Menu

August 26, 2024

Generating a key: To generate a key, we first find 2 prime numbers, we will call p and q. These prime numbers need to multiply to a number greater than 150, n. For an example let's use 7 and 23 for p and q, that gives an n value of 161. We then choose an e value that is relatively prime to (p-1)*(q-1), let's use 5. Now we have our n and e of (161, 5), so we just have to get d now, which is the inverse of e % (p-1)*(q-1). In our case that ends up being 5*d % (132), 53 as 5* 53 % 132 = 1.

Encoding a message: To encode a message we need to first convert a message to a list of ASCII keys. Then with this list, we can apply the FME algorithm to encrypt them into the encoded message. When calling FME, we can't just take the mod of our number as directly using mod can cause issues with very large numbers, leading to potential overflow of ints. Furthermore, FME is much more efficient and generally the accepted practice as compared to directly using mod.

Decoding a message: To decode the same message, we must have access to the private key of 53. Then we use FME again, but use d instead of e as the exponent as it is the inverse of e. Then once we have the decoded numbers, we covert them to text based on the ASCII values. If we don't have d, then we can use some other functions we created to calculate it with the EEA. This is why the EEA is key as it allows us to break codes in which we don't have access to d.

Cracking a code: To break a code, we must find the prime factors of n, this is what our factorize function is doing. Once we have p and q, the factors, we can easily plug into the Find_Private_Key_d function, which uses (p-1) x (q-1) and the EEA find the private key. We can use the Extended Euclidean Algorithm to find the private key d as e x d = 1 (mod (p-1) x (q-1)).

[2]:
```python
def Convert_Binary_String(_int):
    return bin(_int)[2:]

def FME(b, n, m):
    x = 1 #stores final result
    power = b % m
    binary = Convert_Binary_String(n) #get power in binary,
    binary = binary[::-1] # reverse it to iterate right to left
    for i in range(len(binary)): #for loop to run through each bit in binary string
        if binary[i] == '1': #if bit is 1
            x = (x*power) % m   #update result with current power
        power = (power*power) % m #update power for next interation
    return x #final result

def Euclidean_Alg(a, b):
```

```
    while b != 0:
        remainder = a % b #calculate remainder of a/b
        a = b
        b = remainder
    return a #when b is 0, a is GCD and returns

def EEA(a, b):
    #check for pos. int.

    if a <= 0 or b <=0:
        print("Error, inputs must be positive integers")
        return

    #initalize variables
    inital_a = a
    inital_a = b
    (s1, t1) = (1, 0) #coeff. for a
    (s2, t2) = (0, 1) #coeff. for b

    while b > 0: #loop until remainder is 0
        k = a % b #remainder
        q = a // b #integer division to get quotient

        #update m and n
        a = b
        b = k

        (s1H, t1H) = (s2,t2) #"bookkeep to hold current s2,t2"
        (s2H, t2H) = (s1 - q*s2, t1 - q*t2) #update using bezout coeff. equation

        #move current coeffs. to next interation
        (s1,t1) = s1H,t1H
        (s2,t2) = s2H,t2H

    return a,(s1,t1)

def Find_Public_Key_e(p, q):
    n = p*q
    rp = (p-1)*(q-1) #rp is what e should be relatively prime to
    e = 3 #start with small prime number
    while Euclidean_Alg(e, rp)!=1 & e!=p & e!=q: #loop incrementing by 2 for
 ↪odd numbers, to look for relatively prime e
        e += 2
    return (n,e)

def Find_Private_Key_d(e, p, q):
    rp = (p-1)*(q-1)
```

```python
    gcd, (s1,t1) = EEA(e, rp) #EEA to find gcd, and coeffs.

    if gcd!=1: #if gcd is not 1, e and rp aren't co prime
        print("No modular inverse exits")
        return

    d = s1 % rp #x is modular inverse of e

    if d<0: #if d isn't positive, add rp
        d+= rp
    return d

def Convert_Text(_string):
    integer_list = []
    for char in _string:
        integer_list.append(ord(char)) #run through string as a list and convert each char to ascii then add to integer list
    return integer_list

def Convert_Num(_list):
    _string = ''
    for i in _list:
        _string += chr(i)
    return _string

def Encode(n, e, message):
    cipher_text = []
    asc = Convert_Text(message) #convert string to ascii list
    for num in asc:
        cipher_text.append(FME(num,e,n))  #RSA to convert to cipher text

    return cipher_text

def Decode(n, d, cipher_text):
    message = ''
    decrypted = []
    for num in cipher_text:
        decrypted.append(FME(num,d,n)) #RSA decode to get ASCII in decrypted list

    message = Convert_Num(decrypted) #convert ascii list to text
    return message

def factorize(n):
    for i in range(2, n-1):
        if n%i == 0:
            return i
```

```
    return FALSE
```

[3]:
```
def display_menu():
    print("-----Hello! Welcome to the RSA Cryptography Menu------")
    print("Option 1: Encode a message")
    print("Option 2: Decode a message")
    print("Option 3: Break Key")
    print("Option 4: Get random primes to encode (up to 100)")
    print("Option 5: Exit Program")
```

