University of Cape Town

CSC4026Z

Network and Internetwork Security

Practical Report

Cohen, Jaron
CHNJAR003@myuct.ac.za
CHNJAR003

Combrinck, Carl
CMBCAR007@myuct.ac.za
CMBCAR007

Green, Bailey
GRNBAI001@myuct.ac.za
GRNBAI001

June 7, 2022

1 Communication

1.1 Overview

`SecureGroupChat` is an end-to-end secure messaging application, facilitating group communication for connected clients. Several key security services are implemented via a PGP-like cryptosystem.

The application employs a client-server architecture to exchange messages, certificates, and commands. The application server is trusted to provide the basic services of broadcasting certificates and forwarding messages to the intended recipients. This trust does not extend to any of the implemented security services. In this way, the server mediates client-to-client message exchanges.

Clients communicate as both senders and receivers in the application. After performing an initial certificate exchange, clients communicate with the rest of the group via the server by sending separately-encoded messages to the server for distribution to the corresponding recipient clients.

1.2 Protocol

The `CertificateAuthority` (CA) is initially executed and generates the key pair used to sign and verify certificates. The resulting keys are placed in a password-protected, file-based key store for future retrieval, when clients create their certificates with the CA.

The `Server` is then run to facilitate client communication. The server maintains a collection of open sockets with connected client applications.

When a `Client` application is run, the following protocol stages are followed:

1. **Certificate Generation**
   The client first generates its certificate through the CA (an instance of the CA is created which retrieves the previously-generated keys). The certificate follows the `X.509` format and importantly associates the client's alias (identifier) with its public key and is verified through a CA-generated signature.
2. **Certificate/Key Exchange**
   The client will establish a connection (via a TCP socket) to the application server before forwarding its certificate to the server. The server broadcasts the received certificate to all connected clients, each of whom verifies the incoming certificate (using the CA's public key to decrypt the certificate signature and compare the result to the hashed certificate). This ensures that only certificates signed using the CA's private key (and hence must have been generated by the CA) are trusted. After verification, clients store the certificate in their key store, and finally return a copy of their certificate to the sending client, via the server. At this point, all the other clients have the certificate of the newly-added client and have sent their certificates to this client. It is important to note that the certificate exchange does not need to be encrypted as the certificate signature ensures integrity (via the hash) and authenticity of the subject-key pairing in the certificate (the presence of a valid signature means the subject-key association originated from the CA). Confidentiality is not a concern as certificates are publicly available.

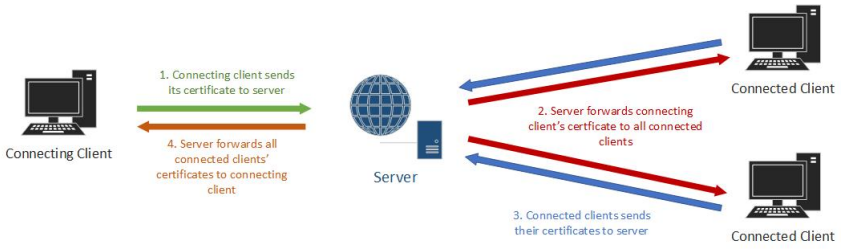

Figure 1: Key exchange between clients and server

3. **Messaging** Once a client has received confirmation from the server that its certificate has been broadcast to the group, it is allowed to begin sending messages (with the guarantee that any message it sends will reach clients after the delivery of its certificate, allowing receiving clients to decode the message). Messages are duplicated and individually PGP-encoded using each of the public key's stored in the client's key store and forwarded, via

the server, to the corresponding recipient (ensuring end-to-end encryption). Receiving clients then decode the message using their private key information and the public key of the sending client, stored in the receiver's key store.

