Data structures

User (username, hashedPassword, symmetricKey, HMACKey, privateKey, signingKey, filesOwned, filesShared)
File (content)
FileObject (FileUUID, NextFileObjectUUID)
FilePointer (OriginalFileObjectUUID, LatestFileObjectUUID)
FileKeys (fileSymmetricKey, fileHMACKey, ownerUsername, usersSharedWith, filePointerUUID)
InvitationKeys (secretKey, HMACKey, invitationID)
Invitation (FileKeysSymmetricKey, fileKeysHMACKey, fileKeysUUID, invitationSalt, senderUsername)
FileOwnerInfo (fileKeysSymmetricKey, fileKeysHMACKey, fileKeysUUID, invitationSalt)

User Authentication

In order to secure both confidentiality, integrity and authentication for the users, we have to implement a couple of schemes. Since usernames are unique to each user, we will be salting the password of the user with their respective username. The salted password is then hashed, and then finally encrypted using a slow password-based key derivation function.. The User-struct is stored in Datastore, while the user's public key and verification key are stored in the Keystore. Brute force attacks are therefore less viable. By creating a helper function UpdateUser(username, hashedPassword) we can ensure that several instances of the same user can be logged in simultaneously. This function retrieves user data, and is called upon in the start of every main function.

File Storage and Retrieval

Users store their files as File-structs, which are symmetric key encrypted and HMAC-ed through the method *StoreFile*. The encrypted User-struct contains dictionaries called “FilesOwned” and “FilesShared”, which contain key-value pairs. The value is the UUID of either the FileOwnerInfo or InvitationKeys of the file, while the key is the users’ padded and hashed filename for the given UUID. If there is no file in the dictionary with the given filename, then the File-struct is instantiated and the content is added to the Datastore, and the corresponding UUID is stored in the dictionary. If there is a file in the dictionary with the given filename, the file is overwritten instead.

Users retrieve their files from the server through the method *LoadFile*. It first checks whether the user has access to the file, by checking if it is either an owner of the file, or if it is shared with the user. It then uses the filename to look through the user's corresponding dictionary, find the corresponding UUID, and then use that to find the correct instance of the File-struct. The File-structs only contain content. Each File-struct has a corresponding FileObject-struct, containing the UUID of its File-struct, and the next FileObject-struct. The FilePointer-struct contains the first and last FileObject-struct of the file. When calling *LoadFile()*, the function traverses from the FilePointer-struct to the first FileObject-struct, then gets the fileContent recursively from the File-structs, which is then returned to the user in one piece. The FilePointer-struct is symmetric key encrypted and HMAC-ed, while the FileObject-struct is encrypted similarly, using a hash of the FilePointers keys. The same applies for the

File-struct, which is encrypted using a hash of the FileObjects keys. The FilePointers keys are stored in the FileKeys-struct, a junction point to which both the Invitation-struct and FileOwnerInfo-struct contain the UUID and keys of.

Efficient file append is being supported by the synergy between the File, FileObject and FilePointer structs. To append data, only the latest FileObject-struct needs to be updated. Since it only contains UUIDs, it is not dependent on the size of the content in the File-struct. This way, appending is efficient, and doesn't depend on either the filename or the size of the content.

