🟧 *: changes to previous Design*

System Design

Data Structures

1. User Struct: stores basic information of a user
    a. Attributes
        i. `String username, password`
        ii. `uuid UUID`
        iii. `Key SignK, Key PrivateK, Key UEncK, Key HMACKey`
        iv. `Map (String filename -> uuid UUIDtemp)`: Maps filenames to location of pertinent information of files that the user owns and has been given access to. The `UUIDtemp` denotes the location containing all the information (`UUIDcf`, `CFEncK`, `CFHMACKey`) needed for file decryption.
    b. Generate `UEncK, HMACKey` using `Argon2Key(password, username, 32)`, which is deterministic and reproducible.
        i. `UEncK` is a symmetric key that serves as a parameter of `SymEnc()` to encrypt the user struct and user data to store on Datastore.
    c. Generate `UUID` using `uuid.New(HMAC(HMACKey, username))`; this could be regenerated every time the user logins.
    d. Use `DSKeyGen()` to generate private and public key pairs `SignK` and `VerifyK`.
        i. User signs messages with `SignK`; receiver verifies authenticity of sender with `VerifyK`.
    e. Use `PKEKeyGen()` to generate encryption key pair `PublicK, PrivateK`.
        i. Other users could encrypt files with the user's `PublicK`; the user decrypts other people's files using his/her `PrivateK`.
    f. Update user's `PrivateK` and `VerifyK`. Store `PublicK` and `VerifyK` in KeyStore under the keys "`username||_enck`" and "`username||_vfyk`" respectively.
    g. Encrypt the user using `UEncK`, HMAC the result with `HMACKey`. Save the encryption and the MAC in `Secret` struct, save the resulting struct into the Datastore with `UUID` as the key.

2. File Struct: contains information of each file (chunk). Attributes:
    a. `uuid UUIDF, uuid source`
    b. `Key FEncK, Key FHMACKey`: encrypt and MAC the File object, save in Datastore
    c. `[]bytes data`: stores content of the File

3. CompFile Struct: represents the entire file containing all associated File objects. Attributes:
    a. `uuid UUIDcf`
    b. `Key CFEncK, Key CFHMACKey`: encrypt and MAC the CompFile object, save in Datastore
    c. `List UUIDF` (tracks of file chunks within each CompFile), `List FEncK, List FHMACKey`
    d. `BackTree BTree:` tracks the sharing history of its corresponding file. In particular, if A sends a file to B, who then shares with C, we want to make sure that if A revokes B's access, then neither B nor C would be able to access the file.

FAQs

1. How is a file stored on the server?
    - Create a new File object. Load contents from the file.

    - Generate its `UUIDF` using `uuid.New()`.
    - Call `Argon2Key(password, UUIDF, 32)` to generate `FEncK`, and `FHMACKey`.
    - Add `UUIDF`, `FEncK`, `FEncK` to corresponding CompFile's *`Lists`*.
    - File object's `source` attribute = `UUIDcf` of the corresponding CompFile object.
- Create a CompFile object.
    - `uuid.New()` to generate `UUIDcf`.
    - Call `Argon2Key(password, UUIDcf, 32)`, split the result into two parts to make `CFEncK`, and `CFHMACKey` for the CompFile object.
- Create `UUIDTemp` to store `UUIDcf`, `CFEncK` and `CFHMACKey`. Add to *`Map`* `(`*`String`*` filename -> `*`uuid`*` UUIDtemp)`.
- Encrypt the File object with `FEncK`, HMAC the encryption with `FHMACKey`. Save the encryption and the MAC in `Secret` struct, and save the resulting struct into the Datastore with `UUIDF` as key. To ensure confidentiality and authenticity, we encrypt then HMAC so that nothing is inferred about the plaintext.
- Encrypt the CompFile object in a similar manner as the encryption of the File object: use `CFEncK` to encrypt the File object, HMAC the result with `CFHMACKey`. Save the encryption and the MAC in `Secret` struct, and save the resulting struct into the Datastore with `UUIDcf` as key.
- To maintain consistency of all files that a user owns and has access to, we generate a `UUIDTemp` = `uuid.New()` for the user to store `CFHMACKey` and `UUIDcf` at `UUIDTemp` in Datastore. Place this entry in the user's *`Map`* `(`*`String`*` filename -> `*`uuid`*` UUIDtemp)`.

