IS53012C Security and Encryption Coursework 2020–21

Total Number of Hours Spent	68
Hours Spent for Algorithm Design	10
Hours spent for Programming	30
Hours spent for Writing Report	15
Hours Spent for Testing	13
Note for the examiner (if any)	The program needs the NumPy library to run Please use the command python3 interface.py in terminal.

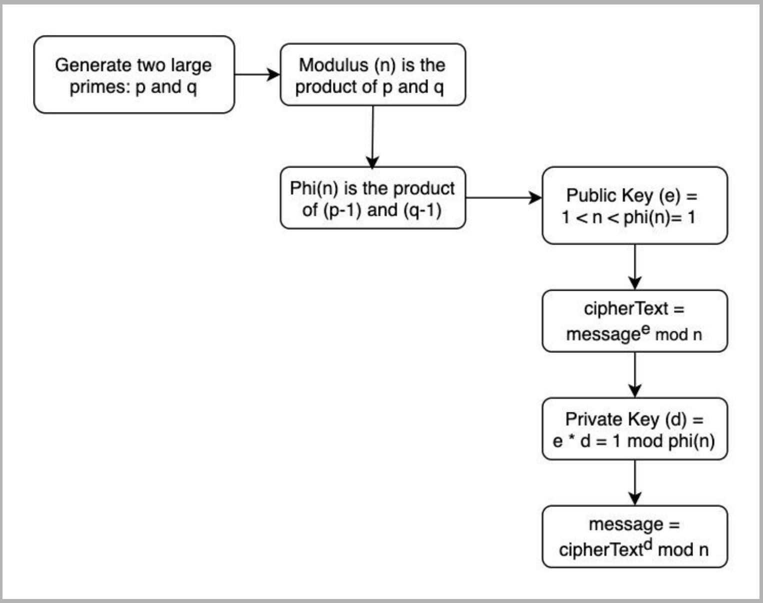
Student: Maxsuel Trajano

The RSA

The RSA, named after its creators (Rivest-Shamir-Adleman), is an asymmetric algorithm used by modern cryptography. An asymmetric algorithm means that one key is used for encryption and the other is used for decryption. The algorithm is based on the fact that finding the prime factors of a large number is hard, but computing such values is relatively easy. RSA uses two keys called public key and private key. There are six main components that are used to create these keys: p, q, n phi, e, d, where:

- p and q are two large prime numbers
- n = p * q, which is used as the modulus
- phi = (p-1) * (q-1), which is used to calculate the public key (e)
- e = 1 < e < phi(n) == 1
- d = e * d = 1 mod phi(n)

The public key (e,n) is released to the public and the private key (d) is kept secret.


Generate two large primes: p and q
Modulus (n) is the product of p and q
Phi(n) is the product of (p-1) and (q-1)
Public Key (e) = 1 < n < phi(n)= 1
cipherText = message^e mod n
Private Key (d) = e * d = 1 mod phi(n)
message = cipherText^d mod n

My Implementation:

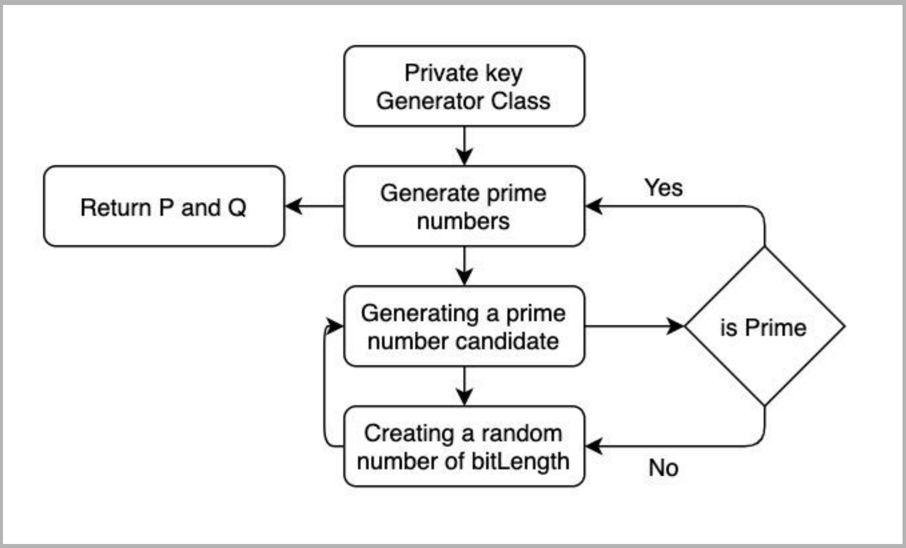
The implementation uses an object-oriented approach, with the idea of abstracting away from the user of all of the required functions to get RSA to work.

