CSC110 Lecture 20: More on Cryptography

Hisbaan Noorani

October 28, 2020

Contents

1 Exercise 1: Implementing the RSA cryptosystem

Now that we've covered the theory behind the RSA cryptosystem, let's see how to implement it in Python! You'll do this in three steps: implementing a function that generates an RSA key pair, then a function to encrypt a number, and finally a function to decrypt a number. (Note that just like the RSA algorithm itself, the first function is the hardest. If you get stuck, skip to the encryption/decryption functions, which are simpler!)

```python
import random
import math
from typing import Tuple


def rsa_generate_key(p: int, q: int) -> \
    Tuple[Tuple[int, int, int], Tuple[int, int]]:
    """Return an RSA key pair generated using primes p and q.

    The return value is a tuple containing two tuples:
    1. The first tuple is the private key, containing (p, q, d).
    2. The second tuple is the public key, containing (n, e).

    Preconditions:
        - p and q are prime
        - p != q

    Hints:
        - If you choose a random number e between 2 and $\varphi(n)$, there isn't a guarantee
          that $gcd(e, \varphi(n)) = 1$. You can use the following pattern to keep picking
          random numbers until you get one that is coprime to $\varphi(n)$.

            e = ... # Pick an initial choice
            while math.gcd(e, ___) > 1:
                e = ... # Pick another random choice

        - You can re-use the functions we developed last week to compute the modular inverse.
    """
    n = p * q
    phi_n = (p - 1) * (q - 1)

```

```python
    e = 2

    while math.gcd(e, phi_n) != 1:
        e += 1

    d = modular_inverse(e, phi_n)

    return ((p, q, d), (n, e))


def rsa_encrypt(public_key: Tuple[int, int], plaintext: int) -> int:
    """Encrypt the given plaintext using the recipient's public key.

    Preconditions:
    - public_key is a valid RSA public key (n, e)
    - 0 < plaintext < public_key[0]
    """
    n = public_key[0]
    e = public_key[1]

    return plaintext ** e % n


def rsa_decrypt(private_key: Tuple[int, int, int]  ciphertext: int) -> int:
    """Decrypt the given ciphertext using the recipient's private key.

    Preconditions:
    - private_key is a valid RSA private key (p, q, d)
    - 0 < ciphertext < private_key[0] * private_key[1]
    """
    d = private_key[2]

    return ciphertext ** d % (p * q)
```