2. How does a file get shared with another user?
    - Let A be the sender and B be the receiver. Check if A has the file.
    - A generates a `UUIDTempB` = `uuid.New()` for B, which is the designated location for B to store the `CFEncK` and `CFHMACKey` of the shared file.
    - A updates the *`BTree`* by adding the `UUIDTempB` as A's child.
    - Generate `access_token`: encrypt CompFile's `UUIDTempB` with B.`PublicK`. Sign `UUIDTempB` with A.`SignK`. Save both as attributes in a `SecretKeys` struct and set the `access_token` as the json version of the `SecretKeys` struct.

3. What is the process of revoking a user's access to a file?
    - Suppose A wants to revoke B's access to a file. Check the existence of the file in hashmaps of A and B.
    - For the CompFile that we are revoking, create a new set of `UUIDcf, CFEncKey` and `CFHMACKey`. Delete the original CompFile in Datastore by calling `DatastoreDelete`.
    - Delete the node B from the CompFile's *`BTree`*.
    - Generate a new set of `UUIDcf, CFEncKey and CFHMACKey.` Traverse from the root of the CompFile's *`BTree`* and update the content stored at nodes of users who still have access (DFS). Users whose access has been revoked would not have access info updated.

4. How does your design support efficient file append?
    - The CompFile struct helps us to append files efficiently. We retrieve and fetch the CompFile from Datastore, decrypt it and verify its integrity using `CFHMACKey`. We then find the CompFile object and add the new File object to its list of associated Files.

- We encrypt the File object and re-encrypt the CompFile object, save both in Datastore.
- Because we don't load all File objects within the CompFile object, we efficiently append the new file and fulfill the bandwidth requirement.

<u>Section 2: System Analysis</u>

1. **MiTM Attack**
    a. A Man-in-the-Middle attack could take place when Alice attempts to share a file with Bob by sending the encryption of the file's uuid and decryption key. Eve, the middle attacker, can tamper with the package by eavesdropping or changing/deleting its contents. Bob is able to successfully decrypt the message using his Secret Key, which Eve does not have access to. Encrypting the file with a symmetric key ensures confidentiality, and Bob verifies the authenticity and integrity using the sign function with Alice's Public Key. If Eve tampers with the data, the encryption and HMAC would be different and Bob would be able to detect this attack.

2. **Gaining access after Revocation**
    a. Suppose Alice shares a file with Bob and Bob proceeds to share Alice's file with Charles; we then revoke Bob's access, expecting Charles to lose access to the file as well. However, an incorrect implementation could result in Bob losing access but Charles still retaining access. Bob could simply ask Charles to regain access. This would be a security breach so our design must prevent this.
    b. In order to prevent this unauthorized file access, we must maintain the sharing history using a hierarchical structure that tracks the original user and the "branching" of file access/sharing. We would see the "children" (in this example, Bob) and "grandchildren" (Charles) of the sharing history. Once we revoke Bob's access, we cut off his "node" such that any descendant of Bob would also be removed in the tree. This ensures the requirement that revoking Bob's access would revoke the access of users who obtained file access from Bob, and "descendents" of Bob could not re-share this file with Bob or distribute to other users.

3. **Eavesdropping When retrieving the file**
    a. Suppose Alice is receiving a CompFile by retrieving it from Datastore. Eve attempts to intercept the ciphertext. Nonetheless, even with the full ciphertext, Eve gains no partial information about the original message, and if she modifies the message sent to Alice, Alice would detect the change.
        i. For Eve to decrypt the ciphertext, Eve must use the CompFile's Encryption Key (`CFEncK`) to decrypt it. Assuming our user struct is secure and implemented correctly, there is no way for Eve to gain any partial information about the original message since we are using Symmetric Key encryption.
        ii. If Eve attempts to modify the message and forward the tampered message to Alice, Alice would notice after checking the ciphertext integrity by comparing her HMAC against the HMAC appended in the ciphertext.