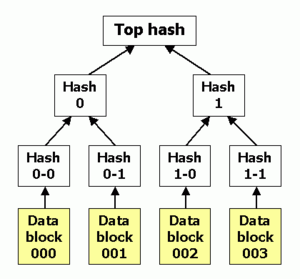
Merkle Tree

Merkle proposed a digital signature scheme that was based on both one-time signatures ([see Question 43](http://x5.net/faqs/crypto/q43.html)) and a hash function (see [Question 94](http://x5.net/faqs/crypto/q94.html)) and that provides an infinite tree of one-time signatures.

One-time signatures normally require the publishing of large amounts of data to authenticate many messages, since each signature can only be used once. Merkle's scheme solves the problem by implementing the signatures via a tree-like scheme. Each message to be signed corresponds a node in a tree, with each node consisting of the verification parameters that are used to sign a message and to authenticate the verification parameters of subsequent nodes. Although the number of messages that can be signed is limited by the size of the tree, the tree can be made arbitrarily large. Merkle's signature scheme is fairly efficient, since it requires only the application of hash functions.

# Hash tree

From Wikipedia, the free encyclopedia

[](http://en.wikipedia.org/wiki/File:Hash_tree.png)

[http://bits.wikimedia.org/skins-1.17/common/images/magnify-clip.png](http://en.wikipedia.org/wiki/File:Hash_tree.png)

A binary hash tree

In [cryptography](http://en.wikipedia.org/wiki/Cryptography) and [computer science](http://en.wikipedia.org/wiki/Computer_science) **Hash trees** or **Merkle trees** are a type of [data structure](http://en.wikipedia.org/wiki/Data_structure) which contains a [tree](http://en.wikipedia.org/wiki/Tree_(data_structure)) of summary information about a larger piece of data – for instance a file – used to verify its contents. Hash trees are an extension of [hash lists](http://en.wikipedia.org/wiki/Hash_list), which in turn are an extension of [hashing](http://en.wikipedia.org/wiki/Hash_function). Hash trees in which the underlying hash function is [Tiger](http://en.wikipedia.org/wiki/Tiger_(cryptography)) are often called **Tiger trees**or **Tiger tree hashes**.

|  |
| --- |
|  |

## Uses

Hash trees can be used to protect any kind of data stored, handled and transferred in and between computers. Currently the main use of hash trees is to make sure that data blocks received from other peers in a [peer-to-peer network](http://en.wikipedia.org/wiki/Peer-to-peer) are received undamaged and unaltered, and even to check that the other peers do not lie and send fake blocks. Suggestions have been made to use hash trees in [trusted computing](http://en.wikipedia.org/wiki/Trusted_computing) systems. [Sun Microsystems](http://en.wikipedia.org/wiki/Sun_Microsystems) has used Hash Trees in the[ZFS](http://en.wikipedia.org/wiki/ZFS) filesystem.[[1]](http://en.wikipedia.org/wiki/Hash_tree#cite_note-0) Hash Trees are used in [Google Wave](http://en.wikipedia.org/wiki/Google_Wave) protocol[[2]](http://en.wikipedia.org/wiki/Hash_tree" \l "cite_note-1) and in [tarsnap](http://en.wikipedia.org/w/index.php?title=Tarsnap&action=edit&redlink=1" \o "Tarsnap (page does not exist)) backup system.

Hash trees were invented in 1979 by [Ralph Merkle](http://en.wikipedia.org/wiki/Ralph_Merkle).[[3]](http://en.wikipedia.org/wiki/Hash_tree#cite_note-2) The original purpose was to make it possible to efficiently handle many [Lamport one-time signatures](http://en.wikipedia.org/wiki/Lamport_signature" \o "Lamport signature). Lamport signatures are believed to still be secure in the event that [quantum computers](http://en.wikipedia.org/wiki/Quantum_computer) become reality. Unfortunately each Lamport key can only be used to sign a single message. But combined with hash trees they can be used for many messages and then become a [fairly efficient digital signature scheme](http://en.wikipedia.org/wiki/Merkle_signature_scheme).

## [[edit](http://en.wikipedia.org/w/index.php?title=Hash_tree&action=edit&section=2)]How hash trees work

A hash tree is a [tree](http://en.wikipedia.org/wiki/Binary_tree) of [hashes](http://en.wikipedia.org/wiki/Hash_function) in which the leaves are hashes of data blocks in, for instance, a file or set of files. Nodes further up in the tree are the hashes of their respective children. For example, in the picture *hash 0* is the result of hashing *hash 0-0* and then *hash 0-1*. That is, *hash 0 = hash( hash 0-0 + hash 0-1 )* where + denotes concatenation.

