CSC4026 NETWORK AND INTERNETWORK SECURITY PRACTICAL 2024

PROJECT TEAM:

Name	Campus ID	Email
Matthew Craig	CRGMAT002	CRGMAT002@myuct.ac.za
Kane Gibson	GBSKAN001	GBSKAN001@myuct.ac.za
Jordy Kafwe	KFWJOR001	KFWJOR001@myuct.ac.za

CRYPTOSYSTEM DESIGN

Our cryptosystem supports two-way communication, allowing a client to both send and receive messages. Although our model operates on a client-server framework, to aid in understanding, the initial illustration in figure 1 will disregard the server and assume that clients send messages directly to each other. In this illustration, Alice is the sender, and Bob is the recipient.

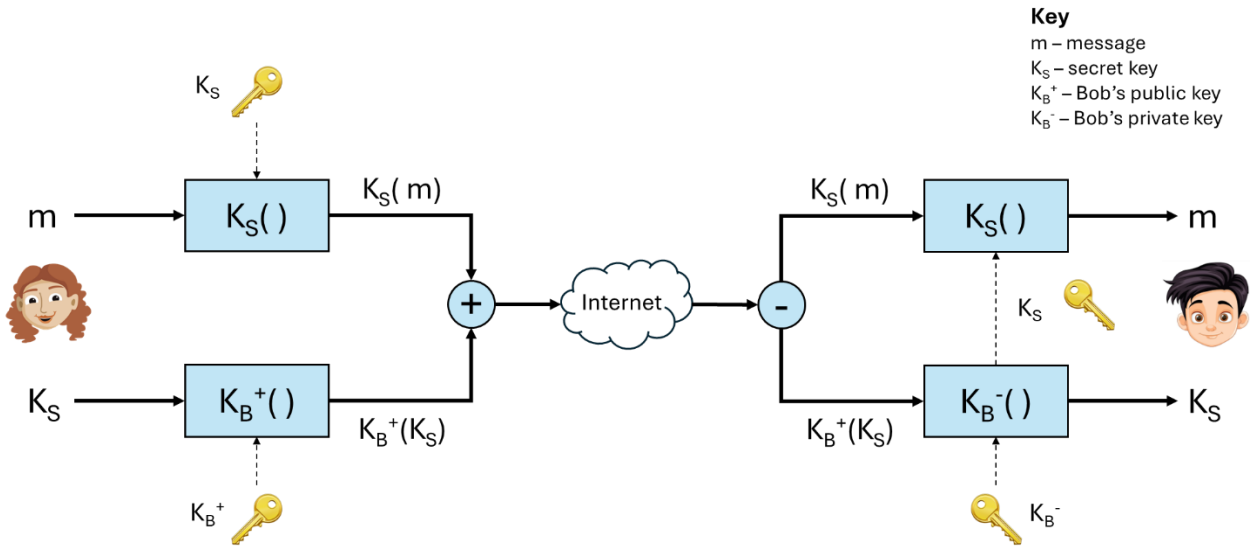

Figure 1: Cryptosystem design, simplified by disregarding the server.

Alice creates a message, *m*, to send to Bob and encrypts it with a newly created symmetric secret key, K_s. To enable Bob to decrypt the message, Alice also needs to send him the secret key. However, the secret key must remain confidential. This is where asymmetric key encryption is used. Alice encrypts the secret key using Bob's public key K_B^+ and sends both the encrypted message and encrypted secret key to Bob.

Upon receiving the message, Bob decrypts the secret key using his private key, K_B^-. With the decrypted secret key, he can then decrypt the message.

COMMUNICATION CONNECTIVITY MODEL

Our communication connectivity model operates on a client-server framework. Messages are transmitted via a server, facilitating communication between clients indirectly. Interactions between servers and clients occur through the exchange of messages.

Although Figure 1 simplified the cryptosystem design by omitting the server, the actual cryptosystem includes a server. The message sent from the server to Bob is identical to the one Alice sent directly to Bob in Figure 1. However, the message Alice sends to Bob is different, as illustrated in Figure 3 below.

