Hash-based Digital Signature Schemes

Johannes Buchmann Erik Dahmen Michael Szydlo

October 29, 2008

Contents

1 Introduction

Digital signatures have become a key technology for making the Internet and other IT-infrastructures secure. Digital signatures provide authenticity, integrity, and non-repudiation of data. Digital signatures are widely used in identification and authentication protocols. Therefore, the existence of secure digital signature algorithms is crucial for maintaining IT-security.

The digital signature algorithms that are used in practice today are RSA [31], DSA [11], and ECDSA [15]. They are not quantum immune since their security relies on the difficulty of factoring large composite integers and computing discrete logarithms.

Hash-based digital signature schemes which are presented in this chapter offer a very interesting alternative. Like any other digital signature scheme, hash-based digital signature schemes use a cryptographic hash function. Their security relies on the collision resistance of that hash function. In fact, we will present hash-based digital signature schemes that are secure if and only if the underlying hash function is collision resistant. The existence of collision resistant hash functions can be viewed as a minimum requirement for the existence of a digital signature scheme that can sign many documents with one private key. That signature scheme maps documents (arbitrarily long bit strings) to digital signatures (bit strings of fixed length). This shows that digital signature algorithms are in fact hash functions. Those hash functions must be collision resistant: if it were possible to construct two documents with the same digital signature, the signature scheme could no longer be considered secure. This argument shows that there exist hash-based digital signature schemes as long as there exists any digital signature scheme that can sign multiple documents using one private key. As a consequence, hash-based signature schemes are the most important post-quantum signature candidates. Although there is no proof of their quantum computer resistance, their security requirements are minimal. Also, each new cryptographic hash function yields a new hash-based signature scheme. So the construction of secure signature schemes is independent of hard algorithmic problems in number theory or algebra. Constructions from symmetric cryptography suffice. This leads to another big advantage of hash-based signature schemes. The underlying hash function can by chosen in view of the hardware and software resources available. For example, if the signature scheme is to be implemented on a chip that already implements AES, an AES based hash function can be used, thereby reducing the code size of the signature scheme and optimizing its running time.

Hash-based signature schemes were invented by Ralph Merkle [23]. Merkle started from one-time signature schemes, in particular that of Lamport and Diffie [18]. One-time signatures are even more fundamental. The construction of a secure one-time signature scheme only requires a one-way function. As shown by Rompel [28], one-way functions are necessary and sufficient for secure digital signatures. So one-time signature schemes are really the most fundamental type of digital signature schemes. However, they have a severe disadvantage. One key-pair consisting of a secret signature key and a public verification key can only be used to sign and verify a single document. This is inadequate for most applications. It was the idea of Merkle to use a hash tree that reduces the validity of many one-time verification keys (the leaves of the hash tree) to the validity of one public key (the root of the hash tree). The initial construction of Merkle was not sufficiently efficient, in particular in comparison to the RSA signature scheme. However in the meantime, many improvements have been found. Now hash-based signatures are the most promising alternative to RSA and elliptic curve signature schemes.

2 Hash based one-time signature schemes

This chapter explains signature schemes whose security is only based on the collision resistance of a cryptographic hash function. Those schemes are particularly good candidates for the post quantum era.

2.1 Lamport–Diffie one-time signature scheme

The Lamport–Diffie one-time signature scheme (LD-OTS) was proposed in [18]. Let n be a positive integer, the security parameter of LD-OTS. LD-OTS uses a one-way function

$$f : \{0,1\}^n \to \{0,1\}^n,$$

and a cryptographic hash function

$$g : \{0,1\}^* \to \{0,1\}^n.$$

LD-OTS key pair generation. The signature key X of LD-OTS consists of $2n$ bit strings of length n chosen uniformly at random,

$$X = \big(x_{n-1}[0], x_{n-1}[1], \ldots, x_1[0], x_1[1], x_0[0], x_0[1]\big) \in_R \{0,1\}^{(n,2n)}. \tag{1}$$

The LD-OTS verification key Y is

$$Y = \big(y_{n-1}[0], y_{n-1}[1], \ldots, y_1[0], y_1[1], y_0[0], y_0[1]\big) \in \{0,1\}^{(n,2n)}, \tag{2}$$

where

$$y_i[j] = f\big(x_i[j]\big), \quad 0 \le i \le n-1, j = 0, 1. \tag{3}$$

So LD-OTS key generation requires $2n$ evaluations of f. The signature and verification keys are $2n$ bit strings of length n.

LD-OTS signature generation. A document $M \in \{0,1\}^*$ is signed using LD-OTS with a signature key X as in Equation (1). Let $g(M) = d = (d_{n-1}, \ldots, d_0)$ be the message digest of M. Then the LD-OTS signature is

$$\sigma = \big(x_{n-1}[d_{n-1}], \ldots, x_1[d_1], x_0[d_0]\big) \in \{0,1\}^{(n,n)}. \tag{4}$$

This signature is a sequence of n bit strings, each of length n. They are chosen as a function of the message digest d. The ith bit string in this signature is $x_i[0]$ if the ith bit in d is 0 and $x_i[1]$, otherwise. Signing requires no evaluations of f. The length of the signature is n^2.

LD-OTS Verification. To verify a signature $\sigma = (\sigma_{n-1}, \ldots, \sigma_0)$ of M as in (4), the verifier calculates the message digest $d = (d_{n-1}, \ldots, d_0)$. Then she checks whether

$$\big(f(\sigma_{n-1}), \ldots, f(\sigma_0)\big) = \big(y_{n-1}[d_{n-1}], \ldots, y_0[d_0]\big). \tag{5}$$

Signature verification requires n evaluations of f.

Example 2.1 *Let $n = 3$, $f : \{0,1\}^3 \to \{0,1\}^3, x \mapsto x + 1 \bmod 8$, and let $d = (1,0,1)$ be the hash value of a message M. We choose the signature key*

$$X = \big(x_2[0], x_2[1], x_1[0], x_1[1], x_0[0], x_0[1]\big) = \begin{pmatrix} 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 \end{pmatrix} \in \{0,1\}^{(3,6)}$$

and compute the corresponding verification key

$$Y = \big(y_2[0], y_2[1], y_1[0], y_1[1], y_0[0], y_0[1]\big) = \begin{pmatrix} 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix} \in \{0,1\}^{(3,6)}.$$

The signature of $d = (1,0,1)$ is

$$\sigma = (\sigma_2, \sigma_1, \sigma_0) = (x_2[1], x_1[0], x_0[1]) = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 0 \end{pmatrix} \in \{0,1\}^{(3,3)}$$

Example 2.2 *We give an example to illustrate why the signature keys of LD-OTS must be used only once. Let $n = 4$. Suppose the signer signs two messages with digests $d_1 = (1,0,1,1)$ and $d_2 = (1,1,1,0)$ using the same signature key. The signatures of these digests are $\sigma_1 = (x_3[1], x_2[0], x_1[1], x_0[1])$ and $\sigma_2 = (x_3[1], x_2[1], x_1[1], x_0[0])$, respectively. Then an attacker knows $x_3[1], x_2[0], x_2[1], x_1[1], x_0[0], x_0[1]$ from the signature key. She can use this information to generate valid signatures for messages with digests $d_3 = (1,0,1,0)$ and $d_4 = (1,1,1,1)$. This example can be generalized to arbitrary security parameters n. Also, the attacker is only able to generate valid signatures for certain digests. As long as the hash function used to compute the message digest is cryptographically secure, she cannot find appropriate messages.*

2.2 Winternitz one-time signature scheme

While the key and signature generation of LD-OTS is very efficient, the size of the signature is quite large. The Winternitz OTS (W-OTS), which is explained in this section, produces significantly shorter signatures. The idea is to use one string in the one-time signature key to simultaneously sign several bits in the message digest. In literature this proposal appears first in Merkle's thesis [23]. Merkle writes that the method was suggested to him by Winternitz in 1979 as a generalization of the Merkle OTS also described in [23]. However, to the best of the authors knowledge, the Winternitz OTS was for the first time described in full detail in [10]. Like LD-OTS, W-OTS uses a one-way function

$$f : \{0,1\}^n \to \{0,1\}^n$$

and a cryptographic hash function

$$g : \{0,1\}^* \to \{0,1\}^n.$$

W-OTS key pair generation. A Winternitz parameter $w \geq 2$ is selected which is the number of bits to be signed simultaneously. Then

$$t_1 = \left\lceil \frac{n}{w} \right\rceil, \quad t_2 = \left\lfloor \frac{\lfloor \log_2 t_1 \rfloor + 1 + w}{w} \right\rfloor, \quad t = t_1 + t_2. \tag{6}$$

are determined. The signature key X is

$$X = \big(x_{t-1}, \ldots, x_1, x_0\big) \in_R \{0,1\}^{(n,t)}. \tag{7}$$

where the bit strings x_i are chosen uniformly at random.

The verification key Y is computed by applying f to each bit string in the signature key $2^w - 1$ times. So we have

$$Y = \big(y_{t-1}, \ldots, y_1, y_0\big) \in \{0,1\}^{(n,t)}, \tag{8}$$

where

$$y_i = f^{2^w-1}\big(x_i\big), 0 \leq i \leq t-1. \tag{9}$$

Key generation requires $t(2^w - 1)$ evaluations of f and the lengths of the signature and verification key are $t \cdot n$ bits, respectively.

W-OTS signature generation. A message M with message digest $g(M) = d = (d_{n-1}, \ldots, d_0)$ is signed. First, a minimum number of zeros is prepended to d such that the length of d is divisible by w. The extended string d is split into t_1 bit strings $b_{t-1}, \ldots, b_{t-t_1}$ of length w. Then

$$d = b_{t-1} \parallel \ldots \parallel b_{t-t_1}, \tag{10}$$

where $\parallel$ denotes concatenation. Next, the bit strings b_i are identified with integers in $\{0, 1, \ldots, 2^w - 1\}$ and the checksum

$$c = \sum_{i=t-t_1}^{t-1} (2^w - b_i) \tag{11}$$

is calculated. Since $c \leq t_1 2^w$, the length of the binary representation of c is less than

$$\lfloor \log_2 t_1 2^w \rfloor + 1 = \lfloor \log_2 t_1 \rfloor + w + 1. \tag{12}$$

A minimum number of zeros is prepended to this binary representation such that the length of the extended string is divisible by w. That extended string is split into t_2 blocks $b_{t_2-1}, \ldots, b_0$ of length w. Then

$$c = b_{t_2-1} || \ldots || b_0.$$

Finally the signature of M is computed as

$$\sigma = \big(f^{b_{t-1}}(x_{t-1}), \ldots, f^{b_1}(x_1), f^{b_0}(x_0)\big). \tag{13}$$

In the worst case, signature generation requires $t(2^w - 1)$ evaluations of f. The W-OTS signature size is $t \cdot n$.

W-OTS verification. For the verification of the signature $\sigma = (\sigma_{t-1}, \ldots, \sigma_0)$ the bit strings $b_{t-1}, \ldots, b_0$ are calculated as explained in the previous section. Then we check if

$$\big(f^{2^w-1-b_{t-1}}(\sigma_{n-1}), \ldots, f^{2^w-1-b_0}(\sigma_0)\big) = \big(y_{n-1}, \ldots, y_0\big). \tag{14}$$

If the signature is valid, then $\sigma_i = f^{b_i}(x_i)$ and therefore

$$f^{2^w-1-b_i}(\sigma_i) = f^{2^w-1}(x_i) = y_i \tag{15}$$

holds for $i = t-1, \ldots, 0$. In the worst case, signature verification requires $t(2^w - 1)$ evaluations of f.

Example 2.3 *Let $n = 3, w = 2$, $f : \{0,1\}^3 \to \{0,1\}^3, x \mapsto x + 1 \bmod 8$ and $d = (1,0,0)$. We get $t_1 = 2, t_2 = 2$, and $t = 4$. We choose the signature key as*

$$X = \big(x_3, x_2, x_1, x_0\big) = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \in \{0,1\}^{(3,4)}$$

and compute the verification key by applying f three times to the bit strings in X:

$$Y = \big(y_3, y_2, y_1, y_0\big) = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix} \in \{0,1\}^{(3,4)}.$$

Prepending one zero to d and splitting the extended string into blocks of length 2 yields $d = 01||00$. The checksum c is $c = (4-1) + (4-0) = 7$. Prepending one zero to the binary representation of c and splitting the extended string into blocks of length 2 yields $c = 01||11$. The signature is

$$\sigma = (\sigma_3, \sigma_2, \sigma_1, \sigma_0) = \big(f(x_3), x_2, f(x_1), f^3(x_0)\big) = \begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \in \{0,1\}^{(3,4)}.$$

The signature is verified by computing

$$\big(f^2(\sigma_3), f^3(\sigma_2), f^2(\sigma_1), \sigma_0\big) = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix} \in \{0,1\}^{(3,4)}$$

and comparing it with the verification key Y.

Example 2.4 *We give an example to illustrate why the signature keys of the W-OTS must be used only once. Let $w = 2$. Suppose the signer signs two messages with digests $d_1 = (1,0,0)$ and $d_2 = (1,1,1)$ using the same signature key. The signatures of these digests are $\sigma_1 = \big(f(x_3), x_2, f(x_1), f^3(x_0)\big)$ and $\sigma_2 = \big(f(x_3), f^3(x_2), f(x_1), x_0\big)$, respectively. The attacker can use this information to compute the signatures for messages with digest $d_3 = (1,1,0)$ given as $\sigma_3 = \big(f(x_3), f^2(x_2), f(x_1), f(x_0)\big)$ Again this example can be generalized to arbitrary security parameters n. Also, the attacker can only produce valid signatures for certain digests. As long as the hash function used to compute the message digest is cryptographically secure, he cannot find appropriate messages.*

3 Merkle's tree authentication scheme

The one-time signature schemes introduced in the last section are inadequate for most practical situations since each key pair can only be used for one signature. In 1979 Ralph Merkle proposed a solution to this problem [23]. His idea is to use a complete binary hash tree to reduce the validity of an arbitrary but fixed number of one time verification keys to the validity of one single public key, the root of the hash tree.

The Merkle signature scheme (MSS) works with any cryptographic hash function and any one-time signature scheme. For the explanation we let $g : \{0,1\}^* \to \{0,1\}^n$ be a cryptographic hash function. We also assume that a one-time signature scheme has been selected.

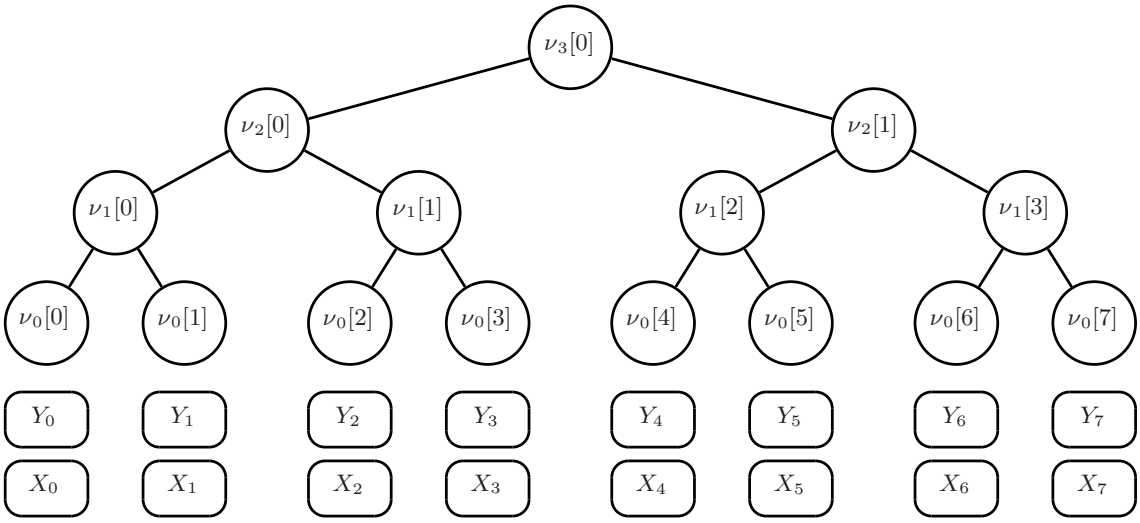

Figure 1: A Merkle tree of height $H = 3$

MSS key pair generation

The signer selects $H \in \mathbb{N}$, $H \geq 2$. Then the key pair to be generated will be able to sign/verify 2^H documents. Note that this is an important difference to signature schemes such as RSA and ECDSA, where potentially arbitrarily many documents can be signed/verified with one key pair. However, in practice this number is also limited by the devices on which the signature is generated or by some policy. The signer generates 2^H one-time key pairs (X_j, Y_j), $0 \leq j < 2^H$. Here X_j is the signature key and Y_j is the verification key. They are both bit strings. The leaves of the Merkle tree are the digests $g(Y_j)$, $0 \leq j < 2^H$. The inner nodes of the Merkle tree are computed according to the following construction rule: a parent node is the hash value of the concatenation of its left and right children. The MSS public key is the root of the Merkle tree. The MSS private key is the sequence of the 2^H one-time signature keys. To be more precise, denote the nodes in the Merkle tree by $\nu_h[j]$, $0 \leq j < 2^{H-h}$, where $h \in \{0, \ldots, H\}$ is the height of the node. Then

$$\nu_h[j] = g(\nu_{h-1}[2j] \parallel \nu_{h-1}[2j+1]), \quad 1 \leq h \leq H, 0 \leq j < 2^{H-h}. \tag{16}$$

Figure 1 shows an example for $H = 3$.

MSS key pair generation requires the computation of 2^H one-time key pairs and $2^{H+1}-1$ evaluations of the hash function.

Efficient root computation

In order to compute the root of the Merkle tree it is not necessary to store the full hash tree. Instead, the treehash algorithm 3.1 is applied. The basic idea of this algorithm is to successively compute leaves and, whenever possible, compute their parents. To store nodes, the treehash algorithm uses a stack STACK equipped with the usual push and pop operations. Input of the tree hash algorithm is the height H of the Merkle tree. Output is the root of the Merkle tree, i.e. the MSS public key. Algorithm 3.1 uses the subroutine LEAFCALC(j) to compute the jth leaf. The LEAFCALC(j) routine computes the jth one-time key pair and computes the jth leaf from the jth one-time verification key as described above.

Algorithm 3.1 Treehash

Input: Height $H \geq 2$

Output: Root of the Merkle tree

1. **for** $j = 0, \ldots, 2^H - 1$ **do**
   (a) Compute the jth leaf: $\text{NODE}_1 \leftarrow \text{LEAFCALC}(j)$
   (b) **While** NODE_1 has the same height as the top node on STACK **do**
      i. Pop the top node from the stack: $\text{NODE}_2 \leftarrow \text{STACK.pop}()$
      ii. Compute their parent node: $\text{NODE}_1 \leftarrow g(\text{NODE}_2 \parallel \text{NODE}_1)$
   (c) Push the parent node on the stack: $\text{STACK.push}(\text{NODE}_1)$
2. Let R be the single node stored on the stack: $R \leftarrow \text{STACK.pop}()$
3. **Return** R

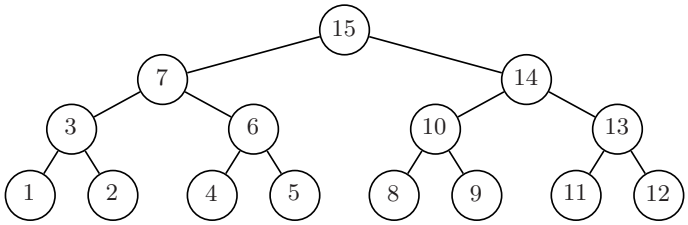

Figure 2: The treehash algorithm

Figure 2 shows the order in which the nodes of a Merkle tree are computed by the treehash algorithm. In this example, the maximum number of nodes that are stored on the stack is 3. This happens after node 11 is generated and pushed on the stack. In general, the treehash algorithm needs to store at most H so-called *tail nodes* on the stack. To compute the root of a Merkle tree of height H, the treehash algorithm requires 2^H calls of the LEAFCALC subroutine, and $2^H - 1$ evaluations of the hash function.