Getting the key components of RSA

Generating Prime Numbers

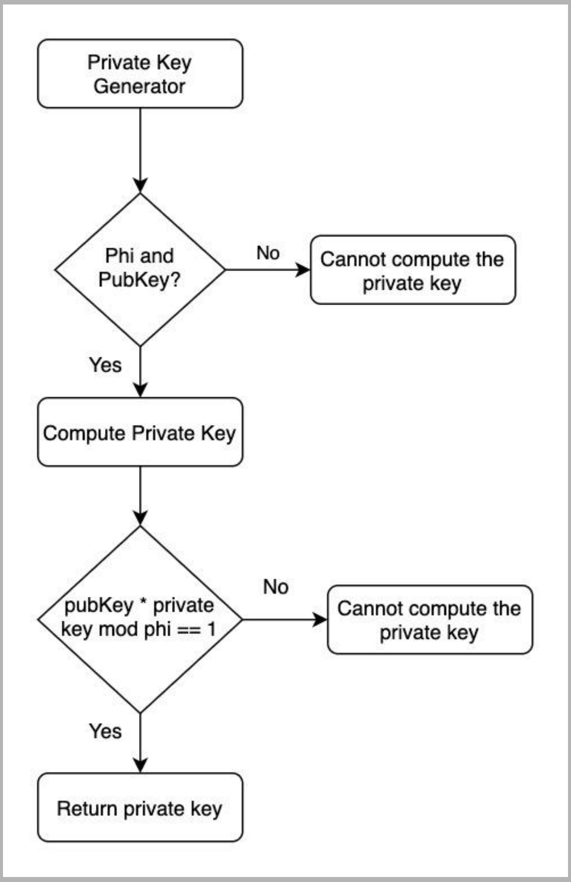
The class primeNumberGen handles the creation of the two prime numbers (p and q). The class constructor takes a number that will be used to generate a bit-length number (default is 20). The number is then halved to avoid overflow errors using python's bitwise right shift operation.

The main function of this class is the generatePrimeNumbers. The said function calls the helper function to generate a prime candidate. If this random number is prime, then return that number to the calling function generatePrimeNumbers which outputs two of such prime numbers. Otherwise, if the random number is not prime, generate another random number and try until this condition is true.


Private key Generator Class
Generate prime numbers
Return P and Q
Yes
Generating a prime number candidate
is Prime
Creating a random number of bitLength
No