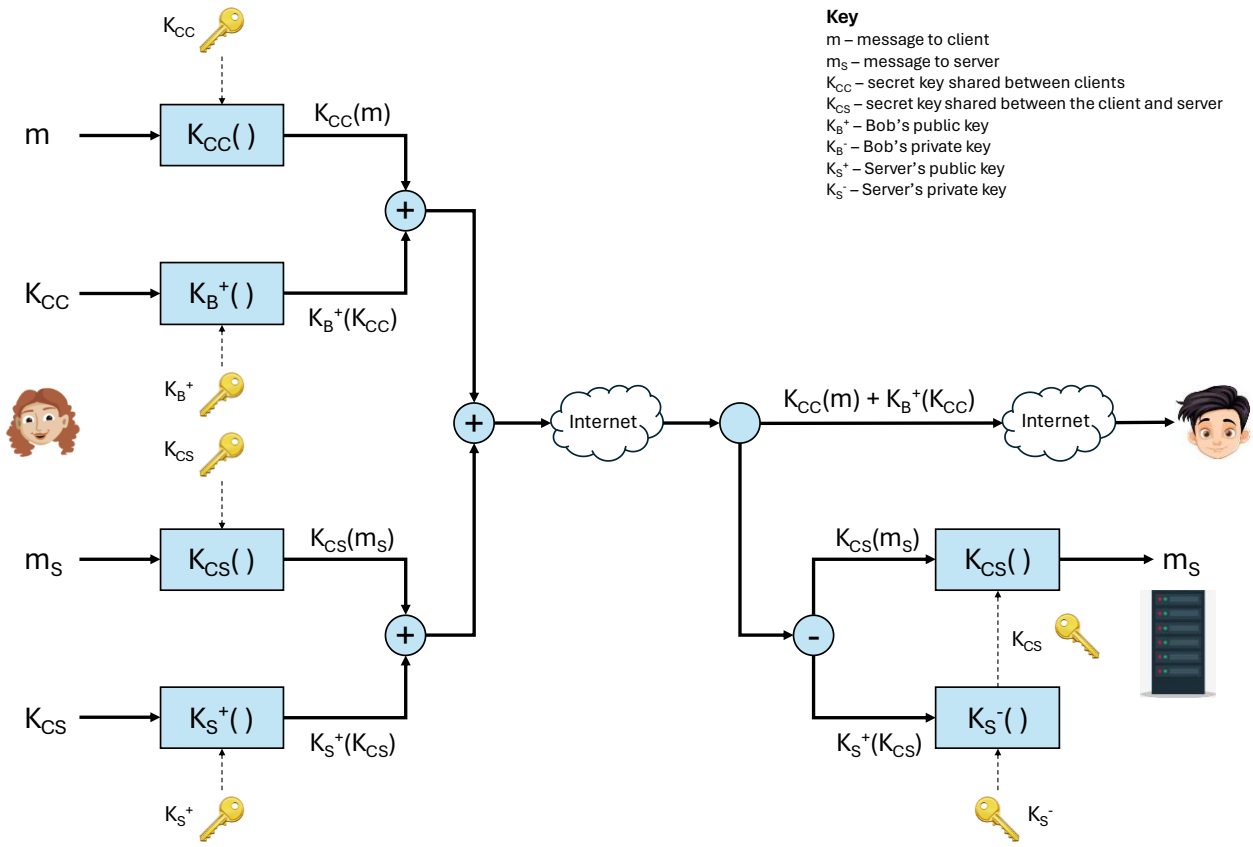

Figure 2: Cryptosystem design including the server.

Alice performs the same steps as in Figure 1 to prepare the message for Bob. However, she also needs to send additional data, m_S, specifically for the server. This data consists of the recipient's username, which the server uses to determine the message's destination. Alice encrypts this data similarly to how she encrypts the message for Bob, but she uses a new symmetric secret key, K_{CS}, and encrypts this key with the server's public key, K_S^+. Both the encrypted data and the encrypted key are sent to the server. The server decrypts the recipient's username to identify the intended recipient and then forwards the encrypted message to Bob without altering it in any way.

Having explained the cryptosystem design involving the server and the interaction between the server and clients for sending and receiving messages, we will now discuss the initial connection (involving the certificate authority server) and additional details regarding client and server features and functionalities.

