How UUID Generated	Enc?	Sym/Asym/Key derivation	Value: struct being stored	Description
UserName: uuid.FromBytes(hash(username)[:17])	Yes	Symmetric sourceKey= argon2key(username + hash(password), salt, 16)	User struct (highlighted in yellow below)	Struct user - Used to perform all file operations. Hmac tag - Used to check for tampering on user struct

FileMetadata: uuid.New()	Yes	EncryptionKey: RandomBytes()
		
		HMACKey: generated from EncryptionKey
			type FileMetadata struct {
			LatestNodeUUID uuid.UUID
			}
				Contains the UUID of the latest appended (or last) FileNode

FileNode: uuid.New()	Yes	EncryptionKey: RandomBytes()
		–Same key as Metadata
		HMACKey: generated from EncryptionKey
			type FileNode struct {
			Content []byte
			PrevNodeUUID uuid.UUID
			}
				Store file contents in bandwidth saving way (reversed linked list)

ShareNode: uuid.New()	No	EncryptionKey: RandomBytes in Invitation
		MacKey: maced using a key that is generated by EncryptionKey
			type ShareNode struct {
			FileMetadataUUID uuid.UUID
			FileDecryptionKey []byte
			Dead bool
			}
				Gives a user information about the fileMetadata and the fileNode

Invitation: uuid.New()	Yes	Encryption: RSA Encryption
		Integrity: Digital Signatures
			type Invitation struct {
			ShareNodeUUID uuid.UUID
			ShareNodeDecryptionKey []byte
			}
				Enables sharing

FileOwnerInfo: uuid.New()	Yes	EncryptionKey:sourceKey HMACKey: generated from EncryptionKey	FileOwnerInfo struct: - File DecryptionKey - FileMetadata UUID - SharedMap (Key: recipient name, value: invitation)	Universal truth for the File decryptionKey and File metadata UUID. Contains mapping of recipient and invitation to make revoke easier
FileMetadata: uuid.New()	Yes	Encrypted and maced using the file decryptionKey	File Metadata struct: - Latest fileNodeUUID	Contains the UUID of the latest fileNode UUID which we update for append

```
type User struct {
    Username          string
    sourceKey         []byte //Argon2Key(hash(username) + hash(password), username, 16)
    PublicRSA         userlib.PKEEncKey
    PrivateRSA        userlib.PKEDecKey //= PKEKeyGen(); // keystore.add(username + “RSA”, PublicRSA);
    PublicSign        userlib.DSVerifyKey
    PrivateSign       userlib.DSSignKey   //DSKeyGen();  // keystore.add(username + “Verify”, PublicSign);
    EncPrivateRSA     userlib.DSVerifyKey //SymEnc(sourceKey, iv1, privateRSA);  //iv1 = RandomBytes(16);
    EncPrivateSign    userlib.DSSignKey   //SynEnc(sourceKey, iv2, privateSign);  //iv2 = RandomBytes(16);
    FileMapUUID       uuid.UUID           //UUID for make(map[string][]byte) //HashMap<key:filename, value:UUID for FileOwnerInfo>
    SharedWithMeUUID uuid.UUID            //UUID for make(map[string][]byte)   //HashMap<key:filename, value:Invitation>
}
```

Design Question: User Authentication: How will you authenticate users?
First, we check to see if UUID(username) exists in Datastore. If it exists, then the username exists. If not, the UUID(username) was either changed or doesn’t exist. Then, we generate a new sourceKey and hmacKey using the user’s inputted username and password. Using the hmacKey we check the hmac tag to verify the struct. If the struct isn’t equal, then the struct has either been changed or the password is incorrect. Then, we simply decrypt the encrypted, serialized user struct and then deserialize it to get the clean user struct.

Design Question: Multiple Devices: How will you ensure that multiple User objects for the same user always see the latest changes reflected?
When a user logs in, using the username and password we deterministically generate a UUID which is used as a pointer to a list of files. The file list itself contains pointers to file_node structs that point to the tail of the reversed linked list representing files. This allows different user objects to point to the same list and most recent file additions and thus updates can be seen from all ends.

Design Question: File Storage and Retrieval: How does a user store and retrieve files?
First a user logs in, this provides the UUID of their unique file map. When storing a file, FileMetadata struct and FileNode struct are initialized. The FileMetadata stores the uuid of the latest FileNode which holds the content and is encrypted. Then this UUID is added to the user’s file list. To retrieve a file’s content, the file list maps to the file metadata and starts decrypting and reading from the end of the chain (the latest update) and reads in reverse.

Design Question: Efficient Append: What is the total bandwidth used in a call to append?
The structure we use to control bandwidth is linked list as the file. To append data instead we reference our file metadata to retrieve the previous node. Instead of decrypting and re-encrypting that older node to update its next pointer, we have our new file node point to that and write in the new contents then encrypt on the entire new node, thus a “reversed” link list where each new node points backwards. The file metadata is then updated with the latest node uuid. The bandwidth size of each append should only scale with the size of the appended portion and not the full file contents since appending only creates a new file node without affecting the previous ones.

Design Question: File Sharing: What gets created on CreateInvitation, and what changes on AcceptInvitation?
On CreateInvitation, the sharer, if they are the owner, creates the sharenode for the recipient containing the filemetadata uuid and file decryption key. An invitation containing the uuid of this sharenode and its decryption key is sent via RSA encryption and also signed. On accepting the invitation the recipient adds this invitation to their mapping “SharedWithMe” which then allows them to reference the file information and make appends/loads as necessary. If a non-owner attempts to share, they simply duplicate the invitation they received from whoever shared with them and pass that on. This works because if Alice shares with Bob, and Bob shares with anyone else, Bob shares the information for HIS shareNode with Alice. This makes revocation easy because if we revoke Bob, changing his shareNode changes information for Bob and everyone else HE shared with and the people they shared with and so on.

Design Question: File Revocation: What values need to be updated when revoking?
To revoke access, an owner can simply load in the current file contents and store it again under the same filename thus overriding the old file by filename reference, changing the filemetadata location, and re-encrypting all the data. The old file chain and metadata then gets deleted making it impossible for the adversary to try regaining access. The new file meta uuid and encryption key are then placed in the share nodes for all those who are supposed to have access and the revoked user is removed from the sharemap for that file.