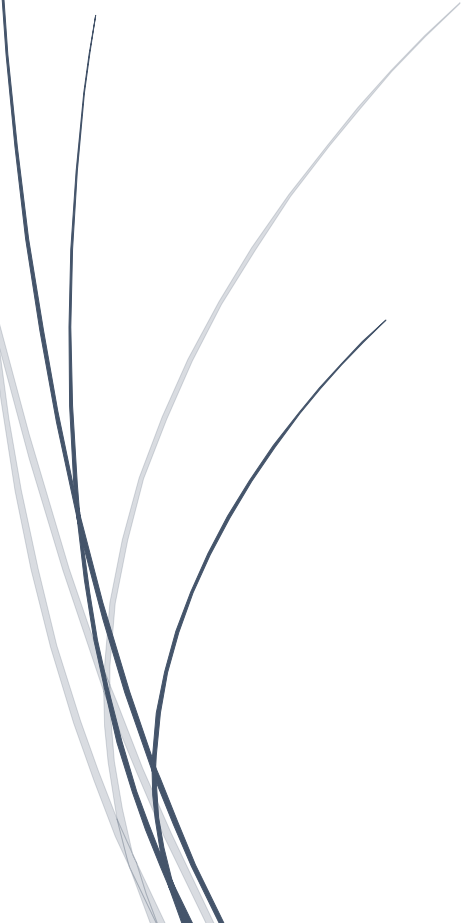


A dark blue vertical bar runs down the left side of the page. A blue arrow points to the right from the bar, containing the text "[Date]".

[Date]

Network & Internet Security

Practical

Several thin, curved lines in dark blue and light grey originate from the bottom left and sweep upwards and to the right.

Sihle Calana
CSC4026Z

The storage and transfer of information between two parties has always needed to be protected from unwanted third parties before the time of computers and networks. The steps taken to secure this information involved using trusted third parties to send information such as a post office, using shared locker facilities where the two parties have keys to access the locker, wax sealing envelopes to ensure no one has opened it, writing information in code (e.g., morse or enigma) or using invisible ink that can only be seen with UV light.

With the data now being stored machines and transferred over networks, more technology based actions need to be taken to ensure confidentiality, message integrity and availability. Before going further, we define each of those terms, so we have a clear understanding going forward.

Confidentiality ensures privacy between two parties such that only the authorised parties can access and *understand* the data.

Integrity ensures that the data has not been *altered* while in transit. Provides certainty that the message received is the message that was sent.

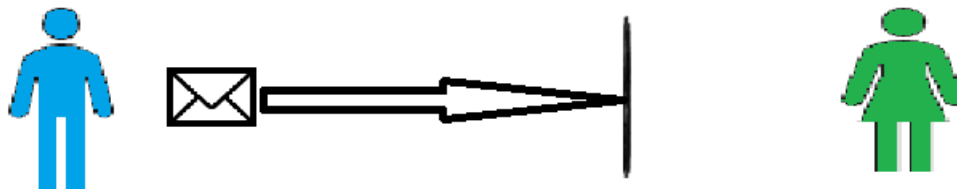
Availability ensures that the right people have *access* to information and no service is denied to the authorised entities.

Authenticity verifies the *identity* of an entity and that they are trustworthy.

Communication between two parties over a network can fall victim to actions that compromise the security of information being sent. These actions include the following threats:

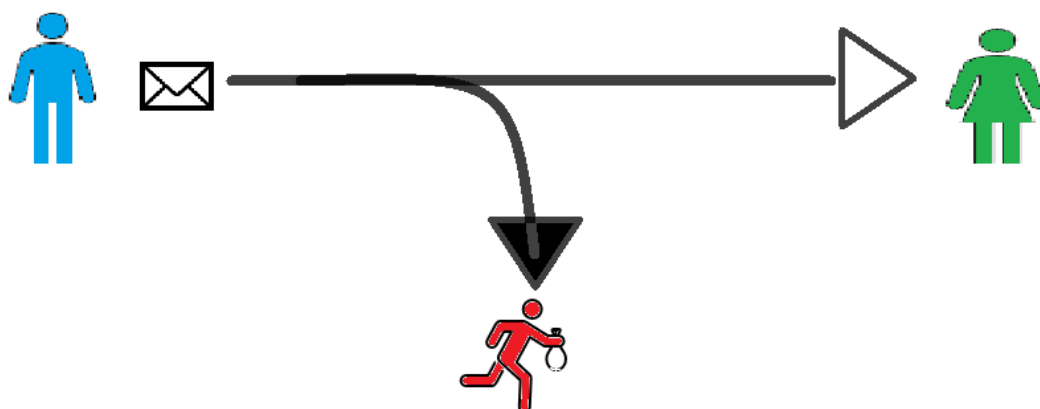
Interruption

Prevents data from getting to where it needs to be. This is an attack on availability, such as a denial of service attack.



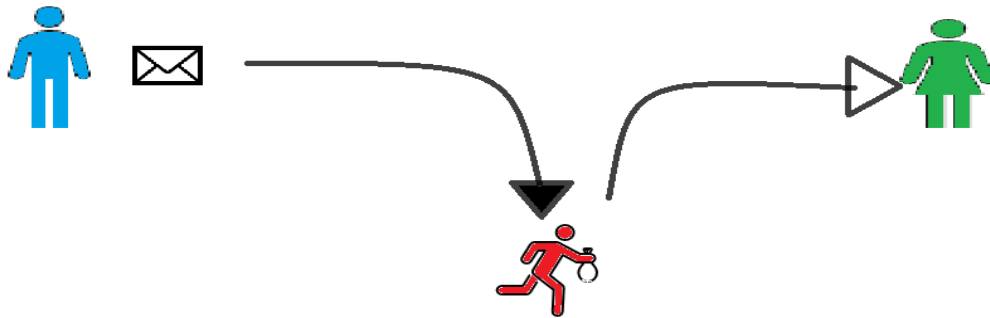
Interception

Unauthorised third parties can eavesdrop and monitor messages that not intended for them. This is an attack is a violation of privacy.



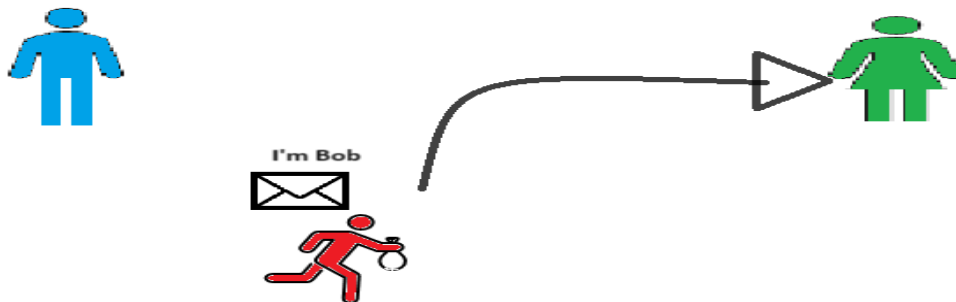
Modification

When a third party alters a message while it is enroute from the sender to the receiver. This is a violation of the original message's integrity.



Fabrication

A third party impersonates another entity like what happens in phishing attacks. This is an attack on authenticity.



With the rise of modern problems, modern solutions are required and a lot of these “modern solutions” are inspired by the old-fashioned outdated ways of manually data transmission. One such a solution I have implemented is detailed below.

Contents

1) Communication Connectivity Model	3
a) Server Side	3
b) Client Side	3
i) Start Up	3
ii) Connecting to other Clients	3
2) Cryptosystem Design	5
a) Key Management	7
b) Cryptographic Algorithms	7
3) Testing & Assumptions	7
4) Methodology	8
5) Appendix	9

I have designed an application for users to securely transfer images to one another.

