Overall Design

User Structure:

The struct we used: {

Username string,
Password string,
RSAPrivKey *rsa.PrivateKey,
HMACKey []byte,
EncryptKey []byte }.

To ensure confidentiality we used CFB encryption on all the fields and then HMAC the entire struct. The keys were generated using the entropy of the username and password. We also create an RSA public key and store it in the KeyStore. When we retrieve a user we recreate the keys and verify the HMAC and then decrypt the data to get the userdata. Note that this means that even an adversarial DataStore will not break the program, because any changes will always be detected and the adversary will not be able to see the data as it is encrypted.

File Structure:

We will be using Merkle Trees on the client-side to confirm integrity of the file. When a file is stored on the server, it is broken up into three types of individual files described below:

- Header: contains the filename, name of the Merkle Root file, and HMAC of the Merkle Root file, file key, and HMAC key
- Merkle Root: contains merkle root and names of all data blocks of all Data Block files.
- Data Block: contains the actual byte data of what we want to store.

Using Merkle Trees, we are able to easily recognize any changes made to the files by comparing merkle roots. Furthermore, this design allows for sharing and tracking changes by allowing each user to create a header file that points to the Merkle Root. When one of the authorized users change a file, all other users' view of the file does not change. It is important to note that when files are encrypted at rest on the untrusted server in the following format:

- IV, E(file), HMAC(file); where the IV is used for decryption and the HMAC provides integrity and authenticity

When data is appended to the file, a new Data Block file is created, the Merkle Root file is updated with a pointer to the new data block, and the header file is updated with the new merkle root. Every time the file is loaded, its Header, Merkle Root, and Data Block files are verified against the HMAC and decrypted. The merkle root of the data blocks is then computed and checked against the previously stored root to detect any changes made to the file. We also note that the encryption and HMAC keys for each file is unique, ensuring that any two Data Blocks or Merkle Root files with the same data under different file names are indistinguishable.

File Naming Convention:

In a motive to increase security, we need ensure that filenames are indistinguishable from random bits. To that end, we highlight the following naming scheme:

- The secured filename for the Header file is HMAC(username + password + filename]), where the HMAC Key is derived from the username and password

- The secured filename for the Merkle Root and Data Block files is given by HMAC(random bytes), where the HMAC Key is stored in the Header file.

Using this scheme, we are able to ensure that an attacker is not able to recognize any files by looking at their names.

Sharing and Receiving Files:

We used a sharingRecord struct to allow a file to be shared using complete confidentiality. The struct is: {

MerkleRoot [ ]byte,
EncryptKey [ ] byte,
HMACKey [ ] byte,
PrevRoot [ ] byte,
RSASign [ ] byte }.

All the fields of the record have been RSA encrypted using the public key of the recipient and the record is RSA signed by the private key of the sender to ensure integrity. We create a random 16 bit msgid that is the key to find the sharingRecord in the Datastore When the recipient tries to receive the file we will recreate the header file using a different filename. First, we check the RSA Verify using the sender's Public Key and then we individually decrypt all the fields using our private key. Using this information we create a new header file that is encrypted using our EncryptionKey and HMACKey. This should give us access to the file only if it was sent by the receiver and without any other information about the original filename.

Revoking Files:

We want for the owner of a file to be able to revoke sharing rights from all the people who have been given them. We do this by creating a copy of the data and deleting the header, merkle_root and all the data from the data blocks from the DataStore. Then we recreate the header file and the merkle_root using the local data blocks. Note that the Merkle Root and Data Block files will be created with random names once again. This way we can maintain access for the owner of the file while removing the access of all other users.

Testing:

Our overall testing strategy was to test basic things first and then test more complex problems. We tested that users could be created and retrieved, that the encryption had enough entropy to distinguish between two files that only differ by name, that we can append and load files, that we can share files, that we can receive files and that we can revoke access to a file. Then in more complex tests we checked that all viewers of a shared file could see changes created by any of the users. We also tested that second-degree sharing is possible and that changes by the third person are also viewable by all users. Then we tested that a revoke does not break the caller's access and that the append function is efficient. Such a comprehensive testing scheme ensured that our solution was secure and met all the requirements.

Security Analysis of the Design

In analyzing the security of our design, we present the following attacks and show how our design can prevent them

1. **Chosen Plaintext Attack:** Consider the situation where an attacker, Eve, knows the possible values that may be stored in a file. She wants to run a CPA to see which encrypted data matches the data Alice has stored. Assume that the attacker Eve is able to get Alice to encrypt and store any plaintext chosen by Eve herself and has a way to see the encrypted data within a file. There are two possible scenarios that we must consider: one where the plaintext belongs to the same file and one where the plaintext belongs in different files.
    a. In the case where the plaintext is stored in the same file, because the encryption key is the same for all Data Blocks within a file, we have a potential vulnerability; however, since we use a random initialization vector for each instance of encryption, we can ensure that no information is leaked, even if the plaintext is the same. This is of course assuming that the attacker does not know the encryption key.
    b. In the case where the same plaintext is stored in different files, the encryption key and the initialization vector would both be different. This ensures that the ciphertext would be different, leaking no information on the plaintext

2. **Modifying Encrypted Data:** Assume that the malicious attacker Mallory has gained root access to the data store. She is able to look at all the filenames and modify any data stored on the server. The design of the file store easily protects against this sort of attack by appending storing data as a combination of ciphertext and an HMAC. Any changes to the ciphertext would result in an error when the ciphertext is compared to the HMAC. The only way for Mallory to successfully pull of this attack is by gaining access to the HMAC keys. To show that this is impossible, we consider the two following situations:
    a. If Mallory knew the HMAC Key for the Header file, she would have to know Alice's username and password, which assume to never be true.
    b. If Mallory knew the HMAC Key for the Merkle Root or Data Block file, she would have to have been able to decrypt the Header file to find out the needed key; however, in order to do this, she would have to have known Alice's username and password, which we assume to never be true.

3. **Swapping Data:** Suppose that Mallory is a malicious attacker, with the ability to track a user's history of the files that were uploaded. We also assume that she has found a way to recognize Header files based on length of the encrypted data; this is highly plausible if Mallory knows the format of the data. Consider the situation where Mallory swaps the data of two Header files. This poses a problem because all of the metadata and data for a file is contained or linked by the Header file. Preventing against this attack comes from a simple check in the code, where we check if the filename stored in the Header matches the filename we queried for. Although this attack does not exploit any weaknesses in the cryptographic protocol, it is an important one to consider nonetheless.