Section 1: System Design

1. **How is each client initialized?**
   Generate a key using Argon2Key function with password and username as input with 128 bits, ensuring the uniqueness of the key, let this be key1 (used to generate other keys). Generate key2 using HMACEval with key1 and (password || username) as arguments. Use key2 to generate UUID for the user-struct. Store the public encryption key in the Keystore with username as the passed in key argument. Store the DSVerifyKey in the Keystore by hashing the username with the marshaled public encryption key as the key argument. Generate key3 using Argon2Key(password, key2, 16). Use key3 to encrypt the marshaled private keys DSSignKey and PKEDecKey. Additionally, we create maps from UUID, created from hashing filename with key1, to 2D byte slices. The first, File_to_data, contains the UUID (of the file, which is directly stored on the Datastore), hashkey, and symmetric encryption key. The second is another map that maps a file to the users that the file has been shared with. Lastly, we generate a hash value of the user struct using the three maps and the private keys. This hash values is stored in the user struct to check for integrity. (We also store 3 keys on the user struct for convenience, but we make sure they're not uploaded to the Datastore). Store these values within the user struct.
2. **How is a file stored on the server?**
   We create a struct for files, and for every file, we create an instance of the file struct. The data of the struct is the encrypted file data and hash code, ensuring we have a way to guarantee integrity. We generate a UUID with the HMAC of key1 and filename as the arguments. This UUID is used to store information about the file within the user struct. A second UUID is generated randomly to store the file struct in Datastore.
3. **How are file names on the server determined?**
   The file-structs are stored using random UUID which are saved in the associated user-struct. We generate a hash value using HMACEval(key1, filename) and store it within the user struct. This allows different users to share file names while avoiding a collision in Datastore.
4. **What is the process of sharing a file with a user?**
   We create a hash value using HMACEval(key1, username (of recipient) || filename) to generate the UUID of a share-struct, let this be key'. We then encrypt the information essential to the file (random UUID, hash-key, encryption-key) using the receiver's public encryption key. The encrypted data is stored on the shared struct. Then, generate a digital signature using the sender's DSSignKey and store it in the share struct to verify the sender. Store the share-struct in Datastore using key'. Generate the magic-string by concatenating the encrypted value of key' and its digital signature (using the sender's DSSignKey).
5. **What is the process of revoking a user's access to a file?**
   Store the file at a new UUID, and re-encrypt it using a new key and generate its hash value with a new hash key. Iterate through the shared list associated with the file and

update the new file info in all the shared users' shared-struct, except for the revoked user. In the process, the revoked user will lose track of the file's location and will lose the ability to read and edit it.

6. **How was each of these components tested?**
   For InitUser, we tested various kinds of edge cases with usernames and passwords, as well as users with the same password. We basically did the same for doing the storage and appending of the files, testing edge cases like an empty file with an empty name or nonexisting file. For share-file, we tested for security threats, where the sender attempts to pose as a different user. In addition, we tested the case, where an intended receiver attempts to decode the magic-string. For revoke-file, we tested the ability of the owner of the file to edit and view the file after revoking. We also tested the un-revoked users' access to the file. Moreover, we tested that any users that gained access through the revoked user would also be revoked. We tested the integrity checking of various components by corrupting the data stored in Datastore.

Section 2: Security Analysis

1. An attacker using chosen plain-text attacks for filenames is unable to locate the file since file UUIDs are generated randomly. Therefore, an attacker is unable to corrupt a file of their choosing.
2. Since magic-strings contain the digital signature of the sender, a malicious user is unable to share any data under a false name.
3. When sharing files, only the file-struct of the shared file is exposed to other users. Thus, users with access to the file are unable to determine any other information about the user. This prevents any attempts to corrupt the user-struct itself. Least privilege.
4. When a user's permission to a file is revoked, the file struct is stored at a new UUID, and the file is encrypted and hashed once again. This ensures that the information (file location, encryption, hash, etc.) that a revoked user has is no longer relevant to the current, updated file. Prevents the revoked user from corrupting the file directly by locating it.
5. Our Shared struct, which contains the UUID, encryption key, and hash key that would allow the shared users to locate the shared file and decrypt it, also contains a signature. This ensures that the shared users can verify that the information in the shared struct is indeed from the right person, preventing a malicious agent47 from potentially redirecting the shared users to a different location.