Project-2 Part-2 Question-6 **Ninh DO**
CS 161 - Computer Security SID 25949105

Design

Part 1

In order to store a file on the server, a client first needs three symmetric keys: k to encrypt the filename, k_e to encrypt the file content and k_a to authenticate the file. In my design, I use k for all files and I use one set of (k_e, k_a) for each file, i.e. for file i I use the key set (k, k_{e_i}, k_{a_i})

The key k is generated specifically for each client when the client is initialized, that is in constructor method I call *make_unique_symmetric_key()*. This key is then encrypted asymmetrically using the client's public key and stored along with its signed encryption using the client private key on the server *client/info*, i.e. we store $client/info : \left(E_{K_{pub}}(k), Sign_{K_{pri}^{-1}}(E_{K_{pub}}(k))\right)$

Whenever we want to upload a file, we generate a key set (k_e, k_a). Similarly, this key set is encrypted, signed as $\left(E_{K_{pub}}((k_e, k_a)), Sign_{K_{pri}^{-1}}(E_{K_{pub}}((k_e, k_a)))\right)$. To store this encryption on the server, we have to make a path specific for the encrypted filename. That is we first have to encrypt the filename into $E_k(filename)$ using the symmetric key k, then the path and the (k_e, k_a) encrytion on the server are $client/E_k(filename)/keys : \left(E_{K_{pub}}((k_e, k_a)), Sign_{K_{pri}^{-1}}(E_{K_{pub}}((k_e, k_a)))\right)$

After having the key set (k, k_{e_i}, k_{a_i}) for file i, we split the file content into a number of chunks with chunk size 512 bytes, then we encrypt the file content chunks using k_e and k_a, AES-HMAC with CBC mode and store the encrytion on the server $client/E_k(filename_i)/chunk\_index$: $AES_{CBC} - HMAC_{k_{e_i}, k_{a_i}}(filename_i, file\_content\_component_i)$

We also store the number of file chunks and encrypt this number on the server $client/E_k(filename_i)$: $AES_{CBC} - HMAC_{k_{e_i}, k_{a_i}}(filename_i, file\_chunk\_num)$. Of course, this number is encrypted using k_e and k_a, AES-HMAC with CBC mode.

If the file is large, we create a Merkle tree of the file in the local client. And we store this tree on the server
$client/E_k(filename_i)/MerkleTree/node\_label : AES_{CBC} - HMAC_{k_{e_i}, k_{a_i}}(filename_i, hash_value)$.
This is to support effective update. The details will be explained in part 3.

Whenever we want to download some file or get keys from the server for using, we have to verify the signature (for keys) or check the integrity (for file content) using the coresponding methods 'digital signature' or 'HMAC' before decrypting it. To download a file, we have to download the number of file chunks first, then we loop through this number to download all file content components, then we verify and concaternate them together.

Part 2

If client 1 want to share a file i with another client, say *client_2*, client 1 follows 4 steps:

1. Share the key set (k_{e_i}, k_{a_i}) with the client 2 by making the encrypted and signed key set available under the client 2 name, that is client 1 create and put onto the server:
$client\_2/E_k(original\ filename)/keys : \left(E_{K_{pub_{client_2}}}((k_e, k_a)), Sign_{K^{-1}_{pri_{client_1}}}(E_{K_{pub_{client_2}}}((k_e, k_a)))\right)$

2. Send the client 2 the key set (k_{e_i}, k_{a_i}) and the encrypted original filename $E_k(original\ filename)$ or eof via an encrypted and signed message msg containing
$\left(E_{K_{pub_{client_2}}}((k_e, k_a, eof)), Sign_{K^{-1}_{pri_{client_1}}}(E_{K_{pub_{client_2}}}((k_e, k_a, eof)))\right)$

3. Make the link to the file content $client\_2/eof$: "$[POINTER]$ $client\_1/eof$"

4. Create and MAC a list of clients the file is share with
$client\_1/E_k(original\ filename)/shared\_with : [client\ 2]$

When client 2 receives the share, he/she check the integrity and authentication of the message, then creating two links:
$client\_2/E_{k_{client\_2}}(new\ filename)/keys$: "$[POINTER]$ $client\_2/E_{k_{client\_1}}(original\ filename)/keys$"
$client\_2/E_{k_{client\_2}}(new\ filename)$: "$[POINTER]$ $client\_2/E_{k_{client\_1}}(original\ filename)$"

Client 2 also has to check the integrity and authentication and re-sign the keys using his/her private key before using the keys.

When client 1 revokes the client 2's access, he/she deletes the links created at step 1 and 3 above, pop client 2 out of the share list, generate a new set of key (k'_{e_i}, k'_{a_i}) for file i, re-encrypt and HMAC the file content and recursively go through the share lists to re-distribute the new key set to the other client. The re-distribution is done by implementing step 1 repeatedly. Since the original client has to sign the key set using his/her private key, the other clients have to check and, if necessary, re-sign this key set whenever they want to use it.

The original client can check the integrity and access the share lists recursively because all the share list are HMACed using the share key k_{e_i}. Thus, whenever the clients are still in a sharing relation wrt a file, the sharer can access the check the integrity and access sharee's share list.

The key points of this design are:

1. Whenever we put any thing on the server, we have to encrypt and sign or HMAC it, and whenever we download something from the server, we have to verify the signature or check its integrity before decrypting it. We may not encrypt the share list because the specs says the share list can be known by third party but we still have to HMAC it to make sure it is not modified.

2. The original client has access to all the shared key sets to modify then in the case of revoke. When he/she re-distributes the new key set, he/she signs it under his/her private key, so the other user must check and re-sign the new key set before using it.

3. The revoke is done by 5 steps: 1. Delete the file link; 2. Delete the key link; 3. Remove the sharee from the share list; 4. Re-encrypt the file with the new key set, and 5. Re-distribute the new key set

Part 3

We store a Merkle tree for each large file on server. The storage of the Merkle tree on server follow the same principle of storing anything on server, that is encryption and AES-HMAC using CBC, whenever we want to use any value of Merkle tree, we have to download its node and verify the value before using.

If we upload the file for the first time, we create a Merkle tree locally and store all the nodes of this tree on the server. The path and encryption of each node is indicated on Part 1, that is we use the label of each node to create its path.

If we upload the file for the second time and on, or in other word we update the file, we create a Merkle tree of this new version locally and we compare this new Merkle tree with the old Merkle tree we stored on the server according to the logarithm algorithm: compare the roots, if their hash values are equal, skip it; otherwise, we compare the children, so on down to the leaves. By this way, we can identify the index of the file chunk that we need to update in logarithmic time, then we update this file chunk and the corresponding hash values.

Performance Analysis

The update is logarithmic time and the performance is 'log size' as indicated by the tests of Part 3. This runtime is achieved because we do not upload the whole new file version but we identify the modified file chunk and upload this chunk onto the server. The algorithm to find the index of this file chunk is we use binary search on the Merkle tree by comparing the hash values of selected nodes on the new and old Merkle tree (the new one on local and the old one on server). Since the binary search is logarithmic in runtime, the update algorithm has logarithmic runtime.