Section 1: System Design:

When a client creates their account for the first time, we generate a new User struct that stores their chosen username and password. We randomly generate public encryption keys and digital signature keys, storing the private/signing keys in the User struct and the public/verify keys in the Keystore. We obtain a Base Key that is 16 bytes long that we use for generating other symmetric encryption keys by using the password hashing function Argon2Key on the chosen user's chosen password with the username as the salt. The User struct is encrypted and stored in the datastore at a UUID that is a hash of the Base Key with a random salt. Each user has has two maps. One maps a file name to a UUID that points to the datastore corresponding to the file (explained further down below) The other maps a file name to the file's symmetric encryption key.

Files are stored on the server by first breaking the file into chunks of 64 bytes each. Using this scheme allows for more efficient appending of files, as when data needs to be appended we only need to load the last chunk of a given file. We encrypt each of these chunks separately with symmetric encryption and generate tags using HMAC for each chunk to check for integrity. The keys for these symmetric encryptions and HMACs are generated from the Base Key using HKDF. We also generate a random UUID for each chunk to store into the Datastore server. Then we create a list of these UUIDs, encrypt it with a key generated from the Base Key, and store this list in the Datastore. The map data structure maps the file name to the UUID of this list of UUIDs and the symmetric key used to encrypt the file.

Therefore file names are not stored directly on the server, but each user will have a personal encrypted User struct, which has a map data structure, stored on the server. The map data structure maps a filename to the UUID to the list of UUIDs where the encrypted file chunks are stored.

To share a file with a user, the owner first obtains the UUID to the list of UUIDs storing the file and the file's symmetric encryption key from his/her map, then encrypts these two values with the receiver's public key and signs this encrypted message with his/her personal signing key, creating a confidential message. The receiver will then receive this message and check it's authenticity and integrity by comparing the digital signature to the message using the sender's verification key (which is public). Once the receiver verifies the message, they can decrypt it with their own private key. Now that the receiver has the unencrypted UUID of the list of UUIDs of the file and the file's encryption key, they can access/modify/share the same file as the owner, with all changes shared since they access the same exact file.

The owner can revoke all access to a file by obtaining the UUID of the file's list of UUIDs from their map, and storing that list in a new randomly generated UUID. This new UUID will be saved as the new value to the filename key in the map, and the old UUID will be used to delete the old copy of the list of UUIDs from the Datastore. Therefore other users will try to access the old UUID when they access a revoked file, and will get a message saying that this key does not exist in the dataStore. The file can also be decrypted and re-encrypted with a new symmetric key for further security.

In addition to the tests that had been provided, we tested two functionalities that were not checked: AppendFile and RevokeFile. The former was tested by uploading a file, then

appending more data to the said file. If no errors were thrown, the file that had been appended to, was then downloaded and compared to an expected string value, that included both the initial and added data. If the two byte strings matched AppendFile had successfully worked. RevokeFile was also fairly simple to test. When a user had the rights to a shared file, they would be able to download said file. All that needed to be tested then was if RevokeFile was called on a file, would a call to LoadFile via a user who didn't have owner privileges trigger an error.

Section 2: Security Analysis:

Three potential attacks our design protects against are chosen plaintext attacks, man-in-the-middle attacks and known plaintext attacks.

An attacker could attempt a known plaintext attack on the share/revoke file functionality of this system. An attacker normally does not have access to another user's data, but if a file was shared with them they would receive access to both a file and its symmetric encryption key. However, this encryption key would not be able to reveal any data about the user since it was generated through a collision-proof, one-way hashing function. The attacker would know both the plaintext of the message and the encrypted version, but since a different encryption key is used for each file the attacker would be unable to decrypt other files even if he had access their encrypted versions. In addition, each encryption key was generated randomly and so does not reveal any information about the owner of the file. Also, if the owner revoked access to the file, any link to the owner is broken since the file itself will have moved to a random location the attacker has no idea of.

A man-in-the-middle attack would try to take advantage of the insecure channel used to transmit the magic string for sharing files. Although the MITM attacker would be able to intercept messages sent between the receiver and sender, the messages are encrypted before signing so no information could be obtained from the ciphertext or signature. The MITM could not pretend to be the receiver, since the receiver's public key is stored on a secure server (the Keystore). The digital signature in the magic string will detect any changes in the file, preserving integrity. If the MITM attacker attempted to pretend to be the sender, the signature would not match since only the sender has access to their private signing key, and the receiver knows the public verifying key which is stored on the secure server Keystore, preserving authenticity.

Unfortunately by using an untrusted server such as the Datastore, a poorly secured system can be susceptible to a chosen plaintext chosen ciphertext attack. An attacker could potentially glean useful information if they had to access to the Datastore and were able to upload a ciphertext version of some plaintext they already have. Our system defends against this by utilizing an authenticated symmetric encryption scheme. The important part of our scheme is how we encrypt every file with its own unique key. This means if an attacker had the opportunity to upload two files with certain similarities between them, they would be unable to notice patterns in the encrypted files. This entropy is doubly enforced by using a different IV for encrypting every chunk of a file using symmetric encryption, protecting against files or repeating or blank values. Note that the symmetric encryption scheme we use already uses CTR mode, so that each chunk within a chunk is also encrypted differently. With our chosen forms of encryption, an attacker would be unable to learn anything about how to decrypt a file even with access to both the encrypted and decrypted versions.