MSS signature generation

MSS uses the one-time signature keys successively for the signature generation. To sign a message M, the signer first computes the n-bit digest $d = g(M)$. Then he generates the one-time signature σ_{OTS} of the digest using the sth one-time signature key X_s, $s \in \{0, \ldots, 2^H - 1\}$. The Merkle signature will contain this one-time signature and the corresponding one-time verification key Y_s. To prove the authenticity of Y_s to the verifier, the signer also includes the index s as well as an authentication path for the verification key Y_s which is a sequence $A_s = (a_0, \ldots, a_{H-1})$ of nodes in the Merkle tree. This index and the authentication path allow the verifier to construct a path from the leaf $g(Y_s)$ to the root of the Merkle tree. Node h in the authentication path is the sibling of the height h node on the path from leaf $g(Y_s)$ to the Merkle tree root:

$$a_h = \begin{cases} \nu_h[s/2^h - 1] & \text{, if } \lfloor s/2^h \rfloor \equiv 1 \bmod 2 \\ \nu_h[s/2^h + 1] & \text{, if } \lfloor s/2^h \rfloor \equiv 0 \bmod 2 \end{cases} \tag{17}$$

for $h = 0, \ldots H - 1$. Figure 3 shows an example for $s = 3$. So the sth Merkle signature is

$$\sigma_s = \big(s, \sigma_{\text{OTS}}, Y_s, (a_0, \ldots, a_{H-1})\big) \tag{18}$$

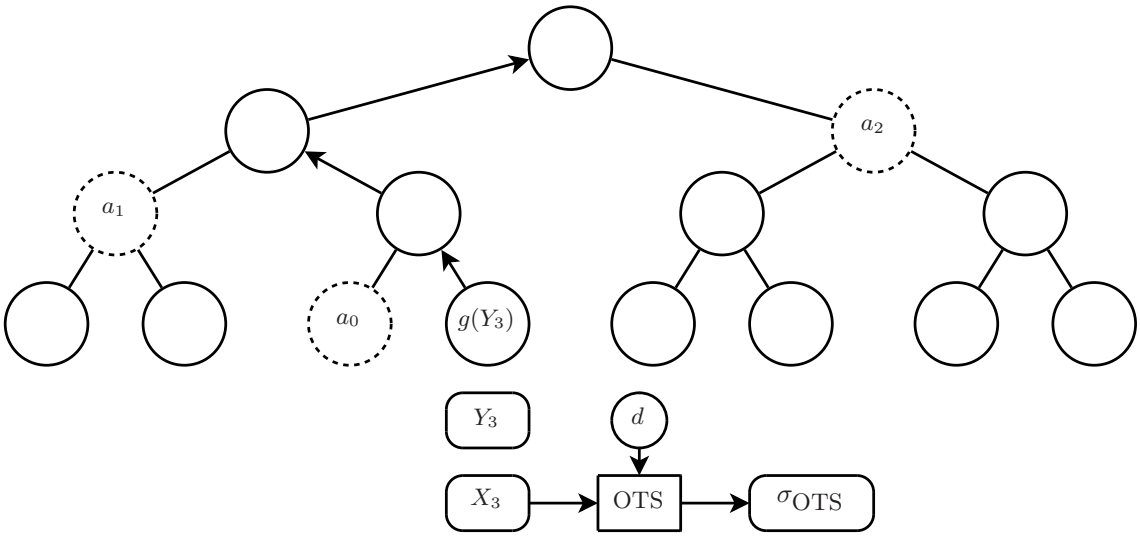

Figure 3: Merkle signature generation for $s = 3$. Dashed nodes denote the authentication path for leaf $g(Y_3)$. Arrows indicate the path from leaf $g(Y_3)$ to the root.

MSS signature verification

Verification of the Merkle signature from the previous section consists of two steps. In the first step, the verifier uses the one-time verification key Y_s to verify the one-time signature σ_{OTS} of the digest d by means of the verification algorithm of the respective one-time signature scheme. In the second step the verifier validates the authenticity of the one-time verification key Y_s by constructing the path $(p_0, \ldots, p_H)$ from the sth leaf $g(Y_s)$ to the root of the Merkle tree. He uses the index s and the authentication path $(a_0, \ldots, a_{H-1})$ and applies the following construction.

$$p_h = \begin{cases} g(a_{h-1}||p_{h-1}) & \text{, if } \lfloor s/2^{h-1} \rfloor \equiv 1 \bmod 2 \\ g(p_{h-1}||a_{h-1}) & \text{, if } \lfloor s/2^{h-1} \rfloor \equiv 0 \bmod 2 \end{cases} \tag{19}$$

for $h = 1, \ldots H$ and $p_0 = g(Y_s)$. The index s is used for deciding in which order the authentication path nodes and the nodes on the path from leaf $g(Y_s)$ to the Merkle tree root are to be concatenated. The authentication of the one-time verification key Y_s is successful if and only if p_H equals the public key.

4 One-time key-pair generation using a PRNG

According to the description of MSS from Section 3, the MSS private key consists of 2^H one-time signature keys. Storing such a huge amount of data is not feasible for most practical applications. As suggested in [3], space can be saved by using a deterministic pseudo random number generator (PRNG) and storing only the seed of that PRNG. Then each one-time signature key must be generated twice, once for the MSS public key generation and once during the signing phase.

In the following, let PRNG be a cryptographically secure pseudo random number generator that on input an n-bit seed SEED_{in} outputs a random number RAND and an updated seed SEED_{out}, both of bit length n.

$$\begin{aligned} \text{PRNG} : \{0,1\}^n &\rightarrow \{0,1\}^n \times \{0,1\}^n \\ \text{SEED}_{\text{in}} &\mapsto (\text{RAND}, \text{SEED}_{\text{out}}) \end{aligned} \tag{20}$$

MSS key pair generation using an PRNG

We explain how MSS key-pair generation using a PRNG works. The first step is to choose an n-bit seed SEED_0 uniformly at random. For the generation of the one-time signature keys we use a sequence of seeds SEEDOTS_j, $0 \leq j < 2^H$. They are computed iteratively using

$$(\text{SEEDOTS}_j, \text{SEED}_{j+1}) = \text{PRNG}(\text{SEED}_j), 0 \leq j < 2^H. \tag{21}$$

Here SEEDOTS_j is used to calculate the jth one-time signature key.

For example, in the case of W-OTS (see Section 2.2) the jth signature key is $X_j = (x_{t-1}, \ldots, x_0)$. The t bit strings of length n in this signature key are generated using SEEDOTS_j.

$$(x_i, \text{SEEDOTS}_j) = \text{PRNG}(\text{SEEDOTS}_j), i = t-1, \ldots, 0 \tag{22}$$

The seed SEEDOTS_j is updated during each call to the PRNG. This shows that in order to calculate the signature key X_j only knowledge of SEED_j is necessary. When SEEDOTS_j is computed, the new seed SEED_{j+1} for the generation of the signature key X_{j+1} is also determined. Figure 4 visualizes the one-time signature key generation using an PRNG.

If this method is used, the MSS private key is initially SEED_0. Its length is n. It is replaced by the seeds SEED_{j+1} determined during the generation of signature key X_j.

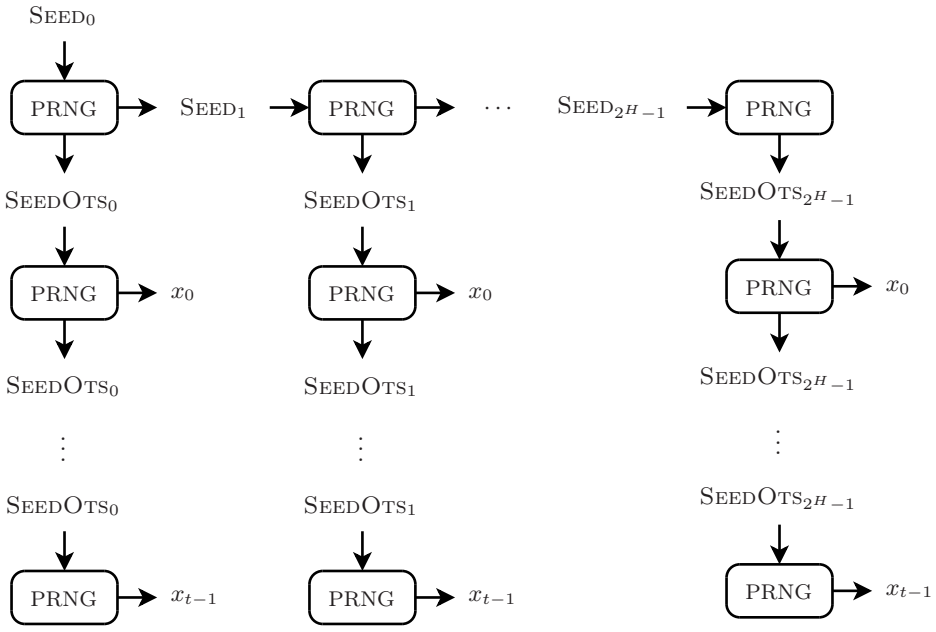

Figure 4: One-time signature key generation using an PRNG

MSS signature generation using an PRNG

In contrast to the original MSS signature generation, the one-time signature key must be computed before the signature is generated. When the signature key is computed the seed is updated for the next signature.

Forward security

In addition to reducing the private key size, using a PRNG for the one-time signature key generation has another benefit. It makes MSS forward secure as long as PRNG is forward

secure which means that calculating previous seeds from the actual seed is infeasible. Forward security of the signature scheme means that all signatures issued before a revocation remain valid. MSS is forward secure, since the actual MSS private key can only be used to generate one-time signature keys for upcoming signatures but not to forge previous.

5 Authentication path computation

In this chapter we will present a variety of techniques for traversal of Merkle trees of height H. The use of the techniques is *transparent* to a verifier, who will not need to know how a set of outputs were generated, but only that they are correct. Therefore, the technique can be employed in any construction for which the generation and output of authentication paths for consecutive leaves is required.

The first traversal algorithm is structurally very simple and allows for various tradeoffs between storage and computation. For one choice of parameters, the total space required is bounded by $1.5H^2/\log H$ hash values, and the worst-case computational effort is $2H/\log H$ tree node computations per output.

The next Merkle tree-traversal algorithm has a better space and time complexity than the previously known algorithms. Specifically, the algorithm requires computation of at most $2H$ tree nodes per round and requires storage of less than $3H$ node values. We also prove that this complexity is optimal in the sense that there can be no Merkle Tree traversal algorithm which requires both less than $O(H)$ time and less than $O(H)$ space.

In the analysis of the first two algorithms, the computation of a leaf and an inner node are each counted as a single elementary operation[1].

The third Merkle tree-traversal algorithm has the same space and time complexity as the second. However it has a significant constant factor improvement and was designed for practical implementation. It distinguishes between leaf computations and the computation of inner nodes. To traverse a tree of height H it roughly requires the computation of $H/2$ leaves and $3H/2$ inner nodes.

5.1 The Classic Traversal

The challenge of Merkle tree traversal is to ensure that all node values are ready when needed, but are computed in a manner which conserves space and time. To motivate the new algorithms, we first discuss what the average per-round computation is expected to be, and review the classic Merkle tree traversal.

Average Costs. Each node in the tree is eventually part of an authentication path, so one useful measure is the total cost of computing each node value exactly once. There are 2^{H-h} right (respectively, left) nodes at height h, and if computed independently, each costs $2^{h+1}-1$ operations. Rounding up, this is $2^{H+1} = 2N$ operations, or two per round. Adding together the costs for each height h ($0 \leq h < H$), we expect, on average, $2H = 2\log(N)$ operations per round to be required.

Three Components. As with a digital signature scheme, the tree-traversal algorithms consists of three components: *key generation*, *output*, and *verification*. During *key generation*, the first authentication path and some upcoming authentication node values are computed.

[1]This differs from the measurement of *total* computational cost, which includes, e.g., the scheduling algorithm itself.

The *output* phase consists of N rounds, one for each leaf $s \in \{0, \ldots, N-1\}$. During round s, the authentication path for the sth leaf, AUTH_i, $i = 0, \ldots, H-1$ is output. Additionally, the algorithm's state is modified in order to prepare for future outputs.

The *verification* phase is identical to the traditional verification phase for Merkle trees described in Section 3.

Notation. In addition to denoting the current authentication nodes AUTH_h, we need some notation to describe the stacks used to compute upcoming needed nodes. Define STACK_h to be an object which contains a stack of node values as in the description of the treehash algorithm in Section 3, Algorithm 3.1. STACK_h.*initialize* and STACK_h.*update* will be methods to setup and incrementally execute treehash.

Algorithm presentation

Key Generation and Setup. The main task of key generation is to compute and publish the root value. This is a direct application of the treehash algorithm described in Section 3. In the process of this computation, every node value is computed, and, it is important to record the initial values AUTH_i, as well as the upcoming values for each of the AUTH_i.

If we denote the jth node at height h by $\nu_h[j]$, we have $\text{AUTH}_h = \nu_h[1]$ (these are right nodes). The "upcoming" authentication node at height h is $\nu_h[0]$ (these are left nodes). These node values are used to initialize STACK_h to be in the state of the treehash algorithm having completed.

Algorithm 5.1 Key-Gen and Setup

1. **Initial Authentication Nodes** For each $h \in \{0, 1, \ldots H-1\}$:
   Calculate $\text{AUTH}_h = \nu_h[1]$.
2. **Initial Next Nodes** For each $h \in \{0, 1, \ldots H-1\}$:
   Setup STACK_h with the single node value $\text{AUTH}_h = \nu_h[0]$.
3. **Public Key** Calculate and publish tree root, $\nu_H[0]$.

Output and Update. Merkle's tree traversal algorithm runs one instance of the treehash algorithm for each height h to compute the next authentication node value for that level. Every 2^h rounds, the authentication path will shift to the right at level h, thus requiring a new node (its sibling) as the height h authentication node.

At each round the state of the treehash algorithm is updated with two units of computation. After 2^h rounds this node value computation will be completed, and a new instance of treehash begins for the next authentication node at that level.

To specify how to refresh the AUTH nodes, we observe how to easily determine which heights need updating: height h needs updating if and only if 2^h divides $s+1$ evenly, where $s \in \{0, \ldots, N-1\}$ denotes the current round. Furthermore, we note that at round $s + 1 + 2^h$, the authentication *path* will pass though the $(s + 1 + 2^h)/2^h$th node at height h. Thus, its *sibling's* value, (the new required upcoming AUTH_h) is determined from the 2^h leaf values starting from leaf number $(s + 1 + 2^h) \oplus 2^h$, where $\oplus$ denotes bitwise XOR.

In this language, we summarize Merkle's classic traversal algorithm in Algorithm 5.2.

Algorithm 5.2 Classic Merkle Tree Traversal

1. Set $s = 0$.
2. **Output:**

- For each $h \in [0, H-1]$ output $\textsc{Auth}_h$.

3. **Refresh Auth Nodes:**
   For all h such that 2^h divides $s+1$:

- Set $\textsc{Auth}_h$ be the sole node value in $\textsc{Stack}_h$.
- Set startnode $= (s + 1 + 2^h) \oplus 2^h$.
- $\textsc{Stack}_h.\mathit{initialize}(\text{startnode}, h)$.

4. **Build Stacks:**
   For all $h \in [0, H-1]$:

- $\textsc{Stack}_h.\mathit{update}(2)$. (Each stack receives two updates)

5. **Loop:**

- Set $s = s + 1$.
- If $s < 2^H$ go to Step 2.

5.2 Fractal Merkle Tree Traversal

The term "fractal" was chosen due to the focus on many smaller binary trees within the larger structure of the Merkle tree.

The crux of this algorithm is the selection of which node values to compute and retain at each step of the output algorithm. We describe this selection by using a collection of subtrees of fixed height h. We begin with some notation and then provide the intuition for the algorithm.

Notation. Starting with a Merkle tree $\textsc{Tree}$ of height H, we introduce further notation to deal with subtrees. First we choose a *subtree height* $h < H$. We let the altitude of a node ν in $\textsc{Tree}$ be the length of the path from ν to a leaf of $\textsc{Tree}$ (therefore, the altitude of a leaf of $\textsc{Tree}$ is zero). Consider a node ν with altitude at least h. We define the h-subtree at ν to be the unique subtree in $\textsc{Tree}$ which has ν as its root and which has height h. For simplicity in the suite, we assume h is a divisor of H, and let the ratio, $L = H/h$, be the number of *levels* of subtrees. We say that an *h-subtree at ν* is "at level i" when it has altitude ih for some $i \in \{1, 2, \ldots H\}$. For each i, there are 2^{H-ih} such h-subtrees at level i.

We say that a series of h-subtrees $\textsc{Tree}_i$ $(i = 1 \ldots L)$ is a *stacked series of h-subtrees*, if for all $i < L$ the root of $\textsc{Tree}_i$ is a leaf of $\textsc{Tree}_{i+1}$. We illustrate the subtree notation and provide a visualization of a *stacked series of h-subtrees* in Figure 5.

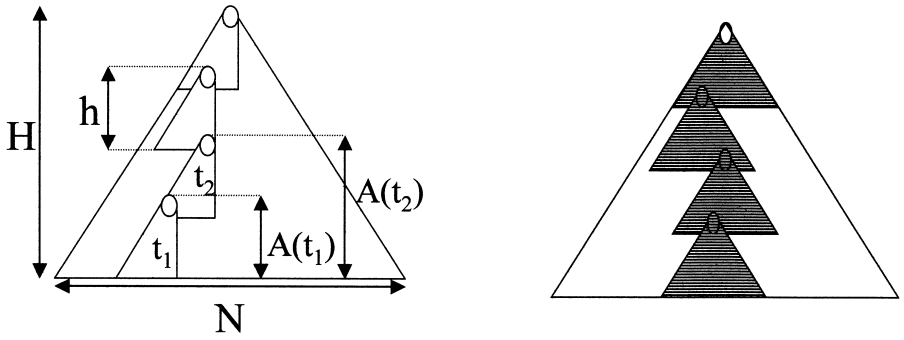

Figure 5: *(Left)* The height of the Merkle tree is H, and thus, the number of leaves is $N = 2^H$. The height of each subtree is h. The altitude $A(t_1)$ and $A(t_2)$ of the subtrees t_1 and t_2 is marked. *(Right)* Instead of storing all tree nodes, we store a smaller set - those within the stacked 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.

Existing and Desired Subtrees

Static view. As previously mentioned, we store some portion of the node values, and update what values are stored over time. Specifically, during any point of the output phase, there will exist a series of stacked *existing* subtrees, as in Figure 2. We say that we place a *pebble* on a node ν of the tree TREE when we store this node. There are always L such subtrees EXIST_i for each $i \in \{1, \dots L\}$, with pebbles on each of their nodes (except their roots). By design, for any leaf in EXIST_1, the corresponding authentication path is completely contained in the stacked set of existing subtrees.

Dynamic view. Apart from the above set of *existing* subtrees, which contain the next required authentication path, we will have a set of *desired* subtrees. If the root of the tree EXIST_i has index a, according to the ordering of the height-ih nodes, then DESIRE_i is defined to be the h-subtree with index $a + 1$ (provided that $a < 2^{H-i \cdot h} - 1$). In case $a = 2^{H-i \cdot h} - 1$, then EXIST_i is the last subtree at this level, and there is no corresponding desired subtree. In particular, there is never a desired subtree at level L. The left part of Figure 6 depicts the adjacent existing and desired subtrees.

As the name suggests, we need to compute the pebbles in the desired subtrees. This is accomplished by adapting an application of the treehash algorithm (Section 3, Algorithm 3.1) to the root of DESIRE_i. For these purposes, the treehash algorithm is altered to save the pebbles needed for DESIRE_i, rather than discarding them, and secondly to terminate one round early, never actually computing the root. Using this variant of treehash, we see that each desired subtree being computed has a *tail* of saved intermediate pebbles. We depict this dynamic computation in the right part of Figure 6, which shows partially completed subtrees and their associated tails.

