Tasnimul Hasnat

190041113

CSE-4743

December 31, 2023

RSA Cryptosystem

Overview

The RSA Cryptosystem is a widely used public-key encryption system that relies on the mathematical properties of large prime numbers. This implementation of the RSA Cryptosystem is encapsulated in a Python class called `RSA`, which provides methods for key generation, encryption, and decryption. The system allows two parties, Alice and Bob, to securely communicate by generating public and private keys.

Implementation Details

Helper Classes

Message

The `Message` class within the RSA implementation handles the conversion between plaintext and ciphertext. It utilizes the `struct.pack()` and `struct.unpack()` methods to convert strings to integers and vice versa.

```
def convert2Int(plaintext: str) -> int:
        return int.from_bytes(pack('B' * len(plaintext), *map(ord,
plaintext)),'big')
def convertFromInt(ciphertext: int) -> str:
        byte_length = (ciphertext.bit_length() + 7) // 8
        return "".join(map(chr,unpack('B' *
byte_length,ciphertext.to_bytes(byte_length, 'big'))))
```

> `struct.pact(format="BB...",ord('h'),ord('i'),...)` returns byte string. Here `B` means unsigned char.

> `struct.unpack(format="BB...",byte_string)` returns a tuple. In this case a tuple of integers.

> Here byte_length is computed by padding the last byte and flooring the value after division by 8.

Core RSA Implementation

The core `RSA` class is responsible for key generation, encryption, and decryption.

- **Prime Generation**: The constructor (`__init__`) initializes the RSA object by generating two large prime numbers (`p` and `q`), ensuring they are not the same. Also the generated primes are checked using ***miller-rabin test***.

```
def __generatePrime(bitSize: int) -> int:
        p = random.getrandbits(bitSize)
        # if even make odd
        if not p % 2:
                p += 1

        # loop until probable prime is found
        while not RSA.__miller_rabin_test(p): # primality testing
                p += 2
        return p
```

- **Key Generation**: Calculates the public and private key pairs based on the prime numbers generated before.

```
def __generateKeys(self) -> Tuple[Tuple[int, int], Tuple[int, int]]:
        # public key e such that, e<phi and co-prime to phi
        e = random.randint(2, self.__phi - 1)
        while gcd(e, self.__phi) != 1:
                e = random.randint(2, self.__phi - 1)

        # private key d such that, d is the modular inverse of e % phi.
```

```python
        d = pow(e, -1, self.__phi)
        return (e, self.__n), (d, self.__n)
```

- **Encryption and Decryption**: The `encrypt` and `decrypt` methods use the public and private keys to perform encryption and decryption operations, respectively. The `Message` class is used for converting between string and integer representations.

```python
def encrypt(self, plaintext: str, key: Tuple[int, int]) -> int:
        e, n = key
        return pow(RSA.Message.convert2Int(plaintext), e, n)
```

```python
def decrypt(self, ciphertext: int) -> str:
        d, n = self.__privateKey
        return self.Message.convertFromInt(pow(ciphertext, d, n))
```

Testing

The `test` function tests the correctness of the RSA implementation by generating random texts and checking various encryption and decryption scenarios. It includes tests for Alice and Bob communicating with each other and ensures that an eavesdropper, Eve, cannot decrypt the messages.

It checks for the following assertions,

```python
# Alice sends OK
(alice.decrypt(alice.encrypt(text, alice.publicKey)) == text) == True
(bob.decrypt(alice.encrypt(text, bob.publicKey)) == text) == True

# Bob sends OK
(bob.decrypt(bob.encrypt(text, bob.publicKey)) == text) == True
(alice.decrypt(bob.encrypt(text, alice.publicKey)) == text) == True

# Alice and Bob dont send OK
(bob.decrypt(alice.encrypt(text, alice.publicKey)) == text) == False
(alice.decrypt(bob.encrypt(text, bob.publicKey)) == text) == False

# eve cant eavesdrop
(eve.decrypt(alice.encrypt(text, alice.publicKey)) == text) == False
(eve.decrypt(alice.encrypt(text, bob.publicKey)) == text) == False
```

```
(eve.decrypt(bob.encrypt(text, alice.publicKey)) == text) == False
(eve.decrypt(bob.encrypt(text, bob.publicKey)) == text) == False
```

With random texts for a thousand times,

```
def gen_random_text():
        # length can only be upto keysize/8 bytes.
        length = random.randint(1, 2**6)
        text = ''.join(random.choice(string.ascii_letters + string.digits +
                string.punctuation + string.whitespace) for _ in range(length))
        return (text, length)
```

Through my testing with different key sizes with any ascii text as plaintext, no errors were found.

Execution Script

The execution script allows users to interact with the RSA cryptosystem. Users can choose to be either Alice or Bob, and they can perform the following actions:

1. **Encrypt a Message**: Users can encrypt a message with the public key of the other party.
2. **Decrypt a Message**: Users can decrypt a ciphertext with their own private key.
3. **Print Private Key**: Users can print their private key.
4. **Print Public Key**: Users can print their public key.
5. **Test for Errors**: Users can run a set of automated tests to ensure the correctness of the implementation.
6. **Exit**: Users can exit the program.

For details about the execution script, check out the other usage instructions pdf.

Conclusion

The provided RSA cryptosystem implementation demonstrates the fundamental concepts of public-key cryptography. Users can securely exchange messages by leveraging the generated public and private keys. The testing mechanism ensures the correctness of the implementation in different scenarios.