***DESIGN:***

In our design, we use the following keys for security: client's public key $K_P$ , client's private key $K^{-1}{}_P$ , symmetric encryption key $K_e$ , MAC key $K_a$ , and name confidentiality key $K_n$ .

For each user we have store a directory $info$, containing the user's metainformation. The directory is encrypted with AES-CBC using symmetric key $K_i$ , which is encrypted with the client's public key $K_P$ . $Enc_{K_p}(K_i)$, $Enc_{K_i}(info)$ is stored under at **information/<username>**. An asymmetric signature $sign_{K^{-1}{}_p}(Enc_{Kp}(K_i), Enc_{ki}(info))$ is stored under **information/<username>/signature**. The symmetric encryption ensures the confidentiality of the metainformation, and the signature ensure the integrity.

In the directory $info$ , the client stores information specific to each file. A file owned by the client stores secret information at **info["files_I_own"][<filename>]**, which contains 2 symmetric keys $K_e$, $K_a$ and 2 strings $R$, $username$ . A file shared to the user stores information at **info["files_shared_to_me"][<filename>]**, which contains 3 strings $link$ , $from\_username$ and $from\_origin$ .

Every file is encrypted with AES-CBC using a unique $K_e$ , and MACed with a unique $K_a$. $Enc_{k_e}(file)$ is stored under **<username>/R** , where $R = H_{SHA-256}(filename \parallel K_n)$ and $username$ is the owner's username. File's confidentiality is ensured by symmetric encryption, while integrity is ensured with MAC: we calculate $MAC_{k_a}(C, R)$ where $C = Enc_{k_e}(file)$. The MAC tag is stored at **<username>/metadata/R.** Uploading and downloading a file owned by the client itself is intuitive, you will using the scheme mentioned above to encrypt and decrypt, validating the integrity from the MAC path.

When the client $c_1$ shares a file, the client will pass the secret information specific to the current file to the other client $c_2$ , including $K_e$ , $K_a$ , $R$ and $from\_user$ . Once $c_2$ knows $K_e$ , $K_a$ , $R$ and $from\_user$ , he will be able to access the shared file, details of which will be explained later. However, instead of sharing the secret information directly over $msg$ generated after calling the $share()$ function, we will store them on the server to achieve revoking. **<c2_username>/shared/<link>** is where we store the secret information, security of which is ensured with signature and symmetric encryption scheme similar to how we store directory information. Then we send **<link>,** which is a random string generated whenever a sharing happens, and the file's original owner's username to the $c_2$ . Such information is not sensitive and thus we only use signature to ensure integrity. Inside of $c_2's\ receive\_share()$ function, after validating integrity, $c_2$ will store in his directory's under the section **info["files_shared_to_me"][<filename>]** with $link$ , $from\_username$ (the user who shares to you), $from\_origin$ (the file's original owner).

If the $c_1$ is the owner of the file, the secret information $K_e$, $K_a$, $R$ and $from\_user$ can be found in his own directory $info$. If the $c_1$ is not the owner of the file, i.e. some other user has shared the file to him, then he would first find the link that points to the secret information by reading **info["files_shared_to_me"][<filename>**], then he download and encrypts the linked secret information. In addition, $c_1$ , if the original owner of the file, would keep track of the root children that he shared the file to, by adding ($c_2$, $c_2's\ link$) to his directory under the section **info["files_I_own"][<filename>]["users"].** If not the original owner, $c_1$ will call update a list under the path **<c1_username>/shared/<c1_link>/children** to keep track of the children that he has shared to. (<c1_link> is what c1 used to get the secret information.)

