CS628A Assignment 1 (Design Document)

Mayank Sharma, 160392 || Shivank Garg, 160658

February 16 2019

Note: We're still not sure of the data types (i.e. string vs byte slices vs hash.Hash), and the below code snippets should be taken as pseudo-codes only. Also, Symmetric Key Encryption is done with CFB Encryption method, with key = SymmetricKey.

1 Simple Upload/Download

1.1 User Structure

```
type User struct {
  Username      string                    // Encrypted with the Symmetric Key
  SymmetricKey  string                    // Argon2(password), given, password has high entropy
  PrivateKey    string                    // Encrypted with the Symmetric Key
  FileKeys      map[string]FileSharingKey // Indexed by hash(filename), FileSharingKey maps to
    the Current Sharing Key of the File
  HMAC          string                    // H(username + SymmetricKey + PrivateKey + FileKeys)
}

type FileSharingKey string // HashValue of (Owner.SymmetricKey + uuid as salt)
```

User struct stores all the credentials of user, and this structure is encrypted with SymmetricKey (≡ Argon2(password)) to assure confidentiality. We use HMAC to check integrity. It needs to be updated regulary, since FileKeys may change. FileKeys acts like a hashMap which stores FileSharingKey that is used to encrypt Block.Content.

```
type Data struct {
  UserData     map[string]User
  FileBlocks   map[string]Block
  FileMetadata map[string]MetaData
}
```

Data Struct will be directly stored in Marshalled form in DataStore. Any person (or Datastore itself) who wishes to access the Users or Files or FileBlocks firstly has to Unmarshal this structure, then access its fields.

1.2 InitUser, GetUser

When a user is created, a new pair of Pub/Priv Keys are generated and the Priv key is stored in encrypted form in User struct. When GetUser is called, the given Argon2(password) is used to Decrypt the whole User Struct, verify integrity with HMAC and return the values accordingly.

1.3 Store File and AppendFile

```
type MetaData struct {
  Owner             string             // hash(Owner)
  LastEditBy        string             // hash(LastEditByUserName)
  LastEditTime      time.Time          // hash(LastEditByUserName)
  FilenameMap       map[string]string  // Map from hash(username) to encrypted filename for that
    user (encrypted with symmetric key of that user)
  GenesisBlock      string             // HashValue(Owner + FilenameMap[Owner] + uuid nonce)
  GenesisUUIDNonce  string
  LastUUIDNonce     string
  LastBlock         string // HashValue(LastEditBy + FilenameMap[LastEditBy] + uuid nonce)
  HMAC              string // HMAC(key = FileSharingKey, Data = Owner, LastEditBy, LastEditTime
    , GenesisBlock, GenesisBlockNonce, LastUUIDNonce, LastBlock)
}
```

This stores the complete metadata about the various blocks created by a user. When StoreFile is called, we generate a GenesisUUIDNonce and use it to create a HashValue, which is used to index into the FileBlocks map of Data Struct. To maintain confidentiality, we use hashedValues instead of direct usernames. Similarly, whenever the user performs an AppendFile operation, a new Block is created and LastUUIDNonce and LastBlock is updated.

```
// Block is encrypted by the FileSharingKey
type Block struct {
  Content       []byte
  PrevBlockHash string
  NextBlockHash string
  HMAC          string
}
```

Suppose Alice creates new file "foobar". Filename is hashed (confidentiality) and this hash is used to index into User.FileKeys where a newly generated FileSharingKey is created. This acts like a Symmetric Key for encrypting/decrypting a "MetaData". A GenesisBlock is created which, which is indexed by HashValue (Owner + FilenameMap [Owner] + uuid as nonce) into the Data.FileBlocks map. The content of Block is encrypted with CFBEncryption (key=FileSharingKey) which is owned by the Owner.

When we AppendFile(), a new Block is created, MetaData.LastUUIDNonce is generated & LastBlock is filled with HashValue(LastEditBy + FilenameMap [Owner] + uuid as nonce) and stored in Data.FileBlocks map.

2 Sharing/Revoking

2.1 ShareFile() and ReceiveFile()

The User.FileKeys map stores the currently used FileSharingKey for a certain filename (indexed with hash(filename)). Suppose Bob wants to share a file ("foobar") with Alice. He encrypts "foobar" FileSharingKey with Alice's public key (which he fetched from keystore), and digitally signs this with his own private key to maintain confidentiality and integrity (communication medium is insecure). Alice, upon receiving this verifies signature, and decrypt the message with her PrivateKey, to get the FileSharingKey which was used to encrypt "foobar"s MetaData.

When Alice calls ReceiveFile(), and assigns some new name (say "slowmo") to this file, she keeps track of "slowmo" in MetaData.FilenameMap (indexed with hash("slowmo"). This hashing is done so as to ensure that Bob doesn't knows what name Alice calls her file (and vice-versa).

If now Alice appends something, she will call AppendFile, which will update LastEditBy (with the hash("Alice")). The rationale behind not storing "Alice" directly into LastEditBy is that if Bob revokes access at sometime later, and then gives access to Charlie, Charlie should't be able to know that Alice once had access to this file, hence ensuring confidentiality.

2.2 RevokeFile()

Note: In our design, we don't allow anyone except the Owner to revoke access to the shared file. Only the owner can revoke accesses that too of all the other persons in one go.

For Revoking the access, Owner (say Bob) changes the FileSharingKey itself. He will have to generate a new FileSharingKey, then use the older key to decrypt all Blocks, then re-encrypt them with new key. Finally, once this is done, Bob will re-encrypt the MetaData with his new Key, thereby blocking access to Alice completely of any read/write that she was able to perform earlier.

The above design ensures that sharing is "transitive" meaning if Bob shares with Alice and Alice shares with Charlie, all three of them access the same file, albeit with their own file-names.