TP 2 AES et RSA

Table des matières

TP 2 AES et RSA

1. Multiplication FF pour AES

Écrivez et testez le programme suivant :

```
1
2 # multiplication entre polynome a et b dans FF(2^8)
3 def multiplicationFF(a,b):
4      print "Init A= ",bin(a),printPolynome(a)
5      print "Init mod= ",bin(283),printPolynome(283)
6      print "Init B= ",bin(b),printPolynome(b)
7      p=0
8      i=1
9      while b!=0:
10          if b & 1:
11               p^=a
12          print 'iteration=',i
13          print "P="+printPolynome(p)+" "+bin(p)
14          if a>=128:
15               a= (a*2)^283
16          else:
17               a*=2
18          print "A="+printPolynome(a)+" "+bin(a)
19          b>>=1
20          i+=1
21     return p
22 def printPolynome(a):
23     b=bin(a)[2::]
24     l=len(b)
25     p=""
26     pol=[]
27     if(b[l-1]=="1"):
28          pol.append("1")
29
30     for i in range(l-2,-1,-1):
31          if b[i]=='1':
32               pol.append("x^"+str(l-i-1))
33     return str(pol)
34 print "234 x 215" , multiplicationFF(234 , 215)
35
```

2. Fast Exponential Algorithm

Testez le programme suivant pour comparer le temps d'exécution de l'algorithme fastExponential (utilisé par RSA) et celui de l'algorithme naïve :

```
def fastExponential(x,n):
     start=timeit.default_timer()
     P=1
     bn=bin(n)[2::]
     l=len(bn)-1
     bn=bn[::-1]
     #print(bn)
     for i in range(l,-1,-1):
          P*=P
          if bn[i]=="1":
               P*=x
     print "finished after ",(timeit.default_timer()-start)
     return P
def naiveExponential(x,n):
     start=timeit.default_timer()
     P=1
     a=0
     while a<n:
          P*=x
          a+=1
     print "finished after ",(timeit.default_timer()-start)
     return P

s=2**16
fastExponential(3,s)
naiveExponential(3,s)
```

3. RSA

Exécutez l'algorithme RSA afin de chiffrer / déchiffrer le bloque "INSERTPYTHONMEME" :

```
1
2
3 def getkeys(p,q):
4         Blocsize=p*q
5         L=(p-1)*(q-1)
6         publicKey=0
7         for i in range(2,L):
8                 if( L%i!=0 and Blocsize%i!=0):
9                         publicKey=i
10                        break
11        keylist=list([filter ( lambda x : (x*publicKey)%L ==1, range(1,Blocsize
  ))])
12        privateKey=keylist[0][0]
13        return (publicKey,privateKey,Blocsize)
14
15 def RSA(data,k,n):
16     cypher=""
17     for c in data:
18          scrambled=(ord(c)**k)%n
19          cypher+=chr(scrambled)
20     return cypher
21 pbk,pvk,n=getkeys(13,17)# generation de cles prives et publiques
22 c=RSA("INSERTPYTHONMEME",pbk,n) # chiffrer le bloque "INSERTPYTHONMEME" avec la
  cle publique
23 print (c)
24 print( RSA(c,pvk,n)) # dechiffrer le bloque avec la cle prive
25
```