ECE 422 – Group Project: Secure File System

Group Name: The ByteKnights

Name	SID	CCID
Parth Dadhania	1722612	pdadhani
Het Bharatkumar Patel	1742431	hetbhara

Abstract

This final report documents the complete design, implementation, and evaluation of the Secure File System (SFS) project. Building on the foundational design presented in the deliverable, the final SFS is a robust, multi-user file system that ensures confidentiality, integrity, and controlled access even on an untrusted file server. The system employs strong authentication and secure communication mechanisms that prevent the transmission of plain-text passwords and safeguard stored credentials via *bcrypt* hashing. Data encryption is enforced using symmetric key *cryptography*, with careful management of encryption keys to protect both file contents and metadata. Additionally, the SFS integrates integrity verification through *HMAC*, enabling immediate detection and notification of any unauthorized file modifications upon user login. A Unix-like permission model, incorporating owner, group, and other access levels, supports secure multi-user collaboration. This report details the technologies, methodologies, and architectural decisions that underpin the system, along with comprehensive user interaction scenarios and a demonstration of key functionalities as outlined in the grading criteria.

Introduction

In today's interconnected world, securing data stored on untrusted servers is more critical than ever. Traditional file systems rely on basic operating system permissions, which are often inadequate against modern threats. The Secure File System (SFS) project addresses this gap by integrating robust cryptographic techniques and access control mechanisms to ensure data confidentiality, integrity, and controlled sharing among multiple internal users. SFS is designed to protect sensitive information even when hosted on potentially compromised external servers. By encrypting file names, directory structures, and file contents, the system prevents unauthorized parties from accessing or inferring critical data. Integrity is maintained through HMAC-based checks, which enable the system to detect any unauthorized modifications and promptly alert file owners. This feature is especially important in environments where data tampering is a significant risk.

At the core of SFS is a multi-layered security approach. User authentication is handled securely using *bcrypt* hashing, ensuring that no plain-text passwords or unsecured communications are used. Furthermore, SFS adopts a Unix-inspired permission model, allowing for granular control over file access through owner, group, and other modes. This design supports secure collaboration while ensuring that users cannot alter permissions or access files without proper authorization.

The command-line interface (CLI) of SFS offers a familiar environment with support for essential file operations such as *pwd, ls, cd, touch, cat, echo, mv, and chmod*. This integration of traditional file system commands with advanced security measures not only simplifies user interaction but also demonstrates the practicality of applying rigorous security standards in everyday operations.

Overall, the SFS project addresses three fundamental security pillars:

- **Confidentiality**: through comprehensive encryption of file names, contents, and directory structures.
- **Integrity**: by implementing cryptographic checks to detect and alert on unauthorized modifications.
- **Controlled Access**: via a robust authentication process and Unix-like permissions, ensuring secure multi-user collaboration.

This final report expands on these core concepts by detailing the architectural decisions, encryption key management strategies, and the implementation of secure metadata and permission handling. It also illustrates how these components work together to provide a scalable and secure solution for storing data on an untrusted file server.

Detailed Design

1. Describe your authentication, encryption, and access permissions design and implementation

1.1 Authentication

The Secure File System (SFS) implements user authentication to ensure that only authorized users gain access to the system. Passwords are securely managed using the *bcrypt* hashing algorithm, chosen for its resistance against brute-force and dictionary attacks. Each password is salted and hashed, ensuring robust storage practices and preventing plain-text password vulnerabilities.

The system introduces an administrative role (admin) to manage user creation and group assignments. Authentication flow involves verifying hashed passwords during login, ensuring secure user verification without transmitting or storing plaintext passwords. The authentication logic resides primarily within the *AuthManager* class (*authentication.py*), which provides methods for creating users, verifying credentials, and managing user-group relationships securely.

1.2 Encryption

Data confidentiality within the SFS is maintained using symmetric encryption provided by the *cryptography* library, specifically leveraging the *Fernet* implementation. Fernet is an authenticated encryption algorithm based on *AES-128 in CBC mode with HMAC-based* integrity verification, offering a robust balance between security and performance.

- **Encryption Key Management:**
  - A single symmetric encryption key is generated using Fernet and securely stored within a dedicated binary file *(sfs_key.bin)* to maintain consistency across sessions. The key file resides within the file system's encrypted directory structure, ensuring restricted access to the key.
  - Upon each system initialization, the encryption key is loaded or created once, as implemented in *sfs_main.py*, minimizing the risk of key exposure.
- **Encrypted Data Storage:**

- Both file contents and file/directory names are encrypted, ensuring that external users accessing the underlying file system storage cannot discern sensitive information. The encryption and decryption operations are encapsulated in the *encryption.py* module.
- The system uses *base64* encoding to safely handle encrypted binary data within file system metadata.

1.3 Integrity Checks

Integrity protection is implemented through cryptographic *HMAC* (Hash-based Message Authentication Codes) using *SHA-256*, ensuring that any unauthorized modification to encrypted data or metadata is detectable. The integrity logic is defined within the *integrity.py* module.

- Upon file creation and modification, HMACs are generated and stored alongside encrypted data and metadata.
- At login, an integrity verification procedure checks for HMAC mismatches or unauthorized changes. Any detected integrity breach triggers immediate user notifications, detailing corrupted or altered files.

1.4 Access Permissions

SFS utilizes a Unix-like permission model, implemented in the *permission.py* module, allowing fine-grained access control at the file and directory level.

- **Permission Levels:**
  - **User**: Default permission where only the owner can access files/directories.
  - **Group**: Access extended to all users within the owner's group.
  - **All**: Universal access to internal system users.
- **Implementation:**
  - Permission checks (read and write) are performed before executing any file operations, ensuring strict enforcement of access rules.
  - Only file owners are permitted to modify permissions, ensuring secure permission management and avoiding unauthorized access changes.

1.5 Metadata Management

Metadata about files and directories—including permissions, ownership, encrypted file/directory names, and associated HMACs—is securely stored as encrypted JSON files within respective directory structures.

- Metadata encryption and storage are managed by the *disk_storage.py* module, ensuring confidentiality and integrity of metadata against unauthorized access.
- During file operations, metadata integrity is consistently verified, and any anomalies prompt immediate user notifications, further strengthening the security posture.

Through this integrated approach, the Secure File System robustly addresses authentication, encryption, and access permission requirements, ensuring secure operation in untrusted environments.

2. Technologies, Methodologies, and Tools

2.1 Programming Languages & Frameworks

- **Python**:
  - Chosen for its simplicity, readability, and extensive ecosystem of libraries.
  - Facilitates rapid and flexible development, particularly suited for command-line interfaces.
- **Cryptography Libraries** (*cryptography* and *bcrypt*):
  - The *cryptography* library provides robust, well-documented cryptographic primitives (*AES* encryption and *HMAC*) crucial for ensuring data confidentiality and integrity.
  - *bcrypt* was selected for secure password hashing due to its built-in salting and resilience against brute-force attacks, significantly enhancing credential security.
- **Data Storage** (Encrypted JSON):
  - Lightweight and easily integrable with Python, JSON was utilized to store encrypted metadata, user and group information, file permissions, and encryption keys securely.
  - Encryption at rest prevents unauthorized access and maintains data confidentiality.

2.2 Methodologies

- **Agile** (Incremental Approach):
  - The Agile methodology was adopted to facilitate iterative development, focusing on incremental feature implementation (authentication, encryption, access control).
  - Regular reviews and testing cycles allowed for continuous improvement and quick adaptability to evolving requirements.
- **Test-Driven Development** (TDD):
  - Core functionalities such as encryption, authentication, and integrity checks were developed and tested incrementally, enabling early bug detection and system reliability.
- **Documentation-Driven Development:**
  - Design diagrams (architecture, class, sequence diagrams) were continuously refined throughout development, ensuring alignment between the design and implemented system.

2.3 Tools

- **Version Control** (Git & GitHub):

- Git and GitHub facilitated collaborative development, efficient code management, and effective code review processes, enhancing overall software quality.

By strategically combining these technologies, methodologies, and tools, we successfully delivered a robust, secure, and maintainable Secure File System (SFS) that meets the core project objectives regarding confidentiality, integrity, and controlled access.

