Part 3 Design Document

Simple Upload/Download (Question 1)

The integrity of the files in our system is secured by a randomly generated 16-bit key that is created when a new file is uploaded. That key is stored in the user's directory of secure keys (which is encrypted with the user's symmetric key A, and is authenticated/ensured via a Message Authentication Code predicated on their symmetric key B), to insure both the integrity and the confidentiality of the symmetric keys. In order to ensure the integrity of the *symmetric* keys, we encrypt and sign them via asymmetric encryption (using `self.private_key.publickey()` for the encryption and `self.private_key` for the decryption.

We provide confidentiality and integrity to the directory by using two symmetric keys A and B. A and B are stored under `user/dir_keys`, encrypted using the Client public key and signed with the Client private key. The encryption is `AES-CBC` with random iv per access, and the signature is an `HMAC (SHA-256)` on the ciphertext. These symmetric keys are easier to rotate out than a user's public-private asymmetric keys, helping to mitigate the effects of a private key getting compromised. Every new file generates two random 16 byte symmetric keys k_1 and k_2. All of the keys that we are using in this chain of encryption are thus guaranteed to be secure, and are unreadable by anyone except the intended party (confidentiality). The keys are also tamper evident due to the inclusion of a signed Message Authentication Code accompanying all of the stored keys (integrity).

We also intentionally obfuscate the location at which a file is stored. Though this location is visible to any *authorized* party, the fact that the directory extension is generated via 32 cryptographically random bytes makes it impossible for an attacker to simply guess where a file would actually reside. When we discuss revocation later on, this implementation (and the fact that we change the file location after revoking) will prevent a malicious user whose privileges have been revoked from trying to upload junk bytes to that file, or even seeing when the file contents are changed in the future (if the file was not relocated after a revocation, then it would be possible for an attacker to keep checking the values and see if the values were periodically changing).

Sharing (Question 2)

We elected to design a system of access points, which exist at locations on the server distinct from the actual file location and data which they access. The access points contain the true location of the file (i.e. the file extension that contains the file contents/data), as well as the symmetric keys k_1 and k_2 by which the file is encrypted. A user can read the access pointers via user level pointers. The user level pointers contain information about where an access point is located, and the 16 byte symmetric keys used to decrypt/verify the data. The user pointers are encrypted with the public key of the receiver and signed with the private key of the sharer, so that they are only accessible by the user with whom they're shared, but are still verifiable.

As a result of this design pattern, there is a difference between how the original owner of a file performs a share, and how other users do. For every share invoked by the original file uploader, a new access point is created, encrypted and signed with its own set of randomly generate 16-byte keys (A_{k1} and A_{k2}, as we used for our original file values). In addition, the location of the access pointer is generated via 32 random bytes. The location of the access point, as well as the keys A_{k1} and A_{k2}, are encrypted and signed asymmetrically, so that only the receiver will be able to use the access point. The final element of share is for the receiver to store it in their own directory (note that this store will be encrypted and signed via a `MAC` using the receiving user's symmetric keys A and B, just as the initial upload of a file was).

Sharing for a user who was not the original uploader is comparatively much simpler. All they must do is to take the location and key values for the relevant access pointer (which they have in the user-specific pointer to the access point they were given), and then encrypt those values asymmetrically via the public key of the user with whom they want to share the file. This will result in the user they are sharing with having all the necessary info to access and decrypt the data stored in the access point. It also means that we can revoke a user's and all of their childrens' access privileges by just affecting access points (without having to worry about the potentially quite extensive chain of shares that might exist).

Revocation (Question 3)

Much of the machinery that we established in the sharing part above was designed to make the revocation process as secure as possible. The main steps for revocation are to first generate a new set of random encryption keys for the file ($k1$ and $k2$). Then a new file location is generated (random 32 bytes are chosen for the extension). The file contents are re-encrypted using the new keys, and are saved at the file's new location. We iterate through the list of all the access points and only update the location and key values for those users whose rights we aren't revoking. Finally, we delete both the access point that we left out, and delete the contents from the old file location.

Thus, when the user (and any other "child shares") try to go through that access point to gain access to the file contents, they will not have access to any of the appropriate metadata (location or encryption keys), and thus their attempts to access the file (or even see if the file has changed) will fail.

