

3

Table of Contents

[Phase 3 Overview 3](#_Toc197098962)

[System Architecture Overview 3](#_Toc197098963)

[Challenges and Limitations Faced 3](#_Toc197098964)

[Implemented Features 4](#_Toc197098965)

[1. File Encryption/Decryption 4](#_Toc197098966)

[2. Integrity Verification 4](#_Toc197098967)

[3. Key Management 5](#_Toc197098968)

[Protocol Flow 5](#_Toc197098969)

[File Download Sequence 5](#_Toc197098970)

[File Upload Sequence 6](#_Toc197098971)

[Cryptographic Design Document 6](#_Toc197098972)

[Cipher Choices 6](#_Toc197098973)

[Architecture Diagram: 8](#_Toc197098974)

[Phase 4 Implementation Plan 8](#_Toc197098975)

[1. Enhanced Credential Management 8](#_Toc197098976)

[2. Credential Storage: 8](#_Toc197098977)

[3. Advanced P2P Features: 8](#_Toc197098978)

[4. Testing Strategy 9](#_Toc197098979)

# Phase 3 Overview

In Phase 3 of the CipherShare project, we enhanced the security of file transfers by implementing confidentiality and integrity mechanisms. Files shared between peers are now encrypted before transmission and verified upon receipt, aligning with the project specifications for secure distributed file sharing.

# Updated System Architecture Overview

The system now includes an encryption/decryption layer that wraps all file transfer operations.

The P2P file sharing system consists of three main components:

1. **Rendezvous Server**: Central peer discovery service
2. **Peer Nodes**: Each acting as both client and server
   1. **Peer/Client**: Encrypts data before sending and decrypts after receiving data (Using of course the cryptographic modules we created in this phase).
   2. **Peer/Server**: Transmits ciphertext without accessing plaintext.
3. **Cryptographic Modules**: Handling encryption and integrity verification

A diagram of a server

Description automatically generated

# Challenges and Limitations Faced

* Ensuring correct handling of ciphertext when header and data arrive together.
* Resolving intermittent WinError 10053 caused by mismatched socket reads.
* Single-key approach simplifies management but poses a compromise risk if key exposed.

# Implemented Features

## 1. File Encryption/Decryption

**Algorithm**: AES-256-GCM (Galois/Counter Mode)

* Provides both confidentiality and authenticity
* Nonce: 12 bytes randomly generated for each encryption
* Key: 32 bytes (256-bit) shared symmetric key

**Implementation (crypto\_utils.py):**

def encrypt(plaintext: bytes, key: bytes) -> bytes:

    aesgcm = AESGCM(key)

    nonce = os.urandom(12)  # 96-bit

    ct = aesgcm.encrypt(nonce, plaintext, None)

    return nonce + ct  # prefix nonce

def decrypt(data: bytes, key: bytes) -> bytes:

    aesgcm = AESGCM(key)

    nonce = data[:12]

    ct = data[12:]

    return aesgcm.decrypt(nonce, ct, None)

## 2. Integrity Verification

**Method**: SHA-256 hashing

* Hash generated before encryption
* Verified after decryption
* Stored alongside encrypted files

**Example Verification Flow**:

Before upload:

plaintext = file.read()

file\_hash = hashlib.sha256(plaintext).hexdigest()

ciphertext = encrypt(plaintext + file\_hash.encode(), KEY)

After download:

decrypted = decrypt(ciphertext, KEY)

received\_data, received\_hash = decrypted[:-64], decrypted[-64:].decode()

computed\_hash = hashlib.sha256(received\_data).hexdigest()

assert computed\_hash == received\_hash

## 3. Key Management

**Current Implementation**:

* Single shared symmetric key (KEY)
* Stored in KEY\_FILE (default: user\_data/key.bin)
* Generated once if not exists
* All peers use same key for simplicity in Phase 3

**Key Generation**:

def load\_key():

    try:

        with open(KEY\_FILE, "rb") as f:

            return f.read()

    except FileNotFoundError:

        key = os.urandom(32)  # 256-bit key

        os.makedirs(os.path.dirname(KEY\_FILE), exist\_ok=True)

        with open(KEY\_FILE, "wb") as f:

            f.write(key)

        return key

# Protocol Flow

## File Download Sequence

1. Client → Server: {token} DOWNLOAD {filename}
2. Server:
   * Verifies token
   * Reads and encrypts file
   * Sends: SIZE:{filesize}\n
3. Client → Server: READY
4. Server → Client: Encrypted file chunks
5. Client:
   * Receives all chunks
   * Decrypts file
   * Verifies integrity

## File Upload Sequence

1. Client → Server: {token} UPLOAD {filename}
2. Server → Client: READY
3. Client → Server: {filesize}\n + encrypted chunks
4. Server:
   * Receives complete file
   * Decrypts and verifies
   * Sends confirmation

# Cryptographic Design Document

## Cipher Choices

**AES-256-GCM** was selected because:

* Provides both confidentiality and authenticity
* Standardized and widely audited
* Hardware acceleration available
* Nonce+tag overhead minimal (28 bytes total)

**SHA-256** was selected for integrity because:

* Collision-resistant
* Fixed output size (32 bytes)
* Fast computation

**Key Management Strategy (current):**

* Single pre-shared key
* Stored in protected file
* All peers must have same key
* Future (phase 4):

A diagram of a client

Description automatically generated

# Architecture Diagram:

A diagram of a computer server

Description automatically generated

# Phase 4 Implementation Plan

## 1. Enhanced Credential Management

**Password Security Upgrade**:

* Replace SHA-256 with **Argon2id** for password hashing

**Key Derivation**:

* Implement PBKDF2HMAC for encryption key generation from passwords

## 2. Credential Storage:

* Client-side encrypted credential vault using AES-256-GCM
* Master password-derived key for vault decryption

## 3. Advanced P2P Features:

* File chunking for large files
* Basic access control lists
* Distributed file indexing

## 4. Testing Strategy

1. **Unit Tests**:
   * Encryption/decryption cycle verification
   * Hash generation/verification
   * Protocol message validation
2. **Integration Tests**:
   * Full file transfer with verification
   * Concurrent transfer stress testing
   * Error condition handling
3. **Security Tests**:
   * Tampered file detection
   * Replay attack prevention
   * Key confidentiality verification
   * MITM attack scenarios
4. **Performance Testing**:

* Transfer speeds with/without encryption
* Network overhead measurements
* Memory usage profiling