# CS161 Project 2 Design Document

Ishaan Mishra, Yash Gupta

**Note on Symmetric Encryption**: Each struct is encrypted and tagged with same "Keys source". The "Keys source" is used as the root key in the `userlib.HashKDF()` function to derive separate encryption and mac keys.

## Data Structures

Our design uses the following structs:

1. `struct User`: Corresponds to a user and has the following attributes: `username string` - The username of the instance of this user struct; `RootKey []byte` - The single, symmetric root key for this user; `DK PKEDecKey`- The private, decryption key for this user; `SK DSSignKey`- The private, signing digital signature for this user.

2. `struct CipherTag`: Allows us to upload encrypted data and its corresponding tag. Has the following attributes: `Ciphertext []byte` - The ciphertext of the marshalled and encrypted data; `Tag []byte` - The tag of the data (generated on the ciphertext).

3. `struct FileNode`: Corresponds to a single append to a file. Has the following attributes: `Content []byte` - The actual content of this append of the file; `Next uuid.UUID` - Pointer to the next append (`FileNode`) of the file. `nil` if this the last (empty) node in the linked list; `NextKeysSrc []byte` - Keys source for the next append (`FileNode`).

4. `struct FileHeader`: Stores a pointer to the first append and the empty node after the last append of the file. Has the following attributes: `HeadFileNodePtr uuid.UUID` - Pointer to the first append (`FileNode`) of the file; `HeadNodeKeysSrc []byte` - Keys source for the first append of the file `TailFileNodePtr uuid.UUID` - Pointer to the empty node after the last append of the file; `TailNodeKeysSrc []byte` - Keys source for this empty node afte the last append.

5. `struct File`: Corresponds to a file in each user's namespace. Has the following attributes: `IsOwned bool` - Whether the file is owned by the this user; `FileHeaderPtr uuid.UUID` - Pointer to the `FileHeader` struct instance of this file. (`nil` if this user is not the owner); `FileHeaderKeysSrc []byte` - Keys source for the `FileHeader` struct instance; `InvitationPtr uuid.UUID` - A pointer to the invitation for this file (`nil` if this user is the owner).

6. `struct InviteSubTree`: Stores sharing/revocation information for each first-level subtree in the invite tree of a file (users who the owner directly shares the file with). Has the following attributes: `FileHeaderPtr uuid.UUID` - Pointer to the `FileHeader` struct instance; `FileHeaderKeysSrc []byte` - Keys source for the `FileHeader` struct instance; `Revoked bool` - Whether or not this sub-tree's (head's) invite has been revoked

7. `struct Invitation`: Corresponds to an invitation from one user to another for a file. Has the following attributes: `InviteSubTreePtr uuid.UUID` - Pointer to the `InviteSubTree` struct instance for this sub-tree for this file invite tree; `InviteSubTreePtr uuid.UUID` - Keys source for the `InviteSubTree` struct instance.

8. `struct FileInvites`: Is accessible only by the owner. Stores information to access all `InviteSubTree` structs for this file (to enable revocation): `InviteSubTreeMap map[string]uuid.UUID` - Map from the

username of an invitee to the `InviteSubTree` struct for the sub-tree rooted at that invitee; `InviteSubTreeKeysSrcs map[string][]byte` - Map from username to the keys source for the `InviteSubTree` struct for the sub-tree rooted at that invitee.

## Database diagram

Keystore entries:

1. Users' public encryption key (`PKEEncKey`): key = `username + "_enckey"`
2. Users' DS verify key (`DSVerifyKey`): uuid = `username + "_verifykey"`

Datastore entries:

1. `User struct`s: uuid = `Hash(username)`; KeysSrc = `Argon2Key(password, salt=Argon2Key([]byte(username), []byte("0xdeadbeef"), 16))`
2. `File struct`s: uuid = `Hash(Hash([]byte(username)) || Hash([]byte(filename)))`; KeysSrc = `HashKDF(rootKey, filename)`
3. `FileHeader struct`s: uuid = Random, stored in: 1) owner's `File` struct; 2)each `InviteSubTree` struct; KeysSrc = same as uuid
4. `FileNode struct`s: uuid = Random, stored in previous `FileNode`'s `Next` (and `FileHeader`); KeysSrc = same as uuid
5. `InviteSubTree struct`s: uuid = Random, stored in 1) `FileInvites` for owner, 2) `Invitation`s for invitees; KeysSrc = Same as uuid
6. `Invitation struct`s: uuid = Random, stored in sharee's `File` struct; PKE keys (before accepting): {Enc/Dec = sharee's public/private encryption key pair, Sign/Verify = sharer's DS sign/verify key pair}; Sym KeysSrc (after accepting) = `HashKDF(rootKey, filename+"_invitation")`
7. `FileInvites struct`s: uuid = `Hash(Hash([]byte(username)) || Hash([]byte(filename) || Hash([]byte("invites"))))`; keysSrc = `HashKDF(rootKey, filename+"_invites")`

## Helper Functions

#### Keys for Structs

`HashKDF16`: This function is the same as `userlib.HashKDF` function, but returns only its first 16 bytes.

`getEncMacKeys`: This function takes in `keysSrc` and uses it to derive an encryption key and mac key: `encKey = HashKDF16(keysSrc, "encryption")`, `macKey = HashKDF16(keysSrc, "mac")`.

#### Symmetric Key Encryption

`encryptAndMac`: This function encrypts and tags marshalled plaintext. It first gets the symmetric key and mac key using the `getEncMacKeys` function. We then encrypt the plaintext using the symKey, and tag that ciphertext with the macKey. We store the ciphertext and mac in the `CipherTag` struct, marshal it and return.

`verifyAndDecrypt`: This function verifies and then decrypts a given ciphertext. We first obtain the symKey and macKey. We unmarshal the encrypted data to get the `CipherTag` struct. We then verify the tag using our macKey by calling HMACEval. Then, we decrypt and return the ciphertext using.

`encryptMacStore`: This function first marshals the data, calls `encryptAndMac` helper function on the marshaled data, and stores the returned encrypted data at the given UUID.

`fetchVerifyDecrypt`: This function gets the data at the given UUID, calls `verifyAndDecrypt` on the encrypted data and return the marshalled, decrypted data. The data has to be unmarshalled after this function.

**Public Key Encryption**

`encryptAndSignPKE`: This function returns the encrypted and signed ciphertag using public key encryption. The ciphertext is first encrypted with the given encryption key, and signed with the sign key. This ciphertag is then marshaled and returned.

`verifyAndDecryptPKE`: This function verifies and decrypts asymmetrically encrypted data. We use our verifyKey to verify the ciphertext, and the public decryption key (decKey) to decrypt the data.

`encryptMacStorePKE`: This function encrypts and stores data using public key encryption. It marshals the data, encrypts it with the recipient's encryption key (from the Keystore), and signs it using the sender's private sign key. The encrypted data is stored on the datastore at the given UUID.

`fetchVerifyDecryptPKE`: This function fetches the data from the datastore, verifies using the sender's public verification key (from the Keystore), decrypts using the invitee's private decryption key, and the returns the decrypted, unmarshaled data.

**Getters for Structs** `getFileHeaderMetaFromFile`: This function gets the FileHeader's UUID and KeysSrc for a user's file. If the file is owned, the FileHeader UUID and KeySrc stored in the File struct are returned. If the file is not owned, we use the File struct to get the Invitation struct, get the InviteSubTree struct from Invitation, and get the FileHeader info from the InviteSubTree struct (if the file is not revoked).

`getFileStruct`: This helper function takes in the filename and returns the corresponding file struct. It derives the file struct's UUID and KeysSrc (see Database diagram). Then, the file struct is fetched using `fetchVerifyDecrypt`, unmarshaled and returned.

`getFileHeaderMeta`: This function uses `getFileStruct` and `getFileHeaderMetaFromFile` to return FileHeader's UUID and KeysSrc from a filename

`getFileMeta`: This function takes in the filename, and returns the file's UUID and KeysSrc.