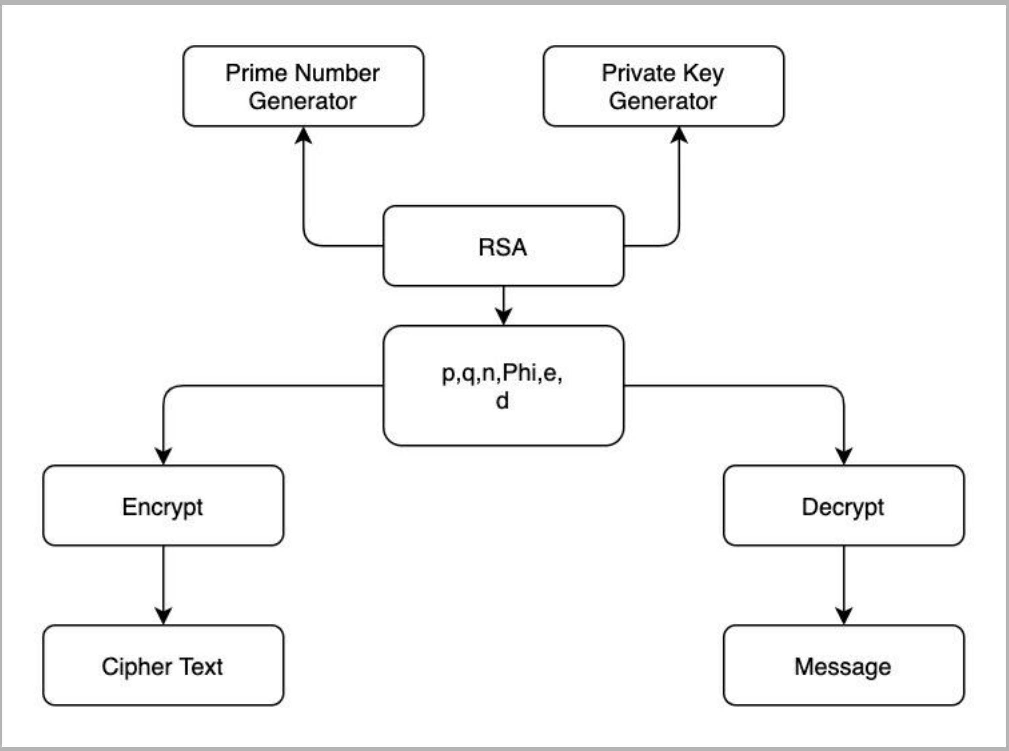
Generating the Private Key

This class requires phi(n) and the public key (e) to compute the private key (d). The private key generator function creates a 2x2 NumPy matrix and while the bottom left of this matrix is not 1, it performs a series of computations. When the bottom left value is 1, it returns the bottom right value, which is the private key.


Private Key Generator
Phi and PubKey?
No
Cannot compute the private key
Yes
Compute Private Key
pubKey * private key mod phi == 1
No
Cannot compute the private key
Yes
Return private key

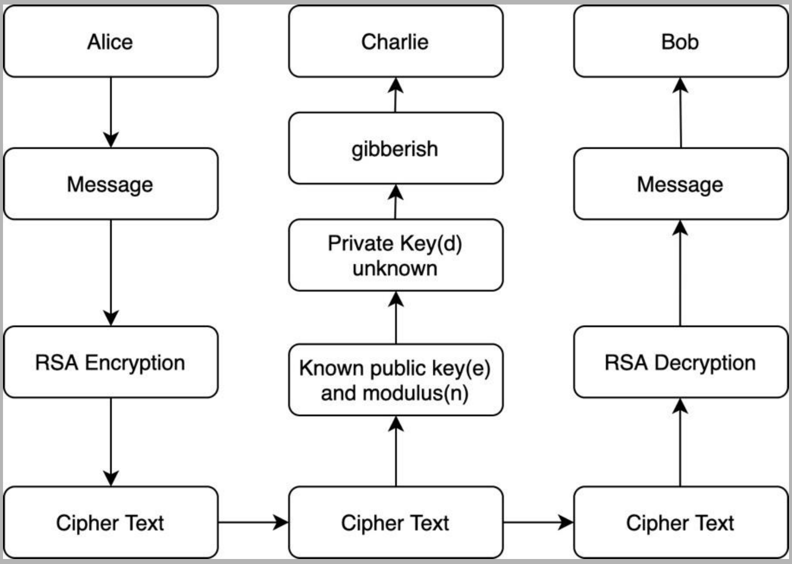
RSA

This is the driver class responsible for the entire application. Through inheritance, some of the methods are used to compute the necessary components for RSA. It only takes the number of bits of the encryption, although it is set to 20 by default.



Charlie : The Hacker

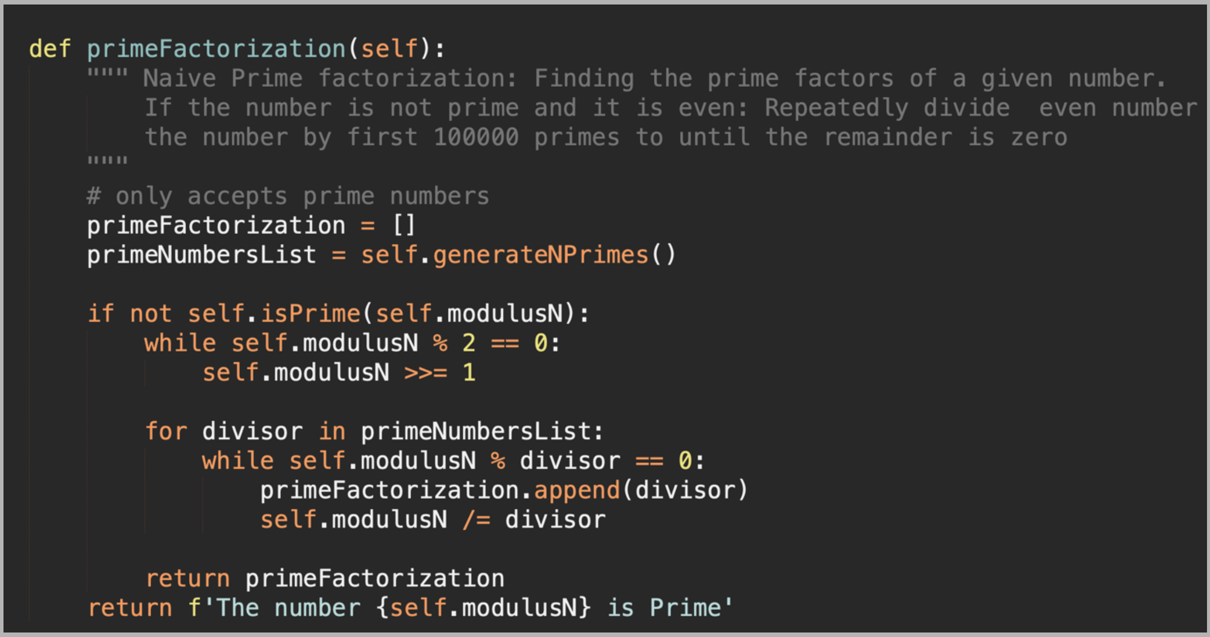
We must assume that there is always someone listening. However, without the private key, it is impossible for Charlie to decrypt the message.



