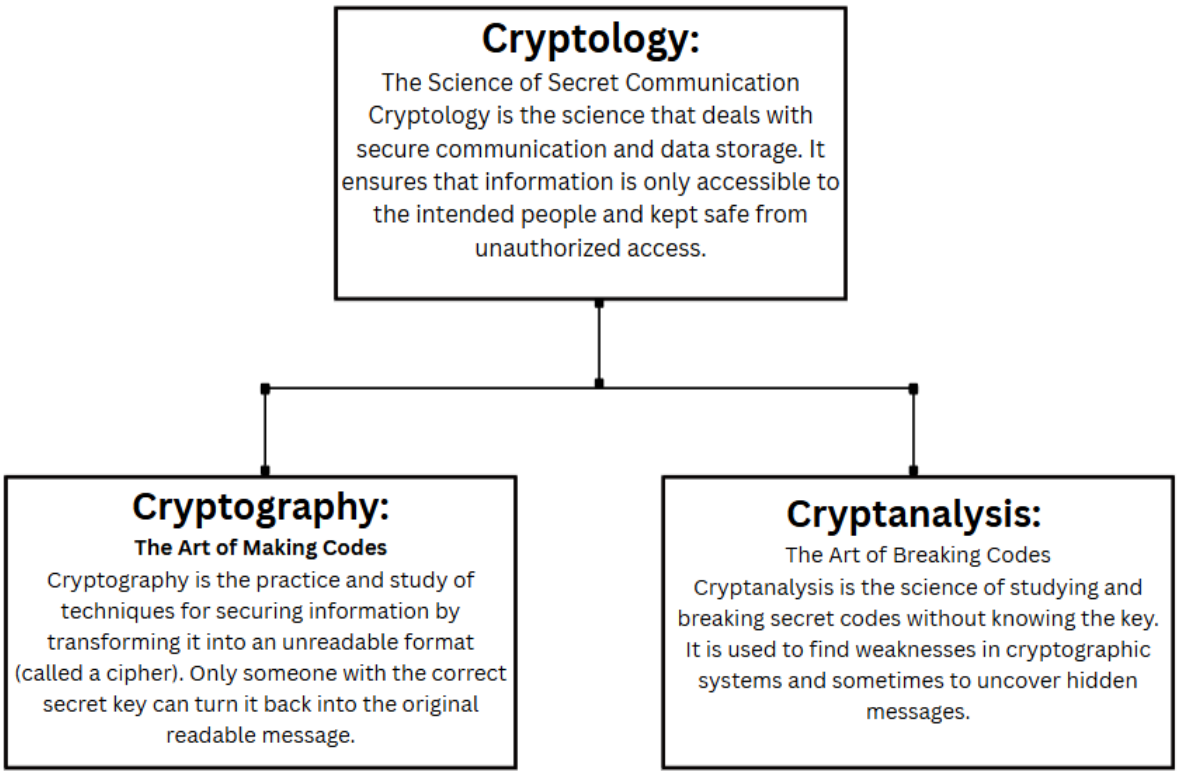
Defence Research & Development Organisation

Internship Project: Cryptography



SSL:

SSL stands for Secure Sockets Layer. It's a technology that helps create a secure and encrypted connection between a web browser (like Chrome or Firefox) and a web server (where a website lives). This ensures that any data sent between you and the website is private and cannot be read or changed by anyone else.

SSL is what makes websites show https:// at the start of their addresses instead of just http:// — the "s" means the connection is secure.

HTTPS:

HTTPS stands for HyperText Transfer Protocol Secure. It's an extension of the regular HTTP, which is the protocol your browser uses to load websites. The difference is that HTTPS adds a layer of security using encryption, so the data exchanged between your browser and the website is private and protected from hackers.

HTTPS uses SSL/TLS (Secure Sockets Layer / Transport Layer Security) to encrypt the communication between your device and the website server. This means:

- When you visit a website starting with https://, your browser first sets up a

secure connection using SSL/TLS.

- All the information you send or receive — like passwords, credit card numbers, messages — is scrambled (encrypted) so nobody else can read it if they intercept the data.

Importance of https:

1. **Data Privacy:** It keeps your information safe from attackers who might try to steal it.
2. **Data Integrity:** It ensures that the data you receive or send is not tampered with or changed by anyone.
3. **Authentication:** It confirms that the website you're visiting is genuine, not a fake or a scam site.

