Data Structure

```
type User struct {
    Username   string
    MasterKey  []byte
    FileKey    []byte
    PublicKey  userlib.PKEEncKey
    PrivateKey userlib.PKEDecKey
    SignKey    userlib.DSSignKey
    VerifyKey  userlib.DSVerifyKey
}
```

- **Username**
    - A simple string identifier for the user (unique, can not be empty).
- **MasterKey**
    - A symmetric key derived during user initialization, derived from password.
    - Used to derive further keys (enc_key and hmac_key to encrypted userdata)
- **FileKey**
    - A random key.
    - Used to derive further keys (e.g., file list encryption key, file list HMAC key, etc.)
- **PublicKey / PrivateKey**
    - **Asymmetric (RSA-like) keypair** for public-key encryption (PKE).
    - `PublicKey` is shared with others so they can encrypt secrets for this user.
    - `PrivateKey` is kept secret and used to decrypt data encrypted with the user's public key.

- **SignKey / VerifyKey**
    - **Asymmetric (DSA-like) keypair** for digital signing (DS).
    - `SignKey` is used to create digital signatures.
    - `VerifyKey` is the corresponding public key used by others to verify those signatures.

```
type FileView struct {
    MetadataUUID uuid.UUID
    EncKey       []byte
    HMACKey      []byte
    Status       string // Own | Shared | Received
    PendingInv   map[string]uuid.UUID // recipientUsername → invitationPtr
}
```

A concise "handle" for a file from the perspective of a particular user.

- **MetadataUUID**
    - A pointer (UUID) into the **datastore** for the file's metadata record (FileMetadata).
- **EncKey** and **HMACKey**
    - Symmetric keys used for **encrypting** and **ensuring integrity** of the FileMetadata.
- **Status**
    - A string indicating the user's relationship to the file. Possible values:
    - **Own**: The user is the original owner.
      Shared: The user has shared the file with someone else (i.e., the user is an owner, but has provided access to others).
    - **Received**: The user has received access to the file from someone else.
- **PendingInv**
    - Record the invitation that the recipient does not accept.

```
type FileMetadata struct {
    Owner          string
    FileName       string
    HeadPtr        uuid.UUID
    TailPtr        uuid.UUID
    NunmberChunk   int
    FileEncKey    []byte
    HMACKey       []byte
    ShareListAddr uuid.UUID
    Version       uint64
}
```

Stores detailed information about a file in the system, including chunk management and sharing references.

- **Owner**
    - The username (string) of the file's original owner.
- **FileName**
    - Logical name of the file, as intended by the owner.
- **HeadPtr / TailPtr**
    - UUIDs referring to the first and last `FileChunk` in a linked list of chunks.
    - This approach allows large files to be split into multiple chunks that can be appended or retrieved individually, which satisfies the bandwidth requirement in `AppendToFile`
- **FileEncKey / HMACKey**
    - Symmetric keys to encrypt and verify integrity of the **file's chunks** (not the same keys as in `FileView`).
    - `FileEncKey`: Encryption key for chunk data.
    - `HMACKey`: HMAC key to detect tampering on chunk data.
- **ShareListAddr**
    - Points to a `SignedShareList` struct that tracks how and with whom the file has been shared.
- **Version**
    - A version number for concurrency control and updates.
- **NumberChunk**
    - Keep track of the number of chunks in the chunk list.

```
type FileChunk struct {
    Data  []byte
    Next  uuid.UUID
}
```

Represents an individual piece (chunk) of a file, linked in a chain.

- **Data**
    - The chunk data.
- **Next**
    - The UUID of the next chunk in the linked list.
    - If this UUID is "empty," it indicates the end of the file.

```
type ShareEntry struct {
    Sender       string
    Recipient    string
    FileKey []byte
    MetadataUUID uuid.UUID
}
```

Keeps track of an instance where a file's master key has been shared from one user to another, linking them to the file metadata. It is stored in the `SignedShareList` (Pointed by `FileMetadata.ShareListAddr`) after the recipient accepts the invitation.

- **Sender / Recipient**
    - Strings identifying who shared the file and who received it.
- **FileKey**
    - The `FileKey` of the **recipient**.

    - This way when revoking the file from the recipient user, the owner could delete the original shared file in the file list of the recipient.
- **MetadataUUID**
    - Points to the `FileMetadata` for quick reference.

```
type Invitation struct {
    EncView      []byte // AES (or other symmetric) encrypted FileView
    EncryptedKey []byte // RSA (or other asymmetric) encrypted symmetric key
    SenderSig    []byte
}
```

Used during the invitation phase of sharing a file. One user (sender) sends this to another (recipient) who can then accept it to gain access.

- **EncView**
    - The serialized `FileView` for the shared file, **symmetrically** encrypted by a **symmetric key** `encKey`.
- **EncryptedKey**
    - `encKey` encrypted with the **recipient's public key**.
- **SenderSig**
    - A digital signature (using the **sender's SignKey**) over EncView, enabling the recipient to verify authenticity and integrity.

```
type SignedShareList struct {
    List map[string][]ShareEntry
}
```

Maintains a record of how a file has been shared. Pointed by `FileMetadata.ShareListAddr`.

- A mapping of sender usernames to a list of `ShareEntry` objects.
- For each sender in the map, there is a slice (array) of `ShareEntry` records enumerating who they have shared the file with.

InitUser Function

Purpose: Creates and initializes a new user in the system.

