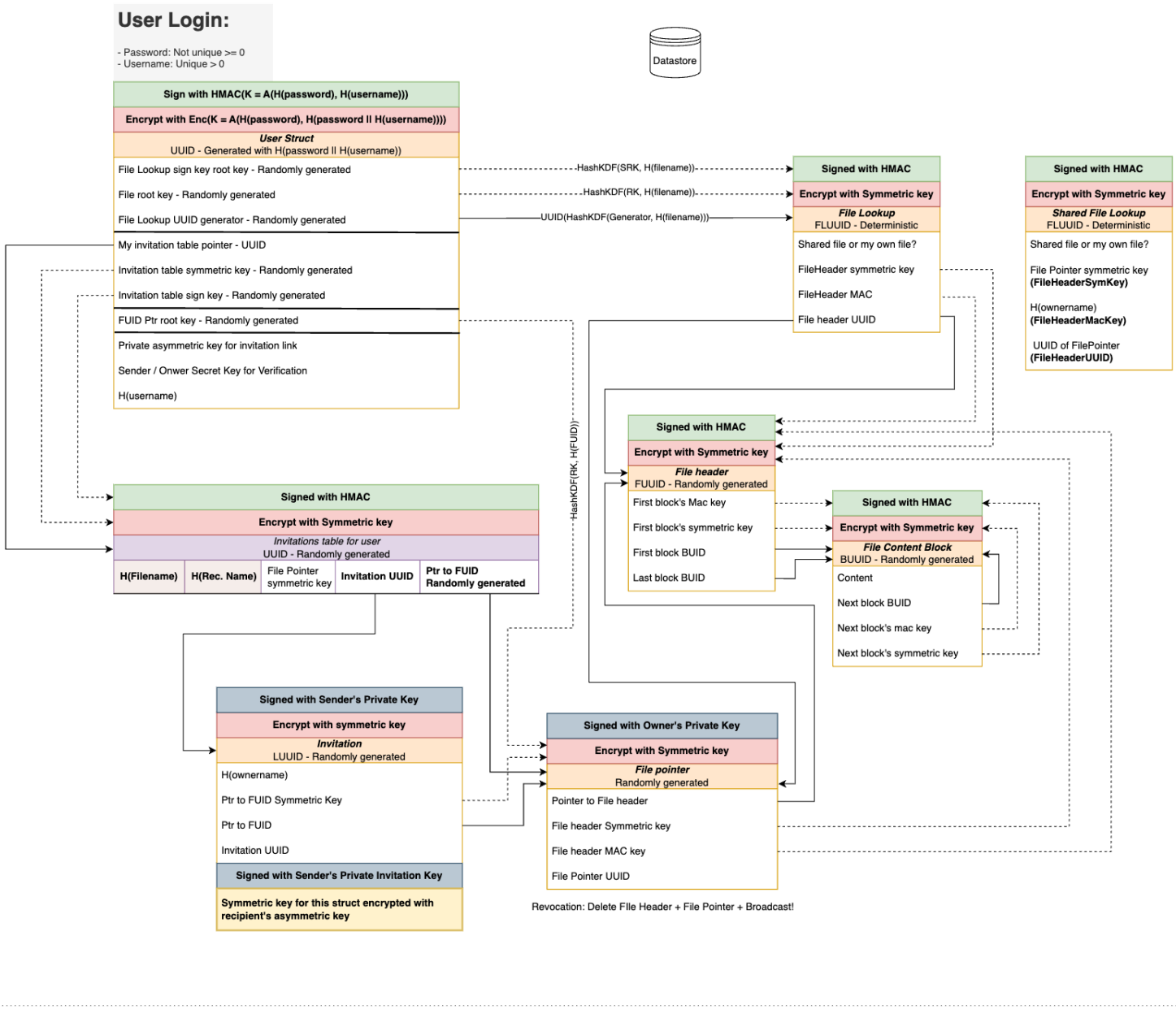
# Project 2 Final Design Document

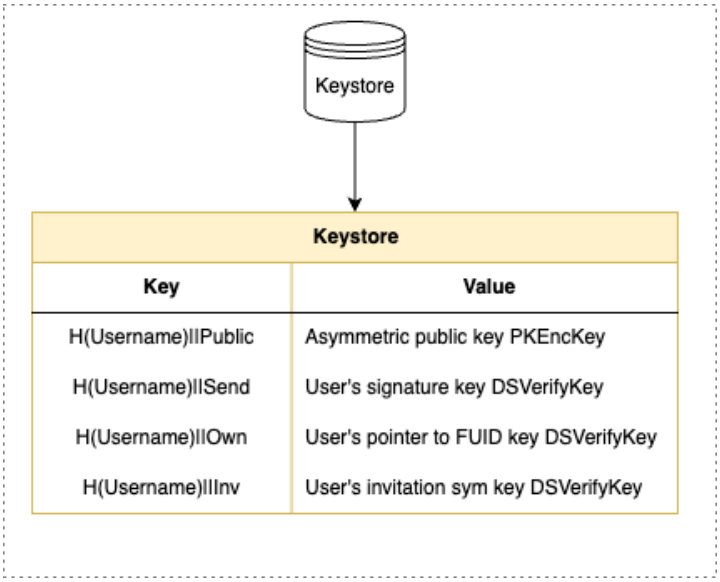

User Login:
- Password: Not unique >= 0
- Username: Unique > 0
Datastore
Sign with HMAC(K = A(H(password), H(username)))
Encrypt with Enc(K = A(H(password), H(password || H(username))))
User Struct
UUID - Generated with H(password || H(username))
File Lookup sign key root key - Randomly generated
File root key - Randomly generated
File Lookup UUID generator - Randomly generated
My invitation table pointer - UUID
Invitation table symmetric key - Randomly generated
Invitation table sign key - Randomly generated
FUID Ptr root key - Randomly generated
Private asymmetric key for invitation link
Sender / Onwer Secret Key for Verification
H(username)
HashKDF(SRK, H(filename))
HashKDF(RK, H(filename))
UUID(HashKDF(Generator, H(filename)))
Signed with HMAC
Encrypt with Symmetric key
File Lookup
FLUUID - Deterministic
Shared file or my own file?
FileHeader symmetric key
FileHeader MAC
File header UUID
Signed with HMAC
Encrypt with Symmetric key
Shared File Lookup
FLUUID - Deterministic
Shared file or my own file?
File Pointer symmetric key
(FileHeaderSymKey)
H(ownername)
(FileHeaderMacKey)
UUID of FilePointer
(FileHeaderUUID)
HashKDF(RK, H(FUID))
Signed with HMAC
Encrypt with Symmetric key
File header
FUUID - Randomly generated
First block's Mac key
First block's symmetric key
First block BUID
Last block BUID
Signed with HMAC
Encrypt with Symmetric key
File Content Block
BUUID - Randomly generated
Content
Next block BUID
Next block's mac key
Next block's symmetric key
Signed with HMAC