Certificate Authority Server & Initial Connection

We make use of a Certificate Authority (CA) server for issuing and managing digital certificates. These certificates are used to establish secure communications. CA is always running and ready to accept requests from servers and clients alike.

After generating their own asymmetric key pair, requesting a certificate is the first step for both the client and server during initial registration. This certificate binds them as the authentic owners of their public and private keys.

The purpose of certification becomes clear during the initial connection process between the client and server. The client connects to the server who sends a random number encrypted with the client's public key and signed with its certificate. The client decrypts the message and sends the signature to the CA server for validation. If valid, the server is authentic. The client then sends the same random number back to the server, encrypted with the server's public key. If this number (decrypted with the server's private key) matches the original, the server can confidently verify the client's authenticity.

Client

Features

Image folder

All a client's images are stored in their "images" folder. To send an image, it must be stored within this folder. All images that are received from other clients are also stored in this folder.

Keys folder

A "key" folder contains a subfolder named after the user. For example, in figure 3, "Alice" is the current user. Each user has two keys stored within this folder: their own public and private keys, labeled "public" and "private" respectively. Additionally, the public key of each contact that the user has is stored here. In figure 3, one of Alice's contacts is Bob, so Alice has stored Bob's public key labeled "bob". In other words, this folder can be seen as Alice's public keyring, along with her own public and private keys.

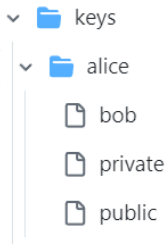

Figure 3: An example of Alice's "keys" folder.

Functionalities

Once a client establishes a connection with a server, they gain access to and are presented with a variety of available actions to choose from. Some secondary functionalities include listing one's contacts, listing and displaying the user's images in their default image viewing application, and closing the app in a non-destructive way. Sending and receiving messages, and requesting certificates are primary functionalities.

Send Image (s)

The user types "s" in the command line to start the "send image" process. First, if an image is available in the user's local "images" directory, the user is first prompted to enter the recipient's name. Next, the user selects the image to send by specifying the file path. Finally, the user is prompted to enter a caption for the image. This information is used to create the message *m* of figure 3. It is formatted as follows.

Figure 4: Sending message format (before server data prepended).

The first 2 bytes are reserved for the caption length, followed by 1 byte for the sender's username length. Next, the image caption, sender's username, and image data are appended. However, this is not yet the final message sent to the server.

First, the recipient's username is encrypted using the secret key K_{CS} of figure 3. The length of this encrypted username is calculated and prepended to the message, followed by the encrypted username and the data from figure 4, referred to as "Message Data" in figure 5. The final data sent to the server is formatted as shown below.

Figure 5: Final sending message format.

Receive Messages

The client does not explicitly request to receive messages. Instead, they automatically receive messages from the server whenever they come online or while they are online. The entire message is decrypted using the client's private key. This decrypted message follows the format shown in figure 4. The client reads the first 2 bytes to determine the caption length and the 3rd byte to determine the sender's username length. Using this information, the caption, sender's username, and image data are extracted. The image data is saved as a file named with the message date, sender's name, and image caption. This file naming format is represented in code as follows:

```
file_name = f"{date_time}--{sender}--{caption}"
```

Request Certificate (c)

Typing "c" into the command line prompts the client CLI app to ask for a recipient's username. The client then connects to the certificate authority to request that user's certificate. Once received, the client CLI app stores the certificate locally.

Server

Features

Online Users

The server stores a list of all users currently online in a dictionary object named “online”. The key-value pair is made up of the client username who is online (key) and their corresponding socket object (value).

Message Queue

If a client sends a message but the recipient is not currently online, the server will need to store this message until that intended recipient comes online. Then only will the message be sent to the recipient. The server stores these queued messages in a dictionary variable named “send_queue”. The key is the recipient’s username, while the value is the actual message data to be sent to the recipient.

Keys Folder

Like client, the server has a folder "key" that contains a subfolder named “server.” This folder stores the server’s public and private key, labeled "public" and "private" respectively. However, unlike the clients, the server does not store client public keys as it simply forwards the already-encrypted message.

Functionalities

Secondary functionalities include listing the online users and viewing queued messages, which simply entails printing out the dictionaries mentioned previously that hold this information. The main server functionalities include receiving and sending (forwarding) messages.

