Merkle Hash Trees for Distributed Audit Logs

Subject proposed by Karthikeyan Bhargavan
Karthikeyan.Bhargavan@inria.fr

April 7, 2015

Modern distributed systems spread their databases across a large number of partially untrusted hosts, to improve their availability to users located across the Internet. In many of these scenarios, the users accessing the data do not fully trust the hosts from which they download data, but they still desire strong integrity guarantees, that is, they want to be able to detect tampering.

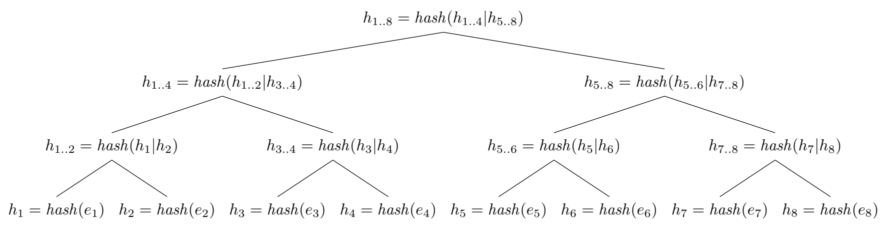
In this project, we are concerned with a particular kind of distributed database called an *audit log*. Audit logs are used to log security events in a distributed system, such as the granting of access to a sensitive resource. For example, when a hacker breaks into an enterprise network, his activities will be logged on machines across the network. To escape detection, the hacker may try to erase or tamper with the log, and hence the integrity of this log is security-critical.

More generally, audit logs are used to maintain a shared history of events between distributed parties, where this history can only be appended to, but not modified. Popular examples of this pattern include the Bitcoin block chain[1] and the Certificate Transparency (CT) global certificate log.[2]

How can a user verify the integrity of a part of a log that it has retrieved from an untrusted server? One solution is for some trusted servers to publish a *cryptographically strong hash* of the full log. So a user can download the log from different servers and then verify that its hash matches the expected value (retrieved from a trusted server). If the hash function is collision resistant, no malicious host can tamper with any part of the log without it being detected.

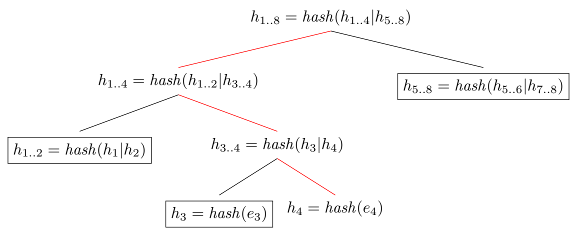
Audit logs can get very large, and waiting to download the full log to verify the integrity of the few events a user may be interested in would consume a lot of redundant time and bandwidth. Hence, distributed auditing systems use an efficient data structure called *Merkle trees* to compute the hash of a log with n events such that the contents of an individual event can be verified in $O(log\ n)$ time. The Merkle hash tree for a log with 8 events $\{e_1, ..e_8\}$ is depicted below. In general, this construction works similarly for any database with n records:

[1] `http://bitcoin.org`
[2] `http://certificate-transparency.org`


$h_{1..8} = hash(h_{1..4}|h_{5..8})$
$h_{1..4} = hash(h_{1..2}|h_{3..4})$
$h_{5..8} = hash(h_{5..6}|h_{7..8})$
$h_{1..2} = hash(h_1|h_2)$
$h_{3..4} = hash(h_3|h_4)$
$h_{5..6} = hash(h_5|h_6)$
$h_{7..8} = hash(h_7|h_8)$
$h_1 = hash(e_1)$
$h_2 = hash(e_2)$
$h_3 = hash(e_3)$
$h_4 = hash(e_4)$
$h_5 = hash(e_5)$
$h_6 = hash(e_6)$
$h_7 = hash(e_7)$
$h_8 = hash(e_8)$

The leaves are the hashes of each individual event $e_1, \ldots, e_8$. These hashes are incrementally combined with the hashes of their neighbors as we travel up the tree. The *root hash* $h_{1..8}$ is effectively a hash of the whole database, computed using the hashes of the left and right subtrees.

