are encrypted using the same unknown key, K, yielding the ciphertexts y and y∗, respectively.

For each of these tuples, we will begin to decrypt the ciphertexts y and y∗, using all possible candidate keys for the last round of the cipher. For each candidate key, we compute the values of certain state bits, and determine if their x-or has a certain value (namely, the most likely value for the given input x-or). Whenever it does, we increment a counter corresponding to the candidate key. At the end of this process, we hope that the candidate key that has the highest frequency count contains the correct values for these key bits.

**Definition** Let $\pi_S : \{0,1\}^m \to \{0,1\}^n$ be an S-box. Consider an (ordered) pair of bitstrings of length $m$, say $(x, x^*)$. We say that the ***input x-or*** of the S-box is $x \oplus x^*$ and the ***output x-or*** is $\pi_S(x) \oplus \pi_S(x^*)$. Note that the output x-or is a bitstring of length $n$.

For any $x' \in \{0,1\}^m$, define the set $\Delta(x')$ to consist of all the ordered pairs $(x, x^*)$ having input x-or equal to $x'$.

It is easy to see that any set $\Delta(x')$ contains $2^m$ pairs, and that

$$\Delta(x') = \{(x, x \oplus x') : x \in \{0,1\}^m\}.$$

For each pair in $\Delta(x')$, we can compute the output x-or of the S-box. Then we can tabulate the resulting distribution of output x-ors. There are $2^m$ output x-ors, which are distributed among $2^n$ possible values. A non-uniform output distribution will be the basis for a successful differential attack.

input x-or. It will be convenient to have some notation to describe the distributions of the output x-ors, so we state the following definition.

**Definition** For a bitstring $x'$ of length $m$ and a bitstring $y'$ of length $n$, define

$$N_D(x', y') = |\{(x, x^*) \in \Delta(x') : \pi_S(x) \oplus \pi_S(x^*) = y'\}|.$$

In other words, $N_D(x', y')$ counts the number of pairs with input x-or equal to $x'$ that also have output x-or equal to $y'$ (for a given S-box)

input to the $i$th S-box in round $r$ of the SPN is denoted $u'^r_{<i>}$, and

$$u'^r_{<i>} = w'^{r-1}_{<i>} \oplus K^r_{<i>}.$$

An input x-or is computed as

$$\begin{aligned} u^r_{<i>} \oplus (u^r_{<i>})^* &= (w^{r-1}_{<i>} \oplus K^r_{<i>}) \oplus ((w^{r-1}_{<i>})^* \oplus K^r_{<i>}) \\ &= w^{r-1}_{<i>} \oplus (w^{r-1}_{<i>})^* \end{aligned}$$

Therefore, this input x-or does not depend on the subkey bits used in round $r$; it is equal to the (permuted) output x-or of round $r-1$. (However, the output x-or of round $r$ certainly does depend on the subkey bits in round $r$.)

Let $a'$ denote an input x-or and let $b'$ denote an output x-or. The pair $(a', b')$ is called a ***differential***. Each entry in the difference distribution table gives rise to an ***x-or propagation ratio*** (or more simply, a ***propagation ratio***) for the corresponding differential.

**Definition** The propagation ratio $R_p(a', b')$ for the differential $(a', b')$ is defined as follows:

$$R_p(a', b') = \frac{N_D(a', b')}{2^m}.$$

$R_p(a', b')$ can be interpreted as a conditional probability:

$$\mathbf{Pr}[\text{output x-or } = b' | \text{ input x-or } = a'] = R_p(a', b').$$

Input x-or of a differential in any round is the same as the (permuted) output x-ors of the differentials in the previous round. Then these differentials can be combined to form a differential trail.

Make the assumption that the various propagation ratios in a differential trail are independent (an assumption that may not be mathematically valid, in fact). This assumption allows us to multiply the propagation ratios of the differentials in order to obtain the propagation ratio of the differential trail.