Establishes Encrypted Communication:

When we connect to a website using SSL, several steps happen behind the scenes to create a safe, encrypted link:

1. **ClientHello**
   The client starts the handshake by sending a ClientHello message with its supported SSL/TLS version, list of cipher suites, compression methods, a random number, and an optional session ID.

2. **ServerHello**
   The server responds with a ServerHello message, selecting the SSL/TLS version, cipher suite, compression method, and providing its own random number and session ID.

3. **ServerKeyExchange**
   If needed (e.g., in ephemeral key exchange like DHE/ECDHE), the server sends its public key or key exchange parameters to help the client establish a shared secret.

4. **ServerHelloDone**
   This message indicates the server has finished its part of the handshake and is waiting for the client's response.

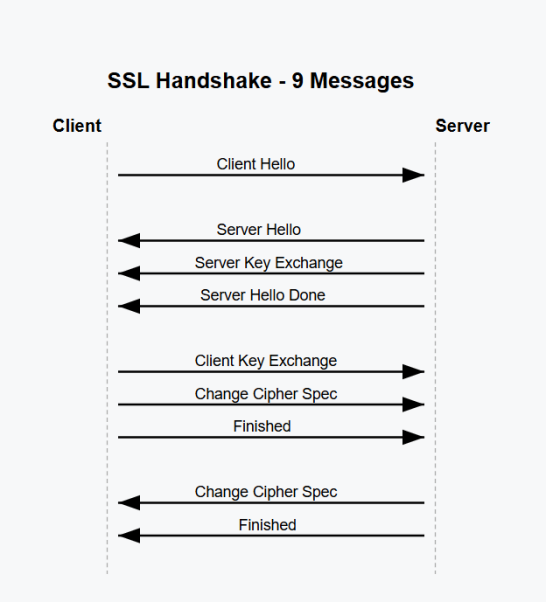
5. **ClientKeyExchange**
   The client responds with a ClientKeyExchange message, sending the pre-master secret encrypted with the server's public key (or using key agreement parameters).

6. **Client ChangeCipherSpec**
   The client tells the server it is switching to encrypted mode using the agreed cipher suite and session key.

7. **Client Finished**
   The client sends a Finished message encrypted with the session key. It includes a hash of all previous handshake messages for integrity verification.

8. **Server ChangeCipherSpec**
   The server also switches to encrypted communication using the same session key.

9. **Server Finished**
   The server sends its own encrypted Finished message to confirm the handshake was successful and the session is secure.



Symmetric Key:

Symmetric key cryptography is a method of encryption where the same key is used to both encrypt and decrypt the message. This means that both the sender and the receiver must have access to the same secret key and must keep it private.

Working of symmetric key:

1. The sender uses the key to convert the original message (called plaintext) into an unreadable format (called ciphertext).

2. The receiver then uses the same key to change the ciphertext back into the original message.

Key Features:

- **Efficiency**
  Symmetric key cryptography is highly efficient, especially when dealing with large volumes of data. It requires fewer resources and processes data faster than asymmetric methods, making it suitable for real-time applications.
- **Simplicity**
  This method uses just one secret key for both encryption and decryption. The simplicity of having only one key makes it easier to implement and manage in many systems.
- **Security**
  The security of symmetric cryptography depends entirely on how well the key is protected. If the key is kept secret and shared securely between the sender and receiver, the communication remains safe.
- **Speed**
  Algorithms like AES (Advanced Encryption Standard) and DES (Data Encryption Standard) are optimized for speed. They are designed to encrypt and decrypt data quickly, making them effective for time-sensitive applications.
- **Symmetry**
  Since the same key is used on both ends, both parties must have access to the same key before communication starts. This requires a secure way to share the key, which can be challenging, especially over long distances or open networks.

Asymmetric Key:

Asymmetric Key Cryptography, also known as public-key cryptography, is a type of encryption that uses two different keys instead of one. These keys are called the public key and the private key.

- The public key is shared openly and used to encrypt (lock) the information.
- The private key is kept secret and used to decrypt (unlock) the information.

Because the keys are different, you don't need to share the private key with anyone. This makes communication safer, especially over the internet or other insecure channels.

This method is very useful for things like:

- Digital signatures: proving who sent a message.
- Secure key exchanges: safely sharing secret information.

Asymmetric cryptography forms the backbone of many secure communication protocols used today.

