Popa & Weaver
Fall 2016

CS 161
Computer Security

Project 2

Part 1 Due: October 4, 2016, 11:59PM

Part 2 Due: October 18, 2016, 11:59PM

Version 1.0: September 22, 2016

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 files on? What if you wanted to securely share and collaborate on files, even if the owner of the server is malicious?

In this project, you'll use the cryptographic tools we've taught you to build a file storage client that's secure and efficient despite storing all of your data on a malicious storage server.

Getting Started

For this project, you can work in teams of up to 2 people. We want you to get your hands dirty designing and implementing your system. There are two parts of the project, each with its own deadline.

We provide you a framework off of which to build for this project. All of your code should go in `client.py`. **Full documentation for the provided code is available online at** https://inst.eecs.berkeley.edu/~cs161/fa16/projects/2/docs/.

You must use Python 3 and the crypto API we provide you (which is based on and requires PyCrypto 2.6.1) for this project.

These requirements are already provided for you on all the following instructional machines:

- `hive{1-32}.cs.berkeley.edu` (330 Soda)
- `s271-{1-35}.cs.berkeley.edu` (271 Soda)
- `s273-{1-34}.cs.berkeley.edu` (273 Soda)
- `s275-{1-34}.cs.berkeley.edu` (275 Soda)
- `s277-{1-30}.cs.berkeley.edu` (277 Soda)

Note that `python` defaults to Python2 on most machines, including Hive. You can invoke Python3 using the `python3` command. If you aren't sure what version of Python you have, check the output of `python --version` and `python3 --version`.

See the Example Workflow section for a step-by-step guide.

You should strive to write clean, secure code. Follow general Python best practices[1].

Remember to follow all the steps in the submission instructions for each part of this project!

Secure File Store

Your task is to design an encrypted file store. This file store can be used to store your own files securely, or to share your 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 who you have shared access to the file with.

Note on security parameters. It is sufficient that these properties hold with very high probability (i.e., no better than brute forcing the key being used). We are not requiring that it is **impossible** for an adversary to do something malicious (e.g., decrypt or modify data) because that would not be possible.

You are given access to two servers:

1. A storage server, which is **untrusted**, where you will store your files. It has three methods:
   - `put(id, value)`, which stores `value` at `id`
   - `get(id)`, which returns the value stored at `id`

[1] "Hitchhiker's Guide to Python" has a good section on Python Style: http://docs.python-guide.org/en/latest/writing/style/. Another good resource is the CS61a Style Guide: http://cs61a.org/articles/style_guide.html.

- `delete(id)`, which deletes the value stored at `id`

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:
   - `get_public_key(username)`, which returns the public key for `username`
   - `put_public_key(username, pubkey)`, which sets the public key for your username

You are not to change the code for either server. 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.

The storage server has one namespace, so anything written by one user can be read or overwritten by any other user who knows the `id`. Clients interacting with the storage server must take care to ensure 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:

- `client.py`
  Where you will write your Client implementation. **Put all of your code in this file.**
- `base_client.py`
  Contains the base class you will subclass for your client.
- `insecure_client.py`
  Contains a baseline implementation which is functionally correct, but has no security built-in. It is described in the Reference Implementation Section. It is very simple (at only 50 lines).
- `crypto.py`
  Contains the provided cryptographic API you must use. Your client will be passed a Crypto object which you will use to access this API.
- `servers.py`
  Contains the (non-malicious) `StorageServer` and `PublicKeyServer` implementations. Your code will be graded using a malicious `StorageServer`.
- `util.py`
  Contains potentially useful utility functions, like object to string serialization function.

- `run_part1_tests.py`
  A Python script to run the provided tests for Part 1.
- `run_part2_tests.py`
  A Python script to run the provided tests for Part 2.

The documentation for all these files is available online at https://inst.eecs.berkeley.edu/~cs161/fa16/projects/2/docs/.

You must use our provided `crypto.py` API for all of your security-critical operations. Do not implement your own versions of symmetric (or asymmetric) key operations. This API has access to all the raw primitives we have taught you. Do not create a new instance of the Crypto object: use the one passed to you during initialization. It also has a secure random-bytes generator and other accessory methods. You should inspect it to see how it calls into PyCrypto, to understand what security properties you can expect out of this module. We have provided this API to you as a cleaner interface than the hundred possible methods in PyCrypto, and operates on strings for easier debugging. But, we expect that you will understand the consequences of how this code behaves. **You must NOT call into PyCrypto yourself**.