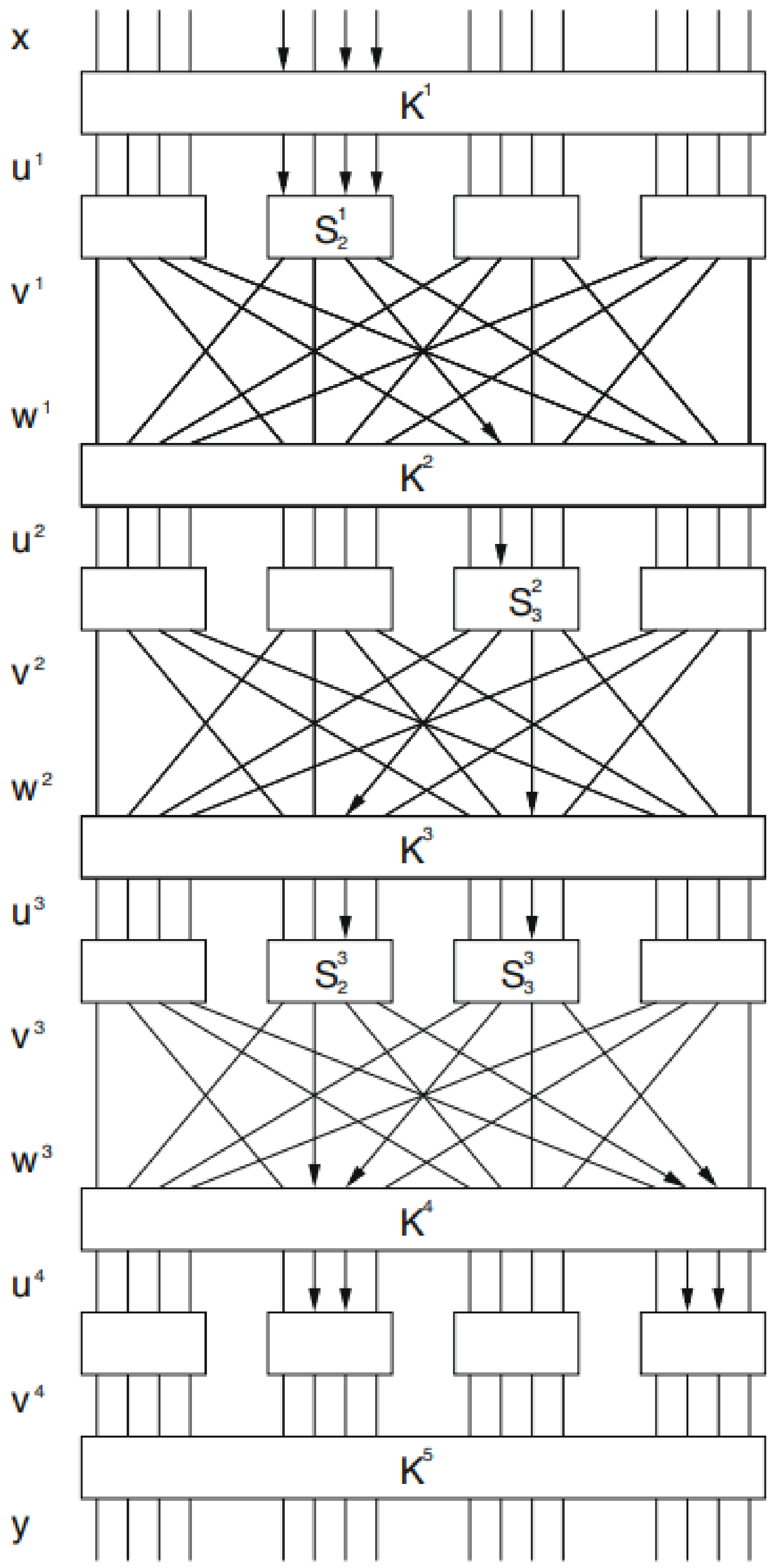


**FIGURE** A differential trail for a substitution-permutation network

Algorithm makes use of a certain filtering operation.

