Design Document

Data Structures

```go
// utility type for authentically encrypted arbitrary data
type TaggedCipherText struct {
    CipherText []byte
    Tag        []byte
}

// uuid: hash("User/" + username);
// symmetric enc key = PBKDF(password, uuid)
// symmetric mac key = HBKDF(password, "mac")
type User struct {
    Username string
    DecKey   PKEDecKey
    SignKey  DSSignKey
    rootKey  []byte // == PBKDF(password, uuid)
}

// if accessing:
//     uuid: hash("FileInfo/" + username + filename)
//     symmetric enc key = HBKDF(User.RootKey, filename + "/encKey")
//     symmetric mac key = HBKDF(User.RootKey, filename + "/macKey")
// if sharing, hybrid encryption:
//     uuid: random
//     enc/dec key: recipient's
//     sign/verify key: sender's
type FileInfo struct {
    selfID      uuid.UUID
    Owner       string // owner's username
    Inviter     string // inviter's username
    RootInviter string // root inviter's username
    FileID      uuid.UUID
    FileKeyID   uuid.UUID // uuid of the file's FileKey
}

// uuid: random
// if sharing:
//     enc/dec key: recipient's
//     sign/verify key: user(sender)'s
// if accessing:
//     enc/dec key: user(recipient)'s
//     sign/verify key: inviter(sender)'s in normal case,
//                      owner's if owner has revoked access of another root inviter;
//                      data is considered valid if it can be verified by either key
type FileKey struct {
```

```
    selfID uuid
    EncKey []byte // AES Key for encryption/decryption
    MacKey []byte // AES Key for MAC
}

// uuid: random
// symmetric enc key = FileKey.EncKey
// symmetric mac key = FileKey.MacKey
type File struct {
    NumBlocks         int // number of FileBlocks associated with this file
    LastBlockId       uuid
    InvitationTableID uuid // uuid of the file's InvitationTable
}

// uuid: random
// symmetric enc key = HBKDF(FileKey.EncKey, "Block/" + block index)
// symmetric mac key = HBKDF(FileKey.MacKey, "Block/" + block index)
type FileBlock struct {
    Data        []byte
    PrevBlockId uuid
}

// uuid: random
// symmetric enc key = HBKDF(FileKey.EncKey, "InvitationTable")
// symmetric mac key = HBKDF(FileKey.MacKey, "InvitationTable")
// e.g. {B: [B's FileKey uuid, D's uuid, E's uuid, F's uuid], C: [C's uuid, G's uuid]}
type InvitationTable map[[]byte][]uuid
```

Helper Functions

1. `func AuthEncrypt(data []byte, encKey []byte, signKey []bytes) TaggedCipherText`
    - Encrypts `data` with `encKey` and puts the MAC of the ciphertext in `tag`.
2. `func AuthDecrypt(data TaggedCipherText, decKey []byte, verKey []byte) ([]byte, error)`
    - Verifies the `tag` with `verKey` and decrypts `data.CipherText` with `decKey`.

User Authentication

When a user signs up, we deterministically generate the uuid for the user by hashing the username. We then check if that uuid already exist (which means the username is already taken). If it is, we return an error. If it is not, we generate a random salt and hash the password with the salt. We then generate a pair of RSA encryption/decryption keys and a pair of RSA signing/verification keys. We store the two public keys in the keystore. We also generate an empty `FileTable` struct. We *encrypt-then-MAC* each of them with symmetric keys generated from the hashed password. Then we store needed fields in a `User` struct and encrypt it with the hashed password. Then we create an `UserContainer` struct and store the salt and the *encrypted* `User` struct in it. Lastly we store the `UserContainer` and the authentically encrypted `FileTable` in the datastore.

When a user logs in, we generate the uuid for the user by hashing the username. We then retrieve the salt from the datastore and hash the inputted password with the salt. We then try to decrypt the encrypted `User` with the hashed inputted password. If the decrypted bytes can be un-marshalled into a valid `User` struct, then we know the password is correct and return the `User` struct, otherwise we return an error.

