Design of the End-to-End Encryption File-Sharing System

I. System design

This system will be able to store, share, revoke access to, and edit files, while also ensuring confidentiality, integrity, and authentication between users and file contents. When a user stores a file, the file will be stored in the user's dictionary of files, where the key is the **hashed** file name, and the value is a struct containing information (or file metadata), such as who the file's owner is, who has access to the file (which is saved in a dictionary that only the owner can see), and a pointer to the invitation struct (which then points to the file itself). The file itself has the structure of a struct, which holds encrypted file contents. Check the **Figure 1** diagram to see the design data structure in more detail. The file contents are encrypted by using an authenticated encryption scheme, where the file contents will go through an AES-CTR encryption scheme (which enforces **confidentiality**), and HMAC (which enforces **integrity**). To provide high-quality confidentiality and integrity, the process in which the algorithm is applied will be **encrypt-then-MAC**.

Files will be shared with another user in this manner: The sender of the file - let's call that person Alice - will use the User.CreateInvitation method to invite the recipient user - let's call that person Bob - to share a file with her. The User.CreateInvitation method will check if the file name is in Alice's namespace and if Alice is the owner of the file.

If Alice is the owner of the file, then Alice will do three things: first **create a new (authentically encrypted) invitation struct** that will have three fields: the owner name, a list of the user names sent to this same struct, and the UUID/pointer to the file struct (with its encrypted content). Every child of the owner will have its own invitation struct, but once that child shares the file themselves, *their children* will be sent to the same invitation struct they are in. That way, it creates a tree-like structure of sharing (designed in the project spec), and when a user is denied access to a file (by the owner), the entire branch will not be able to access the file (i.e. the invitation struct that the owner's child, and every user the child has shared the file with, will be deleted). To see the structure of this design, see **Figure 2** below.

Secondly, Alice will add the invitation struct's UUID/pointer as a value into the dictionary (that only Alice can see) that keeps track of who has access to the file (called the User_Access_Dict seen in **Figures 1 and 3**). Lastly, Alice will create an invitation message, where the invitation_struct's UUID/pointer will be saved as a message, publicly encrypted and digitally signed, and stored in a struct (called "Publicly_Encrypted_Contents", seen in **Figure 1**). That encrypted message will then be sent to Bob.

When Bob accepts the message (using User.AcceptInvitation) Bob will follow the UUID/pointer to the struct containing the publicly encrypted message, verify that the message came from Alice, and then decrypt the invitation message (which will have the actual invitation struct's UUID). Bob will then be able to access the shared file by decrypting the invitation *struct* and following the file's pointer/UUID (saved in the invitation struct) to the file (with its encrypted file contents).

Revoking a user's access from a file will proceed in this manner: the program will check if the file is in the user's namespace, and if the user is the owner by looking at the file's metadata (look at the FileMetada struct in **Figure 1**). Then, the file struct's UUID will change - so that the revoked-user-turned-malicious cannot access the file directly by using the Datastore - and the file's new UUID will be saved in the owner's invitation struct. After that, the program will access the owner's User_Access_Dictionary (saved in the file metadata), and loop through each element within the dictionary. Each element will carry the UUID/pointer to an invitation struct (created when the owner shared the file with a user), and within the invitation struct, the program will check if the recipient's username (the name of the user the owner is revoking access to) is in the Users_sent_here list (see **Figure 2** for more detail). If the recipient's username **is** in the list, then that invitation struct's field of the file's UUID/pointer **will not** be updated with the new UUID/pointer to the file, and the **invitation struct will be deleted from the Datastore**. If the recipient's username **is not** in the list, the invitation struct's field of the UUID to the file **will** be updated with the new UUID/pointer to the file. This way, the revoked user will not be able to use the API to access or edit the file, and the users who should be able to access the file will still be able to access and edit the file.

Lastly, the system will make appending to files efficient by storing each (authentically encrypted) appended file content in a **new struct,** and saving that new struct in the Datastore. Since the file struct saves the original file content's UUID/pointer and keeps track of the number of appends made to the file (see **Figures 1 & 2**), the system will be able to **deterministically create a new UUID** for the newly appended file content struct (by incrementing the number of appends and adding the number of appends to the original file content's UUID). Once the new UUID is created, the system will then authentically encrypt the new file contents, put it in a new struct, and save that (Marshalled) new struct with the new UUID in the Datastore. If a user wants to load the file later on, each of the appended file content structs will be decrypted, saved in a (temporary) new list, where that new list will be appended to the (decrypted) original file contents (once the last appended file content has been decrypted and saved in the new list), and saved in the Datastore. Finally, the appended file content structs will be deleted from the Datastore (since it has been saved and appended to the original file struct), and the number of appends will be reset to 0.

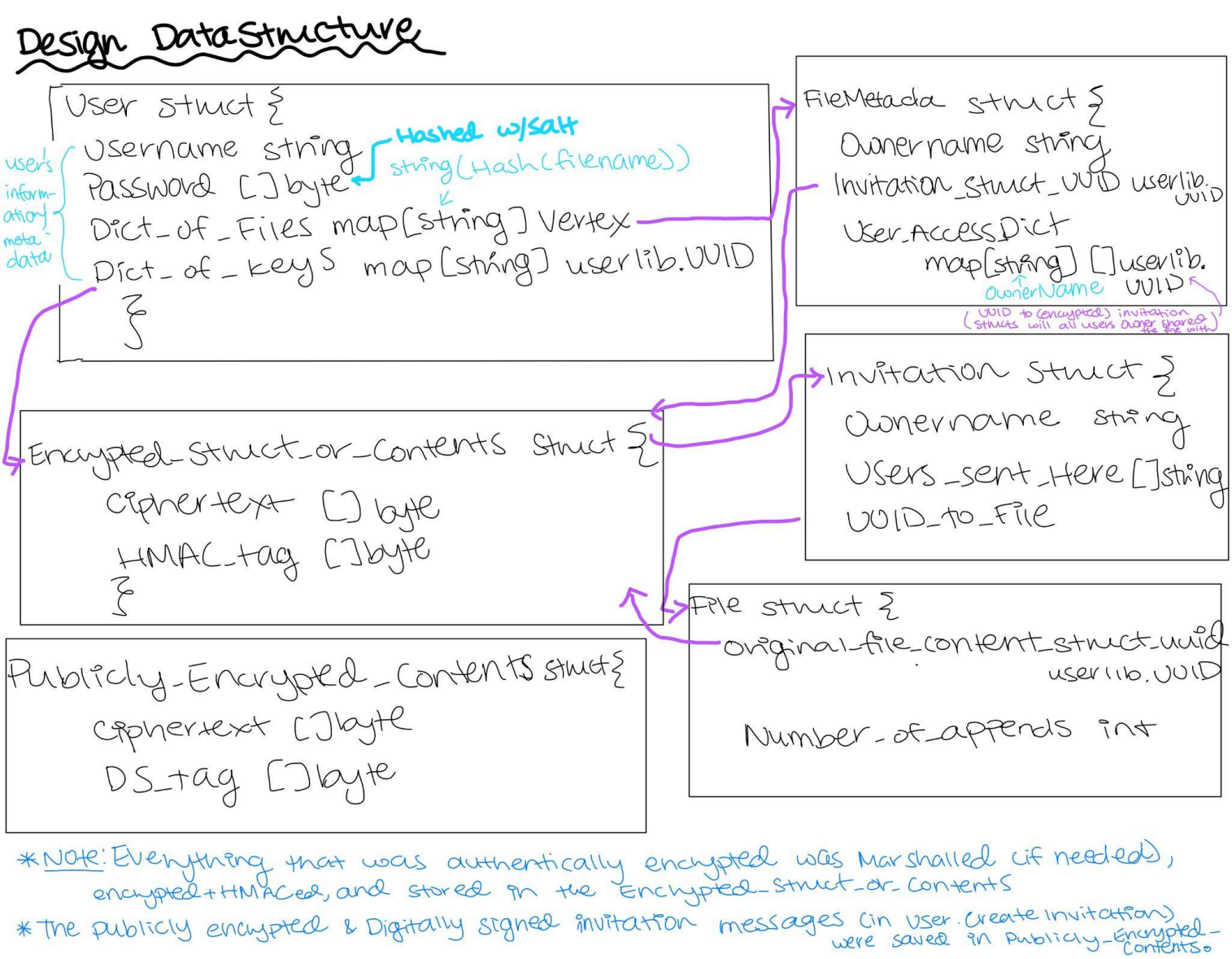
II. Security Analysis

To prevent an attacker from logging in (and acting) as another user, our system will make sure that the only way an attacker can log in as another user (using User.GetUser) is if the attacker knows the victim's actual password. In our system, the victim's password is salted and hashed, so the password itself is never saved in the system. Also, when a user logs in, the system will grab the password the user inputted, hash it (with the salt), and check that it matches with the (salted and hashed) password saved in the database. If it doesn't match, the user/attacker will not be able to log in. Hashing the password with a salt also prevents rainbow table attacks. Moreover, even if the attacker somehow knew the victim's actual password, the attacker would not be able to access a file without knowing the file name the victim saved the file as. This is due to the fact that the system never saves the file names, but only the **hashed** file name, and the system always checks if the (hashed) file is in the user's namespace before performing any act.

If user A is trying to share a file with user B, an attacker cannot intercept the invitation and gain access to user A's file, because our system uses (public key) encryption and digital signatures to make the invitation message confidential and authenticated. User A will encrypt the invitation message with user B's public key, so only user B can decrypt the message (with their private key) and access the information of the message (not the attacker). User A will also digitally sign the message (with user A's private key) so that user B can verify that the message came from user A (by using user A's public verification key). In addition to this, the system also used the encrypt-then-authenticate scheme, by encrypting the invitation message (using the recipient's public encryption key) first, and then signing the encrypted message (using the caller's private signing key). This way, no information can be leaked from the invitation message if it was intercepted by an attacker.

Lastly, when a user's access to a file has been revoked, our system will ensure that the revoked user cannot load the file and/or edit the file. Our system changes the file's location within the database, so that the revoked user cannot access the file directly from the Datastore, and our system blocks the flow of information (or the path to the file) by deleting the invitation struct that leads the user to the shared file. Therefore, the revoked user also cannot use the API to access the file.

Figure 1:



*Note: Everything that was authentically encrypted was Marshalled (if needed), encrypted+HMACed, and stored in the Encrypted_Struct_or_Contents

*The publicly encrypted & Digitally signed invitation messages (in User.CreateInvitation) were saved in Publicly-Encrypted-Contents.

Figure 2:

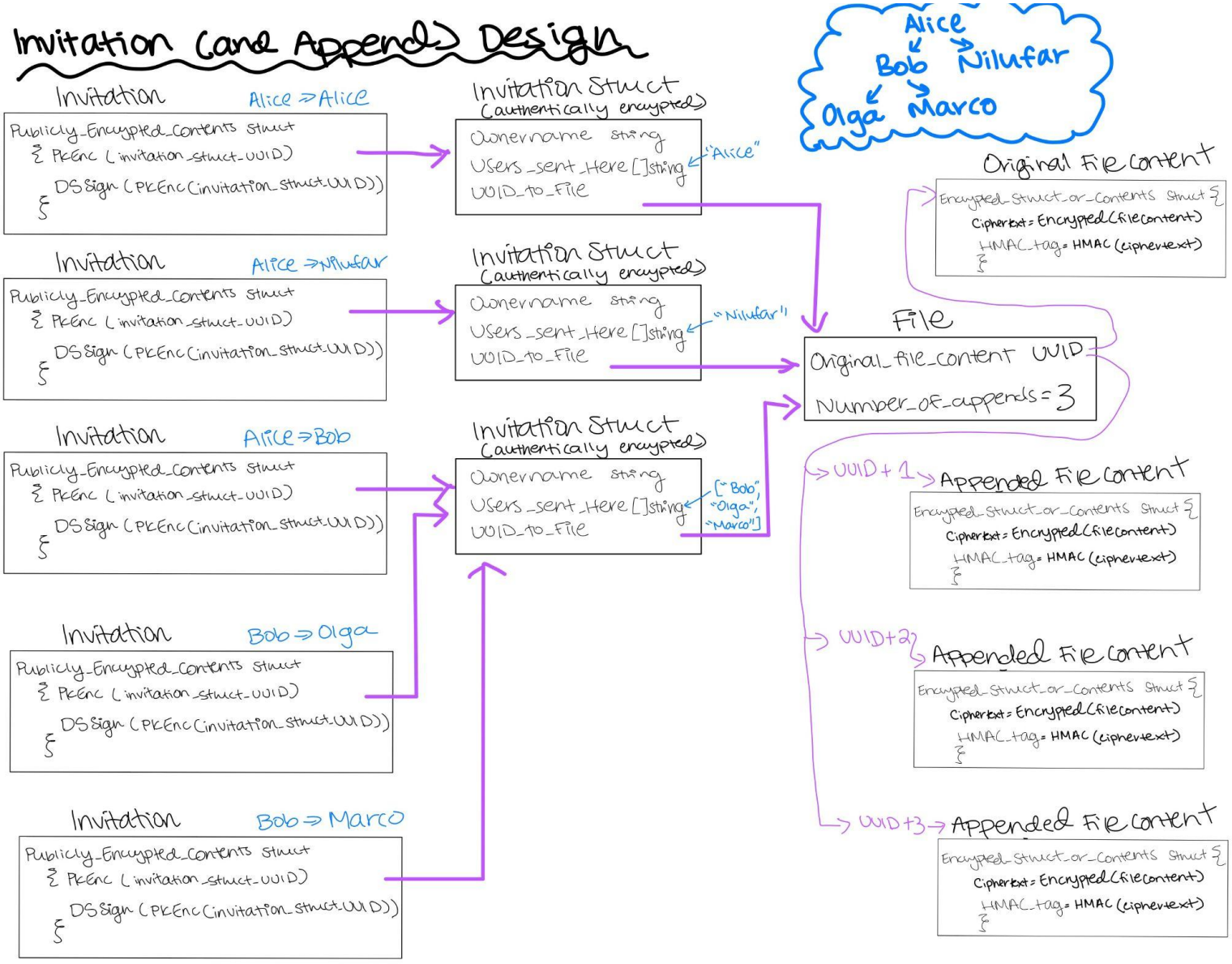