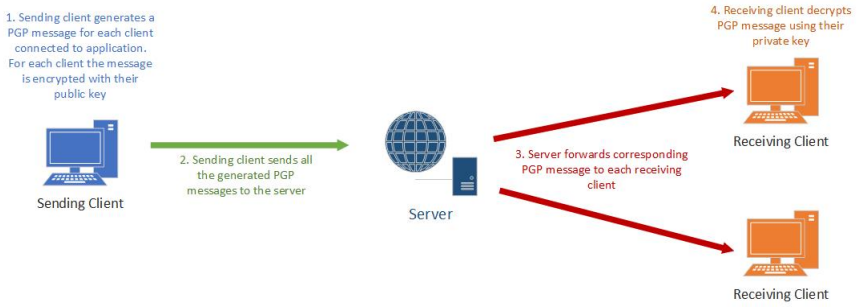

Figure 2: Message exchange between clients and server

4. **Quitting** In order to leave the group, a client enters "`quit`", resulting in a PGP protocol message being individually PGP-encoded and sent, via the server, to all the connected clients. Subsequently, the client instance, and its socket connection to the server, are terminated. The server then marks the connection as inactive and removes the corresponding client handler thread. Each client, after receiving and decrypting the PGP protocol message signalling the associated client's disconnection, will remove the disconnected client's certificate information from its respective key store (to prevent unnecessary transmission of messages to this client in future).

2 Security

2.1 End-to-End Encryption using PGP

Messages are exchanged using end-to-end encryption. This is achieved in the following way, using a combination of symmetric and asymmetric encryption in a PGP encode/decode pipeline:

1. The sender computes a signature by hashing the raw message and encrypting this with its private key.
2. The raw message and message signature are compressed using ZIP compression.
3. The sender generates a 256 bit secret key and 128 bit initialization vector (iv) for use in AES encryption (CBC mode).
4. The compressed message and signature are encrypted symmetrically with AES using the generated secret key and iv (session data).
5. The session data are encrypted with the receiver's public key using RSA encryption in ECB mode.
6. The encrypted, compressed message and signature, and encrypted session data together constitute a PGP message.
7. The PGP message is base 64 encoded and sent to the receiving client (this encoding produces ASCII output which is more human-readable and compatible with a wider range of applications than binary output).
8. Upon receiving the PGP message, the receiving client base 64 decodes the message and can decrypt the session data using their private key.
9. The session key and iv are then used to decrypt the compressed message and signature.
10. After the payload is decompressed, the raw message is retrieved and its signature is verified by decrypting the signature using the sender's public key and comparing this to the hashed raw message.

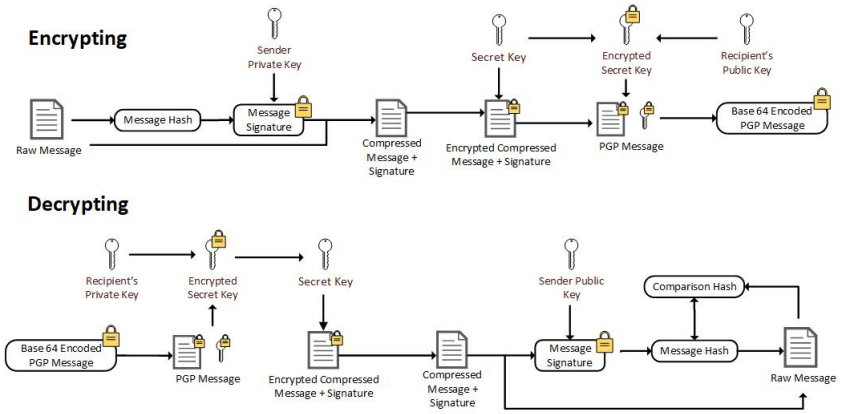

Figure 3: PGP encryption/decryption pipeline

2.2 Message Integrity

To ensure no messages are modified or corrupted during message exchange, a hashing function (SHA-256) is used. This one-way function transforms a variable-sized input message into a fixed-size string of 256 bits (32 bytes) in a way that minimises the chance of two strings hashing to the same value. The resulting hash, known as the message digest, is encrypted with the sender's private key to form the message signature.

Upon receiving the message, the receiving client, using RSA, decrypts the message signature using the sender's public key to obtain the message digest. The raw message that was received along with this signature is then hashed using the same SHA-256 algorithm. The resulting hash is compared to the received message digest. If the two hashes differ, it indicates that the message has been corrupted or modified after the signing of the message as even the slightest of modifications would have produced a vastly different hash (an integral property of a hash function such as SHA-256). Otherwise, a match indicates the message was received as sent, without any modification, thus ensuring integrity.

