Advice and Tips

This section does not contain any design requirements (i.e. you could complete the whole project without reading this section). However, we've compiled some advice about how to approach the project in this section.

Tips on Database Architecture

Whenever you're thinking about storing lists or maps of things in Datastore, it may help to think about how you can "flatten" your data structure - this might help with security, efficiency, and/or code complexity! As an example: let's say we have a list of restaurants, where each restaurant has a list of menu items. Let's say I want to figure out whether a particular restaurant has a specific menu item, and we're using a datastore-like backend (a key-value store).

Consider these two approaches to representing this information in some **key-value store:**

```
// Approach A
{
    "movies": {
        "mcu": ["iron man", "the incredible hulk", "iron man 2", "thor", "captain america", "the avengers"
        "dceu": ["batman"],
        "pixar": ["turning red"]
    }
}

// Approach B
{
   "movies/mcu": ["iron man", "the incredible hulk", "iron man 2", "thor", "captain america", "the avenger
   "movies/dceu": ["batman"],
   "movies/pixar": ["turning red"]
}
```

Both of these represent the same data, except instead of storing all of this information in one place, we're flattening this data out across multiple entries in our key-value store. In approach B, all I need to do is to index into the datastore at the string value I construct (e.g. in this case, `"movies/pixar"`), and I can retrieve all of the movies that are affiliated with Pixar without having to retrieve a bunch of other data that was previously stored at the same location!

If efficiency is measured in read/write bandwidth (like it is in this project), then it's much more efficient to represent our data in this flattened structure.

Minimizing Complexity

If you don't need to store a certain piece of data, then **don't store it.** Just recompute it (or re-fetch it, if it's coming from datastore) "on-the-fly" when you need it. It'll make your life easier!

Authenticated Encryption

In order to build in support for authenticated encryption, you may find it helpful to create a struct with two properties: (a) some arbitrary ciphertext, and (b) some tag (e.g. MAC, signature) over the ciphertext. Then, you could marshal that struct into a sequence of bytes, and store that in datastore. When loading it from datastore, you could (a) unmarshal it and check the tag for authenticity/integrity, and then decrypt the ciphertext and pass the plaintext downstream.

Checking for Errors

Throughout each API method, you'll probably have several dozen error checks. Take a look at the following code for an example of good and bad practice when it comes to handling errors.

```
// This is bad practice: the error is discarded!
value, _ = SomeAPICall(...)


// This is good practice: the error is checked!
value, err = SomeAPICall(...)
if (err != nil) {
    return err;
}
```

When an error is detected (e.g. malicious tampering occurred), your client just needs to immediately return. We don't care about what happens afterwards (e.g. program state could be messed up, could

have file inconsistency, etc. - none of this matters). In other words, we don't care about **recovery** – we solely care about **detection**.

You should almost never discard the output of an error: always, always check to see if an error occurred!

Tips For Writing Test Cases

Here are a few different ways to think about creative tests:

1. Functionality Tests
    a. Consider basic functionality for single + multiple users
    b. Consider different sequences of events of Client API methods
    c. Consider interleaving file sharing/revocation/loading/storing actions
2. Edge Case Tests
    a. Consider edge cases outlined in the Design Overview and other parts of this specification.
3. Security Tests
    a. Consider what happens when an attacker tampers with different pieces of data

In all cases, you should ensure that your Client API is following all requirements listed in the Design Overview.

Coverage Flags Tips

It's okay if your tests don't get all 20 test coverage points! Some flags are very tricky to get, and we don't expect everyone to get all the flags. If you're missing a few flags, a few points won't make or break your project. That said, higher test coverage does mean you're testing more edge cases, which means you can also be more confident in your own implementation if it's passing all your tests.

Writing Advanced Tests

1. Some students have reported success with fuzz testing, which uses lots of different random tampering attacks. Sometimes this can help you catch an edge case you weren't expecting.
2. Remember that your tests must work on any project implementation, not just your own implementation. This means you cannot assume anything about the design except the API calls that everyone implements (InitUser, GetUser, LoadFile, StoreFile, etc). For example, you cannot reference a field in a user struct in your tests, because other implementations might not have that field in their user struct.

a. The userlib has a nifty DatastoreGetMap function which returns the underlying go map structure of the Datastore. This can be used to modify the Datastore directly to simulate attacker action.

b. `DatastoreGetMap` can also be used to learn about how the Datastore is changed as a result of an API call. For example, you can scan the Datastore, perform an API call (e.g. StoreFile), then scan the Datastore again to see what has been updated. This can help you write more sophisticated tests that leverage information about what encrypted Datastore entries correspond to what data.

