ECE 422

Project 1: Secure File System

March 10, 2021

Benjamin Ripka	1465622
Johnas Wong	1529241

1 Abstract

Secure file systems are what give consumers peace of mind, by guaranteeing that the files they choose to store are safe from external attackers or internal users without access privileges. They're confident that their files are only visible to those they know and trust, and that the integrity of their data is of the utmost importance. These secure file systems employ encryption to ensure the privacy of their users' data, employing a myriad of various encryption techniques to thwart attackers. The security and privacy of secure file systems are what attract so many people to use these services to protect their data.

2 Introduction

In this project we attempt to implement a secure file system that can encrypt and store users data by using encryption keys to encrypt and decrypt the data. We employed a client server architecture, where the server is continuously running on a Cloud virtual machine, constantly waiting for commands sent by the client. The client mimics a terminal shell, in which standard operating system operations can be performed, such as those outlined in the project outline. In reality, the client sends these commands to the server, which then performs the actions and reports the results back to the client. The system behaves in a conservative manner, keeping all files and file names fully encrypted on the servers disk, and only selectively decrypting directory names, file names, and file contents if the requesting user has access permissions for that particular asset.

3 Tools and Methodologies

We implemented both the client and server in the Go programming language, leveraging tools and libraries such as the standard Go 'http' and 'crypto' libraries for communication and encryption. We chose Go for its built-in concurrency features, strong typing, and good community documentation and support. Also, neither of us had used Go before, so it seemed a good opportunity to try something new while exploring concepts in the security space.

For storage of user and permission data in the filesystem server, we implemented an SQLite database with several tables. The choice of SQLite as the database management system was made for it's relative simplicity, open source nature, wide adoptance in the software community, and ease of installation. Interactions between our Go code and the database were simplified by leveraging the SQLX database connection and querying library, written in native Go code. SQLX was chosen because it provides intuitive methods for accessing and modifying data, has a limited number of external dependencies, and is a trusted and open source project.

The actual user files are stored within each user's allocated folder, in a shared ‘sfs’ parent directory directly on the servers underlying file system. In this way we were able to leverage the Go ‘os’ package to handle interaction with the underlying Ubuntu file system and disks, and avoid needing to implement such low level operations manually.

4 System Design

The figure below outlines the high level system design of the SFS, with components enumerated to help with the following discussion. In this section we will discuss each component in detail and touch on how they were designed with security in mind, and how they could be improved. Please refer back to this image at any time for context. Note that the server runs in a Cyber Cloud virtual machine running Ubuntu, and the client runs on the users machine.

