(File) System Support Solution (S^3)

joannas, msands, brishima

May 11, 2019

Abstract

Our primary goal for this project is to create a system that maintains the integrity of files that are hosted on the cloud by untrusted servers. We constructed a file system that can be verified from a single 32 byte hash. This is done by creating a B+ Tree that also functions as a Merkle Tree to track files while keeping a low tree depth. To show that this system can be used with different cloud storage providers, our system has an integration with Dropbox. Our implementation is available at `https://github.com/joannasands/SSS`.

1 Introduction

Users now trust online file hosting services to hold a large array of important files. Are there ways a user can be sure that the file they are downloading from the host is the file they uploaded? Merkle Trees can be used to verify if a given leaf node (the file about to be downloaded) is part of the tree and they can be updated if the set of leaf nodes changes.

For this project we wanted to implement a file store on top of Dropbox where Dropbox does not have to be trusted to return the right file. We augmented Dropbox with a Merkle tree to keep track of file and intermediate node hashes so that only the Merkle root node has to be kept persistent on the local client. The system can detect when files from the untrusted server have been tampered with but will not be able to recover the original files from an adversarial server. This project targets integrity but not the confidentiality and availability components of security.

2 Background

2.1 Merkle Trees

Merkle Trees, also known as Hash Trees, are defined as trees where every leaf node is labelled with the hash of its data block and every non-leaf node is labelled with the hash of it's children's labels.

Because each node is based off of its children, we can also label each node with the hashes of its children. When we want to determine if a leaf node is in a specific Merkle Tree, we can start at the leaf's "parent" and then recompute the hash of each level of the tree. Once the hash of the root is recomputed, it can be compared to a previously computed hash of the root stored on the user's computer.

2.2 B+ Trees

B+ trees are balanced search trees that achieve low depth due to their high branching factor. They are commonly used in filesystems and database systems. Data is stored at the leaf nodes of the tree. Internal nodes can have between $\lfloor b/2 \rfloor$ and b children, where b is the branching factor. They hold pointers to their children and have keys which indicate what values split the subtrees. Insertion and deletes are performed in a way which splits and recombines nodes to maintain the invariant that each node (except possibly the root) has between $b/2$ and b children.

All operations take $O(b \log_b n)$ time, and more importantly, only need to access $O(\log_b n)$ nodes.

2.3 Git

Git implements version control by keeping track of the SHA-1 hashes of files and subtrees of files. Internally, git uses a database of data stored as files Files are blobs which can be referenced by their hashes. Directories are represented by object trees which contain object blob hashes and object trees along with names. For example, Fig. 1 could be the representation of a tree object for a directory that contains a directory `bak/` and files `new.txt` and `test.txt`.

```
$ git cat-file -p 3c4e9cd789d88d8d89c1073707c3585e41b0e614
040000 tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579      bak
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a      test.txt
```

Figure 1: Representation of a tree object in git.

This is exactly the description of a Merkle tree, where the intermediate nodes are augmented with metadata like the names of files and permission bits. The names allow for file lookup from a path.

However, the intermediate objects of a git tree can become unbounded as each directory is represented by a single object node. Tree objects store the hashes and names for each directory/file within the corresponding directory. Additionally, the distance of each file from the root hash is exactly the number of directories in the path to the file.

2.4 Blockchains

Merkle trees are commonly used in blockchains to decrease block size while ensuring consistency. In Bitcoin, blocks aiming to be added to the chain contain a Merkle tree of transactions to be added to the chain. Once the block is verified, only the Merkle root is actually added to the chain. This helps keep the overall chain relatively small. Despite the small size of the header data, the existence of all transactions can still be verified as needed by checking if the generated hash matches the root that is stored on the block chain. This is especially important as agents are untrusted in the blockchain pool. Similarly, Ethereum's implementation of account state uses a Merkle Root to encode all of the information about a given account.

3 System Model

In this project, we built a single user Dropbox client which builds a Merkle tree as metadata for the file system. The client is able to verify that files downloaded are the original files that were uploaded to Dropbox. Because we use a Merkle tree, the client only keeps the root hash in persistent memory, everything else is stored in the cloud. Like git, files are identified by their hashes. Merkle tree nodes are augmented with the hashes and path information of their children so that files can be looked up from the Merkle root. Files can be verified by checking $O(depth)$ nodes on the server. Similarly, files can be added, deleted, or modified by changing $O(depth)$ nodes.

3.1 Threat Model

In this project we allow users to verify that a file stored in a cloud system is the file they originally uploaded to said system. This is done under the assumption that foreign system is not trustworthy, while the local copy of the Merkle root is. This will detect when changes are made, however we do not aim to correct these changes.

3.2 Dropbox

We use Dropbox as a key-value store for files and Merkle tree data. This is done through the Dropbox python API, which allows for the syncing of files when given an API Key. In this system, we assume a user has made a Dropbox App for their own system and can provide an API key which matches their own account permissions. This integration with Dropbox allows us to check downloaded files for changes before opening.

