Merkle trees and I.T. clouds

Stefan Genchev

June 2018

Abstract

This document includes basic information about the structure and function of Merkle trees as an efficient way to certify the integrity of data records. The generation of Merkle trees and Merkle audit and consistency proofs is described in detail. Furthermore, a resource-efficient method for using Merkle trees and a sample server implementation of a system log based on a Merkle tree are introduced.

Contents

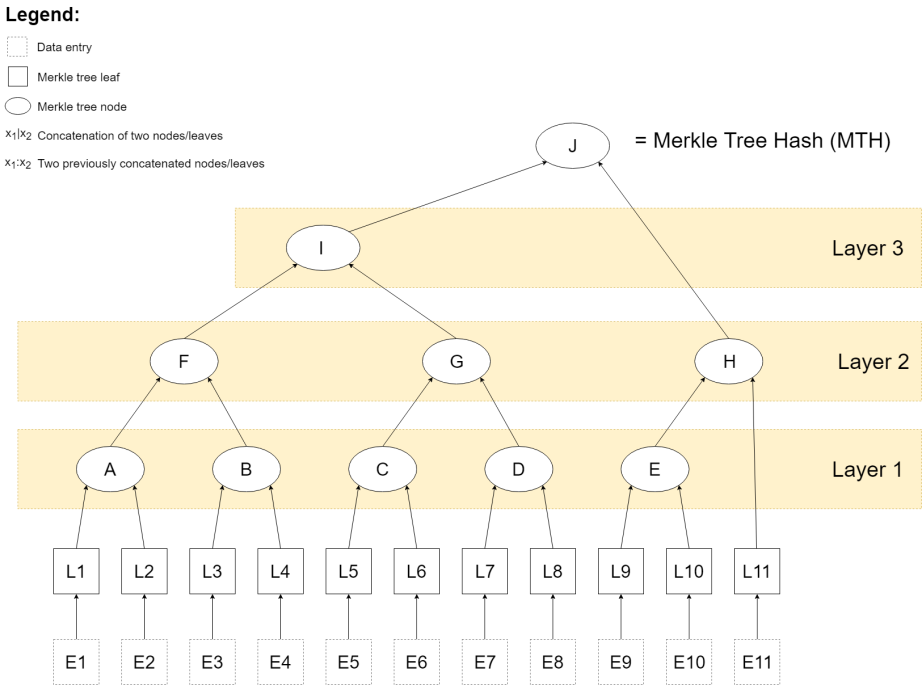
1 Merkle trees

1.1 Definitions

1. $HASH(x)$ - denotes the operation of generating a cryptographic hash over the data x using a specific algorithm
2. $x_1|x_2$ - denotes the concatenation of x_1 and x_2

1.2 Basics

Figure 1: A sample Merkle tree with eleven entries



A Merkle tree consists of leaves, nodes and a tree head (hash). The input to the Merkle tree is an ordered list of binary data entries (E 1-11).

These entries are hashed in order to form the Merkle tree leaves.

$$HASH(E1) = L1, HASH(E2) = L2, ..., HASH(E11) = L11$$

The leaves (the binary data entries that were hashed in the previous step) are now hashed pairwise. Each hash forms a Merkle tree node. Together, all hashes of the Merkle tree leaves form the first layer of Merkle tree nodes. If the number of leaves is odd, the last leaf is not hashed for a second time, as it does not have a partner leaf to be hashed pairwise with.

$$HASH(E1|E2) = A, HASH(E2|E3) = B, ..., HASH(E9|10) = E$$

The Merkle tree nodes from the (first) layer - generated in the previous step - are once again to be hashed pairwise, in this way forming new hashes that then form a new (second) layer of nodes. This cycle continues until only two nodes remain. By hashing them, one receives the Merkle Tree Hash (MTH) of the given tree. The last nodes of odd-sized layers of nodes are hashed cross-layer.

1.3 Important properties of Merkle trees

Merkle trees provide a fast and efficient way to prove that:

1. all data entries have been consistently appended to the database.
2. a particular data entry has been appended to the log.