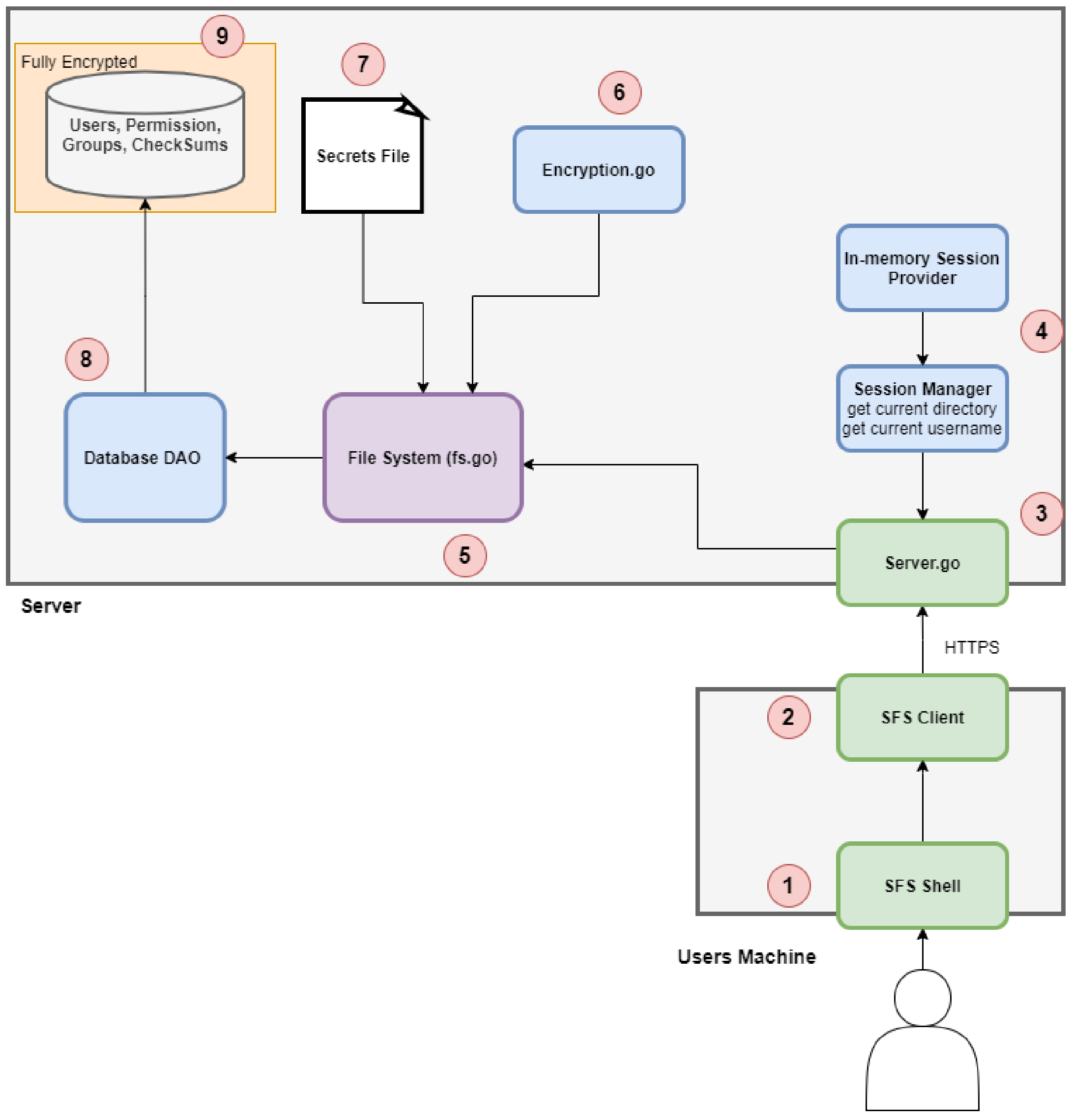

figure 1

4.1 SFS Shell

The SFS Shell component is implemented in a single go file, and holds all of the logic and functionality necessary to interpret, validate and execute user commands. The execution of user commands is done by calling methods on the **SFS Client**. The shell is simply an infinite loop that accepts, parses, validates, and responds to a line of input. When the shell identifies the desired operation and arguments, they are passed to the SFS client for execution.

4.2 SFS Client

The SFS Client is simply a wrapper around the SFS API. It is capable of interacting with all of the REST API's endpoints, managing authentication states in the client, and formatting requests, and cookies. Part of the communication flow involves passing a cookie containing a session ID in each request after the client is authenticated. The client helps make this possible. The client is also responsible for the first layer of username and password encryption. The designs state that the communication should be implemented over HTTPS such that all in-transit data is secured with TLS. This is essential so the session id cannot be hijacked. Unfortunately we were only able to implement it in HTTP with the time constraints, which is a large security flaw that should be fixed in the future.

4.3 Server

The server component of the system is responsible for coordinating the communication between the file system and any connected client applications. It contains several HTTP endpoints, one for each command that the file system supports, plus several endpoints for administrative operations such as signing up users and logging them in. The server and client communication is organized in a session token pattern. This communication protocol is described in *figure 2.*

First, the client application makes a POST request to the login endpoint with an HTTP body containing the users encrypted username and password. The user is then authenticated using the passed credentials, and returned a 200 response request also containing an HTTP cookie containing a generated session ID. The client can then use that sessionID in future requests, bypassing the need for authentication. Finally, after one (1) hour, the server deems the session expired, and the client prompts the user to reauthenticate.