The skeleton provided in `client.py` calls `BaseClient.__init__()`, which sets up the client attributes, and calls the method `generate_public_key_pair()`. This method will automatically put the key to the public key server, and save a copy of your private key to the filesystem. This is the only persistent state that your client can use (you can assume that for the same username, a client will have the same public/private keys even if restarted). Your client code should call this method exactly once.

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. Any adversarial behavior will be seen as cheating.

The only exceptions your code may raise is an `IntegrityError`. Your code should handle all other exceptions.

Part 1: A simple, but secure client

This first part is due early! Focus first on Part 1 and do Part 2 only after you finished Part 1. Part 1 is designed to get you familiar with the API and crypto operations.

Part 1 is due on October 4, 2016, at 11:59PM.

Question 1 ***Simple Upload/Download*** **(25 points)**

Implement a file store with a secure (but possibly inefficient) upload/download interface. For Question 1, your task is to implement the methods `upload` and `download`.

The methods must ensure the following properties hold. See RFC 2119 for the definitions of MUST, MUST NOT, SHOULD, and MAY.

Property 1 (Benign Setting) *When not under attack by the storage server or another user,* ***`download(name)`*** *MUST return the* ***last*** *value stored at* ***`name`*** *by the current user, or* ***`None`*** *if no such file exists. It MUST NOT raise* ***`IntegrityError`*** *or any other error.*

Property 1 (Attack Setting) ***`download(name)`*** *MUST NOT ever return an* ***incorrect*** *value. A* ***`value`*** *(excluding* ***`None`****) is "incorrect" if it is* ***not*** *one of the values currently or previously stored at* ***`name`*** *by the current user.*

`download(name)` *MAY raise* ***`IntegrityError`*** ***or*** *return* ***`None`*** *if under any attack by the server or other users.* ***`download(name)`*** *MUST NOT raise any other errors.*

It SHOULD return ***`None`*** *only if it appears that no value for* ***`name`*** *exists for the user. It SHOULD raise* ***`IntegrityError`*** *only if the file has been tampered with.*

Property 2 ***`upload(name, value)`*** *MUST place the value* ***`value`*** *at* ***`name`*** *so that future* ***`download`****s for* ***`name`*** *return* ***`value`****.*

This function SHOULD return True. It MAY return False if the upload fails due to a malicious server.

Any person other than the owner of ***`name`*** *MUST NOT be able to learn even partial information about* ***`value`*** *or* ***`name`*** *with probability better than random guesses.*

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 this project. The contents of the file can be arbitrary: you must not make any assumptions there, but they are provided as Python unicode strings (for easier debugging). You may assume usernames consist solely of lower-case letters (`[a-z]+`).

The autograder does not look at the return value of the `upload` method.

You do not need to implement any capacity to share files between users (this is Part 2). We have provided an implementation of the storage server—do not change it.

Note that we require you protect the confidentiality and integrity of both the contents of the file you store and the name it is stored under. 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 may be able to overwrite a user's valid data, but any changes should be reported as an `IntegrityError`.

While integrity is an intuitively simple property (a user should only accept data they themself upload 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.) Your client is secure if after the following operations, the adversary cannot learn any bits of the secret message:

(a) the adversary has the client encrypt an arbitrary number of files with arbitrary names;

(b) the user uploads one secret file at a secret name;

(c) the adversary again has the client encrypt an arbitrary number of files with arbitrary names, or decrypt any file which is not the one secret file.

If, after these three steps, the adversary cannot learn any information about the secret file, or its name, then the file has been stored with confidentiality protection.

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 `get`, and the client would not be able to detect this.

Your client must not assume it can keep any state other than its RSA key pair. 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 `upload` insert into the dictionary, make `download` get from the dictionary, and claim to be secure because you send nothing to the `StorageServer`.)

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

Testing your submission: We have a set of tests which we will run on your code, for both functionality and security. In the provided framework, a file called `run_part1_tests.py` contains all the functionality tests we will run on your code, but only 1 security test. (We have many more security tests!) To run these tests, run `python3 run_part1_tests.py`. This will use your `Client` implementation from `client.py`. It will output Pass/Fail for each test we have and a one-sentence explanation of what the test does if it fails.[2]

We will not have any significantly new tests for functionality. If we add any new tests to the framework, we will announce this and release an updated set.

These tests are provided to make sure that anyone who attempts the project will get full points on functionality. This project is to test your ability to write secure code, not to implement a key-value store.

There will be no performance tests (although your code should definitely terminate!). Each test must complete in under one minute, but we expect tests to run in a few seconds.

