# CS628A: Computer Systems Security Assignment 1

Bidya Sarkar          Supriya Suresh

$9^{th}$ August, 2019

Credits: This assignment and supporting material is based on the work of Nick Weaver and Cameron Rasmussen of the University of California, Berkeley.

## Introduction

Storing files on a server and sharing them with friends and collaborators is very useful. Commercial services like Dropbox or Google Drive are popular examples of a file store service (with convenient filesystem interfaces). But what if you couldn't trust the server you wanted to store your files on? What if you wanted to securely share and collaborate on files, even if the owner of the server is malicious? Especially since both Dropbox and Google Drive don't actually encrypt the user data.

In this assignment you will use Go to build the functions needed to implement "encrypted dropbox", a cryptographically authenticated and secure file store (Actually, a Key-Value store; we have abstracted concept of files into key-value pairs so that you don't have to deal with raw files directly).

## Getting Started

We want you to get your hands dirty designing and implementing your system. There are three parts of this project with different deadlines (provided towards end of the document). **Please read whole problem statement including submission deadlines before starting.**

We provide you a framework off of which to build for this assignment, located at
`https://github.com/sarkarbidya/CS628-assn1/blob/master/assn1.zip`

All of your code should go in `assn1.go`. We have provided the necessary functions for encryption and decryption in `userlib.go`, which you need to fetch using the command `go get github.com/sarkarbidya/CS628-assn1/userlib`. These functions include the following:

- Set and Get from the Datastore and Keystore

1

- RSA Key Generation

- RSA Encrypt and Decrypt

- RSA Sign and Verify

- HMAC and SHA256 MAC Creation and Equality Testing

- Argon2 Key Derivation (see course website for info on key derivation functions)

- CFB Encryption and Decryption

**You must use Go and the crypto API we provide you for this assignment. You must not use any other libraries**

You should strive to write clean, secure code.

## Setting Up Go

Go is a nice language but an awful name, it makes web searching a pain. So don't search for "go xyz", search for "golang xyz". Go has similarities to Python like list splicing and C like pointers, and you definitely want to learn about useful tips before getting started.

**Online Tutorial**: `https://tour.golang.org/welcome/1` helps you set up Go workspace from the comfort of your web browser. **Definitely do this before you start the project!**

**Book**: `https://golang.org/doc/effective_go.html` is very good.

**Installing**: `https://golang.org` Make sure you have a recent version (at least 1.9)

If you use emacs, melpa has a good go-mode package.

We will also use some additional external packages that you will have to fetch using `go get`, notably:

- `github.com/sarkarbidya/CS628-assn1/userlib`

- `github.com/google/uuid`

The userlib package includes both the crypto API for you and a `userlib_test.go` file. That test file offers a guide for both using the functions and shows how to write a set of tests for go.

# Secure File Store

Your task is to design and implement a secure file store. This file store can be used to store your own files securely, or to share files with other people you trust.

Your implementation should have two properties:

**Confidentiality.** Any data placed in the file store should be available only to you and people you share the file with. In particular, the server should not be able to learn any bits of information of any file you store, nor of the name of any file you store.

**Integrity.** You should be able to detect if any of your files have been modified while stored on the server and reject them if they have been. More formally, you should only accept changes to a file if the change was performed by either you or someone with whom you have shared access to the file.

**Note on security parameters.** It is sufficient that these properties hold with very high probability(i.e., no more risk than arises from brute forcing well-chosen cryptographic keys).

You are given access to two servers:

1. A storage server, which is **untrusted**, where you will store your files. It has three methods:

   - `DatastoreSet(key string, value []byte)`, which stores `value` at `key`
   - `DatastoreGet(key string)`, which returns the value stored at `key`
   - `DatastoreDelete(key string)`, which deletes that item from the data store.

2. A public key server, which is **trusted**, that allows you to receive other users' public keys. You have a secure channel to the public key server. It has two methods:

   - `KeystoreSet(key string, value []byte)`, which sets the public key for your username `key` to be `value`
   - `KeystoreGet(key string)`, which returns the public key for `key`

   **You are not to change the code for either API.** If you do, your code will not work with our autograder and you will get no credit.

The storage server is in practice, just a key-value store. The files you upload to the server are strings of text data. The storage server is untrusted-it can perform arbitrary malicious actions to any data you store there. **You should protect the confidentiality and integrity of any data you store on the server**.

We have a few additional functions in the datastore that allow you to manipulate the raw data under the hood for building tests.

The storage server has one namespace, so anything written by one user can be read or over-written by any other user who knows the `key`. Clients interacting with the storage server must take care to ensure that their own files are not overwritten by other clients. Other users or clients might be malicious.