Encrypt with Symmetric key
Invitations table for user
UUID - Randomly generated
H(Filename)
H(Rec. Name)
File Pointer symmetric key
Invitation UUID
Ptr to FUID Randomly generated
Signed with Sender's Private Key
Encrypt with symmetric key
Invitation
LUUID - Randomly generated
H(ownername)
Ptr to FUID Symmetric Key
Ptr to FUID
Invitation UUID
Signed with Sender's Private Invitation Key
Symmetric key for this struct encrypted with recipient's asymmetric key
Signed with Owner's Private Key
Encrypt with Symmetric key
File pointer
Randomly generated
Pointer to File header
File header Symmetric key
File header MAC key
File Pointer UUID
Revocation: Delete FIle Header + File Pointer + Broadcast!

## Datastore Design

Structures:

1. User: to store information of a specific user.
2. File Lookup: to store information needed to retrieve file header used to access a file.
3. File Header: to store information about the first file block to access a file and about the last file block for faster appending.
4. File Block: to store content in this block and UUID + keys to get the next file block to retrieve the entire file content; each block has 1024 bytes of content stored.
5. Invitation table: as the owner of all files in this table, the user stores all invitation-related information sent from this user.
6. Invitation: a single invitation sent to the recipient to grant access to a file.
7. File Pointer: for recipients to access the file, and for easier revocation from the owner; stores the shared file's file header UUID and keys to open it; also stores the UUID of this file pointer to prevent swapping attack.