Algorithm Intuition

We now can present intuition for the main algorithm, and explain why the existing subtrees EXIST_i will always be available.

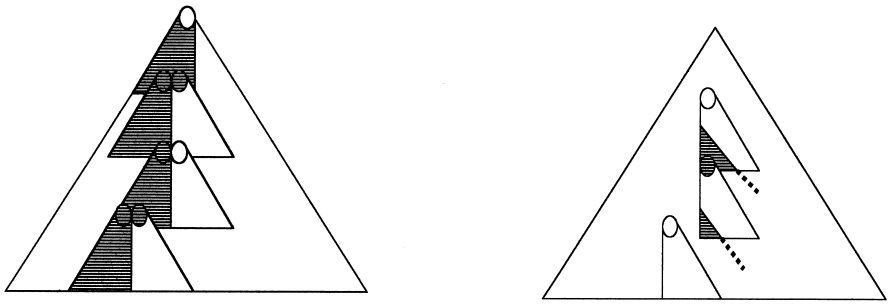

Figure 6: *(Left)* The grey subtrees correspond to the *existing* subtrees (as in figure 5) while the white subtrees correspond to the *desired* subtrees. As the existing subtrees are used up, the desired subtrees are gradually constructed. *(Right)* The figure shows the set of desired subtrees from the previous figure, but with grey portions corresponding to nodes that have been computed and dotted lines corresponding to pebbles in the tail.

Overview. The goal of the traversal is to sequentially output authentication paths. By design, the existing subtrees should always contain the next authentication path to be output, while the desired subtrees contain more and more completed pebbles with each round, until the existing subtree expires.

When $\textsc{Exist}_i$ is used in an output for the last time, we say that it *dies*. At that time, the adjacent subtree, $\textsc{Desire}_i$ will need to have been completed, i.e., have values assigned to all its nodes but its root (since the latter node is already part of the parent tree.) The tree $\textsc{Exist}_i$ is then *reincarnated* as $\textsc{Desire}_i$. First all the old pebbles of $\textsc{Exist}_i$ are discarded; then the pebbles of $\textsc{Desire}_i$ (and their associated values) taken by $\textsc{Exist}_i$. (Once this occurs, the computation of the new and adjacent subtree $\textsc{Desire}_i$ will be initiated.) This way, if one can ensure that the pebbles on trees $\textsc{Desire}_i$ are always computed on time, one can see that there will always be completed existing subtrees $\textsc{Exist}_i$.

Modifying the treehash algorithm. As mentioned above, our tool used to compute the desired tree is a modified version of the classic treehash algorithm applied to the root of $\textsc{Desire}_i$. This version differs in that (1) it stops the algorithm one round earlier (thereby skipping the root calculation), and (2) every pebble of height greater than ih is saved into the tree $\textsc{Desire}_i$. For purposes of counting, we won't consider such saved pebbles as part of the *proper tail*.

Amortizing the computations. For a particular level i, we recall that the computational cost for tree $\textsc{Desire}_i$ is $2 \cdot 2^{ih} - 2$, as we omit the calculation of the root. At the same time, we know that $\textsc{Exist}_i$ will serve for 2^{ih} output rounds. We amortize the computation of $\textsc{Desire}_i$ over this period, by simply computing two iterations of treehash each round. In fact, $\textsc{Desire}_i$ will be ready before it is needed, exactly 1 round in advance!

Thus, for each level, allocating 2 computational units ensures that the desired trees are completed on time. The total computation per round is thus $2(L-1)$.

Solution and Algorithm Presentation

Three phases. We now describe more precisely the main algorithm. There are three phases, the *key generation* phase; the *output* phase; and the *verification* phase. During the key generation phase (which may be performed offline by a relatively powerful computer), the root of the tree is computed and output, taking the role of a public key. Additionally, the iterative output phase needs some setup, namely the computation of pebbles on the initial *existing* subtrees. These are stored on the computer performing the output phase.

The *output* phase consists of a number of rounds. During round s, the authentication path of the sth leaf is output. In addition, some number of pebbles are discarded and some number of pebbles are computed, in order to prepare for future outputs.

The *verification* phase is identical to the traditional verification phase for Merkle trees and has been described above. We remark again that the outputs the algorithm generates will be indistinguishable from the outputs generated by a traditional algorithm. Therefore, we do not detail the verification phase, but merely the key generation phase and output phase.

Key Generation. First, the pebbles of the left-most set of stacked *existing* subtrees are computed and stored. Each associated pebble has a *value*, a *position*, and a *height*. In addition, a list of *desired* subtrees is created, one for each level $i < L$, each initialized with an empty stack for use in the modified treehash algorithm.

Recalling the indexing of the leaves, indexed by $s \in \{0, 1, \ldots N-1\}$, we initialize a counter $\textsc{Desire}_i.\textit{position}$ to be 2^{ih}, indicating which Merkle tree leaf is to be computed next.

Algorithm 5.3 Key-Gen and Setup

1. **Initial Subtrees** For each $i \in \{1, 2, \ldots L\}$:
   - 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. Each round of the execution phase consists of the following portions: *generating an output*, *death and reincarnation of existing subtrees*, and *growing desired subtrees*.

At round s, the output consists of the authentication path associated to the sth leaf. The pebbles for this authentication path will be contained in the existing subtrees.

When the last authentication path requiring pebbles from a given existing subtree has been output, then the subtree is no longer useful, and we say that it "*dies*." By then, the corresponding desired subtree has been completed, and the recently died existing subtree "*reincarnates*" as this completed desired subtree. Notice that a new subtree at level i is needed once every 2^{ih} rounds, and so once per 2^{ih} rounds the pebbles in the existing tree are discarded. More technically, at round s, $s = 0 \pmod{2^{ih}}$ the pebbles in the old tree $\textsc{Exist}_i$ are discarded; the completed tree $\textsc{Desire}_i$ becomes the new tree $\textsc{Exist}_i$; and a new, empty desired subtree is created.

In the last step we grow each desired subtree that is not yet completed a little bit. More specifically, we apply two computational units to the new or already started invocations of the treehash algorithm. We concisely present this algorithm as follows:

Algorithm 5.4 Stratified Merkle Tree Traversal

1. Set $s = 0$.
2. **Output** Authentication Path for leaf number s.
3. **Next Subtree** For each $i \in \{1, 2, \dots L\}$ for which EXIST_i is no longer needed, i.e, $s = 0 \pmod{2^{hi}}$:
   - Remove Pebbles in EXIST_i.
   - Rename tree DESIRE_i as tree EXIST_i.
   - Create new, empty tree DESIRE_i (if $s + 2^{hi} < 2^H$).
4. **Grow Subtrees** For each $i \in \{1, 2, \dots h\}$: Grow tree DESIRE_i by applying 2 units to the modified treehash algorithm (unless DESIRE_i is completed).
5. Increment s and loop back to step 2 (while $s < 2^H$).

Time and Space Analysis

Time. As presented above, the algorithm allocates 2 computational units to each desired subtree. Here, a computational unit is defined to be either a call to LEAFCALC, or the computation of a hash value. Since there are at most $L - 1$ desired subtrees, the total computational cost per round is

$$T_{\max} = 2(L - 1) < 2H/h. \tag{23}$$

Space. The total amount of space required by the algorithm, or equivalently, the number of available pebbles required, may be bounded by simply counting the contributions from (1) the existing subtrees, (2) the desired subtrees, and (3) the tails.

First, there are L existing subtrees and up to $L - 1$ desired subtrees, and each of these contains up to $2^{h+1} - 2$ pebbles, since we do not store the roots. Additionally, the tail associated to a desired subtree at level $i > 1$ contains at most $h \cdot i + 1$ pebbles. If we count only the pebbles in the tail which do not belong to the desired subtree, then this "proper" tail contains at most $h(i - 1) + 1$ pebbles. Adding these contributions, we obtain the sum $(2L - 1)(2^{h+1} - 2) + h \sum_{i=1}^{L-2} i + 1$, and thus the bound:

$$\text{Space}_{\max} \leq (2L - 1)(2^{h+1} - 2) + L - 2 + h(L - 2)(L - 1)/2. \tag{24}$$

A marginally worse bound is simpler to write:

$$\text{Space}_{\max} < 2\,L\,2^{h+1} + H\,L\,/2. \tag{25}$$

Trade-offs. The solution just analyzed presents us with a trade-off between time and space. In general, the larger the subtrees are, the faster the algorithm will run, but the larger the space requirement will be. The parameter affecting the space and time in this

trade-off is h; in terms of h the computational cost is below $2H/h$, the space required is bounded above by $2\,L\,2^{h+1} + H\,L/2$. Alternatively, and in terms of h, the space is bounded above by $2\,H\,2^{h+1}/h + H^2/2\,h$.

Low Space Solution. If one is interested in parameters requiring little space, there is an optimal h, due to the fact that for very small h, the number of tail pebbles increases significantly (when $H^2/2h$ becomes large). An approximation of this value is $h = \log H$. One could find the exact value by differentiating the expression for the space: $2\,H\,2^{h+1}/h + H^2/2\,h$. For this choice of $h = \log H = \log\log N$, we obtain

$$T_{\max} = \frac{2H}{\log H}. \tag{26}$$

$$\text{Space}_{\max} \leq \frac{5}{2} \cdot \frac{H^2}{\log H}. \tag{27}$$

These results are interesting because they asymptotically improve Merkle's result from Section 5.1 with respect to both space and time. Merkle's approach required $T_{\max} = 2H$ and $\text{Space}_{\max} \approx H^2/2$.

Additional Savings

We now return to the main algorithm, and explain how a small technical modification will improve the constants in the space bound, ultimately yielding the claimed result.

Although this modification does not affect the complexity *class* of either the space or time costs, it is of practical interest as it nearly halves the space bound in certain cases. It is presented after the main exposition in order to retain the original simplicity, as this analysis is slightly more technical. The modification is based on two observations: (1) There may be pebbles in existing subtrees which are no longer useful, and (2) The desired subtrees are always in a state of partial completion. In fact, we have found that pebbles in an existing subtree may be discarded *nearly* as fast as pebbles are entered into the corresponding desired subtree. The modifications are as follows:

1. Discard pebbles in the trees $\textsc{Exist}_i$ as soon as they will never again be required.
2. Omit the *first* application of 2 units to the modified treehash algorithm.

We note that with the second modification, the desired subtrees still complete, just in time. With these small changes, for all levels $i < L$, the number of pebbles contained in *both* $\textsc{Exist}_i$, and $\textsc{Desire}_i$ can be bounded by the following expression.

$$\text{Space}_{\textsc{Exist}_i} + \text{Space}_{\textsc{Desire}_i} \leq 2^{ih+1} - 2 + (h-2). \tag{28}$$

This is nearly half of the previous bound of $2 \cdot (2^{ih+1} - 2)$. We remark here that the quantity $h-2$ measures the maximum number of pebbles contained in $\textsc{Desire}_i$ *exceeding* the number of pebbles contained in $\textsc{Exist}_i$ which have been discarded. Using the estimate (28), we revise the space bound computed in the previous section to be

$$\text{Space}_{\max} \leq (L)(2^{h+1} - 2) + (L-1)(h-2) + L - 2 + h(L-2)(L-1)/2. \tag{29}$$

We again round this up to obtain a simpler bound.

$$\text{Space}_{\max} < L\,2^{h+1} H\,L\,/2. \tag{30}$$

Specializing to the choice $h = \log H$, we improve the above result to

$$\text{Space}_{\max} \leq \frac{3}{2} \cdot \frac{H^2}{\log H}. \tag{31}$$

by reducing the constant from 5/2 to 3/2.

Proof of Space Bound. Here we prove the assertion of Equation (28) which states for any level i the number of pebbles in the $\textsc{Exist}_i$ plus the number of pebbles in the $\textsc{Desire}_i$ is less than $2 \cdot 2^{hi} - 2 + (h-2)$. This basic observation reflects the fact that the desired subtree can grow only slightly faster than the existing subtree shrinks. Without loss of generality, in order to simplify the exposition, we do not specify the subtree indices, and restrict our attention to the first existing-desired subtree pair at a given level i.

The first modification ensures that pebbles are returned more continuously than previously, so we quantify this. Subtree $\textsc{Exist}_i$, has 2^h leaves, and as each leaf is no longer required, neither may be some interior nodes above it. These leaves are finished at rounds $2^{(i-1)h}a - 1$ for $a \in \{1, \ldots 2^h\}$. We may determine the number of pebbles returned at these times by observing that a leaf is returned every single round, a pebble at height $i\,h + 1$ every two rounds, one at height $i\,h + 2$ every four rounds, etc. We are interested in the number returned at all times *up to* the time $2^{(i-1)h}a - 1$; this is the sum of the greatest integer functions:

$$A + [A/2] + [A/4] + [A/8] + \ldots + [A/2^h]$$

Writing a in binary notation $a = a_0 + 2^1 a_1 + 2^2 a_2 + \ldots + 2^h a_h$, this sum is also

$$a_0(2^1 - 1) + a_1 \cdot (2^2 - 1) + a_2 \cdot (2^3 - 1) + \ldots + a_h(2^{h+1} - 1).$$

The cost to calculate the corresponding pebbles in $\textsc{Desire}_i$ may also be calculated with a similar expression. Using the fact that a height h_0 node needs $2^{h_0+1} - 1$ units to compute, we see that the desired subtree requires

$$a_0(2^{(i-1)h+1} - 1) + a_1(2 \cdot 2^{(i-1)h+2} - 1) + \ldots + a_h(2 \cdot 2^{ih+1} - 1)$$

computational units to place those same pebbles. This cost is equal to $2 \cdot 2^{(i-1)h} \cdot a - z$, where z denotes the number of nonzero digits in the binary expansion of a.

At time $2^{(i-1)\,h}a - 1$, a total of $2 \cdot 2^{(i-1)\,h}a - 2$ units of computation has been applied to $\textsc{Desire}_i$, (factoring in our 1 round delay). Noting that $2^{(i-1)\,h} - 1$ more rounds may pass before $\textsc{Exist}_i$ loses any more pebbles, we see that the maximal number of pebbles during this interval must be realized at the very end of this interval. At this point in time, the desired subtree has computed exactly the pebbles that have been removed from the existing tree, plus whatever additional pebbles it can compute with its remaining $2 \cdot 2^{ih} - 2 + z - 2$ computational units. The next pebble, (a leaf) costs $2 \cdot 2^{ih} - 1$ which leaves $z - 3$ computational units. Even if all of these units result in new pebbles, the total extra is still less than or equal to $1 + z - 3$. Since $z \leq h$, this number of extra pebbles is bounded by $h - 2$, as claimed, and Equation (28) is proved.

5.3 Merkle Tree Traversal in Log Space and Time

Let us make some observations about the classic traversal algorithm from Section 5.1. We see that with the classic algorithm above, up to H instances of the treehash algorithm may be concurrently active, one for each height less than H. One can conceptualize them as H processes running in parallel, each requiring also a certain amount of space for the "tail

nodes" of the treehash algorithm, and receiving a budget of two hash value computations per round, clearly enough to complete the $2^{h+1} - 1$ hash computations required over the 2^h available rounds.

Because the stack employed by treehash may contain up to $h + 1$ node values, we are only guaranteed a space bound of $1 + 2 + \cdots + H$. The possibility of so many tail nodes is the source of the $\Omega(H^2/2)$ space complexity in the classic algorithm.

Considering that for the larger h, the treehash calculations have many rounds to complete, it appears that it might be wasteful to save so many intermediate nodes at once. Our idea is to schedule the concurrent treehash calculations differently, so that at any given round $s \in \{0, \ldots, 2^H - 1\}$, the associated stacks are mostly empty. We chose a schedule which generally favors computation of upcoming authentication nodes AUTH_h for lower h, (because they are required sooner), but delays beginning of a new instance of the treehash algorithm slightly, waiting until all stacks STACK_i are partially completed, containing no tail nodes of height less than h.

This delay, was motivated by the observation that in general, if the computation of two nodes at the same height in different treehash stacks are computed serially, rather than in parallel, less space will be used. Informally, we call the delay in starting new stack computations "zipping up the tails". We will need to prove the fact, which is no longer obvious, that the upcoming needed nodes will always be ready in time.

The New Traversal Algorithm

In this section we describe the new scheduling algorithm. Comparing to the classic traversal algorithm, the only difference will be in how the budget of $2H$ hash function evaluations will be allocated among the potentially H concurrent treehash processes.

Define $\text{STACK}_h.low$ to be the height of the lowest node in STACK_h, except in two cases: if the stack is empty $\text{STACK}_h.low$ is defined to be h, and if the treehash algorithm has completed $\text{STACK}_h.low$ is defined to be ∞.

Using the idea of zipping up the tails, there is more than one way to invent a scheduling algorithm which will take advantage of this savings. The one we present here is not optimal, but it is simple to describe. Additional practical improvements are discussed in Section 5.5.

This version can be concisely described as follows. The upcoming needed authentication nodes are computed as in the classic traversal, but the various stacks do not all receive equal attention. Each treehash instance can be characterized as being either not started, partially completed, or completed.

Our schedule prefers to complete STACK_h for the lowest h values first, *unless another stack has a lower tail node*. We express this preference by defining $l_{\min}$ be the minimum of the h values $\text{STACK}_h.low$, then choosing to focus our attention on the smallest level h attaining this minimum. (setting $\text{STACK}_h.low = \infty$ for completed stacks effectively skips them over).

In other words, all stacks must be completed to a stage where there are no tail nodes at height h or less before we start a new STACK_h treehash computation. The final algorithm is summarized in Algorithm 5.5.

Correctness and Analysis

In this section we show that our computational budget of $2H - 1$ is indeed sufficient to complete every STACK_h computation before it is required as an authentication node. We also show that the space required for hash values is less than $3H$.

Algorithm 5.5 Logarithmic Merkle Tree Traversal

1. Set $s = 0$.
2. **Output:**

- For each $h \in [0, H-1]$ output AUTH_h.

3. **Refresh Auth Nodes:**
   For all h such that 2^h divides $s+1$:

- Set AUTH_h be the sole node value in STACK_h.
- Set startnode $= (s + 1 + 2^h) \oplus 2^h$.
- $\text{STACK}_h.\textit{initialize}(\text{startnode}, h)$.

4. **Build Stacks:**
   Repeat the following $2H - 1$ times:

- Let $l_{\min}$ be the minimum of $\text{STACK}_h.\textit{low}$.
- Let *focus* be the least h so $\text{STACK}_h.\textit{low} = l_{\min}$.
- $\text{STACK}_{\textit{focus}}.\textit{update}$.

5. **Loop:**

- Set $s = s + 1$.
- If $s < 2^H$ go to Step 2.

Nodes are Computed on Time. As presented above, the algorithm allocates exactly a budget of $2H-1$ computational units per round to spend updating the h stacks. Here, a computational unit is defined to be either a call to LEAFCALC, or the computation of a hash value. We do not model any extra expense due to complex leaf calculations.

To prove this, we focus on a given height h, and consider the period starting from the time STACK_h is created and ending at the time when the upcoming authentication node (denoted NEED_h here) is required to be completed. This is not immediately clear, due to the complicated scheduling algorithm. Our approach to prove that NEED_h is completed on time is to showing that the total budget over this period exceeds the cost of *all* nodes computed within this period which can be computed before NEED_h.

The node NEED_h itself costs only $2^{h+1}-1$ units, a tractable amount given that there are 2^h rounds between the time STACK_h is created, and the time by which NEED_h must be completed. However, a non trivial calculation is required, since in addition to the resources required by NEED_h, many other nodes compete for the total budget of $2H2^h$ computational units available in this period. These nodes include all the future needed nodes NEED_i, $(i < h)$, for lower levels. Finally there may be a partial contribution to a node $\text{NEED}_i, i > h$, so that its stack contains no low nodes by the time NEED_h is computed.

It is easy to count the number of such needed nodes in the interval, and we know the cost of each one. As for the contributions to higher stacks, we at least know that the cost to raise any low node to height h must be less than $2^{h+1}-1$ (the total cost of a height h node). We summarize these quantities and costs in the following figure.