We provide you a framework off of which to build:

- `assn1.go`
  Where you will write your implementation for secure storage. **Put all your code in this file.**

- `assn1_test.go`
  The testing infrastructure for the project. Add tests to this to test your functionality and security!

You must use our provided `userlib.go` API for all of your security-critical operations. Do not implement your own versions of symmetric (or asymmetric) key operations. We have provided all the necessary functions to implement the secure file storage, so carefully examine all functions given to you in `userlib.go`.

Your code must not spawn other processes, read or write to the file system, open any network connections, or otherwise attack the autograder. We will run your code in an isolated sandbox. You must not add any additional go language imports to your file. Any adversarial behaviour will be seen as cheating.

**Your code must not add any global variables**, as testing may involve saving the datastore and keystore, terminating your program, and then restarting it with a restored datastore and keystore. **You may use only the global variable(s) provided in the skeleton code if required but do not add any.**

## Part 1: A simple, but secure client

**Question 1**  *Simple Upload/Download*
Implement a file store with a secure upload/download interface. For Question 1, your task is to implement the methods `InitUser`, `GetUser`, `StoreFile`, `AppendFile`, `LoadFile`. The methods must ensure the following properties hold. See **RFC 2119** for the definitions of MUST, MUST NOT, SHOULD and MAY.

---

**Property 1** `InitUser(username string, password string)`  *MUST take the user's password, which is assumed to have good entropy, and use this to help populate the User data structure (including generating at least one random RSA key), securely store a copy of the data structure in the data store, register a public key in the keystore, and return the newly populated user data structure. The user's name MUST be confidential to the data store.*

**Property 2**  `GetUser(username string, password string)`  *If the username and password are correct this MUST load the appropriate information from the data store to populate the User data structure or, if the data is corrupted, return an error. If either the username or password are not correct it MUST return an error. You MAY return an error that does not distinguish between a bad username, bad password, or corrupted data.*

**Property 3** `StoreFile(filename string, data []byte)` *MUST place the value* `data` *at* `filename` *so that future* `LoadFiles` *for a block in* `filename` *return* `data` *at that particular block in the entire file. Any person other than the owner of the* `filename` *MUST NOT be able to learn even partial information about* `data` *or* `filename` *with probability better than random guesses, other than the length of the data written.*

**Property 4**  `AppendFile(filename string, data []byte)` *MUST append the value* `data` *at* `filename` *so that future* `LoadFiles` *for* `filename` *return* `data` *appended to the previous contents. AppendFile MAY return an error if some parts of the file are corrupted.*

*Any person other than the owner of* `filename` *MUST NOT be able to learn even partial information about* `data` *or* `filename`*, apart from the length of data and the number of appends conducted to the file, with probability better than random guesses.*

**Property 5 (Benign Setting)** *When not under attack by the storage server or another user,* `LoadFile(filename string, blockOffset integer)` *MUST return the* **last** *value stored at* `filename` *at blockOffset by the user or* `nil` *if no such file exists. It MUST NOT raise any error.*

**Property 5 (Attack Setting)** `LoadFile(filename string, blockOffset integer)` MUST NOT *ever return an* **incorrect** *value. A* `value` *(excluding* `nil`*) is "incorrect" if it is* **not** *one of the values currently or previously stored for* `filename` *at blockOffset by the current user.*

---

*LoadFile(filename, blockOffset integer)* MUST raise an error if under **any** attack by the server or other users that successfully corrupts the file. It MUST return an error of the file has been tampered with but some record of it still exists in the data store. It MUST return *nil* if it appears that no value for *filename* exists for the user.

You may assume file names are alphanumeric (they match the regex [A-Za-z0-9]+). File names will not be empty. This will be the case for all parts of the project. The contents of the file can be arbitrary: you must not make any assumptions there. You may assume usernames consist solely of lower-case letters ([a-z]+).

## File Organization

File should be stored in blocks. **You can assume that the length of the data to be stored or appended will always be a multiple of the blocksize.** If it isn't, your code **MUST** return an error. In order to accomplish this you can look at Linux's inode structure for storing files.