Receiving messages

The server receives messages from a client in the format shown in figure 5. It uses the length of the recipient's username to extract and separate the username from the actual message. By decrypting the recipient's username with its own private key, the server identifies the recipient and then adds the message to the recipient's send queue. This queue holds all messages waiting to be delivered to the user once they come online.

Sending messages

Messages can only be sent to a client when they are online. When the server receives a message, it checks if the recipient is online. If they are, the message is sent immediately. If not, the message is stored as previously discussed. When a client comes online, the server forwards any messages belonging to them.

KEY MANAGEMENT

Our cryptosystem emulates the Pretty Good Privacy (PGP) framework by using both symmetric and asymmetric key encryption. Therefore, it makes use of three keys in total: a public, private key and shared session key.

Advanced Encryption Standard (AES) is used for symmetric encryption. The secret key is generated using a cryptographic pseudo-random number generator (os.urandom). The AES key and secret are erased from memory after the session ends.

RSA is used for asymmetric encryption. Each party, server and client alike, has their own public and private key pair that they generate and store themselves. The public key is openly shared and used for encryption, while the private key is kept secret and used for decryption. A party's public key is also stored by the CA when a party applies for a certificate. The central CA facilitates distribution of public keys. That is, parties use the certificate authority to retrieve another party's certificate to gain access to their public key.

For the key exchange, both parties need to securely share the session key to encrypt and decrypt messages. This is achieved through asymmetric encryption, eliminating the need for a manual exchange. The session key is encrypted with the recipient's public key and securely transmitted, ensuring that only the recipient can decrypt it using their private key.

CHOICE OF CRYPTOGRAPHIC ALGORITHMS

Our choices of cryptographic algorithms prioritize security and efficiency.

AES was selected for its speed and efficiency, great for handling large image data with minimal overhead. Implementing AES with key lengths of 256 bits, like we did, provides high-level security as it protects

against known attacks. This functionality was implemented using the AES component of the pyca/cryptography library [1].

RSA was chosen because it is a mature and widely used public key encryption standard. RSA allows us to sign and verify the authenticity of data effectively to ensure high-level integrity and security. Its strong encryption is resistant to many forms of cryptographic attacks.

System Design And Creativity

Our PGP cryptosystem is structured as a series of classes, ensuring that each component performs a single task, and does it effectively. This design has facilitated bug detection in our code, thereby enhancing the security and functionality of our system. Moreover, we have incorporated Python's type annotations throughout our code, which has minimized type-related bugs and fostered confidence when working with each other's code, thereby ensuring a robust system.

We have used common OOP design patterns, such as a singleton for the certificate authority (CA), to ensure that only one central CA can exist at any given time. We also use an SQLite database for data persistence, enabling our servers to recover from outages without data loss. For instance, the CA server stores all the certificates in the SQLite database.

Our servers are multi-threaded, enabling them to handle multiple connections concurrently and effectively. We have also established separate CA and mail sending servers. This design reduces latency since each server performs a single task. For example, the mail server focuses solely on forwarding mail to users, ensuring it is not slowed down by certificate generation. Furthermore, clients store all certificates they receive, allowing them to continue exchanging messages even during CA server downtime, provided the mail server is available. This feature, which enables clients to store and validate certificates independently, enhances the reliability of our system.

As a creative feature, our app allows users to open received images in their default image viewer through input to the CLI app.

Testing Procedures and Assumptions

Our main testing approach was end-to-end testing. Once all our functionality was coded, we ran our cryptosystem through real-life use it could typically experience. This included normal functionality like generating keys, getting certificates, and sending and receiving messages. Our end-to-end testing also focused on edge cases that, if not handled well, could crash our system. Therefore, this testing shows that our app works in normal cases and can also handle abnormal inputs. Evidence of this was recorded in video. We also have some unit tests for essential functionality to ensure that those specific units work. The readme explains how to run these tests.

Instructions on how to execute/run the submitted program(s)

Please find README with instructions in the project root directory.

REFERENCES

[1] Test vectors (no date) *Test vectors - Cryptography 43.0.0.dev1 documentation*. Available at: https://cryptography.io/en/latest/development/test-vectors/ (Accessed: 17 May 2024).