Key Features:

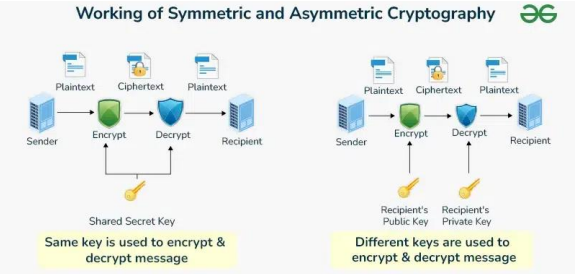
Key Pair: It uses two keys—a public key and a private key—to lock (encrypt) and unlock (decrypt) data. This makes it more secure than using just one key.

Confidentiality: The public key can be shared with anyone, but the private key is kept secret by the owner. This keeps the information safe.

Digital Signatures: It can create digital signatures, which help prove that a message is really from the sender and hasn't been changed.

Key Distribution: Only the public key needs to be shared with others. The private key stays secret, so it's harder for attackers to steal the secret key.

Resource Intensive: It takes more time and computing power compared to other methods (like symmetric cryptography), so it's not ideal for encrypting very large amounts of data.



Source: www.geeksforgeeks.org

Data Encryption Standard:

DES (Data Encryption Standard) is a type of symmetric-key encryption — this means it uses the same key for both encrypting and decrypting data. It was developed and published by NIST (National Institute of Standards and Technology) and was widely used for securing data in the past.

Key Features of DES:

1. **Symmetric Key Cipher**: Same key is used for both encryption and decryption.
2. **Block Cipher**: It works on blocks of data, not individual bits or bytes. In DES, each block is **64 bits** (8 bytes) long.
3. **Feistel Structure**: DES is based on something called the **Feistel Cipher**, which divides the data block into two halves and processes them through **16 rounds** of complex transformations (mixing and substitution operations). Each round increases the security.
4. **Key Size**: The key used is **64 bits** long, but only **56 bits** are actually used in the encryption process. The other 8 bits are used for error checking or are just ignored.

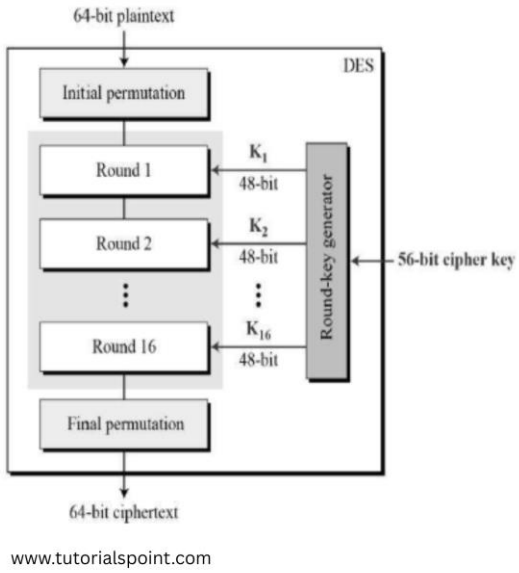
Working of DES:

1. **Input**: 64-bit block of plain text.

2. **Key**: 56-bit effective key is used.

3. **Rounds**: The data goes through **16 rounds** of encryption, involving shifting, substitution, permutation, and mixing.

4. **Output**: A 64-bit block of encrypted data (ciphertext).



Advanced Encryption Standard:

AES (Advanced Encryption Standard) is a symmetric-key encryption algorithm used to secure data. AES is different from DES in the way it processes data:

- It is not a Feistel cipher (like DES).
- Instead, AES uses a method called a Substitution-Permutation Network.

This means encryption is done using a mix of:

- **Substitution**: Replacing data using fixed patterns (like a lookup table).
- **Permutation**: Rearranging the data (like shuffling).

Working of AES:

1. **Block Size:** AES works with 128-bit blocks of data.
    - These 128 bits are grouped into 16 bytes.
    - The bytes are arranged into a 4x4 matrix (4 rows and 4 columns) for processing.

2. **Byte-Level Operations:** Unlike DES, which works on bits, AES does all its processing on bytes, making it more efficient on modern hardware.

