This project is a group project, each group will be 2 to 4 students. Each group will deliver a written report in addition to the developed code by the end of the project that shows the detailed analysis, design, testing … etc. of the project and the description of the used techniques, problem description, detailed solution, limitations, sample output, references, and any additional information related to the project. Online Demo (YouTube) Video, presentation, and discussion will be conducted by the end of the project.

Introduction

This coursework is itemized into several parts to get the 35 marks associated to it.
You must use the templates provided by the instructor to prepare your work.
All assignments and projects will be handed-in electronically, while quizzes and exams are written

Marking Criteria

89% and above:

Your work must be of outstanding quality and fully meet the requirements of the coursework specification and learning outcomes stated. You must show independent thinking and apply this to your work showing originality and consideration of key issues. There must be evidence of wider reading on the subject. In addition, your proposed solution should:

- illustrate a professional ability of drafting construction details,
- express a deep understanding of the in-hand problem definition,
- and applying, masterly, the learned knowledge in the proposed solution.

76% - 89%:

Your work must be of good quality and meet the requirements of the coursework specification and learning outcomes stated. You must demonstrate some originality in your work and show this by applying new learning to the key issues of the coursework. There must be evidence of wider reading on the subject. In addition, your proposed solution should:

- illustrate a Good ability of drafting construction details,
- express a very Good understanding of the in-hand problem definition,
- and applying most of the learned knowledge, correctly, in the proposed solution.

67% - 76%:

Your work must be comprehensive and meet all of the requirements stated by the coursework specification and learning outcomes. You must show a good understanding of the key concepts and be able to apply them to solve the problem set by the coursework. There must be enough depth to your work to provide evidence of wider reading. In addition, your proposed solution should:

- illustrate a moderate ability of drafting construction details,
- express a good understanding of the in-hand problem definition,
- and applying most of the learned knowledge, correctly, in the proposed solution.

60% - 67%:

Your work must be of a standard that meets the requirements stated by the coursework specification and learning outcomes. You must show a reasonable level of understanding of the key concepts and principles and you must have applied this knowledge to the coursework problem. There should be some evidence of wider reading. In addition, your proposed solution should:

- illustrate a fair ability of drafting construction details,
- express a fair understanding of the in-hand problem definition,
- and applying some of the learned knowledge, correctly, in the proposed solution.

Below 60%:

Your work is of poor quality and does not meet the requirements stated by the coursework specification and learning outcomes. There is a lack of understanding of key concepts and knowledge and no evidence of wider reading. In addition, your proposed solution would be:

- Illustrate an inability of drafting construction details,
- Failed to define the parameters, limitations, and offerings of the in-hand problem,
- Failed to apply correctly the learned knowledge for proposing a valid solution.

Academic Misconduct

The University defines Academic Misconduct as 'any case of deliberate, premeditated cheating, collusion, plagiarism or falsification of information, in an attempt to deceive and gain an unfair advantage in assessment'. This includes attempting to gain marks as part of a team without making a contribution. The department takes Academic Misconduct very seriously and any suspected cases will be investigated through the University's standard policy. If you are found guilty, you may be expelled from the University with no award.

It is your responsibility to ensure that you understand what constitutes Academic Misconduct and to ensure that you do not break the rules. If you are unclear about what is required, please ask.

Title: CipherShare: A Secure Distributed File Sharing Platform

1. Project Description:

Project Title: CipherShare: A Secure Distributed File Sharing Platform with User-Centric Credential Management

Project Goal: To design and implement a distributed file sharing platform that prioritizes security and user control over their credentials. CipherShare will enable users to securely share files in a peer-to-peer (P2P) network environment, ensuring confidentiality, integrity, and authenticated access. A key focus will be on implementing robust user authentication and secure credential management, empowering users to control their security within the distributed system.

Project Overview: CipherShare will be a P2P file sharing application where users can connect to a distributed network to share files directly with each other. Security will be enforced through several layers:

- **User Authentication:** Users will register and authenticate using strong password-based authentication combined with cryptographic techniques for enhanced security (e.g., salted password hashing, and potentially a challenge-response mechanism).
- **Credential Management:** The system will emphasize secure credential management on the client-side. Users' passwords (or derived keys) will be protected using strong hashing and potentially client-side encryption for storage. Key derivation functions will be used to generate encryption keys from user passwords.
- **File Encryption:** Files will be encrypted before being shared in the network to ensure confidentiality. Symmetric encryption will be used for file content, and asymmetric cryptography can be used for secure key exchange and access control.
- **Integrity Verification:** Hash functions will be used to ensure the integrity of files during transfer and storage, preventing tampering.
- **Distributed P2P Network:** The system will operate as a distributed network where users act as both clients and servers, sharing file chunks and participating in file discovery. A simplified P2P discovery mechanism will be implemented.

