

Darmstadt University of Technology  
Department of Computer Science  
Cryptography and Computer Algebra

# Merkle Tree Traversal Techniques

Bachelor Thesis  
by  
**Boris Ederov**



Prof. Dr. Johannes Buchmann  
Supervised by Erik Dahmen

April 2007



# Index

<b>Abstract</b>	<b>5</b>
<b>Chapter 1: Introduction</b>	<b>6</b>
<b>Chapter 2: The Basics</b>	<b>8</b>
<b>2.1 Hash and One–Way Functions</b>	<b>8</b>
2.1.1 One - Way Functions	8
2.1.2 Hash functions	9
<b>2.2 Digital Signatures</b>	<b>9</b>
2.2.1 The Definition	9
2.2.2 One - Time Digital Signatures	10
<b>2.3 The Original Merkle Digital Signature</b>	<b>10</b>
2.3.1 Introduction	10
2.3.2 The Lamport-Diffie one-time signature scheme	11
2.3.3 Winternitz improvement	12
2.3.4 Merkle trees	12
2.3.5 Authentication Paths	13
<b>Chapter 3: Tree Traversal Techniques</b>	<b>14</b>
<b>3.1 Authentication Path Regeneration</b>	<b>14</b>
<b>3.2 Efficient Nodes Computation</b>	<b>14</b>
<b>3.3 The Classic Traversal</b>	<b>16</b>
3.3.1 Key Generation and Setup	16
3.3.2 Output and Update	17
3.3.3 Example for the Classic Traversal	18
<b>3.4 Fractal Tree Representation and Traversal</b>	<b>25</b>
3.4.1 Notation	25
3.4.2 Existing and Desired Subtrees	26
3.4.3 Algorithm Presentation	27
3.4.4 Example for the fractal algorithm	29

<b>3.5 Logarithmic Merkle Tree Traversal</b>	<b>33</b>
<b>Chapter 4: Algorithm Comparison</b>	<b>35</b>
<b>4.1 The Classic Traversal</b>	<b>35</b>
<b>4.2 The Improvement</b>	<b>36</b>
<b>4.3 The Fractal Traversal</b>	<b>36</b>
<b>Chapter 5: Conclusions and Future Work</b>	<b>41</b>
<b>Acknowledgements</b>	<b>41</b>
<b>References</b>	<b>42</b>

**Abstract.** We introduce Merkle tree traversal techniques. We start with the original tree traversal and then two improvements. Motivated by the possibility of using Merkle authentication, or digital signatures in slow or space - constrained environments, RSA Laboratories focused on improving the efficiency of the authentication/signature operation, building on earlier, unpublished work of Tom Leighton and Silvio Micali. Although hash functions are very efficient, too many secret leaf values would need to be authenticated for each digital signature. By reducing the time or space cost, we found that for medium - size trees the computational cost can be made sufficiently efficient for practical use. This reinforced the belief that practical, secure signature/authentication protocols are realizable, even if the number theoretic algorithms were not available.

The first published paper, *Fractal Merkle Tree Representation and Traversal* [JLMS03], shows how to modify Merkle's scheduling algorithm to achieve various tradeoffs between storage and computation. The improvement is achieved by means of a careful choice of what nodes to compute, retain, and discard at each stage. The use of this technique is "transparent" to a verifier, who will not need to know how a set of outputs were generated, but only that they are correct. So the technique can be employed in any construction for which the generation and output of consecutive leaf pre - images and corresponding authentication paths is required. The paper was presented at the Cryptographer's Track, RSA Conference 2003 (May 2003). This construction roughly speeds up the signing operation inherent in Merkle's algorithm at a cost of requiring more space. An implementation of this algorithm is publicly available.

The second paper, *Merkle Tree Traversal in Log Space and Time* [SZY04], exhibits an alternate, basic logarithmic space and time algorithm, which is very space efficient, but provides no trade off. The improvement over previous traversal algorithms is achieved as a result of a new approach to scheduling the node computations. The paper has been presented at Eurocrypt (May 2004). This construction is roughly as fast as Merkle's original algorithm [MER79], but requires less space.

Both options increase the options for deployment, even in low - power or low memory devices.

# Chapter 1

## Introduction

Historically, cryptography arose as a means to enable parties to maintain privacy of the information they send to each other, even in the presence of an adversary with access to the communication channel. While providing privacy remains a central goal, the field has expanded to encompass many others, including not just other goals of communication security, such as guaranteeing integrity and authenticity of communications and many more sophisticated and fascinating goals. When you shop on the Internet, for example to buy a book at Amazon, cryptography is used to ensure privacy of your credit card number as it travels from you to the shop's server. Or, in electronic banking, cryptography is used to ensure that your checks cannot be forged. Cryptography has been used almost since writing was invented. For the larger part of its history, cryptography remained a game of chess - the good cryptographer must always consider two points of view - of the defender and the attacker. Although the field retains some of this flavor, the last twenty - five years have brought in something new. The art of cryptography has now been supplemented with a legitimate science. Modern cryptography is a remarkable discipline. It is a cornerstone of computer and communications security, with end products that are imminently practical. Yet its study touches on branches of mathematics that may have been considered esoteric, and it brings together fields like number theory, computational - complexity theory, and probability theory.

The basics that are of great significance for this thesis, we are going to consider in chapter 2. The most important invention in cryptography comes in 1976 with the publication of Diffie and Hellman - "New Directions in Cryptography" [DH76]. In it they introduce the concept of public - key cryptography, a form of cryptography, which generally allows users to communicate securely without having prior access to a shared secret key. This is done by using a pair of cryptographic keys, designated as public key and private key, which are related mathematically. In public key cryptography, the private key is kept secret, while the public key may be widely distributed. In a sense, one key "locks" a lock, while the other is required to unlock it. It should not be feasible to deduce the private key of a pair given the public key, and in high quality algorithms no such technique is known. One analogy is that of a locked store front door with a mail slot. The mail slot is exposed and accessible to the public: its location (the street address) is in essence the public key. Anyone knowing the street address can go to the door and drop a written message through the slot. However, only the person, who possesses the matching private key, the storeowner in this case, can open the door and read the message. Every user can compress a particular message using a process called *hashing*. From a large document for example we get a few lines, called *message digest* (it is not possible change the message digest back into the original document). If the user encrypts the message digest with his private key, the result will be the so - called *digital signature*.

Nowadays a very important role in cryptography plays the quantum computer. Integer factorization is believed to be computationally infeasible with an ordinary computer for large numbers that are the product of two prime numbers of roughly equal size (e.g., products of two 300 - digit primes). By comparison, a quantum computer could solve this problem relatively easily. If a number has  $n$  bits (is  $n$  digits long in its binary representation), then a quantum computer with just over  $2n$  qubits can use some particular algorithms to find its factors. It can also solve a related problem called the discrete logarithm problem. This ability would allow a quantum computer to "break" many of the cryptographic systems in use today, in the sense that there would be a relatively fast (polynomial time in  $n$ ) algorithm for solving the problem. In particular, most of the popular public key ciphers could be much more quickly broken, including forms of RSA, El - Gamal and Diffie - Hellman. These are used to protect secure Web pages, encrypted email, and many other types of data. Breaking these would have significant ramifications for electronic privacy and security. The only way to increase the security of an algorithm like RSA would be to increase the key size and hope that an adversary does not have the resources to build and use a powerful enough quantum computer. It seems plausible that it will always be possible to build classical computers that have more bits than the number of qubits in the largest quantum computer. If that's true, then algorithms like RSA could be made secure by ensuring that key lengths exceed the storage capacities of quantum computers. There are some digital signature schemes that are believed to be secure against quantum computers. Thus the Lamport - Diffie one - time digital signature scheme is presented. Roughly speaking, this is a signature scheme, which is guaranteed to be secure as long as each private key is not used more than once. It is based on a general *hash function*.

In 1979 Ralph C. Merkle presents in [MER79] a multi - time signature scheme, which employs a version of the Lamport - Diffie one - time signature scheme. In fact the Merkle signature scheme transforms any one - time signature scheme into a multi - time one. This extension is made by complete binary trees, which we call Merkle trees. Merkle trees stand as the very basis of this thesis and have found many uses in theoretical cryptographic constructions. The main role they play is to verify the leaf values with respect to the publicly known root value and the *authentication data* of that particular leaf. This data consists of one node value at each height, where these nodes are the siblings of the nodes on the path connecting the leaf to the root.

In chapter 3 and 4 we will consider the main problems in this thesis - the Merkle tree traversal problem, algorithms for solving it and their comparison. The Merkle tree traversal is the task of finding efficient algorithms to output the authentication data for successive leaves. Thus, this purpose is different from other, better - known, tree traversal problems found in the literature. As elegant as Merkle trees are, they are used less than one might expect. One reason is that known traversal techniques require large amount of computation, storage or both. This means that only the smallest trees can be practically used. However, with more efficient traversal techniques, Merkle trees may once again become more compelling, especially given the advantage that cryptographic constructions based on Merkle trees do not require any number theoretic assumptions.

# Chapter 2

## The Basics

In this chapter we are going to introduce the basic mathematical notions in cryptography, which are of great significance to this thesis. We shall talk briefly about digital signatures and one - time digital signatures, about hash and one - way functions, and of course, about the original Merkle digital signature, which is of crucial importance for Merkle tree traversal techniques.

### 2.1 Hash and one - way functions

2.1.1 **One - way functions:** A *one - way function* is a mathematical function that is significantly easier to compute in one direction (the forward direction) than in the opposite direction (the inverse direction). It might be possible, for example, to compute the function in the forward direction in seconds but to compute its inverse could take months or years, if at all possible. A *trapdoor one - way function* is a one - way function for which the inverse direction is easy given a certain piece of information (the trapdoor), but difficult otherwise. Public - key cryptosystems are based on (presumed) trapdoor one - way functions. The public key gives information about the particular instance of the function; the private key gives information about the trapdoor. Whoever knows the trapdoor can compute the function easily in both directions, but anyone lacking the trapdoor can only perform the function easily in the forward direction. The forward direction is used for encryption and signature verification; the inverse direction is used for decryption and signature generation. In almost all public - key systems, the size of the key corresponds to the size of the inputs to the one - way function; the larger the key, the greater the difference between the efforts necessary to compute the function in the forward and inverse directions (for someone lacking the trapdoor). For a digital signature to be secure for years, for example, it is necessary to use a trapdoor one - way function with inputs large enough that someone without the trapdoor would need many years to compute the inverse function (that is, to generate a legitimate signature). All practical public key cryptosystems are based on functions that are believed to be one - way, but no function has been proven to be so. This means it is theoretically possible to discover algorithms that can compute the inverse direction easily without a trapdoor for some of the one - way functions; this development would render any cryptosystem based on these one - way functions insecure and useless. On the other hand, further research in theoretical computer science may result in concrete lower bounds on the difficulty of inverting certain functions; this would be a landmark event with significant positive ramifications for cryptography. One way - functions are basic for this thesis and their major use is for authentication.