1) Communication Connectivity Model

a) Server Side

Upon start up the server instantiates the Certificate Authority (CA) object. The CA first checks if its own self-signed X509 certificate and asymmetric key pair exist. If they do it loads them (along with other valid certificates the server has stored), otherwise they are generated.

The Server runs continuously and listens for connection requests from clients. It is able to handle several clients at a time through the use of threads and each of these threads access have access to the CA - which is treated as a shared class. This ensures the certificate remains consistent across all client handler threads. Locks are utilised to combat data races, particularly when writing the generated certificates and appending them to the list of exiting valid certificates. The client never directly interacts with the CA class.

b) Client Side

i) Start Up

For a first time user of the application, they are required to enter their name. Once this is done asymmetric keys are generated and TCP connection to the CA is made. The application on the client side will create a Certificate Signing Request (CSR), sending the clients name, their digital signature (hash of their name signed by the client's private key) and the client's public key. The second a CSR is received the CA is aware this is a brand new user, so it sends its own certificate to the client so that they may be able to load the CA's public key. The CA will then verify the identity of the client by using the received public key on the received digital signature. If successful this proves that the client owns the private key that corresponds with the public key the CA received, then the CA generates the self-signed X509 digital certificate using the details the client sent and sends it to the client. Because digital certificates do not contain confidential information and the CA is trusted. the transfer is not encrypted.

If the user is not a first time user (has its key pair and certificate) the application welcomes them and immediately loads their key pair, provided the certificate is still valid. If not, it deletes the certificate and creates a new CSR to the CA. This is all done without user input. Ordinarily this process would be abstracted from the user, but for the sake of this assignment it is show in the console.

ii) Connecting to other Clients

If a client (call her Alice) wants to contact another client (call him Bob) - that is open to contact (temporarily listening) – Alice will first check if she has Bob's valid certificate and the session key with Bob exists. If one of these conditions isn't met, the TCP connection is made Alice will signal to Bob to expect the handshake process.

If the conditions are met and Bob is waiting to be contacted, Alice indicates to Bob no handshake is needed the session is resumed. This saves the time wasted doing the handshake. This is also safe to do so for reasons that will be detailed further down.

Again no user input involved beyond Alice indicating she want to contact Bob and Bob indicating he is expecting contact. On Alice's side, if Bob isn't open to communicating the application will indicate to her that Bob isn't available at that moment.

Certificate validation is referred to a lot in the next subsection, so to avoid possible confusion it is define in my implementation by:

- (1) Verifying it was signed by the CA using the CA's public key to check,
- (2) Verifying it is not expired &
- (3) Confirming the subject name matches the sender's name.

This limits access to those with certificates issued by a trusted third party (the CA).