Efficient Updates (Question 5)

Uploading and Downloading Large Files

When the file is uploaded, we generate the Merkle Tree by separating the file into 1024 byte blocks, and then hashing (via SHA1) and combining those blocks. These blocks are encrypted with k_1 and then encrypted with SHA 256 and signed with k_2. The resulting blocks and signatures are then uploaded to the storage server at random locations, along with pointers to their children (the hashes which are combined to form them), and a pointer to the root node of the Merkle tree is then kept (encrypted), so we can retrieve the tree values during a subsequent upload, if needed.

The actual file blocks themselves (which were generated previously, when we were dividing up the input data in order to create our Merkle tree), use a slightly more complex encyption scheme. When we are uploading a file, we use our file key, k_1 to seed a Pseudo Random Number Generator, which works by iteratively hashing the value that we generated on the previous pass. This PRNG then spits out new key values that we use to encrypt the data blocks as they are updated. We also hash the values with SHA 256 and sign them with the file key value k_2. These blocks are then uploaded at a randomly generated location, prefixed by their order (so the first block of the user's data will be considered `$randomLocation/0`, and then the next one will be `$randomLocation/1`, and so on). When decrypting these blocks we start by seeding the PRNG using the same file key as before, k_1. Then, we iterate through all the uploaded blocks via their prefix value, and use thet value that is generated by the PRNG to decrypt (the PRNG is guaranteed to spit out the same values as it did when we were encrypting the file initially, so long as we use the same seed value). We then reassemble/concatenate the values back together, and return that final value to the user.

Efficient Updates

In order to efficiently update files (when a user reuploads a file with different contents but the same file length), we first regenerate a Merkle tree (`regeneratedMerkle`) based on the file that the just uploaded (though we *don't* upload that new Merkle tree to the storage server). We then retrieve the pointer to the root node of our *stored* Merkle tree, and download that value. We check that retrieved value against the root of `regeneratedMerkle`. If it matches, then we know that the uploaded value is the same as the *currently stored* value, which means that we don't actually need to change anything and we can just exit out of the upload function. However, if the values of the stored and regenerated root nodes do not match, then something has changed and we must recursively search through the Merkle tree to identify which nodes are different. To accomplish this, we continually check the child nodes (starting from the root). If the node in the `regeneratedMerkle` matches that in the stored Merkle tree, then we do not have to explore this subtree any further (as all of the files must match below that point). However, if the two hash values *do not* match, then we must continue to call down into that subtree, until we eventually locate the root nodes which changed. Once we have identified those blocks, we then compute their values by running the PRNG from seed (k_1) and encrypting, then uploading the new blocks. We also identify which nodes in the stored Merkle tree need to be changed, and encrypt the new values, sign, and then upload them to the appropriate locations.

We also modified the manner in which we shared files slightly to accomodate for the new way in which they were stored. We still retained the concept of having an "access point" for our files, which provides a level of abstraction between the location of the file and the links that users follow to access the file's contents, which (as before) gives us a simpler way to obfuscate and protect a file from a user after their permissions have been revoked. However, we now have a more complicated structure for storing our data. The access point, instead of providing a direct link to a file, now links to a directory which contains the size of the file (so that the download function knows how many blocks to retrieve), the actual data files themselves, which are ordered and can be iterated over by the `download` or `upload` function, and a pointer to the root node of the stored Merkle tree. When we grant a user access, we still create a new access point which contains a pointer to the file directory, in addition to the file encryption key. Then, when we revoke the file, we still generate a random key as before, and move all the data files into a new directory, but we also regenerate the entire Merkle tree, which completely changes all the locations and which the nodes are stored, to try to prevent a malicious user (without permissions) from tampering with the Merkle nodes and potentially causing unexpected results.

Finally, to reduce the number of bytes that need to be transferred, we keep a local copy of the Merkle tree. If you were the last person to edit the file, then the upload function will use your local copy/cached Merkle tree (to the extent possible) instead of requerying the server to obtain the tree nodes again. However, if we ever lose state, we reupload the entire Merkle tree (based on the data that was passed in).

Performance Analysis

When changing the file's contents while keeping the file length the same, we only pull down exactly as many nodes from our stored Merkle tree as needed to verify which of the data nodes have changed. For example, if our file was 4 KB ($\Rightarrow$ 4 blocks), then the Merkle tree would have 7 nodes (4 at the bottom layer, 2 at the second, and a single root node). If we were to traverse the tree to find the changed node, then we would need to explore a total of 5 Merkle tree nodes, each with a size of 265 bytes $\rightarrow$ 1280 bytes total. Then, once we had identified which node needed to be changed, we would only need to push/overwrite one block (1024 bytes) on the storage server in order to update the file. Thus, as our file size grows exponentially large, we are able to obtain $\log n$ performance.