Steps:
1. Validate that username isn't empty
2. Generate a UUID from the hashed username
3. Use UUID to checks if username already exists in Datastore
4. Create cryptographic keys:
    a. Use `argon2Key` to generate `masterKey` from password and a unique, long and random salt
    b. Use `DeriveKeys` to derive encryption (`encKey`) and HMAC key (`macKey`) from `masterKey` deterministically
    c. Public/private key pair using PKE
    d. Digital signature key pair
5. Store public keys in the keystore
6. Use `encKey` and `macKey` to symmetric encrypt and HMAC the user data
7. Store encrypted user data in Datastore along with salt and HmacTag
   `DatastoreSet(userUUID, salt|userByteEnc|userHmacTag)`

GetUser Function

Purpose: Retrieves and authenticates an existing user.
Steps:
1. Generate UUID from hashed input username
2. Retrieve encrypted user data along with salt and HmacTag from Datastore

3. Use `argon2Key` to generate `masterKey'` from input password and salt, then derive `encKey'` and `macKey'`
4. Verify HMAC integrity check. If it fails, either the password is invalid or the data is tampered.
5. Decrypt user data using `enc_key`

StoreFile Function

Purpose: Stores a file in the system, either creating a new file or overwriting an existing one.

Steps:

1. Load Current User's File List
    a. Create `userFileListID` using username deterministically (`uuid.FromBytes(Hash(senderUsername + "fileList")[:16])`(since **each user will only have one unchanged file list**).
    b. Derive keys for file list (`fileListEncKey`, `fileListHMACKey`) using user `FileKey` deterministically (since **each user will only have one unchanged file list**).
    c. Retrieve, verify and decrypt the user's file list from Datastore using the above keys.
    d. If it doesn't exist, create a new file list.
2. For new files:
    a. Generate new file metadata UUID and file chunk UUID randomly **(after revoking some recipient from accessing the file, need to generate new metadata UUID and chunk UUID**).
    b. Derive keys for chunks (`FileEncKey`, `FileHMACKey`) randomly (**file chunks may need to re-encrypt with new keys)**
    c. Use the above keys to encrypt the chunk with HMAC protection. We **include the chunk's UUID as part of the input to the HMAC computation**. `chunkHMAC = HMACEval(fileHMACKey, chunkEnc||ChunkUUID)` This is done to bind the integrity verification to a specific file chunk, making the HMAC unique not only to the content but also to its identity. By this design, we can detect **malicious mix-ups of chunks**, as the HMAC will fail to validate if the wrong UUID is used
    d. Store the encrypted chunk with hmacTag: `DatastoreSet(chunkUUID,ChunkEnc||ChunkHMAC)`
    e. Store `fileEncKey` and `fileHMACKey` in file metadata.
    f. Builds file metadata structure, with the head pointer pointing to the first chunk and the tail pointer pointing to the empty chunk at the end of the chunk list
    g. Derive keys for metadata (`metadataEncKey`, `metadataHMACKey`) to encrypt and store metadata with integrity protection `DatastoreSet(metadataUUID,metadataEnc||metadataHMAC)`
    h. Store `metadataEncKey` and `metadataHMACKey` in file view
    i. Creates a new file view entry in the user's file list
3. For existing files, need to **overwrite** them:
    a. Deletes all existing chunks
    b. Creates **new initial chunk (new UUID)** with fresh encryption
    c. Updates metadata with new chunk pointers
4. Encrypt user's file list with `fileListEncKey` and `fileListHMACKey`, and store it in Datastore

AppendToFile Function

Purpose: Appends data to an existing file.

Steps:

1. Load Current User's File List
    a. Create `userFileListID` using username deterministically (`uuid.FromBytes(Hash(senderUsername + "fileList")[:16]).`
    b. Derive keys for file list (`fileListEncKey`, `fileListHMACKey`) using user `FileKey` deterministically.
    c. Fetch and decrypt the file list, verifying its integrity with the HMAC.
    d. If the file list does not exist or cannot be decrypted, return an error.
2. Verify the file is stored in the file list (check if the corresponding `FileView` is stored in the file list).
3. Retrieve and verify integrity of the file metadata using the `metadataEncKey`, `metadataHMACKey` stored in `FileView.`
4. Create a new chunk for the appended data
    a. Encrypt chunk with `FileEncKey`, `FileHMACKey` with HMAC protection
    b. **Update tail pointer of metadata** to new location (same as the `Next` in chunk)
5. Updates file metadata (version number, chunk count)

6. Stores the new chunk and updated metadata similar to the steps in `StoreFile`.

Key Points:

Uses same encryption/HMAC keys as original file

LoadFile Function

Purpose: Retrieves and decrypts a stored file.

Steps:

1. Create `userFileListID` using username deterministically (`uuid.FromBytes(Hash(senderUsername + "fileList")[:16])`
2. Derive keys for file list (`fileListEncKey`, `fileListHMACKey`) using user `FileKey` deterministically
3. Load, decrypt, verify integrity of the user's file list using the above keys.
4. Verify the file is stored in the file list (check if the corresponding `FileView` is stored in the file list).
5. Retrieve and verify integrity of the file metadata using the `metadataEncKey`, `metadataHMACKey` stored in `FileView.`
6. Follows the **chain of chunks from head to tail**:
    a. Verifies each chunk's integrity via HMAC using `FileHMACKey`
    b. Decrypts chunk using `FileEncKey`
    c. Concatenates chunk data
7. Returns the complete file content

Key Points:

Follows secure linked list structure.