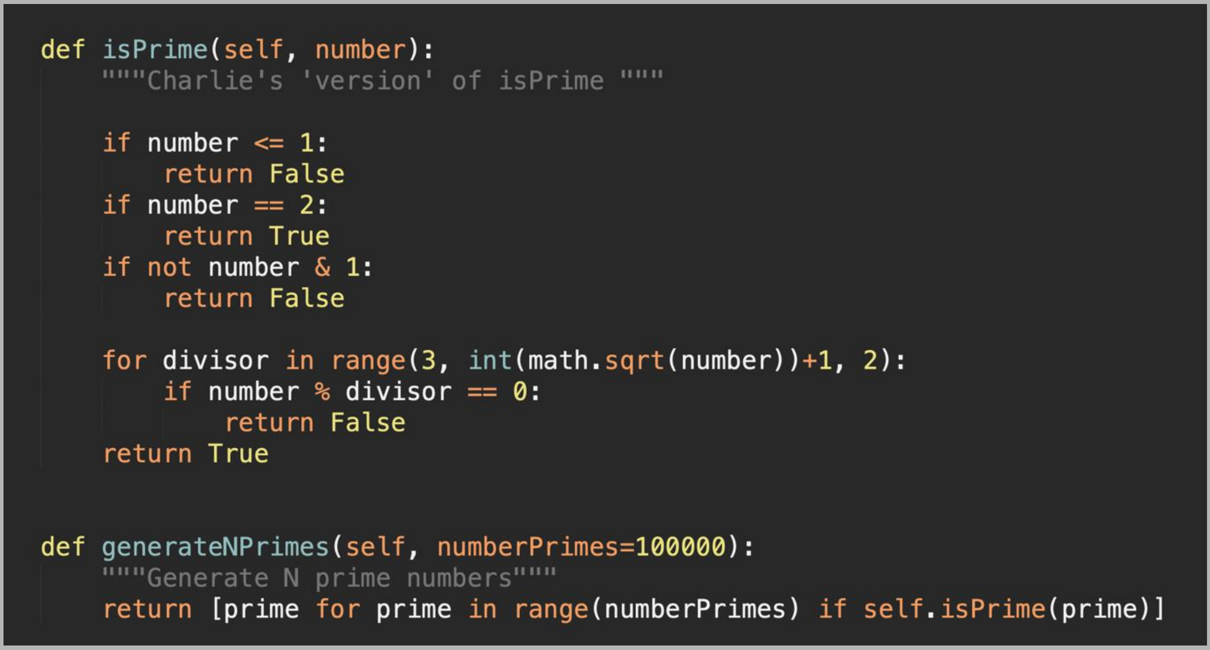
Suppose that Charlie intercepts the following encrypted message:

'3948,210,9161,23036,16186,9161,19516,3126,23036,18845,9161,210,
8302,9161,14362,3126,6469,8302,15852,18845,15425,9947'

And he knows that the public keys (e,n) = (65537, 2436829). Charlie can use prime factorization to find the factors of . That can be accomplished If a small number is used for the public key (n),