Tuples (x, x ∗ , y, y ∗ ) for which the differential holds are often called **right pairs**, and it is the right pairs that allow us to determine the relevant key bits. (Tuples that are not right pairs basically constitute "random noise" that provides no useful information.)

A right pair has

$$(u^4_{<1>})' = (u^4_{<3>})' = 0000.$$

Hence, it follows that a right pair must have y = (y) ∗ and y = (y) ∗ . If a tuple (x, x ∗ , y, y ∗ ) does not satisfy these conditions, then we know that it is not a right pair, and we can discard it. This filtering process increases the efficiency of the attack.

## Working Of the Algorithm

For each tuple (x, x ∗ , y, y ∗ ) ∈ T , we first perform the filtering operation. If (x, x ∗ , y, y ∗ ) is a right pair, then we test each possible candidate subkey (L1, L2) and increment an appropriate counter if a certain x-or is observed. The steps include computing an exclusive-or with candidate subkeys and applying the inverse S-box , followed by computation of the relevant x-or value.

A differential attack based on a differential trail having propagation ratio equal to ε will often be successful if the number of tuples (x, x ∗ , y, y ∗ ), which we denote by T, is approximately $cε^{-1}$ , for a "small" constant c.

After implementing the attack, we found that the attack was often successful if we took T between 50 and 100.

## Code:

```
"""
106119100 Rajneesh Pandey


Differential Cryptanalysis of SPN
"""

################################# BEGIN ###########################################

# The implemention of the SBox function: n in an integer between 0 and 15

def SBox(n):
    val = hex(n)
    if val == '0x0':
        return int(0xe)
    elif val == '0x1':
        return int(0x4)
    elif val == '0x2':
        return int(0xd)
    elif val == '0x3':
        return int(0x1)
    elif val == '0x4':
        return int(0x2)
    elif val == '0x5':
        return int(0xf)
    elif val == '0x6':
        return int(0xb)
    elif val == '0x7':
        return int(0x8)
    elif val == '0x8':
        return int(0x3)
    elif val == '0x9':
        return int(0xa)
    elif val == '0xa':
        return int(0x6)
    elif val == '0xb':
        return int(0xc)
    elif val == '0xc':
        return int(0x5)
    elif val == '0xd':
        return int(0x9)
    elif val == '0xe':
        return int(0x0)
    elif val == '0xf':
        return int(0x7)
```