AES goes through multiple rounds of encryption steps. The number of rounds depends on the key size:

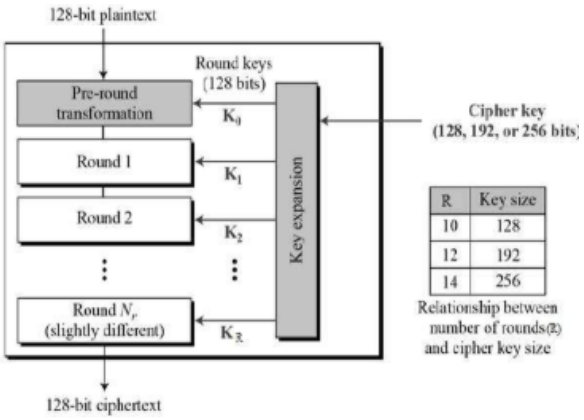
- 10 rounds for a 128-bit key
- 12 rounds for a 192-bit key
- 14 rounds for a 256-bit key

Each round involves several steps, like:

- SubBytes (substitution)
- ShiftRows (row shifting)
- MixColumns (column mixing)
- AddRoundKey (adding a unique round key)

Key Expansion:

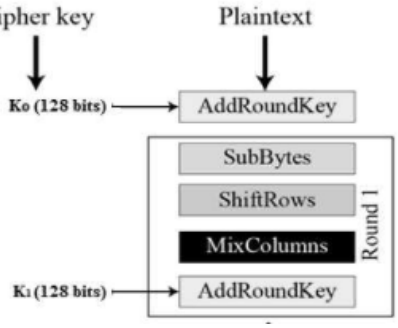
- AES creates a different key for each round.
- These round keys (also 128-bit each) are generated from the original AES key using a process called Key Expansion.



R	Key size
10	128
12	192
14	256

Relationship between number of rounds(R) and cipher key size

Each round comprise of four sub-processes. The first round process is depicted below:



In present day cryptography, AES is widely adopted and supported in both hardware and software. Till date, no practical cryptanalytic attacks against AES have been discovered. Additionally, AES has built-in flexibility of key length, which allows a degree of future-proofing against progress in the ability to perform exhaustive key searches.

Ensuring Data Integrity with Hash Codes:

A hash value is a fixed-length numerical representation that uniquely corresponds to a specific set of data. Since it condenses large data into a smaller, manageable format, hash values are commonly used in digital signatures for efficiency. Instead of signing an entire dataset, it's faster and more practical to sign its hash. Hashing also plays a crucial role in verifying data integrity, especially when information is transmitted over untrusted networks. By comparing the hash of the received data with the original hash, one can detect any tampering or alterations.

Key Aspects:

- **File Integrity Checking**: Our expertise in File Integrity Checking guarantees the integrity of your data. We employ advanced hashing mechanisms to verify file integrity, preventing unauthorized alterations or tampering.
- **Hashing Mechanism Expertise:** Utilizing robust hashing algorithms, we create unique hash values for your data. These values act as digital signatures, ensuring the authenticity and reliability of your files.
- **Automatic Hash Verification:** Our system offers Automatic Hash Verification, allowing seamless checking of active elements against original hash values. This automated process ensures data integrity without manual intervention.
- **Data Integrity Monitoring:** Through Data Integrity Monitoring, we provide continuous surveillance of your files, promptly identifying any discrepancies or unauthorized changes, allowing for immediate corrective actions.
- **File Hash Validation:** Experience precise File Hash Validation, ensuring that files remain unaltered. We employ industry-standard algorithms to generate and validate file hashes, providing a robust defense against data manipulation.

RSA:

RSA (Rivest-Shamir-Adleman) is an asymmetric encryption algorithm, meaning it uses two different keys: a public key and a private key. The public key is shared openly

and is used for encryption, while the private key is kept secret and is used for decryption. This method ensures secure communication between parties. In a simple example, if one person wants to send a confidential message to another, they encrypt it using the receiver's public key. Only the receiver, who holds the corresponding private key, can decrypt and read the message. This way, even if someone intercepts the message, they won't be able to understand it without the private key.

Key Generation:

Start by selecting two large prime numbers, say p and q. These primes must be kept confidential. Next, compute their product: n = p × q. This value n is included in both the public and private keys.
Now, calculate Euler's Totient Function for n, which is given by $\Phi(n) = (p - 1) \times (q - 1)$.

Choosing an encryption exponent e such that:

- $1 < e < \Phi(n)$, and
- e is co-prime with $\Phi(n)$, meaning $\gcd(e, \Phi(n)) = 1$.

