Design Document

Data structures

User

Attributes	Purpose
Username	
Password	
Secret_key	The users private key
Signature_private_key	Private key for signature
master_key	Control the master key
hmac_key	Control the hmay key
Files_owned	The files owned by this user stored as a hash to not leak filename length

File

Attributes	Purpose
Content	Content of the file
Next_uuid	The next file in the linked list of files

FileController

Attributes	Purpose
Start	The start of the linked list of files
End	The end of the linked list of files

FileReferenceOwner

Attributes	Purpose
Uuid_shared_with	The users that this file is shared with
Enc_keys_shared_with	The encryption keys the shared user uses to unlock the file
Hmac_keys_shared_with	Hmac keys shared with the user for the file
File_enc_key	The encryption key for the filecontroller and file
Hmac_key	Hmac key for file controller and file
File controller pointer	Uuid of the related filecontroller

FileReferencePrimary

Attributes	Purpose
File_enc_key	The encryption key for the filecontroller and file
Hmac_key	Hmac key for file controller and file
File controller pointer	Uuid of the related filecontroller

FileReferenceSecondary

Attributes	Purpose
File_reference_primary_enc_key	The encryption key for the related filereferenceprimary
Hmac_key	Hmac key for related filereferenceprimary
File controller pointer	Uuid of the related filereferenceprimary

Invitation

Attributes	Purpose
FRPdk	Filereferenceprimary decryption key
FRPhmk	Filereferenceprimary hmac key

User Authentication

A hash of the password is sent to the datastore from the user. The datastore then computes the hash of this hash along with the salt stored. If this matches the stored hash the user is authenticated and the encrypted private key and HMAC is sent to the user. The user can then easily derive the decryption key using Argon2Key(username+hash(password)). If HMACEqual(HMAC from datastore, HMACEval(Argon2Key(username+hash(password)+1))) returns true then the user knows the information has not been tampered with and it can decrypt the private key. This method supports multiple concurrent users as I get updates userdata inbefore every function starts doing its thing, and also uploads uploaded userdata after every function when userdata has been updated.

File Storage and Retrieval

When a user wants to make a new file he will use StoreFile. If no file with the filename they gave exists in the datastore with the user's access it will make a new one. The check is performed by looking up UUID.FromBytes(username+hash(password)+filename) in the datastore. If it does not exist a new file is created by first creating a new entry in the datastore with UUID.New() as UUID and the content added. The content is

encrypted using a random ContentKey. HashKDF(ContentKeyFile, key for HMAC to ensure integrity of file content) is used to calculate an
HMAC. Now, we need to store some information about this new file in our FileReferenceOwner. Its UUID is UUID.FromBytes(username+hash(password)+filename). Stored at this UUID of the FileReferenceOwner will be the related file UUID, an empty dictionary where one will store the users with access and the ContentKey. The key used for calculating the HMAC is stored in the FileReferenceOwner so that anyone accessing the FileReferenceOwner can use this to check that the content in the file is integral. If the file already exists, the user will overwrite what is in the file while also deleting the linked file UUID, finally encrypting the new content with the ContentKey in the corresponding FileReferenceOwner and append the HMAC and then update the FileReferenceOwner with the correct key for the HMAC calculation.

When someone calls LoadFile with the prompt (username+hash(password)+filename) the user will find the FileReferenceOwner in the datastore IF it is the owner of the file. The user can now decrypt the file as he finds both the ContentKey and the UUID pointing to the file in the FileReferenceOwner. Furthermore, the user can use the HMAC-key stored in FileReferenceOwner to check the integrity of the file. What if the user is not the owner of the file but only shared?
The lookup UUID.FromBytes(username+hash(password)+filename) will then instead take the user to a FileReferenceSecondary struct stored in the datastore with the corresponding UUID. This file is encrypted with the receiver's (when sharing) public key and the user can therefore use his own private key to decrypt this FileReferenceSecondary. In the FileReferenceSecondary the user finds the pointer to the FileReferenceOwner, along with the key to decrypt the FileReferenceOwner and an HMAC. This is all that is needed for the user to have access to the file.

