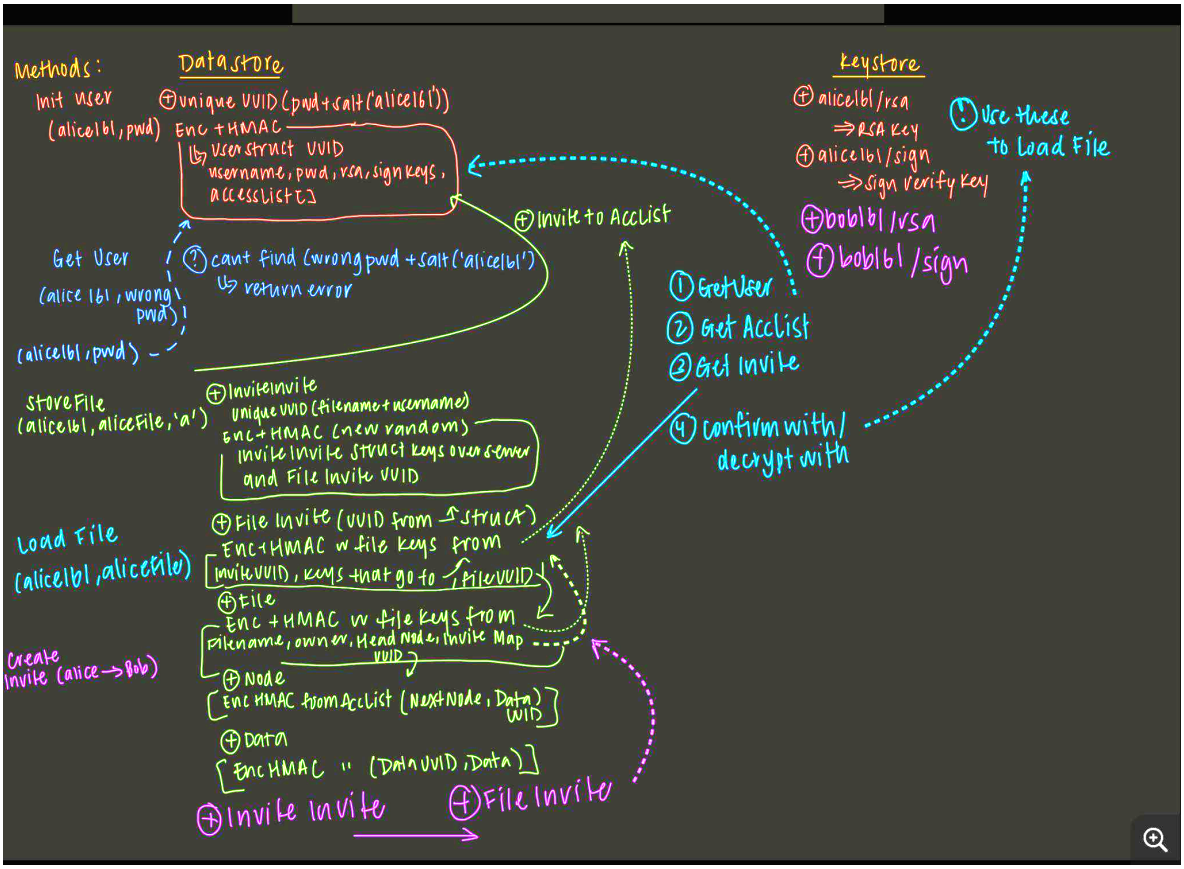
Project 2 Design Document:



Data Structures

```
type User struct {
        UserUUID userlib.UUID
        Username string
        Password string
        RSAKEY userlib.PKEDecKey
        SIGNKEY userlib.DSSignKey
        AccessList map[userlib.UUID]InviteInvite

type InviteInvite struct {
        InvUUID userlib.UUID
        InviteUUID userlib.UUID
        InvSymKey []byte
        InvMacKey []byte
}
type FileInvite struct {
        InviteUUID userlib.UUID
        FileUUID userlib.UUID
        FileSymKey []byte
        FileMacKey []byte
}
type File struct {
        FileUUID userlib.UUID
        Filename string
        FileOwner string
        InviteMap map[userlib.UUID]InviteInvite
        UserMap map[string]userlib.UUID
        HeadNode userlib.UUID
        TailNode userlib.UUID
}
type Node struct {
        NodeUUID userlib.UUID
        NextUUID userlib.UUID
        DataUUID userlib.UUID
}
type Data struct {
        DataUUID userlib.UUID
        Data []bytes
```