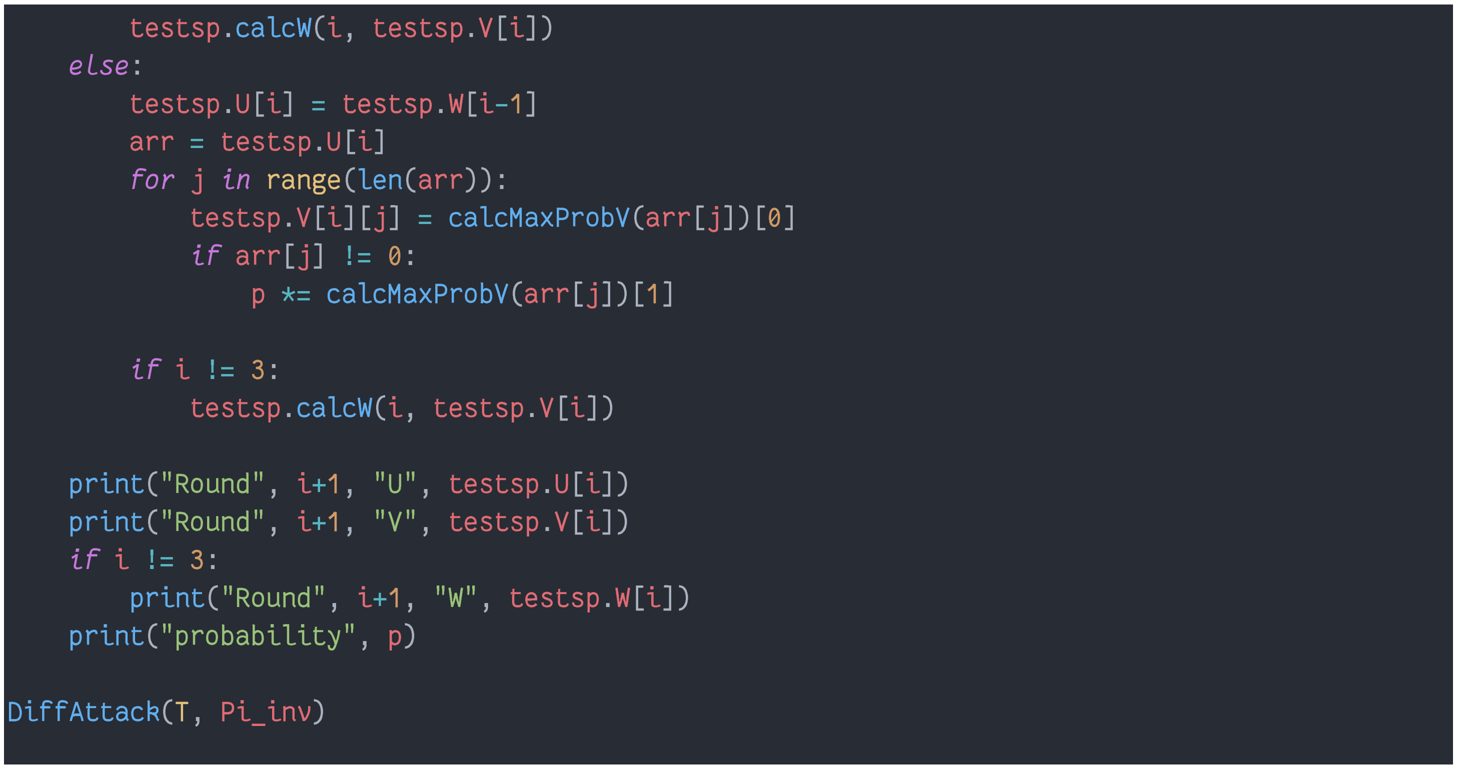
```
def PBox(n): #Implementation of the PBox function. Again, n in an integer between 0 and 15

    val = n + 1
    retval = 0
    if val == 1:
        retval = 1
    elif val == 2:
        retval = 5
    elif val == 3:
        retval = val+6
    elif val == 4:
        retval = (val+9)
    elif val == 5:
        retval = val-3
    elif val == 6:
        retval = val
    elif val == 7:
        retval = val+3
    elif val == 8:
        retval = (val + 6)
    elif val == 9:
        retval = (val-6)
    elif val == 10:
        retval = (val-3)
    elif val == 11:
        retval = val
    elif val == 12:
        retval = (val+3)
    elif val == 13:
        retval = (val-9)
    elif val == 14:
        retval = (val-6)
    elif val == 15:
        retval = (val-3)
    elif val == 16:
        retval = val

    return retval-1

def convertBin(n): #takes an integer between 0 and 15 as input and converts it to an array of
zeroes and ones, representing the binary notation of that number
    p = format(n, '#06b')
    p = p[2:]
    p = [int (x) for x in p]
    return p
def BackToInt(l):#takes an array of size 4 of zeroes and ones, and converts it to integer
```

```
    r = 0
    for item in l:
        r = (r<<1)|item
    return r

def DeltaX(xp):#takes an integer x' in [0, 15] as input, and calculates all possible x*'s
corresponding to x's in [0, 15]
    delX = []
    for i in range(16):
        delX.append(i ^ xp)
    return delX


"""
Substitution- Permutation network
"""
class SPN:

    K = [[], [], [], [], []]#5 round keys for an SPN of 4 rounds
    Key = [3, 10, 9, 4, 13, 6, 3, 15]#the 32-bit key of the SPN
    U = [[], [], [], []]
    V = [[], [], [], []]
    W = [[], [], []]
    Y = []#SPN output
    _X = []#SPN input

#The following lists have been initialised just for the ease of calculation, and are not
relevent since they do not correspond to any significant SPN property
    u = []
    v = []
    w = []
    v_w = []

    def calcU(self, r, w_r_):#calculates u_r for the round r, given the value of w_{r-1}
        u = []
        for j in range(len(w_r_)):
            u.append(w_r_[j] ^ self.K[r][j])
        self.U[r] = u

    def calcV(self, r, u_r):#calculates v_r for the round r, given the value of u_r
        v = []
        for j in range(len(u_r)):
            v.append(SBox(u_r[j]))
        self.V[r] = v

    def calcW(self, r, v_r):#calculates w_r for the round r, given the value of v_r
        w = []
        v_w = []
        for v in v_r:
        #We have represented all the 16-bit SPN parameters as an array of 4 integers in [0,
15].
        #But to calculate w, we need all the 16 bits. Hence, here the required conversion is
done

            v_w.append(convertBin(v))

        w_row = []
        for j in range(4):
            k = j*4
            for m in range (k, k+4):
                w_row.append(v_w[int(PBox(m)/4)][PBox(m)%4])
            w.append(w_row)
            w_row = []
        for j in range(4):
            w[j] = BackToInt(w[j])

        self.W[r] = w

    def calcY(self):#calculates the output Y of the SPN
        y = []
        l = len(self.V)
        k = len(self.K)
        for j in range(len(self.V[l-1])):
            y.append(self.V[l-1][j] ^ self.K[4][j])
        self.Y = y


    def __init__(self, X):
        """
        #Initialises the SPN using the given input X.