CreateInvitation Function

Purpose: Securely creates an invitation so that one user (the sender) can share a file with another user (the recipient).

Steps:

1. Verify Recipient's Public Key
    a. Retrieve the recipient's public key from datastore. (`recipientUsername + "_PK"`).
    b. If the key does not exist in the keystore, return an error.
2. Load the Sender's File List
    a. Derive `fileListEncKey` and `fileListHMACKey` using `DeriveKeys`, based on the sender's `FileKey` and constant salts.
    b. Compute the sender's FileList UUID (`uuid.FromBytes(Hash(senderUsername + "fileList")[:16]`)).
    Fetch and decrypt the file list, verifying its integrity with the HMAC.
    If the file list does not exist or cannot be decrypted, return an error.
3. Check That the File to be Shared Exists
    a. Look up the `FileView` corresponding to the filename in the sender's file list.
    b. If it does not exist, return an error.
4. Load and Verify File Metadata
    a. From the `FileView` , get its `MetadataUUID`, `fileEncKey`, and `fileHMACKey`.
    b. Decrypt the file metadata (`LoadFileMetadata`) to ensure it's valid and not tampered with.
5. Prepare `FileView` Copy
    a. Create a copy of the `FileView` that will be shared (`fvcopy`).
    b. Set Status to "Share".
6. Symmetrically Encrypt `fvcopy` : Generate random keys (`ShareSymKey, ShareHMACKey`) to encrypt and HMAC the `fvcopy` before storing it in datastore (stored at `viewCopyAddr` for reference).
7. Construct and Sign the Invitation (**Hybrid Encryption**)
    a. Serialize `fvcopy` .
    b. Hybrid encrypt that data with the recipient's public key (`HybridEncrypt` uses a symmetric key to encrypt `fvcopy`, and then uses the recipient's public key to encrypt the symmetric key, returns `encryptedKey` + `encryptedView`).
    c. Sign the `encryptedView`with the sender's digital signature key (`DSSign`).

    d. Build an Invitation struct containing: `encryptedKey` , `encryptedView`, `SenderSig` (the sender's signature).
8. Create and Store Invitation
    a. Generate a new UUID (`invID`) for the invitation.
    b. Add user into `pendingInv`.
    c. Serialize the Invitation and store it in datastore.
    d. Return `invID` to the caller.

Key Points

- Hybrid encryption is used to protect the `fvcopy` so that only the intended recipient can decrypt it.
- The sender's signature (`SenderSig`) allows the recipient to verify the invitation's authenticity.

AcceptInvitation Function

Purpose: Allows a recipient to accept a previously created invitation securely, retrieve the shared file's metadata, and add it to their own file list.

Steps:

1. Verify Sender's Signature Key
    a. Retrieve the sender's verification key (`senderUsername + "_DS"`).
    b. If it does not exist in the keystore, return an error.
2. Get and Parse the Invitation
    a. Fetch the serialized `Invitation` object from the datastore through `invitationPtr`.
    b. Unmarshal it into an `Invitation` struct.
3. Verify the Invitation's Signature
    a. Use `DSVerify` to check `Invitation.SenderSig` over `Invitation.EncView`.
    b. If the signature check fails, reject the invitation.
4. Decrypt the Encrypted `fvcopy`
    a. Call `HybridDecrypt` with the recipient's private key to decrypt:
        i. The symmetric key portion (`EncryptedKey`)
        ii. The actual file view data
    b. Unmarshal the resulting plaintext bytes into a `FileView`.
5. Retrieve or Create Recipient's File List

    Create `userFileListID` using recipient's username deterministically (`uuid.FromBytes(Hash(senderUsername + "fileList")[:16])`

    Derive keys for file list (`fileListEncKey`, `fileListHMACKey`) using recipient's `FileKey` deterministically

    Attempt to load and decrypt existing file list (if it exists). Otherwise, use an empty list.
6. Add the `fvcopy` to Recipient's File List
    a. Check if the recipient already has a file with the requested filename. If so, return an error.
    b. Otherwise, set the `fvcopy.Status` to "Received" and add it under filename in the file list.
7. Remove the invitation data from the datastore so it cannot be reused.
8. Load and Verify File Metadata
    a. Fetch the FileMetadata for the shared file from the `MetadataUUID` in the newly added `fvcopy`.
    b. Decrypt and verify its HMAC to ensure integrity.
9. Update the Shared File's ShareList
    a. Retrieve the `SignedShareList` from the `ShareListAddr` in the file metadata.
    b. Check for any existing share entry to prevent duplicate shares.
    c. Add a new `ShareEntry` (with the recipient's username, the recipient's `FileKey`, the recipient's filename and the `MetadataUUID`).
    d. Re-marshal and store the updated `SignedShareList`.
10. Re-encrypt (SymEnc) the updated FileMetadata and store it back under its UUID with a valid HMAC.
11. Persist the Recipient's Updated File List
    a. Re-encrypt and HMAC the entire file list.
    b. Store it back in the datastore under its UUID.

Key Points

- By verifying the sender's signature, the recipient ensures the invitation is truly from that sender and not forged.
- Hybrid decryption ensures only the recipient can access the shared file details.
- Updating the `ShareList` ensures all future operations on the file can be tracked and verified.

RevokeAccess Function

Purpose:
Removes a recipient's access to a shared file and ensures that all users who received the file downstream from that recipient are also revoked.