Table 1: Nodes built during 2^h rounds for NEED_h.

Node Type	Quantity	Cost each
NEED_h	1	$2^{h+1}-1$
NEED_{h-1}	2	2^h-1
$\vdots$	$\vdots$	$\vdots$
NEED_k	2^{h-k}	$2^{k+1}-1$
$\vdots$	$\vdots$	$\vdots$
NEED_0	2^h	1
TAIL	1	$\leq 2^{h+1}-2$

We proceed to tally up the total cost incurred during the interval. Notice that the row beginning NEED_0 requires a total of 2^{h+1} computational units. For every other row in the node chart, the number of nodes of a given type multiplied by the cost per node is less than 2^{h+1}. There are $h+1$ such rows, so the total cost of all nodes represented in the chart is

$$\text{TotalCost}_h < (h+2)2^h. \tag{32}$$

For heights $h \leq H-2$, it is clear that this total cost is less than $2H2^H$. It is also true for the remaining case of $h = H-1$, because there are no tail nodes in this case.

We conclude that, as claimed, the budget of $2H-1$ units per round is indeed always sufficient to prepare NEED_h on time, for any $0 \leq h < H$.

Space is Bounded by 3H. Our motivation leading to this relatively complex scheduling is to use as little space as possible. To prove this, we simply add up the

quantities of each kind of node. We know there are always H nodes $\textsc{Auth}_h$. Let $C < H$ be the number of completed nodes $\textsc{Need}_h$.

$$\#\textsc{Auth}_i + \#\textsc{Need}_i = H + C. \tag{33}$$

We must finally consider the number of tail nodes in the $\textsc{Stack}_h$. As for these, we observe that since a $\textsc{Stack}_h$ never becomes active until all nodes in "higher" stacks are of height at least h, there can never be two distinct stacks, each containing a node of the same height. Furthermore, recalling algorithm treehash, we know there is at most one height for which a stack has two node values. In all, there is at most one tail node at each height ($0 \leq h \leq H - 3$), plus up to one additional tail node per non-completed stack. Thus

$$\#\textsc{Tail} \leq H - 2 + (H - C). \tag{34}$$

Adding all types of nodes we obtain:

$$\#\textsc{Auth}_i + \#\textsc{Need}_i + \#\textsc{Tail} \leq 3H - 2. \tag{35}$$

This proves the assertion. There are at most $3H - 2$ stored nodes.

5.4 Asymptotic Optimality Result

An interesting optimality result states that a traversal algorithm can never beat both time $O(\log(N))$ and space $O(\log(N))$. It is clear that at least $H - 2$ nodes are required for the treehash algorithm, so our task is essentially to show that if space is limited by any constant multiple of $\log(N)$, then the computational complexity must be $\Omega(\log(N))$. Let us be clear that this theorem does not quantify the constants. Clearly, with greater space, computation time can be reduced.

Theorem 5.1 *Suppose that there is a Merkle tree traversal algorithm for which the space is bounded by $\alpha \log(N)$. Then there exists some constant β so that the time required is at least $\beta \log(N)$.*

The theorem simply states that it is not possible to reduce space complexity below logarithmic without increasing the time complexity beyond logarithmic!

The proof of this technical statement is found in the upcoming subsection, but we will briefly describe the approach here. We consider only right nodes for the proof. We divide all right nodes into two groups: those which must be computed (at a cost of $2^{h+1} - 1$), and those which have been saved from some earlier calculation. The proof assumes a sub-logarithmic time complexity and derives a contradiction.

The more nodes in the second category, the faster the traversal can go. However, such a large quantity of nodes would be required to be saved in order to reduce the time complexity to sub-logarithmic, that the average number of saved node values would have to exceed a linear amount! The rather technical proof presented next uses a certain sequence of subtrees to formulate the contradiction.

We now begin the technical proof of Theorem 5.1. This will be a proof by contradiction. We assume that the time complexity is sub logarithmic, and show that this is incompatible with the assumption that the space complexity is $O(\log(N))$. Our strategy to produce a contradiction is to find a bound on some linear combination of the average time and the average amount of space consumed.

Notation. The theorem is an asymptotic statement, so we will be considering trees of height $H = \log(N)$, for large H. We need to consider L *levels* of subtrees of height k, where $kL = H$. Within the main tree, the roots of these subtrees will be at heights $k, 2 \cdot k, 3 \cdot k \ldots H$. We say that the subtree is at *level* i if its root is at height $(i+1)k$. This subtree notation is similar to that used in Section 5.2.

Note that we will only need to consider right nodes to complete our argument. Recall that during a complete tree traversal every single right node is eventually output as part of the authentication data. This prompts us to categorize the right nodes in three classes.

1. Those already present after the key generation: *free nodes*.
2. Those explicitly calculated (e.g. with treehash): *computed nodes*.
3. Those retained from another node's calculation (e.g from another node's treehash): *saved nodes*.

Notice how type 2 nodes require computational effort, whereas type 1 and type 3 nodes require some period of storage. We need further notation to conveniently reason about these nodes. Let a_i denote the number of level i subtrees which contain *at least 1* non-root computed (right) node. Similarly, let b_i denote the number of level i subtrees which contain *zero* computed nodes. Just by counting the total number of level i subtrees we have the relation.

$$a_i + b_i = N/2^{(i+1)k}. \tag{36}$$

Computational costs. Let us tally the cost of some of the computed nodes. There are a_i subtrees containing a node of type 2, which must be of height at least ik. Each such node will cost at least $2^{ik+1} - 1$ operations to compute. Rounding down, we find a simple lower bound for the cost of the nodes at level i.

$$Cost > \sum_{i=0}^{L-1} (a_i 2^{ik}). \tag{37}$$

Storage costs. Let us tally the lifespans of some of the retained nodes. Measuring units of *Space* $\times$ *Rounds* is natural when considering average space consumed. In general, a saved node, S, results from a calculation of some computed node C, say, located at height h. We know that S has been produced before C is even needed, and S will never become an authentication node before C is discarded. We conclude that such a node S must be therefore be stored in memory for at least 2^h rounds.

Even (most of) the free nodes at height h remain in memory for at least 2^{h+1} rounds. In fact, there can be at most one exception: the first right node at level h.

Now consider one of the b_i subtrees at level i containing only free or stored nodes. Except for the leftmost subtree at each level, which may contain a free node waiting in memory less than $2^{(i+1)k}$ rounds, every other node in this subtree takes up space for at least $2^{(i+1)k}$ rounds. There are $2^k - 1$ nodes in a subtree and thus we find a simple lower bound on the *Space* $\times$ *Rounds*.

$$Space \times Rounds \geq \sum_{i=0}^{L-1} (b_i - 1)(2^k - 1)2^{(i+1)k}. \tag{38}$$

Note that the $(b_i - 1)$ term reflects the possible omission of the leftmost level i subtree.

Mixed Bounds. We can now use simple algebra with Equations (36), (37), and (38) to yield combined bounds. First the cost is related to the b_i, which is then related to a space bound.

$$2^k \, Cost > \sum_{i=0}^{L-1} a_i 2^{(i+1)k} = \sum_{i=0}^{L-1} N - 2^{(i+1)k} b_i. \tag{39}$$

As series of similar algebraic manipulations finally yield (somewhat weaker) very useful bounds.

$$2^k \, Cost + \sum_{i=0}^{L-1} 2^{(i+1)k} b_i > NL. \tag{40}$$

$$2^k \, Cost + \sum_{i=0}^{L-1} \frac{2^{(i+1)k}}{2^{k-1}} + \frac{Space \times Rounds}{2^{k-1}} > NL \tag{41}$$

$$2^k \, Cost + 2N + \frac{Space \times Rounds}{2^{k-1}} > NL \tag{42}$$

$$2^k \, AverageCost + \frac{AverageSpace}{2^{k-1}} > (L-2) \geq \frac{L}{2} \tag{43}$$

$$k 2^{k+1} \, AverageCost + \frac{k}{2^{k-2}} AverageSpace > \frac{L}{2} \cdot 2k = H. \tag{44}$$

This last bound on the sum of average cost and space requirements will allow us to find a contradiction.

Proof by Contradiction. Let us assume the opposite of the statement of Theorem 5.1. Then there is some α such that the space is bounded above by $\alpha \log(N)$. Secondly, the time complexity is supposed to be sub-logarithmic, so for every small β the time required is less than $\beta \log(N)$ for sufficiently large N.

With these assumptions we are now able to choose a useful value of k. We pick k to be large enough so that $\alpha > 1/k2^{k+3}$. We also choose β to be less than $1/k2^{k+2}$. With these choices we obtain two relations.

$$k 2^{k+1} \, AverageCost < \frac{H}{2} \tag{45}$$

$$k/2^{k-2} \, AverageSpace < \frac{H}{2} \tag{46}$$

By adding these two last equations, we contradict Equation (44).
QED.

5.5 Improvement of the Log Traversal Algorithm

In this section we describe improvements of the algorithm described in Section 5.3 which are very useful for practical implementations. The main differences are the following. Since left authentication nodes can be computed much cheaper than right nodes, the computation of left and right authentication nodes is done differently. In many cases the number of expensive leaf computations is reduced. Instead of using a separate stack for each instance of the treehash algorithm one shared stack is used. Input for the algorithm is an index $s \in \{0, 1, \ldots, 2^H - 2\}$. The algorithm determines the authentication path $\text{AUTH} = (\text{AUTH}_0, \ldots, \text{AUTH}_{H-1})$ for leaf $s + 1$.

As before, we denote the nodes in the Merkle tree by $\nu_h[j]$, where $h = H, \ldots, 0$ denotes the height of the node in the tree of height H. Leaves have height 0 and the root has height H. MSS uses a cryptographic hash function $g : \{0,1\}^* \to \{0,1\}^n$.

The algorithm determines $\tau = \max\{h : 2^h | (s+1)\}$ which is the height of the first ancestor of the sth leaf which is a left child. If leaf s is a left child itself, then $\tau = 0$. Figure 7 shows an example.

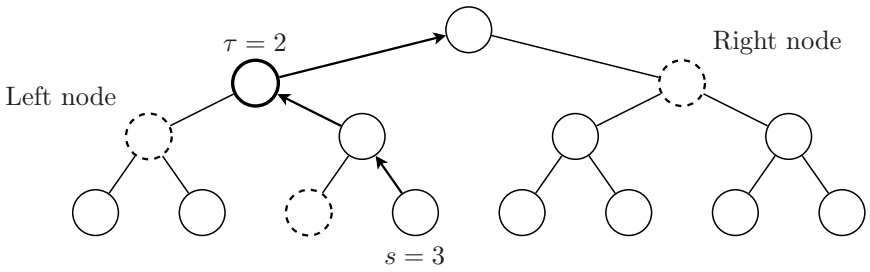

Figure 7: The height of the first ancestor of leaf s that is a left child is $\tau = 2$. The dashed nodes denote the authentication path for leaf s. The arrows indicate the path from leaf s to the root.

The value τ is used to determine on which heights the authentication path for leaf $s+1$ requires new nodes. The authentication path for leaf $s+1$ requires new right authentication nodes on heights $h = 0, \ldots, \tau - 1$ and one new left authentication node on height τ.

Computing left and right authentication nodes

Computing left nodes. As explained above, we require the left node AUTH_τ for the next authentication path. If $\tau = 0$, then we set AUTH_0 to $\text{LEAFCALC}(s)$. Let $\tau > 0$. Then leaf s is a right child. Also, $\text{AUTH}_{\tau-1}$ is the left child of AUTH_τ. We assume that the right child of AUTH_τ is stored in $\text{KEEP}_{\tau-1}$. Then the new node AUTH_τ is computed as

$$\text{AUTH}_\tau = g\big(\text{AUTH}_{\tau-1} \parallel \text{KEEP}_{\tau-1}\big). \tag{47}$$

This requires only one hash evaluation. We also explain how KEEP is updated. If $\lfloor s/2^{\tau+1} \rfloor = 0 \pmod 2$, i.e. if the ancestor on height $\tau + 1$ is a left child, then AUTH_τ is a right node and we store it in KEEP_τ.

Computing right nodes. Unlike authentication nodes that are left children, right authentication nodes are computed from scratch, i.e. starting from the leaves. This is because none of their child nodes were used in previous authentication paths. As before we use the treehash algorithm (Section 3, Algorithm 3.1) for this task.

We use two different methods for computing right nodes. To distinguish those cases we select a positive integer $K \geq 2$ such that $H - K$ is even. Suppose that we wish to compute a right node on height h. If $H - K \leq h \leq H - 2$, then the right node on height h is calculated by RETAIN_h.pop() which pops the top element from a stack RETAIN_h. That stack has been filled with the right nodes $\nu_h[3], \ldots, \nu_h[2^{H-h} - 1]$ during MSS key generation. This is very useful since the nodes close to the root are expensive to compute.

For the computation of a right node on height h with $h < H - K$ we use an instance TREEHASH_h of the treehash algorithm. It is allowed to store one node. Initially, during MSS key generation, the second right node $\nu_h[3]$ is stored in TREEHASH_h. The treehash

instances all share one stack. When it comes to determining a right authentication node on height h this is simply done by TREEHASH_h.pop() for $h = 0, \ldots, \min\{H-K-1, \tau-1\}$. Then all treehash instances for heights $h = 0, \ldots, \min\{H-K-1, \tau-1\}$ are initialized for the computation of the next right node. The index of the leaf they have to begin with is $s+1+3\cdot 2^h$ and the initialization is done using the method TREEHASH_h.initialize($s+1+3\cdot 2^h$). Then the algorithm updates the treehash instances using the TREEHASH_h.update() method. One update corresponds to one round of Algorithm 3.1, i.e. to computing one leaf and computing this leaf's parent nodes using tail nodes stored on the stack.

We allow a budget of $(H-K)/2$ updates in each round. We use the strategy from Section 5.3 to decide which of the $H-K$ treehash instances receives an update. For this, we need the method TREEHASH_h.height() which returns the height of the lowest tail node stored by this treehash instance, either on the stack or in the treehash instance itself. If TREEHASH_h does not store any tail nodes TREEHASH_h.height() returns h and if TREEHASH_h is finished or not initialized TREEHASH_h.height() returns ∞ to skip these instances. The treehash instance that receives an update is the instance where TREEHASH_h.height() returns the smallest value. If there is more than one such instance, we choose the one with the lowest index.

The algorithm

Initialization. The initialization of our algorithm is done during the MSS key pair generation. We store the authentication path for the first leaf ($s=0$): $\text{AUTH}_h = \nu_h[1], h = 0, \ldots, H-1$. Depending on the parameter K, we store the next right authentication node for each height $h = 0, \ldots, H-K-1$ in the treehash instances: TREEHASH_h.push($\nu_h[3]$). Finally we store the right authentication nodes close to the root using the stacks RETAIN_h: RETAIN_h.push($\nu_h[2j+3]$) for $h = H-K, \ldots, H-2$ and $j = 2^{H-h-1}-2, \ldots, 0$.

Update and output phase. Algorithm 5.6 contains the precise description. Input is the index of the current leaf $s \in \{0, \ldots, 2^H-2\}$, the parameters H, K and the algorithm state AUTH, KEEP, RETAIN, TREEHASH prepared in previous rounds or the initialization. Output is the authentication path for the next leaf $s+1$.

Correctness and analysis

In this section we show the correctness of Algorithm 5.6 and estimate its time and space requirements. First we show that the budget of $(H-K)/2$ updates per round is sufficient for the treehash instances to compute the nodes on time. Then we show that it is possible for all treehash instances to share a single stack. Next, we consider the time and space requirements of Algorithm 5.6. In detail we show that

i) The number of tail nodes stored on the stack is bounded by $H-K-2$.

ii) The number of hashes per round is bounded by $3(H-K-1)/2$.

iii) The number of nodes stored in KEEP is bounded by $\lfloor H/2 \rfloor + 1$.

To estimate the space complexity, we have to add the H nodes stored in AUTH, the $H-K$ nodes stored in TREEHASH and the 2^K-K-1 nodes stored in RETAIN. To estimate the time complexity, we have to add the $(H-K)/2$ leaf computations required to determine right nodes and one leaf and one hash to compute left nodes (Lines 3, 4a in Algorithm 5.6). Summing up the total time and space requirements results in the following theorem.

Algorithm 5.6 Authentication path computation

Input: $s \in \{0, \ldots, 2^H - 2\}$, H, K and the algorithm state.

Output: Authentication path for leaf $s + 1$

1. Let $\tau = 0$ if leaf s is a left node or let τ be the height of the first parent of leaf s which is a left node:
   $\tau \leftarrow \max\{h : 2^h | (s + 1)\}$
2. If the parent of leaf s on height $\tau + 1$ is a left node, store the current authentication node on height τ in KEEP_τ:
   if $\lfloor s/2^{\tau+1} \rfloor$ is even **and** $\tau < H - 1$ **then** $\text{KEEP}_\tau \leftarrow \text{AUTH}_\tau$
3. If leaf s is a left node, it is required for the authentication path of leaf $s + 1$:
   if $\tau = 0$ **then** $\text{AUTH}_0 \leftarrow \text{LEAFCALC}(s)$
4. Otherwise, if leaf s is a right node, the authentication path for leaf $s + 1$ changes on heights $0, \ldots, \tau$:
   if $\tau > 0$ **then**
   (a) The authentication path for leaf $s+1$ requires a new left node on height τ. It is computed using the current authentication node on height $\tau - 1$ and the node on height $\tau - 1$ previously stored in $\text{KEEP}_{\tau-1}$. The node stored in $\text{KEEP}_{\tau-1}$ can then be removed:
   $\text{AUTH}_\tau \leftarrow g(\text{AUTH}_{\tau-1} || \text{KEEP}_{\tau-1})$, remove $\text{KEEP}_{\tau-1}$
   (b) The authentication path for leaf $s + 1$ requires new right nodes on heights $h = 0, \ldots, \tau - 1$. For $h < H - K$ these nodes are stored in TREEHASH_h and for $h \geq H - K$ in RETAIN_h:
   for $h = 0$ **to** $\tau - 1$ **do**
   if $h < H - K$ **then** $\text{AUTH}_h \leftarrow \text{TREEHASH}_h.\text{pop}()$
   if $h \geq H - K$ **then** $\text{AUTH}_h \leftarrow \text{RETAIN}_h.\text{pop}()$
   (c) For heights $0, \ldots, \min\{\tau-1, H-K-1\}$ the treehash instances must be initialized anew. The treehash instance on height h is initialized with the start index $s + 1 + 3 \cdot 2^h < 2^H$:
   for $h = 0$ **to** $\min\{\tau - 1, H - K - 1\}$ **do** $\text{TREEHASH}_h.\text{initialize}(s + 1 + 3 \cdot 2^h)$
5. Next we spend the budget of $(H-K)/2$ updates on the treehash instances to prepare upcoming authentication nodes:
   repeat $(H - K)/2$ **times**
   (a) We consider only stacks which are initialized and not finished. Let k be the index of the treehash instance whose lowest tail node has the lowest height. In case there is more than one such instance we choose the instance with the lowest index:
   $k \leftarrow \min \left\{ h : \text{TREEHASH}_h.\text{height}() = \min_{j=0,\ldots,H-K-1} \{\text{TREEHASH}_j.\text{height}()\} \right\}$
   (b) The treehash instance with index k receives one update:
   $\text{TREEHASH}_k.\text{update}()$
6. The last step is to output the authentication path for leaf $s + 1$:
   return $\text{AUTH}_0, \ldots, \text{AUTH}_{H-1}$.

Theorem 5.2 *Let $H \geq 2$ and $K \geq 2$ such that $H - K$ is even. Algorithm 5.6 stores at most $3H + \lfloor H/2 \rfloor - 3K - 2 + 2^K$ nodes, where each node requires n bits of memory. Further, the algorithm requires at most $(H - K)/2 + 1$ leaf computations and $3(H - K - 1)/2 + 1$ hash function evaluations per round to successively compute authentication paths.*

Nodes are computed on time. If TREEHASH_h is initialized in round s, the authentication node on height h computed by this instance is required in round $s + 2^{h+1}$. In these 2^{h+1} rounds there are $(H - K)2^h$ updates available. TREEHASH_h requires 2^h updates. During the 2^{h+1} rounds, $2^{h+1}/2^{i+1}$ treehash instances are initialized on heights $i = 0, \ldots, h-1$, each requiring 2^i updates. In addition, active treehash instances on heights $i = h + 1, \ldots, H - K - 1$ might receive updates until their lowest tail node has height h, thus requiring at most 2^h updates.