Writing Efficiency Tests

Here's a helper function you can use to measure efficiency.

```
// Helper function to measure bandwidth of a particular operation
measureBandwidth := func(probe func()) (bandwidth int) {
   before := userlib.DatastoreGetBandwidth()
   probe()
   after := userlib.DatastoreGetBandwidth()
   return after - before
}

// Example usage
bw = measureBandwidth(func() {
   alice.StoreFile(...)
})
```

Expect(err).ToNot(BeNil())

You should check for errors after every Client API call. This makes sure errors are caught as soon as they happen, instead of them propagating downstream. Don't make any assumptions over which methods will throw and which errors won't: use an `Expect` after each API method.

Remember the core principle around testing: the more lines of the staff solution you hit, the more flags you're likely to acquire! Think about all of the `if err != nil` cases that may be present in the staff code, and see if you're able to write tests to enter those cases.

```
value, err = user.InitUser(...);
```

```
Expect(err).ToNot(BeNil());
```

Notes on Key Reuse

In general, avoid reusing a key for multiple purposes. Why? Here's a snippet from a post that former CS 161 Professor David Wagner wrote:

> *Suppose I use a key `k` to encrypt and MAC a username (say), and elsewhere in the code I use the same key `k` to encrypt and MAC a filename. This might open up some attacks. For instance, an attacker might be able to take the encrypted username and copy-paste it over the encrypted filename. When the client decrypts the filename, everything decrypts fine, and the MAC is valid (so no error is detected), but now the filename has been replaced with a username.*
>
> *I've seen many protocols that have had subtle flaws like this.*
>
> *There are two ways to protect yourself from this. One way is to think really, really hard about these kind of copy-paste attacks whenever you reuse a key for multiple purposes. The other way is to never reuse a key for multiple purposes. I don't like to think hard. If I have to think hard, I often make mistakes. Or, as Brian Kernighan once wrote: "Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?" So, I recommend way #2: never reuse a key for multiple purposes.*
>
> *Hopefully this rationale will help you recognize the motivation behind that advice, which might help you recognize when the advice does or doesn't apply.*

Notes on Key Management

In order to simplify your key management scheme, it may be useful to store a small number of root keys, and re-derive many keys for specific purposes "on-the-fly" (a.k.a. when you need them) using the HashKDF function.

More Notes on Key Reuse

The term "key reuse" can be ambiguous.

As defined in our lecture, "key reuse" refers to the practice of using the same key in multiple algorithms. For example, if you use the same key to encrypt some data and HMAC that same data, this is a case of "key reuse" as defined in lecture. As discussed in lecture, you should always use one key per algorithm.

Note that using the same key to encrypt different pieces of data, or using the same key to MAC/sign different pieces of data, is not considered key reuse (as defined in lecture). Recall that the whole point of discarding one-time pads and introducing block ciphers was to allow for using the same key to encrypt different pieces of data.

Even though using one key per algorithm solves the "key reuse" problem from lecture, it doesn't necessarily mean that other issues with key reuse don't exist.

As an example, suppose you use the same, hard-coded key to encrypt every single value in your design. You use a different key (also hard-coded) to MAC every single value in your design. This is not a case of "key reuse" from lecture, because you did use one key per algorithm. However, this is still an insecure design because an attacker could read the hard-coded keys in your code and decrypt and modify all your stored data.

In summary: Reusing the same key in different algorithms is the "key reuse" problem from lecture, and you should always avoid this. Other cases of key reuse may exist in your design; it's your job to figure out when these cases are problematic.

Strings and Byte Arrays

Be very careful about casting byte arrays to strings. Byte arrays can contain any byte between 0 and 255, including some bytes that correspond to unprintable characters. If you cast a byte array containing unprintable characters to a string, the resulting string could have unexpected behavior. A safer way to convert byte arrays to strings (and back) is to use the `encoding/hex` library, which we've imported for you.

If you instead need to convert strings to byte arrays (and back), you can use `json.Marshal`. Recall that marshalling works on any data structure, including strings.