When appending to a file a lookup for UUID.FromBytes(username+hash(password)+filename) will be done. If the file exists this UUID will point to the FileDataOwned of the file if it is the owner trying to append. The user will then create a new file with UUID.New() as the UUID and add its content normally as if it was creating a new file, HMACed and encrypted using the ContentKey for this respective filename. The UUID of this file is now inserted into the next file UUID pointer in the file before this new file. If the person trying to append is not the owner it will instead send the user to the
FileReferenceSecondary when querying UUID.FromBytes(username + hash(password) + filename) where the user can access the FileDataOwned and then append in the same way as if the user was the owner. This method supports efficient file append as no files are downloaded when appended, only the new content is uploaded.

File Sharing and Revocation

Now, when a file is shared the main idea is to create either FileReferencePrimary or FileReferenceSecondary depending on whether the sharer is the owner or someone else. If the owner shares the file with someone the owner will create a new FileReferencePrimary and put the username into the dictionary in FileReferenceOwner with corresponding values (username, key for decryption of the new FileReferencePrimary). The new FileReferencePrimary will have UUID.New() as its UUID and contain ContentKey for the file to decrypt the file and check its HMAC, and an HMAC using
HashKDF(username, HMAC_of _FileReferencePrimary) as the key where username is the username of the person it gets shared to and the signature of the owner using DSSignKey. Now an Invitation is created which is encrypted using the receiver's public key and HMACed using HashKDF(username, key for HMAC of invitation) . The UUID is created through UUID.New(). In the invitation is the UUID of the FileReferencePrimary and the key to decrypt it. When the receiver calls the AcceptInvitation, the user searches for any UUID he can decrypt with his personal key. He then finds the UUID of the FileReferencePrimary in the invitation. The user creates a new FileReferenceSecondary with the UUID:
UUID.FromBytes(username+hash(password)+filename) making the user able to search it at a later point. This FileReferenceSecondary is filled with the UUID of the FileReferencePrimary and the key to decrypt it, which then in turn grants the user access to the main file. One then attaches an HMAC of the FileReferenceSecondary with

Argon2key(username+hash(password)+filename, HMAC_of_FileReferenceSecondary) and encrypt it using Argon2Key(username+hash(password)+filename, encryption_of_FileReferenceSecondary). However, if the user sharing the file is not the owner, one will reuse the FileReferencePrimary that the user sharing is related to. The Invitation consists of the UUID of this FileReferencePrimary and the key to decrypt it. Invitation is still encrypted using the receiver's public key and HMACed in the same way as if the sharing user was the owner.

Now, when revoking access the owner moves the file to a new UUID and encrypts it with a new key both generated the same way as when initially creating the file. The owner then checks the dictionary in the FileReferenceOwner file and overwrite the FileReferencePrimary struct connected to the username with the new UUID and new ContentKey of the file while not doing it for the users it wants to revoke access for. This will ensure access for any other branches of the tree while revoking it for a branch and all its children. Finally, the old file is deleted.

Helper Methods

UploadUserdata – Used to upload the updated userdata to datastore after each function that updates userdata

getUserdata – used to fetch the updated userdata before every function begins

SendToDatastore – encrypts and computes hmac and sends something to the datastore

RetrieveFromDatastore – Decypts and checks hmac and retrieves something from the datastore