1.3.1 Parameter k

The parameter k is the largest power of two smaller than n (i.e.,$k<n \leq 2k$). In the context of Merkle trees, by calculating k, one receives the largest number of leaves that can be used to build a Merkle tree with entirely even-sized node layers in an original, bigger Merkle tree with odd- or even-sized node layers.

1.3.2 Generating a Merkle Tree Hash

The input to the Merkle Tree Hash (MTH) generation method *GenerateMTH* is an ordered list of data entries D_n of size n. The output is a single hash value.

GenerateMTH(D_n d)

1. Define n as the size of d
2. The MTH of an empty list (n = 0) is the hash of an empty string.

$$Return : HASH().$$

3. The MTH of a list with one entry (n = 1) is defined as a leaf hash.

$$Return : HASH(0x00|d(0)).$$

4. The MTH of a list with more than one entries (n ¿ 1) is generated as follows:

   (a) Calculate the parameter k for n

   (b) Create a new list of data entries $D^k_{0,k} = [d(0), d(1), d(2), ..., d(k-1)]$ of size k

   (c) Create a new list of data entries $D^n_{k,n-k} = [d(k), d(k+1), d(k+2), d(n-1)]$ of size $n-k$

   (d) Using a recursion, the MTH is calculated as follows:

$$Return : HASH(0x01|GenerateMTH(D^k_{0,k})|GenerateMTH(D^n_{k,n-k}))$$

1.3.3 Explaining the recursion during the MTH generation

In a recursion, the abovedescribed method for generating Merkle Tree Hashes (MTH) splits the original Merkle tree into smaller ones and calculates their MTHs. This process of generating smaller and smaller trees continues until the tree leaves are reached. The resulting MTHs of the subtrees are sent back on the chain in the opposite direction of the tree separation in order to form the MTH of the original tree.

First, the parameter k must be calculated based on n. Each tree, beginning with the original one, is then split in two new trees with leaves from 1 to k and, respectively, leaves from k to n. The same operation of recalculating k and splitting the tree in two, smaller trees is performed for these two newly generated trees and then for the resulting from this operation two trees and so on.

When the leaves of the original tree are reached, their leaf hash is calculated. The hash of the concatenated leaf hashes of two entries form the MTH of the smallest trees. The hash of the concatenated MTHs of two related trees, split by k, form the MTH of their parent tree. This calculation of the MTHs from smaller trees to bigger trees continues until only two subtrees exist. The hash of their concatenated MTHs form the MTH of the original tree.

The following figures illustrate this behavior. The orange boxes represent the trees with leaves from 1 to k. The red boxes represent the trees with leaves from k to n.

Figure 2: This is the original Merkle tree. Its MTH (**J**) is calculated by concatenating the MTHs of the two subtrees (I and H) with leaves from 1 to $k = 8$ [$L1 \rightarrow L8$] and with leaves from $k = 8$ to $n = 11$ [$L8 \rightarrow L11$].

