Spring 2017 - CS161 - Project 2
Part 3 Design Document
Brandon Teo, 26494656, cs161-aay

System Design (Q1 - Q3)

Snapshot of Untrusted Storage Server:

$alice/dirkeys$	: $E_{alicepublickey}(k_e \| \| k_a) \| \| Signature_{aliceprivatekey}(.)$
$alice/sdirkeys$	: $E_{alicepublickey}(k'_e \| \| k'_a) \| \| Signature_{aliceprivatekey}(.)$
$alice/dir$	: $\{fn_1 : (10111, k_1, k_2); fn_2 : (11001, k_3, k_4)\}_{k_e} \| \| MAC_{k_a}(.)$
$alice/sdir$	: $\{fn_1 : \{bob : (00000, k_{se}, k_{sa})\}\}_{k'_e} \| \| MAC_{k'_a}(.)$
10111	: [D] $IV_1 \| \| E_{k_1}(10111 \| \| v_1) \| \| MAC_{k_2}(.)$
11001	: [D] $IV_2 \| \| E_{k_3}(11001 \| \| v_2) \| \| MAC_{k_4}(.)$
00000	: [P] $IV_3 \| \| E_{k_{se}}(10111, k_1, k_2) \| \| MAC_{k_{sa}}(.)$
$bob/dirkeys$	: $E_{bobpublickey}(k_e \| \| k_a) \| \| Signature_{bobprivatekey}(.)$
$bob/sdirkeys$	: $E_{bobpublickey}(k'_e \| \| k'_a) \| \| Signature_{bobprivatekey}(.)$
bob/dir	: $\{fn'_1 : (00001, k_{se}, k_{sa})\}_{k_e} \| \| MAC_{k_a}(.)$
$bob/sdir$	: $\{fn'_1 : \{\}\}_{k'_e} \| \| MAC_{k'_a}(.)$
00001	: [P] 00000

Remarks:
- The 5 digit id's are a simplification of randomly generated 16 byte id's.
- Every user has different sets of keys for their directories but to minimize confusion we use the same notation for every user in the snapshot.

Preface

The overall design of client.py for part 3 follows that of part 2 with some minor tweaks to address a couple security flaws and overall aesthetics of the client and the reader can refer to the part 2 design document and easily follow along. This improvement takes some inspiration from the sample solutions provided by the course staff.

Improvements/Tweaks

First off, notice that we have made the names of the data and pointer nodes to be shorter and more concise. This also hides information regarding whose file it is and makes it harder for attackers to interpret any information from the id's.

Secondly, we address the issue of file sharing and the tweaks we made to make it more secure. In our last implementation, when we share a file to another user, we provide the exact keys used to encrypt the file to the shared user and have him store it in his own key directory. We found this to be less ideal and came up with another way to address this. We basically created sharenodes as a location to store the encryption keys that are used to encrypt the files and provide shared users the keys to access these sharenodes. We keep a copy of these sharenode keys in our sharing directory so that we can easily change the contents of the sharenode, i.e. the file encryption keys, when we need to do so and the shared users will not have to do anything on their part. Another major perk of doing this is that when we revoke a specific user, we can easily reencrypt our file and redistribute the new encryption keys with ease.

Third, we address the issue of user revocation where in our previous implementation, we only removed the sharenode and assumed that any revoked user will not be able to trace down the file. To further secure this operation, anytime we revoke a user, we reencrypt the shared file with new keys and redistribute the keys to other shared users as well as remove the sharenodes. Another small implementation flaw in our sharing scheme is that we did not make our sharing message secure and we do so by encrypting it using asymmetric encryption with the shared user's public key and also provide a signature using our own private key.

Efficient Updates

To allow efficient updates when updating files that are previously stored on the server with files of the same size, we implemented the following scheme. When we first upload a file, we upload it "along" with a Merkle tree with two types of nodes — internal nodes and leaf nodes. These nodes are implemented as python lists with internal nodes having 4 fields and leaf nodes having 3 fields. Both types of nodes have fields for a hash value as well as a field that stores the length of the file under that subtree. However, the internal nodes have a left and a right child pointer whereas a leaf node only has one that points to the data node that it directly represents.

Once we have this structure, we can have the owner of a file have its filename point to the root node of this Merkle tree. When an owner or a shared user of a file wants to update that specific file, he/she can grab from his/her directory the id of this root node and obtain it from the storage server and at the same time construct a Merkle tree using the updated value on the local server. Then, the client will first compare the root hashes of the two trees and if they are equal, that means we don't have to do anything to it. We can then recursively perform this action since the internal nodes on both trees have the left and right child pointers and we compare the subtrees until we reach the bottom which are the leaf node layers. When we compare hashes and traverse down the tree, we also identify which nodes that need to have their hashes recomputed and do so recursively on the way back to the top layer.

By doing this, we essentially gain an "efficiency" increase in that we are able to determine which parts of the file we need to change and only change that part of the file. This allows us to send less more data over the transmission channel which can have significant effects when the file on

hand has really big size. In the most blatant case, when we upload the exact same file onto the server without this implementation. The client will just re-upload the whole file onto the server again which utilizes unnecessary resources and achieves virtually nothing. With this scheme, we only need to download the root node of the file of interest which has almost negligible size and we will be able to determine that we in fact don't need to re-upload at all.

Performance Analysis

In this update scheme, we first need to grab our file directory from the storage server to determine the root node of the file of interest. Then, after constructing the Merkle tree for the updated data, we compare the root hash with the root hash of the tree that is on the server. As explained in the section above, we only ever grab nodes from the server that we find out the hashes are not the same. In other words, for each portion of data that is required to be changed for the file to be updated, we only access the nodes from the root all the way to the specific node for that data packet. Once we have located the portions of the file that need to be changed, we only upload these relevant parts back onto the server and also update the hashes of the nodes that are involved and send them back to the server too.

In a rough sense, we can expect this scheme to require the following amounts of data transmission over the network during an update:

Size of directory = S
Size of node = X
Number of nodes involved in updates = N
Actual data that needs to be updated = Y

Amount of data transmitted = S + 2NX + Y