Now the design enables the $download()$ for both original file owner and shared users. If the original file owner, the client get the $K_e$, $K_a$, $R$ and $username$ from his own directory under **info["files_I_own"][<filename>],** download cipher text from **<**$username$**>/<**$R$**>** to validate decrypt. If a shared user, the client get $K_e$, $K_a$, $R$ and $from\_user$ from the link (similar to what described in sharing) and then get cipher text by download **<**$from\_user$**>/<**$R$**>.**

If $c_1$ wants to revoke $c_2$, he first check if the filename is under **info["files_I_own"]**. If yes, $c_1$ generates new $K_e, K_{a,} R$ for this file, and re-upload the file with new keys to new $R$. Then he removes $c_2$ from the root children stored at **info["files_I_own"][<filename>]["users"].** Now for every valid root children and their descendants, we would update their linked data with new $K_e, K_{a,} R$. So that when these valid shared users goes to their link to get secret information, it will be the the new and correct one. But $c_2$ and its descendants will not be able to read or update the new file, because the new file and updates are stored at new $R$, which is never propagated to $c_2$. The finding $c_1$'s root children's descendants is achieved through a search using information stored at **<username>/shared/<link>/children.**

Finally, for uploading optimization, on the server side we store some additional meta-information for each file stored at **<username>/R**, including **size** and **changelog**, under **<username>/R/size and <username>/R/changelog.** The meta-information is encrypted and MACed. On the local client side, we store a cache copy of each file, which we achieve during $download()$ in **self.cache**.

If it is the first time uploading a file, the file size on server differs from the file being uploaded or we are uploading due to revoking, we encrypt, MAC and upload the entire file, with an empty $changelog$. If the file sizes matches, then we check if there is a local cache of the file. If not, we download that file. Then, we get the $changelog$ from **<username>/R/changelog.** We update the local cache with the $changelog$, to make sure the local version is update to date with server. Then we generate a new changelog, storing the difference between the file being uploaded and the local cache calculated with $compute\_edits()$ function. We extend the downloaded server changelog with this new log. And we only encrypt and upload the extended changelog, not the file itself.

When we download the file, we download the ciphertext as usual. But in addition, we download the changelog, and apply the changes the plaintext decrypted.

***SECURITY:***

Attack 1:
Server being very smart, he knows that on revocation we would search the descendants of the valid root children of a shared file, and update them new valid shared secret information to **<valid_descendant>/shared/<link>.** To find out who are the descendant, the server knows that we look at the list from **<valid_descendant>/shared/<link>/children,** which is unencrypted because we don't mind who is sharing to whom a secret. He will try to put a malicious user in the list, or simply replace list of bad boys of his choice. Then on revocation, these bad boys will know the updated secret!

Yucks. But unfortunately for them, we defend against that by signing the **children** list with the **valid_descendant'**s RSA private signature. And every time we get the **children** list, we will verify that signature, raise IntegrityError otherwise. So anyone other than valid_descendant cannot modify the children list.

Attack 2:
Server replace the data under **<username>/shared/link**, tricking a client to get fake $K_e$, $K_{a,}$ $R$ and thus reading a fake file and making updates. The client might upload important information to the file believing it is trusted.

We used RSA signature to verify the integrity of the linked data from **<username>/shared/link**. If anything changed, the data will not verify and raise an IntegrityError. Only the direct parent or the origin owner's signature would verify.

Attack 3:
Server knows that we store important information/directory at **information**/**<username>,** and server wants to see it or modify it, so he would know all the secret shared key, filename mappings.

However, we use battle test security measures to ensure both confidentiality and integrity. We sign everything with RSA signature for integrity. And $Enc_{K_p}(K_i)$, $Enc_{K_i}(info)$ , so the symmetric key $K_i$ is only known to the client to decrypt info.

Attack 4:

Server read the data under **<username>/shared/<link>**, thus stealing $K_e, K_{a,} R$ and have full access as if a shared user.

Data under **<username>/shared/<link>** is encrypted as following, similar to Attack 3: $Enc_{K_p}(K_i), Enc_{K_i}(linked\_data)$. Only the shared user can get $K_i$, and decrypt $linked\_data$

<u>Attack 5:</u>
Man-In-the-Middle attack on the $msg$ generated by $share()$ function: A MITM attacker can modify $msg$ such that *Bob* will visit a fake link, get fake $K_e, K_{a,} R$, and will be able to achieve bad consequences like in Attack 4.

We use RSA signature to sign msg, though $msg$ is not encrypted, the MITM attacker cannot modify it or else it would not verify.