Data Structures
I have listed the struct definitions I implemented and heavily utilized throughout the project below:

User

- Username string
    - unencrypted string
- PKEDecKey, DSSignKey userlib.PrivateKeyType
    - private RSA Decryption Key and private RSA Digital Sign Key
- Datastore_key, HMAC_key, SymEnc_key []byte
    - keys for User encryption
- Files map[userlib.UUID]File_Data
    - each file has its own storage_key that maps to its own file_data, another struct
- Lockboxes map[userlib.UUID][]Lockbox_Data
    - each file has its own storage_key that maps to an array of lockbox_data

File

- Content_length int
    - number of appends that have been made to file, or nodes
- Owner, Start, End userlib.UUID
    - Owner: who initially stored the file
    - Start: pointer to first node in Datastore
    - End: pointer to last node in Datastore

File_Data

- Datastore_key userlib.UUID
    - pointer to encrypted File in Datastore
- File_symenc_key, File_hmac_key []byte
    - symmetric keys to encrypt/decrypt File objects
- Node_symenc_key, Node_hmac_key[]byte
    - symmetric keys to encrypt/decrypt Node objects

Node_Content

- Content []byte
    - file content
- Next userlib.UUID
    - pointer to next node

Lockbox

- File_encrypt_key []byte
    - appended symmetric keys to encrypt/decrypt File objects
- Node_encrypt_key []byte
    - appended symmetric keys to encrypt/decrypt Node objects
- File_location userlib.UUID
    - pointer to encrypted File in Datastore

Lockbox_Data

- Location userlib.UUID
    - pointer to location of encrypted Lockbox in Datastore
- Encrypt_keys []byte
    - symmetric keys to encrypt/decrypt Lockbox in Datastore
- Shared_user string
    - who was recipient or who sent the invite

User Authentication

Each user stores their private keys, the uuid that maps to the encrypted user in Datastore, a map of files and lockboxes in their struct. Every function call, except for InitUser, begins with pulling the most recent version of the user from Datastore. Every function ends with marshaling the user, encrypted under a symmetric encryption scheme with keys that are stored inside of the user struct, then appended with a checksum generated by the HMAC function, before being uploaded to Datastore.

The public RSA keys (such as the public encryption key and digital signature key) are stored inside of Keystore, and easily retrievable by hashing the username.

A user can have multiple client instances because all of the data is continuously being pulled and uploaded to Datastore everytime any changes are made. Because no data is being locally cached, a user can start multiple instances and simply retrieve the appropriate user data from Datastore. The key that maps to the encrypted user struct can be easily derived using the user's password and username, both of which are arguments in GetUser.

The client ensures that the user maintains confidentiality and integrity over its data whenever it is stored in Datastore, as we encrypt it first using a symmetric encryption scheme and then append a check_sum derived from HMAC which is always checked for equality everytime we pull the encrypted data from Datastore.

Relevant Client API Methods: InitUser, GetUser

File Storage and Retrieval
How will you store and retrieve files from the server? How will your design support efficient file append?

Relevant Client API Methods: LoadFile, StoreFile, AppendToFile
StoreFile
As with most of the functions implemented, start off by pulling the most recent version of user from Datastore. Then, verify the integrity of the encrypted user data using the HMAC_Eval function before decrypting the user data into a user struct. For storing and retrieving files, two types of structs were necessary in my implementation: the File struct and the File_Data struct. Each user stores a map of File_Data structs which contain essential information (the symmetric encryption keys, the hmac keys, and the Datastore key for the File) towards decrypting a File, which is encrypted and stored in Datastore. If the file does not exist, then we derive four keys for encrypting the File and a Node struct which will be discussed below. File does not contain the actual content, but follows a linked list structure where the start and end attributes point to the first Node or essentially the first set of content before more appends are made.

If the file does exist, then we simply retrieve it from Datastore using the user's Files map (which contains keys necessary to decrypt and verify the integrity of it) by keying into it.

LoadFile
After pulling the most recent version of user from Datastore and verifying the integrity of it, we check if the user has any lockboxes associated with it (inside of user.Lockboxes[file_name_key]). Sometimes, a file will be deleted but the lockbox will remain which means a user has been revoked. The lockbox of non-revoked users are updated within RevokeAccess so it is necessary to fetch the most recent version of it to retrieve the file and node keys in order to decrypt any Files or Node_Contents. After fetching the file, we decrypt and verify its integrity.

Then, we traverse through all of the Node_Contents associated with this File, by starting the first node which File.Start points to in Datastore. We follow the next pointers (which is an attribute in each node) and retrieve the respective contents for each node. Then we append it all in one string and return.

AppendToFile

Once again, pull the most recent version of user from Datastore and verify its integrity before decrypting it. Then, we check if the file exists in the user's personal namespace by referencing its Files map. We want to pull the most recent version of the lockbox which contains the encryption information for the file that is shared with others as a safeguard against revoked users. We update the file_data associated with this file according to the updated lockbox before proceeding.

After verifying the integrity and decrypting the file contents, we create two nodes where the second node is labeled last_node. This last_node already exists in Datastore for File.End points to it. The first node is our new_node, or essentially what is being appended. The last_node will point to the new_node and the new_node has now taken on the position of being the last node in our file linked list structure. Update file.End to point to the new node which is also encrypted and stored in Datastore.

File Sharing and Revocation

To revoke access, we re-encrypt the file and nodes associated with it and store each of them in different locations. Then, we delete the original key associated with the file and also the original keys associated with each of the nodes. Then, we update the lockboxes of each of the other non-revoked users (or the top-level children of the user) with the information as to where the files and nodes now reside in Datastore along with the keys needed to decrypt them. This way, the revoked user and its children cannot load the file as the key they have to retrieve it from Datastore has been deleted and their lockbox containing the old information has also been deleted.

CreateInvitation

Start off by pulling the most recent version of user from Datastore. First, we verify the file that the sender wants to send an invite for exists in their namespace. Then, we also check if the recipient exists before retrieving their public key from Keystore. We also retrieve the file from Datastore and take note of the owner, because this information can lead us down to one of two paths:

If the owner is not the one sending an invitation, but rather one of its children, we do not create a new lockbox. Instead the sender will ultimately send an invitation pointer that leads the recipient to reference their own lockbox. Since the lockbox already exists in the sender's Lockboxes map we can encrypt the keys using asymmetric encryption. We follow a hybrid encryption scheme where the keys are derived and then the lockbox is encrypted

under a symmetric encryption scheme. Then those keys are encrypted using the recipient's public key and then signing it with the sender's private sign key.

If the user is also the owner, we create a new lockbox, populate it with the appropriate information encrypt it using a hybrid encryption scheme before creating an invite that contains the uuid pointing to this encrypted lockbox in Datastore.

AcceptInvitation
We check the existence of the sender and determine if the user already has a file with the filename in their namespace. We retrieve the invitation struct from Datastore which contains the assymetrically encrypted keys and pointer to the lockbox. We verify the invite came from the sender using their public verification key and also decrypt using our private RSA key. Then, we use the decrypted keys to decrypt the symmetrically encrypted lockbox in Datastore. We populate the fields of file_data associated with this file_name according to the fields in our lockbox.

Helper Methods
I did not create many helper methods but I did create a DeriveFromSourceKey helper method that takes in a source_key, a keyword and creates and stores a new key that is derived from the two arguments.