CS338 Assgnment 5

Kuo Wang

April 2022

1 Diffie Hellman

1. The shared secret is 4.
2. My code is written in Python. It is a brute force approach.

```
def main():
    running = True
    a = 1
    b = 1
    while running == True:
        bigB = (44**a)%59
        bigA = (57**b)%59
        if bigB == bigA:
            print("a="+str(a)+",b="+str(b))
            print("key="+str((11**(a*b))%59))
            return
        elif bigB < bigA:
            a+=1
        elif bigB > bigA:
            b+=1

if __name__ == "__main__":
    main()
```

3. With greater integer values referenced by shared information p, message A, and message B, the program will fail here:

```
bigB = (44**a)%59 #where B is 44
bigA = (57**b)%59 #where A is 57
```

We know that in Python modulo runtime is $O(log(m) * log(n))$ for $(m)mod(n)$. In our case, log(59) is a very small factor for the runtime, and the exponentially growing m (which are (44^a) and (57^b) respectively) will not affect much the overall calculation time and space needed.
Had p been larger, however, the modulo operation will become extremely slow. Recall that modulo runtime is $O(log(m) * log(n))$. With a great enough p that is n, and exponentially increasing B^a and A^b that is m, the runtime $O(log(m) * log(n))$ will eventually become so great that our computers will take virtually forever to calculate bigB and bigA.
Moreover, the runtime effects would be similar had the generated messages been longer, or the correct random numbers a and b are greater. If either of those is the case, m in the runtime equation will become very big very quickly. Also, space complexity will become an issue when trying to store tentative B^a and A^b values when they are so big.

2 RSA

1. The message is, all in one line:
Hey Bob. It's even worse than we thought! Your pal, Alice.
https://www.schneier.com/blog/archives/2022/04/airtags-are-used-for-stalking-far-more-than-previously-reported.html

2. The code is below.

```
def getAllPQ(num):
    allDivisors = []
    for i in range(2,num):
        if num%i==0:
            allDivisors.append((i, int(num/i)))
    return allDivisors

def getLCM(numTuple):
    firstNum = int(numTuple[0])
    secondNum = int(numTuple[1])
    while firstNum!=secondNum:
        # print(str(firstNum)+", "+str(secondNum))
        if firstNum < secondNum:
            firstNum+=int(numTuple[0])
        elif firstNum > secondNum:
            secondNum+=int(numTuple[1])
    return firstNum
```

```
def getLambda(numTuple):
    return getLCM((numTuple[0]-1,numTuple[1]-1))

def getDB():
    allDivisors = getAllPQ(5561)
    # print(allDivisors)
    for i in allDivisors:
        # print("trying "+ str(i[0]) + ", " + str(i[1]))
        currentLambda = int(getLambda(i))
        DA = 1
        x = 13*DA%currentLambda
        while x!=1:
            DA+=1
            if DA==999999:
                break
            x = int(13*DA%currentLambda)
        return DA
    return False

def decipherSingleRSA(secretASCII,DB,NB):
    print(str(secretASCII**DB%NB))
    return chr(secretASCII**DB%NB)

def decipherRSA():
    cipher = [1516, 3860, 2891, 570, 3483, 4022, 3437, 299,
    570, 843, 3433, 5450, 653, 570, 3860, 482,
    3860, 4851, 570, 2187, 4022, 3075, 653, 3860,
    570, 3433, 1511, 2442, 4851, 570, 2187, 3860,
    570, 3433, 1511, 4022, 3411, 5139, 1511, 3433,
    4180, 570, 4169, 4022, 3411, 3075, 570, 3000,
    2442, 2458, 4759, 570, 2863, 2458, 3455, 1106,
    3860, 299, 570, 1511, 3433, 3433, 3000, 653,
    3269, 4951, 4951, 2187, 2187, 2187, 299, 653,
    1106, 1511, 4851, 3860, 3455, 3860, 3075, 299,
    1106, 4022, 3194, 4951, 3437, 2458, 4022, 5139,
    4951, 2442, 3075, 1106, 1511, 3455, 482, 3860,
    653, 4951, 2875, 3668, 2875, 2875, 4951, 3668,
    4063, 4951, 2442, 3455, 3075, 3433, 2442, 5139,
    653, 5077, 2442, 3075, 3860, 5077, 3411, 653,
    3860, 1165, 5077, 2713, 4022, 3075, 5077, 653,
    3433, 2442, 2458, 3409, 3455, 4851, 5139, 5077,
    2713, 2442, 3075, 5077, 3194, 4022, 3075, 3860,
    5077, 3433, 1511, 2442, 4851, 5077, 3000, 3075,
    3860, 482, 3455, 4022, 3411, 653, 2458, 2891,
    5077, 3075, 3860, 3000, 4022, 3075, 3433, 3860,
    1165, 299, 1511, 3433, 3194, 2458]
    DB = getDB()
    NB = 5561
```

```
    decryted = ""
    for i in cipher:
        decryted+=(decipherSingleRSA(int(i),DB,NB))
    return decryted

if __name__ == "__main__":
    print(decipherRSA())
```

3. In my brute force process, the program will first figure out the whole set of (p, q) from $n_b = p_B q_B$. The process is done by $getAllPQ$ in $O(n_B(log^2(n_B)))$ time with checking modulos for all $2 \leq n < n_B$. When n_B becomes large, this first step will be extremely time consuming. Moreover, with a larger n_B, storing all its divisor tuples is not space efficient either.
   Then, each set of divisors are checked up to 9999 times to test d_b values 1-9999 to see whether $e_b * d_B mod\lambda(n_b) == 1$. Here, a weak point arises: if by any chance there is a correct value for d_B that is greater than 9998 such that $e_b * d_B mod\lambda(n_b) == 1$, this program is unable to find it. Therefore, a somewhat big d_B will render the program useless.

4. The encryption is insecure because RSA encryption is individually applied to each letter. This way, the encryption becomes a substitution cipher that is prone to attacks with letter frequency and a dictionary. RSA is better used to encrypt the whole message at once, not letter-by-letter.