Technologies & Concepts Covered:

- **Programming Language:** Python
- **Cryptographic Algorithms:**
  - **Symmetric Ciphers:** AES, ChaCha20 (for file encryption)
  - **Asymmetric Ciphers:** RSA, ECC (for key exchange, access control, potential digital signatures for file ownership - optional)
  - **Hash Functions:** SHA-256, Argon2 (for password hashing and file integrity)
  - **Key Derivation Functions (KDFs):** PBKDF2HMAC, Argon2 (for deriving keys from passwords)
  - **Random Number Generation:** Python's secrets module.
- **Credential Management:** Salted password hashing, secure key derivation, client-side key

storage (encrypted).
- **Distributed Computing Concepts:** Peer-to-Peer network architecture, file chunking and distribution, file discovery (simplified mechanisms - broadcasting or a rudimentary distributed index), node participation.
- **Network Protocols:** TCP/IP (Sockets in Python).
- **File Handling & Chunking:** Techniques for dividing files into chunks for distributed storage and transfer.
- **Agile Development Methodology:** Iterative development, sprint planning, daily stand-ups, reviews, and retrospectives.

Expected Learning Outcomes:

Upon successful completion of this project, students will be able to:

- Design and implement secure user authentication mechanisms, including strong password hashing and potentially challenge-response protocols.
- Apply key derivation functions to securely generate cryptographic keys from user credentials.
- Implement client-side secure credential management and key storage techniques.
- Integrate symmetric and asymmetric encryption for file confidentiality and secure sharing.
- Use hash functions for file integrity verification and potentially for password storage.
- Develop a distributed P2P file sharing application using Python and network programming.
- Understand the challenges and complexities of building secure, distributed systems.
- Enhance their understanding of user-centric security and credential management principles.
- Work collaboratively in teams using Agile development practices.

2. Semi-Formal System Requirements (User Stories):

As a User (Alice):

- As a user, I want to be able to **register an account** with a username and a strong password so that I can securely share files on CipherShare.
- As a user, I want to be able to **log in** to CipherShare using my username and password so that I can access the file sharing network.
- As a user, I want to be able to **securely store my password (or a derived key)** locally so that my credentials are protected.
- As a user, I want to be able to **upload a file** to the CipherShare network so that I can share it with others.
- As a user, I want to be able to **search for files** available on the network based on keywords or filenames so that I can find files I'm interested in.
- As a user, I want to be able to **download a file** from another user in the network so that I can access shared content.
- As a user, I want to **know that my uploaded files are encrypted** in the distributed network so that only authorized users can access them.
- As a user, I want to **control who can access my shared files** (e.g., share with all registered users, or potentially specific users - if complexity allows) so that I can manage file access.

- As a user, I want to be **assured of the integrity of downloaded files** so that I know the files haven't been corrupted or tampered with.
- As a user, I want to be able to **see a list of files I am currently sharing**.
- As a user, I want to be able to **easily manage my shared files** (e.g., stop sharing a file).

Underlying System Requirements (Non-functional):

- The system **must provide strong user authentication** and prevent unauthorized access.
- The system **must ensure secure credential management** on the client-side.
- The system **must ensure confidentiality** of shared files through encryption.
- The system **must ensure integrity** of files during transfer and storage.
- The system **must operate in a distributed peer-to-peer (P2P) manner**, without relying on a central server for file storage (central server for initial discovery might be acceptable to simplify).
- The system **should be modular** to facilitate development and testing.
- The system **should be implemented in Python**, leveraging appropriate libraries.
- The system **should be well-documented**.

Project Plan (Agile):

Phase 1: Basic P2P File Transfer & Unencrypted Sharing (Weeks 1-2)

- **Implemented Part:**
  - Basic P2P network setup: Nodes can connect to each other (potentially via a simple rendezvous server for initial discovery, or using broadcasting within a local network for simplification in early phases).
  - Unencrypted file transfer functionality: Users can select a file to share and another user can download it directly from the sharer.
  - Basic file listing: Display files being "shared" by peers in a rudimentary manner.
  - Initial project setup: Git, documentation structure, team roles.
- **Deliverables:**
  - Working prototype of a basic, unencrypted P2P file sharing application.
  - Rudimentary file listing and discovery mechanism.
  - Codebase in Git.
- **Documentation:**
  - Phase 1 Report: Implemented features, challenges, Phase 2 plan.
  - Initial System Architecture Diagram (basic P2P structure).
  - Basic User Manual (how to run basic file sharing).