[4]:
```
import random
def main():
    while(True):
        display_menu()
        choice = int(input())
        #Choice 1 encode a message
        if choice == 1:
            #taking input
            print("Enter 1st prime number.")
            p = int(input())
            print("Enter 2nd prime number.")
            q = int(input())
            #calculating n and taking e input
            n = p*q
            print("Enter a key (e) value (65537 reccomended).")
            e = int(input())
            print("Enter your message")
            msg = input()
            encoded1 = Encode(n, e, msg) #encoding message
            print(f"Encoded Message: {encoded1}")
        #Choice 2 decoding a msg
        elif choice == 2:
            print("Enter the encoded message")
            encoded_msg2 = [int(num) for num in input()[1:-1].split(",")]␣
↪#split string by commas, ignore brackets and then convert each num to int

            #taking input
            print("Enter n value")
            n_2 = int(input())
            print("Enter private key d")
            d_2 = int(input())
            decrypted_2 = Decode(n_2, d_2, encoded_msg2) #decrypting message
            print(f"Decoded Message: {decrypted_2}")
        #choice 3 breaking a msg with no private key
        elif choice == 3:
            print("Enter encoded message")
```

```
            encoded_msg3 = [int(num) for num in input()[1:-1].split(",")] #split string by commas, ignore brackets and then convert each num to int
            print("Enter n value")
            n_3 = int(input())
            print("Enter key (e) value")
            e_3 = int(input())

            #calculating prime numbers p and q
            p3 = factorize(n_3)
            q3 = n_3//p3

            #calculating private key d
            d3 = Find_Private_Key_d(e_3, p3, q3)
            decrypted_3 = Decode(n_3, d3, encoded_msg3) #decoding message
            print(f"Decoded Message: {decrypted_3}")
        #choice 4 generate prime nums
        elif choice == 4:
            primes = [2, 3, 5, 7, 11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97] #list of prime numbers
            #picking random number in primes list
            prime1 = random.choice(primes)
            prime2= random.choice(primes)

            if(prime2 == prime1): #condition to make sure p and q aren't equal
                prime2 = random.choice(primes )
            print(f"Prime 1: {prime1} Prime 2: {prime2}")
        #exit program choice
        elif choice == 5:
            print("Goodbye!")
            break
        #default case
        else:
            print("invalid option, try again")

main()
```

```
-----Hello! Welcome to the RSA Cryptography Menu------
Option 1: Encode a message
Option 2: Decode a message
Option 3: Break Key
Option 4: Get random primes to encode (up to 100)
Option 5: Exit Program
```

```
 5
```

```
Goodbye!
```

Code Breaking Examples:

[5]:
```python
#Example 1: n, e = 130177, 13 #Public Key
n = 130177
e = 13
cipher = [85308, 48594, 15927, 71285, 61037, 15927, 767, 23406, 71285, 23406,
 ↪48594, 12676, 37298, 100459, 71285, 90170, 61037, 38946, 38783, 23406,
 ↪71285, 90170, 71285, 113492, 38946, 38946, 86792, 15927, 90170, 37298,
 ↪71285, 12676, 767, 71285, 15927, 121493, 15927, 37298, 71285, 12676, 84628,
 ↪71285, 29228, 38946, 38783, 71285, 90170, 110238, 15927, 71285, 81798,
 ↪110238, 38946, 37298, 100459, 126494, 71285, 29228, 38946, 38783, 71285,
 ↪90170, 110238, 15927, 71285, 38946, 37298, 86792, 29228, 71285, 38946,
 ↪84628, 84628, 71285, 61037, 29228, 71285, 90170, 71285, 61037, 12676, 23406,
 ↪10510]
p = factorize(n) #use factoring alg. to get p
q = n//p #divide to get q

d = Find_Private_Key_d(e, p, q) #calculate d

decrypted_msg = Decode(n, d, cipher) #We have d so we can decrypt the cipher
print(decrypted_msg)
print(f"p:{p}, q{q}")
```

```
The best thing about a Boolean is even if you are wrong, you are only off by a
bit.
p:349, q373
```

In this example we have values n and e of 130177 and 13. To decode our given cipher, we apply the same algorithm giving us a p value of 349 and q of 373. Once we obtain these, we just plug into our private key function which uses the EEA to calculate d. Then we can just use our Decode function with n, d and the cipher and that leaves us with the decoded message of "The best thing about a Boolean is even if you are wrong, you are only off by a bit."

[6]:
```python
#Example 2:
cipher = [490, 503, 602, 355, 482, 157, 139, 437, 466, 503, 602, 466, 336, 482,
 ↪157, 336, 297, 363, 602, 355, 466, 139, 105, 512, 466, 336, 503, 692, 512]
n = 713
e = 653
p = factorize(n)
q = n//p
d = Find_Private_Key_d(e, p, q)
decrypt = Decode(n, d, cipher)
print(decrypt)
```

```
Congrats on cracking the code
```

[ ]: