Secure File Store Design Document

Gan Tu

May 13, 2018

1 System Design

In this section, I'll summarize the design of my system for the secure file store, with sharing and revocation functionality as well as efficient updates. I'll explain the major design choices made, including how data is stored on the server.

1.1 Symmetric Encryption

For each client, we generate three random 16-bit master symmetric keys: k_e, k_m, and k_n during client initialization using `get_random_bytes()`, where k_e is used for symmetric encryption, k_m is used for message authentication codes, and k_n is used for name confidentiality.

We store these keys in a key directory of the client, under the ID `<username>/key_dir`. The symmetric keys are stored using the scheme below, with the concatenation of three keys asymmetrically encrypted with ElGamal public key K_R^{-1} and asymmetrically signed using RSA private key K_G:

$$C = E_{K_G}(k_e \parallel k_m \parallel k_n)$$
$$\text{Data Stored} = \text{Sign}_{K_R^{-1}}(C) \parallel C$$

Whenever we need these keys, we can simply retrieve the above content at `<username>/key_dir`, check the integrity of keys by verifying the signature with RSA public key, and decrypt asymmetrically using ElGamal decryption key to get back the three master symmetric keys.

We encrypt all data with `AES` in `CBC` mode, with a new random 16-bit `IV` generated using `get_random_bytes()` at each time. `IV` is then prepended to the ciphertext. We use `SHA256-HMAC` as our `MAC`. We compute the `MAC` over the concatenation of IV-prepended ciphertext and the ID, under which this piece of encrypted data will be stored, in order to detect internal swapping attack by malicious server. We always verify the MAC before decrypting the messages. Specifically, the encryption scheme is described below:

$$C = \text{AES-CBC}_{k_e}(\text{content})$$
$$\text{Data Stored} = IV \parallel C \parallel \text{SHA256-HMAC}_{k_m}(IV \parallel C \parallel \text{id})$$

1.2 Data Storage, Upload, and Download

Inspired by the original insecure client implementation, the values stored in the storage server under *any* ID are either a data node, stored as `[DATA]<value>`, or a pointer node, stored as `[POINTER]<value>`.

To *upload* and store a file initially on a storage server, we create a data node `[DATA]<value>` under a *randomly* generated 16-bit ID `data/`r_1, where `<value>` are the file contents encrypted using two randomly *new* keys k_e' and k_m' according to the symmetric encryption scheme described above. For efficient upload and update of large files, please refer to next section below.

In order to be able to later retrieve, verify, and decrypt the file contents, we need to store these two new keys, k'_e and k'_m. We do so by storing them in a pointer node `[POINTER]`$k'_e \parallel k'_m \parallel$ `data/`r_1, all encrypted using the master k_e key, MACed using the master k_m. This pointer node is then stored at ID `<username>/keys/`r_2 where $r_2 =$ `SHA256(<filename>` $\parallel k_n$) with the master symmetric key k_n to ensure name confidentiality.

Consequently, if we want to *download* the file `filename`, we first compute

$$kid = \texttt{<username>/keys/SHA256(<filename>} \parallel k_n)$$

Then, we obtain the the encryption keys k'_e, k'_m, and a node ID `data/`r_1 by verifying and decrypting the content stored at pointer node of ID `kid` using master k_e, k_m keys.

Then, we verify and decrypt the node at ID `data/`r_1 with k'_e and k'_m. If the node at ID `data/`r_1 is a data node, we return the file contents stored at this data node. If the node at ID `data/`r_1 is another pointer node, which will be the case when `filename` is a shared file to the user, we follow the pointer node up until we reach a data node, encrypting and verifying intermediate pointer nodes using intermediate keys k''_e and k''_m. If we encounter any node that's not either a valid data node or a valid pointer node, we raise `IntegrityError`. We also raise `IntegrityError` if any intermediate node failed the relevant the `MAC` check, which will occur if the data has been tampered with or the user has been revoked access to the file. Specific details about sharing and revocation are described in the sections later.

1.3 Efficient Updates

In order to efficiently update large files, we employ a tree-based method, inspired by Merkle tree. We modify the procedures for `upload` and `download`.

When a file is initially uploaded, we divide the file into chunks of `1024` bytes. We also create a Merkle tree from these file chucks, where each node consists of pointers to its left and right sub-trees, a cryptographic hash of the left and right sub-trees's hash values, and the total length of the file content below this tree node. These information are stored as dictionaries and stored as `json` strings. All of these, both tree nodes and file chucks, are encrypted and authenticated using the encryption method described in the encryption section above. The encryption keys used to encrypt and authenticate file chucks are also reused to encrypt and authenticate the tree nodes themselves in order to implement efficient updates. The tree nodes and file chucks are all stored at random IDs `data/`r_i, generated by a random number generator.