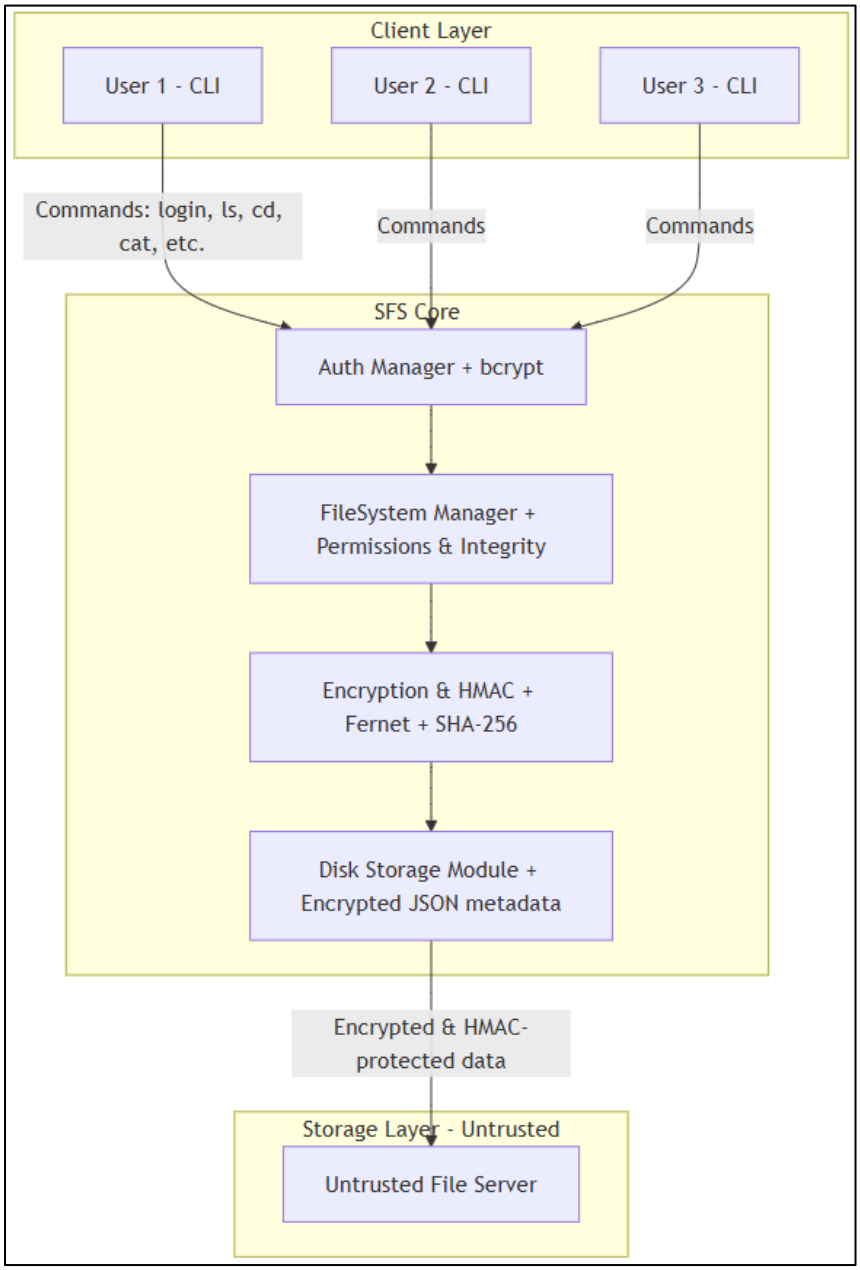

Figure 1. High-Level Architecture of the Secure File System (SFS) – Showcasing user interaction, authentication, encryption, access control, and secure storage.

