Project 3 RSA Block Size 4

For this project, an RSA crypto system was created using a block size of 4 characters. In order to test this, the baseband of the Darwin text file was used, and the encryption/decryption were created with Python.

The encoding of 4 character blocks is such that aaaa = 0, aaab = 1, …., zzzz = (26^4)-1.

To run this program, a file to be operated on must be selected. The Darwin.txt file was selected for encryption. The length of the file is N, and 2 prime numbers P and Q which multiply to create a value close to N must be selected. For this project, P = 683 and Q = 677 were selected, which multiply to create 5000 extra values which won't be used.

P and Q are then sent to the function KeyGen(p,q), which calculates Phi, B from GenB() based off of Phi, and A from GenA() based off of Phi and B. GenB and GenA find randomly valid A and B values. For this paper, A was 256129, B was 9.

Once the values P,Q,N,Phi,B, and A are found, the file can be modified. The text file is split into 4 character chunks, where each chunk is then encoded into a number. The steps to encode this involve reading each character in a chunk, for C[3|2|1|0], where the chunk is iterated through with:
C3 = C3*(26^3), C2 = C2*(26^2), C1 = C1 * 26, and C0 = C0. These values are summed, and are then sent to the encryption function.

SquareandMultiply (SAM) is used for both decryption and encryption. To encrypt, the value X is taken to encrypt, B, and N. SAM is then able to efficiently calculate (X^B) mod (N). This value is then returned, and is stored in a list of encrypted values. To decrypt, the value Y to be decrypted is passed, A, and N are sent, to calculate (Y^A) mod (N).

With the list of encrypted values, these can be decoded to English text using the Decode() function. Decode() takes a value V, number of characters per block S, and the number of values possible for each block N. It then iterates S so S = 0, 1, …. , N, and decodes the values where

$$Cs = floor(Vs/(N^{(C-s)}));$$

$$Vs = (Vs-1 – (N^{(C-s)})*Cs-1);$$

The output is a list Cs, where each C can be mapped to an English character with 0 = a, 1 = b, … , 25 = z. This Decode() function works for both encrypted and decrypted values, like SAM.

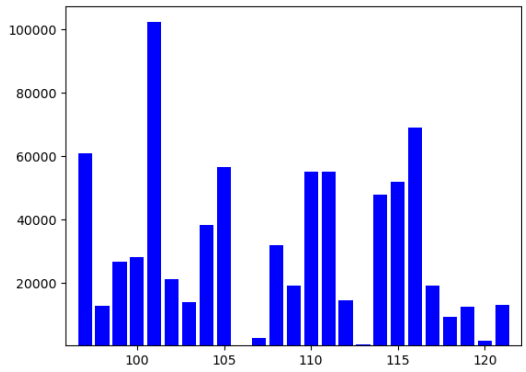
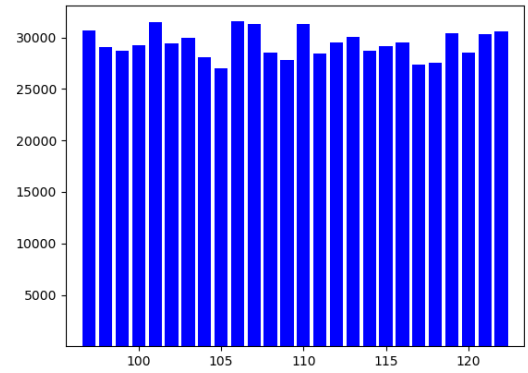
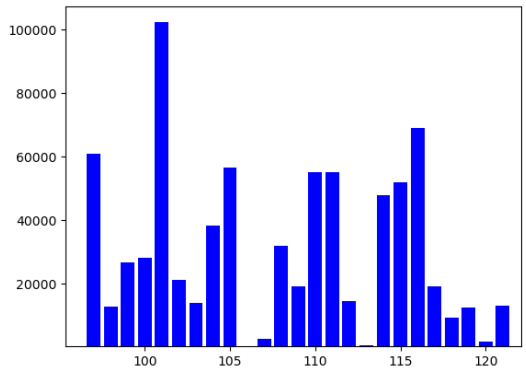
The program which was created runs with the hardcoded keys, generates ciphertext, tracks plaintext, and decrypts the ciphertext to show that the decryption function works. The program also automatically calculates the summary statistics and histograms for the plaintext and ciphertext. A sample of the ciphertext, with corresponding plaintext is shown below:

Kketufieawlokwyxqibfsuwscexwdgixdcapooibtprcjlayre

Ontheoriginofspeciesbymeansofnaturalselectionorthe

The summary statistics of plaintext, ciphertext, and decrypted text with corresponding histograms are shown below:

Summary Stats

LEN:766857 N:462391 A:256129 B:9 P:683 Q:677 PLAINTEXT MAX:122 MIN:97 AVG:107.54857756867 VAR:46.91453337697256 STD:6.849418469985066 ENT:19.545646836206238	
ENCRYPTED TEXT MAX:123 MIN:97 AVG:109.51279118895856 VAR:56.97235232832332 STD:7.548003201398587 ENT:19.545140518207983	
DECRYPTED TEXT MAX:122 MIN:97 AVG:107.54857756867 VAR:46.91453337697256 STD:6.849418469985066 ENT:19.545646836206238	

As can be seen above, the summary statistics of the plaintext and decrypted text are the same, meaning the decryption function was successful. They follow the frequency of the English language. However, the histogram of the ciphertext is very close to a uniform distribution, making frequency analysis of ciphertext almost impossible. The entropy of all three data sets are the same, meaning that no data is lost in the encryption/decryption process. The average of ciphertext is closer to the mid point of the language, which is another indicator of a uniform distribution. This is all with P = 683, P = 677, B = 9, A = 256129.