Most hash tree implementations are binary (two child nodes under each node) but they can just as well use many more child nodes under each node.

Usually, a [cryptographic hash function](http://en.wikipedia.org/wiki/Cryptographic_hash_function) such as [SHA-1](http://en.wikipedia.org/wiki/SHA-1), [Whirlpool](http://en.wikipedia.org/wiki/Whirlpool_(hash)), or [Tiger](http://en.wikipedia.org/wiki/Tiger_(hash)) is used for the hashing. If the hash tree only needs to protect against unintentional damage, much less secure [checksums](http://en.wikipedia.org/wiki/Checksum) such as [CRCs](http://en.wikipedia.org/wiki/Cyclic_redundancy_check) can be used.

In the top of a hash tree there is a *top hash* (or *root hash* or *master hash*). Before downloading a file on a p2p network, in most cases the top hash is acquired from a trusted source, for instance a friend or a web site that is known to have good recommendations of files to download. When the top hash is available, the hash tree can be received from any non-trusted source, like any peer in the p2p network. Then, the received hash tree is checked against the trusted top hash, and if the hash tree is damaged or fake, another hash tree from another source will be tried until the program finds one that matches the top hash.

The main difference from a [hash list](http://en.wikipedia.org/wiki/Hash_list) is that one branch of the hash tree can be downloaded at a time and the integrity of each branch can be checked immediately, even though the whole tree is not available yet. This can be an advantage since it is efficient to split files up in very small data blocks so that only small blocks have to be redownloaded if they get damaged. If the hashed file is very big, such a hash tree or hash list becomes fairly big. But if it is a tree, one small branch can be downloaded quickly, the integrity of the branch can be checked, and then the downloading of data blocks can start.

There are several additional tricks, benefits and details regarding hash trees. See the references and external links below for more in-depth information.

|  |  |
| --- | --- |
|  |  |
|  |  |
| Merkle hash tries (a data structure) |  |

This dart proposes a cryptographic data structure that may be of use in Storm.

A *trie* is a tree which stores a set of strings, with a leaf node for every string, and with an internal node for every common prefix. For example,

[root]

/ \

[0] [1]

/ \

[00] [11]

/ \ \

0001 0010 [111]

/ \

1110 111101

The nodes with one child can also be omitted for efficiency, i.e.

[root]

/ \

/ \

/ \

[00] \

/ \ \

0001 0010 [111]

/ \

1110 111101

A *hash trie* is a trie where the (bit)strings are the *hashes* of the objects you actually want to store.

A Merkle hash trie, then, is a hash trie stored as a Merkle hash tree: The root and internal nodes are labelled with the hash of their children.

In Storm, each node would be a Storm block, and the root and internal nodes would contain the block ids of their child nodes.

Obviously, we would also use Storm ids as the hashes stored in the trie.

Hash tries can also be used to implement a *mapping*, rather than a set: Then, each leaf node stores a key/value pair (where both key and value are hashes), and the key is used for traversing the trie.

Merkle hash tries can be used to--

* give an efficient proof that a certain element is part of a set (this is what vanilla Merkle hash trees also do);
* give an efficient proof that a certain key maps to a certain value-- and *only* that value-- in a mapping;
* give an efficient proof that a certain element or key doesn't exist in a set or mapping;
* given a whole trie, traverse the trie to find the value associated with a given key.

"Efficient" means that for n elements in the trie, the proofs need only O(log n) space (and time to verify).

Also, the data structure leans itself to versioning: Storing an updated version of a trie needs only O(log n) additional space per added/modified/deleted entry.

Finally, the tree automatically approximately balances itself (because the hashes are approximately randomly distributed).

Applications could include, for example, certificate revocation trees.

**File format**

Below, a proposed file (block) format for hash tries, able to represent both sets and mappings.

As proposed above, every node of the tree is one Storm block. The file format is binary, because this stuff is relatively space-intensive. Each block starts with a type byte:

0x00 -- branch node (may be the root)

0x01 -- leaf node of set (one element)

0x02 -- leaf node of map (key/value pair)

Any other type byte is a fatal error; a processor must abort.

A *branch node* contains two hashes (bitprints in binary form, which means 160+192 bits or 44 bytes). The first is the child node for '0' bits, the second the child node for '1' bits.

The empty tree is designated by 44 0x00 (zero) bytes. If a branch node has only one child, the other child is given as 44 zero bytes.

A *leaf node of set* contains a single hash, while a *leaf node of map* contains two hashes, the first being the key, the second being the value.

A trie is referred to by the root's block id.

- Benja