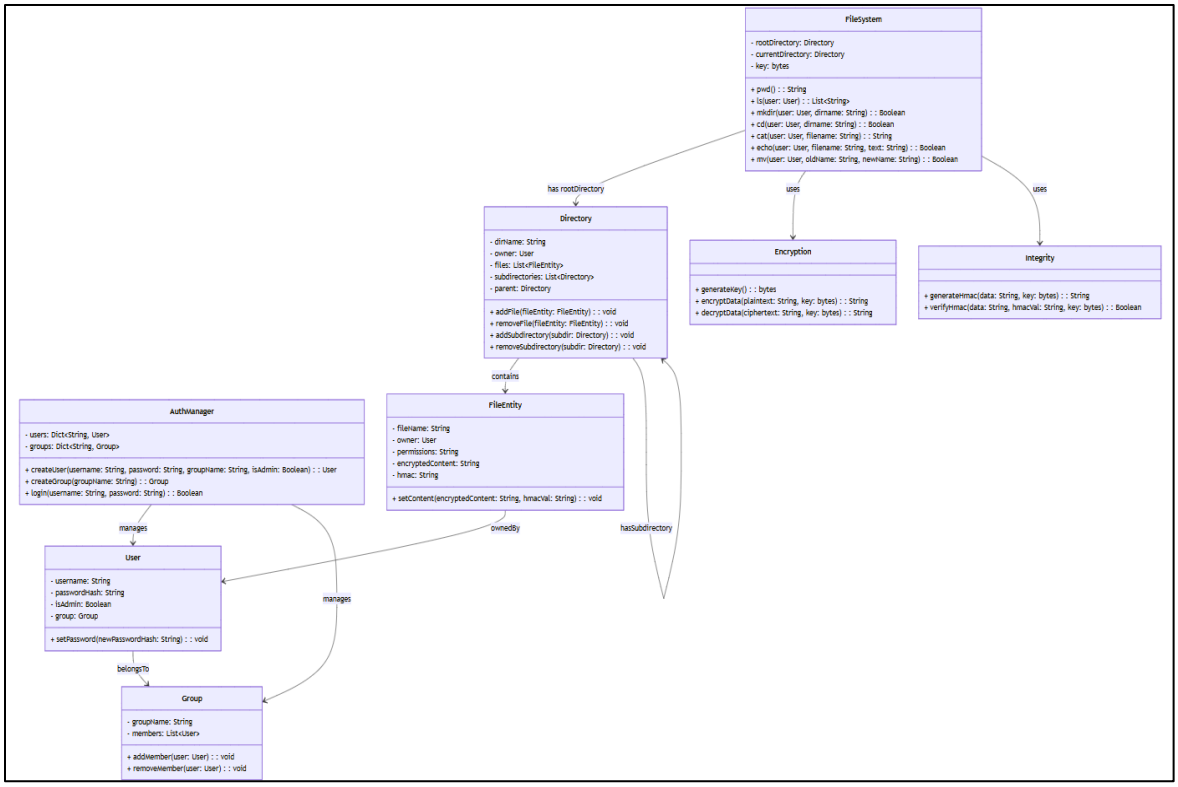

Figure 2. UML Class Diagram of the Secure File System (SFS) – Showcasing key classes, attributes, methods, and relationships, including user authentication, encryption, access control, and hierarchical file management.

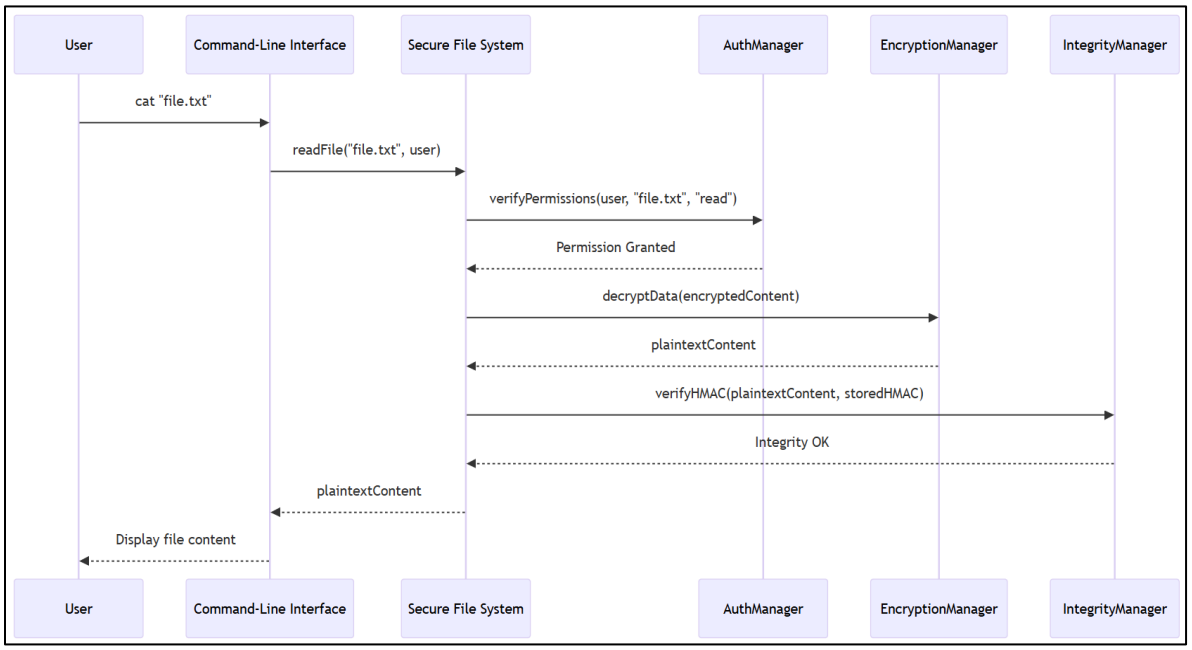

Figure 3. Sequence Diagram for File Read Operation in Secure File System (SFS) – Illustrates step-by-step interactions for authentication, permission verification, decryption, and integrity checks before displaying file content.