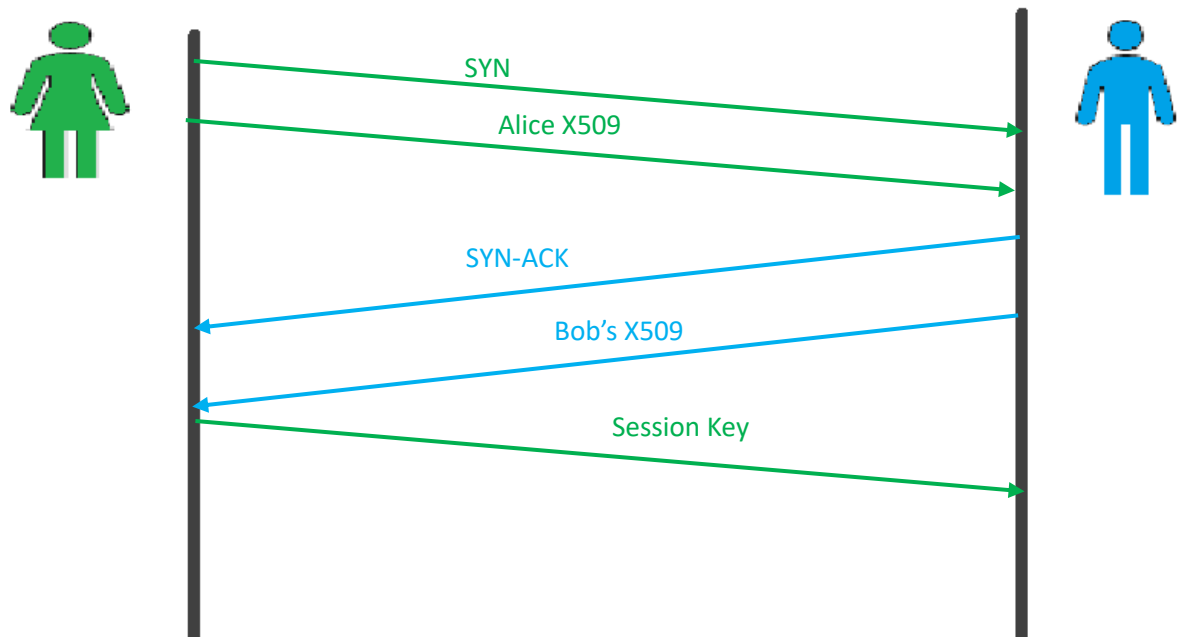
Handshake

During the handshake, Alice (who initiated contact) sends a Client Hello (SYN) to Bob - detailing the encryption, hashing and compression algorithm she wants used in the session – along with her digital certificate (will be referred to as just X509 going forward). Bob compares the SYN against his own preferred protocols and verifies Alice X509 is valid. He then sends a Server Hello (SYN-ACK) saying whether or not he accepts. If his application side doesn't approve the handshake fails, TCP connection closed and they are both returned to the main menu.

Suppose the SYN does align with Bob's preferred protocols and Alice's X509 is valid, Bob will respond with a SYN-ACK and send his own X509 back to Alice. Alice will validate his X509. If successful, she generates the session key for them to use along with a message essentially greeting Bob. A Pretty Good Privacy (PGP) formatted message containing the message and session key (among other things) is sent to Bob using PGP cryptographic functions. If the X509 Bob sent to Alice does indeed belong to Bob, Bob should be able to get the decoded session key and see the message (more on PGP in next section). Bob will then let Alice know. In the scenario where this is the case, the handshake is complete (FIN), and communication is safe so the session begins. Again, the actual users of the application cannot use the interface to interfere with the process as it is done automatically. e.g., sending someone else's certificate during the handshake. This is not possible as the application knows which certificate belongs to the user.

The menu is then presented to both users for them to interact with for sending and receiving images from both sides using the shared session key.

The handshake is inspired by the SSL handshake and loosely inspired by QUIC protocol. For the sake of speed, the SYN & SYN-ACK and the certificate exchange is done in a combined step. The session key is also used in active sessions as it is less computationally expensive than the key pair (for reasons detailed in the next section). Sent images are also compressed as before being sent to speed up file transmission.



2) Cryptosystem Design

As mentioned above, in the final step of the handshake the client who initiated it generates a symmetric key that will be used as a session key. A plaintext greeting message is created as a text file and a message digest is created by using a hashing function on the message. The message digest is then signed by Client A's private key to create a digital signature. By PGP principles, the message (as well as the digital signature) is compressed after adding A's digital signature but before encryption, so naturally this is the next step. The compressed file is then encrypted using the session key to create a ciphered zip. During encryption files are overwritten so the zip won't be encoded as a normal zip to a machine. I was not able to view files in an encrypted zip nor was I able to unzip it on Linux. The session must be kept a secret between A and B as it is used in future encryption for the session. So, A encrypts a *copy* of the session key using B's public key (retrieved from the certificate that B sent). Then the encoded session key and ciphered zip is sent to B. At this point the key pairs are not longer used in the session, only the session key.