2.1.2 **Hash functions:** The term *hash* apparently comes by way of analogy with its standard meaning in the physical world, to "chop and mix" [WK01]. The first use of the concept was in a memo from 1953, some ten years later the term *hash* came into use. Mathematically, a *hash function* (or hash algorithm) is a method of turning data into a number suitable to be handled by a computer. It provides a small digital "fingerprint" from any kind of data. The function substitutes or transposes the data to create that "fingerprint", usually called *hash value*. This value is represented as a short string of random - looking letters and numbers (for example binary data written in hexadecimal notation). By hash functions appears also the term *hash collision* - the situation when two different inputs produce identical outputs. Of course the better the hash function, the smaller number of collisions that occur. Similarly in cryptography we have a cryptographic hash function, which is simply a normal hash function with additional security properties. This is needed for their use in various information security applications, such as authentication and message integrity. Usually a long string of any length is taken as input and a string of fixed length is given as output. The output is sometimes termed a digital fingerprint. One desirable property of cryptographic hash functions is *collision resistance*. A hash function is collision resistant it is "infeasible" to find a collision.

It is trivial to prove that every collision resistant hash function is a one - way function. Because of this we are going to use hash functions instead of one - way ones.

## 2.2 Digital signatures

2.2.1 **The definition:** This chapter considers techniques designed to provide the digital counterpart to a handwritten signature. A *digital signature* of a message is a number dependent on some secret known only to the signer, and, additionally, on the content of the message being signed. Signatures must be verifiable; if a dispute arises whether a party signed a document (caused by either a lying signer, trying to repudiate the signature that the party created, or a fraudulent claimant), an unbiased third party should be able to resolve the matter equitably, without requiring access to the signer's secret information (private key). Digital signatures have many applications in information security, including authentication, data integrity, and non - repudiation. One of the most significant applications of digital signatures is the certification of public keys in large networks. Certification is a means for a trusted third party (TTP) to bind the identity of a user to a public key, so that at some later time, other entities can authenticate a public key without assistance from a trusted third party. The concept and utility of a digital signature was recognized several years before any practical realization was available. Research has resulted in many different digital signature techniques. Some offer significant advantages in terms of functionality and implementation. In more

mathematical manner we can give the following definitions, according to [MOV96]:

1. A *digital signature* is a data string, which associates a message (in digital form) with some originating entity.
2. A *digital signature generation algorithm* is a method for producing a digital signature.
3. A *digital signature verification algorithm* is a method for verifying that a digital signature is authentic (i.e., was indeed created by the specified entity).
4. A *digital signature scheme* (or mechanism) consists of signature generation algorithm and an associated verification algorithm.
5. A *digital signature signing process* (or procedure) consists of a digital signature generation algorithm, along with a method for formatting data into messages, which can be signed.
6. A *digital signature verification process* (or procedure) consists of a verification algorithm, along with a method for recovering data from the message.

2.2.2 **One - time digital signatures:** One - time digital signature schemes are digital signature mechanisms, which can be used to sign, at most, one message; otherwise, signatures can be forged [MOV96]. A new public key is required for each message that is signed. The public information necessary to verify one - time signatures is often referred to as *validation parameters*. When one - time signatures are combined with techniques for authenticating validation parameters, multiple signatures are possible. Most, but not all, one - time digital signature schemes have the advantage that signature generation and verification are very efficient. One - time digital signature schemes are useful in applications such as chip cards, where low computational complexity is required.

## 2.3 The original Merkle digital signature

2.3.1 **Introduction:** In [MER79] Merkle proposed a digital signature scheme that was based on both one - time signatures and hash functions, 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. Every node has a  $k$  bit value and every interior node's value is a hash function of the node values of its children. The public key that helps for authentication is placed as root of the tree. The one - time private keys are used for generation of the leaves and are authenticated with the help of the root. Although the number of messages that

can be signed is limited by the size of the tree, the tree can be made arbitrarily large.

2.3.2 **The Lamport-Diffie one-time signature scheme:** In [DH76] Whitfield Diffie and Martin E. Hellman present a new digital signature, based on hash functions. Such function  $Y = f(X)$  is selected. Each user U chooses  $2n$  random values  $X_0, X_1, \dots, X_{2n-1}$  and computes  $Y_0, Y_1, \dots, Y_{2n-1}$  by  $Y_i = f(X_i)$ . Then U publishes the vector  $Y = (Y_0, Y_1, \dots, Y_{2n-1})$  in a public file under his name (i.e., in a newspaper or in a public file maintained by a trusted center). He can publish as many vectors as the number of signatures he is expected to sign. Now we come to signature generation [EB05]. Alice wants to sign an  $n$  - bit message  $M$  to Bob ( $M = m_0 m_1 \dots m_{n-1}$ ). She then chooses one of his unused vectors from the public file and sends it to Bob. Bob verifies the existence of the vector in the public file. After that Alice and Bob mark the vector as used in the specific file. Alice computes the signature  $S = S_0 S_1 \dots S_{n-1}$  by

$$S_i = \begin{cases} X_{2i}, & \text{if } m_i = 0 \\ X_{2i+1}, & \text{if } m_i = 1 \end{cases}$$

and sends it to Bob. To verify the signature, Bob computes for all  $i$ - s

$$f(S_i) = \begin{cases} Y_{2i}, & \text{if } m_i = 0 \\ Y_{2i+1}, & \text{if } m_i = 1 \end{cases}$$

Talking about the security of the signature scheme we claim that if Bob can invert the hash function  $f$ , he can then forge Alice's signature. Even if he is given a signature of some message using some vector, he still needs to invert the hash function  $f$  in order to forge a different message using the same vector. To make this easier to understand we give the following example:

**Example:** Let  $X = 01001101$  and  $f(X)$  be a function that just changes 0-s with 1-s and vice versa. Then  $Y = 10110010$ . Imagine that Alice wants to sign two messages  $M_1 = 1010$  and  $M_2 = 1101$  with this very same private key  $X$ . Then, according to the definition of the scheme, Alice computes the corresponding signatures  $S_1 = 1010$  and  $S_2 = 1011$ , and sends them to Bob. Knowing  $M_1$ ,  $M_2$ ,  $S_1$  and  $S_2$  it is extremely easy for Bob to forge a new message  $M_3 = 1111$  with its corresponding signature  $S_3 = 1011$ , just combining the previous two.

2.3.3 **Winternitz improvement:** One generalization of the Lamport - Diffie scheme attributed by Merkle to Winternitz in [MER79] is to apply the hash function  $f$  to the secret key iteratively a fixed number of times, resulting in the public key. Briefly the scheme works as follows according to [ECS]. Suppose we wish to sign a  $n$ -bit message  $M$ . First the message is split into  $n/t$  blocks of size  $t$  bits. Let these parts be  $M_1, M_2, \dots, M_{n/t}$ . The secret key is  $sk = \{x_0, \dots, x_{n/t}\}$  where  $x_i$  is a  $l$ -bit value. The public key is

$$pk = \left\{ f^{(2^t-1)n/t}(x_0) \parallel f^{2^t-1}(x_1) \parallel \dots \parallel f^{2^t-1}(x_{n/t}) \right\},$$

where  $f^2(x) = f(f(x))$  is applying  $f$  iteratively.

The signature of a message  $M$  is computed by considering the integer value of the blocks  $Int(M_i) = I_i$ . The signature  $S$  is composed of  $n/t + 1$  values  $\{s_0, \dots, s_{n/t}\}$  where, for  $i \geq 1$

$$s_i = f^{2^t-1-I_i}(x_i) = f^{-I_i}(y_i), \text{ and } s_0 = f^{\sum_i I_i}(x_0)$$

for  $1 \leq i \leq n/t$ . The signature length is  $l(n/t + 1)$ . On average, computing a signature requires  $2 \frac{2^t m}{t}$  evaluations of  $f$ . To verify a signature, one splits the message  $M$  in  $n/t$  blocks of size  $t$ -bits. Let these parts be  $M_1, \dots, M_{n/t}$ . One then verifies that  $pk$  is equal to

$$\left\{ f^{2^t-1-\sum_i I_i}(s_0) \parallel f^{I_1}(s_1) \parallel \dots \parallel f^{I_{n/t}}(s_{n/t}) \right\} \text{ for } 1 \leq i \leq n/t. \text{ It is possible}$$

to prove that forging a signature of a message  $M'$  given a message  $M$ , forging a valid signature  $S$  and the public key, requires inversion of the function  $f$ .

2.3.4 **Merkle trees:** The greatest disadvantage of Lamport - Diffie scheme is the size of the public key. All the verifiers need an authenticated copy of this public key in order to verify the validity of any signature. In [MER79] Merkle proposed the use of binary trees to authenticate a large number of public keys with a single value, namely the root of the tree. That is how the definition of a Merkle tree comes into use. It is a complete binary tree with a  $k$ -bit value associated to each node such that the interior node value is a hash function of the node values of its children (Figure 1):

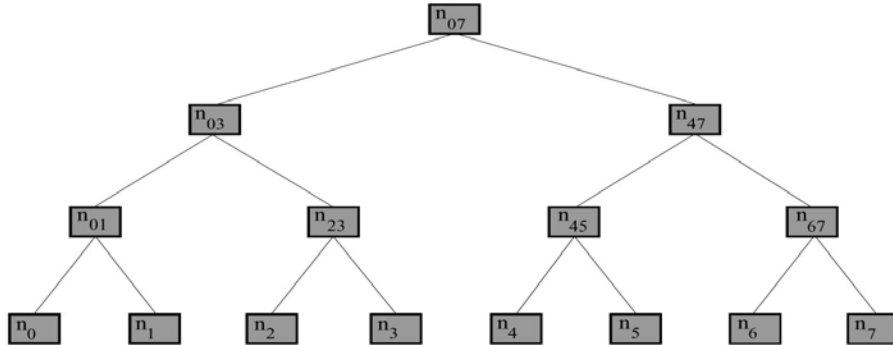
$$P[i, j] = f\left(\langle P[i, (i + j - 1) / 2] \| P[(i + j + 1) / 2, j] \rangle\right).$$

$P$  is the assignment, which maps the set of nodes to the set of their strings of length  $k$ . In other words, for any interior node  $n_{parent}$  and its two child nodes  $n_{left}$  and  $n_{right}$ , the assignment  $P$  is required to satisfy:

$$P(n_{parent}) = f\left(P(n_{left}) \| P(n_{right})\right)$$

The  $N$  values that need to be authenticated are placed at the  $N$  leaves of the tree. We may choose the leaf value arbitrarily, but usually it is a cryptographic hash function of the values that need to be authenticated. In this case these values are called *leaf - preimages*. A leaf can be verified with respect to a publicly known root value and its *authentication path*. We assume that the public keys of the Lamport - Diffie one- time signature scheme are stored at the leaf - preimages of the tree (one public key per leaf - preimage).

2.3.5 **Authentication paths:** Let us set  $Auth_h$  to be the value of the sibling of the node of height  $h$  on the path from the leaf to the root. Every leaf has height 0 and the hash function of two leaves has height 1, etc. In this manner the root has height  $H$  if the tree has  $2^H = N$  leaves. Then for every leaf the authentication path is the set  $\{Auth_h | 0 \leq h < H\}$ . So a leaf can be authenticated as follows: First we apply our hash function  $f$  to the leaf and its sibling  $Auth_0$ , then we apply  $f$  to the result and  $Auth_1$ , and so on all the way up to the root. If the calculated root value is the same as the published root value, then the leaf value is accepted as authentic. This operation requires  $\log_2(N)$  invocations of the hash function  $f$ . For detailed information see [ECS].



**Figure 1:** Merkle tree with 8 leaves. The root can be used to authenticate the complete tree.

## Chapter 3

# Tree traversal techniques

### 3.1 Authentication path regeneration

As Michael Szydlo mentioned in his paper [SZY04], the main purpose of Merkle tree traversal is to sequentially output the leaf values of all nodes, together with the associated authentication data. In our third chapter we will discuss three algorithms. First comes the classic Merkle tree traversal algorithm – a straightforward technique which requires a maximum of  $2\log_2(N)$  invocations of the hash function  $f$  per round and also a maximum storage of  $\log_2^2(N)/2$  outputs of  $f$ . This algorithm is presented for the first time in 1979 by Ralph Merkle [MER79]. The second described algorithm is originally presented in [JLMS03] and is mainly distinguished because it allows tradeoffs between time and space. Thus storage can either be minimized or maximized. In the first case the algorithm requires about  $2\log_2(N)/\log_2(\log_2(N))$  invocations of  $f$ . In the second -  $1.5\log_2^2(N)/\log_2(\log_2(N))$  outputs of  $f$  are required. The third algorithm appears first in [SZY04] and requires  $2\log_2(N)$  time and a maximum storage of  $3\log_2(N)$  outputs of  $f$ . Some of the described algorithms need a technique to effectively calculating node values and the root value. These calculations require  $N-1$  invocations of  $f$ , that are not included in the above mentioned time and storage requirements. They are done by the so called *TREEHASH* algorithm, which will be explained in the next paragraph.

### 3.2 Efficient nodes 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 [L02]). At first the stack is empty and then we start adding leaf values one by one onto it (starting from leaf with index zero). Every round we run a check if the last two values on the stack are of nodes of the same height or not. In the first case we calculate the value of the parent node and add it onto the stack, after removing

the two children values first. In the second case we continue adding leaf values until we reach the first case. The intermediate values stored in the stack during the execution of the algorithm are called *tail node* values and create the *tail* of the stack. This simple implementation is very effective, because for the total of  $2^{H+1} - 1$  rounds (one for each node) it requires a maximum of  $2^H - 1$  computational units for a Merkle tree of maximal height  $H$ . In this number are included the calculations of the leaf values which are done by an oracle *LEAFCALC* (hash computations and *LEAFCALC* computations are counted equally). In the algorithm the leaf indexes are denoted by *leaf* which is incremented every round. So  $LEAFCALC(leaf)$  is simply the value of the leaf with index *leaf*. Another important thing about the algorithm is that it stores a maximum of  $H + 1$  hash values every round, thus conserving space. This is because node values are discarded when they are not needed any more, which is after their parent value is calculated. Now we present the algorithm itself:

**Algorithm 2.1: TREEHASH (start, maxheight)**

- 1 Set  $leaf = start$  and create empty stack.
- 2 **Consolidate:** If top 2 nodes on the stack are equal height:
  - Pop node value  $P(n_{right})$  from stack.
  - Pop node value  $P(n_{left})$  from stack.
  - Compute  $P(n_{parent}) = f\left(P(n_{left}) \parallel P(n_{right})\right)$ .
  - If height of  $P(n_{parent}) = maxheight$ , output  $P(n_{parent})$ .
  - Push  $P(n_{parent})$  onto the stack and stop.
- 3 **New Leaf:** Otherwise:
  - Compute  $P(n_l) = LEAFCALC(leaf)$ .
  - Push  $P(n_l)$  onto the stack.
  - Increment  $leaf$ .
- 4 Loop to step 2.

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 -  $Stack_h.initialize(startnode, h)$  which tells us that we will modify the  $Stack_h$  stack starting with the leaf of index *startnode* and going up to height  $h$  ( $h \leq H$ ). 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 -  $Stack_h.update(2)$ . We use the same stack as in the previous example. It means that for the already known  $Stack_h$  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 [MER79] Ralph Merkle presents a new tree traversal technique, that later becomes a basis for further work and improvements. 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  $\{Auth_i\}$ . 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.

Now we are going to give a brief explanation of the classic traversal algorithm phase by phase. The notations that we need are not much – the current authentication nodes are  $Auth_h$ . As we need to implement the *TREEHASH* in our algorithm, we create an object  $Stack_h$  with a stack of node values. We already explained the two methods *initialize* and *update* in the previous section so we have an idea how  $Stack_h.initialize$  and  $Stack_h.update$  are used.

3.3.1 **Key Generation and Setup:** Let  $n_{h,j}$  be the  $j$ -th node of height  $h$  in our Merkle tree. Thus  $Auth_h = P(n_{h,1})$  are the values of the very first authentication path ( $h = 0, 1, \dots, H-1$ ), which means they are right nodes. The upcoming authentication nodes at height  $h$  are  $P(n_{h,0})$  and they are left nodes. The algorithm is a direct application of the *TREEHASH* algorithm, every node value is computed, but only the described values are stored. In the first step we

store the node values included in the authentication path of the first leaf. In the second step we store  $P(n_{h,0})$  for every  $h = 0, 1, \dots, H - 1$  and we say  $Stack_h$  contains only the value  $P(n_{h,0})$ . In the third step we simply store and output the root value. Here is the algorithm itself:

**Algorithm 3.1: Key Generation and Setup**

- 1 Initial Authentication Nodes:** For each  $h \in \{0, 1, \dots, H - 1\}$ :  
Calculate  $Auth_h = P(n_{h,1})$ .
- 2 Initial Next Nodes:** For each  $h \in \{0, 1, \dots, H - 1\}$ : Set up  $Stack_h$  with the value of the sibling of  $Auth_h - P(n_{h,0})$  (left nodes).
- 3 Public Key:** Calculate and publish tree root value  $P(n_{root})$ .

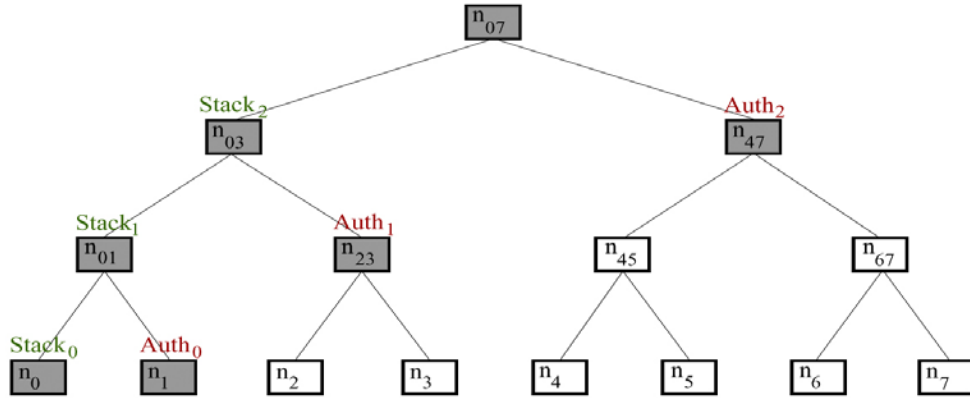
After this phase of the classic Merkle tree traversal we start the next one with already computed leftmost nodes and their siblings.

3.3.2 **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. The authentication node at height  $h$  ( $h = 0, 1, \dots, H - 1$ ) needs to be shifted to the right if  $2^h$  divides  $leaf + 1$  without remainder. The new authentication nodes take their values from the stacks, created in the key generation part. After  $Auth_h$  is changed, we alter also the contents of  $Stack_h$  by using the  $Stack_h.initialize$  and  $Stack_h.update$  methods, thus making further modifications possible. We notice that at round  $leaf + 1 + 2^h$  the authentication path will pass through the  $(leaf + 1 + 2^h) / 2^h$ -th node at height  $h$ . The *startnode* value of the initialization process (and future  $Auth_h$ ) comes from the  $2^h$  leaf values, starting with the leaf with index  $(leaf + 1 + 2^h) \oplus 2^h$ , where  $\oplus$  is a bitwise XOR. Now we present the exact algorithm:

### Algorithm 3.2: Classic Merkle Tree Traversal

<p><b>1</b> Set <math>leaf = 0</math></p> <p><b>2 Output:</b></p> <ul style="list-style-type: none"> <li>• Compute and output <math>P(n_{leaf}) = LEAFCALC(leaf)</math>.</li> <li>• For each <math>h \in [0, H-1]</math> output <math>\{Auth_h\}</math>.</li> </ul> <p><b>3 Refresh Authentication Nodes:</b> For all <math>h</math> such that <math>2^h/leaf + 1</math>:</p> <ul style="list-style-type: none"> <li>• Let <math>Auth_h</math> become equal to the only node value in <math>Stack_h</math>. Empty the stack.</li> <li>• Set <math>startnode = (leaf + 1 + 2^h) \oplus 2^h</math>.</li> <li>• <math>Stack_h.initialize(startnode, h)</math>.</li> </ul> <p><b>4 Build Stacks:</b> For all <math>h \in [0, H-1]</math>:</p> <ul style="list-style-type: none"> <li>• <math>Stack_h.update(2)</math></li> </ul> <p><b>5 Loop:</b></p> <ul style="list-style-type: none"> <li>• Set <math>leaf = leaf + 1</math>.</li> <li>• If <math>leaf &lt; 2^H - 1</math> go to step 2, otherwise stop.</li> </ul>
---

3.3.3 **Example for the classic traversal:** To explain more thorough the way this main algorithm works, we give an example, which follows the algorithm step by step for a Merkle tree of maximal height 3 and thus with  $N = 2^3$  leaves  $n_0, \dots, n_7$ . First we start with algorithm 3.1, which computes  $Auth_h$  and puts its sibling onto  $Stack_h$  for  $h \in \{0, 1, 2\}$ . Thus, in the very beginning  $Auth_0 = P(n_1)$ ,  $Auth_1 = P(n_{23})$ ,  $Auth_2 = P(n_{47})$ . The single value in  $Stack_0$  is  $P(n_0)$ , in  $Stack_1$  is  $P(n_{01})$  and in  $Stack_2$  is  $P(n_{03})$ . So we start the classic traversal with some precomputed nodes, which are denoted by grey boxes in the pictures (see figure 2.1):

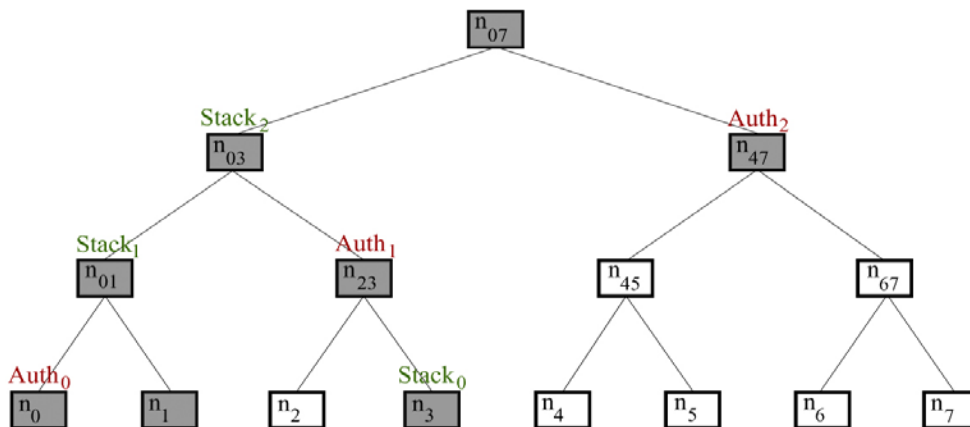


**Figure 2.1: The tree before the algorithm.**

**Step 1:**  $leaf = 0$ .

- $P(n_0)$  with its authentication path  $\{Auth_0, Auth_1, Auth_2\} = \{P(n_1), P(n_{23}), P(n_{47})\}$  are output.
- $h=0$  is the only solution of  $2^h / leaf + 1$ , so  $Auth_0$  becomes  $P(n_0)$ , because it is the only value in  $Stack_0$ .  $Stack_0$  becomes empty.  $startnode = 3$  so we run  $Stack_0.initialize(3, 0)$  as described in 3.2.
- For  $h=0$  we run  $Stack_0.update$ , which only computes  $P(n_3)$ , puts it onto the stack and stops.

So the tree after the first round of the algorithm looks like the one in figure 2.2:



**Figure 2.2: The tree after round 1.**

**Step 2:**  $leaf = 1$ .

- $P(n_1)$  with its authentication path  $\{Auth_0, Auth_1, Auth_2\} = \{P(n_0), P(n_{23}), P(n_{47})\}$  are output.
- $2^h / leaf + 1$  has two solutions so we consider both cases:

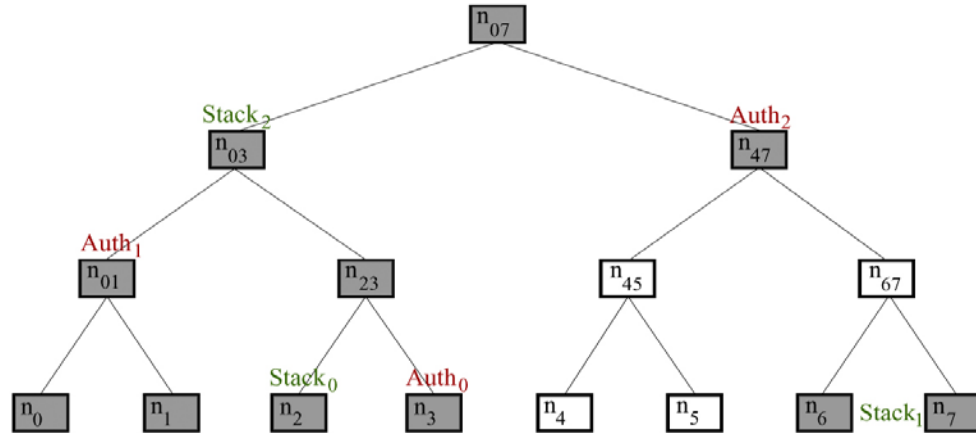
$h = 0$ :

- $Auth_0$  becomes  $P(n_3)$ , because it is the only value in  $Stack_0$ .  $Stack_0$  becomes empty.  $startnode = 2$  so we run  $Stack_0.initialize(2, 0)$ .
- For  $h = 0$  we run  $Stack_0.update$ , which only computes  $P(n_2)$ , puts it onto the stack and stops.

$h = 1$ :

- $Auth_1$  becomes  $P(n_{01})$ , because it is the only value in  $Stack_1$ .  $Stack_1$  becomes empty.  $startnode = 6$  so we run  $Stack_1.initialize(6, 1)$ .
- For  $h = 1$  we run  $Stack_1.update$  two times. It computes  $P(n_6)$  and  $P(n_7)$ , puts them onto the stack and stops.

So the tree after the second round looks like the one in figure 2.3:



**Figure 2.3: The tree after round 2.**

**Step 3:**  $leaf = 2$ .

- $P(n_2)$  with its authentication path  $\{Auth_0, Auth_1, Auth_2\} = \{P(n_3), P(n_{01}), P(n_{47})\}$  are output.
- $h = 0$  is the only solution of  $2^h / leaf + 1$ , so  $Auth_0$  becomes  $P(n_2)$ , because it is the only value in  $Stack_0$ .  $Stack_0$  becomes empty.  $startnode = 5$  so we run  $Stack_0.initialize(5, 0)$ .

- For  $h = 0$  we run  $Stack_0.update$ , which only computes  $P(n_5)$ , puts it onto the stack and stops. For  $h = 1$  we run  $Stack_1.update$ , which removes the previous two values from the stack, computes  $P(n_{67})$  and puts it onto the stack.

So the tree after the third round looks like the one in figure 2.4:

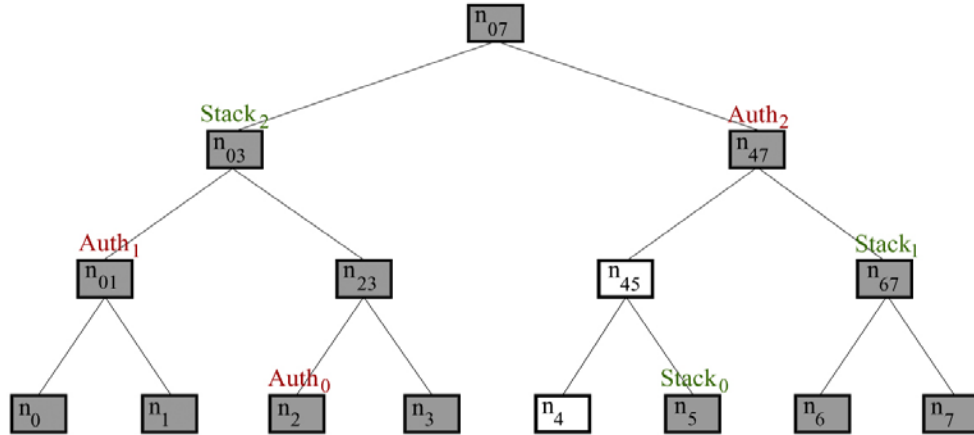


Figure 2.4: The tree after round 3.

**Step 4:**  $leaf = 3$ .

- $P(n_3)$  with its authentication path  $\{Auth_0, Auth_1, Auth_2\} = \{P(n_2), P(n_{01}), P(n_{47})\}$  are output.
- $2^h / leaf + 1$  has three solutions so we consider three cases:

Case 1:  $h = 0$  :

- $Auth_0$  becomes  $P(n_5)$ , because it is the only value in  $Stack_0$ .  $Stack_0$  becomes empty.  $startnode = 4$  so we run  $Stack_0.initialize(4, 0)$ .
- For  $h = 0$  we run  $Stack_0.update$ , which only puts  $P(n_4)$  onto the stack and then stops.

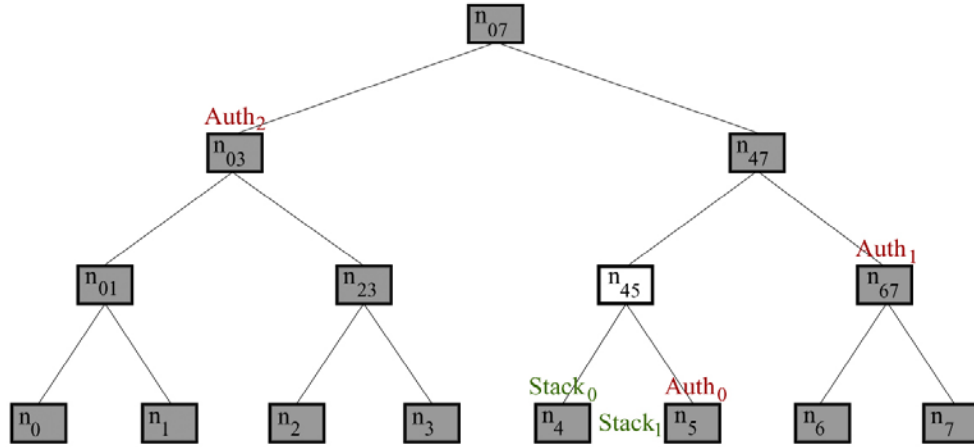
Case 2:  $h = 1$  :

- $Auth_1$  becomes  $P(n_{67})$ , because it is the only value in  $Stack_1$ .  $Stack_1$  becomes empty.  $startnode = 4$  so we run  $Stack_1.initialize(4, 1)$ .
- For  $h = 1$  we run  $Stack_1.update$  two times. First it computes  $P(n_4)$  and puts it onto the stack, then it puts  $P(n_5)$  onto the stack and stops.

Case 3:  $h = 2$  :

- $Auth_2$  becomes  $P(n_{03})$ , because it is the only value in  $Stack_2$ .  $Stack_2$  becomes empty.  $startnode = 13$ , which is too big, so stop.

So the tree, after the fourth round, looks like the one in figure 2.5:



**Figure 2.5: The tree after round 4.**

**Step 5:**  $leaf = 4$ .

- $P(n_4)$  with its authentication path  $\{Auth_0, Auth_1, Auth_2\} = \{P(n_5), P(n_{67}), P(n_{03})\}$  are output.
- $h=0$  is the only solution of  $2^h / leaf + 1$ , so  $Auth_0$  becomes  $P(n_4)$ , because it is the only value in  $Stack_0$ .  $Stack_0$  becomes empty.  $startnode = 7$  so we run  $Stack_0.initialize(7, 0)$ .
- For  $h=0$  we run  $Stack_0.update$ , which only puts  $P(n_7)$  onto the stack and stops. For  $h=1$  we run  $Stack_1.update$ , which removes the previous two values from the stack, computes  $P(n_{45})$  and puts it onto the stack, then stops.

So the tree, after the fifth round, looks like the one in figure 2.6:

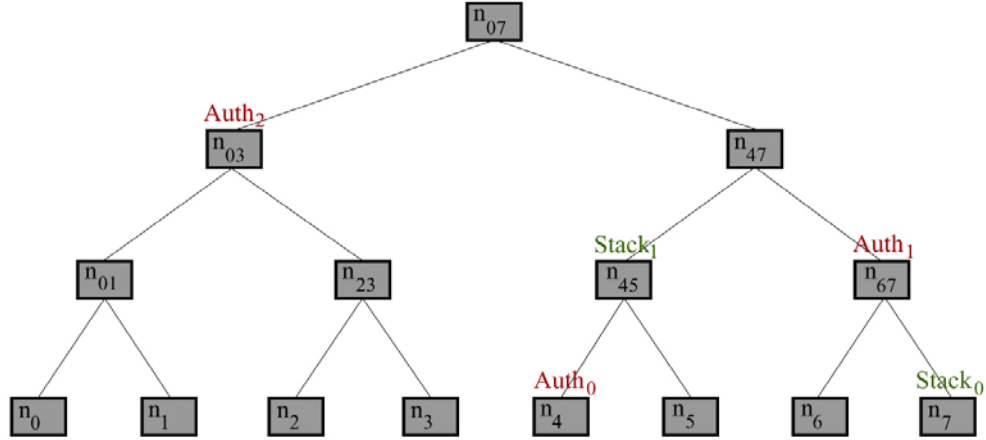


Figure 2.6: The tree after round 5.

