1. System Design

1.1. How is a file stored on the server?

For each file, each user has a user file info struct stored, encrypted, authenticated with deterministically generated symmetric keys and uuids using HKBDF to salt the filename with a PBKDF generated from the user's username and password. This user file info struct contains the fileUUID and symmetric keys to verify and decrypt the file, along with the fileMetadataUUID and symmetric keys to verify and decrypt the file metadata. The user file info struct also contains information about the relationship of the user to the file such as whether they are the owner or not. The file metaData contains a simplified tree of the people the file is shared with along with the total number of appends to the file and the uuid of the very last append.

1.2. How does a file get shared with another user?

For each file there is an unencrypted and unauthenticated map, known as the permissionMap, in the Datastore that maps the usernames of each person the file is shared with to the relevant userFileInfo struct. This userFileInfo struct is encrypted using the person shared with's PKE public key and authenticated with the sharer's private key. Thus, despite the table itself being unprotected, adversarial additions to the table will not affect our implementation and adversarial tampering with existing values will be detected.

So to share a file, the sharer simply generates a userFileInfo struct for the shared user, encrypts and authenticates it, and maps the shared username to the new secured userFileStruct in the permissionMap. For the invitationPtr, the sharer simply sends the uuid of the permissionMap in the datastore. Upon accepting the invitation, the shared user simply verifies and decrypts the userFileStruct, and then, encrypts, authenticates and stores it in the datastore using the aforementioned deterministically generated symmetric keys and uuids using HKBDF to salt the filename with a PBKDF generated from the user's username and password.

1.3. What is the process of revoking a user's access to a file?

1.3.1. Delete revoked user and all children from the sharedTree in the file metadata.
1.3.2. Generate new keys and uuids for the file and file metadata.
1.3.3. Download file contents with all appends and file metadata and delete file and file metadata from the Datastore.
1.3.4. Update metadata appendsListSize to 0 and last append uuid as the new file uuid.
1.3.5. Store secured file and file metadata in Datastore with updated uuid's and keys.
1.3.6. Iterating through updated sharedTree, create, encrypt, and authenticate userFileInfo structs for each user and store in the permissionMap mapping to relevant username.
1.3.7. Each time a shared user performs an operation related to the file, the function checks if the file at that uuid is deleted, and if so, updates the user file info accordingly.

1.4. How does your design support efficient file append?

Let us denote the password based key derived from inputting a given user's username and password into PBKDF, which using HBKDF is also used to secure the userStruct, as the userKey. I then salt the filename with the userKey using HBKDF and then generate a uuidFrombytes using the result to obtain a deterministic UUID for the user file info struct. Next, I salt "filekeys" + filename with the userKey using HBKDF in order to obtain deterministic, yet secure and entropic, keys with which to secure the user file info struct. Now, the userFileStruct, which has a constant size, is fully accessible without having to download information relative to the number of total files, the size of the files, or the number of appends.

Upon downloading the userFileStruct, I use the userFileInfo struct to download the file metaData, which also has a constant size. I then increment the number of appends in the metadata by 1. Now, to complete the append itself, I generate a new uuid, secure the append using the file's symmetric keys, and store the previous appendUUID as the prev attribute in the storedAppendStruct such that to load a file I work backwards from the last append going through and DatastoreGet'ing the prev attribute of each append and constructing the file backwards by iterating backwards through a crude linked list. As such, in completing the append, only the metaData and the append itself are uploaded back to the Datastore.

2. Security Analysis

2.1. Revoked User Attempts to Gain Information Regarding New File Appends and Overwrites

After user A shares a file with user B and later revokes access, user B may try to use userlib.DatastoreGet on the file contents uuid and file metadata uuid to gain information regarding new file appends and overwrites. User B can obtain these uuids's by, prior to revocation, employing a series of writes to the file while calling DatastoreGetMap before and after these writes.To prevent a revoked user from gaining such knowledge, our design changes the location and encryption/authentication keys of the file and file metadata such that the user will not be able to gain any new information by looking at the old uuid's or use the old keys if the new location is somehow found.

2.2. Datastore Adversary Attempts to Discover Information Regarding Filename Lengths

Suppose there is a table, saved either in the user struct or directly in the Datastore, that maps filenames to file information, or userFileInfo struct UUID's in our case. By saving the metadata of the Datastore across several calls from a client.User to storefile, the Datastore adversary could take note of the value in the Datastore that consistently grows larger as files are added throughout the Datastore. Thus, the Datastore adversary could deduce that this key,value pair is used to store filenames. Now, by looking back at their records and also, continuing to observe this key,value pair, the adversary could gain knowledge of filename lengths. Our design prevents this from happening by never storing filenames in any manner, not even hashing, in the Datastore. Instead, we use HBKDF to salt the filename with a PBKDF derived from the username plus password to store our userFileInfo structs, which contain randomly generated uuids and keys for the file contents and metadata. Thus, we are able to generate deterministic, yet secure and entropic, keys for our userFileInfo struct without ever storing the fileName.

2.3. Revoked User Attempts to Gain New File Access Information Directly from the permissionMap using Datastore API

After user A shares a file with user B and later revokes access, user B may try to, since they still know the permissionMapUUID from the invitationPtr, use userlib.DatastoreGet to access the updated permissionMap and procure the new file contents uuid and encryption/authentication keys, thus, regaining access to the file if successful.To prevent a revoked user from regaining access this way, our design, when performing revocation, first deletes the revoked user and all its children shared users from the shared tree in the file metadata. The revocation function then iterates only through those users left, updating, encrypting, and authenticating only their values in the permissionMap. If user B accesses their value in the permissionMap, it will contain the old file uuid and file keys. Any other value is inaccessible to user B as it is encrypted using each user's own PKE public key.