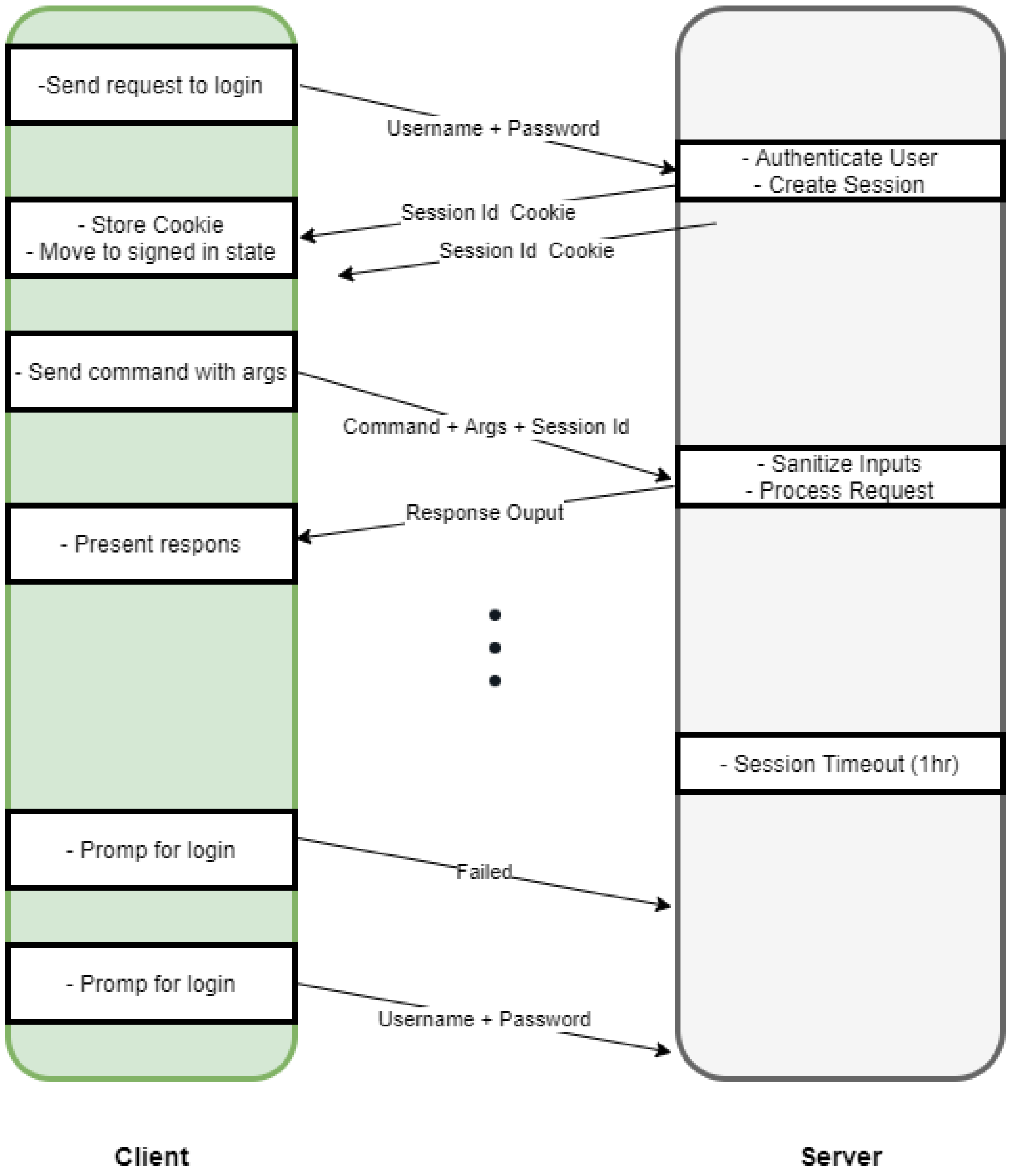

figure 2

This model presents several security vulnerabilities including **session hijacking.** This form of attack however can be mostly mitigated by countermeasures like setting the HTTP header HTTP_ONLY to true, which prevents access to the cookie by any parties not at the destination address. Also, using HTTPS would ensure that any of the data being passed is fully encrypted.

By combining the session data provided by the **session provider** (see next section) with the command arguments passed by the client through HTTP requests, the server is able to properly call functions of the **file system** and report their results back to the client.