**Step 6:**  $leaf = 5$ .

- $P(n_5)$  with its authentication path  $\{Auth_0, Auth_1, Auth_2\} = \{P(n_4), P(n_{67}), P(n_{03})\}$  are output.
- $2^h / leaf + 1$  has two solutions so we consider both cases:

Case 1:  $h = 0$ :

- $Auth_0$  becomes  $P(n_7)$ , because it is the only value in  $Stack_0$ .  $Stack_0$  becomes empty.  $startnode = 6$  so we run  $Stack_0.initialize(6, 0)$ .
- For  $h = 0$  we run  $Stack_0.update$ , which only puts  $P(n_6)$  onto the stack and stops.

Case 2:  $h = 1$ :

- $Auth_1$  becomes  $P(n_{45})$ , because it is the only value in  $Stack_1$ .  $Stack_1$  becomes empty.  $startnode = 10$ , which is too big, so stop.

So the tree, after the sixth round, looks like the one in figure 2.7:

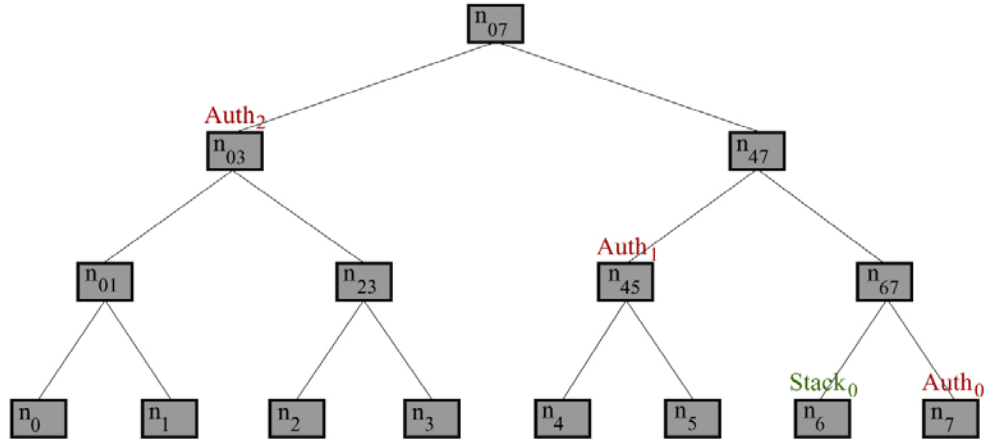


Figure 2.7: The tree after round 6.

**Step 7:**  $leaf = 6$ .

- $P(n_6)$  with its authentication path  $\{Auth_0, Auth_1, Auth_2\} = \{P(n_7), P(n_{45}), P(n_{03})\}$  are output.
- $h=0$  is the only solution of  $2^h / leaf + 1$ , so  $Auth_0$  becomes  $P(n_6)$ , because it is the only value in  $Stack_0$ .  $Stack_0$  becomes empty.  $startnode = 9$ , which is too big, so stop.

So the tree, after the sixth round, looks like the one in figure 2.8:

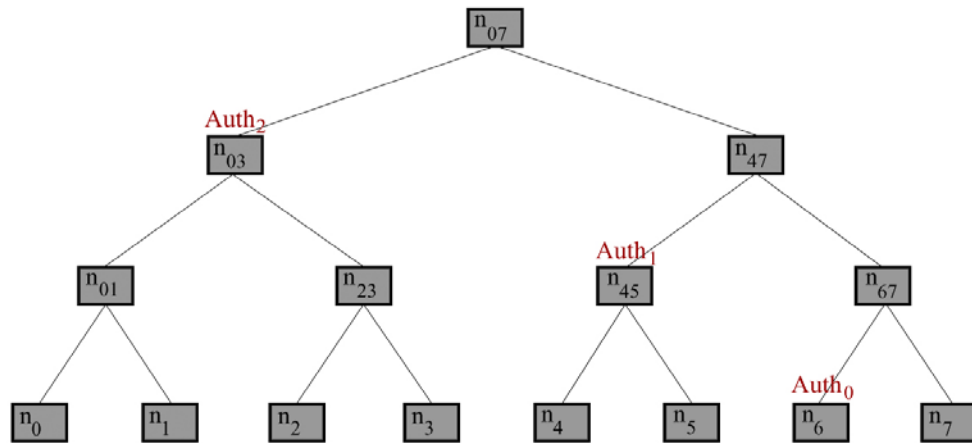


Figure 2.8: The tree after round 7.

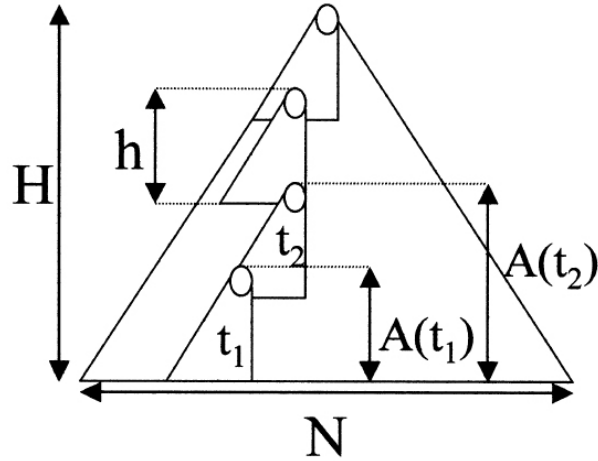
**Step 8:**  $leaf = 7$ .

- $P(n_7)$  with its authentication path  $\{Auth_0, Auth_1, Auth_2\} = \{P(n_6), P(n_{45}), P(n_{03})\}$  are output.

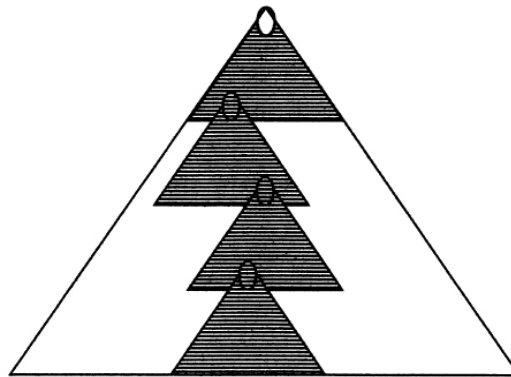
## 3.4 Fractal tree representation and traversal

In 2003 a group of people introduce a new traversal technique with better space and time complexity than any other published algorithm [JLMS03]. Most important about it are two things. The so called fractal traversal is very different from the other two described techniques in this thesis and has the advantage to allow tradeoffs between storage and computation (this will be explained more thorough in the comparison chapter after the algorithm descriptions). We will just mention here that the fractal algorithm requires a maximum of  $2 \log(N) / \log(\log(N))$  hash function evaluations for every output (in its worst case) and maximum storage of  $1.5 \log^2(N) / \log(\log(N))$  hash values. In this part we are going to give a brief description of this Merkle tree traversal technique, but first we need to introduce some notations used in it. As usual, we consider a Merkle tree  $T$  of height  $H$  and  $2^H$  leaves.

**3.4.1 Notations:** The basic idea of the fractal algorithm is that it uses subtrees of  $T$  instead of single nodes (like the authentication nodes in the previous algorithm). That is why we need to choose a fixed height  $h$  ( $1 \leq h \leq H-1$ ) and we call it *subtree height* (for our algorithm we need  $h$  to be a divisor of  $H$ ). Let  $n$  be an arbitrary node in  $T$ . We say that  $n$  has *altitude*  $h$  if the length of the path from  $n$  to the nearest leaf is equal to  $h$ . For a node  $n$  in our tree we can define an  *$h$ -subtree at  $n$* , which is exactly the subtree that has  $n$  for root and has height  $h$ . As  $h$  divides  $H$ , we can also define  $L$  to be the number of levels of  $h$ -subtrees in  $T$  (thus  $L = H/h$ ). So if the root of a subtree has altitude  $ih$  for some  $i = 1, 2, \dots, L$  we say that the subtree is at level  $i$  (there are  $2^{H-ih}$   $h$ -subtrees at each level  $i$ ). A given series of  $h$ -subtrees  $\{subTree_j\}$  with  $j \in [1, \dots, L]$  is called *stacked* if the root of  $subTree_j$  is a leaf of  $subTree_{j+1}$  (see figures 3.1 and 3.2 below).



**Figure 3.1:** The height of the Merkle tree is  $H$ . Thus the number of leaves is  $N = 2^H$ . The height of each subtree is  $h$ . The altitudes  $A(t_1)$  and  $A(t_2)$  of the subtrees  $t_1$  and  $t_2$  are marked.

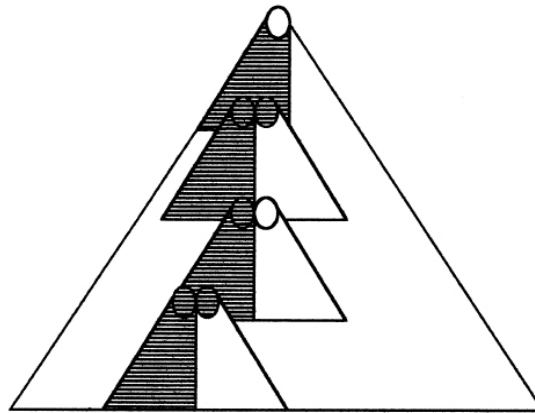


**Figure 3.2:** Instead of storing all tree nodes, we store a smaller set – those within the stacked subtrees. The leaf whose pre – image will be output next is contained in the lowest – most subtree. The entire authentication path is contained in the stacked set of subtrees.

3.4.2 **Existing and desired subtrees:** In the process of the algorithm we always have a stacked series of  $L$  subtrees, called  $Exist_i$ . These existing subtrees are always precomputed, which means that the values of all of their nodes are already calculated and stored (except of the roots) in the beginning of the algorithm. Their role in the traversal is very important – they simply contain the authentication path of every leaf of the lowest subtree  $Exist_1$  (the idea of the algorithm is that  $\{Exist_i\}$  is constructed and after that modified in order to

contain every leaf and its authentication path that have to be output). In the very beginning of the fractal algorithm the set of existing subtrees is always the left-most one in  $T$  (like on figure 3.3).