It is fair to say that this crypto system is not vulnerable to frequency analysis, but since the block size is 4, the list of primes which could be used to create a valid system are small enough that it would be easy to brute force potential A and B values from possible P and Q combinations. Since the system is public key, malicious attackers would be able to see N and B, reducing the amount of time needed to brute force this small block size. The best way to beat this would be to increase the block size. Make it inefficient to compute the SAM. Even though SAM is more efficient than the written (X^B) mod (N), large enough values of A and B make it too costly to crack the system.

The source code is shown below:

RSA.py

```
import math,random,pdb
from matplotlib import pyplot as plt


def GenB(phi):
    for i in range(1,phi):
        count = random.randint(0,50)
        if(math.gcd(phi,i)==1):
            b=i
            if(count == 50): return b
    return b


#reduce a, b to speed up
def GenA(phi,b):
    for i in range(1,phi):
        count = random.randint(0,50)
        if(((b*i)%phi)==1):
            a=i
            if(count == 50): return a
    return a
```

```
def sam(baseint,exponent,mod):
    b = bin(exponent)
    z = 1
    lenb = len(b)
    for i in range(1,lenb):
        z = (z*z)%mod
        if(b[i] == '1'):
            z = (z*baseint)%mod
    return z

#Generate valid RSA keys given two prime numbers
def keyGen(p,q):
    n = p*q
    phi = (p-1) * (q-1)
    b = GenB(phi)
    a = GenA(phi,b)
    return(a,b,n)

def BaseFind(v,m,c):
    return((v-(m*c)))

def Decode(v0,base,charnum):
    char,v = [],[v0]
    for i in range(charnum):
        basecur = (base)**(charnum-1-i)
        char.append(v[i]//basecur)
        v.append(BaseFind(v[i],basecur,char[i]))
    char = [chr(x+97) for x in char]
    return(char)
```

```
def OrdFind(c,base,power):
    c = ord(c)-97
    return(c*(base**power))

def Main(n,a,b,cnum,base,arr):
    lena = len(arr)
    print(f"LEN:{lena}\nN:{n}\nA:{a}\nB:{b}")
    for i in range(lena):
        if(i%200000 == 0):
            print(i)
        if((i+1)%cnum==0):
            cur = content[i-(cnum-1):i+1]
            tot = 0
            for i in range(cnum):
                tot += OrdFind(cur[i],26,cnum-1-i)
            plaintot.append(tot)
            ee = sam(tot,keyb,keyn)
            enctot.append(ee)
            dectot.append(sam(ee,a,n))

    return([plaintot,enctot,dectot])

def SummaryStat(arr):

    maxx,minn,leng = ord(max(arr)),ord(min(arr)),len(arr)
    summ,ss,ec,cc = 0,0,0,0
    yarr,xarr = [0]*(maxx+1),[0]*(maxx+1)
    #get avg
    for i in range(leng):summ += ord(arr[i])
    for i in range(maxx):xarr[i]+=i
```

```
    avgg = summ/leng
    #get entropy and std dev
    for i in range(leng-1):
        yarr[ord(arr[i])] += 1
        ss += abs((ord(arr[i])-(avgg))**2)
        if(ord(arr[i]) > 0):
            cc = ord(arr[i]) / summ
            ec = ec - cc*math.log(cc,2)/math.log(2,2)

    print(f"MAX:{maxx}\nMIN:{minn}\nAVG:{avgg}")
    print(f"VAR:{ss/leng}\nSTD:{math.sqrt(ss/leng)}\nENT:{ec}")
    return([maxx,minn,avgg,(ss/leng),(math.sqrt(ss/leng)),ec,xarr,yarr])

def CheckOutput(arr,base,charnum,name):
    print(name)
    lena,decodearr = len(arr),[]
    for i in range(lena):
        t = Decode(arr[i],base,charnum)
        for ii in range(charnum):
            decodearr.append(t[ii])
    decodearr = ''.join(e for e in decodearr)
    return(decodearr)

def plott(xarr,yarr):
    plt.bar(xarr,yarr,color='blue')
    plt.show()

#Open darwin, set p,q,n,a,b values, find stats, make histogram
f = open("darwinbaseband.txt","r")
content = f.read()
f.close()
```

```
clist,enctot,plaintot,dectot = [],[],[],[]
numchar,zspace = 4,26
p,q = 683,677
lenn = 766857
keyn = 462391
##keya = 371243
##keyb = 267
keyb = 9
keya = 256129
x = Main(keyn,keya,keyb,numchar,zspace,content)
checkplain = CheckOutput(x[0],zspace,numchar,"PLAINTEXT")
checkenc = CheckOutput(x[1],zspace,numchar,"ENCRYPTED TEXT")
checkdec = CheckOutput(x[2],zspace,numchar,"DECRYPTED TEXT")
statsplain = SummaryStat(checkplain)
statsenc = SummaryStat(checkenc)
statsdec = SummaryStat(checkdec)

##plott(statsplain[6],statsplain[7])
##plott(statsenc[6],statsenc[7])
##plott(statsdec[6],statsdec[7])
```