2CSDE93 - Blockchain Technology

Practical 1

Aim: To implement digital signature to sign and verify authenticated users. Also, show a message when tampering is detected.

Author: Darshil Maru 20BCE514

Date: August 23, 2022

Code:

```
import random
from hashlib import sha256


def coprime(a, b):
    while b != 0:
        a, b = b, a % b
    return a


def extended_gcd(aa, bb):
    lastremainder, remainder = abs(aa), abs(bb)
    x, lastx, y, lasty = 0, 1, 1, 0
    while remainder:
        lastremainder, (quotient, remainder) = remainder,
divmod(lastremainder, remainder)
        x, lastx = lastx - quotient*x, x
        y, lasty = lasty - quotient*y, y
    return lastremainder, lastx * (-1 if aa < 0 else 1), lasty * (-1 if bb
< 0 else 1)

#Euclid's extended algorithm for finding the multiplicative inverse of two
numbers
def modinv(a, m):
    g, x, y = extended_gcd(a, m)
    if g != 1:
        raise Exception('Modular inverse does not exist')
    return x % m


def is_prime(num):
    if num == 2:
        return True
    if num < 2 or num % 2 == 0:
        return False
    for n in range(3, int(num**0.5)+2, 2):
        if num % n == 0:
```

```python
            return False
    return True


def generate_keypair(p, q):
    if not (is_prime(p) and is_prime(q)):
        raise ValueError('Both numbers must be prime.')
    elif p == q:
        raise ValueError('p and q cannot be equal')

    n = p * q

    #Phi is the totient of n
    phi = (p-1) * (q-1)

    #Choose an integer e such that e and phi(n) are coprime
    e = random.randrange(1, phi)

    #Use Euclid's Algorithm to verify that e and phi(n) are comprime
    g = coprime(e, phi)

    while g != 1:
        e = random.randrange(1, phi)
        g = coprime(e, phi)

    #Use Extended Euclid's Algorithm to generate the private key
    d = modinv(e, phi)

    #Return public and private keypair
    #Public key is (e, n) and private key is (d, n)
    return ((e, n), (d, n))


def encrypt(privatek, plaintext):
    #Unpack the key into it's components
    key, n = privatek

    #Convert each letter in the plaintext to numbers based on the
character using a^b mod m
```

```
    numberRepr = [ord(char) for char in plaintext]
    print("Number representation before encryption: ", numberRepr)
    cipher = [pow(ord(char),key,n) for char in plaintext]

    #Return the array of bytes
    return cipher


def decrypt(publick, ciphertext):
    #Unpack the key into its components
    key, n = publick

    #Generate the plaintext based on the ciphertext and key using a^b mod
m
    numberRepr = [pow(char, key, n) for char in ciphertext]
    plain = [chr(pow(char, key, n)) for char in ciphertext]

    print("Decrypted number representation is: ", numberRepr)

    #Return the array of bytes as a string
    return ''.join(plain)


def hashFunction(message):
    hashed = sha256(message.encode("UTF-8")).hexdigest()
    return hashed


def verify(receivedHashed, message):
    ourHashed = hashFunction(message)
    if receivedHashed == ourHashed:
        print("Verification successful: ", )
        print(receivedHashed, " = ", ourHashed)
    else:

        print("Verification failed")
        print(receivedHashed, " != ", ourHashed)


def main():
```

```
    p = int(input("Enter a prime number (17, 19, 23, etc): "))
    q = int(input("Enter another prime number (Not one you entered above):
"))
    #p = 17
    #q=23


    print("Generating your public/private keypairs now . . .")
    public, private = generate_keypair(p, q)

    print("Your public key is ", public ," and your private key is ",
private)
    message = input("Enter a message to encrypt with your private key: ")
    print("")

    hashed = hashFunction(message)

    print("Encrypting message with private key ", private ," . . .")
    encrypted_msg = encrypt(private, hashed)
    print("Your encrypted hashed message is: ")
    print(''.join(map(lambda x: str(x), encrypted_msg)))
    #print(encrypted_msg)

    print("")
    print("Decrypting message with public key ", public ," . . .")

    decrypted_msg = decrypt(public, encrypted_msg)
    print("Your decrypted message is:")
    print(decrypted_msg)

    print("")
    print("Verification process . . .")
    verify(decrypted_msg, message)

main()
```

Output:

```
PS E:\Python OOP> python -u "e:\7Sem\BCT\Practical1\PR1.py"
Enter a prime number (17, 19, 23, etc): 11
Enter another prime number (Not one you entered above): 17
Generating your public/private keypairs now . . .
Your public key is  (1, 187)  and your private key is  (1, 187)
Enter a message to encrypt with your private key: abcd
```

```
Encrypting message with private key  (1, 187)  . . .
Number representation before encryption:  [56, 56, 100, 52, 50, 54, 54, 102, 100, 52, 101, 54, 51, 51, 56, 100, 49, 51, 98, 56, 52, 53, 102, 99, 102, 50, 56, 57, 53, 55, 57, 100
, 50, 48, 57, 99, 56, 57, 55, 56, 50, 51, 98, 57, 50, 49, 55, 100, 97, 51, 101, 49, 54, 49, 57, 51, 54, 102, 48, 51, 49, 53, 56, 57]
Your encrypted hashed message is:
56561005250545410210052101545151561004951985652531029910250565753555710050485799565755565051985750495510097511014954495751541024851495356 57

Decrypting message with public key  (1, 187)  . . .
Decrypted number representation is:  [56, 56, 100, 52, 50, 54, 54, 102, 100, 52, 101, 54, 51, 51, 56, 100, 49, 51, 98, 56, 52, 53, 102, 99, 102, 50, 56, 57, 53, 55, 57, 100, 50,
 48, 57, 99, 56, 57, 55, 56, 50, 51, 98, 57, 50, 49, 55, 100, 97, 51, 101, 49, 54, 49, 57, 51, 54, 102, 48, 51, 49, 53, 56, 57]
Your decrypted message is:
88d4266fd4e6338d13b845fcf289579d209c897823b9217da3e161936f031589

Verification process . . .
Verification successful:
88d4266fd4e6338d13b845fcf289579d209c897823b9217da3e161936f031589  =  88d4266fd4e6338d13b845fcf289579d209c897823b9217da3e161936f031589
PS E:\Python OOP>
```