On Merkle Trees

Dr Craig S Wright

By now, it should be well understood that Bitcoin utilises the concept of a Merkle tree to its advantage; by way of background, we provide an explanation in this post. The following aids as a technical description and definition of similar concepts used more generally in Bitcoin and specifically within SPV.

Merkle trees are hierarchical data structures that enable secure verification of collections of data. In a Merkle tree, each node in the tree has been given an index pair (i, j) and is represented as $N(i, j)$. The indices i, j are simply numerical labels that are related to a specific position in the tree. An important feature of the Merkle tree is that the construction of each of its nodes is governed by the following (simplified) equations:

$$N(i,j) = \begin{cases} H(D_i) & i = j \\ H\big(N(i,k) \parallel N(k+1,j)\big) & i \neq j \end{cases},$$

where $k = (i + j - 1)/2$ and H is a cryptographic hash function.

A labelled, binary Merkle tree constructed according to the equations is shown in figure 1, from which we can see that the $i = j$ case corresponds to a leaf node, which is simply the hash of the corresponding i^{th} packet of data D_i. The $i \neq j$ case corresponds to an internal or parent node, which is generated by recursively hashing and concatenating child nodes, until one parent (the Merkle root) is found.

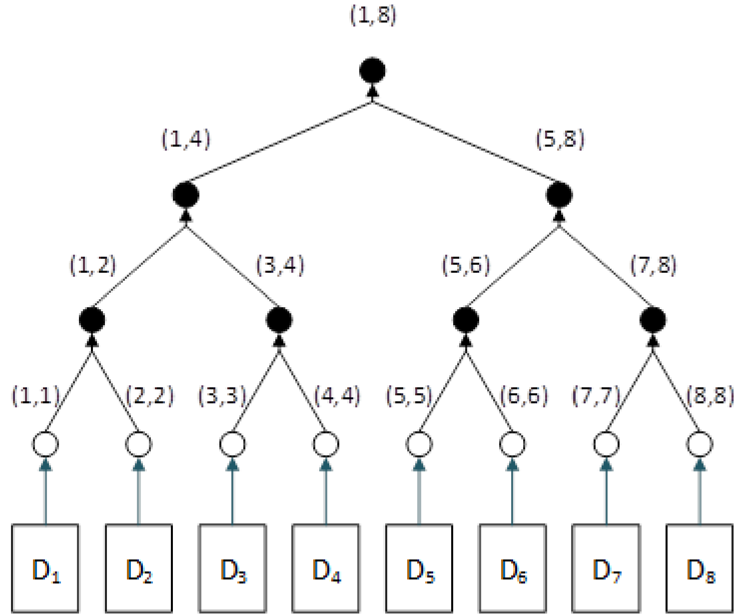

Figure 1. A Merkle tree.

For example, the node $N(1,4)$ is constructed from the four data packets $D_1, \dots, D_4$ as

$$
\begin{aligned}
N(1,4) &= H\big(N(1,2) \parallel N(3,4)\big) \\
&= \big[H\big(N(1,1) \parallel N(2,2)\big) \parallel H\big(N(3,3) \parallel N(4,4)\big)\big] \\
&= \big[H\big(H(D_1) \parallel H(D_2)\big) \parallel H\big(H(D_3) \parallel H(D_4)\big)\big].
\end{aligned}
$$

The tree depth M is defined as the lowest level of nodes in the tree, and the depth m of a node is the level at which the node exists. For example, $m_{root} = 0$ and $m_{leaf} = M$, where $M = 3$ in figure 1.

Merkle Proofs

The primary function of a Merkle tree is to verify that some data packet D_i is a member of a list or set of N data packets $\mathcal{D} \in \{D_1, \dots, D_N\}$. The mechanism for verification is known as a Merkle proof, and consists of obtaining a set of hashes known as the Merkle path for a given data packet D_i and Merkle root R. The Merkle path for a data packet is simply the minimum list of hashes required to reconstruct the root R by way of repeated hashing and concatenation, often referred to as the *authentication path*. A proof of existence could be performed trivially if all packets $D_1, \dots, D_N$ are known to the prover, which does, though, require a much larger storage overhead than the Merkle path and that the entire data set is available to the prover. The comparison between using a Merkle path and using the entire list is shown in the table below, where we have used a binary Merkle tree and assumed that the number of data blocks

N is exactly equal to an integer power 2. If such were not the case, the number of hashes required for the Merkle proof would differ by ±1 in each instance.

					Merkle tree
No. data packets	**8**	**32**	**64**	**256**	$\boldsymbol{N = 2^M}$
No. hashes required for proof of existence	**3**	**5**	**7**	**9**	$\boldsymbol{M = \log_2 N}$

Table: The relationship between the number of leaf nodes in a Merkle tree and the number of hashes required for a Merkle proof.

In this simplified scenario—where the number of data packets is equal to the number of leaf nodes—we find that the number of hash values required to compute a Merkle proof scales logarithmically. It is clearly far more efficient and practical to compute a Merkle proof involving $\log_K N$ hashes than to store N data hashes and compute the trivial proof.

Method

If, given a Merkle root R, we wish to prove that the data block D_1 belongs to the set $\mathcal{D} \in \{D_1, \dots, D_N\}$ represented by R, we can perform a Merkle proof as follows:

i. Obtain the Merkle root R from a trusted source;

ii. Obtain the Merkle path Γ from a source. In this case, Γ is the set of hashes:

$$\Gamma = \{N(2,2),\ N(3,4),\ N(5,8)\};$$

iii. Compute a Merkle proof using D_1 and Γ as follows:

a. Hash the data block to obtain:

$$N(1,1) = H(D_1);$$

b. Concatenate with $N(2,2)$ and hash to obtain:

$$N(1,2) = H\big(N(1,1) \parallel N(2,2)\big);$$

c. Concatenate with $N(3,4)$ and hash to obtain:

$$N(1,4) = H\big(N(1,2) \parallel N(3,4)\big);$$

d. Concatenate with $N(5,8)$ and hash to obtain the root:

$$N(1,8) = H(N(1,4) \parallel N(5,8));$$

$$R' = N(1,8);$$

e. Compare the calculated root R' with the root R obtained in (i):

I. If $R' = R$, the existence of D_1 in the tree and therefore the data set $\mathcal{D}$ are confirmed;

II. If $R' \neq R$, the proof has failed and D_1 is not confirmed to be a member of $\mathcal{D}$.

This is an efficient mechanism for providing a proof of existence for some data as part of the data set represented by a Merkle tree and its root. For example, if the data D_1 corresponded to a blockchain transaction and the root R is publicly available as part of a block header, then we can quickly prove that the transaction was included in the corresponding block.

The process of authenticating the existence of D_1 as part of our example Merkle tree is shown in figure 2, which shows a Merkle proof of existence of data block D_1, in the tree represented by a root R using a Merkle path. It demonstrates that performing the Merkle proof for a given block D_1 and root R is effectively traversing the Merkle tree 'upwards'—by using only the minimum number of hash values necessary.

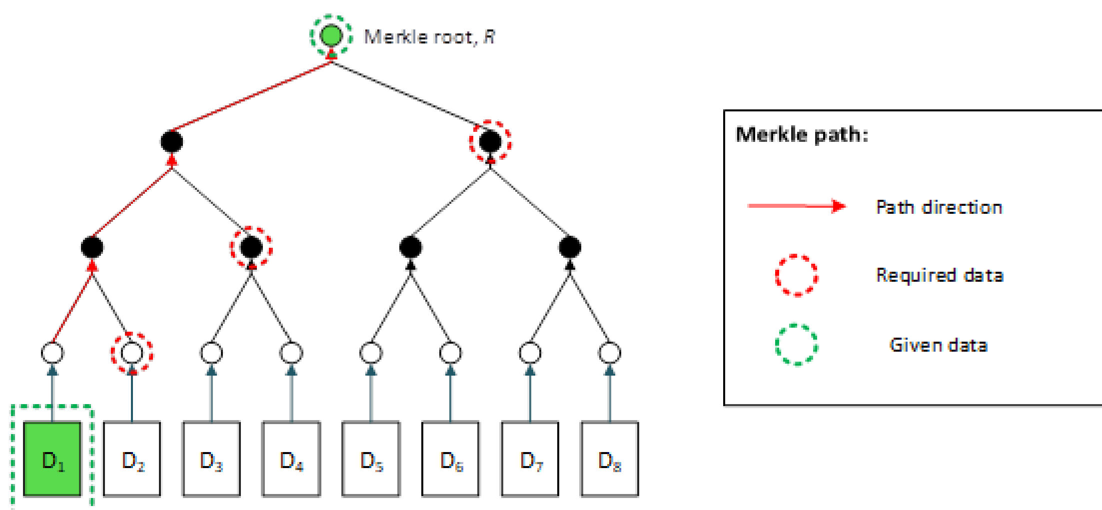

Figure 2.

Such techniques form an important component in implementing SPV in an efficient and secure manner, allowing us to scale and effectively implement a verification solution that provides true peer-to-peer transactioning.

Continue reading the article on my personal blog here.