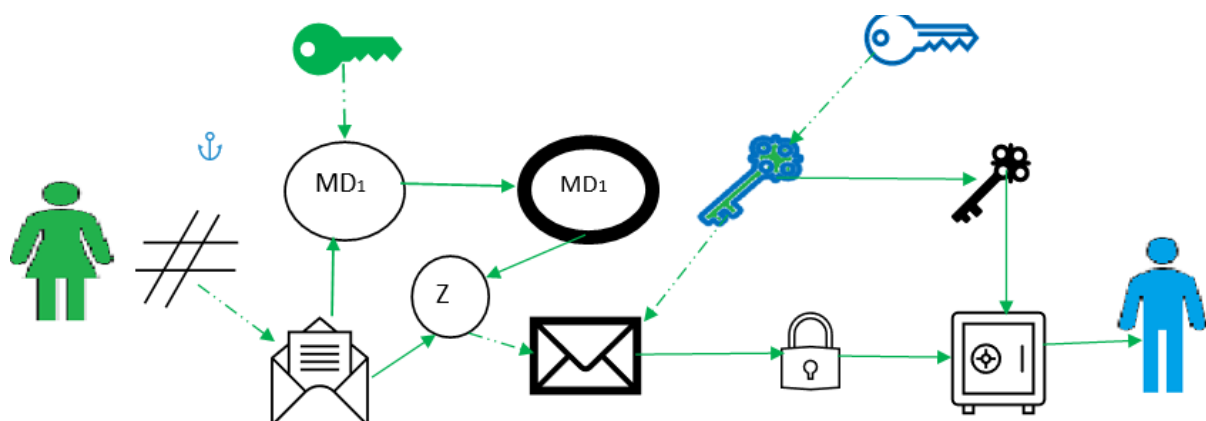


Figure 1: Visualisation of the PGP cryptographic function (The legend is in the appendix)

If someone is impersonating B, they will not be able to decrypt the encoded session key when they receive it as they would need B's private key. Without the session key the imposter won't be able to

view the message or decrypt files received or encrypt files sent. Confidentiality between the real B and A is maintained.