Another set of subtrees we are going to need is called the *set of desired subtrees* -  $\{Desire_i\}$ ,  $i = 1, 2, \dots, L - 1$  (there is no desired subtree at level  $L$ , because the only possible subtree at this level is an existing subtree). Notice that this set is not necessary stacked. The positioning of the desired subtrees is easy to understand – if  $Exist_i$  (which is already built) has root  $n_k$ , then the root of the corresponding desired tree  $Desire_i$  is  $n_{k+1}$  (exactly the next node to the right on the row).



**Figure 3.3:** The grey subtrees correspond to the existing subtrees, while the white subtrees correspond to the desired ones. As the existing subtrees are used up, the desired subtrees are gradually constructed.

The values of the nodes of our desired subtrees need to be calculated and stored during the algorithm. This, of course, is achieved by using a slightly modified version of the *TREEHASH* algorithm, implemented into the main one. The differences with the original *TREEHASH* are two. First, the modified one stops one round earlier, which means that the root of the current  $Desire_i$  is not evaluated. Second, the value of every node of height smaller than  $ih$  is saved into  $Desired_i$ . Now we are going to present the main algorithm itself.

**3.4.3 Algorithm presentation:** Just like all traversal techniques, the fractal traversal one has also three phases – key generation, output and verification, which are precisely the same as the phases of the classic algorithm. An interesting thing is that the results of the output phase of all described algorithms are the same, which is the reason not to describe the verification phase.

### Key Generation and Setup phase:

In this part of the algorithm the set of existing subtrees is defined for a chosen subtree height  $h$ , as already mentioned it consists only of the leftmost  $h$ -subtrees in  $T$ . After that the node values of all existing subtrees are precomputed, except of the roots (meaning that the output phase starts with already precomputed existing subtrees). A set of desired subtrees is also created as described in the previous part (the desired subtrees are the right neighbors of the existing subtrees on the particular level). Every desired subtree is initialized with an empty stack which makes the use of *TREEHASH* possible. We define the starting node of the *TREEHASH* algorithm for  $Desire_i$  to be the leaf with index  $Desire_i.position = 2^{ih}$  ( $i = 1, 2, \dots, L-1$ ). At the end the tree root is published. Here is the algorithm step by step:

#### Algorithm 4.1: Key Generation and Setup

- 1 Initial Subtrees:** For each  $i \in \{1, 2, \dots, L\}$ :
  - Calculate all non – root pebbles in existing subtree at level  $i$ .
  - Create new empty desired subtree at each level  $i$  (except for  $i = L$ ), with leaf position initialized to  $2^{ih}$ .
- 2 Public Key:** Calculate and publish tree root.

### Output and Update Phase:

This part of the algorithm has exactly  $N = 2^H$  rounds, one for each leaf. We use a counter *leaf* that denotes the index of the current leaf ( $leaf = 0, 1, \dots, N-1$ ) and it is regularly increased by one at the end of every round of the algorithm. In the very beginning of every round the value of the current leaf together with its authentication path are output. These outputs are possible because they are part of the already computed nodes of the existing subtrees.

We say that an existing subtree is no longer useful when it does not contain future needed authentication paths. In this case that subtree “dies” and the corresponding desired subtree becomes the new existing one (this means that by this moment the corresponding desired subtree is fully evaluated to become existing). The calculated values in the dead subtree are discarded and a new desired subtree is created to the right of the new existing one. An existing subtree is shifted to the right (taking the place of the corresponding desired subtree) only if its desired subtree is completely calculated. After the shift, it contains the next needed authentication paths. We notice that the existing subtree at level  $i$  (namely  $Exist_i$ ) is shifted to the right every  $2^{ih}$  rounds. From that follows that  $Desire_i$  should be completed before round  $2^{ih}$  to be able to become an existing subtree. This is achieved by applying two units of

*TREEHASH* computation to every desired subtree every round. Here comes the algorithm itself:

**Algorithm 4.2: Fractal Merkle Tree Traversal**

- 1 Set  $leaf = 0$ .
- 2 **Output:** Authentication path for leaf with index  $leaf$ .
- 3 **Next Subtree:** For each  $i$  for which  $Exist_i$  is no longer needed, i.e.  $i \in \{1, 2, \dots, L-1\}$  such that  $leaf = 2^{hi} - 1 \pmod{2^{hi}}$ :
  - Remove pebbles in  $Exist_i$ .
  - Rename tree  $Desire_i$  as tree  $Exist_i$ .
  - Create new, empty tree  $Desire_i$  (if  $leaf + 2^{hi} < 2^H$ ).
- 4 **Grow Subtrees:** For each  $i \in \{1, 2, \dots, L-1\}$ : Grow tree  $Desire_i$  by applying two units to modified *TREEHASH* (unless  $Desire_i$  is completed) starting from leaf with index  $2^{ih}$ .
- 5 Increment  $leaf$  and loop back to step 2 (while  $leaf < 2^H - 1$ ).

3.4.4 **Example for the fractal algorithm:** To explain more thorough the way this main algorithm works, we give an example, which follows the algorithm step by step for a Merkle tree of maximal height  $L=3$  and thus with  $N=2^3$  leaves  $n_0, \dots, n_7$ . First we start with algorithm 4.1, which computes the nodes of all existing subtrees and puts creates empty desired subtrees as on the picture. Thus, in the very beginning  $Auth_0 = P(n_1)$ ,  $Auth_1 = P(n_{23})$ ,  $Auth_2 = P(n_{47})$ .  $Exist_1$  consists of  $n_0$ ,  $n_1$  and  $n_{01}$ .  $Exist_2$  consists of  $n_{01}$ ,  $n_{23}$  and  $n_{03}$ . The last existing subtree is  $Exist_3$  and it consists of  $n_{03}$ ,  $n_{47}$  and the root  $n_{07}$ .  $h$  is always one, so the height of all subtrees is equal to one. Now we start the output phase with the precomputed nodes, which are denoted by grey boxes in the pictures (see figure 4.1):

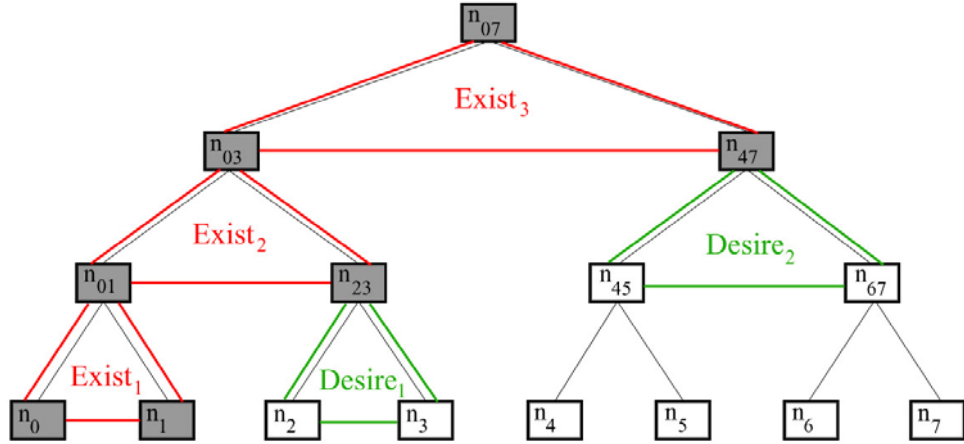


Figure 4.1: The tree before the algorithm.

Step 1:  $leaf = 0$ .

- $P(n_0)$  with its authentication path  $\{Auth_0, Auth_1, Auth_2\} = \{P(n_1), P(n_{23}), P(n_{47})\}$  are output.
- Both  $Exist_1$  and  $Exist_2$  are still needed for future authentication paths, so we go to the next stage.
- For  $i=1,2$  we grow the  $Desire_i$  subtree by applying 2 units of modified *TREEHASH* and thus we calculate  $P(n_2)$  and  $P(n_3)$  for  $Desire_1$  and  $P(n_4), P(n_5)$  for  $Desire_2$ .

So the tree after the first round of the algorithm looks like the one in figure 4.2:

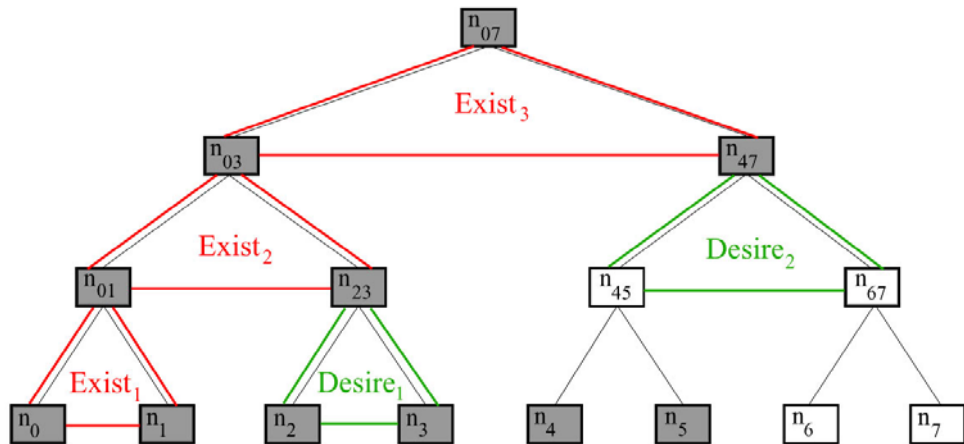
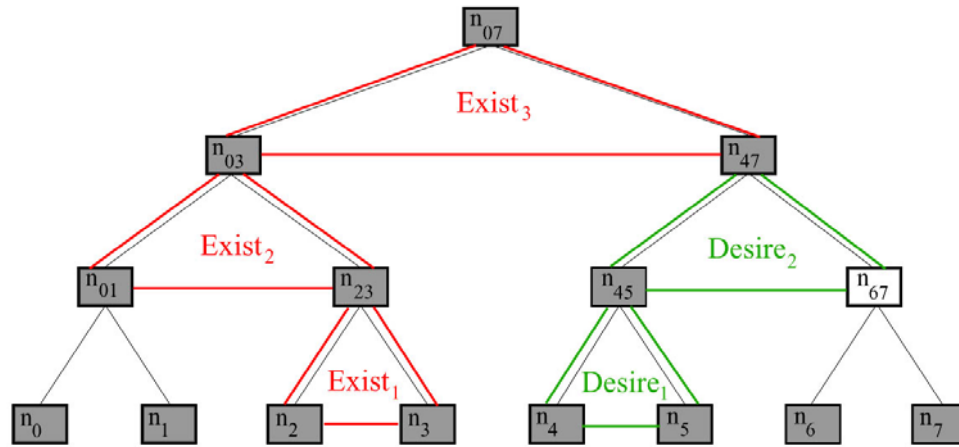


Figure 4.2: The tree after round 1.

Step 2:  $leaf = 1$ .

- $P(n_1)$  with its authentication path  $\{Auth_0, Auth_1, Auth_2\} = \{P(n_0), P(n_{23}), P(n_{47})\}$  are output.
- $Exist_1$  is no more needed for future authentication paths so we remove all pebbles in it.  $Exist_1$  becomes  $Desire_1$  and a new empty  $Desire_1$  is created as denoted on figure 4.3.
- We apply two units of modified *TREEHASH* to the new  $Desire_1$  and thus calculate  $P(n_{45})$ , which is now part of it. In  $Desire_2$  we calculate  $P(n_6)$  and  $P(n_7)$ .

