Project 4 - RSA cryptosystem

Network Security - Spring 2024

Due Date: May 04, 2025

1 Introduction

RSA (Rivest-Shamir-Adleman) is a public-key cryptosystem that is widely used for secure data transmission. It is based on the mathematical properties of large prime numbers and modular arithmetic. The security of RSA relies on the difficulty of factoring the product of two large prime numbers, which makes it computationally infeasible to derive the private key from the public key. RSA is used for various purposes, including secure communication, digital signatures, and key exchange. It is commonly used in protocols such as SSL/TLS for securing web traffic, as well as in email encryption and digital certificates. The goal of this project is to implement the RSA algorithm. The theoretical foundations for the algorithm have been covered in classes.

RSA involves using integers whose bit length can be in hundreds or thousands. However, since Python supports arbitrary precision integers, we can use the built-in `int` type to represent large integers, without using any external library. However, note that it is never a good idea to roll out your own cryptographic library, and you should always use a well-tested library for cryptographic operations. The goal of this project is to implement the RSA algorithm, and not to create a production-ready library.

2 Getting Started

The project requires a working installation of Python 3.8 or later, on your local machine. No VM is required for this project.

- Test your Python version by running the following command:

```
$ python3 --version
```

- A test suite is provided alongside the starting files. In order to be able to run tests, you must install `pytest` first. This can be done by running the following command (assuming `pip` is installed on your system):

```
$ pip install pytest
```

- Once installed, you can run the test suite by running the following command (inside the directory containing the `tests.py` file):

```
$ pytest tests.py
```

  The above command runs all the tests available. In order to run tests for a specific function only (e.g. `generate_private_key`), append `::test_<function_name>` to the above command:

```
$ pytest tests.py::test_<function_name>
So, to run the tests for generate_private_key:
$ pytest tests.py::test_generate_private_key
```

3 Project Tasks

The skeleton code for the implementation of the `RSA` class is provided in the file `rsa.py`. You are supposed to make changes to this file only. Do not modify the interfaces (or method definitions) as the testing mechanism relies on that. Only modify the functions that you have been asked to.

3.1 Task 1: Deriving the public key

Given three prime numbers p, q, e, derive the public key (n, e) where $n = p.q$. Specifically, provide the implementation for the following function:

`def generate_public_key(self) -> Tuple[int, int]:`

The function returns the tuple (n, e).

3.2 Task 2: Deriving the private key

Given three prime numbers p, q, e, derive the private key d. Recall that the private key d satisifes the property: $e.d = 1 \mod \phi(n)$, where $\phi(n) = (p-1).(q-1)$.

Specifically, provide the implementation for the following function:

`def generate_private_key(self) -> int:`

The function returns the private key d.

3.3 Task 3: Message encryption

Given a plaintext integer M, and a public key (n, e), the ciphertext C is obtained by $C = M^e \mod n$. Note that the messages will be encrypted using the public key, and decrypted using the private key.

Provide an implementation for the following method:

`def encrypt(self, plaintext: str) -> int:`

Note that the expected plaintext is a string, but the RSA algorithm expects an integer. For this purpose, a class method `_encode_message(self, message: str) -> int` is provided that takes in a string of characters, and returns an integer encoding of the text.

3.4 Task 4: Message decryption

Given a ciphertext C, and the public-private key pair (n, e) and d, the plaintext M is derived by $M = C^d \mod n$. This returns an integer, which must be decoded back to a string of characters to get the plaintext back.

Provide the implementation for the following function:

`def decrypt(self, ciphertext: int) -> str:`

The function returns the decoded plaintext message.

3.5 Task 5: Message signatures and verification

RSA can be used to sign messages so that the authenticity of the messages can be verified. This can be done by using the **Private Key** to encrypt the message (instead of the Public key). The ciphertext obtained is called the signature and can be attached along with the message. The receiver can decrypt the signature using the sender's **Public Key**, and if the decrypted signature matches the message, it can be proven that the message was indeed signed by the sender's private key (in this case, only the sender has access to the private key). Note that this is not used in practice this way, instead message hashes are computed, and then signed using the algorithm.

Provide the implementation of the following two functions:

```
def sign(self, message: str) -> int:
def verify_signature(self, message: str, signature: int) -> bool:
```

4 Submission

You are only required to submit the `rsa.py` file with your implementation in it. Rename the file as `s_<your_rollnumber>.py` and submit it via LMS.