2.3 Message Authentication

Authentication is achieved via the use of message signatures, produced using RSA encryption and supported by a trusted certificate authority. Clients each obtain CA-signed certificates that associate/bind a client's identity to their public key. The authenticity of this association is guaranteed by the certificate signature, as any party is able to verify whether a given certificate (and hence subject-key association) originated from the trusted CA by verifying the certificate signature using the CA's public key (proving the signature was produced using the corresponding private key only known to the CA).

All messages are accompanied by a signature to verify authenticity. As described in the PGP pipeline 2.1, the raw message is hashed and encrypted with the sender's private key. This allows any recipient of the message to verify that the message originated from the claimed author, by decrypting this signature with the sender's public key (recall the association of this key and the sender's identity is guaranteed by the trusted CA), and comparing the result with the hash of the raw message. A match here means that the signature must have originally been produced using the private key corresponding to the public key associated with the sender. Since, in theory, the sender should be the only party possessing such a key, this verifies that the message must have originated from the sender associated with the public key in the certificate. Importantly, this signature is not transferable as it depends on the message content.

2.4 Message Confidentiality

All application messages are encrypted with a one-time (session) AES key and iv (hence only parties with this session key and iv can view the contents of the message). In order to ensure that only the intended recipient has access to this session data (and hence the message itself), it is encrypted, by the sender, using the public key of the recipient. As discussed, this ensures that only the recipient possessing the private key corresponding to this public key can access the session data to decrypt the message. The certificates, again, ensure that the correct public key is used in this process. Therefore, only parties possessing the private key (in theory only the recipient associated with the public key by the certificate) can view the message contents, providing confidentiality.

3 System Design

3.1 Compression

The message and message signature are compressed using ZIP compression before being encrypted. The ZIP compression implementation takes an array of bytes and makes use of the `java.util.zip.DeflaterOutputStream` class to compress the incoming raw bytes using a stream filter. For decompression, the `InflaterOutputStream` is used, similarly, to decompress the incoming compressed bytes, yielding the original message bytes. This has two major advantages which will be discussed in 3.3, namely, reduced payload size (for more efficient transmission) and reduced redundancy (increasing the difficulty of cryptanalysis).

3.2 Shared Key

A shared key, known as the secret key, is generated by the sending client each time a message is sent. We use the `javax.crypto.KeyGenerator` class to generate a 256 bit AES key. Additionally we use the `java.security.SecureRandom` class to generate a 128 bit initialization vector (iv). The secret key and initialization vector are used to encrypt the compressed message and message signature using AES encryption in cipher block chaining (CBC) mode.

The secret key and iv are encrypted using the receiver's public key using RSA encryption in electronic code book (ECB) mode and sent as part of the PGP message. This ensures that the secret key can only be decrypted using the receiver's private key, which should only be known to the receiver. The use of a different, random key for each message improves security as a compromised key will only allow the decryption of the associated message, providing no information on the keys used to encrypt other messages. As will be discussed in 3.3, the use of a shared key with symmetric encryption also has efficiency advantages, as asymmetric RSA encryption uses much larger keys and is computationally expensive.

3.3 Order Justification

Compression and encryption of messages follow the standard PGP ordering (see the full sequence in 2.1).

A message signature is first generated and concatenated with the message before being compressed to provide authenticity. This ensures that the generated hash does not depend on the underlying compression algorithm. The signature uses a hash function to reduce output size (this is more efficient - in terms of space and time - than signing the entire message). The compressed message and signature are then encrypted for confidentiality. The placement of compression between the signature generation and the encryption of the payload is further justified by the performance benefits of encrypting smaller payloads. Encryption is, relatively speaking, a fairly expensive operation from a computational perspective. Therefore, performing encryption on a compressed payload is more efficient than doing so on an uncompressed payload. This compression will also offset the possible expansion of the PGP message during radix 64 encoding (an optional step used for converting the binary output to ASCII characters for a number of applications such as email, but which increases the message size by 33%). Encryption is performed last as the compression of the data also reduces redundancy, making cryptanalysis more difficult, and therefore strengthening the security of the encrypted message [1]. Symmetric encryption (specifically AES) is used as this is much more efficient than performing asymmetric encryption using RSA on potentially large payloads. While RSA uses much larger keys and offers a high level of security, a sufficient degree of security is obtained using AES with smaller, ephemeral keys with improved performance.