        We expect a list* conataining a single list of size 4

        * : (yes, a list, with a single element in it.... I had something else in my mind when
creating this architecture,
            but ended up doing something completely different)
        """
        for i in range(5):#Generates 5 round keys for the SPN of 4 rounds
            rkey = []
            for j in range(i, i+4):
                rkey.append(self.Key[j])
            self.K[i] = rkey

        for x in X:#This loop runs only for one iteration since X contains just one element
            self._X = x
            if len(x) != len(self.K[1]):
                print("invalid input")
                pass
            else:
                """
                    Since we are using arrays here, we are forced to used zero-indexing.
                    So, here U0 will denote the SBox inputs for the first round.
                    This also means that W0 here is the actual W1.
                    Instead of creating an actual W0, we've directly calculated the actual
U1's using the input X
                """
                for i in range(0, 4):
                    if i == 0:
                        self.calcU(i, x)
                    else:
                        self.calcU(i, self.W[i-1])

                    self.calcV(i, self.U[i])

                    if i != 3:#Calculate W's except for the last round
                        self.calcW(i, self.V[i])
                    else:
                        self.calcY()


    def display(self):#function that displays all the SPN parameters, over all the 4 rounds
        K0_disp = []
        K1_disp = []
        K2_disp = []
        K3_disp = []
        K4_disp = []
        W_disp = []
        V_disp = []
        U_disp = []
        Y_disp = []
        X_disp = []

        for i in range(4):
            K0_disp.append(convertBin(self.K[0][i]))
            K1_disp.append(convertBin(self.K[1][i]))
            K2_disp.append(convertBin(self.K[2][i]))
            K3_disp.append(convertBin(self.K[3][i]))
            K4_disp.append(convertBin(self.K[4][i]))

        for w in self.W:
            w_d = []
            for i in range(len(w)):
```

```python
                w_d.append(convertBin(w[i]))
            W_disp.append(w_d)

        for u in self.U:
            u_d = []
            for i in range(len(u)):
                u_d.append(convertBin(u[i]))
            U_disp.append(u_d)

        for v in self.V:
            v_d = []
            for i in range(len(v)):
                v_d.append(convertBin(v[i]))
            V_disp.append(v_d)

        y_d = []
        for y in self.Y:
            y_d.append(convertBin(y))
        Y_disp.append(y_d)

        x_d = []
        for x in self._X:
            x_d.append(convertBin(x))
        X_disp.append(x_d)

        for w in W_disp:
            print("W is", w)

        for w in U_disp:
            print("U is", w)

        for w in V_disp:
            print("V is", w)

        for w in Y_disp:
            print("Y is", w)

        print("K[0] is ", K0_disp)
        print("K[1] is ", K1_disp)
        print("K[2] is ", K2_disp)
        print("K[3] is ", K3_disp)
        print("K[4] is ", K4_disp)

################################## The Differential Attack ################################

def DiffAttack(T, Pi_inv):
    """
```

