**Chapter 3**

**ANALYSIS OF MERKLE TREE ALGORITHM**

**3.1 Merkle Tree Algorithm**

Merkle trees are typically implemented as binary trees where each non-leaf node is a hash of the two nodes below it. The leaves can either be the data itself or a hash/signature of the data.

Thus, if any difference at the root hash is detected between systems, a binary search can be done through the tree to determine which particular subtree has the problem. Thus typically only log(N) nodes need to be inspected rather than all N nodes to find the problem area.

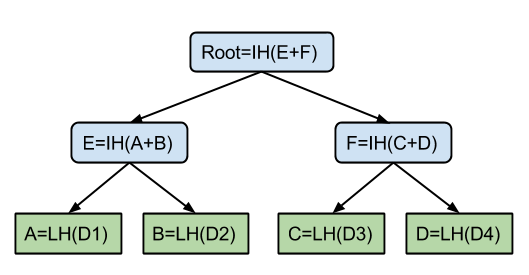
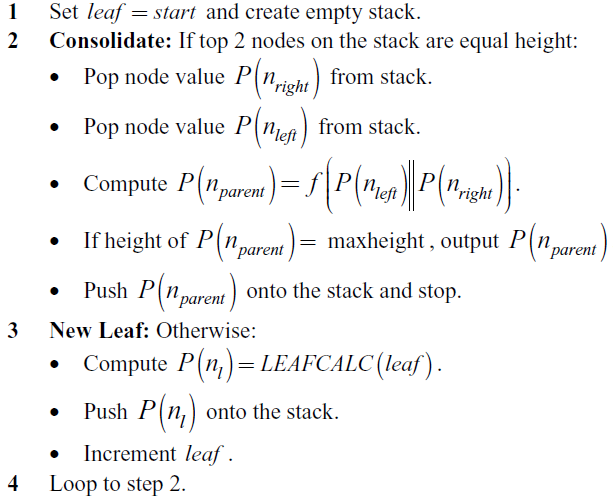


Figure 3.1: Merkle Tree Structure

**3.2 Efficient Node Computation**

As already explained, before the application of a traversal technique we need a technique to quickly pre-calculate a number of node values, together with the root value. This is done by the *TREEHASH* algorithm, which is created in a space conserving manner. The algorithm is really simple. The basic idea is to compute node values at the same height first, before continuing the calculations with a new leaf, and thus up to the root (keep in mind that the value *P*(*n*) of an arbitrary node *n* of the Merkle tree is estimated with the help of the values of its children nodes). The implementation of *TREEHASH* is simply done by a stack (the usage of a stack to simplify the algorithm was influenced by recent work on time stamping. At first the stack is empty and then we start adding leaf values one by one onto it (starting from leaf with index zero).

Algorithm TREEHASH (start, maxheight):



In most cases we need to implement the *TREEHASH* algorithm into the traversal algorithms themselves, which means into bigger algorithms. This is possible by defining an object with two methods. The first method is called *initialize* and it simply sets which leaf we start the *TREEHASH* with, and what the height of the desired output is. For example later we will meet the following statement – *Stackh initialize (startnode , h)* which tells us that we will modify the *Stackh* stack starting with the leaf of index *startnode* and going up to height *h* ( *h*≤ *H* ). The second method is called *update* and it runs either step 2 or step 3 of the *TREEHASH* algorithm, thus modifying the content of the used stack (2 and 3 are the steps that actually change the stack). Just to understand this easier we give another example - *Stackh (update)*. We use the same stack as in the previous example. It means that for the already known *Stackh* stack we spend two units of computation. Which one of the two steps is run depends on the current state of the stack – if we need to add another leaf value or to calculate a parent node value. After

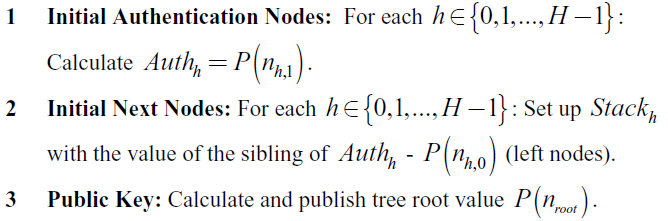
*TREEHASH* is run for a particular Merkle tree, the only remaining value on the stack is the value of the root.

**3.3 The classic traversal**

In a few words only, the algorithm is very simple. We have our Merkle tree, and as usual, the root is the public key and the leaves are the one-time private keys in our digital signature scheme. By definition, every such scheme has three phases with the following description:

1. Key generation: In this phase we calculate the root value of the tree, the first authentication path and some upcoming node values.
2. Output: This phase consists of N rounds, one for each leaf. Every round the current leaf value is output, together with the authentication path for this leaf {Authi}. After that the tree is updated in order to prepare it for the next round.
3. Verification: That is the traditional verification of leaf values in a Merkle tree. The process is already described in section 2.3.5 and is rather simple.

Key Generation and Setup

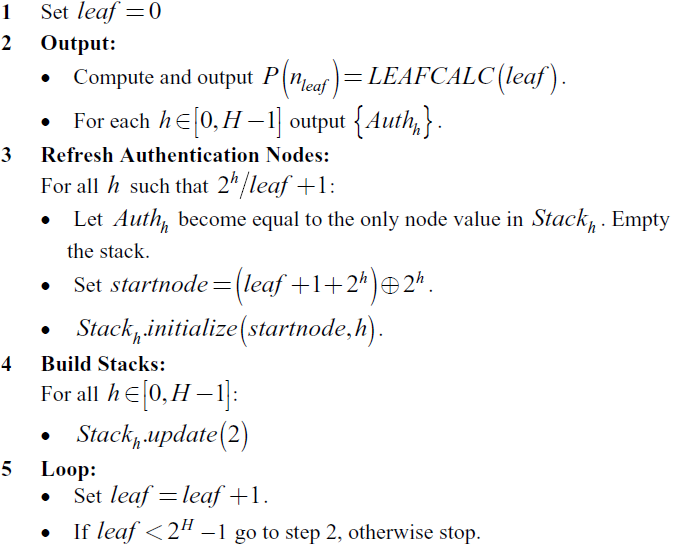


After this phase of the classic Merkle tree traversal we start the next one with already computed leftmost nodes and their siblings.

**3.4 Output and Update**

This is the phase in which we output every leaf value together with the authentication path of the corresponding leaf (we use LEAFCALC for the leaf values computation). For the purpose we have a counter leaf which starts from zero and is increased by one every round, thus denoting the index of the current leaf (we run the algorithm once for each leaf of the tree). The outputs are made in the beginning of every round. After the output is made we come to the refreshment of the authentication nodes (the leftmost nodes in the very beginning of the traversal). The idea is that we shift these authentication nodes to the right when they are no longer needed for upcoming authentication paths.

Classic Merkel Tree Traversal



**3.5 Merkel Tree traversal with time and space requirement**

Time Requirements : 2 log (N)

Space Requirements : log2(N) / 2