Lastly, the session data used to encrypt the message is encrypted using asymmetric encryption (RSA) to ensure confidentiality (the receiving client must be the only other party to have access to this). This is reasonable given that the session data has a small, fixed length. In this way, the message contents are encrypted securely (providing the necessary security services) and in a performant manner. The encrypted session data is sent with the encrypted message payload so the receiving client can decrypt the message.

4 Testing

4.1 Unit Tests

The unit tests described below are executed automatically by Gradle when building/compiling the application with `gradlew build` (hence in order to run the application, it must first have passed the suite of tests after compilation).

Unit Test	Passed
testCompression	Passed
testRSA	Passed
testAES	Passed
testConcatenation	Passed
testBase64	Passed
testPGP	Passed

Unit Test	Passed
testMessageSignatureCorruption	Passed
testUntrustedCertificate	Passed
testIncorrectKeyPair	Passed
testPGPMessageCorruption	Passed

Table 1: Unit test suite

They can be found in the `test/com/securegroupchat/AppTest.java` file. The representative input used during these tests contains a wide range of typical characters that may be found in application messages.

4.1.1 PGP Tests

To comprehensively test the entire PGP encode and decode pipeline, unit tests corresponding to each stage of the pipeline were developed. These test the correct functioning of the implemented ZIP compression algorithm, RSA encryption/decryption, AES encryption/decryption, signature generation and verification, byte concatenation, radix 64 encoding/decoding, and finally, the full PGP encode/decode pipeline. This is achieved by invoking the corresponding `PGPUtilities` method followed by its inverse (where applicable) and checking for equality of the result and the input, for each PGP operation.

4.1.2 Tampering Tests

In order to ensure the application is secure from any tampering of data, unit tests were also developed in which data was intentionally tampered with. Testing was done to ensure that if the message was modified after the message signature was generated, the hash generated by the receiving client would be different to that of the message signature. Another test simulates the case where a certificate was generated and signed by a different certificate authority and ensures that an exception is thrown when an attempt to verify the signature using the trusted certificate authority's public key is made. The third test ensured an exception was thrown when the session data was decrypted using a different key pair to that with which it was encrypted (simulating the case where the receiving client is not the intended recipient). The final test was to make sure any corruption of the final PGP message that is sent is correctly detected (simulating the tampering of the message in transit between clients).

4.2 Empirical Testing

4.2.1 Different Systems

The application was thoroughly tested on Windows, Mac and Linux systems. This was done to ensure the application ran without errors and performed as expected on all three operating systems. All README requirements were observed and instructions followed for each operating system to ensure ease of installation and use.

4.2.2 Debugging Statements

Various debugging statements are used throughout the application for monitoring to ensure it is functioning as intended. Specifically, the client application provides two modes, one for normal use, and one for debugging. In debugging mode, the client application logs all incoming and outgoing transmissions (messages, commands, certificates etc.) and also logs the PGP encode pipeline (if it is sending the message) and the PGP decode pipeline (if it is receiving the message). In the latter case, each of the encrypted components and their decrypted counterparts is logged and can be compared with those of the sender to ensure correct transmission of the message and decryption of its contents. In particular, the decrypted hash from the message signature and the hash computed by the receiving client are logged for visual comparison to ensure that signature verification is working as intended (as these results should be equal, verifying both authenticity and integrity of the message).

All transmissions sent through the server are logged by default (such is the purpose of the server and hence no debugging mode is provided as full transmission logging fulfils this purpose). This includes the forwarding of PGP messages, exchanging of certificates and issuing of command messages. This allows for the protocol implemented (and described in 1.2) to be checked by verifying that for particular stages of the protocol (e.g. certificate exchange) the correct messages are being generated and sent through the server.

References

[1] Simson Garfinkel et al. *PGP: pretty good privacy.* " O'Reilly Media, Inc.", 1995.