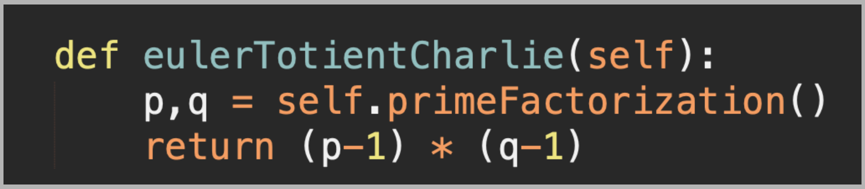


```python
def primeFactorization(self):
    """ Naive Prime factorization: Finding the prime factors of a given number.
        If the number is not prime and it is even: Repeatedly divide  even number
        the number by first 100000 primes to until the remainder is zero
    """
    # only accepts prime numbers
    primeFactorization = []
    primeNumbersList = self.generateNPrimes()

    if not self.isPrime(self.modulusN):
        while self.modulusN % 2 == 0:
            self.modulusN >>= 1

        for divisor in primeNumbersList:
            while self.modulusN % divisor == 0:
                primeFactorization.append(divisor)
                self.modulusN /= divisor

        return primeFactorization
    return f'The number {self.modulusN} is Prime'
```



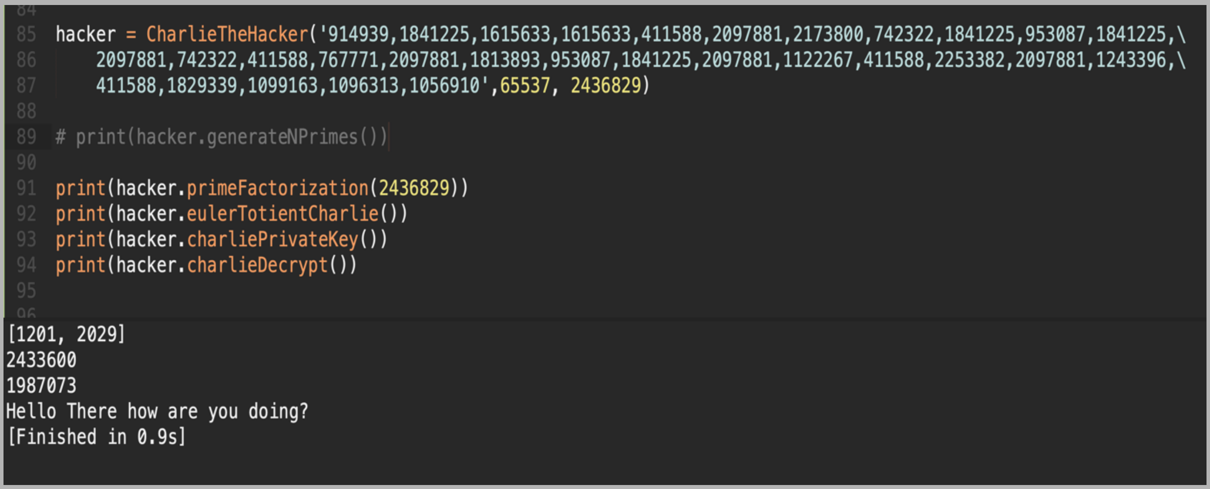
```python
def isPrime(self, number):
    """Charlie's 'version' of isPrime """

    if number <= 1:
        return False
    if number == 2:
        return True
    if not number & 1:
        return False

    for divisor in range(3, int(math.sqrt(number))+1, 2):
        if number % divisor == 0:
            return False
    return True


def generateNPrimes(self, numberPrimes=100000):
    """Generate N prime numbers"""
    return [prime for prime in range(numberPrimes) if self.isPrime(prime)]
```

Once he has the prime factors of N [1201, 2029], he needs to find the phi(n).



```
def eulerTotientCharlie(self):
    p,q = self.primeFactorization()
    return (p-1) * (q-1)
```