4.4 Session Manager and Provider

Sessions have certain data associated with them that must be referenced and used for executing operations requested by users that are signed in. That data includes username and current working directory of the user in the session, and is stored within the **session provider**. Access to the session provider for a specific user is controlled by the **session manager**. The session manager is defined within its own package and used within the server as a singleton, protected by a mutex for access serialization. It contains methods for getting the session provider for an associated session ID, garbage collecting old session providers, and creating new session providers for new sessions. New sessions are allocated a session object and a random base64 encoded hash to serve as it’s session ID. The session provider contains methods for setting and getting key value pairs defining data relevant for the session. As mentioned earlier, this data includes username and current working directory of the session. The session provider is effectively an in-memory cache for data related to the session. [3]

The alternative to storing session data in an in-memory cache would have been to have the client pass the current directory and username on every request, however that introduces vulnerability to attacks in which the client modifies its current working directory or username dynamically during a session, in an attempt to gain access to files for which it has no permission. By keeping the state of the session completely internal, the number of attack vectors is reduced substantially.

4.5 File System (FS)

The file system is a seperate Go package that consists of a set of functions, each representing some operation of the file system that can be executed using the local operating system. By implementing the FS as a separate package, we were able to decouple the file system logic from the communication logic of the server, or the database communication logic of the DAO. Each function in general accepts the username of the current session owner, and the current directory. In this way, the file system package was able to be implemented as a stateless, functional way, reducing the complexity and thus chance of bug introduction. Using the combination of username, current directory and command arguments, the FS is able to verify user permissions (using the DB) and execute the desired commands using the local os Go package. The filesystem layer also enforces encryption of all files, directories, usernames and passwords using the **encryption** package BEFORE they are stored in the database. The FS functions will return errors if permission is denied or something goes wrong, which can then be caught and handled by the server, promoting a consistent error reporting / messaging interface for the client.

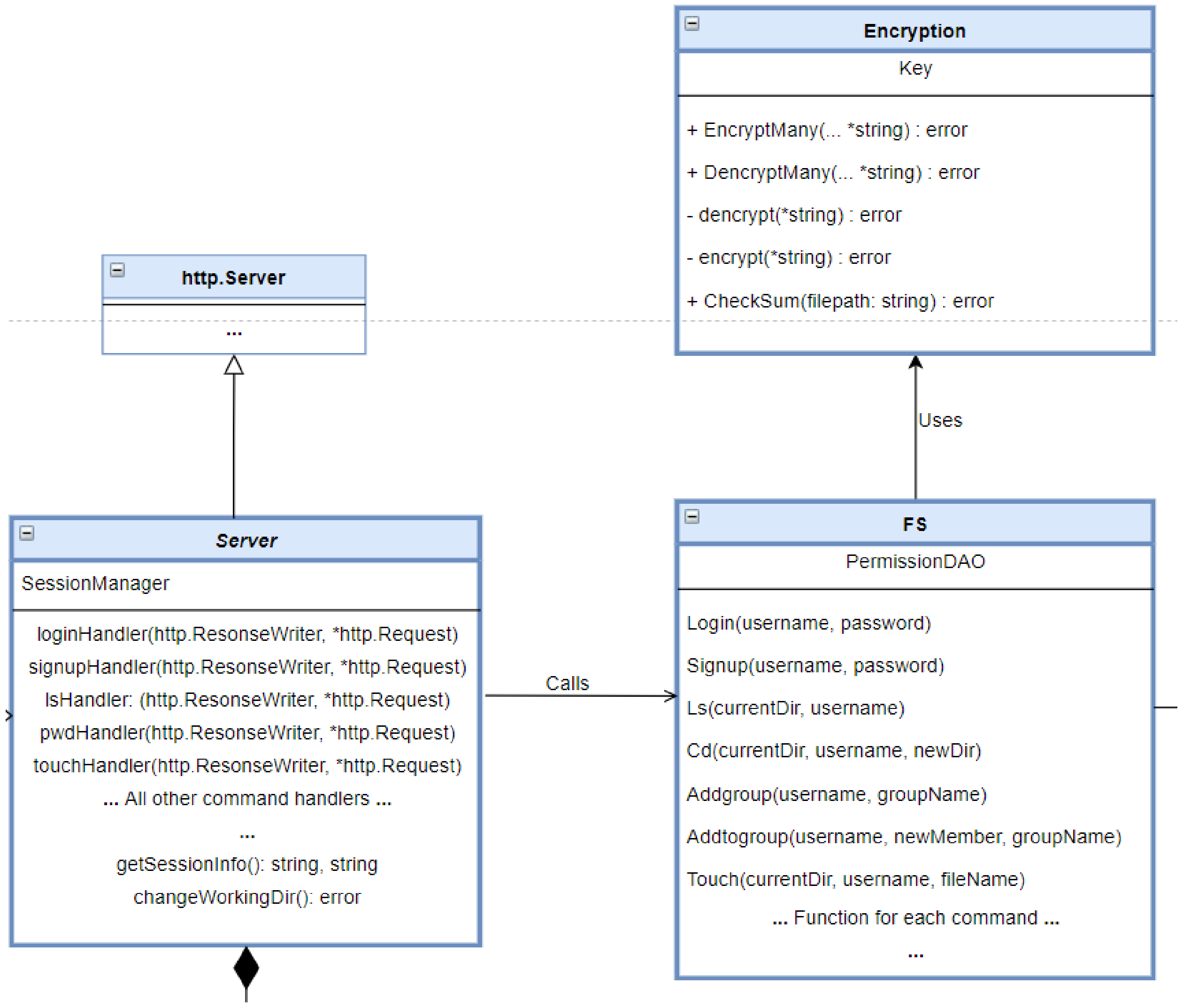
Encryption
Key
+ EncryptMany(... *string) : error
+ DencryptMany(... *string) : error
- dencrypt(*string) : error
- encrypt(*string) : error
+ CheckSum(filepath: string) : error
http.Server
...
Uses
Server
SessionManager
loginHandler(http.ResonseWriter, *http.Request)
signupHandler(http.ResonseWriter, *http.Request)
lsHandler: (http.ResonseWriter, *http.Request)
pwdHandler(http.ResonseWriter, *http.Request)
touchHandler(http.ResonseWriter, *http.Request)
... All other command handlers ...
...
getSessionInfo(): string, string
changeWorkingDir(): error
Calls
FS
PermissionDAO
Login(username, password)
Signup(username, password)
Ls(currentDir, username)
Cd(currentDir, username, newDir)
Addgroup(username, groupName)
Addtogroup(username, newMember, groupName)
Touch(currentDir, username, fileName)
... Function for each command ...
...