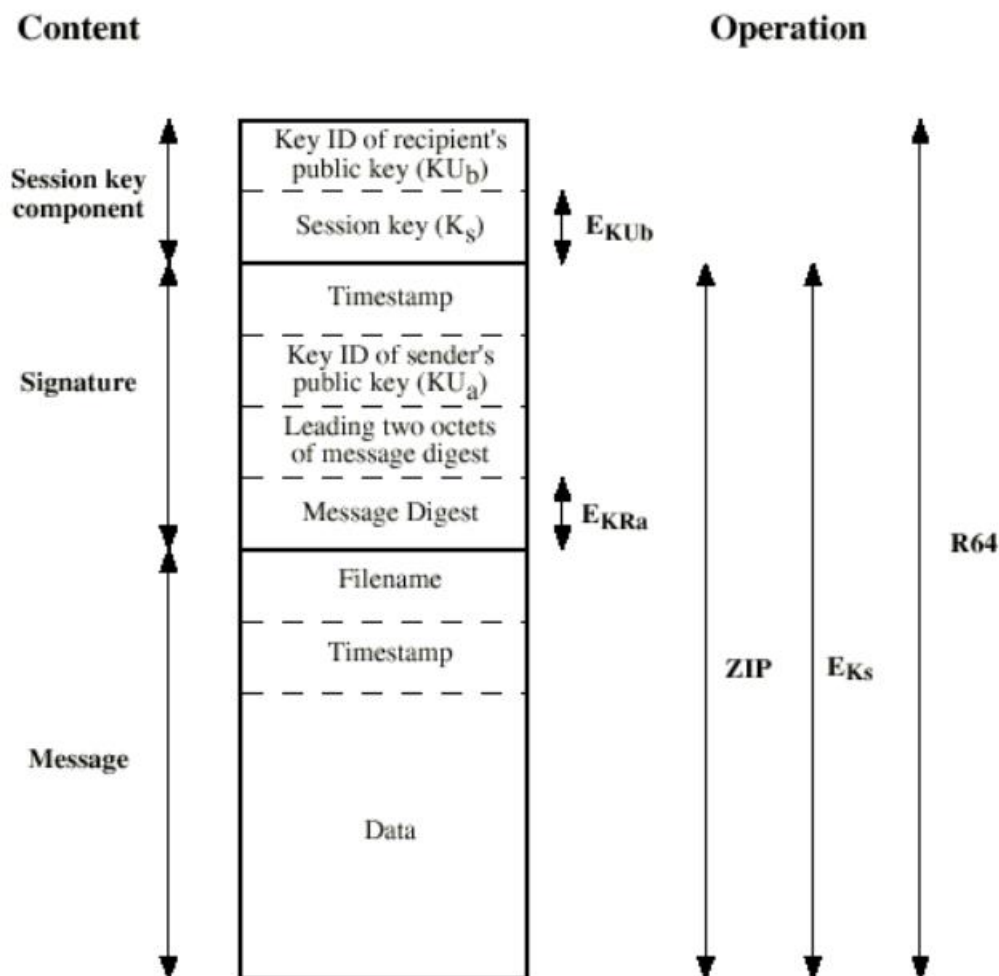


Figure 2: PGP Message Format

Once B receives the encoded session key (reminder it was done using B's public key), B decrypts it using his own private key. Now B can also decipher the encrypted zip file he received. Without doing this B will not be able to unzip as the file isn't encoded as a standard zip file. Once the zip file is decoded it is unzipped (decompressed). B will have to use A's public key (sent during the handshake) to verify the digital signature (to know it was truly Alice who sent it), then create his own message digest on the plaintext greeting message. Remember we use A's public key to verify the sender's identity as the message digest would have been signed by A's private key if the message is really from A. B will then compare his generated message digest to the one that came in the zip. If they are the same, then B can be certain that the message hasn't been altered.

