**CS161 Project 2 Design Doc - Yunsu Ha**

**Data structures: What data structures are you going to use to organize data on Datastore?**

- Struct user:
  - username
  - Hash(Argon2Key(password,salt1)) - for verifying correct user
  - Salt1 and Salt2
  - password - this is not stored when marshaled.
  - SymEnc(Storage) - key is Argon2Key(password, salt 2)
- Struct Storage:
  - Dictionary - dictionary of all fileNames and their corresponding file UUIDs and private keys. If the file was a shared one, the UUID points to their invitation.
  - Hash(Argon2Key(password,salt1)) - for verifying integrity
  - Private key - private key of this user used for public key decryption
- Struct File:
  - contentUUID - uuid that points to the content structs
  - verifier - random bytes that is used to verify the integrity of the invitation structs
  - sharedDictionary - dictionary of users and who they shared the file to
  - invDictionary - dictionary of users and their invitation struct UUIDs
  - Owner - for checking if user has right to revoke
- Struct Content:
  - Content - piece of the actual content of the file
  - prevUUID - uuid that points to the previous node of the content linked list
- Struct Invitation:
  - fileUUID - UUID of the file
  - Key - private key to decrypt the file
  - Verifier - random bytes that are used to verify integrity of invitation
- Struct Tuple: Just for storing multiple values per key in a map

**Helper functions: As you come up with a design, think about any helper functions you might write in addition to the cryptographic functions included here.**

- Dfs - performs dfs on sharedDictionary to get all users that is shared by the given user
- containsVal - checks if a list contains an element

**User Authentication: How will you authenticate users?**

When first initializing a user, we take in the username and password and make sure the username is valid. Generate a salt and then use Argon2Key to slow hash the password along with the salt, and Hash the result after. Create a User Struct and add the salt to it along with the hashed password. Generate a second salt (salt2) and store it in the struct. Create a storage struct and add the hashed password into it. Generate a PKE key pair and store the public key in the keystore, and the private key in the storage struct. Encrypt using Argon2Key(password, salt2) as key and add it to the user struct. Marshal this struct and add it to the Datastore with the UUID of the Argon2Key username. When the user tries to login, look for the uuid that corresponds to the input username that has been Argon2Keyed and verify that the passwords match using the salt and hashed password.

**Multiple Devices: How will you ensure that multiple User objects for the same user always see the latest changes reflected?**

Make sure whenever there is a file update, it is always updated on the DataStore. Whenever a user calls LoadFile, make sure it is retrieved from the DataStore with the latest updates.

**File Storage and Retrieval: How does a user store and retrieve files?**
For FileStore, create a File struct. Generate two random UUIDs contentUUID, and fileUUID. Create a content Struct and store the content in there. SymEncrypt the content struct using a randomly generated private key as the private key. DataStoreSet the encrypted content with the contentUUID. Store contentUUID and verifier into the File Struct. Encrypt the file using SymEnc and the same private key. Using the fileUUID, DataStoreSet the encrypted file. Store the fileUUID and the key inside the dictionary of the user's storage struct.

For LoadFile, get the fileUUID from the storage struct. DataStoreGet the corresponding file. Decrypt the file using the private key. While prevUUID of the content Struct is not null, iterate backwards while reconstructing the data until the entire file content has been reconstructed. This file structure is similar to a linked list, except the data contents are stored in reverse order. If the file the user is trying to access is a file they got access through an invitation, their storage struct dictionary will have the uuid of the invitation. Access the invitation and use the information in that to gain access to the file.

**Efficient Append: What is the total bandwidth used in a call to append?**
Each time AppendToFile is called, a new Content Struct (A) is created. The Content Struct that the contentUUID used to point to (B) is moved to a new UUID, and the contentUUID points to A. A contains a prevUUID to UUID of B. Therefore the total bandwidth of the call to append is the size of the content + the constant cost of calling DataStoreGet on the file to get content A + cost of DataStoreGet on A + DataStoreSet for A + DataStoreSet for B to readjust pointers.

**File Sharing: What gets created on CreateInvitation, and what changes on AcceptInvitation?**
Generate a random UUID called randUUID. Create an Invitation Struct and store the UUID of the file you want to share. Store the key of the file in the Invitation Struct along with the verifier. PublicEnc the struct using the recipient's public key. DataStoreSet the randUUID and the encrypted invitation struct.

For acceptInvitation, decrypt using the private key to get the Invitation Struct. Use the fileUUID stored in the struct invitation to retrieve the file and use the stored private key to decrypt it. Add the creator's name to the sharedDictionary and set the recipient's name to its value. Add the recipient's name to the invDictionary and set its value to the invitation UUID. In the user storage struct, add the fileName and set its value to the fileUUID and the creator username.

**File Revocation: What values need to be updated when revoking?**
Retrieve the file first and the list of users the revokedUser shared to. Go through each of those users and delete their corresponding invitations using the invDictionary and also delete those values from the invDictionary. Generate a new random UUID and assign that as the new pointer for the file, updating the value in the creator's storage struct as well. Additionally generate a new verifier for the files and store it in each remaining invitation struct. Going through all the other users that the file has been shared with, access their invitations and update the UUIDs of the file location.

## User Struct: Contains user information

- UUID: username
- not encrypted
- contents:
  - verifier Hash
  - salt1
  - salt2
  - SymEnc(Storage)

## Storage Struct: Stores file locations

- UUID : N/A
- Private key (for PKE)
- SymEncrypted
  - key= Slowhash(password, salt2)
- Contents:
  - files Dictionary: Contains file names and their UUIDs.

## File Struct: Stores

- UUID: Random
- SymEncrypted
  - Symkey = random bytes
- Contents:
  - invDictionary: dictionary of users and their invitation struct UUID
  - content UUID: points to content struct
  - verifier: used for integrity
  - shared Dictionary: dictionary of users and a list of people they shared to.
  - Rand Hash : For checking if user has revoking rights

## Invitation Struct

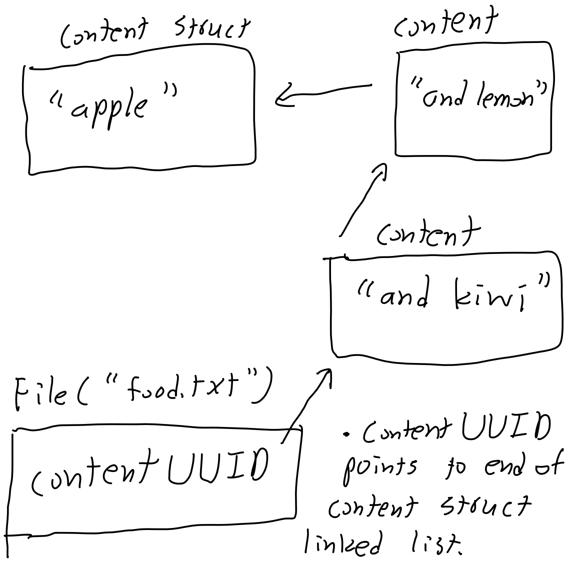
- UUID: random
- Public Enc
  - key- user priv key
- contents:
  - pkey: key to decrypt file
  - file UUID
  - verifier: for maintaing integrity

## Content Struct

- UUID: random
- symEnc
  - key = rand bytes
- Contents:
  - content : piece of the values in the file
  - prev UUID: points to the previous node of the content linked list

Content data structure:

- similar to linked list
- StoreFile ("food.txt", "apple")
- Append to File("food.txt", "and lemon")
- Append to File ("food.txt", "and kiwi")



- Content UUID points to end of content struct linked list.