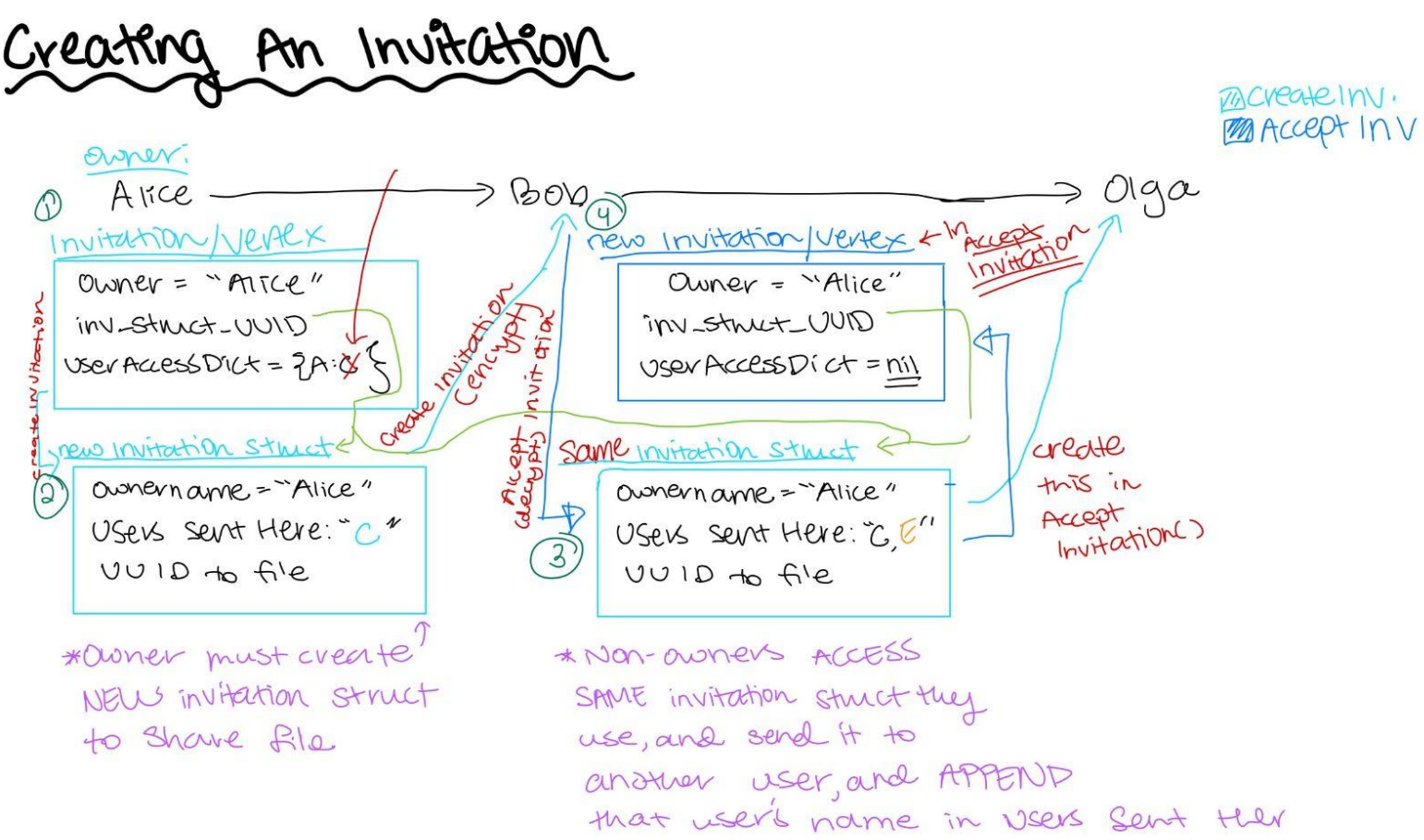