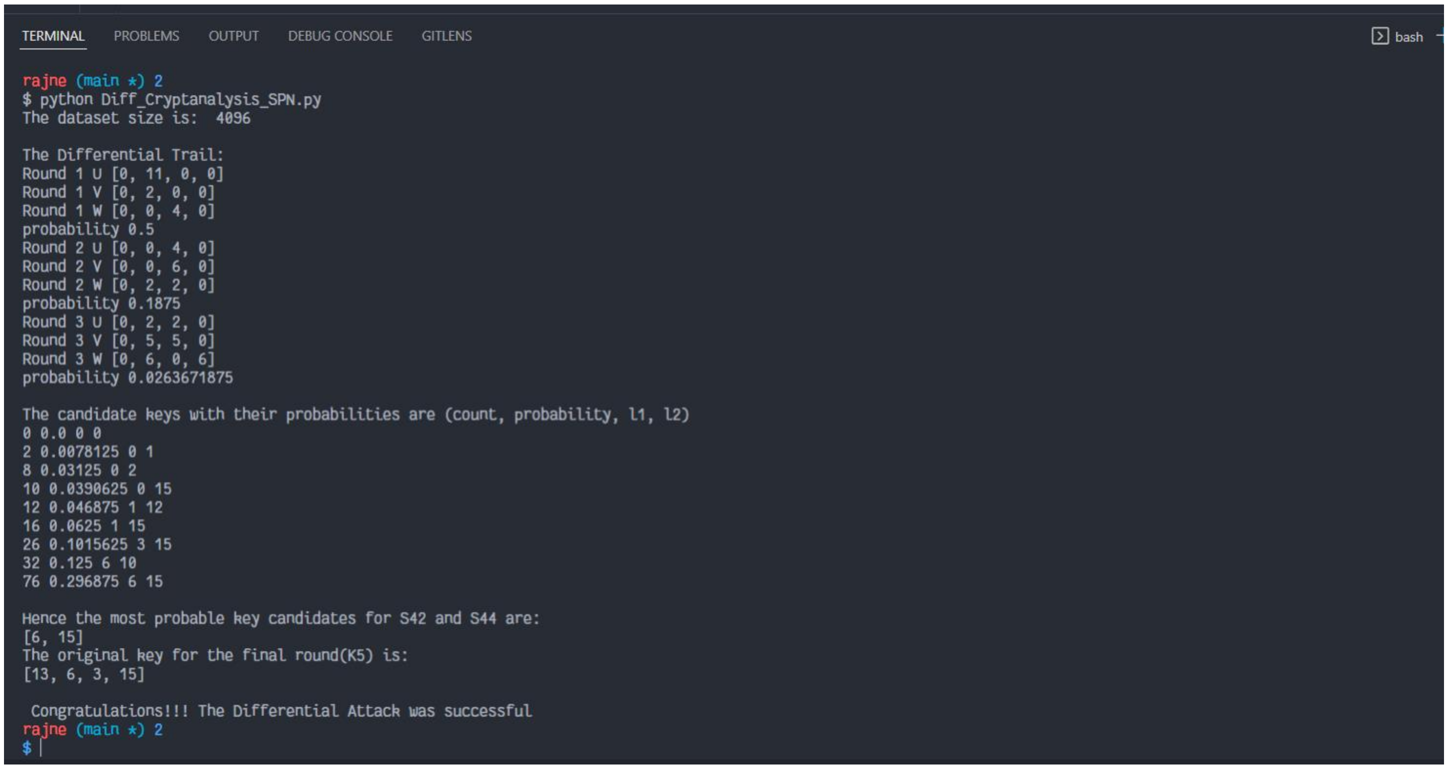
```
        In this function, the tuple T consists of two objects of the SPN class, one 
corresponding to the input x and other corresponding to the input x*
        We assume no access to any parameter of the SPN, except the input x and the output y.
    """
    count = []

    for l1 in range(0, 16):
        """
            We want to find the 8-bit candidate key for a given differential trail. This 8-bit 
candidate forms half of the key in round 5
        """
        countRow = []
        for l2 in range(0, 16):
            countRow.append(0)
        count.append(countRow)

    for [sp, sp_] in T:
        x = sp._X
        x_ = sp_._X
        y = sp.Y
        y_ = sp_.Y
        u42 = 0
        u42_ = 0
        u44 = 0
        u44_ = 0
        if y[0] == y_[0] and y[2] == y_[2]:
            for l1 in range(0, 16):
                for l2 in range(0, 16):
                    v42 = l1 ^ y[1]
                    v44 = l2 ^ y[3]
                    u42 = Pi_inv[v42]
                    u44 = Pi_inv[v44]
                    v42_ = l1 ^ y_[1]
                    v44_ = l2 ^ y_[3]
                    u42_ = Pi_inv[v42_]
                    u44_ = Pi_inv[v44_]
                    u42Prime = u42 ^ u42_
                    u44Prime = u44 ^ u44_
                    if u42Prime == 0b0110 and u44Prime == 0b0110:
                        count[l1][l2] = count[l1][l2] + 1

    _max = -1

    print("\nThe candidate keys with their probabilities are (count, probability, l1, l2)")
    for l1 in range(0, 16):
        for l2 in range(0, 16):
            if count[l1][l2] > _max:
                _max = count[l1][l2]
                print(_max, float(_max/256.0), l1, l2)
                maxkey = [l1, l2]

    maxkey_disp = [bin(maxkey[0]), bin(maxkey[1])]
    print("\nHence the most probable key candidates for S42 and S44 are:")
    print (maxkey)

    print("The original key for the final round(K5) is:")
    print(sp.K[len(sp.K) - 1])
    print("\n Congratulations!!! The Differential Attack was successful")