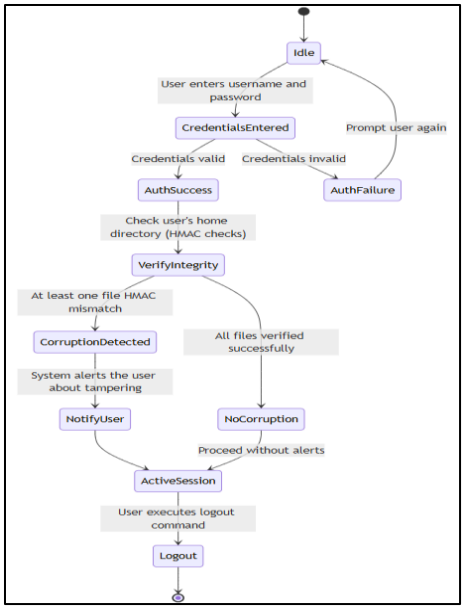

Figure 4. State machine diagram illustrating the user login process in the Secure File System (SFS), including authentication, integrity verification via HMAC, and session transitions based on corruption detection.

User Stories

1. User Account Creation

User Story: As a team lead or system admin, I want to create a user member so that we can collaborate on files under shared permissions.

Acceptance Criteria

1. Only an admin can create a user.
2. The system prompts for a unique username and secure password (hashed, not stored in plaintext).
3. If the username is already taken, the system notifies the admin and requests a different one.
4. Upon successful creation, the user should be able to log in and access a personal home directory.

2. Group Management

User Story: As a team lead or system admin, I want to create a group and add/remove members so that we can collaborate on files under shared permissions.

Acceptance Criteria

1. Only an admin can create a group.
2. The system prevents duplicate group names and returns an error if a group name already exists.
3. Group members can read/write files if granted appropriate permissions, and non-members cannot.

3. File Operations

User Story: As an authenticated user, I want to create, read, write, and rename files so that I can manage my data within the secure file system.

Acceptance Criteria

1. The system encrypts filenames and contents on disk automatically.
2. When I create a file, I become its owner and can set initial permissions (owner/group/all).
3. If I rename a file, the system updates the encrypted filename accordingly, preserving its contents and metadata.
4. Unauthorized users cannot read or modify my files based on permission checks.

4. File Integrity & Tamper Detection

User Story: As a file owner, I want to be notified if my files are tampered with so that I can take action and maintain data integrity.

Acceptance Criteria

1. The system generates an HMAC for each file and verifies it upon login or file access.
2. If an external user modifies a file's contents outside of SFS, the system detects the mismatch and alerts the owner immediately.
3. The alert includes which file(s) were tampered with, and the file remains in a corrupted state until the owner takes manual corrective actions (e.g., restoring from a backup).

5. Encrypted Visibility for External Users

User Story: As an external (untrusted) user, I want to see only encrypted data so that I cannot read or meaningfully modify SFS files.

Acceptance Criteria

1. Any file or directory names appear garbled or encrypted to me if I am not authenticated as an internal user.
2. The file contents remain unreadable when accessed via conventional OS tools.
3. If I modify an encrypted file directly, the system detects the tampering on the next login by the file's owner.

6. File Permission Modification

User Story: As a file owner, I want to modify file or directory permissions (e.g., *chmod*) to determine whether only I, my group, or all internal users can read/write a resource.

Acceptance Criteria:

1. Only the file/directory owner can change its permission.
2. The system supports three permission modes: user (owner), group (members of the owner's group), and all (all authenticated users).

3. If a user attempts to change permissions on a file/directory they do not own, the operation is denied.
4. Updated permissions are securely stored in the file's metadata, and subsequent accesses honor the new permission setting.

Deployment instructions

Follow the steps below to set up and run the Secure File System (SFS). The process uses a Bash script (run.sh) that automates much of the virtual environment creation, dependency installation, and execution.

1. **Clone the Repository**
   - Obtain the project's source code by cloning the private GitHub repository:
     `git clone https://github.com/PrthD/Secure-File-System.git`
   - Navigate to the project directory:
     `cd Secure-File-System`
2. **Make run.sh Executable**
   - Ensure the run.sh script is marked as executable on Unix-like systems:
     `chmod +x run.sh`
3. **Run the Setup and Launch**
   - For the unified SFS interface, use:
     `./run.sh`

User guide

This guide provides a concise overview of how to interact with the Secure File System (SFS) using its command-line interface (CLI). All commands are entered at the SFS prompt (e.g., SFS>).

