CSE350: Programming Assignment 3

Ananya Lohani (2019018), Mihir Chaturvedi (2019061)

RSA

RSA (Rivest–Shamir–Adleman) is a public-key cryptosystem used for secure data and communication transmission. Ron Rivest, Adi Shamir, and Leonard Adleman developed it in 1977.

RSA functions by generating two keys: a public key and a private key. Anyone who wants to send a message to the owner of the private key may freely distribute the public key. The private key is kept secret and is used by the owner to decrypt messages that have been received.

The RSA algorithm encrypts a message by raising it to the power of the public key modulo a very large number. This is the result, which can only be decrypted using the private key.

RSA's security is predicated on the fact that it is exceedingly challenging to factor large numbers into their prime factors. The RSA algorithm presupposes that factoring the product of two large prime numbers is computationally impossible. Even if an attacker intercepts the ciphertext, they cannot easily determine the original message without knowing the private key.

RSA is widely used for secure communication over the internet, such as email encryption, secure web browsing, and secure file transfer. It is one of the most secure encryption algorithms, but it requires large key sizes to maintain security, which can be computationally expensive.

Our Implementation

Our RSA Implementation consists of the following functions:

- **`hash`**: Calculates the SHA-256 hash of the input message.
- **`is_prime`**: Checks whether a number is prime.
- **`generate_key_pair`**: Generates a public-private key pair by first generating two prime numbers and then calculating the values of n, `phi_n`, e, and d. It returns the public key and the private key.
- **`encrypt`**: Encrypts a message using the public key by first hashing the message and then dividing it into blocks of a fixed size. It pads each block with additional bytes as needed, then encrypts each block using the public key and returns the ciphertext in a JSON format.
- **`decrypt`**: Decrypts a ciphertext using the private key by first decoding the JSON-formatted input and then decrypting each block using the private key. It then removes any padding added to the original message, joins the decrypted blocks together into the original message, and verifies the hash of the decrypted message.
- **`pad`**: Adds padding to a message to ensure that it can be divided into blocks of a fixed size.
- **`unpad`**: Removes any padding added to a message by the pad function.

- **`get_num_bytes`**: Calculates the number of bytes needed to represent an integer.
- **`gcd`**: Calculates the greatest common divisor of two integers.
- **`modinv`**: Calculates the modular inverse of a number using the extended Euclidean algorithm.

Public Key Distribution Authority

A Public Key Distribution Authority (PKDA) is an entity responsible for the secure and trusted distribution of public keys to users or devices. The primary function of a PKDA is to verify the identity of users or devices that request public keys and to ensure the authenticity and integrity of the keys they receive.

Typically, a PKDA employs a combination of digital certificates, cryptographic protocols, and trusted third parties to validate the authenticity and integrity of public keys. A PKDA may, for instance, issue digital certificates containing details about the public key, its owner, and the PKDA's digital signature. Recipients can use these certificates to validate the authenticity and integrity of the public key.

gRPC

We selected gRPC for this assignment because it is a high-performance, open-source, universal RPC (Remote Procedure Call) framework that is built on top of HTTP/2, a high-performance binary protocol for sending and receiving data. **Protocol Buffers** is the default serialization format for gRPC. Protocol Buffers provides a technique for serializing structured data that is language-independent, platform-independent, and extendable.

gRPC uses **sockets** to establish and manage connections between the client and the server. When a gRPC client submits a request to the server, it establishes a socket connection with the server and transmits the request through that socket. The server then receives and processes the request using this socket. When sending the response back to the client, the server uses the same socket that the client used to send the request.

PKDA and Client Implementation

The PKDA is implemented as a gRPC service using the PKDA`Servicer` class. Upon initialization, it generates a 1024-bit RSA key pair, and stores the public key and a dictionary of registered clients' information. **Clients generate their own RSA private-public key pair.** Clients can register with the PKDA by sending a `RegisterClientRequest` message containing their ID, network address, and public key. If the client is not already registered, the PKDA stores their information in its dictionary and sends back a `RegisterClientResponse` message containing the PKDA's public key.

Clients communicate with each other via the `ClientServicer` class, which also uses gRPC. Upon initialization, the client generates a 256-bit RSA key pair and initializes various instance variables. To communicate with other clients, it first registers with the PKDA by sending a `RegisterClientRequest` message, and then sends a `PublicKeyRequest` message to retrieve another client's public key from the

PKDA. If the client is not already stored in the client's keystore, the client retrieves the public key from the PKDA and stores it in its keystore.

When a client receives a message from another client, it first decrypts the message using its private key. If it is in the process of establishing a connection with the other client, it will perform a nonce exchange to confirm the connection. Otherwise, it will simply display the decrypted message on the console. The client can also send messages to other clients by encrypting the message using the recipient's public key and sending it via gRPC.