

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Date : 18/04/2022 | | | | | | | |
|  | CSPC63: **Principles of Cryptography**  **Assignment - 2** | | | | | |  |
|  |  | | | | |  | |
|  | | |  |  | | | |
|  | | | Roll no. : 106119100Name : Rajneesh PandeySection : 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.

**Text

Description automatically generated**

**Graphical user interface, text, application, email

Description automatically generated**

Text

Description automatically generated with medium confidence

**Text

Description automatically generated**

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.

Diagram, engineering drawing

Description automatically generated

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

A picture containing text, watch, clock

Description automatically generated

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)):

                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)

        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 :**

Text

Description automatically generated