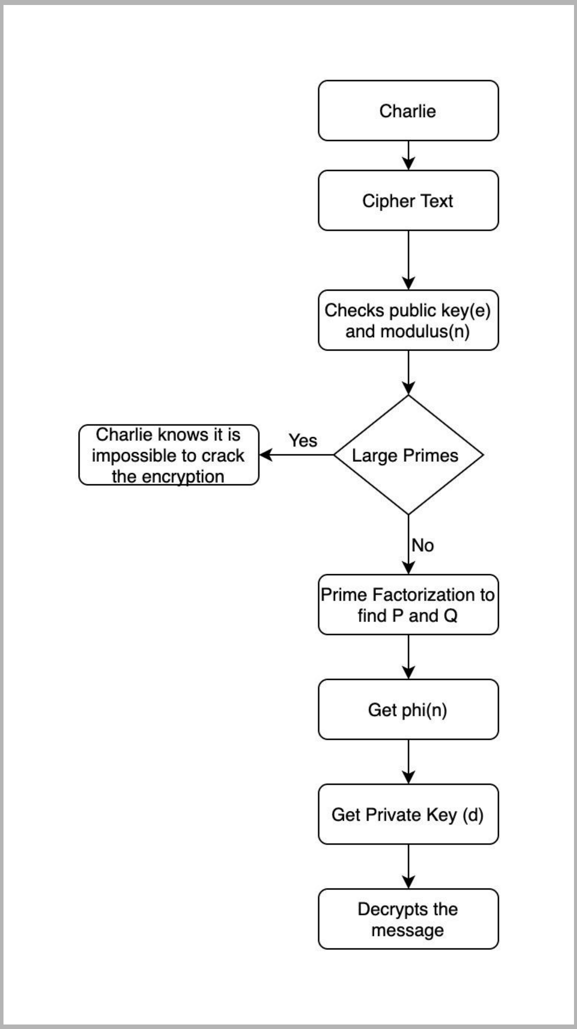
Once he finds that phi(n) is 2433600 he can find the private key and decrypt the message. For the sake of simplicity, the private key generator for Charlie uses the same private key generator as the RSA. With the private key in hand, he can decrypt the cipher text: c = m^d mod n



```
hacker = CharlieTheHacker('914939,1841225,1615633,1615633,411588,2097881,2173800,742322,1841225,953087,1841225,\
    2097881,742322,411588,767771,2097881,1813893,953087,1841225,2097881,1122267,411588,2253382,2097881,1243396,\
    411588,1829339,1099163,1096313,1056910',65537, 2436829)

# print(hacker.generateNPrimes())

print(hacker.primeFactorization(2436829))
print(hacker.eulerTotientCharlie())
print(hacker.charliePrivateKey())
print(hacker.charlieDecrypt())
```

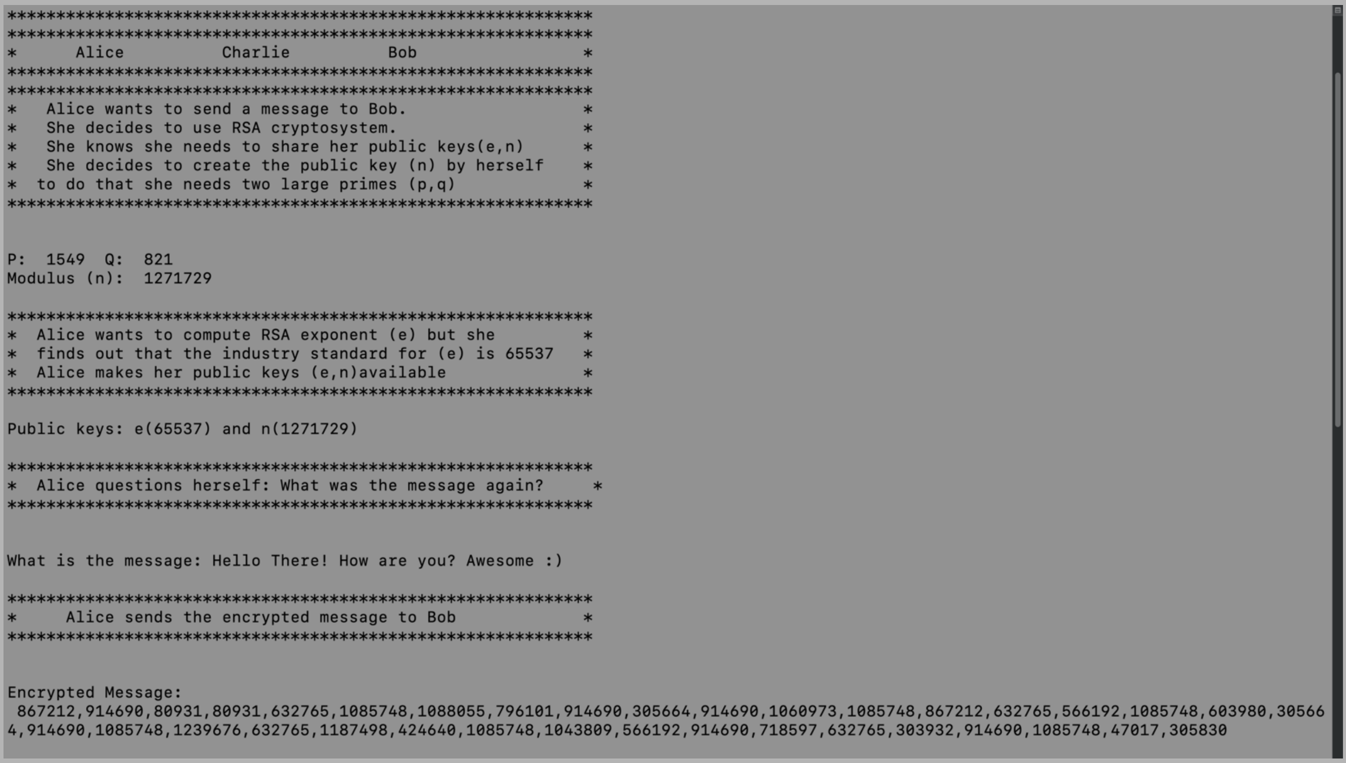
```
[1201, 2029]
2433600
1987073
Hello There how are you doing?
[Finished in 0.9s]
```