4 Implementation

4.1 Data Structures

4.1.1 Nodes

The all metadata of the system is stored in the form of a B+ tree, composed of both internal and leaf nodes. The default branching factor for the system is 100, but this can be changed to depending on ones desired system properties.

Each node contains a list of headers describing its children. The headers are ordered by the key upperbound which is used for searching the tree.

4.1.2 Headers

Headers are data structures to hold information about a node. Each header stores 3 attributes, the subtree hash, the key upperbound, and its node type. The subtree hash is the hash of the serialization of the node that the header is describing. The key upperbound is the largest possible value of a hash of a path that would be stored in the node's subtree. A header can have one of three possible node types: directory, file, or internal.

4.2 Data Representation

Each node is hashed and stored using a serialization of the node. Each node stores a single attribute describing a list of headers that represent its children so their serialization is the concatenation of the serialization of each its children. The headers are serialized by converting each of the header's attributes (subtree hash, key upperbound, and node type) to bytes and concatenating them.

All data is keyed by its SHA-256 hash. On Dropbox, all data is stored under the root with the filename set to the hex encoding of the hash of the data. In this way, Dropbox is used as a key-value store.

The client keeps a cache while it is running. Nodes are looked up by their hash. If a hash isn't in the client cache, the file is accessed from Dropbox, and then the client keeps puts the node in the cache so that it does not have to access Dropbox to find the same node again later.

4.3 Directories

Because the data storage is actually flat in structure, we augment our system with data files listing the contents of each directory, called directory files. Directory files are stored the same way as normal data files, but are not directly removable, downloadable or editable by the end user. Each directory file is inserted into the tree at the time of the directories creation. Directories are initialized only when a file is added to said directory or to one of its children. Empty directories can not be created, but directories can be emptied by removing their children.

4.4 Operations

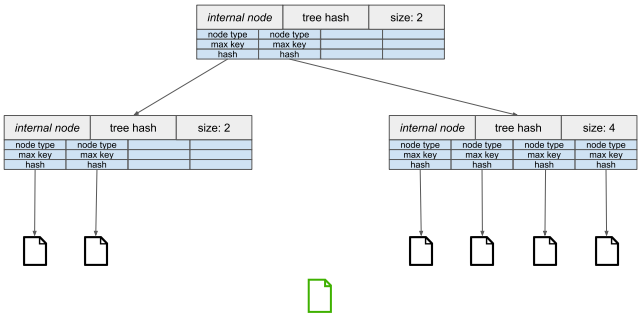
4.4.1 Edit

The edit operation allows a user to change the data of file that is already present in the system. The system first saves the new data for the file as its own file in the cloud, keyed by the hash of the data. It then locates the file original file in the B+ tree using the hash of the path as the key. A new copy of the parent node is made, this time with the pointer to the new version of the file. Its parent is then updated to point to this new parent node and this continues until a new root node has been created. The new root is then saved to client's local file system.

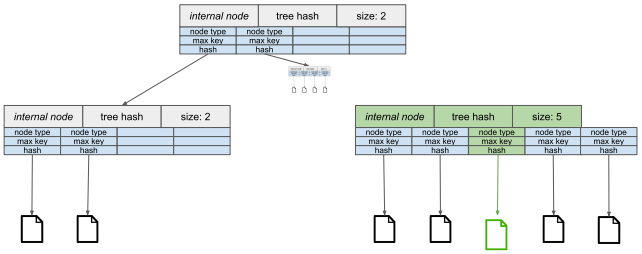
4.4.2 Add

Files can be added to the system with an add operation. When a user wishes to upload a file to the cloud, they can specify for directory location of the new file as well as the source of the data for the new file.

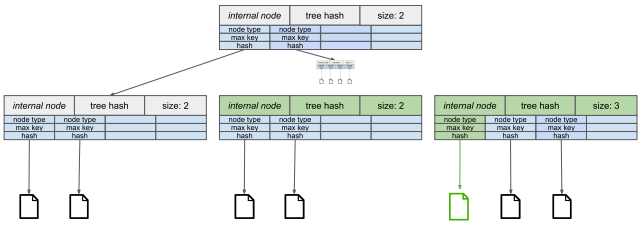
The operation can be broken down into two steps. In the first step, the system checks to make sure the parent directory of the file exists. If it doesn't, the system recursively calls add to add directories until the the parent directory exists. Then the system edits the directory listing to include the new file path. In the second step, the system updates the B+ tree to include the new file. The steps for adding a directory and adding a file look identical.



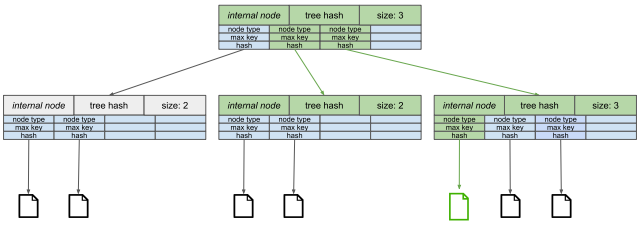
(a) New file to be inserted into tree.