File Sharing and Revocation

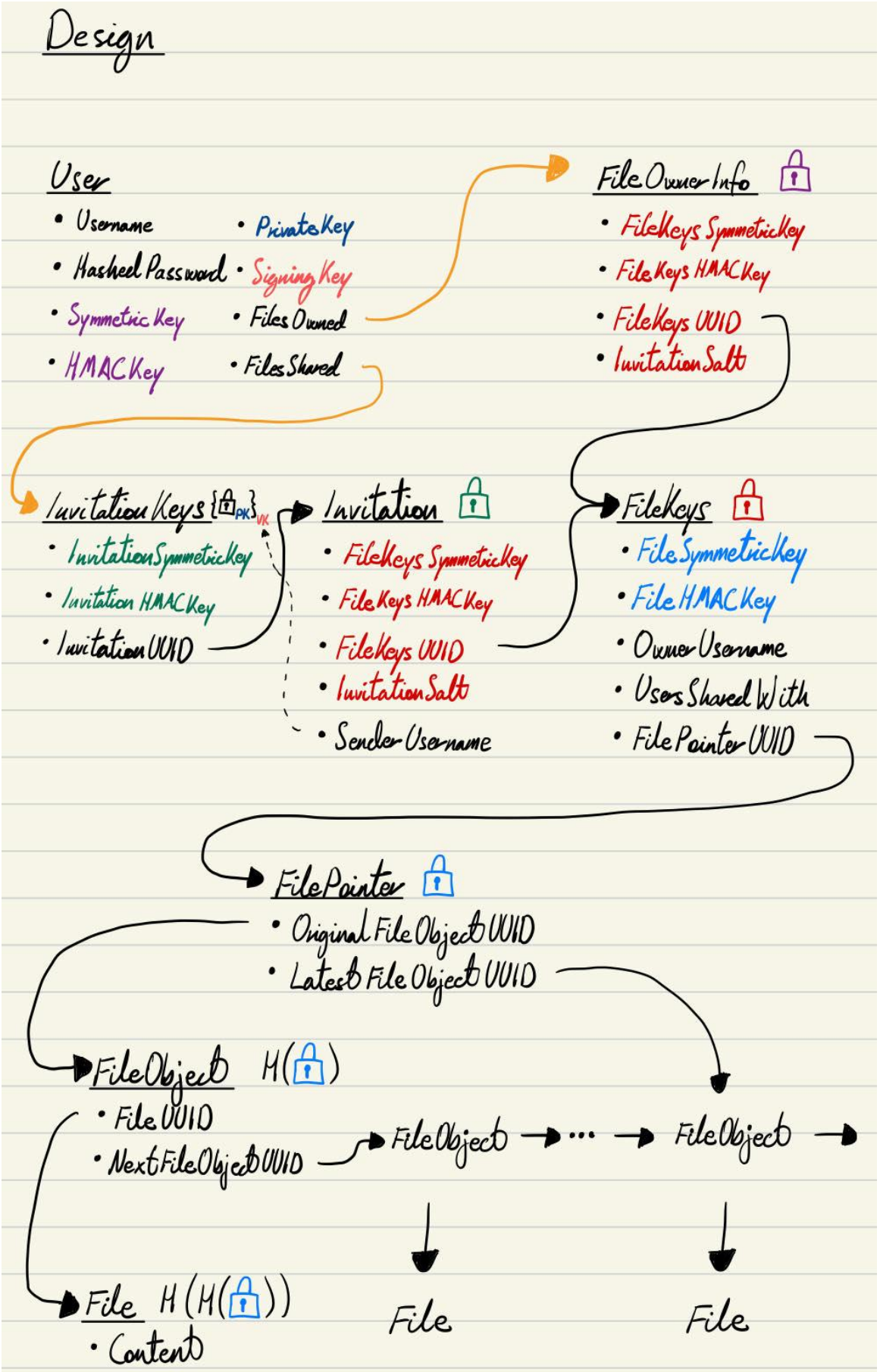
Users share files using the method *CreateInvitation*. It creates a symmetric key encrypted and HMAC-ed Invitation-struct, and a public key encrypted and signed InvitationKeys-struct, and returns the UUID of the InvitationKeys-struct. The UUID is derived from the recipient's username and the InvitationSalt set by the owner in the FileOwnerInfo-struct (unique for the file), which is then added to the Invitation-struct. That way, the UUID is deterministic, unique for all users and files, and allows for invited users to derive new UUIDs when sharing the file with other users. The recipient stores the UUID of the InvitationKeys-struct in their "FilesShared"-dictionary under a padded and hashed filename, which is input by the recipient using the method *AcceptInvitation*. The recipient decrypts the InvitationKeys-struct, uses its keys to decrypt the Invitation-struct, and proceeds to use the SenderUsername field in the Invitation-struct to verify the senders digital signature. The recipient is also added to the FileKeys-struct's field UsersSharedWith, which contains a tree keeping track of who the file is shared with, and by who, represented using a map. Once the files are added to the "FilesShared"-dictionary, the user has access to the shared file.