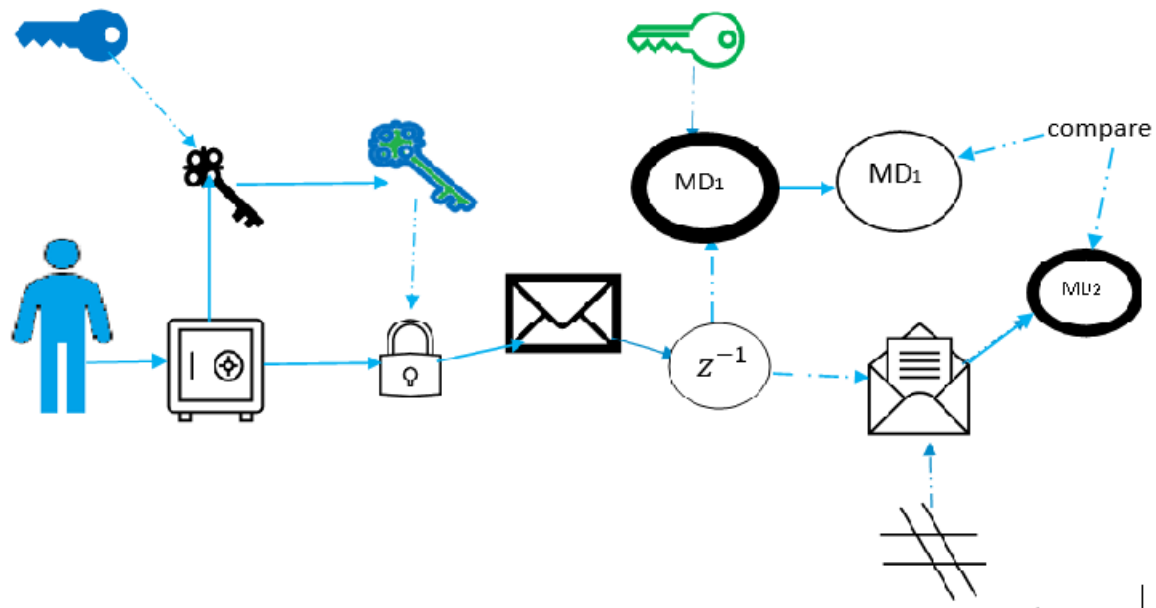


Figure 3: Bob decoding, decompressing, authenticating Alice identity and verifying message integrity.

The above displays how we combat message interception by ensuring privacy with encryption, fight fabrication threats by authenticating the sender's identity through the use of digital signatures. We also counter fabrication threats by authenticating the sender's identity through comparing message digests.

a) Key Management

Now that B knows the session key came from A. It can be used for the session (while they are connected). The reason the session keys are used over the RSA keys is because RSA encryption and decryption is far more computationally expensive than AES encryption and decryption. Key sizes for RSA are also 8 times as large. For the duration of the session images are hashed, compressed, encrypted and sent, then received, decrypted, decompressed and hashes are compared. For the convenience of both users and for time efficiency, the session keys are valid until the app is closed. So users can disconnect from one another, then resume their previous session (without another handshake), provided neither user closed the application. When the application is closed (i.e., code is stopped), all session keys are terminated. On subsequent code runs handshakes will be required again. The session keys are not meant to last an infinite time for the safety of other users.

b) Cryptographic Algorithms

- Asymmetric Key (Private + Public) Pair: RSA 2048 bits
- Symmetric (Session) Key: AES 256 bits
- Compression: Zip
- Hashing Algorithm: SHA256
- Symmetric Encryption: AES/CBC/PKCS5Padding
- Asymmetric Encryption: RSA/ECB/PKCS1Padding

3) Testing & Assumptions

Though some fault tolerance is implemented, I am still assuming follow the instructions carefully. This includes sending files that exist as they are named and properly exiting the application as well as

not trying to manipulate the stored certificates or keys during code execution. I also assume the server is constantly running.

For tests, for the sake of simplicity I copied the methods from the main code and pasted them in a separate class called EncryptMethods tested by EncryptMethodsTest class. I also removed code blocks where files are written. As persistence isn't a requirement for test. Also prevents crowding the folder with unnecessary files.

I tested Session Key encryption using Public Keys and decryption using Private keys. The original session key was then compared to the decrypted one.

I tested if the public key retrieved from a generated certificate matched the public key generated before certificate creation.

I tested String encryption and decryption using AES symmetric keys. Original string and the deciphered string were then compared. I also testing verifying a sender's digital signature and verifying integrity by comparing hashes.

All tests passed. The testing was done using an IDE and JUnit as well as Jupiter libraries.















4) Methodology

I started with trying to see if I could create a TCP connection between two separate machines. This included sending messages, creating compression methods and working on file transfer. Once that worked, I worked on making the server able to hand several clients through multi-threading.

In a separate workspace I worked on key & certificate generation, encryption and decryption. I added break points to between encrypting and decrypting to pause the code so I could verify the files were actually encrypted. I was unable to open them or when they did open the content was unintelligible. The methods for certificate generation heavily relied on the bouncy castle library.

Integrating the two work spaces was a challenge. Version Control was used so the steps taken are replicable. Was also helpful for saving and tracking progress

5) Appendix

	Zip function
	Action
	Next Step/Goes Into/Becomes
	Private Key
	Public Key
	Session Key
	Plaintext
	Hash Function
	PGP Message
	Zipped Message with Digital Signature
	Encoded Session Key
	Ciphered Zip
	Signed Message Digest
	Message Digest