Submission and Grading

You should submit your final version of `client.py` file (and only that file, plus optional feedback) by October 4, 2016, 11:59PM using `glookup` with `submit proj2-part1`.

Every time you submit your project, we will email you a report of your functionality test score and any crashes in your code. **You are responsible for verifying that your code runs against the autograder.**

We are having you submit this part early for your benefit: we plan to grade Part 1 shortly after the deadline. We will grade your last submission, and provide:

- the raw score based on functionality tests (almost identical to those you already have)
- the raw score based on a large set of security tests
- a one sentence summary of any security tests failed (so you have the chance to fix these issues for Part 2)

Your final score on this part of the project will be the minimum of the functionality score and security score. Each failed security test will lower the security score, weighted by the impact of the vulnerability.

We will **not** accept re-grade requests on the autograder results, except in cases where there was a bug in the autograder. If you feel this has occurred, please post a private question on Piazza to instructors and we will look at your code.

[2] We will make minor modifications to the functionality tests before we run them (e.g., by changing the names of keys and values) to ensure that solutions are not hard-coded to pass our tests.

Submission Summary

In summary, you must submit the following directory tree for Part 1:

```
client.py
feedback.txt (optional)
```

Part 2: Adding complex features

Make sure you read instructions for Part 2 fully before starting. You will probably want to make a design for everything all at once, but we split it into multiple questions so you'll know how we will grade you.

Part 2 is due on October 18, 2016, at 11:59PM.

Question 2 ***Sharing*** **(25 points)**

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 `share()` and `receive_share()`.

When Alice wants to share a file with Bob, she will call `msg = alice.share("bob", filename)` to obtain a sharing message. Alice will then pass Bob `msg` through an out-of-band channel (e.g., via email). You may 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 this, if Bob wishes to accept the file, he will call `bob.receive_share("alice", newfilename, msg)`.[3] Bob should now be able to access Alice's file under the name `newfilename`. In other words, Alice accesses the file under the name `filename`; Bob accesses it using the name `newfilename`.

You can only send one sharing message, `msg`, and it will be sent only once. `msg` must be a string in Python. During grading, we will pass this sharing message from one client to another on your behalf.

> **Property 3 (Sharing)** *After* `m = a.share("b", n1); b.receive_share("a", n2, m)`*, user* `b` *MUST now have access to file* `n1` *under the name* `n2`*. Every user who this file has been shared with (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 `download()` 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 writes performed by anyone who has access to the file should be immediately visible to all other users with access to the file. 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

[3] Bob populates the first two arguments himself. They are out of scope for an attacker.

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 $n + m$) 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. The server may rollback state and remove a client from receiving updates. However, this should only be possible if it is a complete reset to an old state.

Grading: The security tests for this question differ significantly from the previous tests. You must ensure you respect all sharing requirements, and that only valid users are able to read or edit a file. We will also test all functionality aspects, including all the tests from Part 1.

If your implementation relies on more out-of-band messages than a single return value from `share()`, you will get no credit for this question (or Question 4).

Question 3 ***Efficient Updates*** **(25 points)**

For this question, you must efficiently handle very large files—potentially multiple gigabytes long. Design and implement a solution for efficiently updating files that are already stored on the server.

This makes maintaining confidentiality and integrity much more difficult. Be aware that when the server is malicious, it can perform arbitrary actions at arbitrary points in time during your execution. For example, you cannot assume that two consecutive calls to `server.get(f)` will return the same value.

You do not need to handle the case where two valid users are interacting with the server simultaneously: 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.)

The requirements are exactly the same as Question 1, except that now we want your solution to be efficient when making a small update to a large file.

By efficiency, we are referring the amount of data that must be transferred over the network connection to the storage server. By "update", we are referring to the case where the user invokes `upload(f, v2)` on a file `f` that was previously uploaded and whose previous contents were `v1`. Your solution only needs to be efficient for updates that replace some bytes of the file in place.[4]

[4]Other kinds of updates (e.g., that insert data somewhere or delete data somewhere) do not have to be especially efficient.

Your client may store state (including, for example, the previous version of a file). But, this must only be an optimization: your client must still work correctly if it loses all of its state except for its public and private key. If your client loses all of its state, you are not required to be able to perform efficient updates, but it must still behave correctly.

If you store state, be careful to validate that your state is **current**. For example, suppose Alice creates a file and keeps a copy of the file locally. She then shares it with Bob, who makes changes to it. If Alice subsequently makes a change, her client must make sure her change does not overwrite Bob's change because her local copy of the file was out-of-date. You should be able to still perform efficient updates if another user updates a different part of the file.