Steps:

1. Load Current User's File List
   a. Create `userFileListID` using username deterministically (`uuid.FromBytes(Hash(senderUsername + "fileList")[:16])`
   b. Derive keys for file list (`fileListEncKey`, `fileListHMACKey`) using user `FileKey` deterministically
   c. Fetch and decrypt the file list, verifying its integrity with the HMAC.
   d. If the file list does not exist or cannot be decrypted, return an error.
2. Validate File Existence and Ownership
   a. Look up the filename in the user's file list.
   b. If the file is not found, return an error.
   c. Load the associated `fileMetadata` using its UUID and keys stored in the `FileView`.
   d. Verify the user is the file owner (i.e., `metadata.Owner` matches `userdata.Username`).
   e. If not the owner, return an error.
3. Load and Verify the `ShareList`
   a. Fetch the `signedShareList` from the address stored in `fileMetadata.ShareListAddr`.
   b. Ensure the list is initialized properly (initialize with empty map if nil).
4. Locate Original Share Entry
   a. Find the share entry where `Sender == userdata.Username` and `Recipient == recipientUsername`.
   b. If no valid `ShareEntry` is found for the recipient (i.e., no official share from the current user to the recipient exists), do not immediately return an error.
   c. Instead, check the current user's `pendingInv` map (from the `FileView`) to see **if the recipient has been invited but has not yet accepted the file**.
   d. If the recipient is found in `pendingInv`, revoke the pending invitation:
      - Delete the invitation entry from the Datastore (using the UUID stored in `pendingInv`).
      - Remove the recipient from the `pendingInv` map.
      - Update the `FileView` status to "Own".
   e. Save the updated file list for the current user.
   f. If the recipient is not found in `pendingInv`, only then return an error indicating that no valid share or invitation was found.
5. Traverse Downstream Recipients **(BFS)**
   a. Initialize a queue with the `recipientUsername`.
   b. For each user in the queue:
      - Find all users they shared the file with (from `ShareList`).
      - Mark these entries as revoked (add to revokeUsers).
      - Append **downstream recipients** to the queue for further traversal.
   c. Keep track of all valid (non-revoked) share entries in remainUsers.
6. Remove `FileView` from Revoked Users' File Lists
   For each user in revokeUsers:
   a. Create `userFileListID` using username (`uuid.FromBytes(Hash(senderUsername + "fileList")[:16])`
   b. Derive keys for file list (`fileListEncKey`, `fileListHMACKey`) using user `FileKey` deterministically
   c. Locate the share entry pointing to the file to retrieve the recipient's `FileKey` and filename.
   d. **Delete the file entry** (keyed by filename) from their list.
   e. Re-encrypt and store their updated file list in the data

7. Re-encrypt File Chunks with New Keys
    a. Generate a new `fileMetadata` object with:
    b. New `metadataEncKey`, `metadataHMACKey`, `fileEncKey` and `fileHMACKey.`
    c. For each chunk:
        - Load and decrypt with the old chunk key.
        - Encrypt with the **new chunk key** `fileEncKey`
        - Store the updated chunk with a **new UUID**.
    d. Update the `fileMetadata`'s ChunkUUIDs accordingly.
    e. Using new keys `metadataEncKey` , `metadataHMACKey` to encrypt the updated `fileMetadata`.
    f. Store the updated encrypted `fileMetadata`.
8. Distribute Updated FileViews to Valid Users
   For each user in remainUsers:
    a. Derive their file list UUID and encryption/HMAC keys.
    b. Load and decrypt their UserFileList.
    c. Create a new `FileView` (pointing to updated `fileMetadata`).
    d. Set Status to "Shared".
    e. Add the new entry under their preferred filename.
    f. Re-encrypt and store their file list.
9. Update the Shared File's `ShareList:` Replace the existing list in `signedShareList` with remainUsers
10. Delete the old `sharList`, chunk list and `fileMetadata`

Design Question: Datastore Adversary

1. **Storing a Hashed Password in Datastore Won't Work:**
Datastore adversaries can read or overwrite H(password) to log in as a user. They can also read, modify, or delete other stored files in Datastore if not securely encrypted. Also, if the encryption key is stored in Datastore unprotected, the adversary can extract it and decrypt all user files.

2. Secure Design for Storing Information in Datastore
Encrypted User Data:
Store encrypted user data. `enc_key` and `mac_key` are not stored as they can be derived again from user `masterKey`.

File Encryption & Secure Storage:
File chunk is encrypted with a random key and generates a HMACTag (**include the chunk's UUID as part of the input to the HMAC computation)** before storing. And the keys are stored in File metadata.
File metadata is encrypted with another random key and generates a HMACTag before storing. And the keys are stored in FileView.
FileView is stored in the FileList.
FileList is encrypted with a key derived from the user `FileKey` and generates a HMACTag before storing. The keys are not stored as they can be re-derived from the `FileKey` .

File Sharing
`fvcopy` is encrypted using random keys and generates a HMACTag before storing. The keys are not stored.

Integrity Protection Against Tampering
Every stored object has a HMACTag. Upon retrieval, the tag is verified. HAMC check failure indicates modification.

Design Question: Helper functions

1. **Authenticated Encryption Helper:**
Combines encryption and integrity protection.

`AuthEncrypt`:
Encrypts data and generates integrity tag. Takes a key, plaintext as input. Outputs ciphertext and hmacTag.