Summing up the number of updates required by all treehash instances yields

$$\sum_{i=0}^{h-1} \frac{2^{h+1}}{2^{i+1}} \cdot 2^i + 2^h + \sum_{i=h+1}^{H-K-1} 2^h = (H - K)2^h \tag{48}$$

as an upper bound for the number of updates required to finish TREEHASH_h on time. For $h = H - K - 1$ this bound is tight.

Sharing a single stack works. To show that it is possible for all treehash instances to share a single stack, we have to show that if TREEHASH_h receives an update and has tail nodes stored on the stack, all these tail nodes are on top of the stack.

When TREEHASH_h receives its first update, the height of the lowest tail node of TREEHASH_i, $i \in \{h + 1, \ldots, H - K - 1\}$ is at least h. This means that TREEHASH_h is completed before TREEHASH_i receives another update and thus tail nodes of higher treehash instances do not interfere with tail nodes of TREEHASH_h.

While TREEHASH_h is active and stores tail nodes on the stack, it is possible that treehash instances on lower heights $i \in \{0, \ldots, h-1\}$ receive updates and store nodes on the stack. If TREEHASH_i receives an update, the height of the lowest tail node of TREEHASH_h has height $\geq i$. This implies that TREEHASH_i is completed before TREEHASH_h receives another update and therefore doesn't store any tail nodes on the stack.

Space required by the stack. We will show that the stack stores at most one tail node on each height $h = 0, \ldots, H - K - 3$ at a time. TREEHASH_h, $h \in \{0, \ldots, H - K - 1\}$ stores up to h tail nodes on different heights to compute the authentication node on height h. The tail node on height $h - 1$ is stored by the treehash instance and the remaining tail nodes on heights $0, \ldots, h - 2$ are stored on the stack. When TREEHASH_h receives its first update, the following two conditions hold: (1) all treehash instances on heights $< h$ are either empty or completed and store no tail nodes on the stack. (2) All treehash instances on heights $> h$ are either empty or completed or have tail nodes of height at least h. If a treehash instance on height $i \in \{h+1, \ldots, H - K - 1\}$ stores a tail node on the stack, then all treehash instances on heights $i + 1, \ldots, H - K - 1$ have tail nodes of height at least i, otherwise the treehash instance on height i wouldn't have received any updates in the first place. This shows that there is at most one tail node on each height $h = 0, \ldots, H - K - 3$ which bounds the number of nodes stored on the stack by $H - K - 2$. This bound is tight for round $s = 2^{H-K+1} - 2$, before the update that completes the treehash instance on height $H - K - 1$.

Number of hashes required per round. For now we assume that the maximum number of hash function evaluations is required in the following case: TREEHASH_{H-K-1} receives all $u = (H-K)/2$ updates and is completed in this round. On input an index s, the number of hashes required by the treehash algorithm is equal to the height of the first parent of leaf s which is a left node. On height h, a left node occurs every 2^h leaves, which means that every 2^h updates at least h hashes are required by treehash. During the u available updates, there are $\lceil u/2^h \rceil$ updates that require at least h hashes for $h = 1, \ldots, \lceil \log_2 u \rceil$. The last update requires $H - K - 1 = 2u - 1$ hashes to complete the treehash instance on height $H - K - 1$. So far only $\lceil \log_2 u \rceil$ of these hashes were counted, so we have to add another $2u - 1 - \lceil \log_2 u \rceil$ hashes. In total, we get the following upper bound for the number of hashes required per round.

$$B = \sum_{h=1}^{\lceil \log_2 u \rceil} \left\lceil \frac{u}{2^h} \right\rceil + 2u - 1 - \lceil \log_2 u \rceil \tag{49}$$

In round $s = 2^{H-K+1} - 2$ this bound is tight. This is the last round before the treehash instance on height $H - K - 1$ must be completed and as explained above, all available updates are required in this case. The desired upper bound is estimated as follows:

$$\begin{aligned} B &\leq \sum_{h=1}^{\lceil \log_2 u \rceil} \left(\frac{u}{2^h} + 1 \right) + 2u - 1 - \lceil \log_2 u \rceil \\ &= u \sum_{h=1}^{\lceil \log_2 u \rceil} \frac{1}{2^h} + 2u - 1 = u \left(1 - \frac{1}{2^{\lceil \log_2 u \rceil}} \right) + 2u - 1 \\ &\leq u \left(1 - \frac{1}{2u} \right) + 2u - 1 = 3u - \frac{3}{2} = \frac{3}{2}(H - K - 1) \end{aligned}$$

The next step is to show that the above mentioned case is indeed the worst case. If a treehash instance on height $< H - K - 1$ receives all updates and is completed in this round, less than B hashes are required. The same holds if the treehash instance receives all updates but is not completed in this round. The last case to consider is the one where the u available updates are spend on treehash instances on different heights. If the active treehash instance has a tail node on height j, it will receive updates until it has a tail node on height $j + 1$, which requires 2^j updates and 2^j hashes. Additional $t \in \{1, \ldots, H - K - j - 2\}$ hashes are required to compute the parent of this node on height $j + t + 1$, if the active treehash instance stores tail nodes on heights $j + 1, \ldots, j + t$ on the stack and in the treehash instance itself. The next treehash instance that receives updates has a tail node of height $\geq j$. Since the stack stores at most one tail node for each height, this instance can receive additional hashes only if there are enough updates to compute a tail node on height $\geq j + t$, the height of the next tail node possibly stored on the stack. But this is the same scenario that appears in the above mentioned worst case, i.e. if a node on height $j + 1$ is computed, the tail nodes on the stack are used to compute its parent on height $j + t + 1$ and the same instance receives the next update.

Space required to compute left nodes. First we show that whenever an authentication node is stored in KEEP_h, $h = 1, \ldots, H-2$, the node stored in KEEP_{h-1} is removed in the same round. This immediately follows from Steps 2 and 4a in Algorithm 5.6. Second we show that if a node gets stored in KEEP_h, $h = 0, \ldots, H - 3$, then KEEP_{h+1} is empty. To see this we have to consider in which rounds a node is stored in KEEP_{h+1}. This is

true for rounds $s \in A_a = \{2^{h+1} - 1 + a \cdot 2^{h+3}, \ldots, 2^{h+2} - 1 + a \cdot 2^{h+3}\}, a \in \mathbb{N}_0$. In rounds $s' = 2^h - 1 + b \cdot 2^{h+2}, b \in \mathbb{N}_0$, a node gets stored in KEEP_h. It is straight forward to compute that $s' \in A_a$ implies that $2a + 1/4 \leq b \leq 2a + 3/4$ which is a contradiction to $b \in \mathbb{N}_0$.

As a result, at most $\lfloor H/2 \rfloor$ nodes are stored in KEEP at a time and two consecutive nodes can share one entry. One additional entry is required to temporarily store the authentication node on height h (Step 2) until node on height $h - 1$ is removed (Step 4a).

Computing leaves using an PRNG

In Section 4, we showed how a PRNG can be used during MSS key pair and signature generation to reduce the private key size. We will now show how to use this concept in Algorithm 5.6 to compute the required leaves using an PRNG. Let SEED_s denote the seed required to compute the one-time key pair corresponding to the sth leaf.

During the authentication path computation, leaves which are up to $3 \cdot 2^{H-K-1}$ steps away from the current leaf must be computed by the treehash instances. Calling the PRNG that many times to obtain the seed required to compute this leaf is too inefficient. Instead we use the following scheduling strategy that requires $H - K$ calls to the PRNG in each round to compute the seeds. We have to store two seeds for each height $h = 0, \ldots, H - K - 1$. The first (SEEDACTIVE) is used to successively compute the leaves for the authentication node currently constructed by TREEHASH_h and the second (SEEDNEXT) is used for upcoming right nodes on this height. SEEDNEXT is updated using the PRNG in each round. During the initialization, we set $\text{SEEDNEXT}_h = \text{SEED}_{3 \cdot 2^h}$ for $h = 0, \ldots, H - K - 1$. In each round, at first all seeds SEEDNEXT_h are updated using the PRNG. If in round s a new treehash instance is initialized on height h, we copy SEEDNEXT_h to SEEDACTIVE_h. In that case $\text{SEEDNEXT}_h = \text{SEED}_{\varphi+1+3 \cdot 2^h}$ holds and thus is the correct seed to begin computing the next authentication node on height h.

The time and space requirements of Algorithm 5.6 change as follows. We have to store additional $2(H - K)$ seeds and each seed requires n bit of memory. We also require additional $H - K$ calls to the PRNG in each round.

Theorem 5.3 *Let $H \geq 2$ and $K \geq 2$ such that $H - K$ is even. The memory requirements of Algorithm 5.6 in combination with a PRNG are*

$$\left(5H + \left\lfloor \frac{H}{2} \right\rfloor - 5K - 2 + 2^K\right) \cdot n \text{ bit.} \tag{50}$$

Further, it requires at most $(H - K)/2 + 1$ leaf computations, $3(H - K - 1)/2 + 1$ hash function evaluations, and $H - K$ calls to the PRNG per round to successively compute authentication paths.

6 Tree chaining

In Section 3 we saw that MSS public key generation requires the computation of the full Merkle hash tree. This means that 2^H leaves and $2^H - 1$ inner nodes have to be determined, which is very time consuming when H is large. The *tree chaining* method [4] solves this problem. The basic idea is similar to the Fractal Merkle Tree Traversal described in Section 5.2. However, in contrast to the Fractal Tree Traversal Method, tree chaining does not split the Merkle tree into smaller subtrees, but instead uses smaller Merkle trees that are independent of each other. The Merkle signature scheme that uses tree chaining is referred to as CMSS.

The idea

We explain the tree chaining idea. CMSS uses $T \geq 2$ *layers* of Merkle trees. Each Merkle tree on each layer is constructed using the Method from Sections 3 and 4. The hashes of a sequence of one-time verification keys are the leafs. We call the corresponding one-time signature keys the *signature keys of the Merkle tree*. Those signature keys are calculated using a pseudo random number generator. We call the respective seed the *seed of the Merke tree*.

The root of the single tree on the top layer 1 is the public CMSS key. The signature keys of the Merkle trees on the bottom layer T are used to sign documents. The signature keys of the Merkle trees on the intermediate layers i, $1 \leq i < T$ sign the roots of the Merkle trees on layer $i+1$.

This is what a tree chaining signature looks like:

$$\begin{aligned} \sigma \quad = \big(s, \quad & \textsc{Sig}_T, Y_T, \textsc{Auth}_T \\ & \textsc{Sig}_{T-1}, Y_{T-1}, \textsc{Auth}_{T-1} \\ & \quad \vdots \\ & \textsc{Sig}_1, Y_1, \textsc{Auth}_1\big). \end{aligned} \tag{51}$$

$\textsc{Sig}_T$ is the one-time signature of the document to be signed. It is generated using a signature key of a Merkle tree on the bottom layer T. The corresponding verification key is Y_T. Also, $\textsc{Auth}_T$ is the authentication path that allows a verifier to construct the path from the verification key Y_T to the root of the corresponding Merkle tree on the bottom layer. Now that root is not known to the verifier. Therefore, the one-time signature $\textsc{Sig}_{T-1}$ of that root is also included in the signature σ. It is constructed using a signature key of a Merkle tree on level $T-1$. The corresponding verification key Y_{T-1} and authentication path $\textsc{Auth}_{T-1}$ are also included in the signature σ. The root of the tree on layer $T-1$ is also not known to the verifier, unless $T = 2$ in which case $T-1 = 1$ and that root is the public key. So further one-time signatures of roots $\textsc{Sig}_i$, one-time verification keys Y_i, and authentication paths $\textsc{Auth}_i$, $i = T-1, \ldots, 1$ are included in the signature σ.

The signature σ is verified as follows. The verifier checks, that $\textsc{Sig}_T$ can be verified using Y_T. Next, he uses Y_T and $\textsc{Auth}_T$ to construct the root of a Merkle tree on layer T. He verifies the signature $\textsc{Sig}_{T-1}$ of that root using the verification key Y_{T-1} and constructs the root of the corresponding Merkle tree on layer $T-1$ from Y_{T-1} and $\textsc{Auth}_{T-1}$. The verifier iterates this procedure until the root of the single tree on layer 1 is constructed. The signature is verified by comparing this root to the public key. If any of those comparisons fails then the signature σ is rejected. Otherwise, it is accepted.

We discuss the advantage of the tree chaining method. For this purpose, we first compute the number of signatures that can be verified using one public key when the tree chaining method is applied. All Merkle trees on layer i have the same height H_i, $1 \leq i \leq T$. As mentioned already, there is a single Merkle tree on the top layer 1. Since the Merkle trees on layer i are used to sign the roots of the Merkle trees on layer $i+1$, $1 \leq i < T$, the number of Merkle trees on layer $i+1$ is $2^{H_1+H_2+\ldots+H_i}$. So the total number of documents that can be signed/verified is 2^H where $H = H_1 + H_2 + \ldots + H_T$.

The advantage of the tree chaining construction is the following. The generation of a public MSS key that can verify 2^H documents requires the construction of a tree of height H, which in turn requires the computation of 2^H one-time key pairs and $2^{H+1} - 1$ evaluations of the hash fuction. When tree chaining is used, the construction of a public CMSS key that can verify 2^H documents only requires the construction of the single Merkle tree on the top layer which is of height H_1. Also, in the tree chaining method, signature

generation requires knowledge of the one-time signature of the root of one Merkle tree on each layer. Those roots and one-time signatures can be successively computed as they are used, whereas the root of the first tree on each layer is generated during the key generation. Hence, the CMSS key pair generation requires the computation of $2^{H_1} + \ldots + 2^{H_T}$ one-time key pairs and $2^{H_1+1} + \ldots + 2^{H_T+1} - T$ evaluations of the hash function. This is a drastic improvement compared to the original MSS key pair generation as illustrated in the following example.

Example 6.1 *Assume that the heights of all Merkle trees are equal, so* $H_1 = \ldots = H_T = H$*. The number of signatures that can be generated with this key pair is* 2^{TH}*. The CMSS key pair generation requires* $T2^H$ *one-time key pairs and* $T2^{H+1} - T$ *evaluations of the hash function. The original MSS key pair generation requires* 2^{TH} *one-time key pairs and* $2^{TH+1} - 1$ *evaluations of the hash function.*

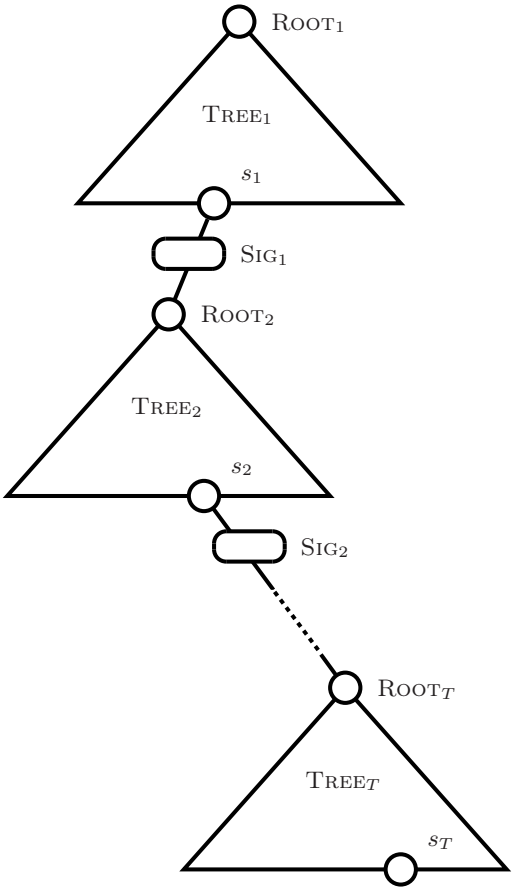

Figure 8: The tree chaining method. TREE_i denotes the active tree on layer i, ROOT_i its root, and SIG_{i-1} this root's one-time signature generated with the s_{i-1}th signature key of the tree on layer $i-1$.

CMSS key pair generation

For the CMSS key pair generation, the number of layers T and the respective heights H_i, $1 \leq i \leq T$ of the trees on layer i are selected. With $H = H_1 + H_2 + \ldots + H_T$ the number of

signatures that can be generated/verified using the key pair to be constructed is 2^H. For each layer, one initial Merkle tree TREE_i is constructed as described in Sections 3 and 4. The CMSS public key is the root of TREE_1. The CMSS secret key is the sequence of the random seeds used to construct the T trees. The signer also stores the one-time signatures of the roots of all those trees generated with the first signature key of the tree on the next layer.

CMSS key pair generation requires the computation of $2^{H_1} + \ldots + 2^{H_T}$ one-time key pairs and $2^{H_1+1} + \ldots + 2^{H_T+1} - T$ evaluations of the hash function.

CMSS signature generation

We use the notation of the previous sections. When a signature is issued, the signer knows one active Merkle tree TREE_i for each layer and the seed SEED_i from which its signature keys can be generated, $i = 1, 2, \ldots, T$. The signer also knows the signature SIG_i of the root of TREE_{i+1}, and the verification key Y_i for that signature, $1 \leq i \leq T-1$. Further, the signer knows the index s_i, $1 \leq i \leq T-1$, of the signature key used to generate the signature SIG_i of the root of the tree TREE_{i+1} and the index s_T of the signature key used to issue the next document signature. The signer constructs the corresponding signature key from the seed SEED_T, he generates the one-time signature SIG_T of the document to be signed and he generates the signature as in Equation (51). The index s in this signature can be recursively computed. Set $t_1 = s_1$ and

$$t_{i+1} = t_i 2^{H_{i+1}} + s_{i+1}, 1 \leq i < T,$$

then $s = t_T$.

After signing, the signer prepares for the next signature by partially constructing the next tree on certain layers using the treehash algorithm of Section 3. He first computes the s_Tth leaf of the next tree on layer T and executes the treehash algorithm with this leaf as input. Then he increments s_T. If $s_T = 2^{H_T}$, then the construction of the next Merkle tree on layer T is completed and its root is available. The signer computes the one-time signature of this root using a signature key of the tree on layer $T-1$ and sets the index s_T to zero. In the same way, the signer constructs the next tree on layer $T-1$ and increments the index s_{T-1}. More generally, the signer partially constructs the next tree on layer i and increments s_i whenever the construction of the next tree on layer $i+1$ is complete, $1 < i < T$. On layer 1, no new tree is required and the signer only increments the index s_1 if the construction of a tree on layer 2 is completed. When $s_1 = 2^{H_1}$, CMSS cannot sign new documents anymore.

Since a CMSS signature consists of T MSS signatures, the signature size increases by a factor T compared to MSS. Also, the computation of the roots of the following trees and their signatures increases the signatures generation time.

CMSS verification

The basics of the CMSS signature verification are straight forward and were already explained above.

We now explain how the verifier uses s to determine a positive integer s_i for each layer i, such that Y_i is the s_ith verification key of the active tree on that layer. The verifier uses s_i to construct the path from Y_i to the root of the corresponding tree on layer i (see Section 3). The following formulas show how this can be accomplished.

$$\begin{aligned} j_T &= \lfloor s/2^{H_T} \rfloor, & j_i &= \lfloor j_{i+1}/2^{H_i} \rfloor, i = T-1, \ldots, 1 \\ s_T &= s \bmod 2^{H_T}, & s_i &= j_{i+1} \bmod 2^{H_i}, i = T-1, \ldots, 1 \end{aligned} \tag{52}$$

7 Distributed signature generation

In this section, we describe *distributed signature generation* [4]. This method counteracts the new problems that arise when using the tree chaining method, namely the increased signature size and signature generation time. It is based on the observation that the one-time signatures of the roots and the authentication paths in upper layers change only infrequently. The idea is to distribute the operations required for the generation of these one-time signatures and authentication paths evenly across each step. This significantly improves the worst case signature generation time. Recall Section 2.2, where we showed that the Winternitz one-time signature scheme uses the parameter w to provide a trade-off between the signature generation time and the signature size. Using the method of distributed signature generation it is possible to choose large values of w for upper layers, which in turn results in smaller signatures. The combination of the tree chaining method, the distributed signature generation, and the original MSS is called GMSS.