Appendix A UML Class Diagram

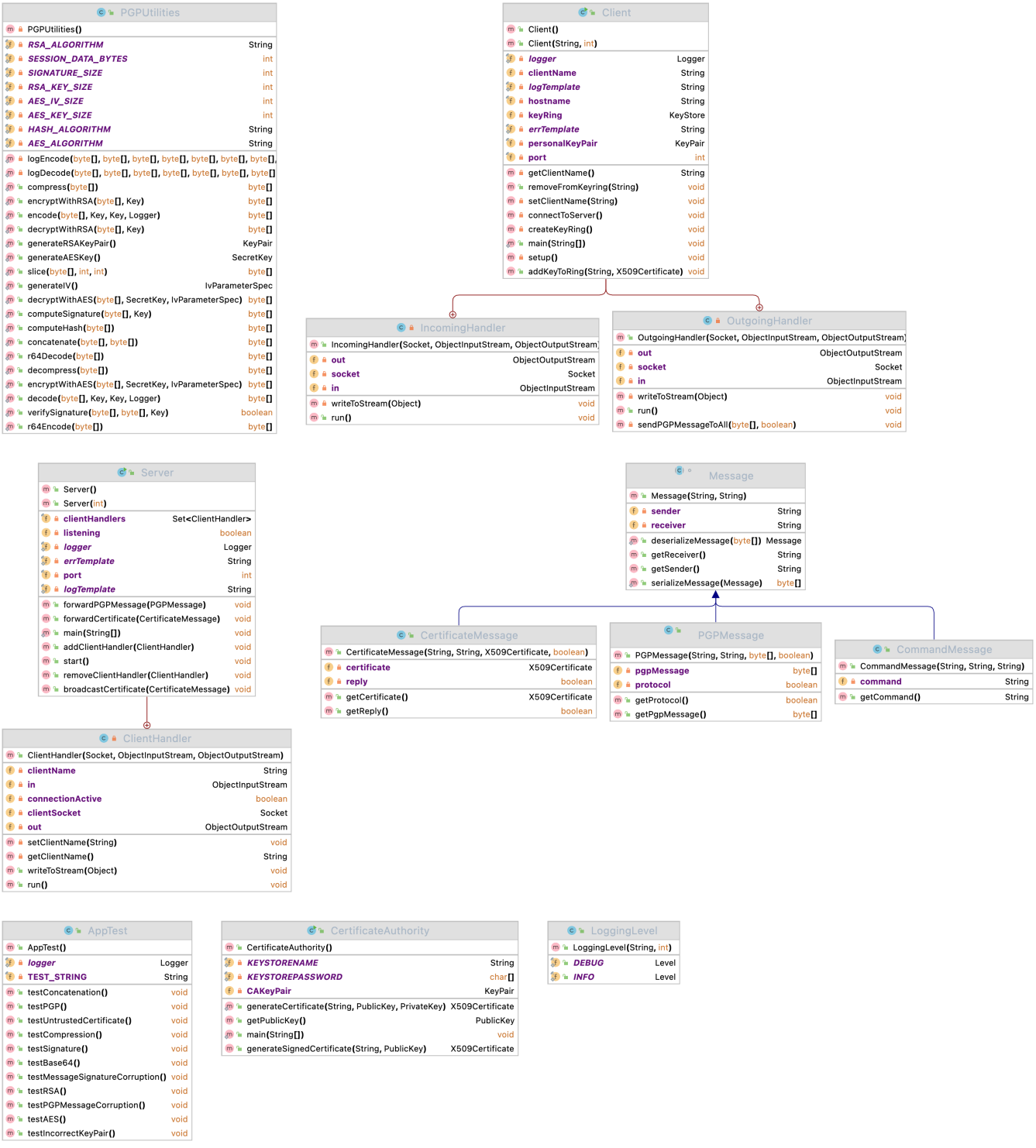

Figure 4: UML Class Diagram of SecureGroupChat Application

Appendix B Screenshots of Application