Invitation Card Append Design
Invitation
Alice → Alice
Publicly_Encrypted_Contents Struct
PkEnc (invitation_struct_UUID)
DSSign (PkEnc(invitation_struct_UUID))
Invitation
Alice → Nilufar
Publicly_Encrypted_Contents Struct
PkEnc (invitation_struct_UUID)
DSSign (PkEnc(invitation_struct_UUID))
Invitation
Alice → Bob
Publicly_Encrypted_Contents Struct
PkEnc (invitation_struct_UUID)
DSSign (PkEnc(invitation_struct_UUID))
Invitation
Bob → Olga
Publicly_Encrypted_Contents Struct
PkEnc (invitation_struct_UUID)
DSSign (PkEnc(invitation_struct_UUID))
Invitation
Bob → Marco
Publicly_Encrypted_Contents Struct
PkEnc (invitation_struct_UUID)
DSSign (PkEnc(invitation_struct_UUID))
Invitation Struct
(authentically encrypted)
Ownername string
Users_sent_Here []string
UUID_to_File
"Alice"
Invitation Struct
(authentically encrypted)
Ownername string
Users_sent_Here []string
UUID_to_File
"Nilufar"
Invitation Struct
(authentically encrypted)
Ownername string
Users_sent_Here []string
UUID_to_File
["Bob", "Olga", "Marco"]
Alice
Bob
Nilufar
Olga
Marco
Original File Content
Encrypted_Struct_or_Contents Struct
Ciphertext = Encrypted(file content)
HMAC_tag = HMAC (ciphertext)
File
Original_file_content UUID
Number_of_appends = 3
UUID + 1
Appended File Content
Encrypted_Struct_or_Contents Struct
Ciphertext = Encrypted(file content)
HMAC_tag = HMAC (ciphertext)
UUID+2
Appended File Content
Encrypted_Struct_or_Contents Struct
Ciphertext = Encrypted(file content)
HMAC_tag = HMAC (ciphertext)
UUID+3
Appended File Content
Encrypted_Struct_or_Contents Struct
Ciphertext = Encrypted(file content)
HMAC_tag = HMAC (ciphertext)

Figure 3:


Creating An Invitation
createInv.
AcceptInv
owner:
Alice
Bob
Olga
Invitation/Vertex
Owner = "Alice"
inv_struct_UUID
userAccessDict = {A:
new invitation struct
ownername = "Alice"
Users sent Here: "C"
UUID to file
create invitation (encrypt)
Accept invitation (decrypt)
new invitation/vertex
Owner = "Alice"
inv_struct_UUID
userAccessDict = nil
Accept invitation
same invitation struct
ownername = "Alice"
Users sent Here: "C, E"
UUID to file
create this in Accept Invitation()
*Owner must create NEW invitation struct to share file
*Non-owners ACCESS SAME invitation struct they use, and send it to another user, and APPEND that user's name in users sent her