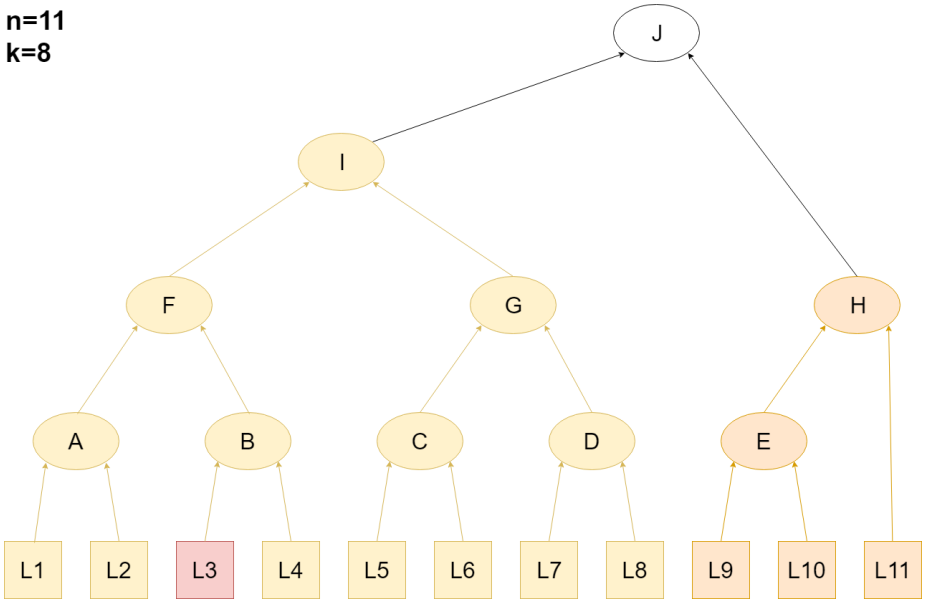


Figure 3: This is the first subtree of the original tree $n = 11$. Its MTH (**I**) is calculated by concatenating the MTHs of its two subtrees (F and G) with leaves from 1 to $k = 4$ [$L1 \rightarrow L4$] and with leaves from $k = 4$ to $n = 8$ [$L5 \rightarrow L8$]. This subtree's MTH is used to calculate its parent tree's MTH (J = **I** + H).

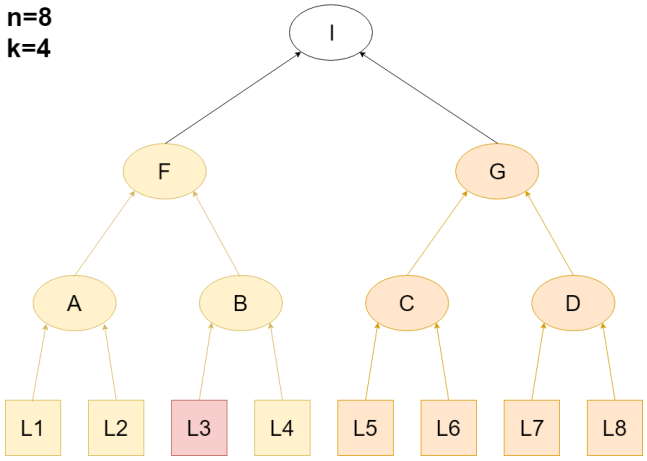


Figure 4: This is the other subtree of the original tree $n = 11$. Its MTH (**H**) is calculated by concatenating the MTH of its one subtree (E) with leaves from 1 to $k = 2$ [$L9 \rightarrow L10$] and with the leaf hash of 3 [L11]. This subtree's MTH is used to calculate its parent tree's MTH (J = I + **H**).

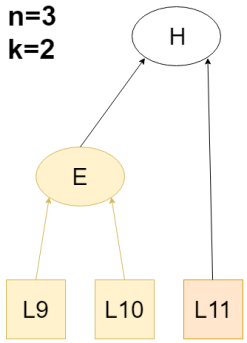


Figure 5: This is the first subtree of the previously generated subtree $n = 8$. Its MTH (**F**) is calculated by concatenating the MTHs of its two subtrees (A and B) with leaves from 1 to $k = 2$ [$L1 \rightarrow L2$] and with leaves from $k = 2$ to $n = 4$ [$L3 \rightarrow L4$]. This subtree's MTH is used to calculate its parent tree's MTH (I = **F** + G).

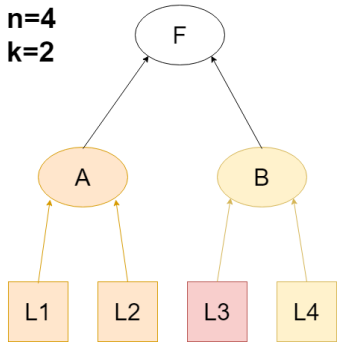
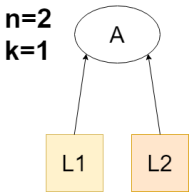


Figure 6: This is the first subtree of the previously generated subtree $n = 4$. Its MTH (**A**) is calculated by concatenating the leaf hashes of its two leaves L1 and L2 and used to calculate its parent tree's MTH (F = **A** + B).



1.4 Verifying proper inclusion

1.5 Verifying consistency

2 Merkle tree usage

2.1 Required database tables

2.1.1 PendingEntries

The *PendingEntries* table stores the data of log entries that were processed by the system but were not merged with the current Merkle tree. The *Id* of the record is a preassigned Merkle tree leaf index and is calculated by getting the size of the current Merkle tree plus the number of pending entries plus one. The structured data of the log entry is stored in the *Data* field of the table. A time stamp is added for information purposes.