The following flowchart shows the flow of Charlie’s steps to break the cipher text:

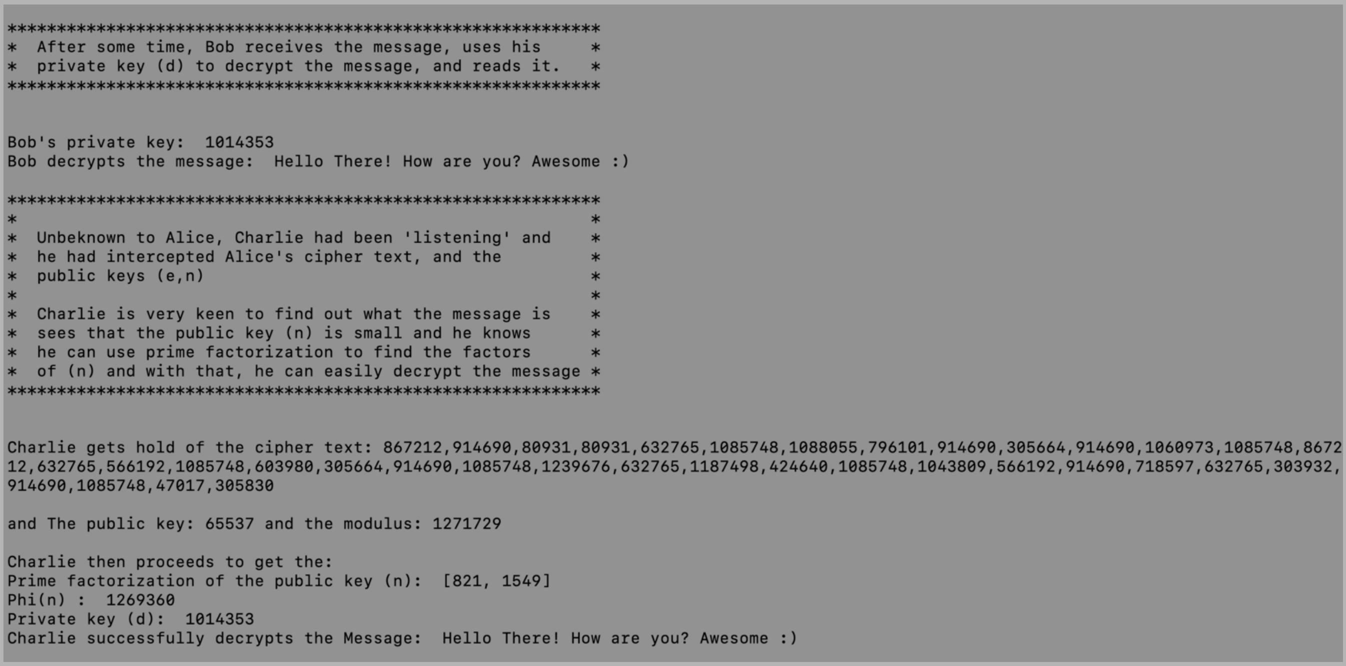

Charlie
Cipher Text
Checks public key(e) and modulus(n)
Large Primes
Yes
Charlie knows it is impossible to crack the encryption
No
Prime Factorization to find P and Q
Get phi(n)
Get Private Key (d)
Decrypts the message