1. **Launching SFS**
   - Run the SFS using the run.sh script (e.g., ./run.sh main) to start a single interactive session.
   - After startup, you will see the SFS prompt:
     ```
     Welcome to the Secure File System. Type 'help' for commands,
     'exit' to quit.
     SFS>
     ```
2. **Authentication**
   - `login <username> <password>`
     - Logs in an existing user.
   - `logout`
     - Logs out the current user.

3. **Basic Navigation**
   - `pwd`
     - Prints the current working directory.
   - `ls`
     - Lists files and subdirectories visible to the current user.
   - `cd <directory>`
     - Changes the working directory to `<directory>`.
     - Use `cd ..` to move to the parent directory.
4. **Directory Management**
   - `mkdir <dirname>`
     - Creates a new subdirectory named `<dirname>` in the current directory.
   - `rmdir <dirname>`
     - Removes an empty subdirectory named `<dirname>` (fails if it contains files or further subdirectories).
5. **File Operations**
   - `touch <filename>`
     - Creates an empty file named `<filename>` in the current directory.
   - `rm <filename>`
     - Removes the specified file from the current directory.
   - `cat <filename>`
     - Displays the contents of `<filename>` if the user has read permission.
   - `echo <filename> <text>`
     - Appends `<text> to <filename>`. Overwrites existing content if the file is already open for writing.
   - `mv <old_name> <new_name>`
     - Renames a file or directory from `<old_name> to <new_name>`.
6. **Permissions and Groups**
   - `chmod <file_or_dir> <user|group|all>`
     - Changes the permission of `<file_or_dir>` to one of the modes:
       - user: Only the owner can read/write.
       - group: Owner and group members can read/write.
       - all: Any logged-in user can read/write.
   - `addgroup <groupname>` (admin only)
     - Creates a new user group.
   - `adduser <username> <password> <groupname>` (admin only)
     - Creates a user, assigns them a secure password, and places them in `<groupname>`.
7. **Integrity Checks**
   - Upon logging in, SFS verifies integrity (HMAC) for files in the user's home directory. If tampering is detected, the user is immediately notified.

8. **Help and Exit**
    - `help`
        - Displays the list of available commands.
    - `exit`
        - Quits the SFS application and returns to your system shell.

Tips

- **Permission Required:** Ensure you have the correct permission level (owner, group, or all) to read or write a file.
- **Admin Functions:** Group and user creation are restricted to the admin user.
- **Encrypted Storage:** All file names and contents are stored in encrypted form on disk, with HMACs to detect unauthorized alterations.
- **Corruption Alerts:** If files are modified outside SFS, you will be alerted at your next login, and the tampered file(s) will be flagged as corrupted.

By following these commands and guidelines, you can securely manage your files and directories within the SFS, confident that your data remains encrypted, integrity-checked, and accessible only to authorized users.

Conclusion

The Secure File System (SFS) presented in this final report demonstrates a robust approach to safeguarding data in untrusted environments through encryption, authentication, and Unix-like permission controls. By integrating cryptographic primitives (*Fernet* encryption and *HMAC* for integrity) with *bcrypt-based* authentication, SFS ensures that both credentials and stored data remain secure against unauthorized access or tampering. Additionally, the system's modular design facilitates straightforward maintenance and extensibility, allowing future enhancements to include advanced features like audit logging or encrypted backups. Overall, this project highlights the synergy between reliable cryptographic practices and classic access control models, affirming that security need not be sacrificed for usability and collaboration.

References

1. **Course Notes & Professor Guidance**
   - Lecture materials and guidance from the course instructor, providing theoretical foundations for file systems security and cryptographic best practices.
2. **Slack Project Channel**
   - Team discussions, code reviews, and peer support for troubleshooting development issues.
3. **Stack Overflow**
   - Quick references for Python-specific questions, cryptography library usage, and debugging tips.
4. **GeeksforGeeks**
   - General coding examples and in-depth articles on data structures, algorithms, and Python functionalities.
5. **Online Resources (Websites & Articles)**
   - Various blog posts and official documentation pages for in-depth understanding of encryption algorithms, HMAC usage, and best practices for secure password handling.
6. **ChatGPT**
   - Assisted with clarifications on code snippets, troubleshooting, and the composition of certain report sections as a lookup and drafting resource.