Table 1: Database table *PendingEntries*

Id	**Data**	**TimeStamp**
INTEGER	BLOB	INTEGER
NOT NULL UNIQUE PRIMARY KEY	NOT NULL	NOT NULL

2.1.2 Entries

The *Entries* table stores all Merkle tree leaves that reside in the current Merkle tree and that were used to calculate the latest Merkle Tree Hash (MTH). The *Id*, *Data* and *TimeStamp* fields are identical to those of the *PendingEntries* table. They are simply copied to this table when a new Merkle tree is being generated. The additional field *TreeSize* denotes the size of the resulting Merkle tree in order to make a connection between a historical Merkle Tree Head of a Merkle tree of size *TreeSize* and a particular Merkle tree leaf. This table has to be append-only.

Table 2: Database table *Entries*

Id	**Data**	**TimeStamp**	**TreeSize**
INTEGER	BLOB	INTEGER	INTEGER
NOT NULL UNIQUE PRIMARY KEY	NOT NULL	NOT NULL	NOT NULL

2.1.3 TreeHeads

The *TreeHeads* table stores all historical Merkle Tree Heads that were generated during the operation of the system. It also includes the latest Merkle Tree Head - this is the Merkle Tree Head of the greatest *TreeSize*. The Merkle Tree Head (Hash) is stored in the *TreeHash* field. The *Id* field is not important and can be set to auto-increment. The signature over the corresponding *TreeHash* is stored in the *Signature* field. *TreeSize* denotes the size of the Merkle tree during the generation of the corresponding Merkle Tree Head, and *TimeStamp* - the time of the generation of the corresponding Merkle Tree Head. This table has to be append-only.

Table 3: Database table *TreeHeads*

Id	**TreeHash**	**TimeStamp**	**TreeSize**	**Signature**
INTEGER	BLOB	INTEGER	INTEGER	BLOB
NOT NULL UNIQUE PRIMARY KEY	NOT NULL	NOT NULL	NOT NULL	NOT NULL

2.1.4 CachedEntries

The *CachedEntries* table stores the hash values of important nodes in the current Merkle Tree. The hash of an important node is stored in the *Hash* field of the table. The *Start* field denotes the leaf index of the first Merkle tree leaf in the sequence, used to generate the particular node, and the *Stop* field - the leaf index of the last Merkle tree leaf in the sequence. The *Id* field is not important and can be set to auto-increment. Please refer to Section 3.1 for more information on how important nodes are selected. As some important nodes change when a new Merkle tree is generated, this table has to be updated regularly.

Table 4: Database table *CachedNodes*

Id	**Start**	**Stop**	**Hash**
INTEGER	INTEGER	INTEGER	BLOB
NOT NULL UNIQUE PRIMARY KEY	NOT NULL	NOT NULL	NOT NULL

2.2 Interactions

2.2.1 Log entry submission and Merkle tree generation

An application submits a log entry through the official LogSentinel API endpoints. The log entry data is then structured (encoded) in a special way, so that two entries with the same data but of different attribute order result in the same structured data. The ASN.1 or TLS encoding schemes could be used for this

purpose. Basic encoding schemes such as key-value concatenation are acceptable as well, as long as they have the desired, above-mentioned property. The hash of the structured (encoded) log entry data is then saved in the *PendingEntries* table in a remote database. The *Id* of the record is a preassigned Merkle tree leaf index and is calculated by getting the size of the current Merkle tree plus the number of pending entries plus one. Please note that the number of pending entries increases with every record inserted.

Every x hours a new Merkle tree is built. The maximum time needed to merge pending entries with the current Merkle tree is called MMD or Maximum Merge Delay. A reasonable MMD is a MMD of 24 hours.

When a new Merkle tree has to be built, all entries in the *PendingEntries* table are transfered to the *Entries* table. The *Id*, *Data* and *TimeStamp* properties of each new record in the *Entries* table are copied from the corresponding record from the *PendingEntries* table without modification. The *TreeSize* property in the new records is calculated by getting the size of the current Merkle tree plus the size of the *PendingEntries* table. When the transfer of all entries in the *Entries* table has been completed, all records in the *PendingEntries* table are to be deleted.