Phase 2: User Authentication & Basic Credential Handling (Weeks 3-4)

- **Implemented Part:**
  - User registration and login functionality: Implement user accounts (username/password).
  - Basic password hashing: Use a simple hashing algorithm (SHA-256 for now, upgrade to Argon2 later) to hash passwords for storage (in-memory or a simple file for this phase).
  - User session management: Maintain user sessions (basic in-memory session management for initial phase).
  - Integrate authentication into file sharing: Only logged-in users can share and download files.
- **Deliverables:**
  - P2P file sharing application with user authentication (login/registration).
  - Basic password hashing implemented.
  - User session management.
- **Documentation:**
  - Phase 2 Report: Authentication implementation, credential handling, challenges, Phase 3 plan.
  - Updated System Architecture Diagram (authentication flow).
  - Authentication Design Document (hashing algorithm, session management).

Phase 3: File Encryption & Integrity Verification (Weeks 5-6)

- **Implemented Part:**
  - File encryption on upload: Encrypt files using symmetric encryption (e.g., AES) before sharing.

- File decryption on download: Decrypt downloaded files.
- File integrity verification using hash functions: Generate and verify hashes (e.g., SHA-256) for files to ensure integrity.
- Key management for file encryption: For simplicity, initially use a single symmetric key for all shared files or explore a basic key exchange for each file sharing session (simplified Diffie-Hellman can be considered).

- **Deliverables:**
  - P2P file sharing application with file encryption and decryption.
  - File integrity verification using hashes.
  - Basic key management for file encryption.
- **Documentation:**
  - Phase 3 Report: Encryption and integrity integration, key management, challenges, Phase 4 plan.
  - Updated System Architecture Diagram (encryption/decryption modules).
  - Cryptographic Design Document (cipher choices, hashing, key management strategy).

Phase 4: Enhanced Credential Management & Distributed Features (Weeks 7-8)

- **Implemented Part:**
  - Enhanced Credential Management: Implement strong password hashing using Argon2 or PBKDF2HMAC. Implement key derivation from passwords to generate encryption keys. Explore client-side encrypted storage for user credentials (basic).
  - Improved P2P features: Refine file discovery mechanisms (e.g., simple broadcasting or a very basic distributed index - time permitting). Implement file chunking for larger files (optional based on time). Basic access control based on users (share files with specific users if complexity allows).
  - User Interface improvements (if time permits, command-line interface is acceptable).
  - Thorough testing and bug fixing.
- **Deliverables:**
  - Final CipherShare application with enhanced credential management, file encryption, integrity, and distributed features.
  - Improved user experience features (to the extent possible).
  - Comprehensive test report and bug fixes.
- **Documentation:**
  - Phase 4 Report: Credential management enhancements, distributed feature improvements, testing results, final project reflection.
  - Final System Architecture Document (detailed system overview).
  - Final User Manual.
  - Project Retrospective Document.

4. Skeleton Code of Different Components (Python):

a) crypto_utils.py (Cryptographic Utility Functions):

(This can be largely the same as the previous project's crypto_utils.py, but with additions for Argon2 password hashing and key derivation functions)

```
# ... (Import statements - similar to previous crypto_utils) ...
from cryptography.hazmat.primitives.kdf.argon2 import Argon2id

# ... (Functions for symmetric encryption, decryption, hash_message,
RSA - same as before) ...

def hash_password(password, salt=None):
    if salt is None:
        salt = secrets.token_bytes(16) # Generate new salt if none
provided
    argon2 = Argon2id(
        salt=salt,
        time_cost=16, # Adjust these parameters based on
performance/security trade-off
        memory_cost=65536,
        parallelism=2,
        hash_len=32,
        backend=default_backend()
    )
    hashed_password = argon2.hash(password.encode('utf-8'))
    return hashed_password, salt # Return both hash and salt

def verify_password(password, hashed_password, salt):
    argon2 = Argon2id(
        salt=salt,
        time_cost=16,
        memory_cost=65536,
        parallelism=2,
        hash_len=32,
        backend=default_backend()
    )
    try:
        argon2.verify_hash(hashed_password, password.encode('utf-8'))
        return True # Password is valid
    except: # cryptography.exceptions.InvalidHash
        return False # Password is invalid

def derive_key_from_password(password, salt):
    kdf = PBKDF2HMAC( # Or use Argon2 for key derivation too
```

```
        algorithm=hashes.SHA256(),
        length=32, # Key length for AES-256
        salt=salt,
        iterations=100000, # Adjust iterations for
security/performance
        backend=default_backend()
    )
    key = kdf.derive(password.encode('utf-8'))
    return key

# ... (Potentially functions for secure key storage if you implement
client-side key encryption) ...
```