(b) Find the insertion point and create a copy of the node with the file inserted.



(c) If it breaks the branching factor invariant, split the node.



(d) Create a new parent node with the header replaced by the new nodes.

Figure 2: Add Operation

Updating the B+ tree to store the new file is very similar to simply editing the tree. The insertion point is found and a new node is created with the new header inserted, and then new nodes are created for its ancestors iteratively to update their list of headers. However, because we are adding data we must be careful to make sure the tree stays balanced. If at any point is this process, a new parent node has more than b children, it splits into two nodes to maintain the branching factor invariant and then updates continue as normal.

This is illustrated in Fig. 3.

4.4.3 Remove

Files can also be removed with the remove operation. First the directory listing for the file's parent directory is edited to not include the removed file. If the file was the only file in the directory, the directory is removed. Then the file is removed from the tree. Removing a directory and removing a file look the same.

The B+ tree is searched for the hash of the path of the file to be removed. A new parent node is created with the header for the file removed. If the removed header causes the node to have fewer than $\lfloor b/2 \rfloor$ children, the system looks for a sibling node (which must exist unless the grandparent node is the root). If the sibling node has more than the minimum number of children, the children are redistributed evenly between the two nodes. Otherwise, together they have at most b children and are combined into a single node.

Then new nodes are created iteratively for all of the ancestors with new headers for the updated nodes. If siblings were combined, the loss of a child can cause ancestor nodes to merge with their siblings as well. If the root gets reduced to having a single child, the root is abandoned and the single child is made the new root.

4.4.4 Download

The download operation verifies that the hash of the file and the current root hash matches the root hash stored on the user's computer. First, it verifies that the stored hashes of each node match the hash of the data of the node while creating the path from the leaf node to root node. Once it verify the data of the file node against its hash, it checks that the node type of the leaf node is a file. If so, it writes the data to the file location specified. Otherwise, it throws a key error.

4.4.5 Verify

The verify operation allows a user to compare the file currently on their computer to the file stored in the cloud. In practice, we download the file and directly compare its contents to the data.

4.4.6 LS

The ls operation returns a list of all files in the directory specified by a user. When adding a file to the tree, a directory node is created for each directory in the file's path. The data of each of these directory nodes is a newline-separated list of the files in the directory. In this regard, directories are treated the same as files except they are marked as directories instead.

When the ls operation is called, the type of the node at the path given is checked. If the node is a directory node, then the data of the node is returned.

```
Joannas-MacBook-Pro-2339:SSS joannasands$ python cli.py repl /
> add /hi/hello -file_path=test.txt
> add /hi/hi -file_path=test2.txt
> download /hi/hello -file_path=/dev/stdout
hello world
```

(a) Using the CLI's repl shell to add and download files

```
> download /hi/hello -file_path=/dev/stdout
ValueError: Hash of data does not match hash key
> download /hi/hi -file_path=/dev/stdout
Never gonna give you up never gonna let you down never turn around
and hurt you never gonna make you cry never gonna say goodbye never
 gonna tell a lie and desert you
>
```

(b) Attempting to download files after the Dropbox file for `test.txt` had been tampered with by a "malicious server"

Figure 3: CLI Examples

4.5 Command Line Interface

The command line interface allows a user to call the previously described operations within their own file system by giving a command, a file storage system path, and a user system path. The CLI supports 2 kinds of file systems, a Dropbox application or a folder on the user's system (including on an external drive). In order to start the CLI, a user must chose a mode. If Dropbox, the user give their Dropbox API key. Otherwise, they have the option to include the path that they intend to store their files at on the disc. The final optional argument takes in the location that the root hash should be stored or the location that the root hash if the file system has already been created. Operations that edit the Merkle Tree append the new root hash to the file specified to contain the root hash.

The CLI can be called in two ways. The first is to call `python cli.py` directly for each command with all of the optional arguments. The second is to call `python cli.py repl` with the optional arguments to enter the read-eval-print-loop shell.

5 Conclusion

In this project, we were able to implement what we set out to do and created a system for ensuring file integrity when storing data in an untrusted cloud server. However, there is more work that can be done to make this a more usable system. There is a large time delay on the addition and removal of files due to the time it takes to upload multiple files to Dropbox.

References

[1] Scott Chacon and Ben Straub. *Pro Git*. Apress, Berkely, CA, USA, 2nd edition, 2014.

[2] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, CRYPTO '87, pages 369–378, London, UK, UK, 1988. Springer-Verlag.

[3] D. Vujičić, D. Jagodić, and S. Ranić. Blockchain technology, bitcoin, and ethereum: A brief overview. In *2018 17th International Symposium INFOTEH-JAHORINA (INFOTEH)*, pages 1–6, March 2018.

[4] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151:1–32, 2014.