The idea

Fix a layer $i \geq 2$. Denote the active tree on layer i by TREE_i. It is currently used to sign roots or documents. The preceding tree on that layer is denoted by TREEPREV_i. The next tree on layer i is TREENEXT_i. The idea of the distributed signature generation is the following. When TREE_i is used, the root of TREENEXT_i is known. The root of TREENEXT_i is signed while the signature keys of TREE_i are used. The root of TREENEXT_i was calculated while TREEPREV_i was used to sign documents or roots.

Distributed root signing

We use the notation from above. We explain how the root of TREENEXT_i is signed while TREE_i is used to sign. By construction, the necessary signature key from layer $i-1$ is known.

We distribute the computation of the signature of the root of TREENEXT_i across the leaves of TREE_i. When the first leaf of TREE_i is used we initialize the Winternitz one-time signature generation by calculating the parameters and executing the padding. Then we calculate the number of hash function evaluations and calls to the PRNG required to compute the one-time signature key and the one-time signature. We divide those numbers by 2^{H_i} where H_i is the height of TREE_i to estimate the number of operations required per step. When a leaf of TREE_i is used, the appropriate amount of computation for the signature of the root of TREENEXT_i is performed. The distributed generation of the one-time signatures is visualized in Figure 9.

We estimate the running time of the distributed root signing. The one-time signature of a root of a tree on layer i is generated using the Winternitz parameter w_{i-1} of layer $i-1$. According to Section 2.2 the generation of this signature requires $(2^{w_{i-1}}-1)t_{w_{i-1}}$ hash function evaluations in the worst case. As shown in Section 4 the generation of the one-time signature requires $t_{w_{i-1}}+1$ calls to the PRNG. Since each tree on layer i has 2^{H_i} leaves, the computation of its root signature is distributed across 2^{H_i} steps. Therefore, the total number of extra operations for each leaf of TREE_i to compute the root signature of

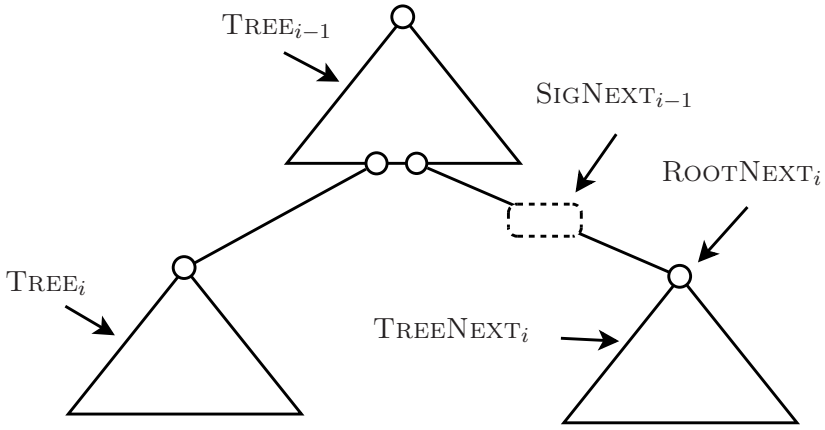

Figure 9: Distributed generation of SIGNEXT$_{i-1}$, the one-time signature of the root of TREENEXT$_i$.

TREENEXT$_i$ is at most

$$c_{\text{sig}}(i) = \left\lceil \frac{(2^{w_{i-1}} - 1)t_{w_{i-1}}}{2^{H_i}} \right\rceil c_{\text{HASH}} + \left\lceil \frac{t_{w_{i-1}} + 1}{2^{H_i}} \right\rceil c_{\text{PRNG}}. \tag{53}$$

Distributed root computation

We explain, how the root of TREENEXT$_i$ is computed while TREEPREV$_i$ is active. This is quite simple. Both TREEPREV$_i$ and TREENEXT$_i$ have the same number of leaves. When a leaf of TREEPREV$_i$ is used, the leaf with the same index in TREENEXT$_i$ is calculated and passed to the treehash algorithm from Section 3.

If $i < T$, i.e. TREENEXT$_i$ is not on the lowest level, the computation of each leaf of TREENEXT$_i$ can also be distributed. This is explained next. Suppose that we want to construct the jth leaf of TREENEXT$_i$ while we are using the jth leaf of TREEPREV$_i$. This computation is distributed across the leaves of the tree TREELOWER on layer $i + 1$ whose root is signed using the jth leaf of TREEPREV$_i$. When the first leaf of TREELOWER is used, we determine the number of hash function evaluations and calls to the PRNG required to compute the jth leaf of TREENEXT$_i$. Recall that the calculation of this leaf requires the computation of a Winternitz one-time key pair. We divide those numbers by $2^{H_{i+1}}$ to obtain the number of operations we will execute in each leaf of TREELOWER. Whenever a leaf of TREELOWER is used, the computation of the jth leaf of TREENEXT is advanced by executing those operations.

Once the jth leaf of TREENEXT$_i$ is generated, it is passed to the treehash algorithm. This contributes to the construction of the root of TREENEXT$_i$. This construction is complete, once we switch from TREEPREV$_i$ to TREE$_i$. So in fact, when TREE$_i$ is used, the root of TREENEXT$_i$ is known. The distributed computation of the roots is visualized in Figure 10. While constructing TREENEXT$_i$, we also perform the initialization steps of the authentication path algorithm of Section 5.5. That is, we store the authentication path of leaf 0 and prepare the algorithm state.

We estimate the extra time required by the distributed root computation. Recall that for the generation of a leaf of TREENEXT$_i$ we first determine the corresponding Winternitz one-time key pair. This key pair is constructed using the Winternitz parameter w_i of layer i. The generation of the one-time signature key requires $t_{w_i} + 1$ calls to the PRNG. The generation of the one-time verification key requires $(2^{w_i} - 1)t_{w_i}$ hash function evaluations

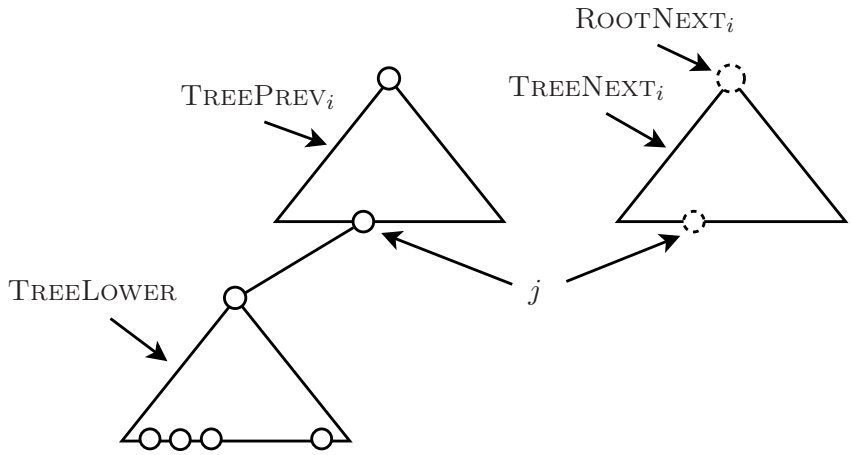

Figure 10: Distributed computation of RootNext$_i$. Leaf j of tree TreeNext$_i$ is precomputed while using tree TreeLower. It is then used to partially compute RootNext$_i$.

and the computation of a leaf of TreeNext$_i$ requires one additional evaluation of the hash function. This has been shown in Sections 2.2 and 4. Since TreeLower has $2^{H_{i+1}}$ leaves, the computation of a leaf of TreeNext$_i$ can be distributed over $2^{H_{i+1}}$ steps. Therefore, the total number of extra operations for each leaf of TreeLower to compute a leaf of TreeNext$_i$ is

$$c^1_{\text{leaf}}(i) = \left\lceil \frac{(2^{w_i}-1)t_{w_i}+1}{2^{H_{i+1}}} \right\rceil c_{\text{Hash}} + \left\lceil \frac{t_{w_i}+1}{2^{H_{i+1}}} \right\rceil c_{\text{PRNG}}. \tag{54}$$

Once a leaf of TreeNext$_i$ is found, it is passed to the treehash algorithm. By the results of Section 3 this costs at most

$$c^2_{\text{leaf}}(i) = H_i \cdot c_{\text{Hash}} \tag{55}$$

additional evaluations of the hash function.

Distributed authentication path computation

Next, we describe the computation of the authentication path of the next leaf of tree Tree$_i$. We use the algorithm described in Section 5.5. This algorithm requires the computation of $(H_i - K_i)/2 + 1$ leaves per round to generate upcoming authentication paths on layer $i = 1, \ldots, T$. As described above, the computation of these leaves is distributed over the $2^{H_{i+1}}$ leaves (or steps) of tree TreeLower, the current tree on the next lower layer $i+1$. Again, this is possible only for leaves in layers $i = 1, \ldots, T-1$. The computation of the leaves in layer T cannot be distributed.

When we use TreeLower for the first time we calculate the number of hash function evaluations and calls to the PRNG required to compute the $(H_i - K_i)/2 + 1$ leaves. Recall that we have to compute a Winternitz one-time key pair to obtain this leaf. Then we divide these costs by $2^{H_{i+1}}$ to estimate the number of operations we have to spend for each leaf of tree TreeLower. At the beginning we don't know which leaves must be computed, we only know how may. Therefore, we have to interact with Algorithm 5.6. We perform the necessary steps to decide which leaf must be computed first. After computing this leaf we pass it to the authentication path algorithm which updates the treehash instance and determines the which leaf must be computed next. This procedure is iterated until

all required leaves are computed. The distributed authentication path computation is visualized in Figure 11.

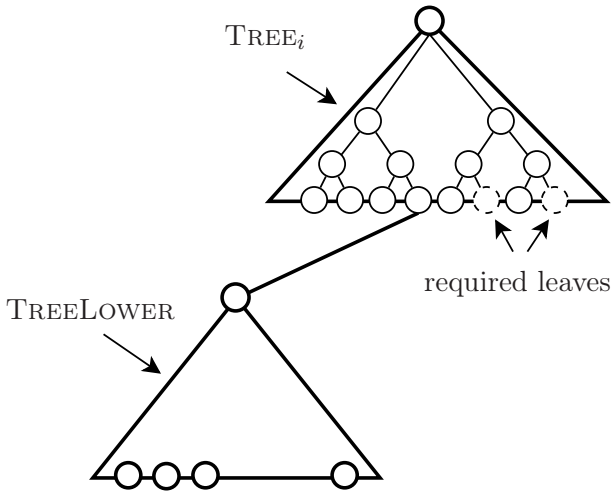

Figure 11: Distributed computation of the next authentication path. The $(H_i - K_i)/2$ required leaves are computed while using tree TREELOWER.

We estimate the cost of the distributed authentication path computation. The algorithm of Section 5.5 requires the computation of $(H_i - K_i)/2 + 1$ leaves for each authentication path. The leaves are computed using the Winternitz parameter w_i of layer i. The generation of one leaf requires $t_{w_i} + 1$ calls to the PRNG and $(2^{w_i} - 1)t_{w_i} + 1$ hash function evaluations, see Sections 2.2 and 4. The computation of the those $(H_i - K_i)/2 + 1$ leaves is distributed over the $2^{H_{i+1}}$ steps in the tree on layer $i + 1$. Therefore, the total number of operations for each leaf of TREELOWER to compute the $(H_i - K_i)/2 + 1$ leaves is

$$c^1_{\text{auth}}(i) = \frac{H_i - K_i + 2}{2} \cdot c^1_{\text{leaf}}(i). \tag{56}$$

The completed leaves are passed to the treehash algorithm that computes their parent nodes. The algorithm of Section 5.5 requires at most $3(H_i - K_i - 1)/2 + 1$ evaluations of the hash function for the computation of parents. Another $H_i - K_i$ calls to the PRNG are required to prepare upcoming seeds. These operations are not distributed but performed at once. Hence, the total number of operations for each leaf of TREE$_i$ is at most

$$c^2_{\text{auth}}(i) = \frac{3(H_i - K_i) - 1}{2} \cdot c_{\text{HASH}} + (H_i - K_i) \cdot c_{\text{PRNG}}. \tag{57}$$

Example 7.1 *This example illustrates how the distributed signature generation improves the signature generation time. Let $H_1 = \ldots = H_T = H$. Further, all layers use the same Winternitz parameter w and the same value for K. Let c_{sig} denote the worst case cost for generating a one-time signature with Winternitz parameter w, let c_{auth} denote the worst case cost for generating an authentication path in a tree of height H using K, and let c_{tree} denote the cost for partially computing the next tree. The worst case cost for the GMSS online and offline part then is*

$$c_{sig} + c_{auth} + c_{tree} + \frac{(T-1)c_{sig} + (T-1)c_{auth} + (T-2)c_{tree}}{2^H}.$$

When the signature generation is not distributed, as in the case of CMSS, the worst case cost for the online and offline part is

$$Tc_{sig} + Tc_{auth} + (T-1)c_{tree}.$$

GMSS key pair generation

We explain GMSS key pair generation, establish the size of the keys, and the cost for computing them. The following parameters are selected. The number T of layers, the heights $H_1, \ldots, H_T$ of the Merkle trees on each layer, the Winternitz parameters $w_1, \ldots, w_T$ for each layer, and the parameters $K_1, \ldots, K_T$ for the authentication path algorithm of Section 5.5.

We use the approach introduced in Section 4 and use an PRNG for the one-time signature generation. Therefore we must choose initial seeds $\textsc{Seed}_i$, for each layer $i = 1, \ldots, T$. The GMSS public key is the root $\textsc{Root}_1$ of the single tree in layer $i = 1$. The GMSS private key consists of the following entries:

$$\begin{array}{rcl|rcl} \textsc{Seed}_i & , \; i = 1, \ldots, T & , & \textsc{SeedNext}_i & , \; i = 2, \ldots, T \\ \textsc{Sig}_i & , \; i = 1, \ldots, T-1 & , & \textsc{RootNext}_i & , \; i = 2, \ldots, T \\ \textsc{Auth}_i & , \; i = 1, \ldots, T & , & \textsc{AuthNext}_i & , \; i = 2, \ldots, T \\ \textsc{State}_i & , \; i = 1, \ldots, T & , & \textsc{StateNext}_i & , \; i = 2, \ldots, T \end{array} \tag{58}$$

The seeds $\textsc{Seed}_i$ are required for the generation of the one-time signature keys used to sign the data and the roots. The seeds $\textsc{SeedNext}_i$ are required for the distributed generation of subsequent roots. These seeds are available after the generation of the roots $\textsc{RootNext}_i$. The one-time signatures $\textsc{Sig}_i$ of the roots are required for the GMSS signatures. The signatures $\textsc{Sig}_i$ do not have to be computed explicitly. They are an intermediate value during the computation of the 0th leaf of tree $\textsc{Tree}_{i-1}$. The roots $\textsc{RootNext}_i$ of the next tree in each layer are required for the distributed generation of the one-time signatures $\textsc{SigNext}_{i-1}$. Also, the authentication path for the first leaf of the first and second tree in each layer is stored. $\textsc{State}_i$ and $\textsc{StateNext}_i$ denote the state of the authentication path algorithm of section 5.5 required to compute authentication paths in trees $\textsc{Tree}_i$ and $\textsc{TreeNext}_i$, respectively. This state contains the seeds and the treehash instance and is initialized during the generation of the root.

The construction of a tree on layer i requires the computation of 2^{H_i} leaves and $2^{H_i} - 1$ evaluations of the hash function to compute inner nodes. Each leaf computation requires $(2^{w_i} - 1) \cdot t_{w_i} + 1$ hash function evaluations and $t_{w_i} + 1$ calls to the PRNG. The total cost for one tree on layer i is given as

$$c_{\text{tree}}(i) = \left(2^{H_i}\left(t_{w_i}(2^{w_i} - 1) + 2\right) - 1\right) c_{\textsc{Hash}} + 2^{H_i}\left(t_{w_i} + 1\right) c_{\textsc{Prng}}. \tag{59}$$

Since we construct two trees on layers $i = 2, \ldots, T$ and one on layer $i = 1$, the total cost for the key pair generation is

$$c_{\text{keygen}} = \sum_{i=1}^{T} c_{\text{tree}}(i) + \sum_{i=2}^{T} c_{\text{tree}}(i). \tag{60}$$

The memory requirements of the keys depend on the output size of the used hash function n. A root is a single hash value and requires n bits. A seed also requires n bits. A one-time signature $\textsc{Sig}_i$ requires $t_{w_{i-1}} \cdot n$ bits. An authentication path together with the algorithm state requires

$$m_{\text{auth}}(i) = \left(3H_i + \left\lfloor \frac{H_i}{2} \right\rfloor - 3K_i - 2 + 2^{K_i}\right) \cdot n \text{ bits}. \tag{61}$$

For each layer $i = 2, \ldots, T$, we store two seeds, two authentication paths and algorithm states, one root and the one-time signature of one root. For layer $i = 1$, we store one seed and one authentication path and algorithm state. The total sizes of the public and the private key are

$$m_{\text{pubkey}} = n \text{ bits}, \tag{62}$$

$$m_{\text{privkey}} = \left(\sum_{i=1}^{T} (m_{\text{auth}}(i) + 1) + \sum_{i=2}^{T} (m_{\text{auth}}(i) + t_{w_{i-1}} + 2) \right) n \text{ bits}. \tag{63}$$

GMSS signature generation

The GMSS signature generation is split in two parts, an online part and an offline part. The online part is equivalent to the CMSS online part. The signer constructs the corresponding signature key from the seed SEED_T and generates the one-time signature SIG_T of the document to be signed. Then he prepares the signature as in Equation (64). The offline part takes care of the distributed computation of upcoming roots, one-time signatures of roots and authentication paths as described above.

$$\begin{aligned} \sigma_s = \big(s, \quad & \text{SIG}_T, Y_T, \text{AUTH}_T, \\ & \text{SIG}_{T-1}, Y_{T-1}, \text{AUTH}_{T-1} \\ & \vdots \\ & \text{SIG}_1, Y_1, \text{AUTH}_1\big). \end{aligned} \tag{64}$$

The online part requires the generation of a single one-time signature. This signature is generated using the Winternitz parameter of the lowest layer T. According to Section 2.2, this requires

$$c_{\text{online}} = (2^{w_T} - 1)t_{w_T} \cdot c_{\text{HASH}} + (t_{w_T} + 1)c_{\text{PRNG}}. \tag{65}$$

operations in the worst case. The size of an GMSS signature is computed with the same formula we used for as the CMSS signatures. It consists of T authentication paths ($H_i \cdot n$ bits) and T one-time signatures ($t_{w_i} \cdot n$ bits), one for each layer $i = 1, \ldots, T$. Adding up yields

$$m_{\text{signature}} = \sum_{i=1}^{T} (H_i + t_{w_i}) \cdot n \text{ bits}. \tag{66}$$

To estimate the computational effort required for the offline part we assume the worst case where we have to advance one leaf on all layers $i = 1, \ldots, T$. The computation of the one-time signature SIGNEXT_i can be distributed for each layers $i = 1, \ldots, T-1$. The computation of the leaves required to construct the root ROOTNEXT_i can be distributed for all layers $i = 2, \ldots, T-1$. For layer $i = T$, the respective leaf of tree TREENEXT_T must be computed at once. Together with the hash function evaluations for the treehash algorithm, this requires at most

$$c^3_{\text{leaf}} = ((2^{w_T} - 1)t_{w_T} + H_T + 1)c_{\text{HASH}} + (t_{w_T} + 1)c_{\text{PRNG}} \tag{67}$$

operations. The leaves required for the computation of upcoming authentication paths can be distributed for all layers $i = 1, \ldots, T-1$. For layer $i = T$, the $(H_T - K_T)/2 + 1$ leaves must be computed at once. Together with the hash function evaluations for the treehash algorithm, this requires at most

$$\begin{aligned} c^3_{\text{auth}} = & \frac{H_T - K_T + 2}{2} \cdot c^3_{\text{leaf}} + \frac{3(H_T - K_T) - 1}{2} \cdot c_{\text{HASH}} \\ & + (H_T - K_T) \cdot c_{\text{PRNG}} \end{aligned} \tag{68}$$

operations. In summary, the number of operations required by the offline part in the worst case are

$$\begin{aligned} c_{\text{offline}} = &\sum_{i=2}^{T} c_{\text{sig}}(i) + \sum_{i=2}^{T-1} \left(c^1_{\text{leaf}}(i) + c^2_{\text{leaf}}(i)\right) + c^3_{\text{leaf}} \\ &+ \sum_{i=1}^{T-1} \left(c^1_{\text{auth}}(i) + c^2_{\text{auth}}(i)\right) + c^3_{\text{auth}}. \end{aligned} \tag{69}$$

The last step is to estimate the space required by the offline part. We have to store the partially constructed one-time signature SIGNEXT_i for layers $i = 1, \ldots, T-1$ which requires at most $t_{w_{i-1}} \cdot n$ bits. We also have to store the treehash stack for the generation of the root ROOTNEXT_i for layers $i = 2, \ldots, T$ which requires $H_i \cdot n$ bits. We further require memory to store partially constructed leaves. One leaf requires at most $t_{w_i} \cdot n$ bits. For the generation of ROOTNEXT_i we have to store at most one leaf for each layer $i = 2, \ldots, T-1$. For the authentication path, we have to store at most one leaf for each layer $i = 1, \ldots, T-1$. Note that since we compute the leaves required for the authentication path successively, we have to store only one partially constructed leaf at a time. Finally, we need to store the partial state STATENEXT_i of the authentication path algorithm for layers $i = 2, \ldots, T$ which requires at most $m_{\text{auth}}(i)$ bits (see Equation (61)). In summary, the memory required by the offline part in the worst case is

$$m_{\text{offline}} = \left(\sum_{i=2}^{T} \left(t_{w_{i-1}} + H_i + m_{\text{auth}}(i)\right) + \sum_{i=2}^{T-1} t_{w_i} + \sum_{i=1}^{T-1} t_{w_i} \right) \text{bits}. \tag{70}$$

GMSS signature verification

Since the main idea of GMSS is to distribute the signature generation, the signature verification doesn't change compared to CMSS. The verifier successively verifies a one-time signature and uses the corresponding authentication path and Equation (52) to compute the root. This is done until the root of the tree in the top layer is computed. If this root matches the signers public key, the signature is valid.