Keep state in memory—you do not need to worry about serialization to disk. Let S be the total number of bytes of data a client has saved on the server. As long as you keep only $O(S)$ bytes of data in the client we guarantee our autograder tests will not cause your program to run out of memory.[5]

See Example Design for Efficient Updates section for one possible design of a solution to this question. You can base your solution on this design, but there are other ways as well.

Grading: We have provided you with a server which counts the total number of bytes you send and receive across the network. We have provided four test cases which we will use to determine your score.

One of these tests algorithmic performance for single byte updates when not changing the size of the file. You should strive to update in logarithmic size, however doing anything better than linear will award most of the points.

The other three tests report just the number of bytes transferred. We have not set thresholds for points here as we do not want to be overly harsh. We will set the curve after the project is due. However: expect that if you do better than the naive implementation by an algorithm factor you will receive "almost all" of the points.

As a guideline, for an efficient implementation, a reasonable amount of data transferred for test `z01_SimplePerformanceTest` is around 10 KB, and for `z03_SharingPerformanceTest` is around 100 KB.

You should not worry about $O(1)$ constants in updates until after your code is algorithmically faster (for example, do not worry about keys being hex encoded, or using JSON). These constants will award a few points, but much less than the algorithmic portion.

Question 4 ***Revocation*** **(25 points)**

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.

[5] This is solely for your benefit. In practice, it would be very bad to assume this—it could be possible to store many more bytes on the server than you have room for on your client. We allow you to assume this just so that you don't have to worry about memory management.

Implement the `revoke()` method, which allows a user to revoke someone else's access to a given file. You can't 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 `revoke()`.

Property 4 (Revocation) *After calling `revoke(otheruser, name)`, `otheruser` MUST not be able to observe new updates to `name`, and anyone with whom `otheruser` shared this file MUST also be revoked. Except for knowing the previous contents of `name`, to `otheruser`, it MUST be as if they never had received the file.*

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 Bob's 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 Bob's access was revoked), and
3. If Bob shared the file with Carol, Carol should also not be able to read or update F.

You may not send any sharing messages during revocation.

You only need to implement functionality to revoke access from direct children. If Alice shares a file with Bob, and Bob shares the file with Carol, you are not required to allow Alice to revoke Carol's access. It must work for Alice to revoke Bob's access.

If Alice shares a file with Bob, and Bob shares the file with Carol, you don't need to provide a way for Bob to revoke Carol's access. We will not test this situation: you only need to ensure that the original creator of the file can revoke others.

If Alice shares a file with Bob, and then revokes Bob's access, it may still be possible for Bob to mount a denial of service (DoS) attack on Alice's file (by overwriting it with all 0s, or deleting ids), but Alice should never accept any changes Bob makes as valid. She should always either raise an `IntegrityError`, 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. Recall that the only state which you can keep in the client is your public and private key. Any other state stored must be only an optimization: it must be recoverable from state stored on the server.

Again, you do not need to worry about rollback attacks with revocation. The server

may rollback state and remove a client from receiving updates, or re-share with an old client. However, this should only be possible if it is a complete reset to an old state.

Grading: It will be very difficult for you to receive any credit on this part if your implementation does not pass the functionality and security tests from Question 2. Since, in this case, functionality and security are tightly bound (revocation is a security behavior), we will not be providing you with difficult functionality tests. We have provided you with trivial tests, but you should definitely implement your own (although we will not ask for your tests). After submission, our autograder will apply some more difficult tests.

Question 5 ***Design Document*** **(25 points)**

Write a clear, concise design document to go along with your code. Your design document should be split into two sections. The first contains the design of your system, and the choices you made; the second contains an analysis of its security. Your design document should explain your complete solution, for Question 1 through 4.

In the first section, summarize the design of your system. Explain the major design choices you made, including how data is stored on the server. The design document should be written in a manner that an average 161 student could take it and re-implement most of your client and achieve a grade similar to yours. A well-written design document receiving full points need not be longer than two pages. You will lose points if your design document is excessively verbose.[6]

The second part of your design document is a security analysis. Present concrete attacks that you have come up with (which were not released with the Part 1 autograder) and how your design protects against each attack. You should not need more than one paragraph to explain how your implementation defends against each attack you present.

You may use as reference the Part 1 design document we provided to you for our reference solution. This is a design document which would receive full credit if we were grading it on Part 1 alone.

Grading: The design document is worth 25 points, split roughly equally between the two sections.