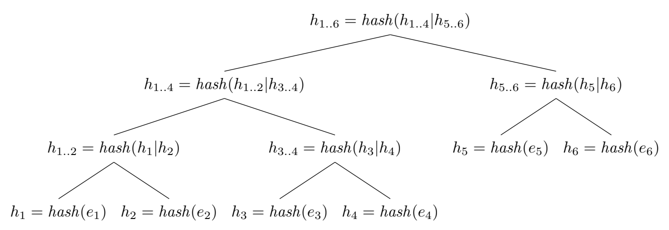
Audit Paths If it is used only to compute a hash of the full log, the Merkle tree is not particularly efficient, since it requires roughly $2n$ hashes to be computed, whereas computing a sequential hash over the database only requires n hashes (or more accurately, compressions) to be applied. The true advantage of the Merkle tree is when we want to prove that the i-th record has not been tampered with. Suppose a user has obtained the root hash $h_{1..8}$ from a trusted server and then retrieved e_4 from an untrusted host. To verify the integrity of e_4, the user needs the untrusted host to also send it the hashes $[h_3, h_{1..2}, h_{5..8}]$, so that it can reconstruct the path leading from e_4 to the root hash, as depicted below:


$h_{1..8} = hash(h_{1..4}|h_{5..8})$
$h_{1..4} = hash(h_{1..2}|h_{3..4})$
$h_{5..8} = hash(h_{5..6}|h_{7..8})$
$h_{1..2} = hash(h_1|h_2)$
$h_{3..4} = hash(h_3|h_4)$
$h_3 = hash(e_3)$
$h_4 = hash(e_4)$

Hence, verifying that any given record r is in fact the 4th event in the log only requires transmitting and computing 3 hashes, not the full tree. Verifying any event in a log of n events only takes *log* n hashes. (Can you prove it?) This sequence of hashes is called an *audit path*

Consistency Proofs Logs grow over time as new events occur. However, they are *append-only*, that is, events may not be modified, deleted, or inserted in the middle. Once an event has occured, its presence in the log must be preserved.

Suppose a user already has the root hash of the log after n events, and now the log has been extended to $m > n$ events. How can we prove to the user that the new log has only appended events to the old log, without the user needing to check the full Merkle trees? Consider the Merkle tree below corresponding to an earlier version of the log with $n = 6$ events.


$h_{1..6} = hash(h_{1..4}|h_{5..6})$
$h_{1..4} = hash(h_{1..2}|h_{3..4})$
$h_{5..6} = hash(h_5|h_6)$
$h_{1..2} = hash(h_1|h_2)$
$h_{3..4} = hash(h_3|h_4)$
$h_5 = hash(e_5)$
$h_6 = hash(e_6)$
$h_1 = hash(e_1)$
$h_2 = hash(e_2)$
$h_3 = hash(e_3)$
$h_4 = hash(e_4)$

To prove that the root hash $h_{1..8}$ for the log with $m = 8$ elements strictly extends the log with $n = 6$ elements, it is enough to provide the user with the three hashes $(h_{1..4}, h_{5..6}, h_{7..8})$. Using these hashes, the user can reconstruct both root hashes $h_{1..6}$ and $h_{1..8}$ and be assured that the second corresponds to a tree that extends the first. This sequence of hashes is called a *consistency proof*.

Programming Tasks

Our goal is to program an audit log server and an auditor who can check the log on behalf of the user. We treat logs as simple text files where each line corresponds to one event. The tasks below are written with Java in mind, but the project could just as easily be coded in OCaml. The concrete design presented below is taken from Certificate Transparency - RFC 6962.[3]

Merkle Hash Tree Define a Java class that represents Merkle Trees. Each tree object represents one node in the tree. It has fields containing the hash of the current tree, pointers to the left and right subtrees, and two fields denoting the beginning and ending index of the range of the log they cover (e.g. $h_{5..8}$ covers records 5 to 8).