*https://www.slashroot.in/inode-and-its-structure-linux*

The relevant things to look at here are single indirect and double indirect pointers as demonstrated below.
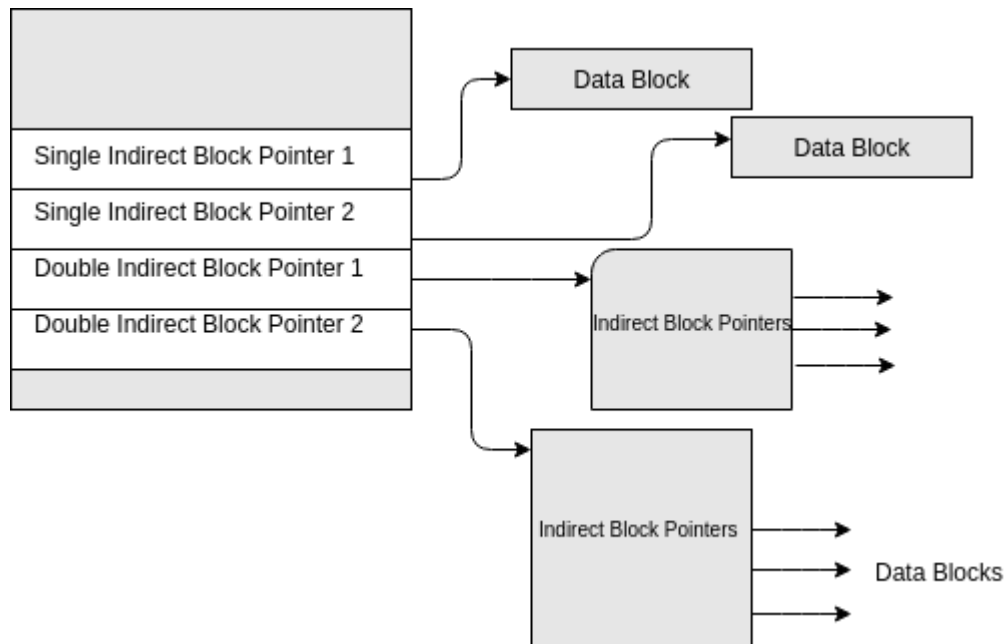


Your implementation of AppendFile should be efficient. When adding on to the end of a file, sending the unchanged bytes again is unnecessary work and should be avoided for a faster runtime. We will check for this in the autograder. As mentioned previously, length of data to append will always be a multiple of the blocksize. If not, it MUST return an error.

Block Size should be configurable in your implementation. In order to facilitate this, you can use the global variable provided in assn1_go - ***configBlockSize.* setBlockSize(newBlockSize integer)** changes the value of this variable and you can use this to get the current blocksize. We'll be testing your code using different block sizes and a call to setBlockSize() will only be made in the beginning of the execution and once the call is made, block size doesn't change for the entire execution, i.e, you can assume that once the block size is set, it will not be modified for the entirety of the execution. The function has been implemented for you. You are not allowed to change the value of this variable.

You will only need to implement 2-level storage, i.e, single indirect and double indirect pointers. Your design should support a file size of at least 800*BlockSize bytes. Supporting a file having a greater size is optional.

We have provided an implementation of the storage server - do not change it.

## Required Security Properties

Note that we require you to protect the confidentiality and integrity of both the contents of the file store, the name it is stored under, and the owner of the file. A malicious storage server must not be able to learn either, or change them. The length of the file and the length of the filename don't need to be kept confidential.

When used in a non-adversarial manner, different users should be allowed to have files with the same name: they should not overwrite each other's files. An adversary who has access to the datastore's list of keys may be to overwrite a user's valid data, but any changes should be reported as an error. An adversary who does not have access to the datastore's list of keys must not be able to overwrite or delete files.

While integrity is an intuitively simple property (a user should only accept data they themselves uploaded at that key), confidentiality has finer points which may not be obvious at first. Your scheme should be secure even if the adversary can convince a valid user to encrypt (or decrypt) arbitrary data. (The adversary controls the server. Moreover, they can trick you into encrypting arbitrary data for them.) Your client is secure if it satisfies the following scenario:

1. The adversary, who controls the server, constructs two file $F_0$ and $F_1$ of its choice, with $F_0 = (name_0, value_0)$, $F_1 = (name_1, value_1)$ where $name_0$ and $name_1$ have the same length, and $value_0$ and $value_1$ have the same length.

2. The adversary sends $F_0$ and $F_1$ to the client, who must choose one of them at random and upload it.

3. The adversary then sends the client any number of $F_i = (name_i, value_i)$ of its choice, where the only restriction is $name_i$ may not be $name_0$ or $name_1$. The client must upload these.

4. The client provides confidentiality of names and values if, after all this, the adversary cannot tell (with probability $> 1/2$) whether the client uploaded $F_0$ or $F_1$.

One specific attack **you are not required to handle** is that of **a rollback attack**: if Alice uploads the file $F$ to the server and then updates it later to $F'$ with a second upload, Alice does not need to detect if the server "rolls back" its state and returns $F$ when Alice requests the file back. This is why Property 1 is written as it is.