`AuthDecrypt`:
Decrypts data and verifies integrity. Takes a key, ciphertext, hmacTag as input. Outputs plaintext.

2. **Easy Encryption Helper:**
Easy encryption and integrity protection using input encKey and HmacKey

`EasyEncrypt`:
Encrypts data and generates integrity tag. Takes encKey, HmacKey, plaintext as input. Outputs ciphertext and hmacTag.

`EasyDecrypt`:
Decrypts data and verifies integrity. Takes encKey, HmacKey, ciphertext, hmacTag as input. Outputs plaintext.

3. **Hybrid Encryption Helper:**
Combines symmetric encryption with public key encryption

`HyEncrypt`:
Uses public key to encrypt a random symmetric key, uses the symmetric key to encrypt data. Takes publicKey, plaintext as input. Outputs encryptedSymKey, encryptedData.

`HyDecrypt`:
Uses private key to decrypt the symmetric key, uses the symmetric key to decrypt data. Takes privateKey, encryptedSymKey, encryptedData as input. Outputs plaintext.

4. **Key Derivation Helper**:
Generates keys from a password or master key.

`DeriveKeys`:
Takes password-derived key, context (eg., “Encryption”, “HMAC”). Outputs encryption key, HMAC key.

5. **File Metadata Helper**:
Stores and retrieves file metadata securely.

`SaveFileMetadata`:
Stores metadata with integrity protection. Take UUID, metadata, encKey, HmacKey as input.

`LoadFileMetadata`:
Loads and verifies metadata. Take UUID, encKey, HmacKey as input. Output file metadata.

6. **User List Helper**:
Stores and retrieves user list securely.

`SaveUserList`:
Stores user list with integrity protection. Take UUID, metadata, ListEncKey, ListHmacKey, fileList as input.

`LoaduserList`:
Loads and verifies metadata. Take UUID, ListEncKey, ListHmacKey as input. Returns a file list.

7. **File Chunk Helper**:
Stores and retrieves file chunk securely.

`SaveFileChunk`:
Stores file chunk with integrity protection. Take UUID, filechunk, fileEncKey, fileHmacKey as input.

`LoadFileChunk`:
Loads and verifies metadata. Take UUID, fileEncKey, fileHmacKey as input. Returns a file chunk.

Design Question: User Authentication

`InitUser()`:

If username already exists in Datastore, return an error. Follow the steps in *PasswordDerived Key (PDK) Generation* part to derive master key, encryption key (`enc_key`) and HMAC key (`mac_key`). Encrypt user metadata with `enc_key`. Derive `hmacTag` with encrypted metadata and `enc_key`. Store salt, encrypted metadata, and `hmacTag` in Datastore. Create a User object in memory and return its pointer.

`GetUser()`:
If username doesn't exist in Datastore, return an error. Retrieve salt, encrypted user metadata and `hmacTag` from Datastore. Use the input password and salt to derive new master key. Use the new master key to derive `enc_key2` and `mac_key2`. Generate `hmacTag2` with encrypted user metadata and `mac_key2`. Verify `hmacTag2` with `hmacTag`. If yes, decrypt the user metadata, create and return the User object. Otherwise, return an error. If the password is incorrect, it will not pass the HMAC tag verification.

Design Question: Multiple Devices

All file operations interact with Datastore. File content is stored in chunks, each with a unique UUID. Metadata tracks chunk list and version number for atomic updates.

1. Centralized Data Storage:
Whenever a file is stored or updated on one device, the change is immediately saved to this datastore.

2. Atomic Operations with Versioning
File Metadata Versioning: Each file's metadata includes a version number. When a change is made (e.g., appending data), the metadata version is incremented.

Example Workflow for EvanBot:
On Laptop: EvanBot stores a file. This file's data is encrypted and stored in the datastore, and the file metadata is updated (including versioning).
On Phone: The phone loads the latest file metadata and file content.
Conflicts: If both devices modify the file simultaneously, both modifications will be stored.

Design Question: Efficient Append

1. Data Downloaded (DataStoreGet)

- User file list: Required to locate and update file metadata (Size: O(1)).
- File Metadata: Required to locate and update chunk references (Size: O(1)).

Total DataStoreGet Bandwidth: Constant (small fixed size).

2. Data Uploaded (DataStoreSet)

- New Encrypted Chunk (Size: |append_data| (scales linearly with append size))
- Updated File Metadata, includes new ChunkUUID, new pointers (Size: O(1)).

Total DataStoreSet Bandwidth: |append_data| + O(1) (small fixed metadata overhead).
Total Bandwidth = (Size of metadata fetched (constant)) + (|append_data| + metadata size)

Design Question: File Storage and Retrieval

Please refer to FileStore and FileLoad functions.

Design Question: File Sharing

`CreateInvitation()`

1. What gets created and What changes in Datastore?

- A new FileView copy (`fvcopy`) is generated, pointing to the existing file's metadata with Status = "Share". It is marshalled and symmetrically encrypted, along with an HMAC to ensure integrity.
  The encrypted `fvcopy` is stored in the Datastore under a new random UUID.

- An invitation struct is created, containing the `fvcopy` and a signature from the sender to verify authenticity. This invitation is then hybrid encrypted with the recipient's public key and stored in the Datastore.

2. **The returned UUID** is the pointer to the invitation object stored in the Datastore.