##################### FINAL Algorithm ################################
X_prime = []

for i in range(0, 16):
    X_prime.append(i)

DeltaX_ = []#A 2d array, such that DeltaX_[x'][x] = x*
X =  []
for i in range(0, 16):
    DeltaX_.append(DeltaX(i))

for i in range(0, 16):
    X.append(i)

XStar = []

for x in X:
    x_starRow = []
    for xp in X_prime:
        x_starRow.append(DeltaX_[x][xp])
    XStar.append(x_starRow)

Y = []
YStar = []

for x_ in X:
    Y.append(SBox(x_))


for x_ in XStar:
    yrow = []
    for j in range(16):
        yrow.append(SBox(x_[j]))
    YStar.append(yrow)
```

```
Y_Prime = []

for y in YStar:
    yrow = []
    for j in range(16):
        yrow.append(Y[j] ^ y[j])
    Y_Prime.append(yrow)

b_prime = []

for y in Y_Prime:
    brow = []
    for i in range(16):
        c = y.count(i)
        brow.append(c)
    b_prime.append(brow)


Nd = []
for b in b_prime:
    ndrow = []
    for j in range(16):
        ndrow.append(float(b[j]))
    Nd.append(ndrow)

def Rp(i, j):#Function that calculates the propogation ratio
    return(Nd[i][j]/16)

Pi_inv = {0:0}#Dictionary that stores the inverse SBox values

for i in range(0, 16):
    Pi_inv[SBox(i)] = i


INP = []###################### x
for i in range(16):
    for j in range(16):
        for k in range(16):
            INP.append([0, i, j, k])

sp = []######################## array of SPN's for all possible values x

for i in range(len(INP)):
    sp.append(SPN([INP[i]]))

INP1 = []####################### x*
for i in range(16):
    for j in range(16):
        for k in range(16):
            INP1.append([0, DeltaX_[11][i], DeltaX_[0][j], DeltaX_[0][k]])

sp1 = []########################array of SPN's for all possible values of x*
for i in range(len(INP1)):
    sp1.append(SPN([INP1[i]]))

OP = []######################## y
for i in range(len(INP)):
    OP.append(sp[i].Y)


OP1 = []######################## y*
for i in range(len(INP1)):
    OP.append(sp1[i].Y)

T =[] # Tuples to be used in the Differential Attack
for i in range(len(sp)):
    T.append([sp[i], sp1[i]])

