CS641 Mid Semester Exam

Modern Cryptology

Indian Institute Of Technology, Kanpur

Group Name :- ANV

Dibbu Amar Raja (21111009)

Idamakanti Venkata Nagarjun Reddy (21111031)

Vikas (21111067)

Q1. Consider a variant of DES algorithm in which all the S-boxes are replaced. The new S-boxes are all identical and defined as follows.

Let $b_1, b_2, \cdots, b_6$ represent the six input bits to an S-box. Its output is $b_1 \oplus (b_2 \cdot b_3 \cdot b_4), (b_3 \cdot b_4 \cdot b_5) \oplus b_6, b_1 \oplus (b_4 \cdot b_5 \cdot b_2), (b_5 \cdot b_2 \cdot b_3) \oplus b_6$.

Here '$\oplus$' is bitwise XOR operation, and '$\cdot$' is bitwise multiplication. Design an algorithm to break 16-round DES with new S-boxes as efficiently as possible.

Solution :-

We use **differential cryptanalysis** to break DES16 with a **14** round characteristic.
link to the code 1.

here(in code), we performed 16 round DES with the given S-box operations. We found such a Xor value of input text pairs which have the max probability at output of all s-boxes in each round,till "r-2" rounds (i.e. 14 rounds) and built a characteristic for our 16DES system.

An "r"-round DES requires "r-2" characteristic.

Probability of characteristic $\Rightarrow$ Product of all probabilites in a characteristic.
Number of plaintext pairs required with probability characteristic "p" is $\approx$20/p

So, DES16 requires a 14 ROUND CHARACTERISTIC.
we found a CHARACTERISTIC (in Hexadecimal format) $\Rightarrow$

['00000002', '00000000', '1', '00000000', '00000002', '0.5', '00000002', '00000000', '1', '00000000', '00000002', '0.5', '00000002', '00000000', '1', '00000000', '00000002', '0.5', '00000002', '00000000', '1', '00000000', '00000002', '0.5', '00000002', '00000000', '1', '00000000', '00000002', '0.5', '00000002', '00000000', '1', '00000000', '00000002', '0.5', '00000002', '00000000', '1', '00000000', '00000002','0.5','00000002','00000000']

$\bullet \underline{Probability of characteristic} we get \Rightarrow$ "0.0078125".$(i.e 1/2^7)$

$\bullet Number of plaintext pairs \approx 2^{11}$

Using this, We reached till 14 round and found "r-2" ,"l-2" XOR values.

Now we can break DES16 with **"Known plaintext attack"** and **"Chosen plaintext attack"**.

Now, $L_{15} = R_{14}$ and $R_{15} = R_{16}$,now we know all XOR values.

Solving last round KEY

Let $E(R_{15}) = \alpha_1\alpha_2\alpha_3...\alpha_8$ and $E(R^{'}_{15}) = \alpha^{'}_1\alpha^{'}_2\alpha^{'}_3...\alpha^{'}_8$ with $|\alpha_i| = 6 = |\alpha^{'}_i|$

Let $\beta_i = \alpha_i \oplus k_{15}$ and $\beta^{'}_i = \alpha_i \oplus k_{15}$ and $|\beta_i| = 6 = |\beta^{'}_i|$

Let $\gamma_i = S_i(\beta_i)$ and $\gamma^{'}_i = S_i(\beta^{'}_i)$ and $|\gamma_i| = 6 = |\gamma^{'}_i|$

We know $\alpha_i, \alpha^{'}_i$ and $\beta_i \oplus \beta^{'}_i = \alpha_i \oplus \alpha^{'}_i$

We also know a value γ such that $\gamma_i \oplus \gamma^{'}_i = \gamma$ with probability $1/2^7$

Define

$$\mathrm{X}_i = (\beta, \beta^{'})|\beta \oplus \beta^{'} = \beta_i \oplus \beta^{'}_i \text{ and } S_i(\beta) \oplus S_i(\beta^{'}) = \gamma$$

Pair $(\beta_i, \beta^{'}_i)\epsilon X_i$ whenever our guess for $\gamma \oplus \gamma^{'}_i = \gamma$ is correct , which happens with probability $1/2^7$.