The main structures we used are above. We used a user struct to store private info about a user that can be accessed when they are logged in with a valid username and password. When creating a new file, the method first returns an Invite to the FileInvite. This invite contains the keys needed to decrypt and verify the Invite from the file owner or the Client. After using the keys to decrypt the FileInvite, there is a UUID pointer to the file header including a File Symmetric and Mac Key to Decrypt and Verify the file struct. Once that is unmarshalled we can

access a pointer to a linked list with the Nodes for each append. In the File struct there is also a list of the invites and the users who have been invited. Each of the nodes has a pointer to the next Node and a data struct with the actual node.

User Authentication

When a user is initialized with call to InitUser(‘alice161’, ‘password’), this creates an addition to Datastore with the unique uuid of the the salted hash of the password and username pointing to the user struct for ‘alice161’. In addition, we also add two unique keys: an rsa key and a signing key to the keystore at the ids ‘alice161/rsa’ and ‘alice161/sign’ to use when encrypting Invitations later.

When she attempts to login by calling GetUser(‘alice161’, password), we assume that the user enters a valid username and password and calculate the uuid with the given password and username.The user symmetric and rsa key is generated to decrypt the data and check that it hasnt been tampered with. If the datastore does not contain the uuid, it cannot be returned and if it hits another set uuid that isnt a user, the key generated with the salted username and password will be unable to encrypt the data and gives the user back garbage. With the incorrect password, neither the uuid lookup of subsequent key generation for hmac and decryption would yield any relevant information or let the user log in.

File Storage and Retrieval

Once a user is logged in and calls to alice.StoreFile(‘aliceFile.txt’, content) we first create invites and initialize a random file uuid from the filename and username. We also initialize a single invitation in the File Invite map and add the credentials to load a file (the keys and the file uuid) to the user struct’s individual access map.

To access a file with alice.LoadFile(‘aliceFile.txt’) we first get the user then we check that the file is in the personal namespace and the corresponding UUID exists in Datastore. Then they can look in the invites dictionary and find the key associated with this invite. Using the invite symmetric key, they can decrypt the invitation and access the symmetric key to decrypt the actual file. Then they can make any changes to the file or view the file contents.

When appending to a file all the steps above to load a file are first executed and once we have the File and node structs decrypted we can add a new Tail Node with a new Data structure with the content and also fetch the previous tail node and change the Node.nextUUID to point to the newly created node. This allows us to append efficiently without downloading the previous node’s data with every append.

File Sharing and Revocation

Given two users, Alice and Bob, if Alice wants to share a file ‘aliceFile.txt’ with Bob we first create an invite from the file owner to Bob. This is sent over the network as an InviteInvite to allow Bob to verify the integrity of the invite and choose to accept the invitation. If Bob accepts the FileInvite, we can use the keys in the InviteInvite to decrypt the FileInvite and get the FileUUID to make changes, append, or override the file.

If Bob (not the owner) attempts to share the file with Charlie, Bob would send his own InviteInvite to open the FileInvite. Only the owner can create new keys for direct shares. In the case that Bob’s invite becomes invalid, since Charlie's Invite points to the same UUID, he would also lose access.

The FileUUID changes if any user's access is revoked meaning they are unable to know where in datastore the file is anymore. In addition we update all the invitations with a new file symmetric and mac key. All the users whose access was revoked because they had an invalid invitation will also have the file taken out of their personal namespace and they will not be able to access it.

Helper Methods

To Implement this Project we made helper methods that would allow us to more efficiently repeat actions.

Store User - useful if we are changing the access list to add/remove invitations to marshal, encrypt etc.

InviteMarshal/Unmarshal = changing valid invitations in, RevokeAccess or CreateInvitation

GetInvite = get from datastore and decrypt and verify, loadFile/storeFile checking Invites

Get/StoreFile/Node/Data/Header = retrieve/put datastore