Almost every struct is signed and encrypted with HMAC and symmetric encryption. We use different MAC-keys and symmetric keys for every struct, so no key reuse. An overlook of our design can be seen in the diagram above. Dotted arrows indicate keys and regular arrows indicate uuid pointers.




## Keystore Design

Our key store is fairly simple. We store 4 values for each user as shown in key-value pair below:

1. H(Username)||Public - the asymmetric public key PKEncKey for the user to receive encrypted data from others.
2. H(Username)||Send - DSVerifyKey used by recipients to verify the authenticity and integrity of the invitation data.
3. H(Username)||Own - DSVerifyKey used by recipients to verify the authenticity and integrity of the file pointer ownership.
4. H(Username)||Inv - DSVerifyKey used by recipients to verify the authenticity and integrity of the whole invitation including the symmetric key generated to decrypt this invitation data.

The contents of  the keystore is found in the diagram above.

## User Authentication

The user can authenticate itself with a username and password. We never store the password in any struct. We only store the hash of the username for making invitations, so the receiver can verify who made the invitation with signatures. This hash is stored in the user struct. For InitUser, in order to retrieve the right user, we calculate a slow hash of a combination of the hash of the username and password, and derive the UUID deterministically this way. We store the user struct here, and both encrypt and sign it using different combinations of the hash of the username and passwords, calculate a slow hash over all this, and derive keys from the resulting hashes. All the relevant keys, pointers and information is stored in this user hash, which is never changed. When we call GetUser, all the hashes and the UUID is calculated again, in order to retrieve the User struct, and since this never changes, multiple instances can use it simultaneously. When creating the user, we also save four public keys in the key store. They are

used for authentication and public encryption for invitation purposes. The presence of these keys are used to determine if a user already exists when trying to call InitUser with an already existing username.

## StoreFile

1. Assume Alice is the user.
2. To store the file, the server first checks if a file with the name already exists in Alice's namespace. The server would use the HashKDF function with Alice's unique file UUID root key as the key and Hash(filename) as the salt to get the UUID root number, which would be used to get the unique UUID corresponding to Alice's file using FromBytes.
3. Similarly, the server uses Alice's unique root keys to get the encryption and Mac keys respectively corresponding to this file.
4. The server tries to fetch the bytes from the generated UUID.
    a. If there is no data stored in the result UUID, then Alice is creating a new file as the owner.
    The server creates the File Look Up structure for this file, with attribute shared = false. In this File Look Up, the server also saves the UUID of the File Header Struct it creates for this file, as well as keys to encrypt and Mac the data.
    The server then stores the file's basic information in the File Header Struct. This information includes the UUID and keys for the first File Block as well as for the last File Block. Each File Block contains 1024 bytes of the content from the file. Each File Block also contains the UUID and keys for the next File Block that contains the next 1024 bytes of the file.
    b. If there is data stored in this generated UUID, then the server should first determine if this file belongs to Alice or not based on the shared attribute stored in File Look Up Struct.
    If this file is owned by Alice, then basically the server is going to overwrite the old file with the new contents. The server would not modify File Look Up nor File Header. However, it will create new File Blocks to store the new file content and "forget" the old ones. The File Look Up Struct of this file will then store the new information about the new File Blocks, which are the UUID and keys for the first new File Block as well as the last new File Block.
    c. If there is data stored in this generated UUID, and the File Look Up Struct indicates that the file is actually shared with Alice instead of owned by her, then the server should first fetch the File Header from the File Pointer Struct created by the file owner. To verify the integrity of the File Pointer, the server first gets the owner's verification key from the keystore. After that, the server uses the symmetric stored in the File Look Up to decrypt this File Pointer.
    The File Pointer Struct contains UUID and keys to access the File Header of the file. The server would then follow the similar procedure in (b) to overwrite the old file with new contents by creating and storing new File Block Structs.

## LoadFile

1. Assume Alice is the user.
2. The server first computes the File Look Up Struct UUID and encryption and Mac keys based on Alice respective root keys with the Hash(filename) as the salt.
3. Based on the information retrieved from the File Look Up Struct, the server determines if the file is shared with Alice or not.
    a. If Alice is the owner, the server would then fetch the File Header. In this File Header, the server has the UUID and keys to access the first File Block and retrieves the content stored in this block. The first File Block also stores the UUID and keys needed to access the second File Block, and so on. This process would stop when the File Block UUID matches the last File Block UUID stored in the File Header. At that point, the server knows all contents of the file have been loaded.
    b. If Alice is not the owner, the server would first go to the File Pointer Struct stored in Alice's File Look Up Struct corresponding to the filename. From there, the server follows a similar procedure as StoreFile (c) to get the File Header of this file. Then the server could follow the same procedure in (a) to get the file content.

