CS 161— Project 2— Part 3 Design Document

Fabiola Lopez, cs161-aab— Pei Jie Sim, cs161-ane

April 18, 2017

Question 5: Efficient Updates

Be sure to include (a) how you perform efficient updates, and (b) a short performance analysis.

Design Description

Question 1: Upload/Download

There were a number of minor changes to upload and download since Part 2. Previously, uploading a file only generated a random r and those r's are mappings to file contents. In Part 3, each file is broken up into blocks and r no longer maps directly to file contents. Instead, it maps to the metadata of a file and every block of a file is pointed to by $r+i$, i.e. the concatenation of r with the index of a block and stored on the server. Those changes in upload are also reflected in download, particularly the reconstruction of a file using the length as well as r. The content of a file is simply the decryption of each block joined together.

Question 2/3: Share/Revoke

As for share, receive_share and revoke, no changes were necessary to ensure it works with our implementation of Part 3.

Question 5: Efficient Updates

In implementing efficient updates, apart from the tweaks to upload and download, we had to build merkle trees to figure out which portions of a file changed. Rather than performing full updates to a file every time upload is called, having merkle trees allowed us to only update the bytes of file that's changed, thus increasing efficiency.

The metadata of a file contains two pieces of information. The first is the length of file and the second is the leave hashes of the current state. With the leave hashes, we were able to build the corresponding merkle tree. We also built a merkle tree for the new value of a file. We then recursively traverse both merkle trees and replace the blocks that are different from the new blocks until all of the blocks match the new ones. The implementation of the merkle tree is similar to that from section 9 where we referenced sibling hashes, uncle hashes as well as the root hash to

propagate any change to a hash up to the root.

Efficient updates only work for files of the same length. If the length of new file is not the same as that of the old file, we replaced the file entirely.

Performance Analysis

During each upload, we obtain the directory of a client and verify its directory listing to make sure it has not been tampered with. This means the number of bytes being transferred over the network for this stage is proportional to the size and number of files in that directory.

The number of bytes of data being sent to the server regardless of whether it's a full or partial update is determined by the size of the metadata of the file plus the size of update being performed. The metadata of every file consists of a tuple of the length and the leave hashes. The size of length is 8 bytes, while the size of the hashes is

$$filesize/blocksize * size\ of\ each\ hash$$

After serializing the tuple, the metadata is encrypted and prepended with a 16-byte IV and its tag is then computed. The tuple of (encrypted meta, tag) is then serialized again and sent through the network to the server. Hence, the total size is

$$size\ of\ encrypted\ meta + 16\ bytes\ for\ IV + size\ of\ MAC$$

As for the file content, the total number of bytes transferred is the *required size of update* that is upper bounded by the size of the entire file split up into blocks. Each block is encrypted (including the 16-byte IV), tagged and the tuple serialized and sent over to the server. Accounting for all the blocks, the total size is

$$number\ of\ blocks * (size\ of\ encrypted\ block + 16\ bytes\ for\ IV + size\ of\ MAC)$$