The constructor for a leaf takes a string containing a single event of the log, converts it to bytes, prepends it by the single byte 0x00 and computes its hash.

h_i = SHA−256(0x00 | utf8(e_i))

The constructor for an internal node takes two Merkle trees for the left and right subtrees. It concatenates the hashes of its two subtrees, prepends them with 0x01, and hashes the result.

$h_{1..8}$ = SHA−256(0x01 | $h_{1..4}$ | $h_{5..8}$)

The use of different byte-tags in the two hashes offers protection against some kinds of hash collisions.

We can use the function SHA256 from java.security.MessageDigest to compute each individual hash; for example, for a byte array (leaf) the hash can be computed as:

```
MessageDigest digest = MessageDigest.getInstance(``SHA-256'');
byte[] hash = digest.digest(leaf);
```

[3]https://tools.ietf.org/html/rfc6962

To convert a string (text) to a byte array, we can use text.getBytes("UTF−8"). Write a function that takes an input text file and computes its Merkle tree (treating each line as a new event). Test your code on large (>1GB) text files.

Log Server Implement a Log Server class that uses a Merkle tree to implement an event log. Define a constructor that reads a full log from a text file to construct the tree. Define a method that returns the current root hash. Define methods to append events to the log one at a time, or as a batch of events appended at once. Compare and discuss the complexity of these operations.

Implement a method genPath that given the current Merkle tree of size n and any index $i <= n$ returns the *audit path* for the i-th event, that is, the hashes needed to verify e_i. For example, in the Merkle tree for 1..8 above, genPath(4) for e_4 would return the array of hashes $[h_3, h_{1..2}, h_{5..8}]$.

Implement a method genProof that given the current Merkle tree of size n and any previous tree size $m <= n$ returns the *consistency proof* that the current tree extends the previous tree. For example, in the Merkle tree for 1..8 above, genPoof(6) would return the array of hashes $[h_{7..8}, h_{5..6}, h_{1..4}]$.

Auditor Implement an Auditor class that calls the Log Server and verifies its behavior on various inputs. Define a constructor that takes an instance of the Log Server as input and retrieves the current size and root hash for the log.

Implement a method isMember that takes an event (as text) and verifies that it exists in the current log by calling genPath to obtain an audit path and verifying the hashes on the audit path: the path should begin with a hash of the event and end with the root hash, and all hashes on the path should be correctly computed. Note that this method does *not* use any knowledge of the full Merkle tree beyond its size and root hash. Test your code by verifying membership for various events.

Implement a method isConsistent that retrieves a new size and root hash from the Log Server and verifies that the new log is consistent with the previous one. If the log has changed, the method calls genProof to get a consistency proof and verifies it using only the previous root hash and size and the current root hash and size.

Evaluation The project will be evaluated on the basis of the clean design and implementation, the careful analysis of the complexity of various methods, and on the quality of tests that have been used to verify the solution.

Both the report and the presentation should answer questions about the design choices made by the student. For example: What is the complexity of appending events, audit path generation and verification? How much memory or time does it take to verify k records? Can you experimentally confirm the time taken for various operations? Does this depend on the type of file, on the size of each event? At what size of the database does it become more efficient to use Merkle trees as opposed to a flat list of hashes?

For extra credit, can you deploy your Auditor and Log Server as independent processes communicating over the network? Can you use your Auditor to check that some public CT log is consistent? You can get the latest root hash in JSON format for the Log Server `ct.googleapis.com/aviator` from `https://ct.googleapis.com/aviator/ct/v1/get-sth`. You can get the consistency proof betwen the hash trees of size m and n by accessing `https://ct.googleapis.com/aviator/ct/v1/get-sth-consistency?first=m&second=n`.

For other CT Log Servers, see `http://www.certificate-transparency.org/known-logs`. For other CT JSON APIs, see `https://tools.ietf.org/html/rfc6962#section-4`.