# Assignment 4

Maxime CHAMBREUIL
McGill ID: 260067572
maxime.chambreuil@mail.mcgill.ca

## Contents



# 1 Exercises from Stinson's book

## 1.1 DES

To demonstrate that $c[DES(x, K)] = DES[c(x), c(K)]$, we have to consider the different steps of the DES encryption :

### 1.1.1 IP and IP$^{-1}$

The multiplication by IP at the beginning and by IP$^{-1}$ at the end do not change anything : It is just a permutation so the complemented input gives the complemented output.

### 1.1.2 From i-1 to i

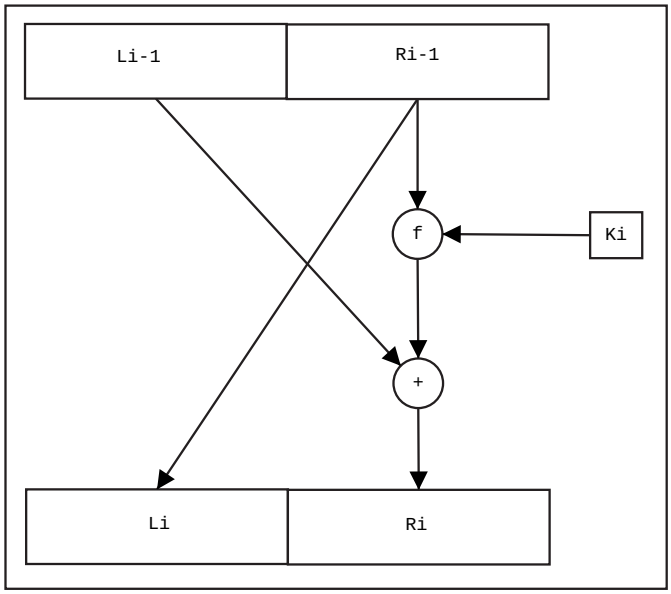


Figure 1: From i-1 to i

We know that $R^{i-1}$ is copied to $L^i$, so complemented $R^{i-1}$ gives complemented $L^i$.

### 1.1.3 The f function

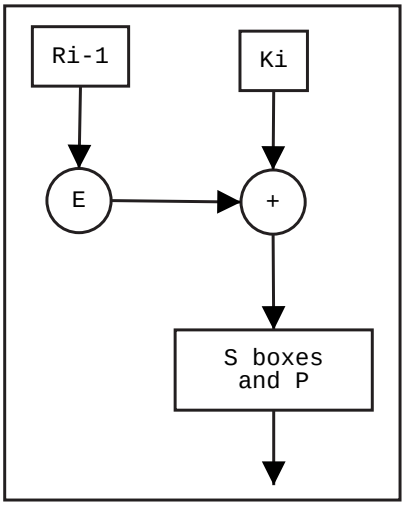


Figure 2: f function

Concerning the input of the f function $R^{i-1}$ and $K^i$, $R^{i-1}$ will be first expanded, so the output of the expansion is the complemented output when the input is complemented. The output of the expansion is xored with the key, so if both of them is complemented, it does not change anything to the output. Application of S-Boxes and the P permutation do not change anything. To sum-up, input and complemented input gives the same output of f.

The xor operation between $L^{i-1}$ and the output of f will lead to the complemented output as only one argument is complemented. Therefore, to obtain the 1-step output of complemented input, we only need to compute the complement of the regular output.

In conclusion, we have demonstrated that the output of complemented input is the complemented output of DES :

