CS161 – Spring 2017 — Project 2 Design Doc

Eoin Morgan, SID 24356939, `cs161-adm`
Aditya Iyengar, SID 24816838, `cs161-ajh`

Basic Part III Design

Parts 1 & 2 Design Our design for parts 1 and 2 of the project are almost identical to the staff solutions [1] with a few small differences. Instead of having a separate 'share' directory, we store the information about which users the client shares a file with inside of the regular directory. This makes the directory larger in size, but simplifies the implementation in terms of keeping directories synchronized with the server. The files are also placed in randomized locations (which are recorded in the user's directory), rather than in a location determined by encrypting the file name.

Efficient File Updates

Each file node will have two metadata fields: the total file *length* in bytes and the *tophash* of the merkle tree of the chunks. The merkle tree has its blocks stored at $key = HASH(E(block)_{K_e}))_{SHA256}$, where K_e is the same symmetric key used to encrypt the file on the server. Each node of the merkle tree is stored at $key = HASH(key_{left} + key_{right})$, and contains the values $key_{left}, key_{right}, data$. Additionally, the root node contains a value $mac = MAC(key_{root})_{K_m}$, where K_m is the same symmetric key used to compute the MAC of the file on the server.

Downloading After following the gateway files to the file node, the client traverses the merkle tree to the leaf nodes (indicated by a key value in the *data* field). The client then downloads those blocks linked to by the *data* field. If the server root node's MAC does not verify, if any node is malformed, if any node's children are missing, or if any blocks are missing, we know that the tree structure has been tampered with. If the root node of merkle tree computed on the downloaded blocks does not match the root stored on the server, we know that the tree structure hash been tampered with. After validation, we should cache the file data and root hash (to avoid future downloads) as well as the block IVs (to avoid having to download before uploading).

[1] `https://d1b10bmlvqabco.cloudfront.net/attach/ixur6mzlrc465l/ixurdqdq2y5352/j1cf0s8e3xe9/design_part2.pdf`

Uploading First the client checks to see if the update preserves file length. If not, we know that we must overwrite the entire old merkle tree, and can select new encryption IVs. If the client has the IVs used to encrypt a file's blocks in its cache, then the client can immediately compute the merkle tree locally. Otherwise, the client must first download the file to get those IVs. After fetching the file node metadata, the client can start traversing from the root of the server's merkle tree, comparing to the local merkle tree. If the node on the server is at the same key, we can prune that branch of the tree. Otherwise, we upload a new node at that location and recurse to the next node. After passing through the entire merkle tree, any modified blocks will be uploaded and their paths to the root of the tree will be updated. It is important to reuse the same IVs if the file length is not changed, to ensure that other clients can utilize their cached IVs. If the total file length changes, then this is irrelevant.

Security Analysis

Security Attack #1: MITM When Sharing Access One attack our design would prevent involves a man-in-the-middle attacker modifying the share message after the sender sends the message and before the other person receives it. Such an attack could potentially allow the attacker to have the receiver save the wrong file name, thus invalidating their shared access to the file. However, in our design, we first use a symmetric key to encrypt the filename and then asymmetrically encrypt this symmetric key using the public key of the recipient. We then create an asymmetric signature that involves the encrypted symmetric key, an IV, and the encrypted message and then concatenate this to our encrypted filename to produce our message. Thus, if an MITM attacker were to intercept our message and make any modifications, the receiver would be unable to verify the message signature with their private key and thus would know that it had been tampered with.

Security Attack #2: Accessing A File After Revocation Another attack our design would prevent would involve revokee trying to directly access the latest version of a file they previously had shared access to. Suppose Alice gave Bob shared access and he then wrote down the location of the file along with the keys that were used to encrypt both the message and the MAC; when Alice revokes his access to the file, he could potentially use the location and keys to try to directly access the file on the server. However, our design defends against this by randomly generating new keys and a new location for the file when Alice revokes access from Bob. While these new keys and location are shared with all other individuals who currently have shared access to the file (through a newly-uploaded updated gateway file), Bob will not have access to these new keys and location, and thus will be unable to obtain the newest updates to the file; he will just end up raising an IntegrityError.

Security Attack #3: Decrypting Another User's Shared Messages/Gateway Nodes Because our client sends (potentially many) messages to other users indicating the presence of gateway nodes, an attacker can know that at least a certain number of gateway nodes exist on the file server, pointing to actual shared files. One potential attack would

be to decrypt gateway nodes not intended for access by a malicious user. However, this is impossible because each gateway node is encrypted with new private symmetric and MAC keys, known only to the original owner and the shared user. It would be intractable to learn these keys from the encrypted share message (even if the attacker could view many of such messages), because each message is itself encrypted with another key for confidentiality. This message key is encrypted asymmetrically using the recipient's public key, which will not leak any information. Thus, the recipient's public key acts as a trust anchor that allows us to set up a secure gateway file that an attacker cannot identify or read (because it looks like any other encrypted file on the server).