The verifier must verify T one-time signatures which in the worst case requires $(2^{w_i} - 1)t_{w_i}$ evaluations of the hash function, for $i = 1, \ldots, T$. Another H_i evaluations of the hash function are required to reconstruct the path to the root using the authentication path. In total, the number of hash function evaluations required in the worst case is

$$c_{\text{verify}} = \sum_{i=1}^{T} \left((2^{w_i} - 1)t_{w_i} + H_i\right) c_{\text{HASH}}. \tag{71}$$

8 Security of the Merkle Signature Scheme

This section deals with the security of the Merkle signature scheme. We will show that the Lamport–Diffie one-time signature scheme is existentially unforgeable under an adaptive chosen message attack (CMA-secure) as long as the used one-way function is preimage resistant. Then we show that the Merkle signature scheme is CMA-secure as long as the used hash function is collision resistant and the underlying one-time signature scheme is CMA-secure. Finally, we estimate the security level of the Merkle signature scheme for a given output length n of the hash function.

8.1 Notations and definitions

We start with some security notions and definitions.

Security notions for hash functions

We present three security notions for hash functions: preimage resistance, second preimage resistance, and collision resistance. The definitions are taken from [30]. We write $x \xleftarrow{\$} S$ for the experiment of choosing a random element from the finite set S with the uniform distribution. Let $\mathcal{G}$ be a family of hash functions, that is, a parameterized set

$$\mathcal{G} = \{g_k : \{0,1\}^* \to \{0,1\}^n | k \in K\} \tag{72}$$

where $n \in \mathbb{N}$ and K is a finite set. The elements of K are called *keys*. An *adversary* Adv is a probabilistic algorithm that takes any number of inputs.

We define *preimage resistance*. In fact, our notion of preimage resistance is a special case of the preimage resistance defined in [30] which is useful in our context. Consider an adversary that attempts to find preimages of the hash functions in $\mathcal{G}$. The adversary takes as input a key $k \in K$ and the image $y = g_k(x)$ of a string $x \in \{0,1\}^n$. Both k and x are chosen randomly with the uniform distribution. The adversary outputs a *preimage* x' of y or `failure`. The success probability of this adversary is denoted by

$$\Pr[k \xleftarrow{\$} K, x \xleftarrow{\$} \{0,1\}^n, y \longleftarrow g_k(x), x' \xleftarrow{\$} \text{Adv}(k,y) : g_k(x') = y]. \tag{73}$$

Let t, ϵ be positive real numbers. The family $\mathcal{G}$ is called (t,ϵ) preimage resistant, if the success probability (73) of any adversary Adv that runs in time t is at most ϵ.

Next, we define *second preimage resistance*. Consider an adversary that attempts to find second preimages of the hash functions in $\mathcal{G}$. The adversary takes as input a key $k \in K$ and a string $x \in \{0,1\}^n$, both chosen randomly with the uniform distribution. He outputs a *second preimage* x' under g_k of $g_k(x)$ which is different from x or `failure`. The success probability of this adversary is denoted by

$$\Pr[k \xleftarrow{\$} K, x \xleftarrow{\$} \{0,1\}^n, x' \xleftarrow{\$} \text{Adv}(k,x) : x \neq x' \wedge g_k(x) = g_k(x')]. \tag{74}$$

Let t, ϵ be positive real numbers. The family $\mathcal{G}$ is called (t,ϵ) second-preimage resistant, if the success probability (74) of any adversary Adv that runs in time t is at most ϵ.

Finally, we define *collision resistance*. Consider an adversary that attemps to find collisions of the hash functions in $\mathcal{G}$. The adversary takes as input a key $k \in K$, chosen randomly with the uniform distribution. He outputs a *collision* of g_k, that is, a pair $x, x' \in \{0,1\}^*$ with $x \neq x'$ and $g(x) = g(x')$ or `failure`. The success probability of this adversary is denoted by

$$\Pr[k \xleftarrow{\$} K, (x,x') \xleftarrow{\$} \text{Adv}(k) : x \neq x' \wedge g_k(x) = g_k(x')]. \tag{75}$$

Let t, ϵ be positive real numbers. The family $\mathcal{G}$ is called (t,ϵ) collision resistant, if the success probability (75) of any adversary Adv that runs in time t is at most ϵ.

Signature schemes

Let Sign be a signature scheme. So Sign is a triple (Gen, Sig, Ver). Gen is the key pair generation algorithm. It takes as input 1^n, the string of n successive 1s where $n \in \mathbb{N}$ is a security parameter. It outputs a pair $(\mathsf{sk}, \mathsf{pk})$ consisting of a private key sk and a public

key pk. SIG is the signature generation algorithm. It takes as input a message M and a private key sk. It outputs a signature σ for the message M. Finally, VER is the verification algorithm. Its input is a message M, a signature σ and a public key pk. It checks whether σ is a valid signature for M using the public key pk. It outputs `true` if the signature is valid and `false` otherwise.

Existential unforgeability

Let $\text{SIGN} = (\text{GEN}, \text{SIG}, \text{VER})$ be a signature scheme and let $(\mathsf{sk}, \mathsf{pk})$ be a key pair generated by GEN. We define *existential unforgeability under an adaptive chosen message attack* of SIGN. This security model assumes a very powerful forger. The forger has access to the public key and a signing oracle $\mathcal{O}(\mathsf{sk}, \cdot)$ that, in turn, has access to the private key. On input of a message the oracle returns the signature of that message. It is the goal of the forger to win the following game. The forger chooses at most q messages and lets the signing oracle find the signatures of those messages. The maximum number q of queries is also an input of the forger. The oracle queries may be adaptive, that is, a message may depend on the oracles answers to previously queried messages. The forger outputs a pair (M', σ'). The forger wins if M is different from all the messages in the oracle queries and if $\text{VER}(M', \sigma', \mathsf{pk}) = \texttt{true}$. We denote such a forger by $\text{FOR}^{\mathcal{O}(\mathsf{sk},\cdot)}(\mathsf{pk})$.

Let t and ϵ be positive real numbers and let q be a positive integer. The signature scheme SIGN is (t, ϵ, q) *existentially unforgeable under an adaptive chosen message attack* if for any forger that runs in time t, the success probability for winning the above game (which depends on q) is at most ϵ. If SIGN has the above property it is also called a (t, ϵ, q) *signature scheme*.

For one-time signatures we must have $q = 1$ since the signature key of a one-time signature scheme must be used only once. For the Merkle signature scheme we must have $q \leq 2^H$.

8.2 Security of the Lamport–Diffie one-time signature scheme

In this section we discuss the security of LD–OTS from Section 2.1. We slightly modify this scheme. Select a security parameter $n \in \mathbb{N}$. Let $K = K(n)$ be a finite set of parameters. Let

$$\mathcal{F} = \left\{ f_k : \{0,1\}^n \to \{0,1\}^n | k \in K \right\}$$

be a family of one-way functions. The key generation of the modified LD–OTS works as follows. On input of 1^n for a security parameter n a key $k \in K(n)$ is selected randomly with the uniform distribution. Then LD–OTS is used with the one-way function f_k. The secret and public keys are generated as described in Section 2.1. The key k is included in the public key. We show that the existential unforgeability under adaptive chosen message attacks of this LD-OTS variant can be reduced to the preimage resistance of the family $\mathcal{F}$.

Suppose that there exists a forger $\text{FOR}^{\mathcal{O}(X,\cdot)}(Y)$ of LD-OTS. Then an adversary $\text{ADV}_{\mathsf{Pre}}$ that determines preimages of functions in $\mathcal{F}$ can be constructed as follows. Fix a security parameter n. Input for $\text{ADV}_{\mathsf{Pre}}$ are a key k and the image $y = f_k(x)$ of a string $x \in \{0,1\}^n$. Both k and x are selected randomly with the uniform distribution. A LD–OTS key pair (X, Y) is generated using the one-way function f_k. The public key Y is of the form $Y = (y_{n-1}[0], y_{n-1}[1], \ldots, y_0[0], y_0[1])$. The adversary selects indices $a \in \{0, \ldots, n-1\}$ and $b \in \{0, 1\}$ randomly with the uniform distribution. He replaces the string $y_a[b]$ with the target string y. Next, $\text{ADV}_{\mathsf{Pre}}$ runs the forger $\text{FOR}^{\mathcal{O}(X,\cdot)}(Y)$ with the modified public key. If the forger asks its oracle to sign a message $M = (m_{n-1}, \ldots, m_0)$ and if $m_a = 1 - b$, then the adversary, playing the role of the oracle, signs the message and returns the signature.

The adversary can sign this message since he knows the original key pair and because of $m_a = 1 - b$, the modified string in the public key is not used. However, if $m_a = b$ then the adversary cannot sign M. So his answer to the oracle query is `failure` which also causes the forger to abort. If the forger's oracle query was successful or if the forger does not ask the oracle at all the forger may produce a message $M' = (m'_{n-1}, \ldots, m'_0)$ and the signature $(\sigma'_{n-1}, \ldots, \sigma'_0)$ of that message. If $m'_a = b$, then σ'_a is the preimage of y which the adversary returns. Otherwise, the adversary returns `failure`. More formally, the adversary is presented in Algorithm 8.1.

Algorithm 8.1 $\textsc{Adv}_{\mathsf{Pre}}$

Input: $k \xleftarrow{\$} K$ and $y = f_k(x)$, where $x \xleftarrow{\$} \{0,1\}^n$
Output: x' such that $y = f_k(x)$ or `failure`

1. Generate an LD–OTS key pair (X, Y).
2. Choose $a \xleftarrow{\$} \{0, \ldots, n-1\}$ and $b \xleftarrow{\$} \{0,1\}$.
3. Replace $y_a[b]$ by y in the LD–OTS verification key Y.
4. Run $\textsc{For}^{\mathcal{O}(X,\cdot)}(Y)$.
5. When $\textsc{For}^{\mathcal{O}(X,\cdot)}(Y)$ asks its only oracle query with $M = (m_{n-1}, \ldots, m_0)$:
   (a) **if** $m_a = (1 - b)$ **then** sign M and respond to the forger $\textsc{For}^{\mathcal{O}(X,\cdot)}(Y)$ with the signature σ.
   (b) **else return** `failure`.
6. When $\textsc{For}^{\mathcal{O}(X,\cdot)}(Y)$ outputs a valid signature $\sigma' = (\sigma'_{n-1}, \ldots, \sigma'_0)$ for message $M' = (m'_0, \ldots, m'_{n-1})$:
   (a) **if** $m'_a = b$ **then return** σ'_a as preimage of y.
   (b) **else return** `failure`.

We now compute the success probability of the adversary $\textsc{Adv}_{\mathsf{Pre}}$. We denote by ϵ the forger's success probability for producing an existential forgery of the LD–OTS and by t its running time. By $t_{\textsc{Gen}}$ and $t_{\textsc{Sig}}$ we denote the times the LD–OTS requires for key and signature generation, respectively.

The adversary $\textsc{Adv}_{\mathsf{Pre}}$ is successful in finding a preimage of y if and only if $\textsc{For}^{\mathcal{O}(X,\cdot)}(Y)$ queries the oracle with a message $M = (m_{n-1}, \ldots, m_0)$ with $m_a = (1 - b)$ (Line 5a) or if he queries the oracle not at all and if the forger returns a valid signature for message $M' = (m'_0, \ldots, m'_{n-1})$ with $m'_a = b$ (Line 6a). Since b is selected randomly with the uniform distribution, the probability for $m_a = (1 - b)$ is $1/2$. Since M' must be different from the queried message M, there exists at least one index c such that $m'_c = 1 - m_c$. $\textsc{Adv}_{\mathsf{Pre}}$ is successful if $c = a$, which happens with probability at least $1/2n$. Hence, the adversary's success probability for finding a preimage in time $t_{\text{OW}} = t + t_{\textsc{Sig}} + t_{\textsc{Gen}}$, is at least $\epsilon/4n$. We have proved the following theorem.

Theorem 8.1 *Let $n \in \mathbb{N}$, let K be a finite parameter set, let t_{OW}, ϵ_{OW} be positive real numbers, and $\mathcal{F} = \{f_k : \{0,1\}^n \to \{0,1\}^n | k \in K\}$ be a family of $(t_{\text{OW}}, \epsilon_{\text{OW}})$ one-way functions. Then the LD–OTS variant that uses $\mathcal{F}$ is $(t_{\text{OTS}}, \epsilon_{\text{OTS}}, 1)$ existentially unforgeable*

under an adaptive chosen message attack with $\epsilon_{\text{OTS}} \leq 4n \cdot \epsilon_{\text{OW}}$ *and* $t_{\text{OTS}} = t_{\text{OW}} - t_{\text{SIG}} - t_{\text{GEN}}$ *where* t_{GEN} *and* t_{SIG} *are the key generation and signing times of LD–OTS, respectively.*

8.3 Security of the Merkle signature scheme

This section discusses the security of the Merkle signature scheme. We modify the Merkle scheme slightly. Select a security parameter $n \in N$. Let $K = K(n)$ be a finite set of parameters. Let

$$\mathcal{G} = \left\{ g_k : \{0,1\}^* \to \{0,1\}^n | k \in K \right\}$$

be a family of hash functions. The key generation of the modified MSS works as follows. On input of 1^n for a security parameter n a key $k \in K(n)$ is selected randomly with the uniform distribution. Then the Merkle signature scheme is used with the hash function g_k and some one-time signature scheme. The secret and public keys are generated as described in Section 3. The parameter k is included in the public key. We show that the existential unforgeability of this MSS variant under an adaptive chosen message attack can be reduced to the collision resistance of the family $\mathcal{G}$ and the existential unforgeability of the underlying one-time signature scheme.

We explain how an existential forger for the Merkle signature scheme can be used to construct an adversary that is either an existential forger for the underlying one-time signature scheme or a collision finder for a hash function in $\mathcal{G}$. The input of the adversary is a one-time signature scheme, a key $k \in K$ chosen randomly with the uniform distribution, and the Merkle tree height H. Input is also a verification key Y_{OTS} and a signing oracle $\mathcal{O}_{\text{OTS}}(X_{\mathsf{OTS}}, \cdot)$, where $(X_{\mathsf{OTS}}, Y_{\mathsf{OTS}})$ is a key pair of the one-time signature scheme.

The adversary is allowed to query the oracle $\mathcal{O}_{\text{OTS}}(X_{\mathsf{OTS}}, \cdot)$ once. He aims to output a collision for the hash function g_k or an existential forgery (M', σ') for the one-time signature scheme that can be verified using the verification key Y_{OTS}. He has access to an adaptive chosen message forger $\text{FOR}^{\mathcal{O}(\mathsf{sk},\cdot)}(\mathsf{pk})$ for the MSS with hash function g_k and tree height H. The forger is allowed to ask 2^H queries to its signature oracle. The adversary is supposed to impersonate that oracle.

The adversary selects randomly with the uniform distribution an index c in the set $\{0, \ldots, 2^H - 1\}$. He generates a Merkle key pair in the usual manner with the only exception that as the cth one-time verification key the one-time verification key Y_{OTS} from the input is used. Then the adversary invokes the adaptive chosen message forger for the Merkle scheme with the hash function g_k and the public Merkle key which he generated before. Without loss of generality, we assume that the forger queries the oracle 2^H times. The oracle answers are given by the adversary. When the forger asks for the ith signature, $i \neq c$, then the adversary produces this signatures using the signature keys which he generated before. However, when the forger asks for the cth signature, the adversary queries the oracle $\mathcal{O}_{\text{OTS}}(X_{\mathsf{OTS}}, \cdot)$. Suppose that the forger is successful and outputs an existential forgery $(M', (s, \sigma', Y', A'))$ where s is the index of the one-time key pair used for this signature, σ' is the one-time signature, Y' is the verification key and A' is the authentication path. The adversary examines the Merkle signature (s, σ, Y, A) of M he returned in response to the forgers sth oracle query.

If $s = c$ and $(Y, A) = (Y', A')$, then the adversary returns (M', σ'). We show that this is an existential forgery of the one-time signature scheme with verification key Y_{OTS}. Since $s = c$ we have $Y = Y' = Y_{\mathsf{OTS}}$. So the verification key in the message returned by the forger is the same as the verification key returned by the oracle when it is queried for the cth time. The same is true for the authentication path. This implies that the message M in the cth oracle query is different from M'. So (M', σ') is an existential forgery.

If $(Y, A) \neq (Y', A')$, then the adversary can construct a collision for the hash function g_k as follows. Consider the path $B = (B_0 = g_k(Y), B_1, \ldots, B_H)$ from Y in the Merkle tree to its root constructed using the hash function g_k and the authentication path $A = (A_0, \ldots, A_{H-1})$. Compare it to the path $B' = (B'_0 = g_k(Y'), B'_1, \ldots, B'_H)$ from Y' in the Merkle tree to its root constructed using the authentication path $A' = (A'_0, \ldots, A'_{H-1})$. First assume that B and B' are different. For example, this is true when $Y \neq Y'$. Since $B_H = B'_H$ is the MSS public key, there is an index $0 \leq i < H$ with $B_{i+1} = B'_{i+1}$ and $B_i \neq B'_i$. Since B_{i+1} is the hash value of the concatenation of B_i and A_i (in the appropriate order), and since B'_{i+1} is the hash value of the concatenation of B'_i and A'_i (in the appropriate order), a collision of g_k is found. Next, assume that B and B' are equal. Therefore $g_k(Y) = B_0 = B'_0 = g_k(Y')$ holds. If $Y \neq Y'$ a collision is found. If $Y = Y'$ then A and A' are different. Assume that $A_i \neq A'_i$ for some index $i < H$. Since B_{i+1} is the hash value of the concatenation of B_i and A_i (in the appropriate order), and since B'_{i+1} is the hash value of the concatenation of B'_i and A'_i (in the appropriate order) again a collision is found. That collision is returned by the adversary. In all other cases the adversary returns failure. Algorithm 8.2 summarizes our description.