Define

$$K_i = \left\{k|\alpha_i \oplus k = \beta \text{ and } (\beta, \beta^{'})\epsilon X_i \text{ for some } \beta^{'}\right\}$$

Since $(\beta_i, \beta^{'}_i)\epsilon X_i$ with probability , we have $k_{16,i}\epsilon K_i$ with probability $\geq 1/2^7$.

We have $|K_i| = |X_i|$ since α_i and $\beta \oplus \beta_i$ is fixed for $(\beta, \beta^{'})\epsilon X_i$ Instead We do as Follows, Let $k_1, k_2 ... k_{1,2^{11}}$ be set of possible subkeys, each containing $k_{16,i}$ with probability $\geq 1/2^7$.

On Careful analysis we get.

- If $\gamma \neq \gamma_i \oplus \gamma_i^{'}$ then $K_{16,i}$ becomes wrong value.
- Hence, $pr[k_{16,i} \epsilon K_{i,s} | \gamma \neq \gamma_i \oplus \gamma_i^{'}] = \frac{|K_{i,s}|}{64}$.
- Therefore the expected number of sets containing $k_{16,i}$ would be, $\geq \frac{1}{2^7} l + \sum_{1 \leq s \leq l} \frac{K_{i,s}}{64}$

In Comparison, expected number of sets containing $a \neq k_{16,i}$ would be,

$\sum_{1 \leq s \leq l} \frac{|K_{i,s}|}{64} + \sum_{1 \leq s \leq l} \frac{|K_{i,s}|}{64}$

Gap between the two numbers is minimum when all the $K_{i,s}$ have maximum possible size 16.
Then the number of $_{16,i} is$:

$\geq \frac{1}{128} l + \frac{127}{512} l = \frac{131}{512} l$

And the number of $a \neq k_{16,i}$ is : $\frac{1}{2^7} l$

Choosing $l \geq 20$ would give sufficient gap between the two expexted values.
Then $K_{16,i}$ can be identified as the most frequently occuring value in the sets $K_{1,i}, K_{2,i} .... K_{i,16}$

```
#AND operation
def AND(e1,e2,e3):
    if (e1=='0' or e2=='0' or e3=='0'):
        return '0'
    else:
        return '1'

#XOR opereation
def XOR(e1,e2):
    if e1==e2:
        return '0'
    else:
        return '1'

exp_d = [31,  0,  1,  2,  3,  4,  3,  4,  5,  6,  7,  8,  7,  8,  9, 10, 11,
      12, 11, 12, 13, 14, 15, 16, 15, 16, 17, 18, 19, 20, 19, 20, 21, 22,
      23, 24, 23, 24, 25, 26, 27, 28, 27, 28, 29, 30, 31,  0]


per = [15,  6, 19, 20, 28, 11, 27, 16,  0, 14, 22, 25,  4, 17, 30,
9,  1,
        7, 23, 13, 31, 26,  2,  8, 18, 12, 29,  5, 21, 10,  3, 24]

#EXPANSION OPERATION
def expansion(b):
    exp = ''
    for ind in exp_d:
        exp+=b[ind]
    return exp

#PERMUTATION OPERATION
def permutation(b):
    perm=''
    for ind in per:
```

```
        perm += b[ind]
    return perm

#S-BOX OPERATION
def s_box(b):
    output = ''
    output += (XOR (b[0] ,AND(b[1], b[2], b[3])))
    output += (XOR (b[5] ,AND(b[2], b[3], b[4])))
    output += (XOR (b[0] ,AND(b[3], b[4], b[1])))
    output += (XOR (b[5] ,AND(b[4], b[1], b[2])))

    return output

def xor_bitwise(e1,e2):            #gives xor value of i/p pairs
    op=''
    for i in range(len(e1)):
        op += XOR(e1[i],e2[i])
    return op

#all 2^7 comb are stored for 6bits
all_xors=[]
for i in range(64):
    all_xors.append("{0:06b}".format(i))



#for a specific xor value what COMBINATIONS OF i/p gives us
d={}
for i in all_xors:
    t = []

    for j in all_xors:
        ip_pairs = xor_bitwise(j,i)
        t.append((j,ip_pairs))
```

```
    d[i] = t

#for every o/p xor value we will have 64 combinations of plaintext pairs



probability={}

for key,val in d.items():
    xor_prob={}

    for j in val:
        s0 = s_box(j[0])
        s1 = s_box(j[1])

        #XOR OF OUTPUT 4 BITS
        s_xor = xor_bitwise(s0,s1)

        #CALCULATING FREQ
        if s_xor in xor_prob:
            xor_prob[s_xor]+=1
        else:
            xor_prob[s_xor]=1

    probability[key] = xor_prob




#possible intial values
l_xor=[]
for i in range(2**22):                    #2^22 as 2^32 values lead to out of m
    l_xor.append(f'{i:0{32}b}')
```

```
#expansion and s-box
def exp_sbox(r):
    op=''
    prob=1

    #expansion
    e=expansion(r)

    for i in range(0,len(e),6):
        p=probability[e[i:i+6]]

        max_op = ''
        max_count = 0

        for k,v in p.items():
            if v > max_count:
                max_count = v
                max_op = k

        op+=max_op

        prob*=(p[max_op]/64)
    return op,prob

#14-round characteristic
characteristic =[]

#circuit of all rounds
def des16(l,r,flag=0):
    tot_prob = 1

    for i in range(15):
        #expansion and s-box
        s_op , p = exp_sbox(r)
```

```
        if flag == 1:
            characteristic.append(l)
            characteristic.append(r)
            characteristic.append(p)
        if i==14:break
        tot_prob*=p

        #permutation
        p=permutation(s_op)

        temp=r

        #XOR operation
        r = xor_bitwise(p,l)

        l=temp

    #NUMBER OF PLAINTEXT PAIRS
    txts = 20/tot_prob

    #getting nearest 2^x
    c=1
    for x in range(100):
        c*=2
        if c>txts:
            return x                    #highest x is returned

    return 10**6

opt_power = 0
optimal_ip_xor = 0

val=10**9
```

```
for j in range(1,2**14):

    t = des16(l_xor[j],'00000000000000000000000000000000')        #for 14
    if t<val:
        print("Number of text pairs required")
        print('2 power',t)
        print('j is :',j)
        print('\n')

        opt_power = t
        optimal_ip_xor = j

        val=t

print('Optimal XOR value for L0 at initial stage is :',l_xor[optimal_ip_xor])

#DES16 with the optimal probability XOR value

l0 = l_xor[optimal_ip_xor]
r0 = '00000000000000000000000000000000'
des16(l0,r0,1)
#

print("14 ROUND CHARACTERISTIC is: ")
print(characteristic,end='')


#PROBABILITY OF CHARACTERISTIC
s=1
for i in range(2,len(characteristic),3):
    s *= characteristic[i]

print("PROBABILITY OF CHARACTERISTIC is : ",s)
# (1/2^7)
```

1/128

Q2. Suppose Anubha and Braj decide to do key-exchange using Diffie-Hellman scheme except for the choice of group used. Instead of using F_p^* as in Diffie-Hellman, they use S_n, the group of permutations of numbers in the range $[1, n]$. It is well-known that $|S| = n!$ and therefore, even for $n = 100$, the group has very large size. The key-exchange happens as follows:

> An element $g \in S_n$ is chosen such that g has large order, say l. Anubha randomly chooses a random number $c \in [1, l-1]$, and sends g^c to Braj. Braj choses another random number $d \in [1, l-1]$ and sends g^d to Anubha. Anubha computes $k = (g^d)^c$ and Braj computes $k = (g^c)^d$.

Show that an attacker Ela can compute the key k efficiently.