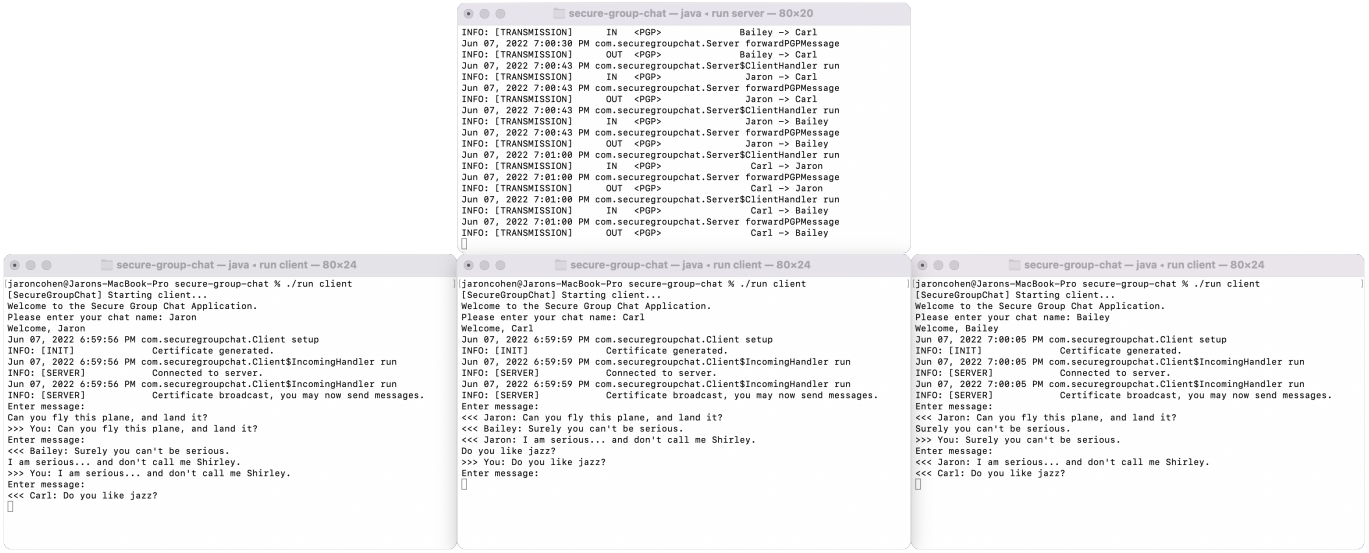

Figure 5: Three clients exchanging messages in non-debug mode

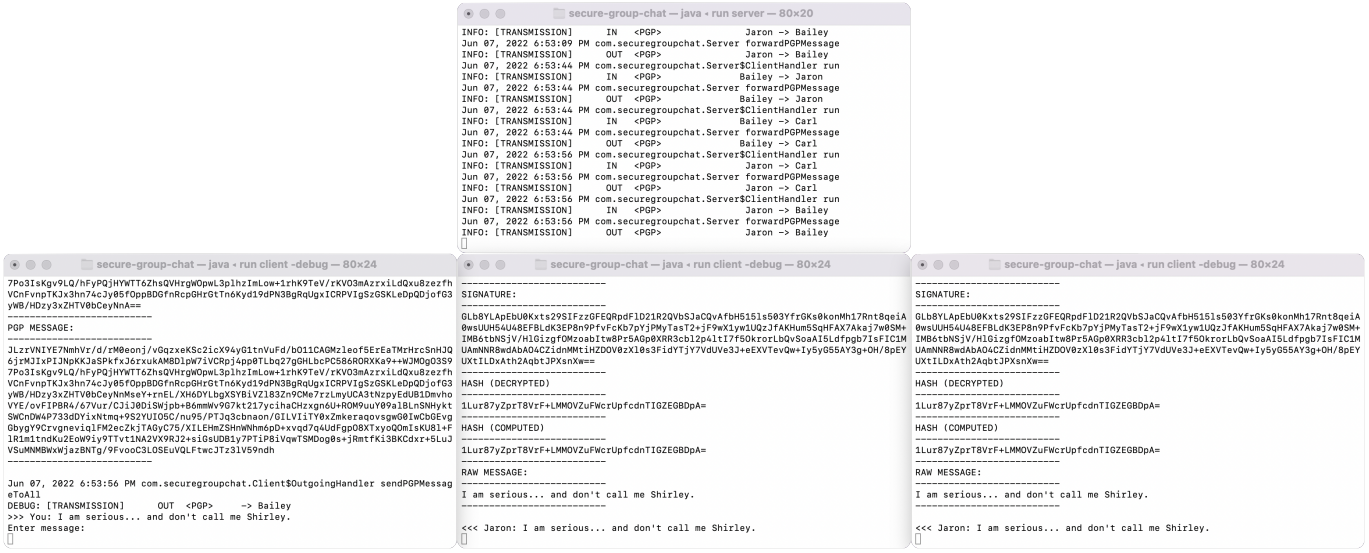

Figure 6: Three clients exchanging messages in debug mode