Data Structures

All structs are encrypted and then MACed before being placed in datastore; we verify the MAC of all structs before decrypting and using the contents; all structs use separate keys for MAC and encryption; all keys are randomly generated except for the key used to encrypt the User struct which is generated using a salt and the user's password; for anything with a MAC or signature, the hash is appended to the ciphertext and stored in datastore that way

User Struct

Attribute Name	Type	Description
Username	string	This is where the user's username is stored
Password	string	This is where the user's password is stored
SignKey	DSSignKey	When we create a user, we generate a private sign key (to use later for signing files) and store it here
RSAKey	PKEDecKey	When we create a user, we generate a private RSA decrypt key (to use later for file encryption) and store it here
MapsUUID	UUID	We generate a UUID which marks a location in datastore to the maps held by the user; all of the user's maps are stored in a separate struct, which is encrypted, MACed, and stored in a datastore
SymPKey	[]byte	We generate a random symmetric key when the user is created to encrypt and decrypt the user's maps in datastore; this key should never change
MACPKey	[]byte	We generate a random private key when the user is created to MAC and verify the user's maps in datastore; this key should never change

UserMaps Struct

Attribute Name	Type	Description
OwnedFiles	map[string]FilePtr	Files created/owned by the user are placed in this map (key = filename; value = FilePtr)
SharedWithMe	map[string]FilePtr	Files not owned by the user but shared with the user are placed in this map; together with the owned files, these two maps create the user's filespace (key = filename; value = FilePtr)
FilesIShared	map[string]map[string]FilePtr	Files owned by the user and also shared with other users are place in this; this is a nested map that maps a filename to another map; for a given filename, a map mapping usernames to FilePtrs is returned

FilePtr Struct

Attribute Name	Type	Description
H	UUID	This contains a randomly generated UUID that marks the location of another FilePtr or FileChunk in datastore
S	[]byte	This contains a randomly generated symmetric key that is used to encrypt and decrypt the FilePtr or FileChunk that this FilePtr points to
M	[]byte	This contains a randomly generated key that is used to MAC and verify the MAC of the FilePtr or FileChunk this FilePtr points to

FileChunk Struct

Attribute Name	Type	Description
IsTail	bool	This marks whether or not this chunk is the last chunk in this linked list of FileChunks (whether or not PrevChunkPtr contains garbage)
PrevChunkPtr	FilePtr	This is a FilePtr that allows us to get the previous FileChunk; contains the UUID, encrypt/decrypt key, and MAC key of the previous FileChunk in datastore
Content	[]byte	This represents the content held by this FileChunk (not necessarily the full contents of the file)

User Authentication

Creating Users:

- When we make a new user, we encrypt the user struct by generating a key with Argon2Key(), using the user's password and a randomly generated salt
- The salt is saved in datastore under the UUID generated from the string: username + "-salt"
- We generate public and private keys for digital signature verification, signing the encrypted userdata with the user's private sign key and storing the digital signature by appending it to the ciphertext and uploading it all to datastore under the UUID generated using the user's username

Authenticating Users:

- We retrieve the user's stored data from datastore by generating the UUID using the user's username
- We verify the digital signature attached to the ciphertext by getting the user's public key from Keystore and using DSVerify
- We then get the salt from datastore by generating the UUID for the salt using the string: username + "-salt"; if DSVerify is successful we continue and if not we throw an error
- We use the salt and the user's password to generate the user's key using Argon2Key() and then decrypt the ciphertext
- If the decryption is unsuccessful then the password must be incorrect in which case we throw an error and the user is not authenticated
- If decryption is successful we log the user in by returning the decrypted User struct

User Authentication

Information Stored with Each User:

- Username
- Password
- DSSignKey (private key for signing files)
- RSAKey (private key for decrypting files)
- UUID of user's maps in datastore
- A randomly generated key for encrypting/decrypting the maps in datastore
- A randomly generated key for MACing the maps in datastore

All of the above are attributes of the User struct and are stored in datastore; in keystore, we store:

- The public RSA key for encrypting files stored under the string: username + "-Encrypt"
- The public verification key for verifying the signature of files; stored under the string: (username "-Verify")

To support multiple client instances, we only store static data in the User struct, that is, data that should never change. Instead, the User struct contains UUIDs that point to objects in datastore that do hold changing information. This makes it so one user who is changing data can push changes directly to datastore and another user who is logged in as the same user but different client can see those changes.