Then, determine the decryption exponent d so that it satisfies the condition: $(d \times e) \equiv 1 \bmod \Phi(n)$.
This means d is the modular multiplicative inverse of e modulo $\Phi(n)$. You can compute d using methods like the Extended Euclidean Algorithm or Fermat's Little Theorem.

There can be several valid values of d that satisfy this condition, but any one of them can be used, as all will correctly decrypt the message.

At the end, we have:

- Public Key = (n, e)
- Private Key = (n, d)

Encryption:

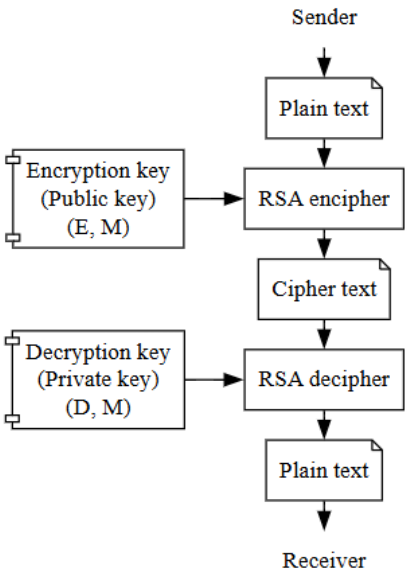
To encrypt a message M, it is first converted to numerical representation using ASCII and other encoding schemes. Now, use the public key (n, e) to encrypt the message and get the cipher text using the formula:

C = Me mod n, where C is the Cipher text and e and n are parts of public key.

Decryption:

To decrypt the ciphertext C, use the private key (n, d) and get the original data from the formula:

M = Cd mod n, where M is the message and d and n are parts of private key.



OBJECTIVE:

To develop a client which can send and receive messages/files. If two clients are installed on different computers(for testing you can use the same computer but with different ports) then they can communicate to each other by sending messages/files. This can be written in code in Python or any other programming language. Further implement the following functionalities (similar to SSL/TLS Protocol):

1. Encryption and decryption scenario(using standard algorithm AES)
2. Hash (SHA) based integrity check for messages and files.
3. Key exchange mechanism using RSA algorithm
4. Auto discovery of active clients in the network (more oriented towards network programming)

Week 1:

Peer-to-Peer Communication and AES-style Encryption

Objective:
To implement a peer-to-peer communication system in Python that supports:

- Encrypted message exchange using a symmetric key algorithm (XOR-based AES alternative)
- Secure file transfer
- Communication between two clients on the same computer using different ports

Tools and Technologies Used:

- Python 3.x
- `socket` (for communication)
- `threading` (for concurrent listening and sending)

Implementation:
Two clients are set up using different ports on the same machine. Each user enters a shared secret password which is used to derive a symmetric key. The message/file content is then encrypted using a custom XOR-based encryption algorithm (mimicking AES logic in a simplified way). This ensures that the data transferred over the network is not visible in plaintext.

- The encryption function XORs each byte of the data with the password-derived key.
- The same key is used for decryption on the receiver's side.

Outcome:
The system successfully allowed **secure encrypted communication** between peers. Only clients who knew the shared password were able to **decrypt and read the messages or files**, validating the confidentiality of the communication.

Week 2:

Integrity Check Using Custom Hash Function

Objective:
To implement an integrity verification mechanism for both messages and files by introducing a custom hash-based check that ensures the data has not been tampered with during transmission.

Implementation:

1. Before sending:
    - Compute the hash of the data.
    - Append the hash to the original data.

- Encrypt the combined content using the XOR-based method.

2. After receiving:
   - Decrypt the message/file.
   - Separate and extract the hash.
   - Recompute the hash on the received content.
   - Compare both hashes:
     - If they match → Accept and display/save content.
     - If they don't → Display integrity error and discard the data.

Outcome:

This week's implementation ensured that any tampering or data corruption during transmission is detected on the receiver side. Users are warned with a clear error message if the hash does not match, which adds a layer of data integrity verification in addition to encryption.

