```
questions:

- total bw not scaled with `total file size` so a file cannot reside in one
uuid?
- essentially necessitates (at least conceptually) some kind of linked-list like
append scheme.
- since len_user_name, len_password, # of users file shared with could not scale
either, this means none of those metadata could be store in the same uuid.

# RU-adversary
- ru has access to `DatastoreGet` and `DatastoreSet` on UUID that they have
access to (no global view of Datastore).
- NO rollback attacks on uuid.
- might have access to value associated with a UUID prior to revocation.
- Must protect ru from learning about future writes or appends to file.

# DS-adversary
- read/modify/add ALL name-value pairs in datastore.
- has global view of datastore.
- can see all arguments to functions (StoreFile)
- NO rollback attacks on UUID.



---------------------------------------------------------------------------
+ Keystore:  name -> public keys
+ Datastore: UUID -> data

# User
- InitUser:
  + Check if `username` already in Datastore. If yes, report error. If no,
create new user:
        + let `sym`=Argon2Key(pw); `pub, priv` = PKEKeyGen(); `salt` =
RandomBytes();
    + KS: `username` -> `pub`
        + DS: Hash(`username`) -> `salt` || Hash(`salt`, `plain_pw`);
        + DS: Hash(`username_User`) -> E_sym(`User{ priv }`)
- GetUser:
  + Check keystore for `username`.
  + Retrieve `salt` and makes sure Hash(`salt`, `plain_pw`) matches the value
retrieved.
  + If yes, retrieve Hash(`username_User`) and decrypt `User` using the
symmetric key obtained by Argon2Key(`plain_pw`).
  + If the adversary doesn't know the plain_pw, obtaining `User` would be
impossible.

# File operations:
- Store file:
  + newsym = RandomBytes();
  + DS: Hash(`salt` || `filename` || "key") -> { Hash(`salt` || `filename`);
Tree(`username`, PKEEnc(`User.pub`, newsym)) };

** Linked list structure:
0. Each file block has reserved 16 bytes (UUID) at the end as a pointer to
```

another file block.
1. First file block has reserved 16 bytes at the end to point to the last block in linked list.
2. Each non-first block points to the prior ones.

- Load file:
  + if Hash(`salt` || `filename` || "key") doesn't exist, then file doesn't exist.
  + else, load DS[Hash(`salt` || `filename`)]:
        + if the value is something like `(E_newsym(M || ptr), HMAC_K(E_newsym(M || ptr)))`, check integrity.
                + If the "ptr" is non-zero, travels the linked list and check integrity for each block.
                + travel linked list and reconstruct the file.
        + if the value of Hash(`salt` || `filename` || "key") is an invitationPtr:
                - try decrypting the encrypted key with our username in the corresponding tree.
                        If no encrypted key associated with our username, access has been revoked. outputs error.

- Append file:
  + if entry is invitationPtr, obtains `newsym` and go there.
  + allocate a new UUID and puts our content there `(E_newsym(M || ptr), HMAC_K(E_newsym(M || ptr)))`
  + adjust the first block pointer to points to the new UUID.
  + append 16 bytes UUID to the new value that points to the previous block

# Revocation:
Idea: Have a global location to store a list of PK(user)-encrypted key for decrypting a file. Each user will try to decrypt each key in the list with their private key and if they can, then they have access to the file decryption key.

- CreateInvitation:
  + let `pub_recip` be recipient's public key (implicit in this is the check that the recipient exists)
  + decrypt the file encryption key in entry Hash(`salt` || `filename` || "key") using our private key. let's call it `sym`
  + append to entry Hash(`salt` || `filename` || "key") the value PKEEnc(`pub_recip`, `sym`)
  + returns Hash(`salt` || `filename` || "key")

- AcceptInvitation:
  + create entry DS: Hash(`my_salt` || `my_filename` || "key") -> invitationPtr

- RevokeAccess:
  + obtain Hash(`salt` || `filename` || "key"):
        + if is invitationPtr; follows the entry.
        + Once we get to an entry with the `Tree`, we remove the revoked user and all their descendants.
        + we reencrypt all file nodes using a new key.
        + we then update the new key to each tree node, encrypted with the corresponding user's public key.

--> The ru adversary can't learn about future updates because each file block is reencrypted and the ptr is also encrypted so the RU cannot see if the ptr points to some newer UUID entry (which indicates an update)