The first section is graded on your ability to explain your design to the reader effectively. Be sure to include the following in the document:

- For Q2, what state is stored on the server to allow for sharing, as well as the contents of the sharing message.
- For Q3, how you perform efficient updates and a short performance analysis.
- For Q4, what state is changed to revoke a file, and how you meet all the revocation requirements.

[6] If after writing your design document, you realize you have a 10-page document with 100 lines of code and think to yourself "My 162 GSI would be proud of this", you will be disappointed in your grade. That is not a design document. That is an implementation with comments.

The second section is graded on the attacks and defenses you present. You should have at least five attacks and your defenses to get full points. If you give more attacks, we will grade them in order and grade your best 5 (so place your strongest attacks first to make it easier for the readers). Do not give more than 10.

Question 6 ***Extra Credit*** **(10 points (bonus))**

After we run all of our test cases, we will manually review any submissions which have passed every test. Up to 10 points of extra credit on this project are available for submissions which resist targeted attack and have a well-written security analysis. We do not expect many (if any) students to receive this credit.

Submission and Grading

For Part 2, you should submit your final version of `client.py` with all of the functionality for Questions 1-4 by October 18, 2016, 11:59PM using `glookup` with `submit proj2-part2`. You will also submit your design document as `design.pdf`, plus optional feedback.

As in Part 1, we provide a set of functionality tests for Part 2, in `run_part2_tests.py`. You should also make sure that you still pass all the autograder tests we gave you for Part 1.

You can submit your project multiple times. For each submission you will receive an email with your autograder results for functionality tests.

We will grade the last submission in an identical manner to Part 1: Your score on Part 2 will be the minimum of your functionality and security scores. We will re-run all the functionality and security tests we ran for Part 1—you cannot get full credit if your code fails any of the previous functionality or security tests.

Submission Summary

In summary, you must submit the following directory tree:

```
client.py
design.pdf
feedback.txt (optional)
```

Appendix

Reference Implementation

We have written an inefficient, insecure implementation of a client. We have provided this to you in `insecure_client.py`. This code is also an example of how to extend the `BaseClient` class. This implementation provides all the functionality requirements of this project, but has no security properties at all.[7]

This client gives each user their own "namespace" within the master server by concatenating the username, a slash, and then the filename and using that as the `id` for the storage server.

The client works by maintaining two types of objects on the server storage: pointers and data. A data object has the contents of a file. A pointer acts as a reference to the file. (If you've taken operating systems, you can think of pointers as symlinks.) When a user updates a file that is a pointer, she follows the pointers until a data file is reached, and then updates the corresponding data file. Sharing works by providing the other user with a pointer to the file, and revocation removes the pointer. This satisfies the revocation properties that sub-children are also revoked.

Example Workflow

An example workflow for developing on your personal machine is as follows:

- Copy the framework code into a folder named `project2`.
- Sync changes to your class account using `scp`:

  ```
  scp -r project2/ cs161-xxx@hiveXX.cs.berkeley.edu:~/project2
  ```

  This will copy the `project2` folder and its contents to your home directory.
- SSH into a Hive machine with your class account.
- Do all Python console work using `python3` or `ipython3` in your SSH session.
- Run all Python code using python3, e.g. `python3 client.py`.

You can also do all of this while seated at a machine in the instructional labs, or inside an SSH session using vim or emacs.

While not officially supported by course staff, it is also possible to set up Python3 and install PyCrypto on your own machine. You should double check your code by re-running the functionality tests against your code on Hive. **Remember to follow all the steps in the submission instructions for each part of this project!**

[7] Submitting this client will earn you 0 points on the project.

Example Design for Efficient Updates

The following is a snippet of a design document. You can base your solution for Question 3 on this design, but there are other ways as well.

To perform efficient updates, we use a tree-based approach. When a file is initially uploaded, we create a binary tree. Internal nodes of the tree contain several pieces of information (all stored with authenticated encryption):

1. A pointer to the left and right sub-trees, or None to indicate a leaf.
2. A cryptographic MAC of all the data at this node and below.
3. A cryptographic hash of all the data at this node and below.
4. The length of the data stored at this node and below.

We describe our update procedure recursively. To update a file we compare the hash of the new file against the hash stored at the root node on the server. If the hashes are equal, then we have no more work to do. If they differ, then we split our file into two pieces according to the lengths of the sub-trees stored on the server. We then recursively call update on the left and right child. When we reach a leaf node, if the hash does not match, we replace the leaf with the new data. Our leaf nodes are 128 bytes. Then, we recompute the hash for each node as the hash of the concatenation of the hash of the left child and the right child. If the upload would result in a larger file than simply re-uploading the entire file, we do that instead.