With all entries in the *Entries* table, the program generates a list of all leaves that are to be included in the new Merkle tree. This list should only include the leaf index of the corresponding leaves. Practically, this list has to contain all indexes from 1 to n, where n is the size of the *Entries* table. Then, a function is called that generates a Merkle Tree Head given a list of leaf indexes, generated in the previous step.

The MTH generation function returns a hash that has to be signed by the server and inserted in the *TreeHash* field of the *TreeHeads* table. This database record also includes the time of generation (*TimeStamp*) and the size of the current Merkle Tree (*TreeSize*). The record incorporates the calculated cryptographic signature as well.

2.2.2 Merkle inclusion proof generation

If a client wants to verify whether a particular log entry resides in the current Merkle tree, the system can produce an inclusion proof that is used for verification. For this purpose, the index of the leaf to be verified for inclusion is needed. The client can directly specify the leaf index or perform a search:

1. raw log entry data is submitted and the system generates the corresponding structured data hash. The systems performs a search in the database based on it. A leaf index is returned if a single entry is found. If there are more than one tree leaves of the same content, a list of indexes is returned.

2. the log entry data is structured and hashed client-side and the resulting hash is submitted. The systems performs a search in the database based on it. A leaf index is returned if a single entry is found. If there are more than one tree leaves of the same content, a list of indexes is returned.

The program generates a list of all leaves that are included in the current Merkle tree. This list should only include the leaf index of the corresponding leaves. Given this list and the leaf index, a function is then called that returns a list of intermediate nodes - the inclusion proof. By using them, the client can reconstruct the current Merkle tree and calculate the Merkle Tree Hash. It can then be compared to the one, advertised by the system.

2.2.3 Merkle consistency proof generation

If a client wants to verify the consistency between two advertised Merkle Tree Hashes, i.e. verify that any two versions of a log are consistent, the system can produce a consistency proof that is used for verification. For this purpose, the tree size of the previous (*tree_size1*) and the tree size of the newer tree (*tree_size2*) are needed (0 <*tree_size1* <*tree_size2*). If only one tree size is specified, the system assumes that the consistency between the specified Merkle tree and the current Merkle tree should be returned.

The program generates a list of leaves of size *tree_size2*. This list should only include the leaf index of the corresponding leaves. Given this list and *tree_size1*, a function is then called that returns a list of nodes - the consistency proof. By using them, the client can reconstruct the current Merkle tree (and the old one) and calculate the Merkle Tree Hash. It can then be compared to the one, advertised by the system. Additionally, the Merkle Tree Hash of the reconstructed old Merkle tree can be compared to the externally fetched one.

2.3 API endpoints

2.3.1 Retrieve general information

GET https://api.logsentinel.com/api/merkle/info

Outputs:

1. hashAlgorithm: The OID of the hash algorithm that is used by the system.
2. signatureAlgorithm: The OID of the signature algorithm that is used by the system.
3. publicKey: The public key that corresponds to the cryptographic key pair used by the system to sign Merkle Tree Hashes (MTHs).

2.3.2 Retrieve the current Merkle Tree Hash (MTH)

GET https://api.logsentinel.com/api/merkle/sth

Outputs:

1. treeSize: The size of the current Merkle tree.
2. mth: The Merkle Tree Hash (MTH) of the current Merkle tree.

3. signature: The cryptographic signature over the current Merkle Tree Hash (MTH).

4. timestamp: The time of generation of the current Merkle Tree Hash (MTH).

2.3.3 Submit a log entry for inclusion in the Merkle tree

POST https://api.logsentinel.com/api/merkle/submit

Authentication: basic

Inputs:

1. applicationId: Application ID, identifying a target application (obtained from the API credentials page). Required.

2. entry: Base64-encoded entry of any format about the event that is to be stored in the Merkle tree. Required.

Outputs:

1. leafIndex: The preassigned leaf index of the submitted entry.

Error codes:

Table 5: Error codes

Code	Meaning
400	One of the required inputs is missing.
401	The specified *applicationId* was not found or the provided authentication credentials were wrong.
500	The system cannot include this entry in the Merkle tree for unknown reasons.