Why do we not require this? Without additional state, it would be impossible. The server could always rollback to the "empty" state where it contains no data at all, and return `None` for every `DatastoreGet`, and the client would not be able to detect this.

Your client must not assume it can keep any state outside the UserData structure and that state must either be directly derived from the password and/or loaded from the datastore.

You must assume that your client can be killed and restarted, and everything should still work. (For example: you cannot place a dictionary in your client, make `StoreFile` insert into the dictionary, make `LoadFile` get from the dictionary, and claim to be secure because you send nothing to the `Datastore`.)

Note: this means if you require temporary symmetric keys, you will need to be able to save and restore them using only the one high entropy password the client is given.

You do not need to handle the case where two users interact with the server *concurrently*: you can assume only one user will interact with the server at any point in time. (That is, you do not need to worry about implementing locking - if one user has issued an API call, then no other user does so until that API call completes.)

## Part 2: Sharing and revocation

**Make sure you read instructions for Part 2 fully before starting.** It is likely that the design of your system for Part 2 will be significantly different from your previous solution if you did it separately, so consider this during your initial design.

**Question 2** *Sharing*
A file store becomes much more interesting when you can use it to share files with your collaborators. Implement the sharing functionality by implementing the methods `ShareFile()` and `Receive_Share()`.

When Alice wants to share a file with Bob, she will call `sharing, err := v.ShareFile("fubar", "bob")` to obtain a sharing message. Alice will then pass Bob `sharing` through an out-of-band channel(e.g. via email). You must not assume that this channel is secure. A man-in-the-middle might receive or modify the sharing message after Alice sends it but before Bob receives it.

After Alice passes `sharing` to Bob, if Bob wishes to accept the file, he will call `b.ReceiveFile("garp", "alice", sharing)`[1]. Bob should now be able to access Alice's file under the name `garp`. In other words, Alice accesses the file under the name `fubar`; Bob accesses it under the name `garp`.

`sharing` must be a string. During grading, we will pass `sharing` from one client to another on your behalf. Sharing a document must not require any other communication between the clients.

---

**Property 6 (Sharing)** *After `m = a.ShareFile(n1, "b"); b.ReceiveFile(n2, "a", m)`, user `b` MUST now have access to file `n1` under the name `n2`. Every user with whom this file has been shared (including the owner) MUST see any updates made to this file immediately. To user `b`, it MUST be as if this file was created by them: they MUST be able to read, modify, or re-share this file.*

This also changes Property 1 and Property 2 from above. A `LoadFile()` operation MUST return the last value written by anyone with access to the file (the owner, or anyone with whom the file was shared). Only those with access to the file should be able to read or modify it.

---

Sharing is tricky. Note that both filenames refer to the same underlying file, and any updates performed by anyone who has access to the file should be immediately visible to all other users with access to the file. By "update", we are including the case where a user invokes `AppendFile(f, v2)` on a file `f` that was previously uploaded and whose previous contents were `v1`. We are also including the case where a user overwrites a shared file.

---
[1]Bob populates the first two arguments himself after he receives what he believes is a valid `sharing` from Alice . Consequently, these two arguments cannot be tampered with by the man-in-the -middle attacker

Sharing should be transitive. If Alice shares a file with Bob who shares it with Carol, any changes to this file by any of the three should be visible to all three immediately. Sharing a file with someone who has already received it results in unspecified behavior (you may do whatever you choose). It is okay if the storage server learns which other users you have shared a file with.

We require a minimal amount of efficiency: assuming a file (of size $m$) is shared with n users, and Alice shares the file with a new user, you may perform a linear (in $O(n+m)$) number of either public or private key encryption operations. This is simple to achieve: any reasonable scheme should be at least this efficient. It is possible to do significantly better$-$and you are free to do so if you choose$-$but we will not evaluate you on this.

Your client may only keep state for performance reasons. Your implementation must work if your client is restarted in between every operation. Any state maintained on your client must be able to be reconstructed from data that exists on the server. Your clients may not directly communicate with each other.

Again, you do not need to worry about rollback attacks with sharing. For example, the server could rollback state and remove a client from receiving updates. You do not have to mitigate this. But remember, if you do notice any discrepancy during operation, you should return an error.

**Question 3** *Revocation*

Remote collaboration is a difficult thing, and, unfortunately, one of your collaborators has betrayed you, and you can no longer trust them. You realize that you need to revoke their access to your files. We will only require the revocation to revoke all other users access to the file, not just a specific user. This implementation requires much less effort than one which requires revocation of only a specific user.