$$c[DES(x, K)] = DES[c(x), c(K)]$$

## 1.2 AES Key Expansion

```
#####################################################
KeyExpansion := proc(key,poly):
        RCon[1] := convert(16^6,hex):
        RCon[2] := convert(2*16^6,hex):
        RCon[3] := convert(4*16^6,hex):
        RCon[4] := convert(8*16^6,hex):
        RCon[5] := convert(16^7,hex):
        RCon[6] := convert(2*16^7,hex):
        RCon[7] := convert(4*16^7,hex):
        RCon[8] := convert(8*16^7,hex):
        RCon[9] := convert(16^7+11*16^6,hex):
        RCon[10] := convert(3*16^7+6*16^6,hex):

        for i from 1 to 4 do:
                w[i] := [key[4*i-3],key[4*i-2],key[4*i-1],key[4*i]]:
        end do:

        for i from 4 to 43 do:
                tmp := w[i]:
                if (i = 0 mod 4) then:
                        tmp := Xor(SubWord(RotWord(tmp),poly),RCon[i/4+1],poly):
                end if:
                w[i+1] := Xor(w[i-3],tmp,poly):
        end do:
        return w:
end proc:

#####################################################
getPoly := proc(deg):
        myPolynom;
        myPolynom := RandomTools[Generate](polynom(integer(range=0..1),
                                                z,degree=deg)):
        while (not (Irreduc(myPolynom) mod 2)) do
                myPolynom := RandomTools[Generate](polynom(integer(
                                        range=0..1),z,degree=deg)):
        end do;
```

```
        myPolynom;
end proc:

#####################################################
Xor := proc(x,y,poly):
        F := GF(2,128,poly);
        X := F[input](x);
        Y := F[input](y);
        R := F['+'](X,Y);
        r := F[output](R);
        return r;
end proc:

#####################################################
SubWord := proc(word,poly):
        for i from 1 to 4 do:
                res[i] := SubBytes(convert(word[i],binary),poly):
        end do:
        return res;
end proc:

#####################################################
RotWord := proc(word):
        res := (word[2],word[3],word[4],word[1]);
        return res;
end proc:

#####################################################
SubBytes := proc(a,poly):
        z := BinaryToField(a,poly):
        if (z &lt;&gt; 0) then:
                z := FieldInv(z,poly);
        end if:
        a1 := FieldToBinary(z,poly):
        c := convert(99,binary):
        for i from 1 to 8 do:
                b[i] := (a1[i] + a1[(i+4) mod 8] + a1[(i+5) mod 8] +
                        a1[(i+6) mod 8] + a1[(i+7) mod 8] + c[i]) mod 2:
        end do:
        return b:
end proc:

#####################################################
FieldInv := proc(z,poly):
        F := GF(2,128,poly);
        one := F[input](1):
        r := F['/'](one,z):
        return r:
end proc:

#####################################################
FieldToBinary := proc(z,poly):
        F := GF(2,128,poly):
        tmp := F[output](z);
```

```
        r := convert(tmp,binary):
        return r;
end proc:

#####################################################
BinaryToField := proc(z,poly):
        tmp := convert(z,binary):
        F := GF(2,128,poly):
        r := F[input](z):
        return r:
end proc:

#####################################################
key[1] := convert(16*2+11,hex):
key[2] := convert(7*16+14,hex):
key[3] := convert(16+5,hex):
key[4] := convert(16+6,hex):
key[5] := convert(16*2+8,hex):
key[6] := convert(16*10+14,hex):
key[7] := convert(16*13+2,hex):
key[8] := convert(16*10+6,hex):
key[9] := convert(16*10+11,hex):
key[10] := convert(16*15+7,hex):
key[11] := convert(16+5,hex):
key[12] := convert(16*8+8,hex):
key[13] := convert(9,hex):
key[14] := convert(16*12+15,hex):
key[15] := convert(16*4+15,hex):
key[16] := convert(16*3+12,hex):

poly := getPoly(128):

KeyExpansion(key,poly);
```

### 1.3 AES Encryption

```
#####################################################
AES_enc := proc(plain,key):
        poly := getPoly(128):
        roundKey := KeyExpansion(key,poly):
        tmp := initState(plain):
        tmp := AddRoundKey(tmp,roundKey);
        tmp := ShiftRows(tmp):
        for i from 1 by 1 to Nr-1 do:
                SubBytes():
                ShiftRows():
                MixColumn(tmp,poly):
                AddRoundKey(tmp,roundKey):
        end do:
        tmp := SubBytes(tmp);
        tmp := ShiftRows(tmp);
        tmp := AddRoundKey(tmp,roundKey):
        output := initState(tmp):
        return y:
```

```
end proc:

#####################################################
initState := proc(input):
        for i from 1 by 2 to 8 do:
                output[ceil(i/2)] := [input[i..i+1],input[8+i..9+i],
               input[16+i..i+17],input[24+i..i+25]]:
        end do:
        return output:
end proc:

#####################################################
AddRoundKey := proc(input,roundKey):
        output := Xor(input,roundKey);
        return output:
end proc:

#####################################################
ShiftRows := proc(input):
        output[1] := input[1]:
        output[2] := [input[2][4],input[2][1],input[2][2],input[2][3]]:
        output[3] := [input[3][3],input[3][4],input[3][1],input[3][2]]:
        output[4] := [input[4][2],input[4][3],input[4][4],input[4][1]]:
        return output:
end proc:

#####################################################
MixColumn := proc(input,poly):
        for i from 1 by 1 to 4 do:
                #t[i] := BinaryToField(input[i][],poly):
        end do:
        u[1] := Xor(t[4],Xor(t[3],Xor(FieldMult(x+1,t[2]),FieldMult(x,t[1])))):
        u[2] := Xor(t[1],Xor(t[4],Xor(FieldMult(x+1,t[3]),FieldMult(x,t[2])))):
        u[3] := Xor(t[2],Xor(t[1],Xor(FieldMult(x+1,t[4]),FieldMult(x,t[3])))):
        u[4] := Xor(t[3],Xor(t[2],Xor(FieldMult(x+1,t[1]),FieldMult(x,t[4])))):
        for i from 1 by 1 to 4 do:
                #:= FieldToBinary(u[i]);
        end do:
end proc:

#####################################################
FieldMult := proc(op1,op2,poly):
        F := GF(2,128,poly);
        r := F['*'](op1,op2):
        return r:
end proc:

#####################################################
plain := "3243F6A8885A308D313198A2E0370734":
x := AES_enc(plain,key):
```