Code Snippets:

1. Key Derivation-

```
def derive_keys(password: bytes, salt: bytes) -> tuple:
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=64,  # 32 bytes for AES key, 32 bytes for HMAC key
        salt=salt,
        iterations=100_000,
        backend=backend
    )
    key = kdf.derive(password)
    return key[:32], key[32:]
```

2. AES Encryption with SHA-based Integrity-

```
def encrypt(data: bytes, password: bytes) -> bytes:
    salt = os.urandom(16)
    iv = os.urandom(16)
    enc_key, mac_key = derive_keys(password, salt)

    padding_len = 16 - (len(data) % 16)
    padded_data = data + bytes([padding_len] * padding_len)

    cipher = Cipher(algorithms.AES(enc_key), modes.CBC(iv), backend=backend)
    encryptor = cipher.encryptor()
    ciphertext = encryptor.update(padded_data) + encryptor.finalize()

    h = hmac.HMAC(mac_key, hashes.SHA256(), backend=backend)
    h.update(ciphertext)
    tag = h.finalize()

    return salt + iv + tag + ciphertext
```

3. AES Decryption with Integrity Verification-

```
def decrypt(enc_data: bytes, password: bytes) -> bytes:
    salt = enc_data[:16]
    iv = enc_data[16:32]
    tag = enc_data[32:64]
    ciphertext = enc_data[64:]

    enc_key, mac_key = derive_keys(password, salt)

    h = hmac.HMAC(mac_key, hashes.SHA256(), backend=backend)
    h.update(ciphertext)
    h.verify(tag)

    cipher = Cipher(algorithms.AES(enc_key), modes.CBC(iv), backend=backend)
    decryptor = cipher.decryptor()
    padded_data = decryptor.update(ciphertext) + decryptor.finalize()

    padding_len = padded_data[-1]
    return padded_data[:-padding_len]
```

4. Peer Communication and Threaded Listening-

```
def handle_peer(sock, password):
    while True:
        try:
            data, addr = sock.recvfrom(65536)
            if data.startswith(b"msg"):
                decrypted = decrypt(data[3:],
password)
                print(f"[MESSAGE from {addr}]
{decrypted.decode()}")
            elif data.startswith(b"file"):
                decrypted = decrypt(data[4:],
password)
                filename =
f"received_file_from_{addr[1]}.bin"
                with open(filename, "wb") as f:
                    f.write(decrypted)
                print(f"[RECEIVED] File saved as
'{filename}'")
        except Exception as e:
            print(f"[ERROR] Integrity check failed
or invalid message: {e}")
```

5. Sending Encrypted Messages and Files-

```
def send_message(sock, password,
target_port):
    msg = input("Enter message: ").encode()
    encrypted = encrypt(msg, password)
    sock.sendto(b"msg" + encrypted,
("localhost", target_port))

def send_file(sock, password, target_port):
    filepath = input("Enter file path: ").strip()
    with open(filepath, "rb") as f:
        data = f.read()
    encrypted = encrypt(data, password)
    sock.sendto(b"file" + encrypted,
("localhost", target_port))
```

Key Exchange Mechanism using RSA (teammate):

Code Snippets:

1. RSA key pair generation-

```
def generate_rsa_key_pair():
```

Returns: A tuple containing:

- private_key: The RSA private key.
- public_key: The corresponding RSA public key.

2. Conversion of public key into a byte string-

```
def serialize_public_key(public_key):
```

- Returns: Serialized public key in PEM format (bytes)

3. Conversion of private key into PEM format-

```
def serialize_private_key(private_key,
password=None):
```

- Returns: Serialized private key in PEM format (bytes).

4. Conversion of public key back to usable public key object-

```
def deserialize_public_key(pem_data):
```

- Returns: Deserialized public key object.

5. Conversion of private key back to usable private key object-

```
def deserialize_private_key(pem_data,
password=None):
```

- Returns: Deserialized private key object

6. Symmetric key generation-

```
def
generate_symmetric_key(length_bytes=32):
```

- Returns: The symmetric key of a specified byte length

7. Symmetric key encryption-

```
def encrypt_symmetric_key(symmetric_key,
recipient_public_key):
```

- Returns: Encrypted symmetric key (bytes).

8. Symmetric key decryption-

def decrypt_symmetric_key(encrypted_key, recipient_private_key):

- Returns: The original symmetric key (bytes)

Workflow:

1. Alice and Bob each generate RSA key pairs.
2. Alice generates a random symmetric key.
3. Alice encrypts it using Bob's public key.
4. Bob decrypts it using his private key.
5. The symmetric key is compared before and after to verify success.

Output:

- Public keys.
- Original symmetric key.
- Encrypted key.
- Decrypted key.
- Success or failure message.