> Implement the `RevokeFile()` method, which revokes all other users' access to a given file. You cant stop them from remembering whatever they've already learned or keeping a copy of anything they've previously downloaded, but you can stop them from learning any new information about updates to this file. Only the user who initially created the file may call `RevokeFile()`.
>
> **Property 7 (Revocation)** *If the original creator of the underlying file calls `RevokeFile(filename string)`, then afterwards all other users MUST NOT be able to observe new updates to `filename`, and anyone with whom other users shared this file MUST also be revoked. Except for knowing the previous contents of `filename`, to other users, it MUST be as if they never had received the file. It is undefined behavior if someone other than the original creator invokes `RevokeFile`, but it is acceptable for this to revoke access for everyone other than the user invoking `RevokeFile`.*

This single property has several hidden implications which may not be clear right away. Suppose that in the past, Alice granted Bob access to file $F$, and now Alice revokes Bobs access. Then we want all the following to be true subsequently:

1. Bob should not be able to update $F$ ,

2. Bob should not be able to read the updated contents of $F$ (for any updates that happen after Bobs access was revoked), and

3. If Bob shared the file with Carol, Carol should also not be able to read or update $F$.

4. Bob should not be able to regain access to $F$ by calling `Receive_File()` with Alices previous `msg`.

**Revocation must not require any communication between clients.**

You only need to implement functionality to revoke access from all other users.

If Alice shares a file with Bob, and then revokes Bobs access, it may still be possible (depending on the design of your system) for Bob to mount a denial of service (DoS) attack on Alices file (for example, by overwriting it with all 0s, or deleting `id`s), but Alice should never accept any changes Bob makes as valid. She should always either return an error, or return `None` (if Bob deleted her files).

Similar to Question 2, we will not grade you on efficiency. You may make a linear number of operations proportional to the size of the file, and the number of users who have received this file.

All the requirements from the previous parts are carried over to this part. Any other state stored must be only an optimization: it must be recoverable from state stored on the server.

As before, you do not need to worry about rollback attacks with revocation. For example, the server may rollback state and remove a client from receiving updates, or re-share with an old client. You do not have to mitigate this. But if you do notice any discrepancy during operation, you should return an error.

## Design Document

Write a clear, concise design document summarizing the design of your system and the choices you made. Explain the major design choices you made, including how data is stored on the server. The design should be written in a manner such that an average CS628A student could take it, re-implement your client, and achieve a grade similar to yours.

A well-written design receiving full points need not be longer than two pages and must not be longer than 3 pages.

> Before starting your design, studying skeleton codes and userlib API is a must and will significantly help you in designing your solution. Don't start implementing right away but definitely think about how will you do it using APIs provided to you.

## Design Review

In the second phase of the assignment, you will be asked to write a review of one of your classmates' designs. The purpose of the review is twofold.

At some point in your future lives as practising computing professionals, you may be asked to comment on the design, particularly the security-related aspects of the design, of some proposal to build some system. In such a scenario, reading a design document and evaluating whether the design meets its expected security objectives is an important skill. We're hoping that reviewing your classmates' designs will give you some practice with building up such skills.

Secondly, it is often much easier to spot flaws in the designs of others. Therefore, we're hoping your friends will be able spot design flaws in your design. This will make sure you don't waste your time writing code that is never going to pass the security tests.

The review document should not be more than **one page in length** and must describe a few of the biggest flaws (if any) in the design that you are reviewing.

## Implementation

The final phase will be the implementation phase where you will write actually the code. Make sure to have carefully thought through your design before starting the implementation. This assignment does not require writing much code, but it does require writing the correct code, and for this your design must be carefully thought through.

## Testing

We have provided a framework for testing for you to add more tests. One of the nice things about working with Go is that adding tests is extremely easy! You can simply add tests in

the `assn1_test.go` file and then run `go test` and all your new tests will run. You should use this to write tests to check both functionality and correctness under some attacks.

## Submission Summary

1. Design Document                                                                  (**25%**)

   • Due date: **16<sup>th</sup>Aug, 2019 | 11 : 55PM**

   • Submission: design.pdf

2. Design Review

   • Due date: **22<sup>nd</sup>Aug, 2019 | 11 : 55PM**

   • Submission: review.pdf

3. Implementation

   • Due date: **1<sup>st</sup>Sep, 2019 | 11 : 55PM**

   • Submission: assn1.go

   • You need to host your code on GitHub in private repository. Add sarkarbidya and supriya363 as collaborators. **You should make regular commits to the repository.**

---

More information about Design Review and Implementation phases will be provided in due time.
**You are not to start with Design Review or Implementation phases until you are instructed to do so.**

---

All tests will be run in a sandbox, and if your code is in any way malicious, we will notice.