# 2 Pseudo-Random Permutation Generator (PRΠG)

## 2.1 $\pi_k(x, y)$ is a permutation

We know that:

$$\pi_k(x, y) = [y, x \oplus f_k(y)] = [u, v]$$

Demonstrating that $\pi_k$ is a permutation consists in inverting the function, ie expressing the input x and y with the output u and v. Finding y is trivial : y = u, so let's focuse on x :

$$v = x \oplus f_k(v) \Rightarrow x = v \oplus f_k(v)$$

In conclusion, $\pi_k(x, y)$ is a permutation.

## 2.2 $\pi_k(x, y)$ is not a PRΠG

$\pi_k(x, y)$ is not a PRΠG if we can distinguish the output of $\pi_k$ from a random generator.

Let's ask to the 2 generators many output of known input, we can distinguish the output $\pi_k$ because after a certain amount of questions, we will notice that one generator copies a part of the input to a part of the output: $y \rightarrow u$.

So $\pi_k(x, y)$ is not a PRΠG.

## 2.3 $\pi_{k_1,k_2}(x, y)$ is not a PRΠG

The proof is mostly the same, we first ask for the output of x and y :

$$\pi_{k_1,k_2}(x, y) = \Pi_{k_1}[t, u] = \Pi_{k_1}[\Pi_{k_1}(x, y)] = [v, w]$$

So we have:

$$\begin{cases} t &= y \\ u &= x \oplus f_{k_2}(t) \end{cases} \quad \begin{cases} v &= u \\ w &= t \oplus f_{k_1}(v) \end{cases} \Rightarrow \begin{cases} v &= u = x \oplus f_{k_2}(y) \\ w &= y \oplus f_{k_1}(v) = y \oplus f_{k_1}[w \oplus f_{k_1}(v)] \end{cases}$$

Now let assume we have the stupid idea of asking to the generator the output of v and y. So we obtain:

$$\pi_{k_1,k_2}(v, y) = [a, b]$$

So we have:

$$\begin{cases} a &= v \oplus f_{k_2}(y) = x \oplus f_{k_2}(y) \oplus f_{k_2}(y) = x \\ b &= y \oplus f_{k_1}(a) \end{cases}$$

We have found a way to distinguish $\pi_{k_1,k_2}(x, y)$ from a random generator. Now we can initiate the T function to ask ( v , y ) after asking ( x , y ). With a high probability, the T function is going to distinguish $\pi_{k_1,k_2}(x, y)$.

In conclusion, we have proved that $\pi_{k_1,k_2}(x, y)$ is not a PRΠG.