Algorithm 8.2 $\textsc{Adv}_{\mathsf{CR},\mathsf{OTS}}$

Input: Key for the hash function $k \xleftarrow{\$} K$, height of the tree $H \geq 2$, one instance of the underlying OTS consisting of a verification key Y_{OTS} and the corresponding signing oracle $\mathcal{O}_{\mathrm{OTS}}(X_{\mathsf{OTS}}, \cdot)$.
Output: A collision of g_k, an existential forgery for the supplied instance of the OTS, or `failure`

1. Set $c \xleftarrow{\$} \{0, \ldots, 2^H - 1\}$.
2. Generate OTS key pairs $(X_j, Y_j), j = 0, \ldots, 2^H - 1, j \neq c$ and set $Y_c \leftarrow Y_{\mathsf{OTS}}$.
3. Complete the Merkle key pair generation and obtain $(\mathsf{sk}, \mathsf{pk})$.
4. Run $\textsc{For}^{\mathcal{O}(\mathsf{sk},\cdot)}(\mathsf{pk})$.
5. When $\textsc{For}^{\mathcal{O}(\mathsf{sk},\cdot)}(\mathsf{pk})$ asks its qth oracle query $(0 \leq q \leq 2^H - 1)$:
   (a) **if** $q = c$ **then** query the signing oracle $\mathcal{O}_{\mathrm{OTS}}(X_{\mathsf{OTS}}, \cdot)$.
   (b) **else** compute the one-time signature σ using the qth signature key X_q.
   (c) Return the corresponding Merkle signature to the forger.
6. If the forger outputs an existential forgery $(M', (s, \sigma', Y', A'))$, examine the Merkle signature (s, σ, Y, A) returned in response to the forgers sth oracle query.
   (a) **if** $(Y', A') \neq (Y, A)$ **then return** a collision of g_k.
   (b) **else**
      i. **if** $s = c$ **then return** (M', σ') as forgery for the supplied instance of the one-time signature scheme.
      ii. **else return** `failure`.

We now estimate the success probability of the adversary $\textsc{Adv}_{\mathsf{CR},\mathsf{OTS}}$. In the following, ϵ denotes the success probability and t the running time of the forger. Also, $t_{\textsc{Gen}}, t_{\textsc{Sig}}$, and $t_{\textsc{Ver}}$ denote the times MSS requires for key generation, signature generation, and

verification, respectively.

If $(Y', A') \neq (Y, A)$, then the adversary returns collision. His (conditional) probability ϵ_{CR} for returning a collision in time $t_{\text{CR}} = t + 2^H \cdot t_{\text{SIG}} + t_{\text{VER}} + t_{\text{GEN}}$ is at least ϵ. If $(Y', A') = (Y, A)$ the adversary returns an existential forgery if $s = c$. His (conditional) probability ϵ_{OTS} for finding an existential forgery in time $t_{\text{OTS}} = t + 2^H \cdot t_{\text{SIG}} + t_{\text{VER}} + t_{\text{GEN}}$ is at least $\epsilon \cdot 1/2^H$. Since both cases are mutually exclusive, one of them occurs with probability at least $1/2$. So we have proved the following theorem.

Theorem 8.2 *Let K be a finite set, let $H \in \mathbb{N}$, $t_{\text{CR}}, t_{\text{OTS}}, \epsilon_{\text{CR}}, \epsilon_{\text{OTS}} \in \mathbb{R}_{>0}$, $\epsilon_{\text{CR}} \leq 1/2$, $\epsilon_{\text{OTS}} \leq 1/2^{H+1}$, and let $\mathcal{G} = \{g_k : \{0,1\}^* \to \{0,1\}^n | k \in K\}$ be a family of $(t_{\text{CR}}, \epsilon_{\text{CR}})$ collision resistant hash functions. Consider MSS using a $(t_{\text{OTS}}, \epsilon_{\text{OTS}}, 1)$ signature scheme. Then MSS is a $(t, \epsilon, 2^H)$ signature scheme with*

$$\epsilon \leq 2 \cdot \max\{\epsilon_{\text{CR}}, 2^H \cdot \epsilon_{\text{OTS}}\} \tag{76}$$

$$t = \min\{t_{\text{CR}}, t_{\text{OTS}}\} - 2^H \cdot t_{\text{SIG}} - t_{\text{VER}} - t_{\text{GEN}}. \tag{77}$$

This theorem tell us that if there is no adversary that breaks the collision resistance of the family $\mathcal{G}$ in time at most t_{CR} with probability greater than ϵ_{CR} and there is no adversary that is able to produce an existential forgery for the one-time signature scheme used in MSS in time at most t_{OTS} with probability greater than ϵ_{OTS}, then there exists no forger for MSS running in time at most $\min\{t_{\text{CR}}, t_{\text{OTS}}\} - 2^H \cdot t_{\text{SIG}} - t_{\text{VER}} - t_{\text{GEN}}$ and success probability greater then $2 \cdot \max\{\epsilon_{\text{CR}}, 2^H \cdot \epsilon_{\text{OTS}}\}$.

8.4 The security level of MSS

The goal of this section is to estimate the security level of the Merkle signature scheme when used with the Lamport–Diffie one-time signature scheme for a given output length n of the hash function. Let $b \in \mathbb{N}$. We say that MSS has security level 2^b if the expected number of hash function evaluations required for the generation of an existential forgery is at least 2^b. This security level can be computed as t/ϵ where t is the running time of an existential forger and ϵ is its success probability. We also say that the signature scheme has b *bits of security* or that the *bit security* is b. In this section let $\epsilon_{\text{CR}}, t_{\text{CR}}, \epsilon_{\text{OW}}, t_{\text{OW}} \in \mathbb{R}_{>0}$, let K be a finite set, and let

$$\mathcal{G} = \{g_k : \{0,1\}^* \to \{0,1\}^n | k \in K\} \tag{78}$$

be a family of $(t_{\text{CR}}, \epsilon_{\text{CR}})$ collision resistant and $(t_{\text{OW}}, \epsilon_{\text{OW}})$ preimage resistant hash functions.

Since we consider MSS using LD-OTS, we first combine Theorems 8.1 and 8.2. This is achieved by substituting the values for ϵ_{OTS} and t_{OTS} from Theorem 8.1 in Equations (76) and (77) from Theorem 8.2. This yields

$$\epsilon \leq 2 \cdot \max\{\epsilon_{\text{CR}}, 2^H \cdot 4n \cdot \epsilon_{\text{OW}}\} \tag{79}$$

$$t = \min\{t_{\text{CR}}, t_{\text{OW}}\} - 2^H \cdot t_{\text{SIG}} - t_{\text{VER}} - t_{\text{GEN}}. \tag{80}$$

Note that we can replace t_{OTS} by t_{OW} rather than $t_{\text{OW}} - t_{\text{SIG}} - t_{\text{GEN}}$, since the time LD-OTS requires for signature and key generation is already included in the signature and key generation time of the MSS in Theorem 8.2. We also require $\epsilon_{\text{CR}} \leq 1/2$ and $\epsilon_{\text{OW}} \leq 1/(2^{H+1} \cdot 4n)$ to ensure $\epsilon \leq 1$.

To estimate the security level, we need explicit values for the key pair generation, signature generation and verification times of MSS using LD-OTS. We will use the following upper bounds.

$$t_{\text{GEN}} \leq 2^H \cdot 6n, \quad t_{\text{SIG}} \leq 4n(H+1), \quad t_{\text{VER}} \leq n + H$$

We also make assumptions for the values of $(t_{\text{CR}}, \epsilon_{\text{CR}})$ and $(t_{\text{OW}}, \epsilon_{\text{OW}})$. We distinguish between attacks that use classic computers only and attacks with quantum computers.

Using classical computers

In our security analysis of MSS we assume that the hash functions under consideration have output length n and only admit generic attacks against their preimage and collision resistance. Those generic attacks are exhaustive search and the birthday attack. When classical computers are used, then a birthday attack that inspects $2^{n/2}$ hash values has a success probability of approximately $1/2$. Also, an exhaustive search of $2^{n/2}$ random strings yields a preimage of a given hash value with probability $1/2^{n/2}$. Therefore, we assume that the hash function family $\mathcal{G}$ is $(2^{n/2}, 1/2)$ collision resistant and $(2^{n/2}, 1/2^{n/2})$ preimage resistant. In this situation, we prove the following theorem.

Theorem 8.3 (Classic case) *The security level of the Merkle signature scheme combined with the Lamport-Diffie one-time signature scheme is at least*

$$b = n/2 - 1 \tag{81}$$

if the height of the Merkle tree is at most $H \leq n/3$ and the output length of the hash function is at least $n \geq 87$.

To prove Theorem 8.3 we use our assumption and Equations (79) and (80) and obtain the following estimate for the security level.

$$\frac{t}{\epsilon} \geq \frac{2^{n/2} - 2^H \cdot t_{\text{SIG}} - t_{\text{VER}} - t_{\text{GEN}}}{2 \cdot \max\{1/2, 2^H \cdot 4n \cdot 1/2^{n/2}\}}. \tag{82}$$

Using $H \leq n/3$, the maximum in the denominator is $1/2$ as long as

$$n/3 \leq n/2 - \log_2 4n - 1 \tag{83}$$

which holds for $n \geq 53$. Using the upper bounds for t_{SIG}, t_{VER}, and t_{GEN} estimated above, Equation (82) implies

$$\frac{t}{\epsilon} \geq 2^{n/2} - 2^H \cdot 4n(H+1) - (n+H) - 2^H \cdot 6n. \tag{84}$$

Using $H \leq n/3$, the desired lower bound for the security level of $2^{n/2-1}$ holds as long as

$$2^{n/3}(4/3 \cdot n^2 + 4n) + 4/3 \cdot n + 2^{n/3} \cdot 6n \leq 2^{n/2-1} \tag{85}$$

which is true for $n \geq 87$.

Using quantum computers

Again, we assume that our hash functions only admit generic attacks against their collision and preimage resistance. However, when quantum computers are available, the Grover algorithm [13] can be used in those generic attacks. Grovers algorithm requires $2^{n/3}$ evaluations of the hash function to find a collision with probability at most $1/2$. So we assume that our hash functions are $(2^{n/3}, 1/2)$ collision resistant. Also, we may by virtue of Grover's algorithm assume that our hash functions are $(2^{n/3}, 1/2^{n/3})$ preimage resistant. In this situation, we prove the following theorem.

Theorem 8.4 (Quantum case) *The security level of the Merkle signature scheme combined with the Lamport-Diffie one-time signature scheme is at least*

$$b = n/3 - 1 \tag{86}$$

if the height of the Merkle tree is at most $H \leq n/4$ and the output length of the hash function is at least $n \geq 196$.

To prove Theorem 8.4 we use the same approach as for the proof of Theorem 8.3. We use our assumption on the hash function and Equations (79) and (80) and obtain the following estimate for the security level.

$$\frac{t}{\epsilon} \geq \frac{2^{n/3} - 2^H \cdot t_{\text{SIG}} - t_{\text{VER}} - t_{\text{GEN}}}{2 \cdot \max\{1/2, 2^H \cdot 4n \cdot 1/2^{n/3}\}}. \tag{87}$$

Using $H \leq n/4$, the maximum in the denominator is $1/2$ as long as

$$n/4 \leq n/3 - \log_2 4n - 1 \tag{88}$$

which holds for $n \geq 119$. Using the upper bounds for t_{SIG}, t_{VER}, and t_{GEN} estimated above, Equation (87) implies

$$\frac{t}{\epsilon} \geq 2^{n/3} - 2^H \cdot 4n(H+1) - (n+H) - 2^H \cdot 6n. \tag{89}$$

Using $H \leq n/4$, the desired lower bound for the security level of $2^{n/3-1}$ holds as long as

$$2^{n/4}(n^2 + 4n) + 5/4 \cdot n + 2^{n/4} \cdot 6n \leq 2^{n/3-1} \tag{90}$$

which is true for $n \geq 196$.

Comparison of the bit security

Table 2 shows the security level for some output lenghts n of the hash function. This table also shows the maximum value for H such that the security level holds.

Table 2: Security level of the Merkle signature scheme combined with the Lamport–Diffie one-time signature scheme in bits.

Output length n	128	160	224	256	384	512
Classic case						
bit security b	63	79	111	127	191	255
Maximum value for H	42	53	74	85	128	170
Quantum case						
bit security b	–	–	73	84	127	169
Maximum value for H	–	–	56	64	96	128

This table shows, that state-of-the-art hash functions can be used to ensure a high security level of the Merkle signature scheme, even against attacks by quantum computers. For all practical applications the maximum height of the Merkle tree and the resulting number of messages that can be signed with one key pair is sufficiently large.

References

[1] Bellare, M., Rogaway, P.: Optimal asymmetric encryption. In *Advances in Cryptology - EUROCRYPT'94*, LNCS 950, pages 92–111. Springer, 1995.

[2] Berman, P., Karpinski, M., Nekrich, Y.: Optimal Trade-Off for Merkle Tree Traversal. Theoretical Computer Science, volume 372, issue 1, pages 26–36, 2007.

[3] Buchmann, J., Coronado, C., Dahmen, E., Döring, M., Klintsevich, E.: CMSS – an improved Merkle signature scheme. In *Progress in Cryptology - INDOCRYPT 2006*, LNCS 4329, pages 349–363. Springer-Verlag, 2006.

[4] Buchmann, J., Dahmen, E., Klintsevich, E., Okeya, K., Vuillaume, C.: Merkle signatures with virtually unlimited signature capacity. In *Applied Cryptography and Network Security - ACNS 2007*, LNCS 4521, pages 31–45. Springer, 2007.

[5] Buchmann, J., Dahmen, E., Schneider, M.: Merkle tree traversal revisited. 2nd International Workshop on Post-Quantum Cryptography - PQCrypto 2008, LNCS 5299, pages 63–77. Springer, 2008.

[6] Boneh, D., Mironov, I., Shoup, V.: A secure signature scheme from bilinear maps. In *Topics in Cryptology - CT-RSA 2003*, LNCS 2612, pages 98–110. Springer, 2003.

[7] Coppersmith, D., Jakobsson, M.: Almost Optimal Hash Sequence Traversal. Financial Crypto '02. Available at `www.markus-jakobsson.com`.

[8] Coronado, C.: On the security and the efficiency of the Merkle signature scheme. Cryptology ePrint Archive, Report 2005/192, 2005. `http://eprint.iacr.org/`.

[9] Dahmen, E., Okeya, K., Takagi, T., Vuillaume, C.: Digital Signatures out of Second-Preimage Resistant Hash Functions. 2nd International Workshop on Post-Quantum Cryptography - PQCrypto 2008, LNCS 5299, pages 109–123. Springer, 2008.

[10] Dods, C., Smart, N., Stam, M.: Hash based digital signature schemes. In *Cryptography and Coding*, LNCS 3796, pages 96–115. Springer, 2005.

[11] ElGamal, T.: A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. Advances in Cryptology – CRYPTO '84, LNCS 196, pages 10–18. Springer, 1985.

[12] Goldwasser, S., Micali, S., Rivest, R.L.: A digital signature scheme secure against adaptive chosen-message attacks. In SIAM Journal on Computing, 17(2), pages 281–308, 1988.

[13] Grover, L. K.: A fast quantum mechanical algorithm for database search. Proceedings of the Twenty-Eighth Annual Symposium on the Theory of Computing, pages 212–219, New York, 1996. ACM Press.

[14] Jakobsson, M.: Fractal Hash Sequence Representation and Traversal. ISIT '02, p. 437. Available at `www.markus-jakobsson.com`.

[15] Johnson, D. and Menezes, A.: The Elliptic Curve Digital Signature Algorithm (ECDSA). Technical Report CORR 99-34, University of Waterloo, 1999. Available at `http://www.cacr.math.uwaterloo.ca`.

[16] Jakobsson, M., Leighton, T., Micali, S., Szydlo, M.: Fractal Merkle Tree Representation and Traversal. In RSA Cryptographers Track, RSA Security Conference 2003.

[17] Jutla, C., Yung, M.: PayTree: Amortized-Signature for Flexible Micropayments. 2nd USENIX Workshop on Electronic Commerce, pp. 213–221, 1996.

[18] Lamport, L.: Constructing digital signatures from a one way function. Technical Report SRI-CSL-98, SRI International Computer Science Laboratory, 1979.

[19] Lipmaa, H.: On Optimal Hash Tree Traversal for Interval Time-Stamping. In Proceedings of Information Security Conference 2002, LNCS 2433, pp. 357–371, Springer, 2002. Available at `www.tcs.hut.fi/ ~ helger/papers/lip02a/`.

[20] Malkin, T., Micciancio, D., Miner, S.: Efficient Generic Forward-Secure Signatures With An Unbounded Number Of Time Periods. Proceedings of Eurocrypt '02, pages 400–417.

[21] Merkle, R.C.: Secrecy, Authentication, and Public Key Systems. UMI Research Press, 1982. Also appears as a Stanford Ph.D. thesis in 1979.

[22] Merkle, R.C.: A Digital Signature Based on a Conventional Encryption Function. Proceedings of Crypto '87, pp. 369–378.

[23] Merkle, R.C.: A certified digital signature. Advances in Cryptology - CRYPTO '89 Proceedings, LNCS 435, pages 218–238, Springer, 1989.

[24] Micali, S.: Efficient Certificate Revocation. In RSA Cryptographers Track, RSA Security Conference 1997, and U.S. Patent No. 5,666,416.

[25] Naor, D., Shenhav, A., Wool, A.: One-time signatures revisited: Have they become practical. Cryptology ePrint Archive, Report 2005/442, 2005. `http://eprint.iacr.org/`.

[26] Naor, D., Shenhav, A., Wool, A.: One-time signatures revisited: Practical fast signatures using fractal merkle tree traversal. IEEE – 24th Convention of Electrical and Electronics Engineers in Israel, pages 255–259, 2006.

[27] Perrig, A., Canetti, R., Tygar, D., Song, D.: The TESLA Broadcast Authentication Protocol. Cryptobytes, Volume 5, No. 2 (RSA Laboratories, Summer/Fall 2002), pages 2–13. Available at `www.rsasecurity.com/rsalabs/cryptobytes/`.

[28] Rompel, J.: One-way Functions are Necessary and Sufficient for Secure Signatures. Proceedings of ACM STOC'90, pages 387–394, 1990.

[29] Rivest, R., Shamir, A.: PayWord and MicroMint–Two Simple Micropayment Schemes. CryptoBytes, Volume 2, No. 1 (RSA Laboratories, Spring 1996), pp. 7–11. Available at `www.rsasecurity.com/rsalabs/cryptobytes/`.

[30] Rogaway, P., Shrimpton, T.: Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In *Fast Software Encryption - FSE 2004*, LNCS 3017, pages 371–388. Springer, 2004.

[31] Rivest, R. L., Shamir, A., and Adleman, L.: A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. Communications of the ACM, 21(2):120–126, 1978.

[32] FIPS PUB 180-1, Secure Hash Standard, SHA-1. Available at `www.itl.nist.gov/fipspubs/fip180-1.htm`.

[33] Szydlo, M.: Merkle Tree Traversal in Log Space and Time. Advances in Cryptology - EUROCRYPT 2004, LNCS 3027, pages 541–554, Springer, 2004

[34] Szydlo, M.: Merkle Tree Traversal in Log Space and Time. Preprint, available at `www.szydlo.com`, 2003.

Index