## User Authentication

For each user, we create their root key, private decryption key, and private DS signing key in the User struct. These attributes are never changed again, and any other user-related changes (like files and invitations) are directly on the DataStore. Thus, all User methods maintain sync across all client instances of the same user. To encrypt, we first generate a `salt = Argon2Key(username, "0xdeadbeef")` in InitUser(). This makes a random-looking, public salt. Then we use Argon2Key(password, salt) to generate the keys source that will be used to encrypt/decrypt and tag/verify. The user struct is then encrypted and mac-ed with the derived enc and mac keys, and stored at uuid of `Hash(username)`. To get an encrypted user, we first get the user's uuid as `Hash(username)`. We then generate the keys source (same as above). Finally, we can obtain the User struct by calling the fetchVerifyDecrypt helper function on uuid, userkeysSrc and struct name.

## File Storage and Retrieval

For each file, we have `FileNode structs`, and a `FileHeader struct`. The `FileNode` structs store the contents of subsequent appends of a file in a linked list structure, and the `FileHeader` struct stores pointers to

the first and last `FileNode` of a file. Then, for each file for each user, we have a `File struct`, which stores metadata about the file and the user. If the file is owned, the File struct has the uuid/keysSrc for the FileHeader. If the file is not owned, the File struct has the uuid/keysSrc for the Invitation struct, which as the uuid/keysSrc for the InviteSubTree, which has the uuid/keysSrc for the FileHeader. To store a file, we first create the Head FileNode, and create random uuid/keysSrc for the FileNode. We then store the fileNode using `encryptMacStore`. If the file already existed: We decrypt the FileHeader (using uuid/keysSrc from `getFileHeaderMeta`), modify the FileHeader, and encrypt and store it using the same uuid/keysSrc. If the file doesn't already exist: We create and store a fileHeader (using random uuid and keysSrc). Then, we create (with empty maps) and store the FileInvites struct, using derived keysSrc and uuuid (see database diagram), and `encryptMacStore`. Next, we create the File Struct, derive its keysSrc and uuid (see database diagram) and store it using `encryptMacStore`. The keysSrc/uuid for the FileHeader are stored in the File Struct.

To load a file, we do the following, we first fetch the fileHeader for the given filename using `getFileHeaderMeta` and `fetchVerifyDecrypt`, and unmarshal the obtained fileHeader. Then, we traverse through the Linked List of FileNodes: use the keysSrc and uuid of the previous FileNode or FileHeader and `fetchVerifyDecrypt` to get the current FileNode, and append its contents to the return value. Finally, we return the appended contents.

To append to a file, we get the FileHeader using `getFileHeaderMeta` and `fetchVerifyDecrypt`, create a new FileNode for the new content at a new location, and store the FileNode using encryptMacStore, update the FileHeader and store it on the DataStore.

We can see that appending to a file does not depend on the size of the file or previous append (none of the previous filecontents need to be loaded/stored), the number of files the user has (since all file data for other files is stored in separate File structs with derive-able uuids/keys), or the number of users the file is shared with (since the file is not copied, and info on invitees is stored in separate FileInvites struct with derivable uuids/keys)