## AppendToFile

1. Assume Alice is the user.
2. The server would first fetch the File Header based on a similar procedure as above: if Alice is the owner, fetch the File Header directly; if not, get the File Pointer first to get the File Header.
3. Append the new content to the content stored at the last File Block. Then store this combined contents starting at the Last Block. If needed, create new File Block(s) and store the UUID and keys of new File Block(s) in the previous block.
4. Update the File Header to make the UUID and keys pointing to the last File Block for future appends.

## CreateInvitation

1. Assume Alice is the user.
2. If Alice is the owner of the file to be shared, go to Alice's Invitation Table and see if an old invitation for the same filename and recipient already exists. If it does, get the invitation UUID of that invitation and return.
   If the old invitation does not exist, create a new Invitation Struct to store the Hash(Owner Name), UUID and encryption key for the File Pointer corresponding to that file, and the Invitation UUID of itself. The server will then use a randomly generated symmetric key to

encrypt this structure, use Alice's signing key to sign the encrypted Struct, encrypt the symmetric key with the recipient's asymmetric key, append the encrypted symmetric key, and then use another signing key from Alice to sign the entire thing. Add this invitation to Alice's Invitation Table for future use.
The server would also create a new File Pointer corresponding to the file that stores information about the File Header of this file. This Struct is also encrypted and signed.

3. If Alice is not the owner of this file, the server would create a new Invitation Struct from the File Header. However, this Invitation would not be stored in Alice's Invitation Table because she is not the owner. The Invitation would also be encrypted and signed based on the same procedure as in (2).

## AcceptInvitaion

1. Assume Alice is the recipient.
2. If Alice cannot accept this invitation because of errors mentioned in the instruction, the server would throw an error.
3. If Alice can, the server would create a new File Lookup Struct based on the information in the Invitation Struct sent to Alice. In this File Lookup, the attribute Shared would be set to true, and it would also store information needed to access the shared file's File Pointer created by the owner to access the File Header of the file.

## RevokeAccess

1. Assume Alice is the user and the owner.
2. To revoke the access on the file from Recipient B, the server would first get all entries corresponding to that file from the Invitation Table. The server then deletes the entry corresponding to the recipient B as well as the File Pointer Struct pointed by that entry.
3. For other recipients of the file, the server would first move the file content to another place and then update the UUID in their File Pointer Structs to point to the new File Header of the file. This is the broadcasting process of our design.

## Helper methods

In order to limit repetition in our code, we created helper methods.

- verifyMacAndLoadContents()
    - This function takes a pointer to an arbitrary struct, a UUID, MAC and symmetric key, and populates the struct pointed to with the verified and decrypted contents of the datastore at the provided UUID. These steps are repeated in many of our API functions, as we are retrieving structs that are encrypted and MAC'd (often, and this saves us a lot of repeated code.
- encryptThenMac()
    - This function, instead of retrieving a struct from the DataStore, stores a struct to the datastore. It also takes in a pointer to an arbitrary struct, a UUID, MAC and symmetric keys. It encrypts, MACs the struct and stores it in the datastore at the provided UUID. These steps are also repeated many times in our implementation, and saves us a lot of repetition in the code.
- getFileLookupFromFileName()
    - Since every file lookup is deterministically saved given root keys, retrieving the fileLookup takes a lot of steps. This functionality is also used in many of our API functions, and we created a helper function. It takes in a user pointer, filename and a pointer to the file lookup we want to populate with the retrieved information.
- createFileLookup()
    - This saves the file lookup given the security keys, and is the "opposite" of the above function.
- getFilePointer()
    - The retrieval of a FilePointer is needed for every shared file, and is used for both loading and appending a shared file.
- createFile()
    - A subfunction of StoreFile - this function is also used in RevokeAccess, when we need to change the location (essentially delete the file from datastore, and create it again in another location) of the shared file so the revoked user can't access it anymore.
- equalBytes()
    - Used to compare byte slices