2.3.4 Get an inclusion proof for a log entry

GET https://api.logsentinel.com/api/merkle/inclusion

Authentication: basic

Inputs:

1. applicationId: Application ID, identifying a target application (obtained from the API credentials page). Required.

2. leafIndex: Leaf index of the entry for which an inclusion proof has to be returned. Can be found using the search option of the API. Required.

Outputs:

1. inclusion: A list of Base64-encoded nodes or leaves that prove the inclusion of the specified entry in the current Merkle tree.

2. sth: The current Merkle Signed Tree Head (STH).

Error codes:

Table 6: Error codes

Code	Meaning
400	One of the required inputs is missing.
401	The specified *applicationId* was not found or the provided authentication credentials were wrong.
404	The specified *leafIndex* was not found.
500	The system cannot provide the inclusion proof for this entry in the Merkle tree for unknown reasons.

2.3.5 Retrieve the consistency proof for two versions of the Merkle tree

GET https://api.logsentinel.com/api/merkle/consistency

Authentication: basic

Inputs:

1. applicationId: Application ID, identifying a target application (obtained from the API credentials page). Required.

2. firstTreeSize: Size of the older tree. Required.

3. secondTreeSize: Size of the newer tree. Optional. If not specified, the system assumes that a consistency proof between the older tree and the current tree has to be returned.

Outputs:

1. consistency: A list of Base64-encoded nodes that prove the consistency between the two specified versions of the Merkle tree.

Error codes:

Table 7: Error codes

Code	Meaning
400	One of the required inputs is missing or *firstTreeSize* is greater than *secondTreeSize*.
401	The specified *applicationId* was not found or the provided authentication credentials were wrong.
404	The specified tree(s) was/were not found.
500	The system cannot provide the consistency proof for the specified versions of the Merkle tree for unknown reasons.

2.3.6 Retrieve leaves from database

GET https://api.logsentinel.com/api/merkle/get

Authentication: basic

Inputs:

1. applicationId: Application ID, identifying a target application (obtained from the API credentials page). Required.
2. start: 0-based index of first leaf to retrieve. Required.
3. stop: 0-based index of the last leaf to retrieve. Optional. If not specified, the system assumes that all leaves from *start* to *size* of the current Merkle tree have to be returned.

Outputs:

1. leaves: A list of Base64-encoded leaves with leaf indexes between *start* and *stop*.

Error codes:

Table 8: Error codes

Code	Meaning
400	One of the required inputs is missing or *start* is greater than *stop*.
401	The specified *applicationId* was not found or the provided authentication credentials were wrong.
500	The system cannot provide the requested leaves for unknown reasons.

2.3.7 Search for a specific leaf in the database

GET https://api.logsentinel.com/api/merkle/search

Authentication: basic

Inputs:

1. applicationId: Application ID, identifying a target application (obtained from the API credentials page). Required.
2. hash: The hash of the log entry, for which a leaf index has to be returned. Required.

Outputs:

1. indexes: A list of indexes of leafs that match the search criteria. If only one entry was found, matching the search criteria, the list contains only one leaf index.

Error codes:

Table 9: Error codes

Code	Meaning
400	One of the required inputs is missing.
401	The specified *applicationId* was not found or the provided authentication credentials were wrong.
404	No leaves were found matching the specified hash.
500	The system cannot provide the requested leaf indexes for unknown reasons.

3 Resource-efficient storage and generation of Merkle trees

3.1 Saving important nodes in the database

[Laurie et al., 2018] [Dahlberg et al., 2016] [certificate transparency.org, 2018]

References

[certificate transparency.org, 2018] certificate transparency.org (2018). Certificate transparency. Google Sites. `https://www.certificate-transparency.org`.

[Dahlberg et al., 2016] Dahlberg, R., Pulls, T., and Peeters, R. (2016). Efficient sparse merkle trees: Caching strategies and secure (non-)membership proofs. Cryptology ePrint Archive, Report 2016/683. `https://eprint.iacr.org/2016/683`.

[Laurie et al., 2018] Laurie, B., Langley, A., Kasper, E., Messeri, E., and Stradling, R. (2018). Certificate transparency version 2.0. *IETF Tools*.