`AcceptInvitation()`

1. What changes when CodaBot calls AcceptInvitation?

- A new entry is added to CodaBot's own `UserFileList`, under a chosen filename, pointing to the `FileView`.
- The invitation is deleted from the Datastore after acceptance.
- Add CodaBot to `signedSharList` and save the updated `signedSharList.`

2. How does CodaBot access the file later?

CodaBot looks up his `UserFileList`, finds the `FileView`, and uses the stored `MetadataUUID`, `EncKey`, and `HMACKey` to load the `FileMetadata`, the linked file chunks, and reassemble the file contents.

CodaBot (non-owner) shares the file with PintoBot

Same step as above.

Design Question: File Revocation

1. What Happens When A Revokes B's Access?

B, D, E, and F lose access. C and G retain access. B cannot decrypt the file or track future updates.

2. Values Updated in Datastore

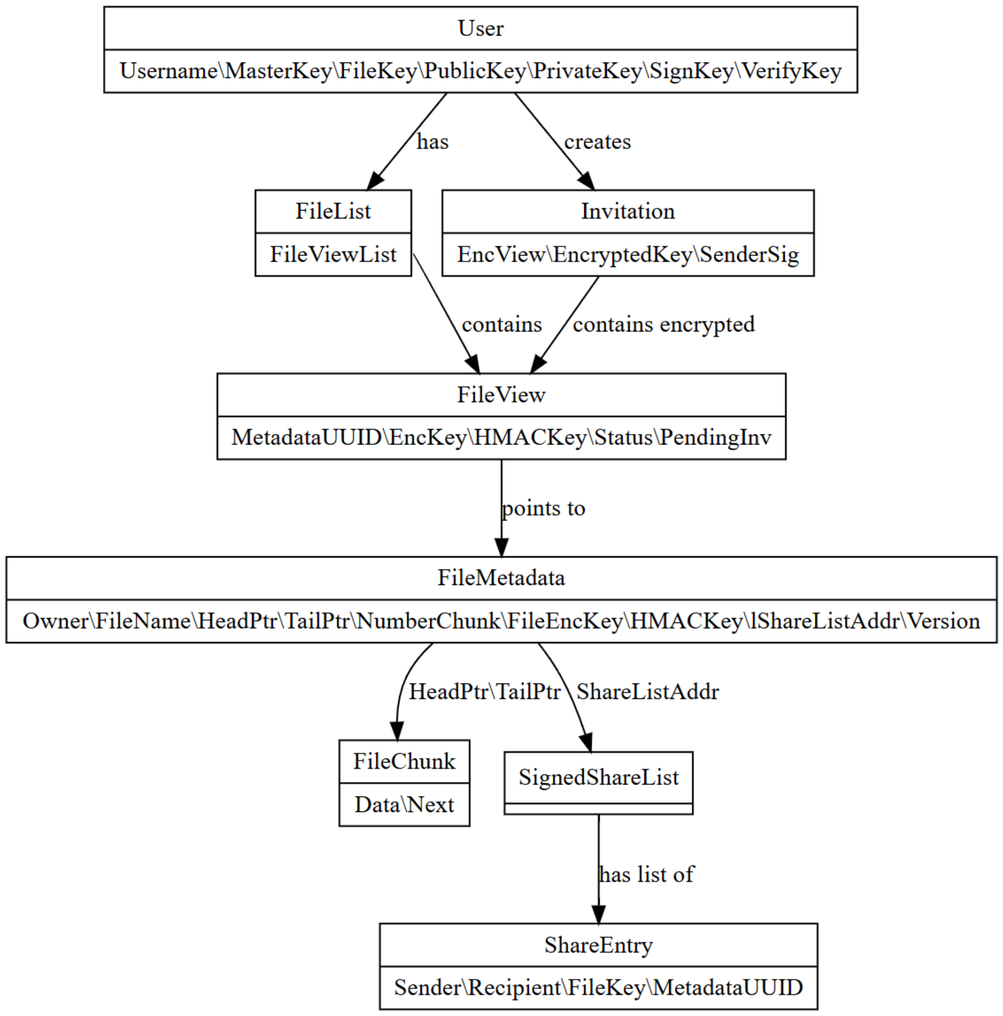
1. `FileMetadata` and all associated chunks are re-encrypted with newly generated encryption and HMAC keys, rendering any previously held keys useless (Prevents B from using any previously obtained key). The updated `FileMetadata` is saved under a new UUID.
2. A fresh `SignedShareList` is created that excludes B and all downstream users (such as D, E, and F), while preserving entries for valid users like C and G.
3. FileViews containing the new keys and metadata pointers are issued to these valid users and stored in their respective `UserFileList`.
4. The revoked users' `UserFileList` entries for the file are deleted.

These steps ensure that B and their downstream users cannot decrypt or modify the file–even if they saved old values–since all cryptographic materials have changed. HMACs protect data integrity, so any tampering is detectable. Additionally, because new UUIDs and keys are derived using secure randomness and not guessable, revoked users cannot track or infer when the file is updated, preserving forward secrecy and access pattern privacy.