Solution :-

We will show that an attacker Ela can compute the key k efficiently. The attacker Ela knows g, g^c and g^d. Ela will first find 'd' and then calculate $(g^c)^d$. Size of the element 'g' is n. For example, if n is 5, then there will be 5 numbers in g.

The naive solution to find 'd' is the following:

- We need to maintain a counter variable 'd', a variable 'a' to store product. 'a' is initially equal to 'g'.
- In each iteration, we multiply a with g and increment the variable 'd' and check whether a equals to g^d. If a equals to g^d, then we have obtained the value 'd'.

The time complexity for this naive solution is n * d. This is very inefficient and not practically feasible to compute for large values of n and d. we need to develop an efficient solution to calculate d from g and g^d. We can think of g as a combination of some disjoint cycles. Order of an element g is the lcm of it's cycle lengths.

For example, if g is $\begin{matrix}(1 & 2 & 3 & 4 & 5) \\ (2 & 3 & 1 & 5 & 4)\end{matrix}$ then we can divide this into two disjoint cycles $\begin{matrix}(1 & 2 & 3) \\ (2 & 3 & 1)\end{matrix}$ and $\begin{matrix}(4 & 5) \\ (5 & 4)\end{matrix}$

First, we need to write a function pow(x,p) which calculates x raised to the power p. Here x is any element of the permutation group of size n. We need to develop a method to calculate this power efficiently.The naive method has a time complexity of O(n*p).

We can do better. After we divide x into some disjoint cycles, for each cycle, we need to find the element that is mapped to that particular position.

If x is $\begin{matrix}(1 & 2 & 3 & 4 & 5) \\ (2 & 3 & 1 & 5 & 4)\end{matrix}$ then two disjoint cycles $\begin{matrix}(1 & 2 & 3) \\ (2 & 3 & 1)\end{matrix}$ and $\begin{matrix}(4 & 5) \\ (5 & 4)\end{matrix}$

If x is multiplied by x, 2 3 1 will change to 3 1 2 and 5 4 will change to 4 5. The number of position shifts in the cycle can be calculated by performing the modulus of the power p with the cycle length.

For example, if the power p is 2, then the position shifts in the cycle is 2 % 3 which is 2. So, 2 3 1 will change to 1 2 3.If the power p is 6, then the position shifts in the cycle is 6%3 which is 0. So, 2 3 1 will remain as 2 3 1.

In this way, we can calculate pow(x,p) efficiently. Time complexity here is O(n) (O(n) for detection of all cycles and after the cycles are detected, O(n) for position shift calculation).

The next important part is calculation of d from g^d and g.The naive method has a time complexity of O(n*p).

To calculate d efficiently, we do the following:

1.) We first divide g and g^d into cycles.

2.) Then for each cycle in g and g^d, we calculate the position difference between them. For example if a cycle of g is (2 3 1) and cycle of g^d is (1 2 3), then the difference 'b' is 2.

3.) The cycle length is denoted by 'cl'. The cycle length of (2 3 1) cycle is cl=3.

Then d can be written as d = cl*z + b. Here, d is the power. cl is cycle length. z is some arbitrary integer. b is position difference.
If there are some q cycles in g, d can be written as:

d = $\mathrm{cl}_1 * z_1 + b_1$

$d = cl_2 * z_2 + b_2$

$d = cl_3 * z_3 + b_3$

.

.

.

d = $\mathrm{cl}_q * z_q + b_q$

To be less formal, we need to find an integer d such that d when divided by cl_1 gives a remainder of b_1, d when divided by cl_2 gives a remainder of b_2 .... d when divided by cl_q gives a remainder of b_q. For finding this d, we will use Chinese Remainder Theorem for non coprime moduli.

The time complexity for finding all cl values and b values is O(n).The time complexity for chinese remainder theorem for non co prime moduli is O(n*log(L)) where L is the LCM of all the cycle lengths i.e LCM of $(cl_1, cl_2.......cl_q)$. So, the total time complexity for finding the value of d is $O(n * log(L))$.

