We implemented our Part 1 based on the Design 1 of the sample designs given. Our approach is to maintain a directory of IDs, keys, dictionary of clients and the asymmetric keys used to open the IDs and symmetric keys that will locate the file in the server and unlock the data.

During upload, we set up our IVs, r (which is the ID of the file on the server), and symmetric keys using get_random_bytes of length 16. Then we decrypt our <username>/dir_keys directory to access our encryption key (key_e) and authentication key (key_a). We use these keys to decrypt our directory. If our directory does not exist yet, then we create one using an IV and symmetric keys. We then extract the info we want to access such as r, the symmetric keys, dictionary of clients we shared this file with, and the original creator of the file corresponding to the name of the file. If no such file exists yet in our directory, then we symmetrically encrypt it with ke and ka.

Next, we check if the file has a different owner than the client currently accessing it. If we are the owner, then we can go ahead and encrypt the files and store them under <username>/files/r. If we are not the owner, then we know that we have gotten this file from a different client and will need to store the ID of that file location, the owner of the file, and the keys used to decrypt <owner>/keys/ID to access the r, k1, and k2 that will unlock <owner>/files/r and the data inside it.

Within download, we yet again decrypt our <username>/dir_keys, get the directory, decrypt it, and extract the same information as before. Next, we again check if we are the owner of this file. If we are, then we can go ahead and decrypt <username>/files/r because we have the keys to do so. If we are not the owner, then we access owner/keys/ID, verify the MAC, and decrypt the info to find out r, k1, and k2 that will unlock owner/files/r.

When sharing, we yet again access our key_e and key_a, decrypt the directory, and extract the same information as before. If we are the owner of this file, then we create encryption and authentication keys that will be used to unlock owner/keys/ID. Then we encrypt our data with the shared keys and add the new client we are sharing with as the dictionary key and the ID to find owner/keys/ID and the encryption and authentication keys to open it in our dictionary of shared clients within our directory and send this to the shared client. If we are not the owner of the file, then all already have all this information within our directory, so we add the new client we are sharing the file with to our dictionary of shared clients within our directory, encrypt it, and send it to the new shared client along with r, the encryption and authentication keys, and owner. The shared client should then be able to see the owner of the file that is being shared and the same keys and can decrypt owner/keys/ID to download the contents of the file. This is what receive_share does.

If we revoke, then we go through the same initial steps as before, then we delete owner/keys/ID from the server. Next step we make new k1 and k2 keys to encrypt owner/files/r with, go through our dictionary of shared clients, delete the one we are revoking, and go through the rest of the shared clients and update their k1, k2 with the new k1 and k2 keys. So, say Alejandro shared a

file with Beto and Diego and Beto shared the file with Chuy. And say Ale revokes Beto. Since Chuy received the same encryption and authentication keys as Beto, when he unlocks owner/keys/ID with them, the k1 and k2 values there will be outdated and can't open owner/files/r anymore, and even so, we delete owner/keys/ID so he can't even access it. But Diego will have the new updated keys and can still access the file.

One attack could be say Ale revoked Beto, and Beto wants revenge so he had saved the r, k1 and k2 keys so that even if Ale deletes owner/keys/ID, he still knows what k1 and k2 is to unlock owner/files/r. But we account for this scenario by making new k1 and k2 keys and re-encrypting owner/files/r and sending the new keys to everyone we previously shared with except Beto. So, the k1 and k2 keys Beto has are useless.

In another attack say Beto saved the encryption and authentication keys used for owner/keys/ID and tries to open another owner/keys/ID2 for the same or even different file. Since each of the encryption and authentication keys were unique for each file and each shared client, Beto cannot use these keys to open any of the other r,k1,and k2's encrypted within owner/keys.

In another attack, let's say Ale shared a file with Beto and Chuy. In this scenario, Beto is malicious and wishes to feed Chuy malicious data. Ale realizes Beto is malicious, but before he can do anything, Beto revokes Ale from his own file. Beto then updates the file, giving Chuy bogus information. Our design would protect against such an attack since the owner has direct access to the keys which opens his or her own file, while a shared client must access these keys through a set of additional keys which both clients share. If Beto were to try to revoke Ale, it would result in him trying to change key values which Ale never even accesses when trying to open his own file.

In yet another attack assume Ale shares a file with both Beto and Diego. Beto then shares this same file with Chuy. Diego however wishes to deny Chuy access to this file at any means necessary. Diego realizes that every time a person is revoked from Ale's file, Ale changes the file's encryption and authentication keys for each person with whom he has shared his file. Diego believes that by getting himself revoked, he will deny Chuy access, since Ale did not directly share his file with Chuy. Our design protects against this type of attack since Chuy would still have access to the encryption and authentication keys which give him access to the encryption and authentication keys corresponding to the shared file. Ale never changes these extra sets of keys, meaning Diego has failed in his attack.

Lastly, let's say we have the same case of Ale sharing a file with Beto and Diego. Beto shares his file Chuy. In this scenario, Beto and Chuy are malicious collaborators. Ale realizes this and revokes Beto. However, Beto believes he can still gain access to the file through Chuy. Our design accounts for this, since Ale deletes owner/keys/ID which Beto had access to. This is the same place on our server Chuy would look to for opening the shared file. Since it has been deleted, both Chuy and Beto have been denied access.