figure 3 (excerpt from UML)

4.6 Secrets File

The secrets file is used to hold the encryption keys being used by the file system for encryption of all data. It is loaded into memory by the encryption module on start up. Obviously this presents significant security risks, as an external user of the file system server could gain access to the encryption keys. A more robust solution would be to use an externally mounted disk partition with sole access owned by the file system server user, or even a dedicated hardware device.

4.7 Encryption

The encryption module contains various functions used for the cryptographic operations including AES data encryption / decryption, and SHA-256 Checksum calculation. It is used by the FS to validate the checksums of all of a user's files on login and report discrepancies to the user, as well as encrypt all users data and file contents. The encryption is carried out using an AES block cipher in Galois Counter Mode (GCM) with a constant nonce. The use of a constant nonce as opposed to a randomly generated nonce certainly reduces the effectiveness of the encryption, but has the added benefit of preserving data equality comparisons, even in encrypted form. This always allows the FS to compare for example the provided password with the encrypted password without needing to decrypt the password in-memory.

4.8 Database DAO

The database DAO is an implementation of the Data Access Object (DAO) pattern, which enables the data access logic to be decoupled from the main logic of the file system. The DAO exists in the system as a singleton that is instantiated on initialization of the filesystem, ensuring that data access is serialized even among parallel requests to the server by different users. The DAO makes use of many sql query string templates to execute, select, insert, and update queries with the data passed to them by the FS as needed. The DAO is also responsible for running the database initialization script on server startup, as well as creating and maintaining database connections.

4.9 Database

The Database was implemented as an SQLite DB consisting of 5 tables, as shown in the below ER diagram of *figure 4*. The database is simply stored locally on the ubuntu BM in a file called sfs.db, and is recreated every time the server is restarted for ease of development and demonstration. The SQLite is capable of handling multiple connections at the same time, however our system serializes access to the DB by locking the database DAO with a mutex. Any data that is stored on disk is **guaranteed** to be encrypted, because the FS encrypts prior to storage.