After calculating the value of d, we then calculate $(g^c)^d$ using the our power function pow i.e:- $pow(g^c, d) = Key$.
In this way, Ela can calculate the key value efficiently.

The code for the calculation is given below:-

```
import numpy as np
from math import inf,lcm ,gcd
from random import randint
#order function is used to calculate the
#order of the element 'g' belonging
#to the permutation group S_n.
def order(g):
    o=g.copy()
    l=len(g)
    v=[0 for i in range(l)]
    ans=1
    for i in range(l):
        if v[i]==0:
            v[i]=1
            c=1
            j=g[i]
            while v[j]==0:
                c+=1
                v[j]=1
                j=g[j]
            ans=lcm(ans,c)
    return ans
```

```
#pow function is used for calculating the
#'p' th power of an element 'g' belonging
#to the permutation group S_n.
def pow(g,p):
    q=[]
    ind={}
    l=len(g)
    v=[0 for i in range(l)]
    for i in range(l):
        if v[i]==0:
            v[i]=1
            q.append([g[i]])
            ind[g[i]]=[len(q)-1,0]
            j=g[i]
            while v[j]==0:
                ind[g[j]]=[len(q)-1,len(q[-1])]
                q[-1].append(g[j])
                v[j]=1
                j=g[j]
    x=[0 for i in range(l)]
    for i in range(l):
        a,b=ind[g[i]]
        cl=len(q[a])
        x[i]=q[a][(b+p%cl)%cl]
    return x

n=100000
```

```
g=[i for i in range(n)]
np.random.shuffle(g)
orde= order(g)
c,d=randint(0,orde-1),randint(0,orde-1)

g_c=pow(g,c)
g_d=pow(g,d)

#g_c_d=pow(g_c,d)

#find_g_c_d function is used
#for calculating the value of (g^c)^d from g, g_c and g_d
def find_g_c_d(g,g_c,g_d):
# The function f is used in order to calculate 'd' from g and g^
    def f(g,z):
        q=[]
        ind={}
        l=len(g)
        v=[0 for i in range(l)]
        for i in range(l):
            if v[i]==0:
                v[i]=1
                q.append([g[i]])
                ind[g[i]]=[len(q)-1,0]
                j=g[i]
                while v[j]==0:
                    ind[g[j]]=[len(q)-1,len(q[-1])]
                    q[-1].append(g[j])
```

```
                v[j]=1
                j=g[j]
    v=set()
    x=[]
    for i in range(l):
        a,b=ind[z[i]]
        if a not in v:
            v.add(a)
            x.append([len(q[a]),b])
    l=len(x)
    if l==1:return x[0][1]
    def gcdExtended(a, b):
        if a == 0 :
            return 0,1
        x1,y1 = gcdExtended(b%a, a)
        x = y1 - (b//a) * x1
        y = x1
        return x,y
    n = len(x)
    x.sort(key=lambda i:i[1],reverse=True)
    #chinese remainder theorem for non relative primes
    a1 = x[0][1]
    m1 = x[0][0]
    for i in range(1,n):
        a2 = x[i][1]
        m2 = x[i][0]
        if m2==0:
```

```
                a1*=a2
                continue
            gc=gcd(m1,m2)
            p,q=gcdExtended(m1//gc, m2//gc)
            mod = (m1*m2)//gc
            zzz = (a1*(m2//gc)*q + a2*(m1//gc)*p)%mod
            a1 = zzz
            if (a1 < 0):
                a1 += mod
            m1 = mod

        return a1
    dd=f(g,g_d)
    return pow(g_c,dd)

print(find_g_c_d(g,g_c,g_d))#printing the (g^c)^d
# verifying whether the answer is correct or not
print(find_g_c_d(g,g_c,g_d)==pow(g_c,d))
```

CODES

LINKS to code

1 Q1_code_midsem.ipynb

2 Q2_code_midsem.py

RESOURCES

https://en.wikipedia.org/wiki/Chinese_remainder_theorem

https://en.wikipedia.org/wiki/Permutation_group