To `update` a file, we first check if the file length of the new file content has increased from the old content. If so, we simply reupload the entire file. Otherwise, we compute a merkle tree of the newly updated file. Then, we check to see if we have a local copy of the merkle tree that is consistent with the merkle tree stored on the server, by checking the root hash of the two trees. If so, we compare the two trees locally to find all the updates needed to perform on the tree nodes and file chucks, and update only those blocks on the server. If not, we compare the new merkle tree with the merkle tree on the server instead.

To compare and update two trees, we check the root hash of both trees. If they are the same, we have no more work to do. If they are different, we need to update the hash of the old tree to the hash of the new tree. We update the length field if the file length covered under the new tree has decreased compared to the old tree. We next fetch the left and right child, and then recursively efficiently compare and update them as well, until we reach the data nodes for the file chucks themselves, in which case we will just simply update the file contents themselves.

We always store a local copy of the recent merkle tree whenever a file is uploaded, updated, or downloaded for caching and performance purposes. We also update the local copy if the local copy is out of sync from the tree on the server accordingly.

To `download` the file, we walk the tree in post-order, decrypting and verifying the blocks along the way and we return the file contents if the file hash is valid. We raise `IntegrityError` whenever we encounter

a node where its hash is not the hash of the child subtree's hashes, or the file hash is inconsistent with the hash stored at the leaf tree node above the file chucks.

1.4 Sharing

Whenever a user `a` shares a file `<filename>` with a user `b`, we create a *new* pointer node `[POINTER]`$k'_e \parallel k'_m \parallel$ `data/`r_1 under a *randomly* generated 16-bit ID `data/`r_3, where k'_e and k'_m are keys used to encrypt and MAC the file `<filename>` and `data/`r_1 is the ID of the data node storing the encrypted file contents. Note the content of this pointer node is exactly the same as the pointer node of the file owner client in the previous section. The only difference is that instead of encrypting using the master k_e, k_m keys of the file owner, we create two *new* encryption keys k''_e, k''_m keys to encrypt this pointer node. We will pass these two new k''_e and k''_m keys to the user, whom we share the file with, via a message `m`.

To construct a message, we encrypt $k''_e \parallel k''_m \parallel$ `data/`r_3, which is the concatenation of the keys used to encrypt the new pointer node and the ID of that pointer node, who in term points to the actual data node. We append the `username` of the user whom we will share the file with to the ciphertext, and asymmetrically encrypt it using the other user's ElGamal public key and sign using the file owner's RSA private key. We pass this message to the other user.

To receive the file, the user verifies the integrity of the message `m` using file owner's RSA public key and decrypt using user's own ElGamal private key. The user then creates a separate pointer node $k''_e \parallel k''_m \parallel$ `data/`r_3 encrypted using the other user's own master k_e, k_m keys and store it under ID `<username>/keys/`r_4 where $r_4 =$ `SHA256(<new_filename>` $\parallel k_n$) with the master symmetric key k_n to ensure name confidentiality.

In addition to all above, we also keep a `shares` directory for each client to store information about the users and the shared files, which we will later use to implement the revocation procedures. Specifically, the `shares` directory is a dictionary, stored under ID `<username>/shares` of the client, mapping any `filename` to the list of `users` that the client has shared the `filename` with. Each user of the `users` is also a dictionary, storing the ID of the shared data node `did`, the encryption key k'_e used to encrypt the values at `did`, and the key k'_m used to `MAC` the values at `did`. Specifically, the structure is like this:

```
shares = {
    filename: {
        "username": dict(did=..., ke=..., km=...),
        ...
    },
    ...
}
```

Whenever user `a` share a file with user `b`, the file owner `a` adds user `b`'s `username` to the list of shared users for `filename` under `shares` directory. Then, we store the ID (`did`) of the new pointer node created at the time of sharing and the two new k''_e, k''_m keys (used to encrypt that new pointer node) under the `username` field of the `filename` in the `shares` directory of the file owner `a`.

1.5 Revocation

Whenever user `a` revokes user `b`'s access of a file `filename`, we re-encrypt the data node, at which the contents of file `filename` is stored, with two new random 16-bit encryption keys k_{e2} and k_{m2}. We remove user `b` from our `shares` directory for file `filename`. Then, we go through all the non-revoked users of `filename` in our `shares` directory, and replace the contents of their pointer nodes at ID `did` (created at the time of creation) from `[POINTER]`$k_e \parallel k_m \parallel$ `data/`r_1 to `[POINTER]`$k_{e2} \parallel k_{m2} \parallel$ `data/`r_1 so the non-revoked users will have the updated, new encryption keys for the file. Note, under this scheme, the revoked user will no longer have the new encryption keys for the file stored at `data/`r_1. Thus the revoked user is unable to decrypt the contents of the file `filename` stored at `did` anymore. We delete the pointer node saved at ID `did` for safety.

2 Security Analysis

In this section, I will present three concrete attacks that I have come up with, and explain how my design defends against each attack.