So the tree after the second round looks like the one in figure 4.3:



**Figure 4.3: The tree after round 2.**

**Step 3:**  $leaf = 2$ .

- $P(n_2)$  with its authentication path  $\{Auth_0, Auth_1, Auth_2\} = \{P(n_3), P(n_{01}), P(n_{47})\}$  are output.
- There is no  $Exist_i$  which is not needed for future authentication paths to be output, so we go to the next stage.
- We apply two units of modified *TREEHASH* to  $Desire_2$  and thus calculate  $P(n_{67})$ .

So the tree after the third round looks like the one in figure 4.4.

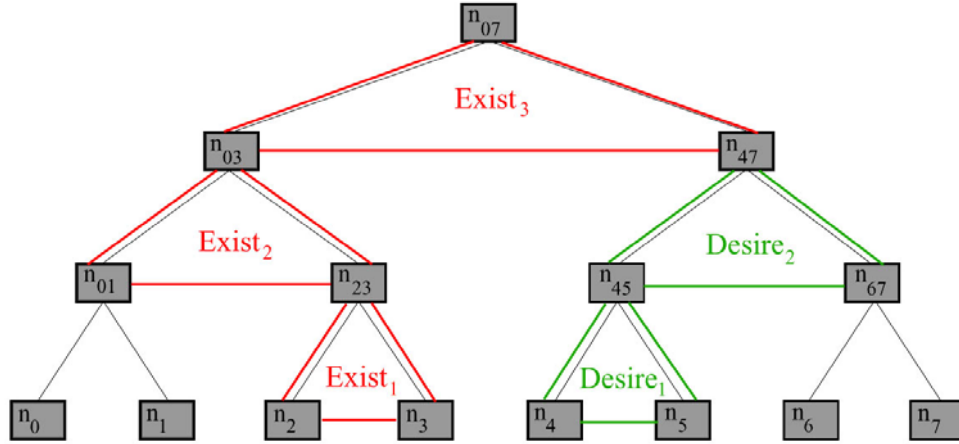


Figure 4.4: The tree after round 4.

**Step 4:**  $leaf = 3$ .

- $P(n_3)$  with its authentication path  $\{Auth_0, Auth_1, Auth_2\} = \{P(n_2), P(n_{01}), P(n_{47})\}$  are output.
- Both  $Exist_1$  and  $Exist_2$  are not needed for future authentication so we remove all pebbles in them.  $Exist_1$  becomes  $Desire_1$ ,  $Exist_2$  becomes  $Desire_2$  and a new empty  $Desire_1$  is created as denoted on figure 4.5.
- The new  $Desire_1$  is complete so we make no calculations.

So the tree, after the fourth round, looks like the one in figure 4.5:

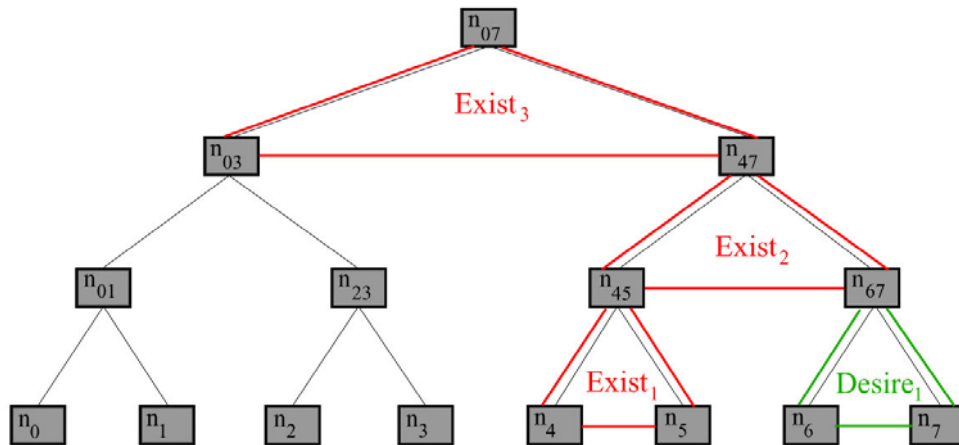


Figure 4.5: The tree after round 4.

**Step 5:**  $leaf = 4$ .

- $P(n_4)$  with its authentication path  $\{Auth_0, Auth_1, Auth_2\} = \{P(n_5), P(n_{67}), P(n_{03})\}$  are output.
- $Desire_1$  is still needed for future authentication so we do not change the tree.
- $Desire_1$  is complete so we do not make any calculations.

So the tree, after the fifth round, remains the same as figure 4.5:

In step 6 we shift  $Exist_1$  for last time and output  $P(n_5)$  with its authentication path. In steps 7 and 8 we only output  $P(n_6)$  and  $P(n_7)$  together with their authentication paths.

### 3.5 Logarithmic Merkle Tree Traversal

In [SZY04] Michael Szydlo presents an improvement of the classic Merkle tree traversal, called the *logarithmic traversal*. This improvement differs slightly from the original Merkle algorithm only in the output phase, more particularly, in the part where the stacks are built and updated. This means that the key generation and verification phases are absolutely the same as these of the classic traversal. That is why we are going to concentrate our attention only on the output phase of our new logarithmic algorithm. The upcoming authentication nodes that have to be output are computed just like in the classic technique, after that the stacks are initialized normally with a starting node and height. Now comes the difference in updating the stacks. In the classic algorithm all stacks receive equal attention, meaning that the same number of computations is spent for each one of them every step of the algorithm. Because of this, we can consider the updates of the stacks to be  $H$  different *TREEHASH* processes running in parallel. For large  $h$  values the *TREEHASH* computations are done a great number of rounds until the stack is completed, which requires many intermediate nodes to be stored at once. Now the purpose of the algorithm is to make these computations serially, which in general requires less space (the main idea is to compute the leftmost stacks first, because they will be needed for future outputs before the others). That is why in the new technique we want to complete  $Stack_h$  for the lowest height  $h$  ( $h \in [0, H - 1]$ ). If one of the stacks has a lower tail node than the others, but a bigger height, it is still the first stack to be updated and grown. To follow this idea we define  $l_{\min} = \min\{Stack_h.low\}$ , where  $Stack_h.low$  denotes the height of the lowest node in  $Stack_h$  ( $Stack_h.low = h$  if the stack is empty and  $Stack_h.low = \infty$  if the stack is complete). If two or more stacks have the same  $Stack_h.low$  value, we choose the stack with the smallest index  $h$  (which we call *focus*) and update it. Thus, after each step of the algorithm, no stacks should contain values of

nodes of height smaller than  $h$  (the stacks will then be completed according to the initializations made earlier in the same round). This is accomplished by applying up to  $2H - 1$  stack updates every step. Here is the logarithmic algorithm itself:

**Algorithm 5.1: Logarithmic Merkle Tree Traversal**

- 1** Set  $leaf = 0$
- 2 Output:**
  - Compute and output  $P(n_{leaf}) = LEAFCALC(leaf)$ .
  - For each  $h \in [0, H - 1]$  output  $\{Auth_h\}$ .
- 3 Refresh Authentication Nodes:**  
For all  $h$  such that  $2^h/leaf + 1$ :
  - Let  $Auth_h$  become equal to the only node value in  $Stack_h$ . Empty the stack.
  - Set  $startnode = (leaf + 1 + 2^h) \oplus 2^h$ .
  - $Stack_h.initialize(startnode, h)$ .
- 4 Build Stacks:**  
Repeat the following  $2H - 1$  times:
  - Let  $l_{min}$  be the minimum of  $\{Stack_h.low\}$ .
  - Let  $focus$  be the least  $h$  for which  $Stack_h.low = l_{min}$ .
  - $Stack_{focus}.update(1)$ .
- 5 Loop:**
  - Set  $leaf = leaf + 1$ .
  - If  $leaf < 2^H$  go to step 2, otherwise stop.

Because of the similarity between the Classic and Logarithmic algorithms we are not going to consider a special example for the latter.

## Chapter 4

# Algorithm Comparison

In this part of the thesis we will focus on the similarities and differences between the already described algorithms. Our main purpose is to consider both time and space requirements of every traversal technique from the previous chapter and to compare their efficiency in various situations. The size of the Merkle tree and the potentialities of the computer system we use are of course the main circumstances we need to consider. We are first going to start our arguments about the classic tree traversal and its improvement – the logarithmic one. The characteristics of the fractal traversal technique will be given aside from the other two, because of its totally different idea and structure. At the very end of this chapter we will give an example of the work of the three algorithms for Merkle trees of fixed sizes. Before we continue with the analysis of the traversals we should explain that in this chapter by using the term *operation* we mean elementary operation such as computation of a leaf value or hash function estimation.

4.1 **The classic traversal:** As described in the previous chapter, the classic Merkle tree traversal is based on the *TREEHASH* algorithm for more effective calculation of the tree nodes. For a tree of maximal height  $H$ , we run one instance of *TREEHASH* for every height  $h$  ( $0 \leq h < H$ ) and thus  $H$  different instances are updated in parallel every round of the algorithm. At height  $h$  we have exactly  $2^{H-h-1}$  right nodes and the same number of left ones. Of course we use *TREEHASH* to calculate the values of these nodes, and logically, the number of needed operations depends on  $h$ . For a node of height  $h$  the required number of *TREEHASH* operations is exactly  $2^{h+1} - 1$ . Thus the total amount of computations needed for a particular height  $h$  may be rounded up to  $2^{h+1} = 2N$ , which is exactly 2 computations per round (this is the reason why every *TREEHASH* instance is updated with two units of computation every round). If we add together the required operations for all heights  $h$  we will get the total number of required operations per round of the classic algorithm, which is  $2H = 2\log(N)$  operations.

Talking about the storage requirements of the classic algorithm we make the following arguments. As already explained, we may have up to  $H$  simultaneously running *TREEHASH* processes (one for every height  $h$ ). For a given  $h$  we observe that the stack at this height may contain up to  $h+1$  values. This means that after every round of the algorithm a maximum of  $1+2+\dots+H$  values are saved into the stacks. Thus we estimate the space complexity of the algorithm to be  $O(H^2/2) = O(\log^2(N)/2)$ . Imagine that we are given a big enough Merkle tree and we apply the classic traversal to it.

The bigger the height of the *TREEHASH* instance, the greater the number of saved node values in its stack, which leads us to huge storage requirements and makes the classic traversal highly inefficient for larger trees.