Multiple Devices

Note that the `User` struct is the only thing we reuse (stores locally instead of fetch on-the-fly) and none of its fields are mutable (never changes after the user is created). So multiple devices are trivially supported.

File Storage and Retrieval

When user stores a file, we first determine if the file already exist in the user's `FileTable`.
If it does, we retrieve the file's uuid and keys from the `FileTable` and fetch the file from the datastore. Then we verify and decrypt the `File` struct with the file's key. We retrieve the `LastBlockID` and fetch the `FileBlock` from the datastore. Now we verify and decrypt the `FileBlock` with the file's keys, retrieve the `PrevBlockID`, and delete this current `FileBlock` from the datastore. We repeat the process until we reach the first `FileBlock`. Now we create a new `FileBlock` with the content, authentically encrypt it with the file's keys, and store it in the datastore under a random uuid. We then update the `File` struct with the new `FileBlock` 's uuid and authentically encrypt it with the file's keys.
If the file does not exist, we first create two random symmetric keys for the file. Then we create a new `FileBlock` with the content, authentically encrypt it with the file's keys and store it under a random uuid. Now we create a new `File` with the `FileBlock` 's uuid as the `LastBlockID`, authentically encrypt it with the file's keys, and store it under a random uuid. We then update the `FileTable` with the new file's uuid and keys.

When user loads a file, we first determine if the file exist in the user's `FileTable`. If it doesn't, we return an error. Otherwise, we retrieve the file's `UpdateKeysInvitationID` and check if this uuid exist in datastore. If so, we fetch the `Invitation` struct from the datastore and verify it with owner's verification key and decrypt it with our decryption key. Then we update the file's key with the ones from this `Invitation`, delete the `Invitation`, and replace the `UpdateKeysInvitationID` with a new random uuid. Now we are sure that we have the up-to-date keys for the file, we fetch the `File` struct from the datastore and verifies and decrypt it with the file's keys. If we updated the keys, we also need to find the corresponding `InvitationNode` and copy the new `UpdateKeysInvitationID` to there.
We retrieve the `LastBlockID` and fetch the `FileBlock` from the datastore. Now we verify and decrypt the `FileBlock` with the file's keys and retrieve the block's content and `PrevBlockID`. We repeat the process until we reach the first `FileBlock`. Now we concatenate all the `FileBlock` s' data and return it.

Efficient Append

We need to download the user's `FileTable` and the corresponding `File` struct. We also need to upload the new `FileBlock` and the updated the `File` struct. The total size of data is scaled with only the size of the append.

File Sharing

When creating the invitation, we create a `Invitation` struct containing the file's uuid and keys. We then authentically encrypt the `Invitation` struct with the recipient's public encryption key and the sender's private signing key and store it in the datastore under a random uuid. We then update the `InvitationTable` with the new invitation's uuid. This uuid is returned by the function.
When the recipient accepts the invitation, we first fetch the `Invitation` struct from the datastore, then verify it with the sender's public verifying key and decrypt it with the recipient's private decryption key. Now we fetch the `File` from datastore and verify and decrypt it with the file's keys. We then find the corresponding `InvitationNode` in the `InvitationTable` and change the `InvitationID` field to `nil`. We then update the recipient's `FileTable` with the file's uuid and keys. Now the recipient has access to the file. The process is the same for both the file owner and non-owners.

Take the diagram as reference and suppose A revokes B's access. We first fetch the `File` struct from the datastore and decrypt it with the file's keys. We then fetch the `InvitationTable` from the datastore. We iterate through the all `InvitationNode`s for each root node in `InvitationTable` and delete each `Invitation` associated with the node. Then we delete the key `"B"` from the map `InvitationTable`. Now we generate new keys for the file. We then decrypt all `FileBlocks` of the file with the old keys, and encrypt them with the new keys as well as the `File` struct. Then we update the owner's `FileTable` with the new keys. Then we iterate through the `InvitationNode`s of each node in `InvitationTable` (now without key `"B"`). For each node, we create a new `Invitation` struct with the file's uuid and new keys. We then encrypt the `Invitation` struct with the recipient's public encryption key and the owner's private signing key and store it in the datastore under the uuid specified in the `UpdateKeysInvitationID` field.