2.1 Regain Access After Revocation of Another User

Attack: Because we have a `shares` directory for each client, an incorrect implementation of the revocation process may forget to remove the revoked client from the `shares` directory and update new encryption keys for all non-revoked users without checking if the user has already been revoked access. If this vulnerability exists, one potential attack will be that, after being revoked access to a file, the man-in-the-middle will get access to the file again due to the unchecked redistribution of new encryption keys phase, if the man-in-the-middle manages to let the file owner revoke the file access to another client (for example, another client username controlled by the man-in-the-middle).

Defense: One defense is to add a boolean variable in the `shares` directory, so the file owner won't distribute the new encryption keys after a revocation process to any user that has already been revoked access before. This defense is memory inefficient as it keeps information of revoked users, so naturally a second, improved defense will be simply removing the revoked user from the `shares` directory.

2.2 Shared Access Abuse

Attack: If shared files are stored with IDs of a structured or standardized naming convention, a man-in-the-middle who have hijacked the ID of any shared file (for example, by controlling a user account that the sender has shared any files with) may be able to guess the IDs of other files and gain access to them by following the pattern. If the encryption keys used to encrypt shared files are symmetric keys of the file owner and they are also shared with the receiver, the receiver is able to decrypt other files of the sender. If the encryption keys for shared files are same for all the shared files between sender and the same receiver, once the encryption keys for one shared file is compromised, all other shared files are also compromised.

Defense: We generate a *random* ID for each file stored so the man-in-the-middle cannot guess IDs of other files easily. We also generate a new set of encryption keys for each file share and pointer node, so encryption keys are never repeated across shared files and thus compromise of one set of encryption keys still keep other files safe. By creating a new set of encryption keys for each pointer node of shared files, the master symmetric keys of file sender will also not be leaked.

2.3 Message Hijacking and File Swapping

Attack: The man-in-the-middle could hijack a message during the process of sharing. It may try to read or modify the message. The man-in-the-middle can fool another user to read an incorrect file, or get access to the shared file secretly by calling `receive_share` on the hijacked message itself, if the message itself can be faked or decrypted by the man-in-the-middle. Even if the message itself is encrypted, the man-in-the-middle can swap the messages sent to two different users to so the users will get incorrect files, or the man-in-the-middle can swap two different messages sent to the same user but from different file owners to mess things up.

Defense: We encrypt the message using receiver's ElGamal public key and sign it using sender's RSA private key. The receiver always verify the message before decrypting it. Because the public key server is trusted, the man-in-the-middle won't be able to modify the message without failing the RSA verification because it doesn't have the RSA private key of the sender. The man-in-the-middle won't be able to receive the file neither because it doesn't have the ElGamal private key of the intended receiver to decrypt. Note that the to man-in-the-middle cannot swap messages sent to the receiver from different senders neither, because a sender's RSA public key won't be able to used to verify a message sent by

another users To prevent the man-in-the-middle from swapping messages sent to different receivers, we append the intended receiver's username to the ciphertext and compute the `MAC` on the concatenation in the message construction phase. When receiving a file, the receiver always checks if the received MAC matches the MAC computed on the received ciphertext concatenated with receiver's username. Note we don't encrypt the receiver's username inside the ciphertext, because we want to prevent the man-in-the-middle from learning any information from observing our decryption failures.

3 Performance Analysis

The encryption scheme is $\Theta(N)$, where N is the size of the value to be encrypted.

The efficient update scheme by walking the tree on the server is on average $\Theta(2NS + 1024X) = \Theta(NS + X)$, where N is the number of tree nodes involved in the efficient update, S is the size of each node, X is the number of the `1024`-byte file chucks needed for update. Note it's $2NS$ because once we decrypt to retrive the node, once we update that node. If we have a consistent local copy of the merkle tree with the server, it is still roughly $\Theta(NS + X)$ as we can drop the constant term.

To construct a merkle tree for a file content of size N, we will create about 2N + N = 3N blocks. If each block is of size S, then it takes $\Theta(3SN) = \Theta(SN)$ time to construct such a merkle tree, including encryption.

The sharing scheme is $\Theta(1)$, because to construct a message for sharing, we always create a constant-sized pointer node with new constant-sized encryption keys and constant-sized file data node's ID. The encryption of a constant sized node is also constant. To receive the share, it's also constant time decrypting the constant sized message and creating the new constant-sized key pointer node on the receiver's side.

To revoke scheme is $\Theta(N + M)$, where N is the size of the revoked file and M is the number of users that the file is still *directly* shared with by the user after the revocation. Note that it takes $\Theta(N)$ time to re-encrypt the file, and it takes $\Theta(1)$ time to update the encryption keys for each shared user's constant-sized, shared pointer node and we may have M such direcrtly-shared users.

For all schemes above, if a file is deeply shared across many users (e.g. main user shares with user B, who shares with user C, who shares with user D, ...). There is a constant $\Theta(N)$ addition time needed to resolve the file data ID, where N is the length of the indirect share chain.