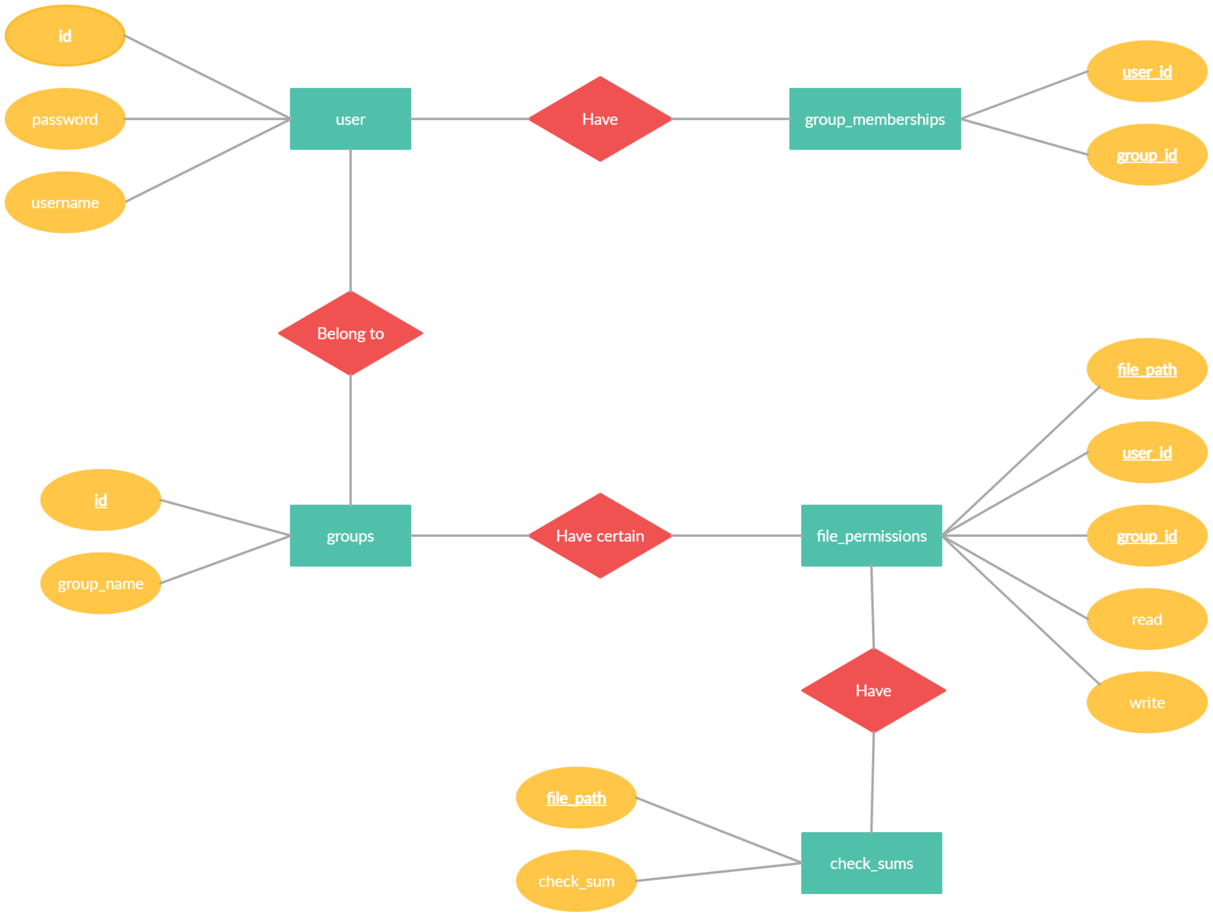

figure 4

5 Deployment instructions

The SFS system consists of two separate pieces of software running in two separate locations. The sfs server is to be deployed to a Cyber Cloud Ubuntu VM, while the SFS Shell is to be run locally on the machine of any user. The deployment instructions below have been split into two sections, the first describes setting up and running the client, and the second describes deploying and running the server.