Diagram from template

UUID	**Encrypted**	**Key derivation**	**Value at UUID**	**Description**
Hash(Username)	Encrypted with the user's master key	Symmetric, deterministic. Argon2Key(username+hash(password)) and Argon2Key(username+hash(password)+1)	User struct jsoned and encrypted and with an hmac appended	The user bytes information
UUID.New() for a new file	The content in the file is encrypted using the ContentKey which is generated from RNG. It is encrypted symmetrically with IV from the RNG.	Will find the ContentKey in the FileReferenceOwner or FileReferencePrimary	Encrypted content, HMAC, UUID for the next linked file if existing and an HMAC.	A file in the datastore, contains encrypted content, HMAC and UUID for another file if this file is linked to another one. There is also an HMAC to ensure the integrity of the file.
UUID.FromBytes(username+Password +filename) for FileReferenceOwner	Yes, symmetrically and deterministically	Key derived using PBKDF(username+hash(password)+filename, encrypt) if the user is the owner of the file	Related file UUID, list of people with access to file, ContentKey and HMAC	A FileReferenceOwner struct found by the owner of a file
UUID.New() for FileReferencePrimary	yes, symmetrical and deterministically	PBKDF(username+hash(password)+filaname), found in the invitation	Related file UUID, ContentKey and HMAC	The struct used when owner shares a file, it references the original file

UUID.FromBytes(username+password+filename)	yes, symmetrical and deterministically	PBKDF(username+hash(password)+filename)	Related FileReferencePrimary UUID, ContentKey and HMAC	The struct used when someone else than an owner shares the file, it references the FileReferencePrimary that was shared to the sharer originally
UUID.New() for Invitation	yes, asymmetrical and randomly	Private key of receiver	The UUID to a FileReferencePrimary and the key to decrypt it	Used when sharing a file with someone
UUID.New() for appended content for a file	The content in the file is encrypted using the ContentKey which is generated from RNG. It is encrypted symmetrically with IV from the RNG.	Will find the ContentKey in the FileReferenceOwner or FileReferencePrimary	Encrypted content, HMAC, UUID for the next linked file if existing and an HMAC.	A file in the datastore, contains encrypted content, HMAC and UUID for another file if this file is linked to another one. There is also an HMAC to ensure the integrity of the file.

Tests

1. Design Requirement 3.2.1
   - 1.1. Log in as Alice
   - 1.2. Log in as Bob
   - 1.3. Expect that both Alice and Bob's application is working properly, that is, all other tests still pass while both are logged in
2. Design Requirement 3.2.3
   - 2.1. Log in as Alice
   - 2.2. Log in as Bob on different device
   - 2.3. Alice shares file with Bob
   - 2.4. Bob shares file with someone else while Alice simultaneously revokes Bob's access
   - 2.5. Expect that the actions are performed serially and not cause a crash, that is, expect that either Bob shares the file first and then Bob's and his children's access is revoked or Bob gets an error when trying to share as he no longer has access
3. Design requirement 3.3.1
   - 3.1. Define countPublicKeys(), a method counting number of public keys in the KeyStore
   - 3.2. Create multiple users
   - 3.3. Share files between them
   - 3.4. Expect countPublicKeys() to be larger than the number of user's registered
4. Design requirement 3.3.4
   - 4.1. Define method countData(), a method that counts the amount of data in the keystore and datastore combined
   - 4.2. Create user
   - 4.3. Upload content to various files
   - 4.4. Expect countData() to never be less than the amount uploaded by the user + the size of all the keys generated in keystore
5. Design requirement 3.6.2

5.1. Create Alice
5.2. Create Bob
5.3. Alice creates file
5.4. Alice shares file with Bob
5.5. Create Lisa
5.6. Create Paul
5.7. Alice shares file with Paul
5.8. Bob shares file with Lisa
5.9. Everyone calls LoadFile()
5.10. Expect that everyone can read the file
5.11. Everyone calls StoreFile() incrementing a counter in the file by 1 each time
5.12. Expect the file content counter to be 4
5.13. Everyone calls AppendToFile with each their respective number
5.14. Expect that when LoadFile is called everyone's number is there
5.15. Everyone calls CreateInvitation()
5.16. Expect the Datastore to contain 4 new Invitations

6. Design requirement 3.6.4
6.1. Define method countFiles(), a method that counts the number of Files in the Datastore
6.2. Create Alice
6.3. Alice creates file
6.4. Expect countFiles() to return 1
6.5. Create Bob
6.6. Alice shares file with Bob
6.7. Expect countFiles() to return 1