b) fileshare_client.py (File Share Client Skeleton - P2P Client):

```
import socket
import crypto_utils
import os

# ... (Constants for ports, network addresses, file chunk size etc.)
...

class FileShareClient:
    def __init__(self):
        self.client_socket = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
        self.username = None
        self.session_key = None # For symmetric encryption with peers

    def connect_to_peer(self, peer_address):
        try:
            self.client_socket.connect(peer_address)
            print(f"Connected to peer at {peer_address}")
            return True
        except Exception as e:
            print(f"Error connecting to peer {peer_address}: {e}")
            return False

    def register_user(self, username, password):
        # ... (Implement registration process - send username, hashed
password+salt to a registration service/peer - how to distribute user
info in P2P? - Simplification needed, perhaps a dedicated 'user
registry' peer initially or file-based for simplicity) ...
        # ... (Client-side password hashing and salt generation) ...
        pass
```

```
    def login_user(self, username, password):
        # ... (Implement login process - send username, password -
server/peer authenticates against stored hashed password - handle
session - simplified session management for P2P could be token-based
or direct connection based) ...
        # ... (Client-side password hashing to compare against stored
hash) ...
        pass

    def upload_file(self, filepath):
        # ... (Read file in chunks, encrypt chunks, send chunks to
peer - need to implement P2P file transfer protocol - simplified) ...
        # ... (File encryption using crypto_utils, integrity hash
generation) ...
        pass

    def download_file(self, file_id, destination_path):
        # ... (Request file from peer, receive encrypted chunks,
decrypt chunks, verify integrity, save file) ...
        # ... (File decryption, integrity verification) ...
        pass

    def search_files(self, keyword):
        # ... (Implement file search in the P2P network -
broadcasting? Distributed Index? - Simplification required) ...
        pass

    def list_shared_files(self):
        # ... (Keep track of locally shared files and display them)
...
        pass

    # ... (Methods for P2P message handling, network discovery -
simplified) ...

# ... (Client program entry point, user interface loop) ...
```

c) fileshare_peer.py (File Share Peer Node Skeleton - P2P Node):

```
import socket
import threading
import crypto_utils
import os
# ... (Data structures for user info, shared files, peer lists etc.)
...
```

```
class FileSharePeer:
    def __init__(self, port):
        self.peer_socket = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
        self.port = port
        self.host = '0.0.0.0' # Listen on all interfaces
        self.users = {} # {username: {hashed_password, salt, ...}} -
In-memory for simplicity, consider file-based storage for persistence
        self.shared_files = {} # {file_id: {filepath, owner_username,
...}} - Track files shared by this peer

    def start_peer(self):
        self.peer_socket.bind((self.host, self.port))
        self.peer_socket.listen(5)
        print(f"Peer listening on port {self.port}")

        while True:
            client_socket, client_address = self.peer_socket.accept()
            client_thread =
threading.Thread(target=self.handle_client_connection,
args=(client_socket, client_address))
            client_thread.start()

    def handle_client_connection(self, client_socket, client_address):
        print(f"Accepted connection from {client_address}")
        try:
            while True:
                # ... (Receive commands from client - register, login,
upload, download, search, etc. - define a simple protocol) ...
                command = client_socket.recv(1024).decode() # Example
- define command structure

                if command == "REGISTER":
                    # ... (Handle user registration - receive
username, hashed password+salt, store user info) ...
                    pass
                elif command == "LOGIN":
                    # ... (Handle login - receive username, password,
verify password against stored hash, create session - simplified) ...
                    pass
                elif command == "UPLOAD":
                    # ... (Receive file metadata, then encrypted file
chunks, store chunks, update shared_files list) ...
                    pass
                elif command == "DOWNLOAD":
                    # ... (Receive file ID, retrieve encrypted file
```

```
chunks, send chunks to requesting client) ...
                    pass
                elif command == "SEARCH":
                    # ... (Receive search keyword, search local shared
files, respond with file list - for simplified P2P search) ...
                    pass
                # ... (Handle other commands) ...

        except Exception as e:
            print(f"Error handling client {client_address}: {e}")
        finally:
            client_socket.close()

    # ... (Methods for user registration, login, file upload,
download, search, P2P network functions) ...

# ... (Peer program entry point - start the peer node) ...
```

Notes on Code Skeletons:

- **P2P Simplification:** Implementing a fully robust P2P network in 8 weeks is challenging. The skeleton suggests simplification:
  - **Centralized User Registry (Optional Simplification for early phases):** Initially, a designated "registry peer" or even a simple file can store user account information for easier registration and login management. Later, consider distributed user management concepts if time permits.
  - **Simplified Discovery:** Basic broadcasting for file search or a rudimentary distributed index (maybe a shared list of file metadata amongst peers) can be considered instead of a full DHT implementation.
  - **Direct Peer Connections:** Focus on direct client-to-peer connections for file transfer, without complex routing in early stages.
- **Security Caveats:** Again, these skeletons are for educational purposes. Real-world secure P2P file sharing is complex and requires careful consideration of many security aspects. Students should be aware of the simplified nature of this project and the many deeper security challenges in a true distributed environment.