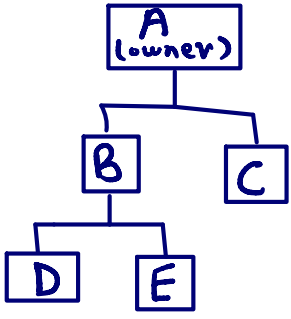
## File Sharing and Revocation

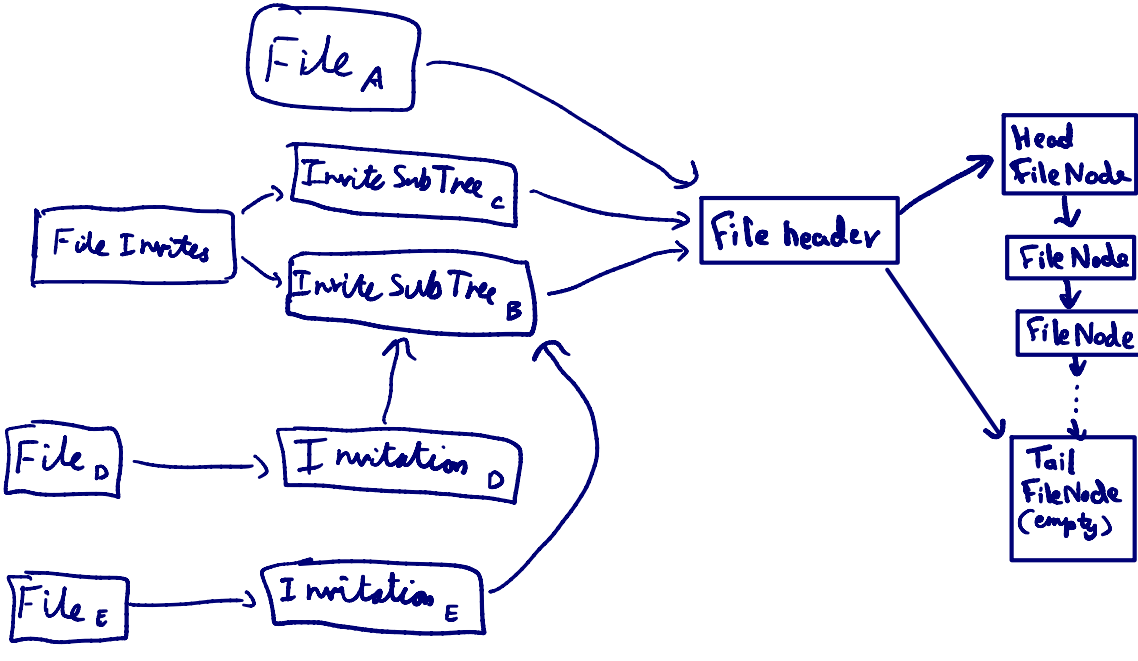
To create an invitation: If we are the owner: We create a InviteSubTree struct, with data to access the FileHeader (from the owner's file struct), create its uuid/keysSrc (random), and store it. We then enter the new InviteSubTree struct's info in the FileInvites struct and store it. If not the owner: We get our own invitation, get the InviteSubTree's keysSrc/uuid from our invitation. We then create the new invitation with the InviteSubTree's uuid/keysSrc (either from our invitation or newly created). We then encrypt it using asymmetric encryption (invitee's public encryption key), sign it with sender's private DS sign key, store it, and return the UUID.

To accept an invitation, we first re-encrypt our invitation with our own symmetric keys. Then, we create a file struct, similar to StoreFile (but without creating FileInvites), and store information for the invitation in the file struct. We then just encrypt and store the file struct in the same way as StoreFile.

To revoke access from a given user and people they've shared a file with, we first update the revoked user's InviteSubTree to show revoked. We then remove the revoked user's entries from the FileInvites struct. Next, we shift the FileNodes to new locations with new keys to prevent a revoked user from accessing the file/gaining information about it: We get the FileHeader using `getFileHeaderMeta(filename)`, and `fetchVerifyDecrypt`. We then traverse the linked list of FileNodes and update the FileNodes; pointer and keysSrc for each fileNode. Finally, we create a new FileHeader and store it at a new location using a new key. The new FileHeader information is stored in the owner's file struct. We also update the non-revoked users' InviteSubTree structs, which we find in FileInvites.

Now, since the keys and locations have changed, the revoked users will be unable to verify any file contents. Further, they shouldn't be able to do anything too malicious because file locations have also changed.


A
(owner)
B
C
D
E


File A
Invite SubTree C
File Invites
Invite SubTree B
File header
Head FileNode
FileNode
FileNode
Tail FileNode (empty)
File D
Invitation D
File E
Invitation E