File Storage and Retrieval

Users have two maps for file storage: one for files they own, and one for files they don't own (files that were shared with them). These two maps combined form the user's filespace. The UUID of the maps is stored in the User struct (this never changes so that way multiple clients can access the files at the same time) and the maps are stored in datastore under the UUID. Both maps map filename to FilePtr (a struct I've defined). The process for storing a file that exists is as follows:

- Get the User's maps from datastore using the UUID in the User struct
- Use the User's keys stored in their User struct for decryption and verifying the MAC of the maps in datastore
- Retrieve the FilePtr in the maps using the name of the file
- The FilePtr contained in the map will be a FilePtr to another FilePtr in datastore
- Use the UUID and the keys in the FilePtr retrieved from the maps to get, decrypt and verify the MAC of the FilePtr stored in datastore
- Now, this FilePtr (the one retrieved from datastore) will contain a UUID that points to a FileChunk (another struct I've defined)
- Using the UUID and the keys in the FilePtr, we get, decrypt, and verify the MAC of the FileChunk under the UUID in datastore
- Now that we have the FileChunk, we iterate over each chunk deleting each chunk from DataStore
- We create a new FilePtr, generate a random UUID, and keys (one for encryption, one for MACing), and store them in this FilePtr
- We create a new FileChunk, and put the content passed into the storefile function in the FileChunk
- Using the data stored in the new FilePtr, we encrypt, MAC, and store the FileChunk in datastore (under the UUID in FilePtr)
- Using the data in the FilePtr from the user's maps, we encrypt, MAC, and store the new FilePtr in datastore

File Storage and Retrieval

For a file that does not exist:

- We create three objects: two are FilePtrs and one is a FileChunk; one FilePtr is stored in the user's maps, while the other FilePtr and FileChunk should be stored in DataStore
- We generate two sets of random keys and UUIDs; one set is stored in each FilePtr
- We put the content in the FileChunk
- Then we encrypt, MAC and store the FileChunk in datastore using the keys and UUID stored in the FilePtr that will be stored in Datastore
- Using the keys and UUID stored in the FilePtr that is stored in the User maps, we encrypt, MAC, and store the other FilePtr in datastore
- The remaining FilePtr is stored in the User's maps
- The maps are then encrypted, MACed, and stored in datastore using the keys and UUID stored in the User struct

To retrieve a file:

- We get the user's maps from datastore using the UUID and keys stored in the User struct, verifying the MAC and then decrypting (two separate keys in User struct)
- We then get the FilePtr associated with the filename from the User's maps
- Using the FilePtr, we get the FilePtr in datastore using the UUID and the keys obtained from the FilePtr in the User's maps; we verify the MAC and then decrypt
- This FilePtr now points us to the head FileChunk
- Again, we use the keys and UUID stored in the FilePtr (the one from datastore) to get, verify the MAC, and decrpyt the FileChunk
- The FileChunk contains content, a FilePtr to the next chunk, and a flag which indicates whether or not this current chunk is the tail of the list
- We loop to traverse the list of chunks, using the keys and UUID stored in the previous chunk to get the next chunk, and appending content as we go from chunk to chunk, reassembling the original message
- Once we hit the tail, we return the message/data to the User

File Storage and Retrieval

Efficient Append:

- Every time we store a file, we add an extra chunk to the front that contains no content ("") and points to the actual first chunk in the list
- When we append, we stick in the new appended chunk between the empty head and the first chunk of actual data; this guarantees that the bandwidth of our append is only dependent on the size of the appended content since nothing extra has to be downloaded or reuploaded to datastore (except for the empty head but this is a small constant)
- We get the user's maps from datastore, use the filename to get the FIlePtr, use that FilePtr to get the next FilePtr, and then use this FilePtr to get the empty head chunk
- We create a new FileChunk
- The data in the empty head chunk (except for the content which is empty) is copied over into the new FileChunk so that the appended FileChunk now points to the actual FileChunk head and has the necessary keys verify the MAC and decrypt
- We put the appended content in the new FileChunk
- We generate a random key for encryption, a random key for MACS, and a random UUID and store this info in the empty head chunk
- Using the UUID and keys in the empty head chunk, we encrypt, MAC, and then store the new chunk in datastore
- Using the keys and UUID in the FilePtr from datastore, we encrypt, MAC, and then store the empty FileChunk head in datastore

Sharing and Revocation

Sharing a File:

- To share a file, we get the user's maps from datastore and retrieve the FilePtr associated with the filename
- We create a new FilePtr (call this FilePtr1) and copy all data in the FilePtr from the User's maps into this FilePtr
- FilePtr1 is also saved in another User map (FilesIShared) to keep track of the invites we've sent out
- We map the shared filename to another map and map the username of the recipient to a FilePtr, in this case, FilePtr1
- We create another FilePtr (call this FilePtr2) that contains the UUID of the first empty head FileChunk in our linked list and the keys used to decrypt and verify this chunk
- We encrypt and MAC FilePtr2 using the keys in FilePtr1 and store FilePtr2 in datastore under the UUID in FilePtr1
- We then encrypt FilePtr1 using the recipient's public RSA (from keystore) and sign using the sender's private sign key (from the User's struct)
- We generate a random UUID and store FilePtr1 in datastore under this UUID and send the UUID to the recipient user
- When the recipient accepts the invite, they go to the UUID sent to them in datastore, get the FilePtr, verify the signature using the public verify key of the sender (from keystore) and decrypt using their private RSA decrypt key (from their User struct)
- They then add this FilePtr to their user's maps (under the new filename given) and delete it from datastore (the deletion is just for efficiency's sake) and that completes the process of accepting the invite since they can now follow the same steps to load this file just like any other

Sharing and Revocation

Revoking a User from Shared File:

- First, we get the user's maps from datastore and use the FilesIShared map to get the FilePtr associated with the filename and user to revoke for that file
- We use the FilePtr (the UUID, encrypt key, and MAC key within) to get the head FileChunk
- We iterate over the list of FileChunks, deleting the previous FileChunk from datastore and reconstructing the message by appending the content of each FileChunk together as we go
- Once we've deleted all the FileChunks, we delete the remaining FilePtr in datastore associated with the user to revoke
- We then remove the entry for the filename in theOwnedFiles map of the user who owns the file
- We call StoreFile() on the name of the file we just removed from our map to reinstate the FilePtr in our map, the FilePtr in datastore, and the head FileChunk in datastore using new randomly generated keys for encrypting and MACing as well as under new randomly generated UUIDs in datastore
- We get the new UUID of the head FileChunk and the keys used to encrypt and MAC it that were created when we called StoreFile(); we retrieve these using the OwnedFiles map to get the FilePtr to the FilePtr in datastore which holds these values
- We then iterate over the remaining users in the FilesIShared map for that filename
- We retrieve the FilePtr stored in datastore for each user and update it to contain the UUID of the new FileChunk and update the keys to the correct decrypt and MAC verify keys
- The revoked user will no longer have access to the file since it was re-encrypted and reMACed using different keys that were shared with all the other users accept the revoked user

Helper Functions

GetStruct():

- This takes as input a decrypt key, a MAC key, and a UUID
- This function returns a marshalled struct
- GetStruct() gets the cipertext and hash stored at the UUID in datastore, uses the MAC key to verify the hash, and the decrypt key to decrypt the struct
- The result is a marshalled struct which is then returned to be unmarshalled and used by the caller

StoreStruct():

- This takes as input marshalled data, an encrypt key, a MAC key, and a UUID
- This function returns nothing (except possibly an error)
- StoreStruct() encrypts the marshalled data using the encrypt key, MACs the ciphertext using the MAC key (appending the hash to the ciphertext), and stores the result in datastore under the provided UUID