Code Snippets:

Python
```
import socket
import threading
import json
import time # Used for time.sleep to introduce delays

class ClientDiscovery:
    def __init__(self, broadcast_port=5002):
        """Initializes the ClientDiscovery module for broadcasting and listening.
        Sets up a UDP socket for broadcast communication on a specified port."""
        self.broadcast_port = broadcast_port
        self.active_clients = [] # List to store information of all discovered active clients
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM) # Create a UDP (Datagram) socket
        self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1) # Enable the broadcast option on the socket
        self.sock.bind(('', self.broadcast_port)) # Bind the socket to listen on all available network interfaces on the broadcast port

    def broadcast_presence(self, host, port):
        """Continuously broadcasts the client's own presence on the network.
        Sends its host and port in a JSON-encoded message to the broadcast address."""
        message = json.dumps({"host": host, "port": port}).encode()
        while True:
            # Send the broadcast message to the network's general broadcast address
            # This allows other devices on the same local network segment to receive it
            self.sock.sendto(message, ('255.255.255.255', self.broadcast_port))
            time.sleep(5) # Pause for 5 seconds before the next broadcast, reducing network congestion

    def listen_for_clients(self):
        """Listens for incoming broadcast messages from other clients.
        Discovers new clients and adds their information to the active_clients list, printing a notification."""
        while True:
            data, addr = self.sock.recvfrom(1024) # Receive up to 1024 bytes of data
            client_info = json.loads(data.decode()) # Decode the JSON message to extract client information
            if client_info not in self.active_clients:
                self.active_clients.append(client_info)
                print(f"Discovered client: {client_info}") # Print a notification when a new client is discovered
```

Explanation:

This Python module enables automatic discovery of other clients on the same local network. It does this using the ClientDiscovery class, which sets up a UDP socket that can send and receive broadcast messages. The module works without any central server or manual configuration, making it easy to connect devices in a chat or peer-to-peer application.

Working:

- The broadcast_presence method runs in a separate thread and sends a broadcast message every 5 seconds. This message contains the device's IP address and port in JSON format, effectively announcing its presence to the entire network.
- The listen_for_clients method runs in another thread and constantly listens for incoming broadcast messages from other devices.
- When a message is received, it checks if the sending client is new. If so, it adds that client to the active_clients list.

Week 3:

Backend Development and Initial Basic GUI

Objective:

- Implement all 4 components of the project:
    - AES encryption + HMAC (SHA-based) integrity
    - RSA key exchange
    - Peer auto-discovery on local network
    - UDP-based peer-to-peer communication
- Design and test a basic GUI to interact with the backend

Execution:

- Created individual Python modules:
    - p2pclient.py for message & file encryption/decryption using AES & HMAC
    - rsa.py for secure RSA public/private key generation and symmetric key exchange
    - autodis.py for discovering peers over the LAN using UDP broadcast
- Built a basic GUI using standard input prompts and print statements
    - User manually enters ports, messages, and file paths
    - Outputs appear in terminal
- Set up multithreading to allow simultaneous listening and user interaction

Outcomes:

- All 4 components successfully implemented and tested in an integrated flow
- Basic working GUI (terminal-based) created to demonstrate core functionality

Week 4:

GUI Modernization with Streamlit

Objective:

- Replace terminal interface with a modern, user-friendly GUI using Streamlit

- Enable easy interaction with encryption functions, file sharing, and key exchange

Execution:

- Developed a full Streamlit-based GUI (gui.py):
  - Text input for messages
  - Buttons for Send Message, Send File, and Key Exchange
  - Text area to display chat logs
  - Real-time status indicators for active peers
- Integrated backend logic from previous week:
  - AES encryption/decryption hooks
  - RSA key exchange triggered via button
  - Auto-discovery continuously updates peer list
- Added user feedback and error handling:
  - Warnings for missing peer
  - Confirmation messages on send
  - Live display of symmetric key on exchange

Outcomes:

- Replaced basic GUI with a modern, interactive Streamlit GUI
- Enabled complete P2P communication via buttons and form fields
- Made the tool suitable for real-world testing and future demo

Week 5:

Auto Public Key Exchange in Tkinter-Based Secure Chat

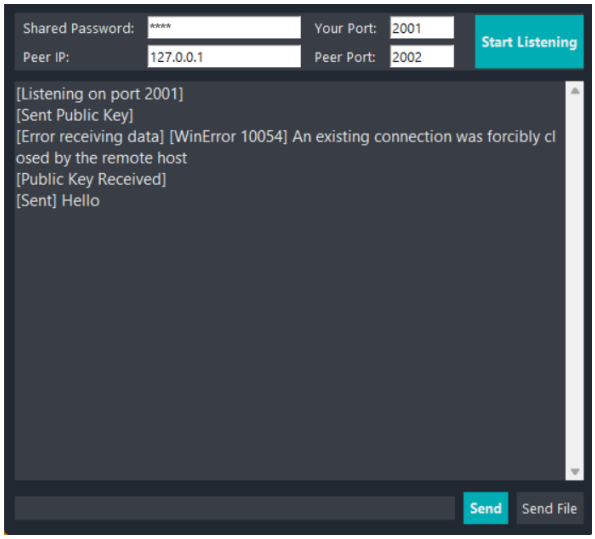
Objective:

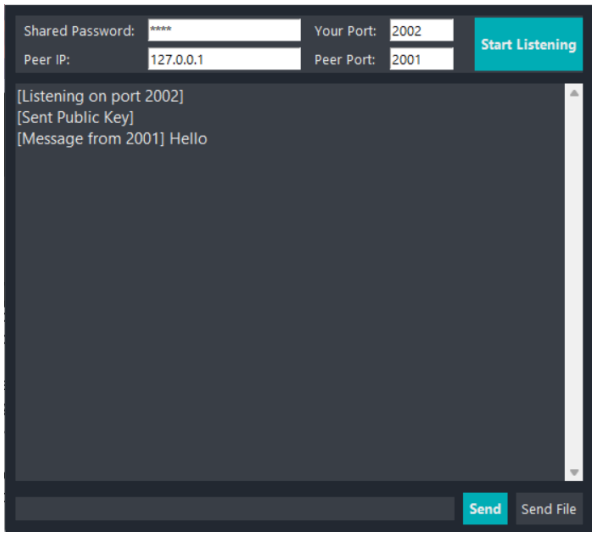
- Transition from a basic Streamlit setup to a robust, standalone Tkinter GUI.
- Implement automatic RSA public key broadcast as part of peer discovery.
- Enable seamless and secure communication (messaging + file transfer) without prior manual key exchange.
- Integrate all cryptographic operations (AES, RSA, HMAC) into a modular, user-friendly GUI.

Execution:

- GUI Migration & Enhancement (Tkinter):
  - Developed a dark-themed GUI with fields for shared password, local/peer ports, and IP.
  - Implemented real-time status logs, file selection dialogs, and button-based operations (send message, send file, key exchange).
  - Ensured threading for background network listeners without blocking the interface.
- Auto-Discovery with RSA Key Broadcast:
  - Modified broadcast_presence() to include the serialized public RSA key in each UDP broadcast.
  - Enhanced listen_for_peers() to parse and auto-import public keys from incoming broadcasts.
  - Retained fallback "Send Public Key" button for manual exchange when needed.
- Secure Communication Integration:

    - Messages/files encrypted with AES-256 in CBC mode using IV + salt.
    - Used HMAC-SHA256 for message and file integrity verification.
    - RSA public key used to securely transmit AES keys for forward secrecy.
    - File transfers split into 1024-byte chunks, each encrypted and verified before reassembly.
- Modularity & UX Improvements:
    - Split backend into independent modules (rsa.py, autodis.py, p2pclient.py, gui.py).
    - Displayed real-time confirmation or error logs for all operations (discovery, encryption, key status).
    - Handled malformed packets, missing keys, and decryption errors gracefully.





Outcomes:

- Achieved automatic RSA public key exchange using network-level discovery alone.
- Allowed clients to communicate securely without manual key input, streamlining the startup process.
- Maintained a fully functional and simple Tkinter-based GUI for all operations (message, file, key exchange).
- Established a more realistic and autonomous peer-to-peer communication system, while still using a beginner-friendly Python GUI.

REFERENCES:

1. *Cryptography and Network Security - Principles and Practice (Stallings William)*
2. *SSL and TLS Essential*
3. *What is Cryptography | why cryptography? Introduction to Cryptography*
4. *https://www.tutorialspoint.com/*

GITHUB LINK:

https://github.com/akshita123454/Cryptography_Internship_Project