print("The dataset size is: ",  len(INP)) #Size of the input dataset

def calcMaxProbV(x):#Returns the value that is most likely to occur looking at the propogation
ratios for a given value of x
    prob = []
    retval = 0
    for i in range(16):
        prob.append(Rp(x, i))

    retval = prob.index(max(prob))
    return [retval, max(prob)]

testsp = SPN([[0, 11, 0, 0]])#Constant x' is 0000 1011 0000 0000

print("\nThe Differential Trail: ")
p = 0
for i in range(3):
    if i == 0:
        testsp.U[i] = testsp._X
        arr = testsp.U[i]
        for j in range(len(arr)):
            testsp.V[i][j] = calcMaxProbV(arr[j])[0]
            if arr[j] != 0:
                p = calcMaxProbV(arr[j])[1]
            # print(p)
```



```python
        testsp.calcW(i, testsp.V[i])
    else:
        testsp.U[i] = testsp.W[i-1]
        arr = testsp.U[i]
        for j in range(len(arr)):
            testsp.V[i][j] = calcMaxProbV(arr[j])[0]
            if arr[j] != 0:
                p *= calcMaxProbV(arr[j])[1]

        if i != 3:
            testsp.calcW(i, testsp.V[i])

    print("Round", i+1, "U", testsp.U[i])
    print("Round", i+1, "V", testsp.V[i])
    if i != 3:
        print("Round", i+1, "W", testsp.W[i])
    print("probability", p)

DiffAttack(T, Pi_inv)
```

## Output :



```
rajne (main *) 2
$ python Diff_Cryptanalysis_SPN.py
The dataset size is:  4096

The Differential Trail:
Round 1 U [0, 11, 0, 0]
Round 1 V [0, 2, 0, 0]
Round 1 W [0, 0, 4, 0]
probability 0.5
Round 2 U [0, 0, 4, 0]
Round 2 V [0, 0, 6, 0]
Round 2 W [0, 2, 2, 0]
probability 0.1875
Round 3 U [0, 2, 2, 0]
Round 3 V [0, 5, 5, 0]
Round 3 W [0, 6, 0, 6]
probability 0.0263671875

The candidate keys with their probabilities are (count, probability, l1, l2)
0 0.0 0 0
2 0.0078125 0 1
8 0.03125 0 2
10 0.0390625 0 15
12 0.046875 1 12
16 0.0625 1 15
26 0.1015625 3 15
32 0.125 6 10
76 0.296875 6 15

Hence the most probable key candidates for S42 and S44 are:
[6, 15]
The original key for the final round(K5) is:
[13, 6, 3, 15]

 Congratulations!!! The Differential Attack was successful
rajne (main *) 2
$ |
```