# 3 Factoring

## 3.1 $\sqrt{p} - \sqrt{q} < \sqrt{2} \Rightarrow \lceil \sqrt{n} \rceil = \frac{p+q}{2}$

We know that $n = p \times q$, so :

$$\begin{aligned}
\sqrt{p} - \sqrt{q} < \sqrt{2} &\Rightarrow (\sqrt{p} - \sqrt{q})^2 < 2 \\
&\Rightarrow p + q - 2\sqrt{p \times q} < 2 \\
&\Rightarrow p + q < 2\sqrt{n} + 2 \\
&\Rightarrow \frac{p+q}{2} < \sqrt{n} + 1 \\
&\Rightarrow \frac{p+q}{2} - \sqrt{n} < 1 \\
&\Rightarrow \lceil \sqrt{n} \rceil = \frac{p+q}{2}
\end{aligned}$$

## 3.2 Efficient algorithm to factor RSA modulus

Assuming that p and q verify $\sqrt{p} - \sqrt{q} < \sqrt{2}$, the algorithm mostly consists in solving a system of 2 equations to find our 2 unknowns p and q. The system is :

$$\begin{cases} n &= p \times q \\ \lceil \sqrt{n} \rceil &= \frac{1}{2}p + \frac{1}{2}q \end{cases}$$

So any algorithm solving system of equation can be chosen.

## 3.3 Generalization

To generalize this method, we have to add a loop on a and another one inside, on b. For each couple ( a , b ), we can compute k and solve the following system of equation :

$$\begin{cases} n &= p \times q \\ \lceil \sqrt{k \times n} \rceil &= \frac{a}{2}p + \frac{b}{2}q \end{cases}$$

## 3.4 Maple$^{TM}$

```
factor := proc(n):
        for a from 1 by 1 to 10 do:
                for b from 1 by 1 to 10 do:
                        k := a*b;
                        mid := ceil(sqrt(k*n)):
                        if (evalf(mid^2 - k*n) > 0) then :
                                q := evalf((mid + sqrt(mid^2 - k*n))/b);
                                p := ceil(n/q);
                                print(a):
                                print(b):
                                if (isprime(p) and isprime(q)) then:
                                        print(p);
                                        print(q);
                                end if:
                        end if:
                end do:
        end do:
end proc:
factor(n);
```

# 4 Block Cipher Modes of Operations

```
with(numtheory):
enc:= (p,e,m) -> m&^e mod p;
dec:= (p,e,c) -> c&^(e^(-1) mod (p-1)) mod p;
p:=nextprime(2^64+1000027);
e:=16755434444356788983;

#####################################################
getPoly := proc(deg):
        myPolynom;
        myPolynom := RandomTools[Generate](polynom(integer(range=0..1),
                                                    z,degree=deg)):
        while (not (Irreduc(myPolynom) mod 2)) do
                myPolynom := RandomTools[Generate](polynom(integer(
                                        range=0..1),z,degree=deg)):
        end do;
        myPolynom;
end proc:

#####################################################
CBC_enc := proc(IV,LIST,poly):
        y := IV;
        for i from 1 to 9 do :
                x := LIST[i];
                m := Xor(x,y,poly);
                c := enc(p,e,m);
                C[i] := c;
                y := c;
        end do:
        return C;
end proc:

#####################################################
CBC_dec := proc(IV,LIST,poly):
        y := IV;
        for i from 1 to 9 do:
                c := LIST[i]:
                m := dec(p,e,c):
                x := Xor(y,m,poly):
                X[i] := x;
                y := c:
        end do:
        return X:
end proc:

#####################################################
CBC_mac := proc(IV,LIST,poly):
        CIPHER := CBC_enc(0,LIST,poly):
        CIPHER[9];
end proc:

#####################################################
Xor := proc(x,y,poly):
```

```
        F := GF(2,64,poly);
        X := F[input](x);
        Y := F[input](y);
        R := F[‘+‘](X,Y);
        r := F[output](R);
        return r;
end proc:

#######################################################
LIST := [2,7182818, 2845904523, 53, 6028, 747135, 2662, 497757,51]:
IV := 260067572:
poly := getPoly(64);
C := CBC_enc(IV,LIST,poly):

C := ([1 = 14234119755384022983,
        2 = 12854045216180404268,
        3 = 15510786419853357683,
        4 = 6264414815738077594,
        5 = 5832262802199696360,
        6 = 636564044016271865,
        7 = 12051844559106660086,
        8 = 16651342758141191731,
        9 = 17839007384925899666])

X := CBC_dec(IV,LIST,poly):

X := ([1 = 7994377085772277046,
        2 = 786611987444478897,
        3 = 10123423765670063232,
        4 = 17600641552839169863,
        5 = 12091993564561445566,
        6 = 9454150191315461950,
        7 = 17733235922899929185,
        8 = 13547871925566075631,
        9 = 16651690207461933746])

MAC := CBC_mac(IV,LIST,poly);

MAC := 15713183176855184421
```