Only the owner can revoke a user's access to a file using the method *RevokeAccess*. The users' access is revoked, removing them from "UsersSharedWith" in the FileKeys-struct, along with all users the revoked user has shared the file with. The UUID and keys of the all the structs (except UUID of InvitationKeys and FileOwnerInfo) are then changed, before they are updated in the Datastore and in the "FilesShared"-dictionary of all users who still have access. The contents in the structs' previous UUIDs are then deleted to deny revoked users access to the old content. Since the UUID changes, the revoked user does not know the location of the file in Datastore, and can therefore not take any malicious actions on the file.

Helper Methods

We have used a ton of helper methods, and it serves no purpose listing them all here. What they all have in common, though, is that they are used to make encryption and decryption both easier to implement, as we don't need to reuse code, as well as making the code much easier to read. Since our design had its flaws from the beginning, we saved much time by using helper methods. A change in the design therefore only required a small change to a few helper methods.

Diagram


Design
User
Username
Hashed Password
Symmetric Key
HMAC Key
Private Key
Signing Key
Files Owned
Files Shared
File Owner Info
FileKeys SymmetricKey
FileKeys HMAC Key
FileKeys UUID
Invitation Salt
Invitation Keys
Invitation SymmetricKey
Invitation HMAC Key
Invitation UUID
Invitation
FileKeys SymmetricKey
FileKeys HMAC Key
FileKeys UUID
Invitation Salt
Sender Username
FileKeys
File SymmetricKey
File HMAC Key
Owner Username
Users Shared With
File Pointer UUID
File Pointer
Original File Object UUID
Latest File Object UUID
File Object
File UUID
Next File Object UUID
File Object
File Object
File
File
File
Content

Test cases

1) Test that passwords can be empty (3.1.2.c)
    a) Create user with InitUser() with username “Alice” and password “”
    b) Create user with InitUser() username “Bob” and password “michael”
    c) Call GetUser() on Alice and Bob without receiving any errors

2) Test that usernames cannot be empty (3.1.c)
    a) Attempt to create a user without a username and password “123”
    b) The server should return an error

3) Test that users can have the same password (3.1.2.a)
    a) Create user with username “Alice” and password “123”
    b) Create user with username “Bob” and password “123”
    c) Call GetUser on Alice and Bob without receiving any errors

4) Test that files can be empty (3.5.6)
    a) Create a file with StoreFile() with the filename “empty.txt” and the content “”
    b) Create a file with StoreFile() with the filename “hi.txt” and the content “hello”
    c) Call LoadFile() on empty.txt and hi.txt without receiving any errors

5) Test that files can have same name across different users (3.5.7)
    a) Create user Alice and Bob
    b) Log in to Alice’s user
    c) Create a file with StoreFile() with filename “AliceFile.txt” and content “This is Alice’s file.”
    d) Share the file with Bob using CreateInvitation()
    e) Log in to Bob’s user
    f) Accept the invitation using AcceptInvitation(), and setting the filename “BobFile.txt”
    g) Call LoadFile(), using getUUID(“BobFile.txt”) as the UUID
    h) Expected result is “This is Alice’s file.”

6) Test that the file appending efficiency is not dependant on original file (3.7.1)
    a) Create file “small.txt” with content “A”
    b) Create file “large.txt” with content “A” * n; n >> 1
    c) Append “B” to small.txt and large.txt
    d) Time of append to small.txt and large.txt should be roughly the same