CS 161 Project 2-3: Efficient Updates Writeup

Jonathan Sun

cs161-acz

1 Simple Upload/Download

For this portion, the client creates two symmetric keys of 16 bytes each using `get_random_bytes()` and places them into a tuple and encrypts it with the public key from the public key server before signing the cipher text using its private key. The encrypted keys, signature, and a randomly generated 16-bit ID to identify the file are then stored into the storage server under `<username>/dir_keys/file.txt` as $(ID, E_{pubK}(symmK_1, symmK_2), S_{priK}(E_{pubK}(symmK_1, symmK_2)))$. The ID is also stored in `<username>/files/ID` and the file data encrypted using authenticated encryption and hMACed as the value.
To download file.txt, the user would need to get the keys from `<username>/dir_keys/` and verify the signature before decrypting the cipher text. Then they would need to access `<username>/files/` for the data, check the hMAC, and then decrypt the cipher text.

2 Sharing

For this portion, my plan is to add people that will get a file shared to them as a collaborator. For example, if Alice wants to share `file.txt` to Bob then Bob needs to be added as a collaborator for the file. By using the ID of `file.txt` from upload, Alice will access the server at `Alice/collaborators/ID` and will initialize a dictionary to store the collaborators in here if there is nothing in the directory already. The dictionary will be formatted to look like {'$user_1$' : ['$user_2$']} where $user_1$ will be the person sharing a file and $user_2$ being the person receiving a file. In this case, the dictionary would look like {'Alice' : ['Bob']} and if Bob further shares it to Charlie and Donny then it would look like {'Alice' : ['Bob'], 'Bob' : ['Charlie', 'Donny']}. This dictionary will be stored with a hMAC of this dictionary and will be used to see who can upload and download. Therefore, this dictionary must be secure against attacks compromising its integrity and authenticity.
When Alice is ready to share a file to Bob, then (owner, hMAC(owner) and the file ID along with the keys used to decrypt the data are sent to Bob. The keys used to decrypt the data are also encrypted with Bob's public key so he can just decrypt them himself and also signed with the sender's private key.

When Bob is ready to `receive_share`, then he adds in (ID, keys, signature) into the server's `Bob/dir_keys/file.txt` and (owner, sender, hMAC(owner)) into the server's `Bob/shared_files/ID`. Therefore, when Bob wants to get the `file.txt` and update it, he will use the owner's information in `Bob/shared_files/ID` and then decrypt the files using the keys in `Bob/dir_keys/file.txt`. Alice's dictionary of `Alice/collaborators/ID` will check if Bob can upload and download.

3 Revocation

For this portion, to revoke someone means that the owner's dictionary needs to be changed. For example, if Alice finds out that Bob is actually a spy and therefore wants to revoke Bob, then Alice will need to delete all entries of Bob in the dictionary's values. If Bob is a key in the dictionary, then a recursive deletion can remove all those that Bob shared the file with along with those down the chain. This ensures property 4 of revocation is in place. The owner also needs to update the keys for each collaborator and re-upload the file with the newly generated keys.

4 Efficient Updates

For this portion, when the user uploads a file, that file will be broken up into parts that would be stored onto the server. This idea will allow parts of a file to be modified without needing to re-upload the entire file again. The parts will be stored in the server as `<username>/parts/fileID/part#` where part is the integer that will represent which part it is. So, part will be between 0 and the (number of parts - 1). The parts are stored using authenticated encryption. Aside from adding in the parts of the file into the server, the information for the file itself will be added into the server. The information consists of (number of parts, most recent user who uploaded the file, number of updates) This information will be MAC'ed. and stored in the server as `<username>/parts/fileID/info`. Furthermore, a log of changes (list consisting of part numbers of the parts that were modified) will be stored in the server as `<username>/parts/fileID/log`.

The part sizes are of size 2048 and the client will store the part size, parts, and log in its state. The parts are kept in a hashlist dictionary which maps fileIDs to hashlists. The hashlist will have of list of tuples made up of (part, hash(part)) and when `receive_share` is called, the user calling it will also store these things into the state.

When uploading, if the fileID and number of parts of the modified file are the same as the fileID and number of parts of the old file, then the hashlist for that fileID is updated along with the modified parts in the server. The hashlist will be used to check which parts have been modified and the log will be updated to reflect which parts have been updated.