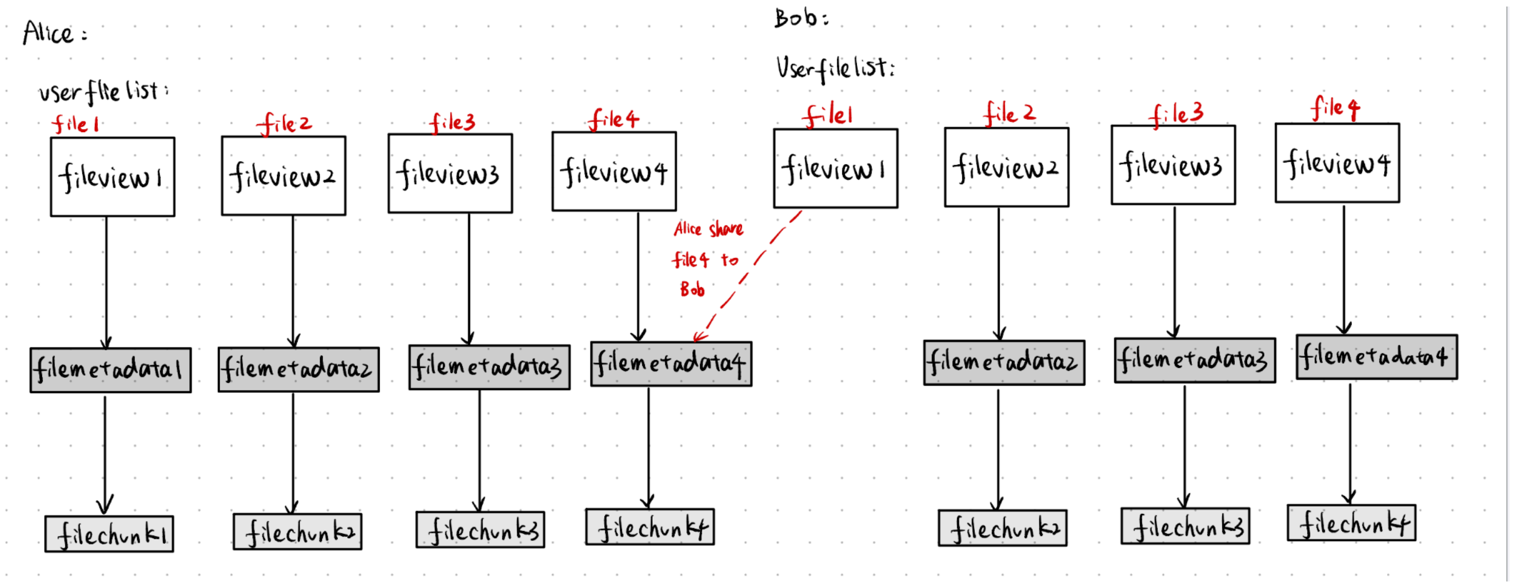
3. Recursive Revocation (Propagating to Shared Users)

If B had shared the file, D, E, F also lost access. Use BreadthFirst Search (BFS) to revoke all subusers.

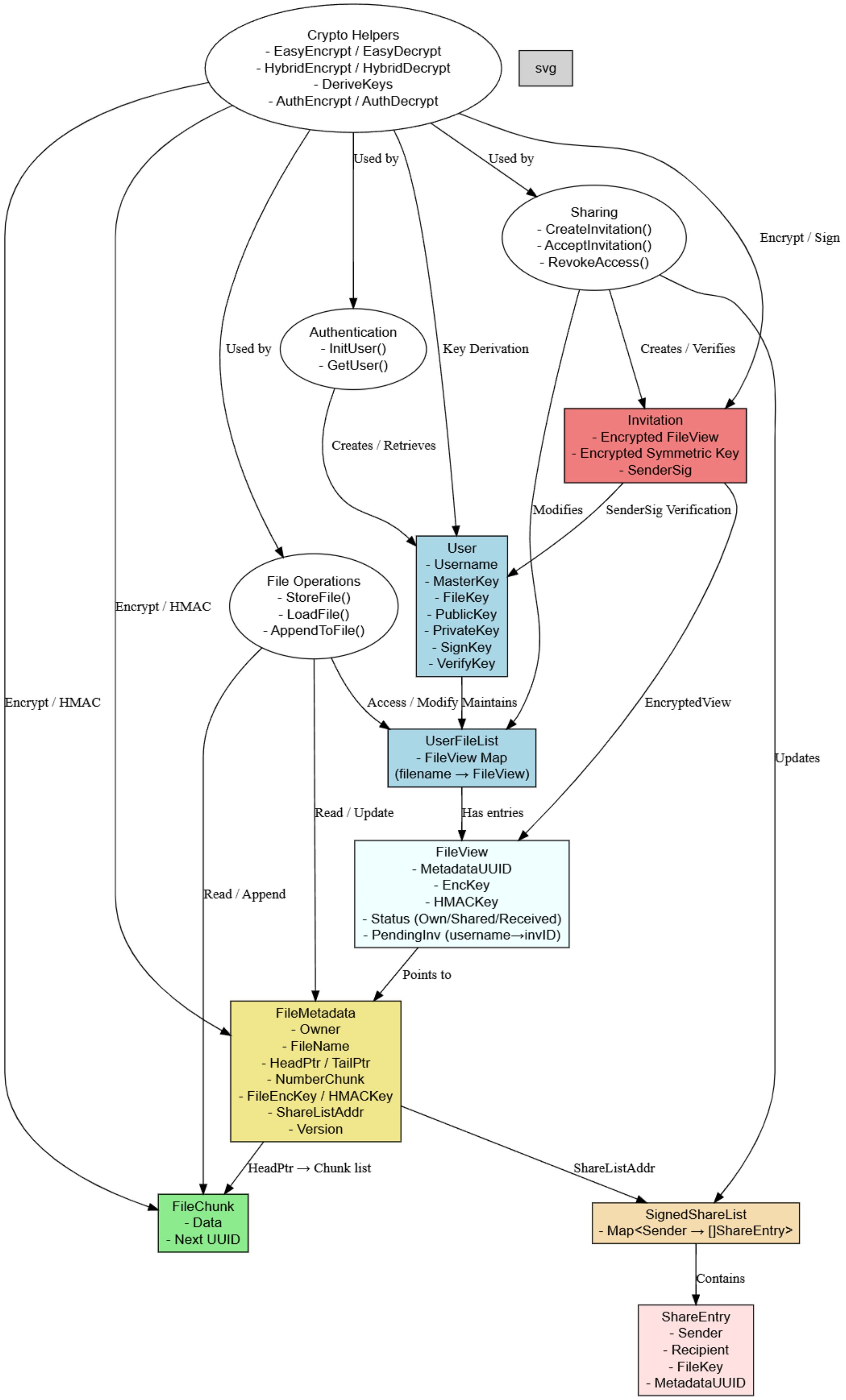
Data Structure:


User
Username\MasterKey\FileKey\PublicKey\PrivateKey\SignKey\VerifyKey
has
creates
FileList
FileViewList
Invitation
EncView\EncryptedKey\SenderSig
contains
contains encrypted
FileView
MetadataUUID\EncKey\HMACKey\Status\PendingInv
points to
FileMetadata
Owner\FileName\HeadPtr\TailPtr\NumberChunk\FileEncKey\HMACKey\lShareListAddr\Version
HeadPtr\TailPtr
ShareListAddr
FileChunk
Data\Next
SignedShareList
has list of
ShareEntry
Sender\Recipient\FileKey\MetadataUUID

Shared File Graph:



Graph of the System:



ıear line 62

Webgraphviz

```
digraph SecureFileSharingSystem {
  rankdir=TB;
  node [shape=box, style=filled, fontname="Helvetica"];

  // User Data
  User [label="User\n- Username\n- MasterKey\n- FileKey\n- PublicKey\n- PrivateKey\n- SignKey\n- VerifyKey",
fillcolor=lightblue];

  UserFileList [label="UserFileList\n- FileView Map\n(filename → FileView)", fillcolor=lightblue];

  FileView [label="FileView\n- MetadataUUID\n- EncKey\n- HMACKey\n- Status (Own/Shared/Received)\n- PendingInv
(username→invID)", fillcolor=azure];

  // File Metadata & Chunking
  FileMetadata [label="FileMetadata\n- Owner\n- FileName\n- HeadPtr / TailPtr\n- NumberChunk\n- FileEncKey /
HMACKey\n- ShareListAddr\n- Version", fillcolor=khaki];

  FileChunk [label="FileChunk\n- Data\n- Next UUID", fillcolor=lightgreen];

  // File Sharing Structures
  Invitation [label="Invitation\n- Encrypted FileView\n- Encrypted Symmetric Key\n- SenderSig", fillcolor=lightcoral];

  SignedShareList [label="SignedShareList\n- Map<Sender → []ShareEntry>", fillcolor=wheat];

  ShareEntry [label="ShareEntry\n- Sender\n- Recipient\n- FileKey\n- MetadataUUID", fillcolor=mistyrose];

  // Security & Crypto Modules
  CryptoHelpers [label="Crypto Helpers\n- EasyEncrypt / EasyDecrypt\n- HybridEncrypt / HybridDecrypt\n- DeriveKeys\n-
AuthEncrypt / AuthDecrypt", shape=ellipse, fillcolor=white];

  AuthSystem [label="Authentication\n- InitUser()\n- GetUser()", shape=ellipse, fillcolor=white];

  FileOps [label="File Operations\n- StoreFile()\n- LoadFile()\n- AppendToFile()", shape=ellipse, fillcolor=white];

  SharingOps [label="Sharing\n- CreateInvitation()\n- AcceptInvitation()\n- RevokeAccess()", shape=ellipse, fillcolor=white];

  // Arrows (Relationships)
  User -> UserFileList [label="Maintains"];
  UserFileList -> FileView [label="Has entries"];
  FileView -> FileMetadata [label="Points to"];
  FileMetadata -> FileChunk [label="HeadPtr → Chunk list"];
  FileMetadata -> SignedShareList [label="ShareListAddr"];
  SignedShareList -> ShareEntry [label="Contains"];
  Invitation -> FileView [label="EncryptedView"];
  Invitation -> User [label="SenderSig Verification"];

  // Functional connections
  FileOps -> UserFileList [label="Access / Modify"];
  FileOps -> FileMetadata [label="Read / Update"];
  FileOps -> FileChunk [label="Read / Append"];

  SharingOps -> Invitation [label="Creates / Verifies"];
  SharingOps -> SignedShareList [label="Updates"];
  SharingOps -> UserFileList [label="Modifies"];

  AuthSystem -> User [label="Creates / Retrieves"];
```

```
    CryptoHelpers -> User [label="Key Derivation"];
    CryptoHelpers -> FileChunk [label="Encrypt / HMAC"];
    CryptoHelpers -> FileMetadata [label="Encrypt / HMAC"];
    CryptoHelpers -> Invitation [label="Encrypt / Sign"];

    // Security links
    CryptoHelpers -> AuthSystem [label="Used by"];
    CryptoHelpers -> SharingOps [label="Used by"];
    CryptoHelpers -> FileOps [label="Used by"];
}
```