4.2 **The improvement:** Years after the original Merkle algorithm was created, an improved version appears. Michael Szydlo realizes that the scheduling strategy (step four in algorithm 3.2) of the classic traversal is what makes the storage requirements so big. The improved algorithm that he presents is called logarithmic traversal (description may be found in part 3.5 of the thesis) and the way the stacks are modified is changed. The new scheduling strategy makes up to  $2H - 1$  updates of a particularly chosen stack and every update is now done by spending just one unit of *TREEHASH* computation (instead of two by the classic traversal). Thus, the number of operations per round remains the same - proof of this claim can be found in [SZY04]. Now if we analyze the storage requirements of the new traversal technique, we see that the new stack update method requires less space, which is its main idea. The number of  $Auth_i$  nodes in the Merkle tree is always  $H$ . If we add the number of already completed nodes and the number of intermediate node values in the stacks we get the storage requirements of the algorithm – up to  $3H - 2$  node values are saved every round (a detailed proof is in [SZY04]). This results a  $O(3H) = O(3\log(N))$  space complexity, which compared to the  $O(\log^2(N)/2)$  space complexity of the classic traversal is much better. Thus, we prefer the usage of the logarithmic traversal for bigger trees. Actually, the new scheduling strategy of the presented logarithmic algorithm is not the best one. In an earlier version of his paper [SZY03], Michael Szydlo presented an algorithm which is even more effective than the described one. In it the left nodes of the tree are calculated almost for free and this halves the required hash computations each round. Unfortunately, the algorithm was more complicated and was not included in the original version of the paper.

4.3 **The fractal traversal:** We already presented the fractal tree representation and traversal technique in part 3.4 of the thesis (Algorithm 4.2). It is also an improvement of the classic traversal, but it has a different strategy. The fractal scheduling pattern comes from methods for fractal traversal of hash chains [CJ02], [J02], [S03]. The algorithm itself is simple and is the only one which allows tradeoffs between storage and computation, depending on the choice of the parameter  $h$  (which, at this point, makes it unique). The time requirements are easy to estimate – we see that the algorithm in its fourth step applies two units of *TREEHASH* computation to every desired subtree. As the maximal number of desired subtrees is  $L - 1$  (remember that  $L = H/h$ ), the maximal number of operations each round is  $2(L - 1) < 2H/h$ . This number of calculations is considerably smaller than the number of calculations of any previously described algorithm.

Now we want to estimate the storage requirements of the fractal algorithm. We need to keep the values of all nodes of both all existing and desired subtrees.

We have  $L$  existing subtrees in our Merkle tree and up to  $L-1$  desired subtrees. For them we need a maximum space of  $(2L-1)(2^{h+1}-2)$ , because every subtree has  $2^{h+1}-2$  nodes (the root is not included). We also need space for the intermediate nodes of the *TREEHASH* stacks, but we have to exclude those nodes that are part of the desired subtrees themselves. For a desired subtree with root of height  $ih$  (subtree at level  $i$ ,  $i > 1$ ) there are no more than  $h(i-1)+1$  such nodes in the stack tail. After making the calculations (that may be found in [JLMS03]) we estimate the maximal space required by the algorithm and bound it from above by  $2L2^{h+1} + HL/2 = 2H2^{h+1}/h + H^2/2h$ .

Considering the fractal algorithm, the only problem is that the bigger the subtree height  $h$  is, the faster the algorithm will run, but the more space it will require. This is exactly the reason why we should find an optimal  $h$  to minimize the storage requirements. We see that for very small  $h$  and big tree height  $H$  our space requirements are still huge, because  $H^2/2h$  remains large. Thus, by differentiation, an optimal value  $h$  is found. A better looking approximation of this value is  $h = \log(H) = \log \log(N)$ . For this  $h$  the storage requirements become  $5/2 \log^2(N)/\log \log(N)$  and the number of operations is bounded by  $2 \log(N)/\log \log(N)$ . The space requirements may be reduced even a little more by a slight modification, explained in [JLMS03]. The idea is that actually in almost all cases we may discard intermediate values of existing subtrees as soon as they have entered the corresponding desired subtree. Thus the space requirements are reduced to  $1,5 \log^2(N)/2 \log \log(N)$ . These results improve both time and space requirements of all previously published traversal techniques, but for bigger Merkle trees we still have great space consumption, which makes the fractal technique more inefficient than the logarithmic traversals presented by M. Szydło in 2003 and 2004.

To generalize the explanations made earlier we present a table with the requirements of all described algorithms from the previous chapter plus the version of Szydło's algorithm from 2003:

**The traversals with their space and time requirements:**

Traversal Technique:	Time Requirements:	Storage Requirements:
Classic Merkle Traversal 1979	$2 \log(N)$	$\log^2(N)/2$
Logarithmic Traversal 2003	$\log(N)$	$3 \log(N)$
Logarithmic Traversal 2004	$2 \log(N)$	$3 \log(N)$
Fractal Traversal 2003	$2 \log(N)/\log \log(N)$	$1,5 \log^2(N)/2 \log \log(N)$

To be a little more specific and to investigate the behavior of the four techniques we give some examples on how they would work in real life, using three exemplary trees. These Merkle trees have different heights and give an idea in which particular situation our algorithms may be used. Let  $T_i$  ( $i=1,2,3$ ) be Merkle trees of height  $H_1 = 5$ ,  $H_2 = 10$  and  $H_3 = 20$ . The time requirements for these trees (in number of elementary operations) are given in the following table:

**Time requirements in number elementary operations:**

Traversal Technique:	Time Requirements for $T_1$	Time Requirements for $T_2$	Time Requirements for $T_3$
Classic Merkle Traversal 1979	10	20	40
Logarithmic Traversal 2003	5	10	20
Logarithmic Traversal 2004	10	20	40
Fractal Traversal 2003	4,3068	6,0207	9,2552

Now we show the space requirements (in number of saved hash values) for the same Merkle trees. More drastic differences may be seen in the following table:

**Space requirements in number of saved hash values:**

Traversal Technique:	Space Requirements for $T_1$	Space Requirements for $T_2$	Space Requirements for $T_3$
Classic Merkle Traversal 1979	12,5	50	200
Logarithmic Traversal 2003	15	30	60
Logarithmic Traversal 2004	15	30	60
Fractal Traversal 2003	8,0753	22,5774	69,4139

Talking about space requirements we have to notice the fact that the real space requirements of all algorithms depend on the output length of our hash function. The next and last two tables show us the space requirements in bits for hash functions with 160-bit output and with 256-bit output:

**Space requirements for a 160-bit hash function:**

Traversal Technique:	Space Requirements for $T_1$	Space Requirements for $T_2$	Space Requirements for $T_3$
Classic Merkle Traversal 1979	2000 bits (0,25kB)	8000 bits (1kB)	32000 bits (4kB)
Logarithmic Traversal 2003	2400 bits (0,3kB)	4800 bits (0,6kB)	9600 bits (1,2kB)
Logarithmic Traversal 2004	2400 bits (0,3kB)	4800 bits (0,6kB)	9600 bits (1,2kB)
Fractal Traversal 2003	1292,048 bits (0,1615kB)	3612,384 bits (0,4515kB)	11106,224 bits (1,3883kB)

**Space requirements for a 256-bit hash function:**

Traversal Technique:	Space Requirements for $T_1$	Space Requirements for $T_2$	Space Requirements for $T_3$
Classic Merkle Traversal 1979	3200 bits (0,25kB)	12800 bits (1,6kB)	51200 bits (6,4kB)
Logarithmic Traversal 2003	3840 bits (0,48kB)	7680 bits (0,96kB)	15360 bits (1,92kB)
Logarithmic Traversal 2004	3840 bits (0,48kB)	7680 bits (0,96kB)	15360 bits (1,92kB)
Fractal Traversal 2003	2067,2768 bits (0,2584kB)	5779,8144 bits (0,7225kB)	17769,9584 bits (2,2212kB)

We can easily see the results of our comparison in the given tables and notice that the fractal algorithm is the fastest, but not always the most economy one. Smaller space requirements are possible for bigger trees using the logarithmic traversal from 2003. Such conclusions may be done for every particular situation, depending on what our needs are and how much time and space we may spend. Thus, general conclusions are not a good idea, conclusions for a particular cases are preferable.

## Chapter 5

# Conclusions and Future Work

We are now in 21<sup>st</sup> century – an era of Internet and IT infrastructures. Since 1994 it is well known that if anyone can build a large enough quantum computer, he can break all most popular public key cryptosystems like RSA, DSA and so on. According to the physicists this may happen the next 20 years. And since security is the heart of cryptography, now we are standing in front of one of the most serious problems nowadays – the improvement of post-quantum cryptography and digital signatures – what kind of signatures should we use to avoid the advantages of quantum computers. As Merkle scheme does not rely on any number theoretic assumptions (like for example the RSA is based on the well known integer factorization problem), but requires only the existence of collision resistant hash functions, it is a perfect candidate for a post-quantum signature system.

However a number of additional problems also appear like key pair generation and larger secret keys. Solutions may be found (for example in [CMSS06]), but the most important question still remains - how can we apply the Merkle scheme with its limited number of possible signatures. As we have already seen in the thesis, nowadays there are a number of improvements of the classic Merkle signature, but this question still remains unsolved. So that is why we still ask ourselves: Can we use the Merkle signature scheme for the modern applications at all?

**Acknowledgements:** I wish to thank to my supervisor Erik Dahmen for the helpful discussions and useful comments and corrections.

# References

- [MER79] Ralph Merkle, “Secrecy, Authentication and Public Key Systems/ A certified digital signature”, Ph.D. dissertation, Dept. of Electrical Engineering, Stanford University, 1979.
- [SZY03] Michael Szydlo, “Merkle Tree Traversal in Log Space and Time” (preprint version, 2003).
- [SZY04] Michael Szydlo, “Merkle Tree Traversal in Log Space and Time”, Eurocrypt 2004.
- [JLMS03] Markus Jakobsson, Tom Leighton, Silvio Micali and Michael Szydlo, “Fractal Merkle Tree Representation and Traversal”, RSA-CT '03.
- [DH76] Whitfield Diffie and Martin Hellman, “New Directions in Cryptography”, November 1976.
- [MOV96] A. Menezes, P. Van Oorschot and S. Vanstone, “Handbook of Applied Cryptography”, CRC Press, 1996.
- [WK01] [www.wikipedia.com](http://www.wikipedia.com): “The free online encyclopedia”.
- [EB05] Eli Biham, “Hashing. One-Time Signatures and MACs”, May 2007.
- [ECS] Stefaan Seys, Bart Preneel, “Efficient Cooperative Signatures: A Novel Authentication Scheme for Sensor Networks”.
- [CMSS06] J. Buchmann, L. C. Coronado Garcia, Erik Dahmen, M. Döring and E. Klintsevich “CMSS - An Improved Merkle Signature Scheme”, 2006.
- [CJ02] D. Coppersmith and M. Jakobsson, “Almost Optimal Hash Sequence Traversal”, Financial Crypto 2002.
- [J02] M. Jakobsson, “Fractal Hash Sequence Representation and Traversal”, ISIT 2002, page 437.
- [S03] Y. Sella, “On the Computation-Storage Tradeoffs of Hash Chain Traversal”, Financial Crypto 2003.
- [L02] H. Lipmaa, “On Optimal Hash Tree Traversal for Interval Time-Stamping”, In Proceedings of Information Security Conference 2002.