5.1 SFS Shell Deployment

To deploy the SFS Shell, ensure that the source code is loaded onto your local machine. To do so you can either download the code, or pull it from the Github repository. Once the source code is available:

1. Open the root folder of the project in the terminal.
2. Ensure that the Go CLI and compiler is installed on your machine. For references help see https://golang.org/doc/install. You can verify this by running:

   go --version
3. Navigate to the **shell** directory
4. If not using our cloud VM host, change client.go line 10 to new hostname.
5. Run:

   go build shell.go

   This should generate an executable file called **shell** or **shell.exe** if running on a Windows system.
6. Ensure that you have execution permissions on the executable. If not, add them.
7. Execute the shell using:

   ./shell

5.1 SFS Server Deployment

To deploy the SFS Server, ensure that the source code is loaded onto the server that you wish to deploy to. You can do this by remoting into the Cloud VM using SSH and cloning the git repository directly onto the machine. After this is finished, navigate to the project root directory and follow these steps:

1. Ensure that you have a GCC compiler installed. This is required to run the SQLite driver. To check, run:

   Gcc --version

   If not installed, run:

   sudo apt install build-essential

2. Set **CGO_ENABLED** to true by running

   sudo export CGO_ENABLED=true
3. Ensure that an SQLite driver is present. If not, run:

   sudo apt-get install sqlite3
4. Navigate to the sfs-server directory in the project.
5. Run:

   go build server.go

   If go complains about missing dependencies, install them by running the suggested commands and try again:

   go get <url>

   You should now see the generated **server** executable in the same directory
6. Run the server by calling

   Sudo ./server

   It is important to run the server as super user, to ensure it has proper file access privileges.
7. The server should now be running!

6 User guide for your SFS

The SFS shell simply needs to be deployed and run on the user's local machine as shown in section 5.2. Once the SFS shell is running, the user can always use the command **help** to receive a detailed list of commands and explanations, just like those given below. The general usage is like this: Create an account with the signup command. Once signed up, you can execute any commands, add any groups, and add users to your groups as you please. When you're finished, you can use the logout command to end the session with the server.

1. **signup <username> <password>** - Signup for a new account in the sfs
2. **login <username> <password>** - Login to the SFS
3. **addgroup <groupname>** - Create a new user group
4. **addtogroup <username> <groupname>** - Add a user to a group
5. **ls** - List the contents of the current directory
6. **pwd** - show the current directory path
7. **mkdir** <directory_name> - Create a new directory in current directory

8. **cd** - Change the current directory (support ~/./..)
9. **cat** <file_name> - Show contents of file, line by line.
10. **touch** <file_name> - create a new file with provided name in current directory
11. **mv** <old_path> <new_path> - move a file from one location to another
12. **rm** <file_name> - Delete file
13. **write** <file_name> <contents…> - Write data to a file

7 Conclusion

The objective for this project was to design and implement a secure file system that could support multiple users of varying permissions. The users can create, modify, and store data onto the server, where the data is automatically encrypted and decrypted when needed. Although initially the vagueness of the project outline led to confusion and uncertainty, it allowed for brainstorming and let us think of our own ways of meeting the requirements. We ultimately decided on a client-server application in which the client sends requests to the server that the server will then perform. We successfully uploaded the server onto the cloud, and successfully established an https connection between the client and server.

8 References

[1] Youngkin, R., 2020. *Create Secure Clients and Servers in Golang Using HTTPS*. [online] BetterProgramming. Available at: <https://betterprogramming.pub/create-secure-clients-and-servers-in-golang-using-https-aa970ba36a13>.

[2] Hiwarale, U., 2020. *Secure HTTPS servers in Go*. [online] Medium. Available at: <https://medium.com/rungo/secure-https-servers-in-go-a783008b36da>.

[3] Astaxie.gitbooks.io. n.d. *How to use session in Go · Build web application with Golang*. [online] Available at: <https://astaxie.gitbooks.io/build-web-application-with-golang/content/en/06.2.html>.

[4] Bendersky, E., 2019. *On concurrency in Go HTTP servers - Eli Bendersky's website*. [online] Eli.thegreenplace.net. Available at: <https://eli.thegreenplace.net/2019/on-concurrency-in-go-http-servers#:~:text=Go's%20built%2Din%20net%2Fhttp,also%20lead%20to%20some%20gotchas>.

[5] Turtle, E., 2019. *Read a File to String - GolangCode*. [online] GolangCode. Available at: <https://golangcode.com/read-a-files-contents/>.