Demonstration

The only file that needs to be run is called **Interface.py**. The user only needs to type the message, and the RSA algorithm is run automatically showing all the steps taken by Alice, Bob's private key and Charlie's hacking.



```
************************************************************
************************************************************
*      Alice          Charlie          Bob                 *
************************************************************
************************************************************
*   Alice wants to send a message to Bob.                  *
*   She decides to use RSA cryptosystem.                   *
*   She knows she needs to share her public keys(e,n)      *
*   She decides to create the public key (n) by herself    *
*  to do that she needs two large primes (p,q)             *
************************************************************


P:  1549  Q:  821
Modulus (n):  1271729

************************************************************
*  Alice wants to compute RSA exponent (e) but she         *
*  finds out that the industry standard for (e) is 65537   *
*  Alice makes her public keys (e,n)available              *
************************************************************

Public keys: e(65537) and n(1271729)

************************************************************
*  Alice questions herself: What was the message again?     *
************************************************************


What is the message: Hello There! How are you? Awesome :)

************************************************************
*     Alice sends the encrypted message to Bob             *
************************************************************


Encrypted Message:
 867212,914690,80931,80931,632765,1085748,1088055,796101,914690,305664,914690,1060973,1085748,867212,632765,566192,1085748,603980,30566
4,914690,1085748,1239676,632765,1187498,424640,1085748,1043809,566192,914690,718597,632765,303932,914690,1085748,47017,305830
```



```
*************************************************************
*  After some time, Bob receives the message, uses his     *
*  private key (d) to decrypt the message, and reads it.   *
*************************************************************


Bob's private key:  1014353
Bob decrypts the message:  Hello There! How are you? Awesome :)

*************************************************************
*                                                           *
*  Unbeknown to Alice, Charlie had been 'listening' and     *
*  he had intercepted Alice's cipher text, and the          *
*  public keys (e,n)                                        *
*                                                           *
*  Charlie is very keen to find out what the message is     *
*  sees that the public key (n) is small and he knows       *
*  he can use prime factorization to find the factors       *
*  of (n) and with that, he can easily decrypt the message *
*************************************************************


Charlie gets hold of the cipher text: 867212,914690,80931,80931,632765,1085748,1088055,796101,914690,305664,914690,1060973,1085748,8672
12,632765,566192,1085748,603980,305664,914690,1085748,1239676,632765,1187498,424640,1085748,1043809,566192,914690,718597,632765,303932,
914690,1085748,47017,305830

and The public key: 65537 and the modulus: 1271729

Charlie then proceeds to get the:
Prime factorization of the public key (n):  [821, 1549]
Phi(n) :  1269360
Private key (d):  1014353
Charlie successfully decrypts the Message:  Hello There! How are you? Awesome :)
```

Discussion

One problem that I had was when generating the two prime numbers, sometimes the method would return a None instead of a prime number. The solution to this problem was to ensure that the helper function would keep generating random odd numbers until it satisfied the condition that the number was prime.

Another issue was that sometimes the private key generator would raise as RuntimeWarning: divide by zero encountered in long_scalars. The first step I took to handle this problem was to enclose the code in a try catch block so the code would exit without crashing the application.

The second problem was that I would get an overflow error, so I set the NumPy array to 64 bits to handle large numbers. Furthermore, the bit length of the prime numbers needed to be roughly the same as the public key (e) otherwise, the decrypted message would simple not make any sense. To fix this, I put a default bit length of 20 bits, meaning about 10 bits per number. If the big length is either too small, less than 10 bits or too large, greater than 30 bits, an exception is raised.

Bibliography

Some of my work is inspired by the work of these creators.

Prime Factorization - https://paulrohan.medium.com/prime-factorization-of-a-number-in-python-and-why-we-check-upto-the-square-root-of-the-number-111de56541f

Finding the Private key - https://www.youtube.com/watch?v=Z8M2BTscoD4&ab_channel=AnthonyVance

Python bitwise Operations : https://wiki.python.org/moin/BitwiseOperators

Structure of the Code: https://dusty.phillips.codes/2018/09/13/an-intermediate-guide-to-rsa/