

Darmstadt University of Technology  
Department of Computer Science  
Cryptography and Computeralgebra  
Prof. Dr. Johannes Buchmann

Diploma Thesis

# Efficient Java Implementation of GMSS



Sebastian Blume  
August 6, 2007

Supervisor: Erik Dahmen



# Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Diplomarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, August 2007

Sebastian Blume



# Acknowledgements

First of all, I would like to thank Prof. J. Buchmann for introducing me to the field of cryptography and for having aroused my interest in this topic. I thank Erik Dahmen for his helpful advice and support throughout the entire work on the thesis. I also want to thank my parents for their constant support in all intents and purposes. And finally, special thanks to Brita for her words of encouragement and for cheering me up at any time.



# Abstract

Digital signatures, today are one of the most important application of public key cryptography. Digital signatures are widely used in identification and authentication protocols for example as a part of SSL/TLS in the scope of the internet. Commonly uses digital signature schemes are RSA [16], DSA [9] and ECDSA [11]. Those schemes are based on number theoretic assumptions. Even today there exist quantum algorithms, that will solve all those problems in polynomial time if a large scale quantum computer will be build [17][2]. This means, all number theoretic assumptions currently used in digital signature schemes will become insecure. In consideration of the proceeding research of quantum computers, it is important to look for alternatives which do not rely on number theoretic assumptions.

A promising alternative are one-time signature schemes, such as the Winternitz One-Time Signature Scheme [6]. Their security does not rely on number theory but solely on the existence of cryptographic hash functions. Unfortunately, the property, that they can only be used to generate one signature, leads to an unpracticable key management problem. The Merkle Signature Scheme (MSS) proposed by Merkle in [13] addresses this problem, introducing the method of tree authentication. MSS is superimposed on an arbitrary one-time signature scheme. It uses a special tree structure to validate each single one-time signature key. MSS allows to verify up to  $2^{20}$  signatures using solely one public key. Because of this bounded number of signatures, especially in conjunction with its inefficient key pair generation times, MSS does not satisfy the demands of the most applications.

The focus of this thesis lies on the development of GMSS [3], a generalized variant of MSS, which allows a cryptographically unlimited signature capacity and provides competitive timings by using more efficient algorithms. GMSS only supports the Winternitz Signature Scheme as underlying one-time signature scheme. Among other factors, this makes GMSS extremely flexible because it provides the possibility to select appropriate parameters that determine the performance. The main task of this thesis was to implement a competitive and applicable version of GMSS using Java and integrate it as a module to the cryptographic service provider *FlexiProvider* [10]. This permits easy integration into applications that use the Java Cryptography Architecture (JCA) [14]. For the implementation, new efficient algorithms had to be formulated and those, that have already existed, had to be generalized. Particular attention was given to algorithms that evenly distribute the varying computation cost of the signatures.

Finally the timing characteristic of the implementation were tested using different parameters. We show that GMSS with a capacity of  $2^{80}$  signatures, is competitive compared to common signature schemes under certain circumstances.



# Contents

<b>1. Introduction</b>	<b>17</b>
1.1. Principles . . . . .	17
<b>2. MSS – Merkle Signature Scheme</b>	<b>21</b>
2.1. One Time Signature Schemes . . . . .	21
2.1.1. Lamport Diffie Signature Scheme . . . . .	21
2.1.2. Winternitz One-Time Signature Scheme . . . . .	22
2.2. Tree Authentication . . . . .	24
2.2.1. Merkle’s Tree Authentication Scheme . . . . .	26
2.3. MSS - Merkle Signature Scheme . . . . .	27
<b>3. GMSS – Generalized Merkle Signature Scheme</b>	<b>29</b>
3.1. General Construction . . . . .	30
3.2. Initial Seeds . . . . .	33
3.3. Winternitz OTSS Algorithms . . . . .	33
3.4. GMSS Key Pair Generation . . . . .	36
3.5. GMSS Signature Generation . . . . .	40
3.5.1. Online Part . . . . .	40
3.5.2. Offline Part . . . . .	41
3.6. GMSS Signature Verification . . . . .	49
<b>4. Implementation and Efficiency</b>	<b>51</b>
4.1. Java Implementation of GMSS . . . . .	51
4.2. Cost Functions . . . . .	52
4.3. Timings . . . . .	53
<b>5. Conclusions</b>	<b>57</b>
<b>A. ASN.1 Encoding</b>	<b>61</b>
<b>B. Object Identifiers</b>	<b>63</b>
<b>C. Java Examples</b>	<b>65</b>

*Contents*

# List of Algorithms

1.	Winternitz OTSS private key generation . . . . .	34
2.	Winternitz OTSS Key Pair Generation . . . . .	35
3.	Winternitz OTSS Signature Generation . . . . .	35
4.	Winternitz OTSS Signature Verification . . . . .	36
5.	Modified Treehash Algorithm . . . . .	37
6.	GMSS Key Pair Generation . . . . .	39
7.	GMSS Signature Generation . . . . .	41
8.	Distributed Root Signature Initialization . . . . .	43
9.	Distributed Root Signature Update . . . . .	43
10.	Distributed Leaf Calculation Initialization . . . . .	45
11.	Distributed Leaf Calculation Update . . . . .	46
12.	GMSS Update of Private Key . . . . .	48
13.	GMSS Signature Verification . . . . .	49

*List of Algorithms*

# List of Figures

2.1.	A binary tree of height 2 . . . . .	25
2.2.	Generation steps of a Merkle tree of height 2 . . . . .	25
2.3.	Authentication path of $l_2$ . . . . .	26
3.1.	General construction of GMSS . . . . .	30
3.2.	Generation of initial OTS seeds . . . . .	34
3.3.	The values of the key pair generation . . . . .	38
3.4.	Partial one-time signature generation . . . . .	42
3.5.	Partial root computation . . . . .	44
3.6.	Partial leaf computation . . . . .	44

*List of Figures*

# List of Tables

- 4.1. Expected timings and memory requirements on Sun Fire X2200 M2 . 54
- 4.2. Measured timings and memory requirements on Sun Fire X2200 M2 54
- 4.3. Expected timings and memory requirements on Asus V6J . . . . . 55
- 4.4. Measured timings and memory requirements on Asus V6J . . . . . 55
  
- B.1. OIDs assigned to GMSS . . . . . 63

*List of Tables*

# 1. Introduction

One of the basic properties of a conventional signature is, that the assertion represented by the document originates from the signatory, and this can be verified by the receiver of the document. Therefore, only the signer shall be able to generate the signature, but it shall be verifiable by anyone. Digital Signatures, designated to subscribe digital documents, can be regarded as being the electronic counterpart to handwritten signatures.

In the following we assume that the reader is familiar with the general notion of a cryptographic system. A detailed description can be found in [4]. This chapter describes the main principles of digital signatures. The important role of one-way functions is explained and a brief introduction in one-time signatures is given.

## 1.1. Principles

In contrast to conventional signatures a digital signature has to satisfy stronger requirements. The fact, that any digital data inherently can be copied precisely, necessitates a stronger connection between the data and the dedicated digital signature. Otherwise one could easily copy a known digital signature and attach it to an arbitrary document. To provide such a connection the signature is created as a function of the message. According to this, signing of different messages by the same signer must lead to different signatures and it must not be possible to derive a signature of a message  $m_2$  from a signature of a message  $m_1$  ( $m_1 \neq m_2$ ).

Digital Signatures are used whenever an information has to be definitely associated with an origin or a sender. The sender is able to attest that the message originates from him and was not modified. The receiver is able to prove that only the sender could have been the signer of the message. After a signature is generated it can no longer be repudiated by the sender.

The requirements of a qualified digital signature are:

- *Identity*: the ability to prove the the connection of signer and document
- *Non-Reusability*: a signature cannot be reused
- *Integrity*: modifications of the document after the signing can be detected
- *Non-Repudiation*: the singer cannot repudiate the act of signing.

The most digital signature schemes used in practical applications are based on asymmetric cryptosystems or more common *public key cryptosystems*. Some of them are

## 1. Introduction

at the same time encryption schemes, some are dedicated to signatures only. The most common are RSA [16], DSA [9] and ECDSA [11].

### Asymmetric Cryptography

The basic concept of asymmetric cryptography is to use a pair of keys for each communication partner. Each key pair consists of a private key  $d$  and a public key  $e$ . The private key is kept secret and the public key is distributed. The keys are mathematically related but it is not efficiently possible to derive the private key from the public key ( $f^{-1}$  cannot be computed). The private key is used to encrypt a message or to verify a signature, the public key is used to decrypt a ciphertext or to sign a message. A message is signed by applying the encryption function  $E$  with the public key of the receiver to the message. It can only be decrypted by using the decryption function  $D$  with the private key of the receiver. Hence, the receiver is the only one who can decrypt this message. The notation  $E_e(m)$  means, that the message  $m$  is encrypted using the encryption function  $E$  with the public key  $e$ .

An asymmetric cryptosystem has the following properties [7]:

1. It is easy to generate key pairs  $(e,d)$ ,  $d = f(e)$ ,  $D_d(E_e(m)) = m$
2. The encryption function  $E$  and the decryption function  $D$  are easy to compute
3. It is not efficiently possible to derive  $d$  from  $e$

An asymmetric cryptosystem that can be used for digital signatures must additionally satisfy the following requirement :

$$E_e(D_d(m)) = D_d(E_e(m)) = m$$

To sign a document the signer uses the decryption function  $D$  with his private key  $d$ . The verifier can verify the message by applying the encryption function  $E$  and the public key  $e$  to the signature and check if the result matches the message. In the context of digital signatures the private key is also referred to as *signature key*, the public key as *verification key*, the decryption function as *signature function* and the encryption function as *verification function*.

### One-Way Functions

The basic principle of any signature scheme are one-way functions. One-way functions are functions that are easy to compute but hard to invert.

A function  $f : X \rightarrow Y$  is called *one-way function*, if

1. for all  $x \in X$  the value of the function  $f(x)$  is easily to compute and
2. given  $y$  there is no efficient method to compute the  $x = f^{-1}(y)$ .

According to this definition, the one-way property of a function basically depends on the efficiency of existing algorithms that can be used to calculate the value of the function and its inverse. The existence of one way functions is a open conjecture, it cannot be proved that a function satisfies the one-way property. However, there are functions that are one-way functions according to the today's state of knowledge. They are easy to compute but there is no efficient inverting algorithm known.

Before a message of arbitrary length can be signed, it has to be compressed to a fixed length. This is done with a special type of one-way function, a cryptographic hash function. A cryptographic hash function does not only compress the message, but it also has to satisfy some security requirements.

### Cryptographic Hash Functions

Given two finite alphabets  $X$  and  $Y$ . A cryptographic hash function maps strings of arbitrary length to strings of constant length

$$H : X^* \rightarrow Y^n$$

and has the following properties:

1.  $H$  is a one-way function. (preimage resistance)
2. Given input  $m$ , the hash value  $h = H(m)$  is easy to compute.
3. Given hash value  $h = H(m)$  it is impossible to find a message  $m'$  ( $m \neq m'$ ) with  $H(m') = h$ . (second preimage resistance)
4. It is impossible to find  $m$  and  $m'$  ( $m \neq m'$ ) with  $H(m) = H(m')$ . (collision resistance)

The preimage and collision resistance of a cryptographic hash function prevent that an attacker could find other messages that fit to the signature and claim they were signed.

### Trapdoor One-Way Functions

The signature and verification functions of asymmetric cryptosystems are realized by another special variant of one-way function, the *trapdoor one-way function*. A trapdoor one-way function is easy to compute in one direction and hard to invert unless some secret information, the trapdoor, is known. In a signature scheme, this trapdoor is the signature key. The verification can be easily computed by anyone. The signature can only be computed with the secret information, the signature key. Signature schemes like RSA, DSA and ECDSA for example are based on number theoretic assumptions. Their security relies on the intractability of the integer factorization problem (RSA) and the difficulty of computing discrete logarithms in the multiplicative group of a prime field (DSA) or in the group of points of an elliptic curve over a finite field (ECDSA). Those problems can only be solved with an additional information, included in the private key.

## 1. Introduction

A function  $f : X \rightarrow Y$  is called *trapdoor one-way function*, if

1. for all  $x \in X$  the value of the function  $f(x)$  is easily to compute and
2. there exists some additional information, such that  $f^{-1}(y)$  efficiently can be computed.

### **One-Time Signatures**

One-time signature schemes in contrast are solely based on hash functions [5]. They are of special interest because one-way functions without a trapdoor are simpler to implement and typically more efficient to compute. This especially is an advantage for implementations in constraint devices. Additionally one-time signatures do not rely on number theoretic assumptions, hence they will remain secure even if a large scale quantum computer will be build or if algorithms will be discovered that solve the number theoretic problems efficiently. The security of one-time signatures relies only on the security of cryptographic hash functions. SHA-1 for example is a cryptographic hash function.

## 2. MSS – Merkle Signature Scheme

The Merkle signature scheme (MSS) [13] is a method of tree authentication. It consists of two parts. The first part is an arbitrary one-time signature scheme, e.g. the Winternitz one-time signature scheme. The second part is the Merkle's tree authentication scheme which is superimposed on the underlying signature scheme. It provides a mechanism to authenticate multiple one-time signature public keys with just one "master" key and some additional information, the so-called *authentication path*. To compute this authentication path, Merkle's tree authentication scheme makes use of a special binary tree, called *Merkle tree*, whose node values have to satisfy a certain constraint. Summarily, the Merkle signature scheme upgrades any one-time signature scheme to a multi time one.

In this section, first of all, the Winternitz Signature Scheme [6] is described on the basis of the Lamport-Diffie Signature Scheme [12]. Then, the principle of Merkle's tree authentication is described in detail and finally, the Merkle Signature Scheme is specified.

### 2.1. One Time Signature Schemes

One-time signature keys can be viewed as a set of public arrangements to a set of secrets, chosen by the signer. These arrangements are delivered to the verifier in an authenticated manner in advance. The signer signs a message by revealing a subset of his secrets depending on the content of the message. The verifier authenticates the message by checking if the secrets that were revealed are valid and if they correspond to the prior arrangements.

Unfortunately one-time signatures have the big disadvantage of its "one-timedness", the property that they can sign only one single message. If more than one signature would be generated using one key pair, an attacker may get enough information to forge a signatures. Hence, a new key pair is required for each signature. This leads to a key management problem, because for each intended signature a separate key pair has to be stored.

The best known one-time signature scheme is the Lamport-Diffie Signature Scheme which represents the basis of more advanced schemes like the Winternitz one-time signature scheme.

#### 2.1.1. Lamport Diffie Signature Scheme

The Lamport-Diffie One-Time Signature Scheme is based on the concept of one way functions. In the following we describe the basic idea of this scheme and present

## 2. MSS – Merkle Signature Scheme

some improvements.

Let us suppose  $A$  wants to sign a  $n$  bit message  $d = (d_1, \dots, d_n)$  and  $B$  wants to verify. Be  $F$  a one way function,  $A$  chooses  $2n$  random values  $x_1, \dots, x_{2n}$  and computes  $y_i = F(x_i)$  and exchanges  $(y_1, \dots, y_{2n})$  in authenticated manner with  $B$ .  $A$  generates the signature  $S = s_1 s_2 \dots s_{n-1} s_n$  by computing

$$s_i = \begin{cases} x_{2i-1} & \text{if } d_i = 0 \\ x_{2i} & \text{if } d_i = 1 \end{cases}.$$

To ensure the authenticity  $B$  can now verify  $S$  by checking if

$$F(s_i) = \begin{cases} y_{2i-1} & \text{if } d_i = 0 \\ y_{2i} & \text{if } d_i = 1 \end{cases},$$

for  $i = 1, \dots, n$ . If all the checks were successful he can be sure that only  $A$  was the sender.  $B$  can also demonstrate that  $A$  actually sent the information by presenting the received  $x_i$ . The only way  $B$  could have learned  $x_i$  is that  $A$  has revealed it.

The one-time property of the scheme can be demonstrated by a simple example. Assume that  $A$  would sign two messages  $M_1 = 1001$  and  $M_2 = 0010$ . Signing  $M_1$ ,  $A$  reveals  $x_2, x_3, x_5, x_8$ . Signing  $M_2$ , he reveals  $x_1, x_3, x_6, x_7$ . With those information anyone could for example sign the message 1011 on behalf of  $A$  by revealing  $x_2, x_3, x_6, x_8$ . The checksum does not avert this attack. Though an attacker cannot sign any message, he can sign some messages. And the more signatures are generated by  $A$ , the more messages will be possible to forge. That's why one-time signatures are only secure for a single signature because with each signature a part of the private key is revealed.

After all, this general method just signs every bit individually. In practical signature systems this results in a huge storage requirement for every participant. Robert Winternitz developed an improvement of the Lamport-Diffie method which reduces the signed message size by a factor  $w$ .

### 2.1.2. Winternitz One-Time Signature Scheme

In the Lamport Diffie method a user signs a bit by either make  $x_i$  or  $x_{i+1}$  public. The idea in the Winternitz method is still the same but now we process multiple bits at once and compute  $y$  by applying  $F$  repeatedly. We will illustrate the procedure with the following case. If we want to sign 4 bits at once, we compute  $y = F^{16}(x)$  and publish it. A 4 bit information is then signed by applying  $F$   $j$  times, where  $j$  is numeric value represented by the 4 bits.

For example when signing the 4 bit message 1001, the signer computes  $F^9(x)$  and reveals it. No one except the signer can generate this value, but anyone can verify that  $F^7(F^9(x)) = y$ . The amount of simultaneously processed bits is variable and we call it the *Winternitz parameter*.

In the Lamport Diffie scheme an altering of the signed message was prevented by using two secrets per message. In the Winternitz scheme this is ensured by a checksum  $C$ .

This scheme theoretically allows to process bit-sequences of user-defined length. On the one hand more simultaneously processed bits lead to a reduced signature size because one signature bit represents multiple message bits, but on the other hand the computational effort will increase when more bits are processed at once. Every raise of the Winternitz parameter by 1 will double the amount of necessary function calls  $F$  for computing the public  $y$ . This is a trade-off between time and space and the optimal parameter has to be found.

The following section describes the key generation, signature generation and signature verification procedures of the Winternitz One-Time Signature Scheme (Winternitz OTSS) in detail.

### Definition of the Scheme

$w$  denotes the Winternitz parameter.  $H$  denotes a hash function with domain  $\{0, 1\}$  and codomain  $\{0, 1\}^s$ .  $t$  denotes the amount of simultaneously computed blocks and is defined as

$$t = \lceil s/w \rceil + \lceil (\lfloor \log_2 \lceil s/w \rceil \rfloor + 1 + w)/w \rceil.$$

The first summand  $\lceil s/w \rceil$  refers to the message and the second summand  $\lceil (\lfloor \log_2 \lceil s/w \rceil \rfloor + 1 + w)/w \rceil$  refers to the checksum.

**Key Generation** The key pair generation process produces  $t$  random values  $x_1, x_2, \dots, x_t$ . The one-time signature key is

$$X = (x_1, x_2, \dots, x_t).$$

The public verification key  $Y$  is then computed by first computing

$$y_k = H^{2^w - 1}(x_k),$$

for  $k = 1, \dots, t$  and then computing

$$Y = H(y_1 || y_2 || \dots || y_t),$$

where  $F^k(x)$  denotes the one-way function applied  $k$  times and  $||$  the concatenation of two strings.

**Signature Generation** The hash of the message  $m$  is divided into  $\lceil s/w \rceil$  blocks  $b_1, b_2, \dots, b_{\lceil s/w \rceil}$  of  $w$ -bits length (padded with zero if necessary). Treating the  $b_k$  as integer we calculate the checksum

$$C = \sum_{k=1}^{\lceil s/w \rceil} 2^w - b_k$$

## 2. MSS – Merkle Signature Scheme

The binary representation of  $C$  is again divided into  $\lceil (\lceil \log_2 \lceil s/w \rceil \rceil + 1 + w)/w \rceil$  blocks  $b_{\lceil s/w \rceil}, \dots, b_t$  of  $w$ -bits length (padded with zero if necessary). Finally, the signature is computed by calculating

$$\sigma_k = H^{b_k}(x_k),$$

for  $k = 1, \dots, t$ . The signature of  $m$  is  $SIG = (\sigma_1, \sigma_2, \dots, \sigma_t)$ . The signature size is  $t * s$  bits.

**Verification** Given the hash of the message  $m$ , the signature  $SIG = (\sigma_1, \sigma_2, \dots, \sigma_t)$ , and the verification key  $Y$ , the blocks  $b_1, b_2, \dots, b_t$  are computed as in the signing process and then the signature parts  $\sigma_k$  are hashed  $2^w - 1 - b_k$  times.

$$y_k = H^{2^w - 1 - b_k}(\sigma_k).$$

Thereafter each of them has been hashed  $2^w - 1$  times in the signature and verification process together. The hashed concatenation of them should now equal the verification key  $Y$ . This means, the signature is accepted if  $Y = H(y_1 || y_2 || \dots || y_t)$ .

### Improvements

There is still a large amount of data to be stored when signing multiple messages. As the name implies, one-time signatures, more precisely the key pair of the one-time signature scheme, can be used *once* only. This means, for all the messages  $A$  wants to send to  $B$ ,  $B$  has to store the same amount of verification keys  $Y_i$  in advance. To reduce this space requirement one could transmit the respective  $Y_i$  together with the signature. Just adding the verification keys  $Y_i$ , when transmitting the signature, of course does not solve the storage problem. Anyone could claim to be  $A$  and send its own verification key  $Y_i$ . To authenticate those transmitted  $Y_i$ ,  $B$  still needs stored copies of  $A$ 's  $Y_i$ . Hence, the problem is to authenticate  $A$ 's  $Y_i$  without storing all the verification keys. Merkle developed a method which uses one single "master" public key to authenticate the transmitted  $Y_i$  in turn. This method is called *Merkle's tree authentication scheme*. It authenticates any  $Y_i$  of any user quickly and with minimal storage requirements. The Merkle's tree authentication scheme in conjunction with an one-time signature scheme (e.g. Winternitz signature scheme) is referred to as the Merkle signature scheme (MSS) [13]. As a basic principle it turns every one-time signature scheme into a multi time one. The next chapter describes how it works.

## 2.2. Tree Authentication

**Complete Binary Trees** A complete binary tree is a tree in which each node has exactly two children and in which all leaves are in the same depth. A complete binary tree has *height*  $h$  if it has  $2^h$  leaves and  $2^h - 1$  interior nodes. Leaves have the height 0, the root has the height  $h$  and thus the height of an interior node is the length of the path to an underlying leaf. A node of the tree is denoted by  $n_{i,j}$ . The index  $i$  labels on which height  $0, \dots, h$  of the tree the node is located. The  $i$  index of the root is

$h$ , the  $i$  index of a leaf is 0. The index  $j$  labels the horizontal position  $0, \dots, 2^i - 1$  of the node related to the number of nodes on the respective level. Each node of a binary tree may have a value or a condition or may even represent a separate data structure. Figure 2.1 shows a complete binary tree for height  $h = 2$ .

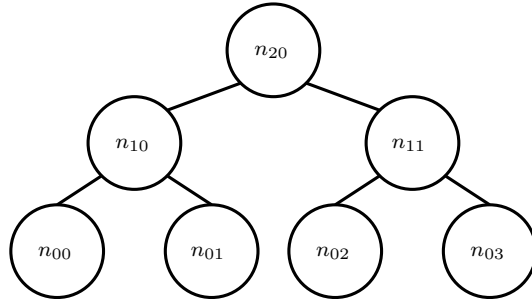


Figure 2.1.: A binary tree of height 2

**Merkle Trees** A Merkle tree is a complete binary tree, equipped with a one way function  $H$  and an assignment  $\Phi, n \rightarrow \Phi(n) \in \{0, 1\}^k$ , which maps a node  $n$  to a string of length  $k$ .  $\Phi(n)$  denotes the *value* of node  $n$ .  $H$  is typically a hash function such as SHA-1. The two child nodes  $n_{left}$  and  $n_{right}$ , of any interior node  $n_{parent}$  have to satisfy the following requirement:

$$\Phi(n_{parent}) = H(\Phi(n_{left}) || \Phi(n_{right}))$$

This means, that the value of an interior node always results from the hash values of the concatenated values of its left and right child. For each leaf  $l_j$  ( $l_j$  is a simplified notation for  $n_{0,j}$ ), the value  $\Phi(l_j)$  may be chosen randomly. The leaf values and the equation above, then determine the values of all the interior nodes. For simplification, in the following, we use the term "node" ( $n_{i,j}$ ) even if we actually mean the "node's value" ( $\Phi(n_{i,j})$ ). Figure 2.2 outlines the generation steps of a Merkle tree for height  $h = 2$ .

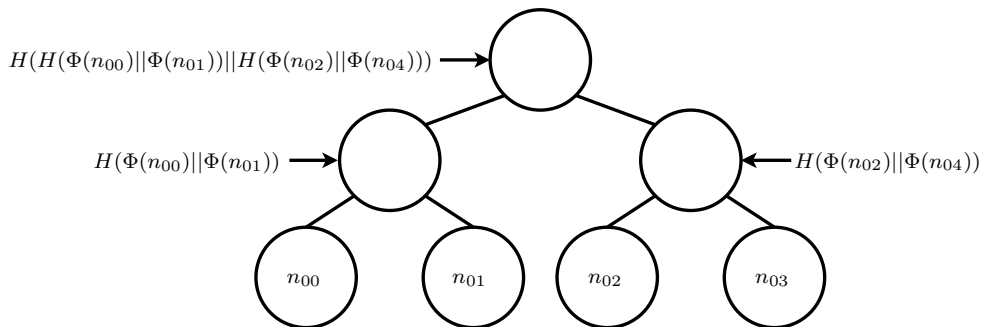


Figure 2.2.: Generation steps of a Merkle tree of height 2

### 2.2.1. Merkle's Tree Authentication Scheme

**Merkle Tree Generation** The idea of Merkle's tree authentication [13] is to use a Merkle tree to authenticate multiple one-time verification keys with one public key. The hash values of the one-time verification keys  $H(Y_i)$  are treated as the leaves of the Merkle tree. A Merkle tree of height  $h$  has  $2^h$  leaves and therewith it can authenticate  $2^h$  one-time verification keys. This means, the amount of possible one-time signatures depends on the height of the Merkle tree. The more one-time verification keys are to be authenticated the higher the Merkle tree has to be. According to the equation described in section 2.2, the interior nodes  $n_{i,j}$  up to the root are calculated depending on the leaf values  $l_j = n_{0,j}$ , or rather in this case the hash value of the one-time verification keys  $H(Y_j) = n_{0,j}$ . The nodes  $n_{i,j}$  on level  $i = 0, \dots, h$  are recursively defined as

$$n_{i,j} = H(n_{(i-1),2j} || n_{(i-1),2j+1}),$$

for  $j = 0, \dots, 2^i - 1$ .

The such computed root node  $n_{h,0}$  represents the Merkle public key referred to as *MSS public key*. Instead of many one-time verification keys, the MSS public key is the only information which has to be authenticated in advance. Thereafter additionally to this key only a specific set of nodes of the Merkle tree, the *authentication path*, are needed to authenticate each individual one-time verification key.

**Authentication Path Generation** Every leaf  $l_j$  has its own authentication path. The authentication path  $A$  consists of  $h$  nodes  $a_0, \dots, a_{h-1}$ . The authentication path is the sequence of sibling nodes of every node along the path from the leaf to the root. The nodes  $a_0, \dots, a_{h-1}$  of the authentication path of  $Y_j$  are generated by calculating

$$k = \left\lfloor \frac{j}{2^i} \right\rfloor$$

and

$$a_j = \begin{cases} n_{i,k+1} & \text{if } k \text{ is even} \\ n_{i,k-1} & \text{if } k \text{ is odd} \end{cases},$$

for  $i = 0, \dots, h - 1$ , where  $k$  indicates whether the node  $n_{i,k}$  is a left or a right child.

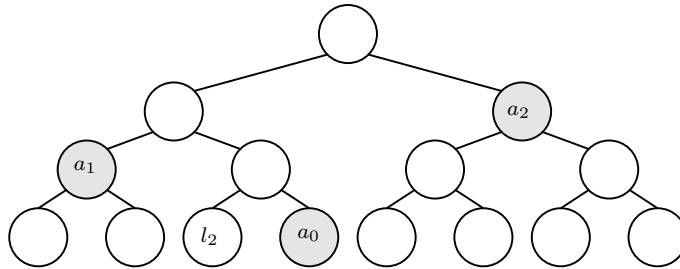


Figure 2.3.: Authentication path of  $l_2$

This means, the authentication path can be generated by following the path from the leaf to the root node by node. For each node on the way, the corresponding sibling node is added to the sequence of authentication path nodes. When arriving at the root all necessary sibling nodes have been added and the authentication path is generated. Figure 2.3 shows a Merkle tree for height  $h = 3$ . The authentication path nodes of leaf  $l_2$  are highlighted.

**Authentication Procedure** To authenticate a leaf  $l_j$ , or rather the one-time verification key  $Y_j$ , the receiver then just tries to reconstruct the root of the Merkle tree. The first node in the authentication path sequence is the sibling node of  $l_j$ . According to the equation  $n_{i,j} = H(n_{(i-1),2j} || n_{(i-1),2j+1})$ , the parent node of  $l_j$  is the hash value of the concatenation of  $l_j$  and the first node of the authentication path sequence. The next node in the authentication path sequence is then again the sibling of the just computed parent node. Those two yield the next node. This can be done iteratively until the root is computed. The authentication path always provides the necessary sibling node for each step.

In detail the reconstruction of the root works as follows. The receiver calculates the path  $P_i$ . As initial value he sets  $P_0 = H(Y_j)$ . For  $i = 0, \dots, h - 1$  he computes

$$k = \left\lfloor \frac{j}{2^i} \right\rfloor$$

and

$$P_{i+1} = \begin{cases} H(P_i || a_i) & \text{if } k \text{ is even} \\ H(a_i || P_i) & \text{if } k \text{ is odd} \end{cases},$$

where  $k$  indicates whether the node  $P_i$  is a left or a right child. When finished,  $P_h$  represents the reconstructed root. The leaf is only validated if the thus computed root  $P_h$  equals the MSS public key which actually is the prior arranged root of the sender's original tree.

## 2.3. MSS - Merkle Signature Scheme

The Merkle signature scheme basically is a  $N$ -time signature scheme consisting of an arbitrary one-time signature scheme and the Merkle's tree authentication scheme. The number of possible signatures  $N$  depends on the height  $h$  of the authentication tree  $N = 2^h$ . This section describes the key pair generation, signature generation and signature verification procedure of the Merkle signature scheme.

Assume that a cryptographic hash function  $H : \{0, 1\}^* \rightarrow \{0, 1\}^s$  which maps an arbitrary message to a  $s$ -bit hash value and a one-time signature scheme (OTSS) are given. Let  $h \in \mathbb{N}$  and suppose that  $2^h$  signatures are to be generated that are verifiable with only one MSS public key.

## 2. MSS – Merkle Signature Scheme

**MSS Key Pair Generation** At first, generate the  $2^h$  OTSS key pairs  $(X_j, Y_j)$ ,  $j = 0, \dots, 2^h - 1$ . The  $X_j$  are the OTSS signature keys, the  $Y_j$  are the OTSS verification keys. The sequence of the  $2^h$  signature keys  $X_j$  may be qualified as the MSS private key. To determine the MSS public key a Merkle tree of height  $h$  has to be constructed. The hash values of the one-time verification keys  $H(Y_j)$  form the leaves of the Merkle tree. According to section 2.2.1 the values of each inner node is the hash value of the concatenation of its two children. The MSS public key is the root of the thus computed Merkle tree.

**MSS Signature Generation** The OTSS key pairs are used sequentially. The MSS signature of a document  $d$  using the  $j$ th key pair  $(X_j, Y_j)$  consists of four parts. First the index  $j$  of the current one-time signature, second the OTSS signature  $\sigma$  of document  $d$  computed with the  $j$ th signature key  $X_j$ , third the  $j$ th verification key  $Y_j$ , and fourth the authentication path sequence  $A = (a_0, \dots, a_{h-1})$  for the verification key  $Y_j$ . The authentication path consists of the siblings of the  $h - 1$  nodes on the path from the  $j$ th leaf to the root which are calculated according to the algorithm in section 2.2.1. The resulting MSS signature is the tuple  $(j, Y_j, \sigma, A_j)$ .

**MSS Signature Verification** To verify a MSS signature  $(j, Y_j, \sigma, A_j)$  of a document  $d$  the verifier first verifies the one-time signature  $\sigma$  with the verification key  $Y_j$ . If this verification fails, the verifier already rejects the whole MSS signature as invalid. Otherwise, he still has to validate the authenticity of the supplied verification key  $Y_j$ . Using the index  $j$  and the authentication path  $A_j$  the verifier re-computes the path from the  $j$ th leaf to the root. Beginning with the verification key  $Y_j$  as initial node, the verifier recursively generates the parent node by computing the hash value of the concatenation of the node and its respective sibling from the authentication path, as described in section 2.2.1. If the thus recovered root equals the MSS public key, the signature is valid and accepted by the verifier.

### 3. GMSS – Generalized Merkle Signature Scheme

The Merkle Signature Scheme is limited to a tree height of  $h = 20$  for performance reasons. A higher tree would lead to an intolerable high key pair generation time. Another disadvantage is the large private key size because the values of all leaves have to be stored. Hence, the Merkle Signature Scheme permits only  $2^{20}$  signatures in an efficient manner. Due to those limits some improvements were developed which tend to make the scheme reasonable and competitive to common signature schemes.

In this thesis we propose GMSS [3], a variant of MSS, that increases the signature capacity from  $N = 2^{20}$  to  $N = 2^{80}$  and improves the timing properties. GMSS does not just increase the height of the Merkle tree to come up to  $2^{80}$  possible signatures. As mentioned above this would lead to an impractical high key generation time and private key size. GMSS makes use of another technique. For this purpose, multiple Merkle authentication trees, of arbitrary height, are arranged one upon the other. The bottom tree is used by applying MSS in order to sign the data. However, the root of this tree is not treated as the public key. In GMSS the root of a tree then again is signed by using MSS with the tree on the next layer. Finally, the root of the top tree is the new public key. When the a is depleted, that means  $2^h$  signatures have been used, a new tree at this layer is constructed, whose root again will be signed by the tree one layer above.

Using this method,  $2^{h_i}$  subtrees can be authenticated by each superior tree. Due to the arbitrary number of tree layers this theoretically allows an unlimited number of signatures, but the high key generation time and the increasing private key still limit the possibilities. Using GMSS with 4 layers of Merkle authentication trees, each having a height of  $h = 20$ , produces reasonable signature sizes and key generation costs. This permits a signature capacity of  $2^{80}$  which can be regarded as cryptographically unlimited signatures.

GMSS can be customized for special applications by selecting parameters that determine a trade-off between the possible number of signatures, the signature size and the signing and verification time. GMSS is a key-involving signature scheme, this means the private key is updated after every signature step. GMSS is designed to use solely the Winternitz one-time signature scheme.

In the following, GMSS is described in detail. First, we explain the general construction and principles of GMSS. Then the key pair generation, signature generation along with the necessary steps for the update of the private key, and the signature generation are described. At last, we give an overview about the timing-characteristics and the applicability of GMSS.

### 3.1. General Construction

The construction of GMSS can be considered as a tree with  $T$  layers where each node of this tree is in turn a Merkle tree. The heights of the trees in a certain layer  $i$  ( $i = 0, \dots, T - 1$ ) are denoted by  $h_i$ . Trees of different layers are allowed to differ in height. A Merkle Tree in layer  $i$  has  $2^{h_i}$  leafs. That implies, each Merkle tree in layer  $i$  is parent to  $2^{h_i}$  Merkle trees. The first layer  $i = 0$  contains only one Merkle Tree denoted by  $\mathcal{T}_{0,0}$ . Each further layer contains  $2^{h_0+\dots+h_{i-1}}$  Merkle trees. A Merkle tree in layer  $i$  is denoted by  $\mathcal{T}_{i,j}$ , where  $j = 0, \dots, 2^{h_0+\dots+h_{i-1}} - 1$  indicates their position from left to right. In contrary to the MSS the leaf values of the Merkle trees used by GMSS are the one-time verification keys  $Y_i$  themselves, instead of their hash values  $H(Y_i)$ . This is possible because the Winternitz OTSS verification keys are already hash values with the same length.

The principle of the GMSS authentication tree is to construct an authentication path from a Merkle tree on the deepest layer to the single tree on the top. This is realized by the following relationship between a parent Merkle tree and its children [3]: The root of a child tree is signed by the one-time signature key corresponding to a leaf of his parent tree.

In the following, when talking about leafs in context of signing, we mean the corresponding one-time signature key and in context of authentication in the MSS we mean the leafs value, respective the one-time verification key.

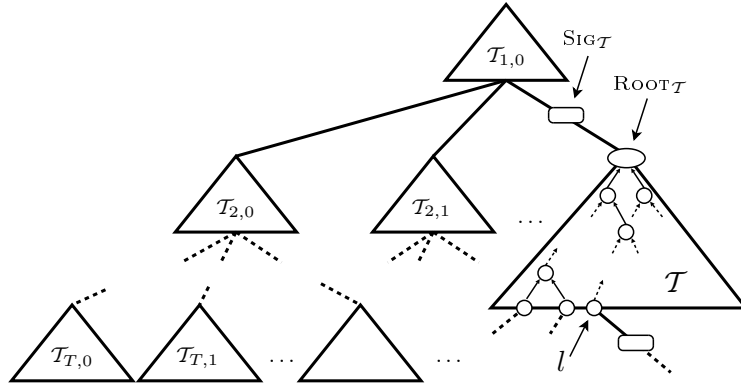


Figure 3.1.: General construction of GMSS

The root of a tree  $\mathcal{T}$  is denoted by  $\text{ROOT}_{\mathcal{T}}$ .  $\text{SIG}_{\mathcal{T}}$  denotes the one-time signature of  $\text{ROOT}_{\mathcal{T}}$ , which is generated using the respective leaf  $l$ , of  $\mathcal{T}$ 's parent. For any given signature  $\mathcal{S}$ , there is a unique path  $p$  from the Merkle tree on layer  $T - 1$  to the Merkle tree  $\mathcal{T}_{0,0}$  on the top layer 0. This path involves one Merkle tree at each

layer. In consideration of MSS each parent Merkle tree authenticates its  $2^{h_i}$  child Merkle trees. The authentication path of a leaf  $l$  of a Merkle tree  $\mathcal{T}$  is denoted by  $\text{AUTH}_{\mathcal{T},l}$ . That way, the root of the top Merkle tree recursively authenticates each individual of the  $2^{h_0+\dots+h_{T-1}}$  leafs of the Merkle trees on the deepest layer  $T-1$ . The leafs of the Merkle trees on the deepest layer  $T-1$  are used to sign the message digests  $d$ . Their signatures are called  $\text{SIG}_d$ .

The main advantage of this construction is, that only one Merkle tree of each layer has to be generated at the same time. This reduces the storage requirements. If one tree  $\mathcal{T}_{i,j}$  is depleted, the next one ( $\mathcal{T}_{i,j+1}$ ) is generated, and its root is signed by the next leaf of its parent tree. In the deepest layer, due to this layer hierarchy, successively  $2^{h_0+\dots+h_{T-2}}$  Merkle trees are generated. Hence,  $2^{h_0+\dots+h_{T-1}}$  signatures can be generated using one GMSS key pair. The root of the top Merkle tree  $\text{ROOT}_{\mathcal{T}_{0,0}}$  is the GMSS public key. In practice there are some additional information stored in the public key. This is described more detailed in section 3.4.

The GMSS private key consists of all information that are necessary for immediate signature generation. This means, that all the values, except of the one-time signature of  $d$ , that are necessary to generate a signature of a message  $m$ , are already precomputed and are available in the moment of signing. The computation of the one-time signature of  $d$  and the output of the GMSS signature is called the *online part*. The ensuing prearrangements and precalculations to provide the necessary information required for the upcoming signatures are called the *offline part*.

The sequences of leafs of each layer, or rather their corresponding Winternitz signature keys, are computed using a pseudo random number generator (PRNG) combined with an initial prearranged random seed  $\text{SEED}_{\mathcal{T}}$  for each layer. Consequently it is no longer necessary to remember every leaf, as a leaf can be computed from this seed anytime again. This reduces the private key size once more.

GMSS uses solely the Winternitz one-time signature scheme for signing digest  $d$  and  $\text{ROOT}_{\mathcal{T}}$ . The Winternitz parameter  $w_i$  can be specified for each individual layer. The sequence of Merkle tree heights and the sequence of the different Winternitz parameters  $w_i$  of each layer are combined to the *GMSS parameter set*

$$\mathcal{P} = (T, (h_0, \dots, h_{T-1}), (w_0, \dots, w_{T-1})).$$

The variability of the parameter set allows a trade-off between the number of possible signatures, the signature size and the signature and key pair generation time. Raising the height of the Merkle trees, for example, leads to more possible signatures, but also to more necessary calculations for the generation of the tree. Another example how the timings of signature and key pair generation can be affected, is to alter the value of  $w$ . The greater  $w$  is chosen, the smaller the signature becomes because the Winternitz OTSS then processes more bits at once. On the other hand the computational effort for signature and key generation increases as more hash function calls are needed. This flexibility makes GMSS very adaptive to different applications.

GMSS uses just a sole prearranged hash function  $H$  for all hash calculations of the Winternitz signature scheme, of the Merkle's authentication tree as well as for the

### 3. GMSS – Generalized Merkle Signature Scheme

generation of the seeds with the PRNG. This function can be chosen in advance but then is applied all over the entire GMSS system. A commonly used hash function for example is SHA-1.

In the following it is assumed that the chosen parameter set  $\mathcal{P}$  and the chosen hash function  $H$  are known. In a practical implementation it is a suitable solution to append this information to the GMSS private and public key.

#### General Procedure

To generate a GMSS key pair the signer has to compute the first Merkle tree of each layer to obtain the authentication paths  $\text{AUTH}_{\mathcal{T},l}$ , and except for the top tree, to generate the their root signatures  $\text{SIG}_{\mathcal{T}}$ . Those together with each layer's initial seed  $\text{SEED}_{\mathcal{T},l}$  are part of the GMSS private key. The root of the top tree is the essential part of the GMSS public key. The detailed key pair generation procedure is described in section 3.4.

To sign a message  $m$  with GMSS the signer has to compute a message digest  $d$  of  $m$  as the first part of the GMSS signature. The digest  $d$  is signed with the Winternitz one-time signature key corresponding to the current leaf of the current deepest Merkle tree. Note, that this leaf represents the Winternitz one-time verification key. To make the verifier capable to authenticate this leaf the signer has to add some further information. First, the authentication paths  $\text{AUTH}_{\mathcal{T},l}$  of all Merkle tree on the path  $p$  are needed and also the signatures of their roots  $\text{SIG}_{\mathcal{T}}$ , except for the top tree, are added. Section 3.5 describes the GMSS signature generation process in detail. Finally the GMSS signature contains the Winternitz signature of the message  $\text{SIG}_d$ , the  $T - 1$  signatures of the roots of the Merkle trees  $\text{SIG}_{\mathcal{T}}$  and  $T$  authentication paths  $\text{AUTH}_{\mathcal{T},l}$  of the involved Merkle trees.

The GMSS signature verification process is performed analogically. The one-time signatures of the message digest  $d$  and the root signatures of the Merkle trees on path  $p$  are verified. Therefore, the roots of the trees are necessary. The verifier obtains them for free during the MSS verification procedure. The verification procedure of MSS is applied successively for every signature  $\text{SIG}_{\mathcal{T}}$  (or  $\text{SIG}_d$ ) on path  $p$ . If any part of the GMSS verification fails, the signature will be rejected. The GMSS signature verification is described in detail in section 3.6.

After a signature generation procedure, the next authentication path for the next leaf has to be computed. If necessary, even new trees have to be generated and their roots have to be signed. Since those information are part of the GMSS private key, it has to be updated after every usage. For this reason GMSS is denoted as a *key involving signature scheme*. The amount of necessary recalculations varies depending on the position of the upcoming leaf. At the best only the deepest tree is involved, but at the worst the update requires the generation of new trees and the computation of authentication paths on all layers. To minimize this variance of the computational effort GMSS evenly distributes certain parts of the prospective calculations over all leaves. The algorithms of those distributed calculations are described in section 3.5.

## 3.2. Initial Seeds

For the generation of every leaf, or rather every Winternitz one-time verification key, the generation of random data is required. Since the one-time keys as well as the random data cannot be stored altogether, we need a pseudo random number generator (PRNG) which provides the possibility to regenerate all the random values at any time whenever they are required.

Let  $H : \{0, 1\}^* \rightarrow \{0, 1\}^s$  be the cryptographic hash function with output length  $s$ . The pseudo random number generator (PRNG)  $f : \{0, 1\}^s \rightarrow \{0, 1\}^s \times \{0, 1\}^s$ ,  $\text{SEED}_{\text{IN}} \mapsto (\text{SEED}_{\text{OUT}}, \text{RAND})$  maps an incoming  $\text{SEED}_{\text{IN}}$  to an random value  $\text{RAND}$  and an outgoing seed  $\text{SEED}_{\text{OUT}}$ . To assure interoperability we use the PRNG described in [8], which requires only one single call to the hash function  $H$ :

$$\begin{aligned} \text{RAND} &\leftarrow H(\text{SEED}_{\text{IN}}), \\ \text{SEED}_{\text{OUT}} &\leftarrow (1 + \text{SEED}_{\text{IN}} + \text{RAND}) \pmod{2^n} \end{aligned}$$

This function generates an arbitrary number of random values which each depend on one initial seed and therefore may be regenerated easily. Because of forward-security and performance based reasons not all random values over all trees should depend on only one seed. Hence, every layer of the GMSS tree has its own initial seed which is denoted by  $\text{SEED}_{\mathcal{T}_{i,0},0}$ . Starting with this initial seed, the PRNG sequentially generates the random value for every leaf in that layer.

## 3.3. Winternitz OTSS Algorithms

### Winternitz OTSS Key Generation

To generate the internal seed for the  $l$ th Winternitz OTS signature key of Merkle tree  $\mathcal{T}_{i,j}$  we need the seed for leaf  $l$   $\text{SEED}_{\mathcal{T}_{i,j},l}$  and apply the PRNG  $f$  to compute a random value denoted as  $\text{SEED}_{\text{OTS}}$ .

$$(\text{SEED}_{\mathcal{T}_{i,j},l+1}, \text{SEED}_{\text{OTS}}) \leftarrow f(\text{SEED}_{\mathcal{T}_{i,j},l})$$

The  $\text{SEED}_{\mathcal{T}_{i,j},l+1}$  is stored and used to generate the  $(l+1)$ th signature key, whereas the random value  $\text{SEED}_{\text{OTS}}$  is used as initial value for the one-time signature key generation within the Winternitz OTSS key pair generation algorithm. If the current signature key is associated with the last leaf of tree  $\mathcal{T}_{i,j}$  ( $l = 2^{h_i} - 1$ ), the updated seed is used as initial seed for the next Merkle tree  $\mathcal{T}_{i,j+1}$ , i.e.  $(\text{SEED}_{\mathcal{T}_{i,j+1},0}, \text{SEED}_{\text{OTS}}) \leftarrow f(\text{SEED}_{\mathcal{T}_{i,j},2^{h_i}-1})$ .

### 3. GMSS – Generalized Merkle Signature Scheme

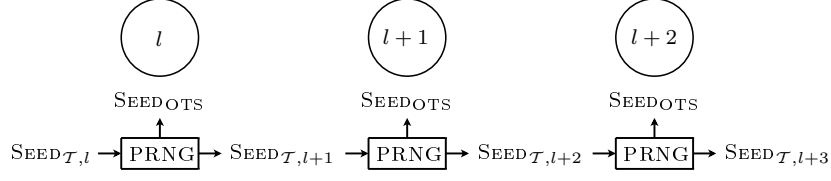


Figure 3.2.: Generation of initial OTS seeds

The Winternitz OTS signature key is computed by applying

$$(\text{SEED}_{\text{OTS}}, x_k) \leftarrow f(\text{SEED}_{\text{OTS}}), k = 1, \dots, t_{w_i}$$

sequentially  $t_{w_i}$  times where  $t_{w_i} = \lceil s/w_i \rceil + \lceil (\lceil \log_2 \lceil s/w_i \rceil \rceil + 1 + w_i)/w_i \rceil$ .  $X = (x_1, \dots, x_{t_{w_i}})$  is the one-time signature key. The one-time signature key generation is described in Algorithm 1.

---

#### Algorithm 1 Winternitz OTSS private key generation

---

**Input:** a seed  $\text{SEED}_{\text{OTS}} \in_R \{0, 1\}^s$  chosen uniformly at random

**Output:** a Winternitz OTSS private key  $X$

- 1: set  $s_0 = \text{SEED}_{\text{OTS}}$
  - 2: **for**  $k = 1, \dots, t$  **do**
  - 3:     compute  $(s_k, x_k) = f(s_{k-1})$
  - 4: set  $X = (x_1, \dots, x_t)$
  - 5: **return**  $X$
- 

Having computed the one-time signature key sequence  $X = (x_1, \dots, x_{t_{w_i}})$ , the one-time verification key can be constructed. The value of leaf  $l$  of Merkle tree  $\mathcal{T}_{i,j}$  which actually also is the one-time verification key  $Y$  is given as

$$Y = H(H^{2^{w_i}-1}(x_1) || \dots || H^{2^{w_i}-1}(x_{t_w})).$$

$H^k(x)$  denotes the hash function applied  $k$  times and  $||$  the concatenation of two strings.

The entire OTSS key pair generation process is described in Algorithm 2. The input parameter of the key pair generation is the initial seed  $\text{SEED}_{\text{OTS}}$ . In the first part, Algorithm 1 is applied, by passing the  $\text{SEED}_{\text{OTS}}$  as parameter. The output of Algorithm 1 is the one-time signature key  $X$  which then is used in the second part of Algorithm 2 to compute the one-time verification key  $Y$ . The output of Algorithm 2 is the generated Winternitz OTSS key pair  $(X, Y)$ .

---

**Algorithm 2** Winternitz OTSS Key Pair Generation

---

**Input:** a seed  $\text{SEED}_{\text{OTS}} \in_R \{0, 1\}^s$  chosen uniformly at random**Output:** a Winternitz OTSS key pair  $(X, Y)$ 

- 1: compute the private key  $X$ , where  $X = (x_1, \dots, x_t)$  :  
 $X \leftarrow \text{Algorithm 1}(\text{SEED}_{\text{OTS}})$
  - 2: compute  $y_k = H^{2^w-1}(x_k)$  for  $k = 1, \dots, t$ .
  - 3: compute  $Y = H(y_1 || \dots || y_t)$ , where  $||$  denotes concatenation.
  - 4: **return**  $(X, Y)$
- 

**Winternitz OTSS Signature Generation**

The Winternitz one-time signature generation works as described in section 2.1.2. Given  $w_i$  related to a layer  $i$  and a hash function  $H : \{0, 1\}^* \rightarrow \{0, 1\}^s$ . The one-time signature is computed by splitting the the digest  $H(d)$  into  $\lceil s/w_i \rceil$  blocks  $b_1, \dots, b_{\lceil s/w_i \rceil}$  of length  $w_i$ . Then compute a checksum  $C$  which in turn is split in  $\lceil (\lceil \log_2 \lceil s/w_i \rceil \rceil + 1 + w_i)/w_i \rceil$  blocks  $b_{\lceil s/w_i \rceil + 1}, \dots, b_t$  blocks of length  $w_i$ . The content of each of the resulting  $t$  blocks,  $t = \lceil s/w_i \rceil + \lceil (\lceil \log_2 \lceil s/w_i \rceil \rceil + 1 + w_i)/w_i \rceil$ , affects the amount of hash function calls that are applied to the corresponding one-time signature key part. The resulting one-time signature has a length of  $t * s$  bits and is defined by

$$\sigma = (H^{b_1}(x_1), \dots, H^{b_t}(x_t))$$

---

**Algorithm 3** Winternitz OTSS Signature Generation

---

**Input:** document  $d$ , signature key  $X$ **Output:** one-time signature  $\sigma$  of  $d$ 

- 1: compute the  $s$  bit hash value  $H(d)$  of document  $d$ .
- 2: split the binary representation of  $H(d)$  into  $\lceil s/w \rceil$  blocks  $b_1, \dots, b_{\lceil s/w \rceil}$  of length  $w$ , padding  $H(d)$  with zeros from the left if required.
- 3: treat  $b_i$  as the integer encoded by the respective block and compute the checksum

$$C = \sum_{k=1}^{\lceil s/w \rceil} 2^w - b_k.$$

- 4: split the binary representation of  $C$  into  $\lceil (\lceil \log_2 \lceil s/w \rceil \rceil + 1 + w)/w \rceil$  blocks  $b_{\lceil s/w \rceil + 1}, \dots, b_t$  of length  $w$ , padding  $C$  with zeros from the left if required.
  - 5: treat  $b_k$  as the integer encoded by the respective block and compute  $\sigma_k = H^{b_k}(x_k)$ ,  $k = 1, \dots, t$ , where  $H^0(x) := x$ .
  - 6: **return**  $\sigma = (\sigma_1, \dots, \sigma_t)$ .
-

### 3. GMSS – Generalized Merkle Signature Scheme

#### Winternitz OTSS Signature Verification

The Winternitz OTSS signature verification algorithm is slightly simplified compared to the procedure described in section 2.1.2. Algorithm 4, which is used by GMSS does not require the verification key as argument and for that reason does not return *true* or *false* but just the such computed key which is expected to match the one-time verification key  $Y$  corresponding to the signature  $\sigma$ . The test, if the return value actually equals  $Y$  has to be performed external. The fact, that the algorithm returns the alleged verification key instead of a boolean, has the advantage, that it can also be used to compute the verification key from an already generated signature. This is possible because the signature is an intermediate value in the generation process from the signature key  $X$  to the verification key  $Y$ . On average this halves the necessary amount of hash function calls.

---

**Algorithm 4** Winternitz OTSS Signature Verification

---

**Input:** document  $d$ , signature  $\sigma = (\sigma_1, \dots, \sigma_t)$

**Output:** the one-time verification key  $\Phi$

- 1: compute  $b_1, \dots, b_t$  as in Algorithm 3.
  - 2: compute  $\phi_k = H^{2^{w-1}-b_k}(\sigma_k)$  for  $k = 1, \dots, t$ .
  - 3: compute  $\Phi = H(\phi_1 || \dots || \phi_t)$ .
  - 4: **return**  $\Phi$
- 

#### 3.4. GMSS Key Pair Generation

The key pair generation computes the public and private keys for GMSS from the parameter  $\mathcal{P}$  and the initial seeds  $\text{SEED}_{\mathcal{T}_{0,0}}, \dots, \text{SEED}_{\mathcal{T}_{T-1,0}}$ .

##### GMSS Public Key

The main part of the GMSS public key essentially is the root of the top level tree  $\text{ROOT}_{\mathcal{T}_{0,0}}$ . As the root is the value generated by the hash function  $H : \{0, 1\}^* \rightarrow \{0, 1\}^s$  its length is  $s$  bits. Additionally the parameter set  $\mathcal{P} = (T, (h_0, \dots, h_{T-1}), (w_0, \dots, w_{T-1}))$  is a component of the GMSS public key. This is necessary because the information about the Winternitz parameter  $w_i$  as well as the height  $h_i$  of the authentication path trees are needed during the verification process. The size of the parameter set depends on the number of layers  $T$  of the GMSS tree. For each layer  $i$  the Winternitz parameter  $w_i$  and the height  $h_i$  is stored.

GMSS Public Key:  $(\text{ROOT}_{\mathcal{T}_{0,0}}, \mathcal{P})$

**GMSS Private Key**

The key pair generation step computes the following components of the GMSS private key:

- $\text{SEED}_{\mathcal{T}_{i,0,0}}$ , the seed of the first tree of layer  $i = 0, \dots, T - 1$
- $\text{AUTH}_{\mathcal{T}_{i,0,0}}$ , the authentication path of the first tree of layer  $i = 0, \dots, T - 1$
- $\text{SIG}_{\mathcal{T}_{i,0}}$ , the signatures of the roots of the first tree of layer  $i = 1, \dots, T - 1$
  
- $\text{AUTH}_{\mathcal{T}_{i,1,0}}$ , the authentication path of the second tree of layer  $i = 1, \dots, T - 1$
- $\text{ROOT}_{\mathcal{T}_{i,1}}$ , the root of the second tree of layer  $i = 1, \dots, T - 1$
  
- $\text{SEED}_{\mathcal{T}_{i,2,0}}$ , the seed of the third tree of layer  $i = 1, \dots, T - 1$

**GMSS Key Pair Generation Procedure**

The key pair generation procedure is described in Algorithm 6. The first part of the key pair generation is the computation of the root of the first tree in each layer  $\text{ROOT}_{\mathcal{T}_{i,0}}$ ,  $i = 0, \dots, T - 1$ . This is realized by a classical algorithm referred to as treehash [18]. GMSS uses a slightly modified version of this algorithm, described in Algorithm 5, which additionally computes the authentication path  $\text{AUTH}_{\mathcal{T}_{i,0,0}}$  of the first leaf by the way.

---

**Algorithm 5** Modified Treehash Algorithm

---

**Input:** a leaf value  $Y$ , algorithm stack  $S$ , sequence of nodes  $A$

**Output:** updated stack  $S$  and updated sequence  $A$

- 1: push  $l$  to  $S$
  - 2: **while** top two nodes of  $S$  have the same height **do**
  - 3:     pop  $n_1$  from  $S$
  - 4:     **if**  $n_1$  has greater height than last node in  $A$  or  $A$  is empty **then**
  - 5:         append top node of stack to  $A$
  - 6:     pop  $n_2$  from  $stack$
  - 7:     push  $in = H(n_2||n_1)$  to  $S$
  - 8: **return** stack  $S$ , sequence of nodes  $A$
- 

The root of a tree can be computed by successively applying Algorithm 5  $2^{h_i}$  times, where the input value are the  $2^{h_i}$  leafs of the tree, which are supplied in sequential order from left to right. The necessary leaf values  $Y$  are generated using  $\text{SEED}_{\mathcal{T}_{i,j,l}}$  and Algorithm 2. For each call, the algorithm generates the highest interior nodes which are computable with the leafs which have been passed so far. A stack  $S$  of nodes is used to store those intermediate node values. Whenever a node, which is part of the first authentication path, arises on the stack the algorithm appends

### 3. GMSS – Generalized Merkle Signature Scheme

it to the sequence of nodes  $A$ . This allows us to generate the first authentication path  $\text{AUTH}_{\mathcal{T}_{i,0,0}}$  for free, since all necessary nodes are passed during the calculation of the root. After  $2^{h_i}$  calls, the stack  $S$  contains the root of the tree  $\text{ROOT}_{\mathcal{T}_{i,0}}$  and the sequence of nodes  $A$  is the authentication path of the first leaf  $\text{AUTH}_{\mathcal{T}_{i,0,0}}$ .

Besides, the signatures of all roots, except of the top root,  $\text{SIG}_{\mathcal{T}_{i,0}}$ ,  $i = 1, \dots, T - 1$  are generated. Algorithm 6 implements the root calculation in reverse order beginning with tree  $\mathcal{T}_{T-1,0}$  up to tree  $\mathcal{T}_{0,0}$ . Consequently the root  $\text{ROOT}_{\mathcal{T}_{i,0}}$  of the tree  $\mathcal{T}_{i,0}$  is available when the first leaf of tree  $\mathcal{T}_{i-1,0}$  is generated. The signature of the root  $\text{SIG}_{\mathcal{T}_{i,0}}$  is an intermediate value of this leaf generation. This is realised by computing the leaf not as usual, but by first signing the root with the signature key  $X$  which is generated using Algorithm 1 and then verifying the just computed signature with Algorithm 4 which yields the verification key or the leaf value  $Y$ . That way, the signatures of the roots can be obtained for free during the generation of the first leaf.

Next, the roots  $\text{ROOT}_{\mathcal{T}_{i,1}}$  and the authentication paths  $\text{AUTH}_{\mathcal{T}_{i,1,0}}$  of the succeeding trees  $\mathcal{T}_{i,1}$  in each layer are computed with Algorithm 5. The trees  $\mathcal{T}_{i,1}$  are only generated for the layers  $i = 1, \dots, T - 1$ , because in the top layer  $i = 0$  there exists only one tree. Having generated the last leaves of the first trees, the initial seeds for the succeeding trees are already available as described above ( $\text{SEED}_{\mathcal{T}_{i,0,2^{h_i}}} = \text{SEED}_{\mathcal{T}_{i,1,0}}$ ). When  $\text{ROOT}_{\mathcal{T}_{i,1}}$  and  $\text{AUTH}_{\mathcal{T}_{i,1,0}}$  have been computed, the last generated seeds of the trees  $\mathcal{T}_{i,1}$  again are the initial seeds for the trees  $\mathcal{T}_{i,2}$  ( $\text{SEED}_{\mathcal{T}_{i,2,0}}$ ), which are stored as part of the private key to allow an efficient precalculation of the trees  $\mathcal{T}_{i,2}$  during the signing process.

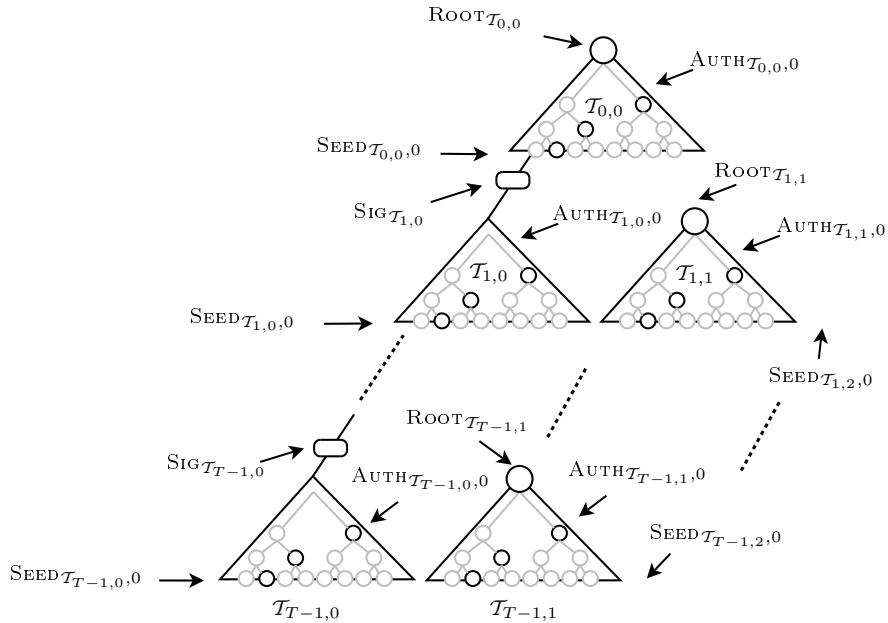


Figure 3.3.: The values of the key pair generation

**Algorithm 6** GMSS Key Pair Generation

---

**Input:** parameter set  $\mathcal{P} = (T, (h_0, \dots, h_{T-1}), (w_0, \dots, w_{T-1}))$ ,  
 $T$  seeds  $(\text{SEED}_{\mathcal{T}_{0,0,0}}, \dots, \text{SEED}_{\mathcal{T}_{T-1,0,0}})$  chosen uniformly at random in  $\{0, 1\}^s$

**Output:** a GMSS key pair  $(priv, pub)$

- 1: **for**  $i = T - 1, \dots, 0$  **do**
- 2:   set  $N = 2^{h_i}$
- 3:   compute seed for Winternitz key pair generation:  
 $(\text{SEED}_{\mathcal{T}_{i,0,1}}, \text{SEED}_{\text{OTS}}) \leftarrow f(\text{SEED}_{\mathcal{T}_{i,0,0}})$
- 4:   initialize empty stack  $\mathcal{S}_i$  and empty sequence of nodes  $\text{AUTH}_{\mathcal{T}_{i,0,0}}$
- 5:   **if**  $i = T - 1$  **then**
- 6:     compute first Winternitz key pair  $(X_0, Y_0) \leftarrow \text{Algorithm 2}(\text{SEED}_{\text{OTS}})$
- 7:     compute  $(\mathcal{S}_i, \text{AUTH}_{\mathcal{T}_{i,0,0}}) \leftarrow \text{Algorithm 5}(Y_0, \mathcal{S}_i, \text{AUTH}_{\mathcal{T}_{i,0,0}})$
- 8:   **else**
- 9:     compute Winternitz private key  $(X_0) \leftarrow \text{Algorithm 1}(\text{SEED}_{\text{OTS}})$
- 10:    compute one-time signature of  $\text{ROOT}_{\mathcal{T}_{i-1,0}}$  :  
 $\text{RSIG}_{\mathcal{T}_{i-1,0}} \leftarrow \text{Algorithm 3}(\text{ROOT}_{\mathcal{T}_{i-1,0}}, X_0)$
- 11:    compute Winternitz public key  $Y_0$  by verifying  $\text{SIG}_{\mathcal{T}_{i-1,0}}$ :  
 $Y_0 \leftarrow \text{Algorithm 4}(\text{ROOT}_{\mathcal{T}_{i-1,0}}, \text{SIG}_{\mathcal{T}_{i-1,0}})$
- 12:    compute  $(\mathcal{S}_i, \text{AUTH}_{\mathcal{T}_{i,0,0}}) \leftarrow \text{Algorithm 5}(Y_0, \mathcal{S}_i, \text{AUTH}_{\mathcal{T}_{i,0,0}})$
- 13:    **for**  $l = 1, \dots, N - 1$  **do**
- 14:     compute seed for Winternitz key pair generation:  
 $(\text{SEED}_{\mathcal{T}_{i,0,l+1}}, \text{SEED}_{\text{OTS}}) \leftarrow f(\text{SEED}_{\mathcal{T}_{i,0,l}})$
- 15:     compute Winternitz key pair  $(X_l, Y_l) \leftarrow \text{Algorithm 2}(\text{SEED}_{\text{OTS}})$
- 16:     compute  $(\mathcal{S}_i, \text{AUTH}_{\mathcal{T}_{i,0,0}}) \leftarrow \text{Algorithm 5}(Y_l, \mathcal{S}_i, \text{AUTH}_{\mathcal{T}_{i,0,0}})$
- 17:     $\text{ROOT}_{\mathcal{T}_{i,0}}$  is the single node in  $\mathcal{S}_i$ ;
- 18:    set the initial seed of the next tree:  $\text{SEED}_{\mathcal{T}_{i,1,0}} \leftarrow \text{SEED}_{\mathcal{T}_{i,0,N}}$
- 19: **for**  $i = T - 1, \dots, 1$  **do**
- 20:   set  $N = 2^{h_i}$
- 21:   initialize empty stack  $\mathcal{S}_i$  and empty sequence of nodes  $\text{AUTH}_{\mathcal{T}_{i,1,0}}$
- 22:   **for**  $l = 0, \dots, N - 1$  **do**
- 23:     compute seed for Winternitz key pair generation:  
 $(\text{SEED}_{\mathcal{T}_{i,1,l+1}}, \text{SEED}_{\text{OTS}}) \leftarrow f(\text{SEED}_{\mathcal{T}_{i,1,l}})$
- 24:     compute  $(X_l, Y_l) \leftarrow \text{Algorithm 2}(\text{SEED}_{\text{OTS}})$
- 25:     compute  $(\mathcal{S}_i, \text{AUTH}_{\mathcal{T}_{i,1,0}}) \leftarrow \text{Algorithm 5}(Y_l, \mathcal{S}_i, \text{AUTH}_{\mathcal{T}_{i,1,0}})$
- 26:     $\text{ROOT}_{\mathcal{T}_{i,1}}$  is the single node in  $\mathcal{S}_i$
- 27:    set the initial seed of the next tree:  $\text{SEED}_{\mathcal{T}_{i,2,0}} \leftarrow \text{SEED}_{\mathcal{T}_{i,1,N}}$
- 28: set  $pub = (\text{ROOT}_{\mathcal{T}_{0,0}}, P)$
- 29: set  $priv = ((\text{SEED}_{\mathcal{T}_{0,0,0}}, \dots, \text{SEED}_{\mathcal{T}_{T-1,0,0}}),$   
 $(\text{SEED}_{\mathcal{T}_{1,2,0}}, \dots, \text{SEED}_{\mathcal{T}_{T-1,2,0}}),$   
 $(\text{AUTH}_{\mathcal{T}_{0,0,0}}, \dots, \text{AUTH}_{\mathcal{T}_{T-1,0,0}}),$   
 $(\text{AUTH}_{\mathcal{T}_{0,1,0}}, \dots, \text{AUTH}_{\mathcal{T}_{T-1,1,0}}),$   
 $(\text{ROOT}_{\mathcal{T}_{1,1}}, \dots, \text{ROOT}_{\mathcal{T}_{T-1,1}}),$   
 $(\text{SIG}_{\mathcal{T}_{1,0}}, \dots, \text{SIG}_{\mathcal{T}_{T-1,0}}))$
- 30: **return**  $(priv, pub)$

---

### 3.5. GMSS Signature Generation

The GMSS signature generation is carried out in an *online* and an *offline* part. First, in the online part, the one-time signature of document  $d$  is computed and the GMSS signature is generated. Then, in the offline part, the private key is updated with the necessary values for the next signature. Due to the property of a changing private key, GMSS is called a *key evolving signature scheme*. In the course of the key update, authentication paths and one-time signatures required for upcoming signatures are precomputed partially. The offline part for the first signature was performed during the key pair generation process. For the  $n$ th GMSS signature ( $n \in \{0, \dots, 2^{h_0+\dots+h_{T-1}} - 1\}$ ), we want to indicate the leafs which are involved in the path  $p$ . In the following, the values  $l_i$  and  $j_i$  are corresponding to the current processed signature. The following equations recursively defines the indices  $j_i$  of the involved trees and the indices  $l_i$  of their leafs for each layer  $i$ :

$$\begin{aligned} j_{T-1} &= \lfloor n/2^{h_{T-1}} \rfloor, & l_{T-1} &= n \bmod 2^{h_{T-1}}, \\ j_i &= \lfloor j_{i+1}/2^{h_i} \rfloor, & l_i &= j_{i+1} \bmod 2^{h_i}, i = 1, \dots, T-1 \end{aligned}$$

The path  $p$  from the lowest tree  $\mathcal{T}_{T-1, j_{T-1}}$  to the top tree  $\mathcal{T}_{0,0}$  used to authenticate the  $n$ th signature is given as  $(\mathcal{T}_{T-1, j_{T-1}}, \mathcal{T}_{T-2, j_{T-2}}, \dots, \mathcal{T}_{1, j_1}, \mathcal{T}_{0,0})$ . For every layer  $i = 1, \dots, T-1$ , the one-time signature  $\text{SIG}_{\mathcal{T}_{i, j_i}}$  of the root of tree  $\mathcal{T}_{i, j_i}$  is generated using leaf  $l_{i-1}$  of tree  $\mathcal{T}_{i-1, j_{i-1}}$ .  $\text{SIG}_{\mathcal{T}_{i, j_i}}$ ,  $i = 1, \dots, T-1$  as well as the  $\text{AUTH}_{\mathcal{T}_{i, j_i}, l_i}$  of each leaf  $l_i$  of path  $p$ , were computed during the offline part of the previous signature or the key pair generation, and thus are already available as part of the GMSS private key.

#### 3.5.1. Online Part

During the online part of the GMSS signature generation the signature  $\text{SIG}_{\mathcal{T}_d}$  of the message digest  $d$  is calculated using leaf  $l_{T-1}$  of tree  $\mathcal{T}_{T-1, j_{T-1}}$  as one-time signature key. Then the GMSS signature is generated. It consists of

1. The leafs  $l_i, i = 0, \dots, T-1$
2. The one-time signature  $\text{SIG}_d$
3. The one-time signatures  $\text{SIG}_{\mathcal{T}_{i, j_i}}, i = 1, \dots, T-1$
4. The authentication paths  $\text{AUTH}_{\mathcal{T}_{i, j_i}, l_i}, i = 0, \dots, T-1$

See Appendix A for a detailed specification of the GMSS signature in Abstract Syntax Notation number one (ASN.1). Algorithm 7 describes the online part of the GMSS signature generation. In line 6 the offline part is carried out.

---

**Algorithm 7** GMSS Signature Generation

---

**Input:** document  $d$ , GMSS private key  $priv$ **Output:** signature  $sig$  of  $d$ , updated private key  $priv$ , or **STOP** if no more signatures can be generated

- 1: **if**  $2^{h_0+\dots+h_{T-1}}$  signatures have already been generated **then STOP**
  - 2: obtain an OTSS signature key:  $(X) \leftarrow \text{Algorithm 1}(\text{SEED}_{\mathcal{T}_{T-1,j},l_{T-1}})$
  - 3: compute the one-time signature of  $d$ :  $\sigma \leftarrow \text{Algorithm 3}(d, X)$
  - 4: set  $sig = ((l_0, \dots, l_{T-1}), \sigma, (\text{SIG}_{\mathcal{T}_{1,j_1}}, \dots, \text{SIG}_{\mathcal{T}_{T-1,j_{T-1}}}), (\text{AUTH}_{\mathcal{T}_{0,j_0},l_0} \dots, \text{AUTH}_{\mathcal{T}_{T-1,j_{T-1}},l_{T-1}}))$
  - 5: update the private key:  $priv \leftarrow \text{Algorithm 12}(T-1, priv)$
  - 6: **return** the GMSS signature  $sig$  of  $d$  and the updated private key  $priv$
- 

### 3.5.2. Offline Part

The basic task of the offline part of the GMSS signature generation is to calculate the new authentication path for the upcoming leaf. If the currently used tree is depleted the next one has to be generated and the authentication path of its first leaf is calculated. Additionally, to ensure a continuous path  $p$ , the root of the new tree has to be signed by using the next leaf of the tree one layer above. Therefore the same procedure has to be applied to the layer above. In the worst case this recurs up to the top layer.

Note that the higher the layer the less frequently changes of the tree are necessary. This observation enables us to distribute the costs of prospective calculations by precalculating them partially when advancing on lower layers.

#### Authentication Path Generation

The generation of the new authentication path for the upcoming leaf  $\text{AUTH}_{\mathcal{T}_{i,j_i},l_{i+1}}$  is performed with an algorithm of Szydło. Input to this algorithm are the authentication path  $\text{AUTH}_{\mathcal{T}_{i,j_i},l_i}$  and the  $\text{SEED}_{\mathcal{T}_{i,j_i},l_i}$  of the current leaf and an algorithm stack. Output are  $\text{AUTH}_{\mathcal{T}_{i,j_i},l_{i+1}}$  and the updated stack. The higher a node's position in the authentication path, the less frequently it will change. The Szydło Algorithm takes advantage of this fact, and only computes the changing values. Additionally, the stack is used to remember old values that will be required later again. This turns the algorithm of Szydło to an efficient method for the calculation of the authentication path. A detailed description of the algorithm can be found here [18].

#### Distributed Signature Generation

When advancing from leaf  $2^{h_i}$  of tree  $\mathcal{T}_{i,j_i}$  to leaf 0 of tree  $\mathcal{T}_{i,j_{i+1}}$ , the signature of the root of the next tree  $\text{SIG}_{\mathcal{T}_{i,j_{i+1}}}$  has to be ready because tree  $\mathcal{T}_{i,j_{i+1}}$  will be used by the next GMSS signature.  $\text{SIG}_{\mathcal{T}_{i,j_{i+1}}}$  is generated using the one-time signature key that

### 3. GMSS – Generalized Merkle Signature Scheme

corresponds to either leaf  $l_{i-1} + 1$  of tree  $\mathcal{T}_{i-1, j_{i-1}}$  or leaf 0 of tree  $\mathcal{T}_{i-1, j_{i-1}+1}$ . The latter case occurs if the tree  $\mathcal{T}_{i-1, j_{i-1}}$  in the layer  $(i - 1)$  is depleted. In this case, we have to advance to the next tree in this layer  $\mathcal{T}_{i-1, j_{i-1}+1}$ , too. We assume that  $\text{ROOT}_{\mathcal{T}_{i, j_i+1}}$  is available when tree  $\mathcal{T}_{i, j_i}$  is used for the first time, i.e. signature generation with  $l_i = 0$ . The generation of  $\text{SIG}_{\mathcal{T}_{i, j_i+1}}$  is distributed evenly over the  $2^{h_i}$  leaves of tree  $\mathcal{T}_{i, j_i}$ . Thus  $\text{SIG}_{\mathcal{T}_{i, j_i+1}}$  is computed step by step when advancing in tree  $\mathcal{T}_{i, j_i}$  and will be ready when the tree is depleted ( $l_i = 2^{h_i}$ ).

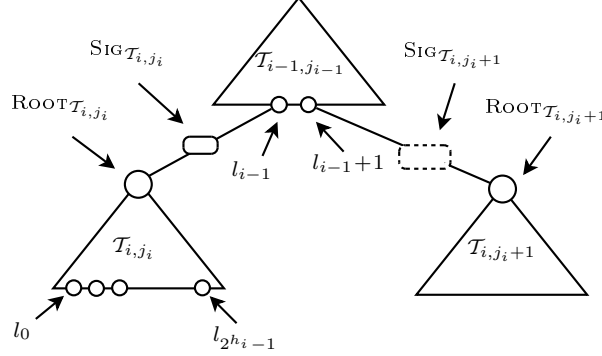


Figure 3.4.:  $\text{SIG}_{\mathcal{T}_{i, j_i+1}}$  is precomputed from  $\text{ROOT}_{\mathcal{T}_{i, j_i+1}}$  while using tree  $\mathcal{T}_{i, j_i}$

The first step, if  $l_i = 0$ , is to perform an initialization of the Winternitz one-time signature scheme, described in Algorithm 8.  $\text{ROOT}_{\mathcal{T}_{i, j_i+1}}$  and the initial one-time signature seed corresponding to leaf  $l_{i-1} + 1$  are passed to the initialization algorithm. Since the hash function calls are the time-critical elements, they have to be distributed evenly over the  $2^{h_i}$  steps of computation. The necessary number of hash function calls per step are calculated in the initialization algorithm by the following formula:

$$\lceil (t + \sum_{k=1}^t b_k) / 2^h \rceil$$

where  $b_k$  are the  $t$  blocks of the message and  $t = \lceil s/w \rceil + \lceil (\lceil \log_2 \lceil s/w \rceil \rceil + 1 + w) / w \rceil$ , assuming that  $s$  is the output size of the used hash function  $H : \{0, 1\}^* \rightarrow \{0, 1\}^s$ .

The first  $t$  hash function calls are needed for the generation of the  $t$  one-time signature keys  $(x_1, \dots, x_t)$ . The sum of  $b_k$  yields the number of hash function calls needed during the signing process. Remember, each part of the one-time signature key  $x_k$  needs  $b_k$  calls of the hash function. To memorize the intermediate values, GMSS uses a root-signature object *sigstate*, that represents the actual status of the partially computed signature. This object can be initialized or updated. *sigstate* contains the index  $k$ , the necessary steps *steps*, the message blocks  $(b_1, \dots, b_t)$ , the internal seed  $\text{SEED}_{\text{OTS}}$  and the values  $(x_1, \dots, x_t)$ .

---

**Algorithm 8** Distributed Root Signature Initialization

---

**Input:** document  $d$ , initial seed SEED<sub>OTS</sub>

- 1: compute the  $s$  bit hash value  $H(d)$  of document  $d$ .
  - 2: split the binary representation of  $H(d)$  into  $\lceil s/w \rceil$  blocks  $b_1, \dots, b_{\lceil s/w \rceil}$  of length  $w$ , padding  $H(d)$  with zeros from the left if required.
  - 3: treat  $b_k$  as the integer encoded by the respective block and compute the checksum
 
$$C = \sum_{k=1}^{\lceil s/w \rceil} 2^w - b_k.$$
  - 4: split the binary representation of  $C$  into  $\lceil (\lceil \log_2 \lceil s/w \rceil \rceil + 1 + w)/w \rceil$  blocks  $b_{\lceil s/w \rceil + 1}, \dots, b_t$  of length  $w$ , padding  $C$  with zeros from the left if required.
  - 5: treat  $b_k$  as the integer encoded by the respective block and compute the necessary number of hash function calls
 
$$steps \leftarrow \lceil (t + \sum_{k=1}^t b_k) / 2^h \rceil.$$
  - 6: initialize algorithm state variable  $k = 0$
- 

Each time we advance one leaf in tree  $\mathcal{T}_{i,j_i}$  we compute the next part of  $\text{SIG}_{\mathcal{T}_{i,j_i+1}}$  by performing an update to the corresponding leaf object  $sigstate_i$ . Algorithm 9 describes this update step. Each time the update is called,  $steps$  hash function calls are performed depending on where they are needed. This can either be for the generation of the next part of the one-time signature key or for the calculation of a part of the signature. If  $l_i = 2^{h_i} - 1$  the signature generation is completed.  $x = (x_1, \dots, x_t)$  now is the completely computed signature of the root  $\text{SIG}_{\mathcal{T}_{i,j_i+1}}$ .

---

**Algorithm 9** Distributed Root Signature Update

---

- 1: **for**  $1, \dots, steps$  **do**
  - 2:     **if**  $b_k = 0 \vee k = 0$  **then**
  - 3:          $k = k + 1$
  - 4:         compute signature key part  $x_k : (\text{SEED}_{\text{OTS}}, x_k) \leftarrow \text{PRNG}(\text{SEED}_{\text{OTS}})$
  - 5:     **else if**  $k < t \vee b_t > 0$  **then**
  - 6:          $b_k = b_k - 1$
  - 7:         compute the hash value of  $x_k : x_k \leftarrow H(x_k)$
- 

**Distributed Root Generation**

Above, we assumed that  $\text{ROOT}_{\mathcal{T}_{i,j_i+1}}$  is available when we first use tree  $\mathcal{T}_{i,j_i}$ . This certainly holds if  $j_i = 0$ , since  $\text{ROOT}_{\mathcal{T}_{i,1}}$  was computed during the key generation but for  $j_i > 0$  we must precompute  $\text{ROOT}_{\mathcal{T}_{i,j_i+2}}$  while proceeding in tree  $\mathcal{T}_{i,j_i}$ . The root must be ready when we switch to tree  $\mathcal{T}_{i,j_i+1}$  and want to start the generation of  $\text{SIG}_{\mathcal{T}_{i,j_i+2}}$ . The distributed computation of  $\text{ROOT}_{\mathcal{T}_{i,j_i+2}}$  is realized by successively

### 3. GMSS – Generalized Merkle Signature Scheme

computing the leafs of tree  $\mathcal{T}_{i,j_i+2}$  and passing them to the modified treehash algorithm introduced above (Algorithm 5). While using leaf  $l_i$  of tree  $\mathcal{T}_{i,j_i}$ , we compute leaf  $l_i$  of tree  $\mathcal{T}_{i,j_i+2}$  and pass it to treehash. Since, treehash must be applied  $2^{h_i}$  times to finish the construction of the root,  $\text{ROOT}_{\mathcal{T}_{i,j_i+2}}$  will be ready when switching from tree  $\mathcal{T}_{i,j_i}$  to  $\mathcal{T}_{i,j_i+1}$  after processing the  $2^{h_i}$  leafs. Treehash additionally computes  $\text{AUTH}_{\mathcal{T}_{i,j_i+2},0}$ . Both, the intermediate stack values for the root as well as the intermediate authentication path values are stored and updated in the GMSS private key each step. The computation of leaf 0 of tree  $\mathcal{T}_{i,j_i+2}$  requires the seed  $\text{SEED}_{\mathcal{T}_{i,j_i+2},0}$  which is already part of the GMSS private key, because it was obtained during the generation of  $\text{ROOT}_{\mathcal{T}_{i,j_i+1}}$ .

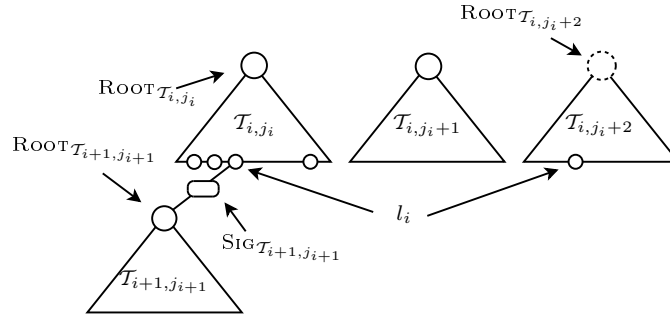


Figure 3.5.:  $\text{ROOT}_{\mathcal{T}_{i,j_i+2}}$  of tree  $\mathcal{T}_{i,j_i+2}$  is precomputed while advancing in tree  $\mathcal{T}_{i,j_i}$

#### Distributed Leaf Generation

The distributed computation of the roots  $\text{ROOT}_{\mathcal{T}_{i,j_i+2}}$  necessitates the computation of a new leaf of tree  $\mathcal{T}_{i,j_i+2}$  each step. Leaf  $l_{T-1}$  of a tree  $\mathcal{T}_{T-1,j_{T-1}+2}$  in the lowest layer has to be computed at once, when advancing in tree  $\mathcal{T}_{T-1,j_{T-1}}$ , but the computation of a leaf of a tree  $\mathcal{T}_{i,j_i+2}$  in a higher layer ( $i < T - 1$ ) can again be distributed. We compute leaf  $l_i$  of tree  $\mathcal{T}_{i,j_i+2}$ , while using leaf  $l_i$  of tree  $\mathcal{T}_{i,j_i}$ , or respectively while processing the  $2^{h_{i+1}}$  leafs of the underlying tree  $\mathcal{T}_{i+1,j_{i+1}}$ .

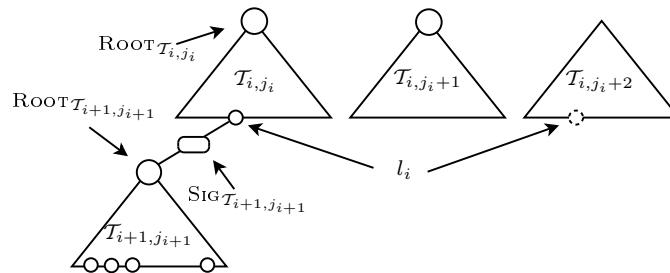


Figure 3.6.: Leaf  $l_i$  of tree  $\mathcal{T}_{i,j_i+2}$  is precomputed while using tree  $\mathcal{T}_{i+1,j_{i+1}}$

Generating a leaf means to compute the corresponding one-time verification key  $Y$ . Thus the Winternitz OTSS key pair generation has to be divided in  $2^{h_{i+1}}$  steps. Compared with the distributed root signature this is easier, since the number of hash function calls are constant and do not depend on a variable message. The first step of the distributed leaf calculation, if  $l_{i+1} = 0$ , is to perform an initialization wherein the necessary hash function calls per step are calculated and the initial seed is passed and the algorithm indices  $k, g$  are reseted. This is described in Algorithm 10.

The necessary number of hash function calls per step are calculated in the initialization algorithm by the following formula:

$$\lceil (2^w - 1 * t + t + 1) / 2^h \rceil$$

where  $t = \lceil s/w \rceil + \lceil (\lceil \log_2 \lceil s/w \rceil \rceil + 1 + w) / w \rceil$ , assuming that  $s$  is the output size of the used hash function  $H : \{0, 1\}^* \rightarrow \{0, 1\}^s$ . The generation of the one-time signature key  $X = (x_1, \dots, x_t)$  costs  $t$  hash function calls, the computation of the one-time verification key costs  $2^w - 1$  calls of the hash function for each of the  $t$  signature key parts and one last call yields the leaf value  $Y$ , the hash value of the concatenation of the  $t$  verification key parts.

Again an object, in this case a leaf-object *leafstate*, is used to represent the actual status of the calculation. *leafstate* contains the internal indices  $k$  and  $g$ , the necessary steps *steps* the internal seed  $\text{SEED}_{\text{OTS}}$ , the values  $(x_1, \dots, x_t)$  and the value of the leaf  $\text{LEAF}$ .

---

**Algorithm 10** Distributed Leaf Calculation Initialization
 

---

**Input:** the *seed* corresponding to the leaf

- 1: compute initial seed for one-time signature key:  $(\_, \text{SEED}_{\text{OTS}}) \leftarrow f(\text{seed})$
  - 2: calculate necessary steps:  $\text{steps} \leftarrow \lceil (2^w - 1 * t + t + 1) / 2^h \rceil$
  - 3: initialize  $k = 0, g = 0$  **return** *state*
- 

Each time we advance one leaf in tree  $\mathcal{T}_{i+1, j_{i+1}}$  we compute the next part of leaf  $l_i$  of tree  $\mathcal{T}_{i, j_{i+2}}$  by performing an update to the corresponding leaf-object *leafstate* <sub>$i$</sub> . Algorithm 11 describes this update step. Each time the update is called, *steps* hash function calls are performed depending on where they are needed. This can either be for the generation of the next part of the one-time signature key or for the calculation of the  $2^w - 1$  hash function calls needed to compute a verification key part. If  $l_{i+1} = 2^{h_{i+1}} - 1$  the signature generation is completed.  $\text{LEAF}$  in *leafstate* <sub>$i$</sub>  now contains the completely computed leaf  $l_i$  of tree  $\mathcal{T}_{i, j_{i+2}}$ . Thus the leaf is finished just when its needed by the next step of the distributed root calculation of tree  $\mathcal{T}_{i, j_{i+2}}$ .

### 3. GMSS – Generalized Merkle Signature Scheme

---

#### Algorithm 11 Distributed Leaf Calculation Update

---

```

1: for  $1, \dots, steps$  do
2:   if  $g = 2^w - 1 \wedge k = t$  then
3:     compute the hash value of LEAF :  $LEAF \rightarrow H(x_1 || \dots || x_t)$ 
4:   else if  $g = 2^w - 1 \vee k = 0$  then
5:      $k = k + 1$ 
6:      $g = 0$ 
7:     compute  $(SEED_{OTS}, x_k) \leftarrow PRNG(SEED_{OTS})$ 
8:   else
9:      $g = g + 1$ 
10:    compute the hash value of  $x_k$  :  $x_k \leftarrow H(x_k)$ 

```

---

#### Private Key Update

Algorithm 12 describes the entire offline part of the GMSS signature generation. The initial call of Algorithm 12 occurs when advancing from  $l_{T-1}$  to  $l_{T-1} + 1$  in the lowest layer  $T - 1$ . Algorithm 12 then basically works according to the following recursive scheme:

```

1: if  $\mathcal{T}_{i,j_i}$  in layer  $i$  is depleted then
2:   proceed to the next tree in this layer  $\mathcal{T}_{i,j_i+1}$ 
3:   recursively apply the algorithm to layer  $i - 1$ 
4: else
5:   calculate necessary values for the succeeding leaf  $l_i + 1$ 

```

Due to its recursivity, Algorithm 12 expects the current layer index  $i$  as argument. For the first call, after the online part of the signature generation,  $i = T - 1$  is passed. In the case that for each layer  $i = T - 1, \dots, 1$  a new tree has to be generated, the algorithm will be recursively recall itself  $T$  times, in the final step with  $i = 0$  as argument. The *if*-clause in line 1 divides the algorithm in two parts.

The first part from line 2 to line 15 is executed if we simply advance to the next leaf and the necessary computation and precalculation steps have to be performed. In this part, first of all, the authentication path for the upcoming leaf  $AUTH_{\mathcal{T}_{i,j_i},l_i+1}$  is generated by applying the Szydło Algorithm. After that, the next part of the distributed signature and leaf computation may be performed depending on the layer  $i$  which is currently processed. For the next step of the distributed generation of  $ROOT_{\mathcal{T}_{i,j_i+2}}$  the next leaf of tree  $\mathcal{T}_{i,j_i+2}$  is required. For layer  $i = T - 1$ , the leaf value must be computed at once, but for the layers  $i < T - 1$  the leaf can be obtained from *leafstate* <sub>$i$</sub>  which is ready at this time. The new values ( $AUTH_{\mathcal{T}_{i,j_i},l_i+1}$ ,  $SEED_{\mathcal{T}_{i,j_i+2},l_i+1}$ ,  $AUTH_{\mathcal{T}_{i,j_i+2},0}$ , *stackR* <sub>$i$</sub> , *leafstate* <sub>$i$</sub>  and *sigstate* <sub>$i$</sub> ) are updated to the GMSS private key and replace their former counterparts. Finally the algorithm returns the updated GMSS private key.

### 3.5. GMSS Signature Generation

The second part, the *else*-case (line 16 - 23), is executed if a new tree has to be generated. In this part the new seed  $\text{SEED}_{\mathcal{T}_{i,j_i+1},0}$  is calculated and the last steps of the distributed calculation of  $\text{ROOT}_{\mathcal{T}_{i,j_i+2}}$  and  $\text{AUTH}_{\mathcal{T}_{i,j_i+2},0}$ , of  $\text{SIG}_{\mathcal{T}_{i,j_i+1}}$  and of leaf  $l_{i-1}$  of tree  $\mathcal{T}_{i-1,j+2}$  are performed. The distributed signature computation has been finished and *statesig<sub>i</sub>* provides  $\text{SIG}_{\mathcal{T}_{i,j_i+1}}$ . After that, the distributed signature computation for the upcoming root  $\text{ROOT}_{\mathcal{T}_{i,j_i+2}}$  is initialized. The new values  $\text{SEED}_{\mathcal{T}_{i,j_i+1},0}$ ,  $\text{ROOT}_{\mathcal{T}_{i,j_i+2}}$ ,  $\text{AUTH}_{\mathcal{T}_{i,j_i+2},0}$ ,  $\text{SIG}_{\mathcal{T}_{i,j_i+1}}$ , the updated object *leafstate<sub>i</sub>* and the new initialized *sigstate<sub>i</sub>* are updated to the GMSS private key, and replace their former counterparts. Finally Algorithm 12 is applied recursively to the layer above because switching to a new tree on layer  $i$  necessitates switching to the next leaf on layer  $i - 1$ .

As we described above, the new values and intermediate precalculation states are stored to the GMSS private key after each step. The former values in the key are no more needed and thus are replaced. In doing so, the GMSS private key always keeps a constant size, it permanently consists of the following parts:

$l_i$	the current leaf of layer $i = 0, \dots, T - 1$
$\text{SEED}_{\mathcal{T}_{i,j_i},l_i}$ ,	the seed of the first tree of layer, $i = 0, \dots, T - 1$
$\text{AUTH}_{\mathcal{T}_{i,j_i},l_i}$ ,	the authentication path of the first tree of layer, $i = 0, \dots, T - 1$
$\text{SIG}_{\mathcal{T}_{i,j_i}}$ ,	the signatures of the roots of the first tree of layer, $i = 1, \dots, T - 1$
$\text{AUTH}_{\mathcal{T}_{i,j_i+1},0}$ ,	the authentication path of the second tree of layer, $i = 1, \dots, T - 1$
$\text{ROOT}_{\mathcal{T}_{i,j_i+1}}$ ,	the root of the second tree of layer, $i = 1, \dots, T - 1$
<i>sigstate<sub>i</sub></i> ,	the signature object for $\text{SIG}_{\mathcal{T}_{i,j_i+1}}$ , $i = 1, \dots, T - 1$
$\text{SEED}_{\mathcal{T}_{i,j_i+2},l_i}$ ,	the seed of the third tree of layer, $i = 1, \dots, T - 1$
$\text{AUTH}_{\mathcal{T}_{i,j_i+2},0}$ ,	the partial authentication path of $\mathcal{T}_{i,j_i+2}$ , $i = 1, \dots, T - 1$
<i>leafstate<sub>i</sub></i> ,	the leaf object for $l_i$ of $\mathcal{T}_{i,j_i+2}$ , $i = 1, \dots, T - 2$
<i>stackR<sub>i</sub></i> ,	the stack for the generation of $\text{ROOT}_{\mathcal{T}_{i,j_i+2}}$ , $i = 1, \dots, T - 1$
<i>stackA<sub>i</sub></i> ,	the stacks used by the Szydlo Algorithm, $i = 1, \dots, T - 2$

### 3. GMSS – Generalized Merkle Signature Scheme

---

#### Algorithm 12 GMSS Update of Private Key

---

**Input:** processed layer  $i$ , GMSS private key  $priv$

**Output:** updated private key  $priv$

- 1: **if**  $l_i < 2^{h_i}$  **then**
  - 2:     compute the next authentication path for the current tree,  
        $(AUTH_{\mathcal{T}_{i,j_i},l_{i+1}}, stackA_i) \leftarrow \text{Szydlo.auth}(AUTH_{\mathcal{T}_{i,j_i},l_i}, SEED_{\mathcal{T}_{i,j_i},l_i}, stackA_i)$   
       and replace  $AUTH_{\mathcal{T}_{i,j_i},l_i}$  in  $priv$  by  $AUTH_{\mathcal{T}_{i,j_i},l_{i+1}}$
  - 3:     **if**  $i > 0$  **then**
  - 4:         partial calculation of the signature of  $ROOT_{\mathcal{T}_{i,j_{i+1}}}$ :  
        update  $(sigstate_i)$ .Algorithm 9 in  $priv$  .
  - 5:         **if**  $i > 1$  **then**
  - 6:             partial calculation of leaf  $l_{i-1}$  of tree  $\mathcal{T}_{i-1,j+2}$ ,  
            update  $(leafstate_{i-1})$ .Algorithm 11 in  $priv$ .
  - 7:             obtain the next leaf of tree  $\mathcal{T}_{i,j+2}$ :
  - 8:             compute  $(SEED_{\mathcal{T}_{i,j+2},l_{i+1}}, SEED_{OTS}) \leftarrow f((SEED_{\mathcal{T}_{i,j+2},l_i}),$   
            and replace  $SEED_{\mathcal{T}_{i,j+2},l_i}$  in  $priv$  by  $SEED_{\mathcal{T}_{i,j+2},l_{i+1}}$
  - 9:             **if**  $i = T - 1$  **then**
  - 10:                  $((X, Y)) \leftarrow \text{Algorithm 2}(SEED_{OTS})$
  - 11:             **else**
  - 12:                 set  $Y =$  the ready leaf value in  $leafstate_i$
  - 13:                 initialize  $leafstate_i$  for next partial leaf generation:  
                 $(leafstate_i)$ .Algorithm 10( $SEED_{\mathcal{T}_{i,j+2},l_{i+1}}$ )
  - 14:             partial calculation of the root of tree  $\mathcal{T}_{i,j+2}$ :  
             $(stackR_i, AUTH_{\mathcal{T}_{i,j+2},0}) \leftarrow \text{Algorithm 5}(Y, stackR_i, AUTH_{\mathcal{T}_{i,j+2},0})$   
            and update  $stackR_i$  and  $AUTH_{\mathcal{T}_{i,j+2},0}$  in  $priv$
  - 15:         set  $l_i = l_i + 1$
  - 16:     **else if** the tree is not already the last one on this layer **then**
  - 17:         calculate seed:  $(SEED_{\mathcal{T}_{i,j_{i+1}},0}, \_ ) \leftarrow f(SEED_{\mathcal{T}_{i,j_i},l_i}),$   
        and replace  $SEED_{\mathcal{T}_{i,j_i},0}$  in  $priv$  by  $SEED_{\mathcal{T}_{i,j_{i+1}},0}$
  - 18:         last step of partial calculation of the signature of  $ROOT_{\mathcal{T}_{i,j+1}}$ :  
         $(sigstate_i)$ .Algorithm 9  
        and let  $SIG_{\mathcal{T}_{i,j_{i+1}}}$  be the actual signature value in  $sigstate_i$ ,  
        replace  $SIG_{\mathcal{T}_{i,j_i}}$  in  $priv$  by  $SIG_{\mathcal{T}_{i,j_{i+1}}}$
  - 19:         last step of partial calculation of leaf  $l_{i-1}$  of tree  $\mathcal{T}_{i-1,j+2}$ ,  
        update  $(leafstate_{i-1})$ .Algorithm 11 in  $priv$
  - 20:         perform last step of distributed root and authpath generation  
         $(stackR_i, AUTH_{\mathcal{T}_{i,j+2},0}) \leftarrow \text{Algorithm 9}(stackR_i),$   
        and let  $ROOT_{\mathcal{T}_{i,j+2}}$  be the single value in  $stackR_i$ ,  
        replace  $ROOT_{\mathcal{T}_{i,j+1}}$  in  $priv$  by  $ROOT_{\mathcal{T}_{i,j+2}}$  and  $AUTH_{\mathcal{T}_{i,j_i},0}$  by  $AUTH_{\mathcal{T}_{i,j+2},0}$
  - 21:         initialize partial signature generation:  
         $(SEED_{\mathcal{T}_{i-1,j_{i-1}},l_{i-1}+3}, SEED_{OTS}) \leftarrow f(SEED_{\mathcal{T}_{i-1,j_{i-1}},l_{i-1}+2})$      update  
         $(sigstate_i) \leftarrow \text{Algorithm 8}(ROOT_{\mathcal{T}_{i,j+2}}, SEED_{OTS})$  in  $priv$ ,
  - 22:         set  $l_i = 0$
  - 23:         recursively recall Algorithm 12 with  $i = i - 1$  and updated  $priv$
  - 24: **return** the updated private key  $priv$
-

### 3.6. GMSS Signature Verification

In this section we describe how a GMSS signature  $sig$  of a message  $m$  can be verified. The GMSS verification algorithm, described in Algorithm 13, is based on the MSS verification procedure (see section 2.3). Remember, the signature  $sig$  contains:

- The current processed leafs of each tree  $l_i, i = 0, \dots, T - 1$
- The one-time signature  $SIG_d$
- The one-time signatures  $SIG_{\mathcal{T}_{i,j_i}}, i = 1, \dots, T - 1$
- The authentication paths  $AUTH_{\mathcal{T}_{i,j_i},l_i}, i = 0, \dots, T - 1$

The verifier knows the GMSS public key ( $ROOT_{\mathcal{T}_{0,0}}, \mathcal{P}$ ) of the alleged signer. At first, he computes the digest  $d$  by applying the prearranged hash function  $H$  to message  $m$ ,  $d \leftarrow H(m)$ . Then, he verifies the one-time signature  $SIG_d$  of the digest  $d$  using the Winternitz OTS verification algorithm, Algorithm 4. The necessary value of  $w_T$  is part of the known parameter set  $\mathcal{P}$ . Algorithm 4 yields the one-time verification key of  $SIG_d$ , which actually is leaf  $l_{T-1}$  of tree  $\mathcal{T}_{T-1,j_{T-1}}$ . Then the verifier uses leaf  $l_{T-1}$  and  $AUTH_{\mathcal{T}_{T-1,j_{T-1}},l_{T-1}}$  to construct  $ROOT_{\mathcal{T}_{T-1,j_{T-1}}}$ . This is done according to the MSS Signature Verification procedure (see section 2.3). The verifier repeats the following steps for  $i = T - 2, \dots, 0$ .

- (1) use  $ROOT_{\mathcal{T}_{i+1,j_{i+1}}}$  and verify  $SIG_{\mathcal{T}_{i+1,j_{i+1}}}$  to obtain  $l_i$ .
- (2) use  $l_i$  and  $AUTH_{\mathcal{T}_{i,j_i},l_i}$  to compute  $ROOT_{\mathcal{T}_{i,j_i}}$

In doing so,  $ROOT_{\mathcal{T}_{0,0}}$  successively is computed. Finally, the signature is only accepted if the such computed root matches the root of the signer's public key.

---

#### Algorithm 13 GMSS Signature Verification

---

**Input:** document  $d$ , GMSS signature  $sig = (l_i, SIG_d, SIG_{\mathcal{T}_{i,j_i}}, AUTH_{\mathcal{T}_{i,j_i},l_i})$ , GMSS public key  $R$

**Output:** TRUE if the signature is valid, FALSE otherwise.

- 1: **for**  $i = T, \dots, 0$  **do**
  - 2:     obtain the one-time verification key  $Y$ :  
 $Y_i \leftarrow \text{Algorithm 4}(ROOT_{\mathcal{T}_{i,j_i}}, SIG_{\mathcal{T}_{i-1,j_{i-1}}})$ ,  
where  $SIG_d$  is substituted for  $SIG_{\mathcal{T}_{i,j_i}}$  and  $d$  is substituted for  $ROOT_{\mathcal{T}_{i,j_i}}$
  - 3:     use  $Y_i$  and  $AUTH_{\mathcal{T}_{i,j_i},l_i}$  to compute  $ROOT_{\mathcal{T}_{i,j_i}}$  as in the case of  
MSS signature verification .
  - 4: **if**  $ROOT_{\mathcal{T}_{0,j_0}}$  is equal to the GMSS public key  $R$  **then return** TRUE
  - 5: **else return** FALSE
-

### 3. *GMSS – Generalized Merkle Signature Scheme*

## 4. Implementation and Efficiency

This thesis included the implementation of GMSS using the Java programming language. In this chapter we first describe the general concept of the implementation and the data structures. Then we calculate the expected performance of GMSS by the means of cost functions, and finally we present effective timings of a test using our GMSS Java implementation

### 4.1. Java Implementation of GMSS

The idea of our implementation was to utilize the flexibility of the Java Cryptographic Architecture (JCA) [14]. JCA is a part of the Java security API. It provides the interfaces for the general cryptographic services of the Java Platform, such as digital signatures, message digest, key pair generators and key factories. We implemented GMSS as a module of the *FlexiProvider* [10] which is a Cryptographic Service Provider (JSP). The FlexiProvider is an open source toolkit for the Java Cryptographic Architecture and Java Cryptographic Extension (JCE) [15] which provides various cryptographic modules. The FlexiProvider in conjunction with JCA and JCE allows a dynamic exchange of cryptographic algorithms. Therefore GMSS can be easily plugged into applications that are built on the basis of the JCA and JCE. GMSS is implemented to be universal adaptable. For example the underlying message digest can be exchanged easily. Some predefined configuration can be comfortably accessed by the use of Object Identifiers (OID) within the FlexiProvider. In Appendix B the OIDs associated to GMSS are specified.

The implementation of GMSS can be divided in three components: key pair generation, signature generation and signature verification. The prearranged choice of the message digest and the cryptographic service provider are part of the certificate and have to be passed to the key pair generation and the signature generation component. The input of the GMSS key pair generation is the prearranged parameter set. The resulting private and public keys are encoded and stored using Abstract Syntax Notation One (ASN.1) [1]. This ensures interoperability between different applications and efficient generation of X.509 certificates. We decided to store the parameter set in both, the public and the private key to ensure the availability of the necessary values. The inputs of the signature generation are the encoded GMSS private key and the message we want to sign.

The GMSS signature is defined as a sequence of bytes. The integer value of  $l_i$  is converted to a four byte sequence. The one-time signature consists of  $t$   $s$ -bit hash values, and is represented by a byte sequence of length  $t * s/8$ . The authentication path  $\text{AUTH}_{\mathcal{T}_{i,j_i,l_i}}$  is represented by the node sequence  $(a_0, \dots, a_{h_i-1})$  in ascending

#### 4. Implementation and Efficiency

order. A node in turn is a  $s$ -bit hash value, which is represented by a  $s/8$ -byte sequence. The GMSS signature bytes are arranged in the following order:

$$\begin{pmatrix} (l_{T-1}, & \text{SIG}_d, & \text{AUTH}_{\mathcal{T}_{T-1,j_{T-1}},l_{T-1}}), \\ (l_{T-2}, & \text{SIG}_{\mathcal{T}_{T-1,j_{T-1}}}, & \text{AUTH}_{\mathcal{T}_{T-2,j_{T-2}},l_{T-2}}), \\ \vdots & & \\ (l_0, & \text{SIG}_{\mathcal{T}_{1,j_1}}, & \text{AUTH}_{\mathcal{T}_{0,j_0},l_0}) \end{pmatrix}$$

The inputs of the verification procedure are the GMSS signature and the original message as inputs. A detailed specification of the GMSS Key Pair and the GMSS Signature using the Abstract Syntax Notation number One (ASN.1) can be found in Appendix A.

In Appendix C we give an example of the GMSS key pair generation, signing and verification procedure in the Java environment. The implementation of GMSS can be found as part of the PQC packages in the FlexiProvider which can be downloaded at [10].

#### 4.2. Cost Functions

In [3] the authors develop formulas for the estimated costs for the GMSS algorithms. The time critical values are the hash function  $H$  and the PRNG which basically includes one hash function call. The cost of a hash function call is denoted by  $c_{\text{HASH}}$  and the cost of a PRNG are denoted by  $c_{\text{PRNG}}$ . Thus the cost formulas are a function of  $c_{\text{HASH}}$  and  $c_{\text{PRNG}}$ . In the following we list the cost formulas of the several GMSS parts and additionally the memory requirements for the keys and the signature. A detailed proof and description can be found in [3]. The total cost for the key generation is

$$c_{\text{key gen}} = \sum_{i=1}^T c_{\text{tree}}(i) + \sum_{i=2}^T c_{\text{tree}}(i) \quad (4.1)$$

where  $c_{\text{tree}}(i) = (2^{h_i} (t_{w_i} (2^{w_i} - 1) + 1) + 2^{h_i} - 1) c_{\text{HASH}} + 2^{h_i} (t_{w_i} + 1) c_{\text{PRNG}}$ . The average cost for the online signing part is

$$c_{\text{online}} = (2^{w_T} - 1) t_{w_T} / 2 \cdot c_{\text{HASH}} + (t_{w_T} + 1) c_{\text{PRNG}}. \quad (4.2)$$

The offline signing costs consists of three parts. The distributed computation of a signature  $c_{\text{sig}}$ , a leaf  $c_{\text{leaf}}$  and a root  $c_{\text{root}}$  and the calculation of the authentication path  $c_{\text{auth}}$ . For the distributed signature, we require on average,

$$c_{\text{sig}}(i) = \left\lceil \frac{(2^{w_{i-1}} - 1) t_{w_{i-1}}}{2^{h_i + 1}} \right\rceil c_{\text{HASH}} + \left\lceil \frac{t_{w_{i-1}} + 1}{2^{h_i}} \right\rceil c_{\text{PRNG}} \quad (4.3)$$

operations each time we advance one leaf in  $\mathcal{T}_{i,j_i}$  to compute  $\text{SIG}_{\mathcal{T}_{i,j_i+1}}$ . The computation of the root requires

$$c_{\text{root}}(i) = h_i \cdot c_{\text{HASH}} \text{ (at most)} \quad (4.4)$$

operations each time we advance one leaf in  $\mathcal{T}_{i,j_i}$ , to compute  $\text{ROOT}_{\mathcal{T}_{i,j_i+2}}$ . The leaf computation requires

$$c_{\text{leaf}}(i) = \left\lceil \frac{(2^{w_i}-1)t_{w_i}+1}{2^{h_{i+1}}} \right\rceil c_{\text{HASH}} + \left\lceil \frac{t_{w_i}+1}{2^{h_{i+1}}} \right\rceil c_{\text{PRNG}} \quad (4.5)$$

operations each time we advance one leaf in  $\mathcal{T}_{i+1,j_{i+1}}$ . For the authentication path computation we require at most

$$c_{\text{auth}}(i) = h_i \cdot c_{\text{leaf}}(i) + \left\lceil \frac{2^{h_i-2}}{2^{h_{i+1}}} \right\rceil c_{\text{PRNG}} + h_i \cdot c_{\text{HASH}} \quad (4.6)$$

operations each time we advance one leaf in  $\mathcal{T}_{i,j_i}$  to compute  $\text{AUTH}_{\mathcal{T}_{i,j_i},l_{i+1}}$ . The combination of  $c_{\text{sig}}$ ,  $c_{\text{root}}$ ,  $c_{\text{leaf}}$  and  $c_{\text{auth}}$  yields the worst case costs for the offline part:

$$c_{\text{offline}} = \sum_{i=2}^T (c_{\text{sig}}(i) + c_{\text{root}}(i) + c_{\text{leaf}}(i)) + \sum_{i=1}^T (c_{\text{auth}}(i)) \quad (4.7)$$

The average cost for the verification is

$$c_{\text{verify}} = \sum_{i=1}^T ((2^{w_i}-1)t_{w_i}/2 + h_i) c_{\text{HASH}}. \quad (4.8)$$

The memory requirements for the keys are

$$\begin{aligned} m_{\text{pubkey}} &= s \text{ bits} \\ m_{\text{privkey}} &= \left( \sum_{i=1}^T (h_i + 1) + \sum_{i=2}^T (5h_i + 2t_{w_{i-1}} + 2) + 3h_i \right) s \text{ bits} \end{aligned} \quad (4.9)$$

The size of a signature is

$$m_{\text{signature}} = \sum_{i=1}^T (h_i + t_{w_i}) \cdot s \text{ bits}. \quad (4.10)$$

### 4.3. Timings

In this section, we present the practical timing analysis of our Java implementation of GMSS and compare it to the estimated values that are calculated on the basis of the cost functions of section 4.2. Therefor we use different parameter sets  $\mathcal{P}_k = (T, (h_1, \dots, h_T), (w_1, \dots, w_T))$ , which were developed in [3] and promise to provide the optimal results.

$$\begin{aligned} \mathcal{P}_{40} &= (2, (20, 20), (10, 5)) & \mathcal{P}_{80} &= (4, (20, 20, 20, 20), (8, 8, 8, 5)) \\ \mathcal{P}'_{40} &= (2, (20, 20), (9, 3)) & \mathcal{P}'_{80} &= (4, (20, 20, 20, 20), (7, 7, 7, 3)) \end{aligned}$$

The experiment includes both, the timings for a signature capacity of  $2^{40}$  and the timings for a signature capacity of  $2^{80}$ . In each case, two different configurations of the parameter  $w$  are used. One configuration contains higher values for  $w$  and

#### 4. Implementation and Efficiency

is expected to provide small signatures. The other one consists of small values for  $w$  and is expected to provide significantly faster key pair generation, signing and verification timings but at the expense of the signature size, i.e. larger signatures. In all parameter sets, the  $w$  of the lowest layer is smaller than the  $w$  of the layers above. This is expected to result in a better performance because the upper layers are less frequently processed than the lower layer. Thus the frequent calculations in the lowest layer can be processed faster.

Each parameter set was tested on two computers of different power. In the first place, on a fast Sun Fire X2200 M2 (AMD Opteron 2218 2.6GHz, 5 GByte RAM, Linux Ubuntu, Java 1.6.0-b105) and in the second place, on a Asus V6J (Intel Pentium Core-Duo T2400 1.83 GHz, 1 GByte RAM, Windows XP, Java 1.6.0\_01-b06). The test with the Asus V6J will provide more representative values while the Sun Fire X2200 M2 will show the potentials of a fast machine.

Table 4.1 summarizes the expected values for the Sun Fire X2200 M2. The values are calculated on the basis of the cost functions and the ratio  $c_{\text{HASH}}$  and  $c_{\text{PRNG}}$ . The hash function we use is SHA1. The costs for the hash and the PRNG is obtained by using a Java implementation of SHA1 and our Java implementation of the PRNG on the Sun Fire X2200 M2. This yields  $c_{\text{HASH}} = 0.000743$  and  $c_{\text{PRNG}} = 0.000926$ .

Table 4.1.: Expected timings and memory requirements on Sun Fire X2200 M2

	$m_{\text{public key}}$	$m_{\text{private key}}$	$m_{\text{signature}}$	$t_{\text{keygen}}$	$t_{\text{sign}}$	$t_{\text{verify}}$
$\mathcal{P}_{40}$	20 bytes	4800 bytes	1860 bytes	269 min	9.7 ms	7.3 ms
$\mathcal{P}'_{40}$	20 bytes	4880 bytes	2340 bytes	145 min	4.1 ms	4.0 ms
$\mathcal{P}_{80}$	20 bytes	10740 bytes	3620 bytes	396 min	9.8 ms	6.7 ms
$\mathcal{P}'_{80}$	20 bytes	12000 bytes	4240 bytes	221 min	4.2 ms	3.7 ms

Table 4.2 shows the effective values measured when running our GMSS implementation on the Sun Fire X2200 M2. To obtain a conclusive result for the average signing and verification times, a quantity of  $2^{21}$  signatures have been signed. This assures that trees in upper layers are processed within the test procedure.

Table 4.2.: Measured timings and memory requirements on Sun Fire X2200 M2

	$m_{\text{public key}}$	$m_{\text{private key}}$	$m_{\text{signature}}$	$t_{\text{keygen}}$	$t_{\text{sign}}$	$t_{\text{verify}}$
$\mathcal{P}_{40}$	67 bytes	5467 bytes	1868 bytes	228 min	9.6 ms	7.5 ms
$\mathcal{P}'_{40}$	67 bytes	5547 bytes	2348 bytes	125 min	5.1 ms	3.8 ms
$\mathcal{P}_{80}$	79 bytes	14251 bytes	3636 bytes	339 min	9.5 ms	6.1 ms
$\mathcal{P}'_{80}$	79 bytes	14731 bytes	4256 bytes	198 min	5.1 ms	2.8 ms

Table 4.3 summarizes the expected values for the Asus V6J. The costs for the hash

### 4.3. Timings

and the PRNG is obtained by using a Java implementation of SHA1 and our Java implementation of the PRNG on the Asus V6J.  $c_{\text{HASH}} = 0.0017$  and  $c_{\text{PRNG}} = 0.0021$ .

Table 4.3.: Expected timings and memory requirements on Asus V6J

	$m_{\text{public key}}$	$m_{\text{private key}}$	$m_{\text{signature}}$	$t_{\text{keygen}}$	$t_{\text{sign}}$	$t_{\text{verify}}$
$\mathcal{P}_{40}$	20 bytes	4800 bytes	1860 bytes	609 min	22.0 ms	16.5 ms
$\mathcal{P}'_{40}$	20 bytes	4880 bytes	2340 bytes	329 min	9.3 ms	9.0 ms
$\mathcal{P}_{80}$	20 bytes	10740 bytes	3620 bytes	897 min	22.1 ms	15.2 ms
$\mathcal{P}'_{80}$	20 bytes	12000 bytes	4240 bytes	500 min	9.4 ms	8.5 ms

Table 4.4 shows the effective values measured when running our GMSS implementation on the Asus V6J.

Table 4.4.: Measured timings and memory requirements on Asus V6J

	$m_{\text{public key}}$	$m_{\text{private key}}$	$m_{\text{signature}}$	$t_{\text{keygen}}$	$t_{\text{sign}}$	$t_{\text{verify}}$
$\mathcal{P}_{40}$	67 bytes	5467 bytes	1868 bytes	579 min	22.6 ms	19.4 ms
$\mathcal{P}'_{40}$	67 bytes	5547 bytes	2348 bytes	321 min	11.6 ms	10.6 ms
$\mathcal{P}_{80}$	79 bytes	14251 bytes	3636 bytes	868 min	22.6 ms	15.5 ms
$\mathcal{P}'_{80}$	79 bytes	14731 bytes	4256 bytes	498 min	11.6 ms	9.5 ms

The tables shows, that on the whole the estimated values are confirmed by the experiment and that the GMSS implementation offers competitive signing and verification times even with a total signature capacity of  $2^{80}$ . Furthermore, this result convincingly proves the flexibility of GMSS. The measured values show the expected characteristics. High values of  $w$  produce a small signature size, small  $w$  produce fast key generation, signing and verification times. The difference in the public key size is caused by the public key information which we additionally store in the keys, but key sizes are not crucial values and differences in this range do not affect the applicability of GMSS.

#### 4. *Implementation and Efficiency*

## 5. Conclusions

In this Thesis we present an efficient implementation of GMSS as a module of the FlexiProvider. Due to the possibility to choose different parameter sets  $\mathcal{P}$ , GMSS is very flexible and thus is adaptable to the requirements of different applications. Our implementation is based on the Java Cryptographic Service Provider and therefore can be easily plugged into any application that uses the JCA. GMSS allows a cryptographically unlimited signature capacity of  $2^{80}$  with competitive timings and memory requirements. The moderate key pair generation times can be disregarded for the most applications, because the high signature capacity requires no further key pair generations. This additionally makes GMSS invulnerable to denial of service attacks.

Not only the parameters  $w$  and  $h$  effect the performance of the implementation. During the phase of implementing GMSS, especially when Java specific problems occurred, we were frequently confronted with the trade off to either save memory or to make the computation faster. There still exist improvements which are expected to speed up the signature generation time, but which are not implemented yet. One approach for example is to compute a needed leaf of an interior tree, which is processed by the Szydlo algorithm, not at once, but by verifying the signature of the root of the underlying tree. Our GMSS implementation supports this improvement only within the key pair generation.

The variability of the computation costs of each individual signature is reduced by algorithms for distributed computation. Until now, our implementation does not support the distributed computation of the upcoming authentication path. The development of a distributed Szydlo Algorithm is expected to balance the computation timings of each signature once more.

The main intention of improving GMSS is to reduce the signature size. This is realized by using large Winternitz parameters. Since large  $w$ 's lead to more computational effort, any improvements of the performance of GMSS will also make smaller signatures possible.

All in all, GMSS already now provides competitive timings. Of special interest is the research of using GMSS on constraint devices, i.e. smartcards. Furthermore, GMSS does not rely on number theoretic assumptions and thus will not become insecure if a quantum computer will be build. This makes GMSS to an absorbing alternative to common signature schemes.

## 5. *Conclusions*

# Bibliography

- [1] Information technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation. ITU standard, International Telecommunications Union, 2002.
- [2] Johannes Buchmann, Carlos Coronado, Martin Döring, Daniela Engelbert, Christoph Ludwig, Raphael Overbeck, Arthur Schmidt, Ulrich Vollmer, and Ralf-Philipp Weinmann. Post-quantum signatures.
- [3] Johannes Buchmann, Erik Dahmen, Elena Klintsevich, Katsuyuki Okeya, and Camille Vuillaume. Merkle signatures with virtually unlimited signature capacity. In Jonathan Katz and Moti Yung, editors, *ACNS*, volume 4521 of *Lecture Notes in Computer Science*, pages 31–45. Springer, 2007.
- [4] Professor Dr. Johannes Buchmann. Einführung in die Kryptographie. Cryptology ePrint Archive, Report 2005/442, 2004. <http://eprint.iacr.org/>.
- [5] C. Dods, Nigel Smart, and Martijn Stam. Hash based digital signature schemes. In *Cryptography and Coding, Springer LNCS 3796*, pages 96–115. Springer Verlag, November 2005.
- [6] Chris Dods, Nigel Smart, and Martijn Stam. Hash based digital signature schemes. In *Proc. Cryptography and Coding*, volume 3796 of *Lecture Notes in Computer Science*, pages 96–115. Springer-Verlag, 2005.
- [7] Claudia Eckert. *IT-Sicherheit*. Oldenbourg, München [u.a.], 2004.
- [8] Digital signature standard (DSS). FIPS PUB 186-2, 2007. Available at <http://csrc.nist.gov/publications/fips/>.
- [9] Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Proceedings of CRYPTO 84 on Advances in cryptology*, pages 10–18, New York, NY, USA, 1985. Springer-Verlag New York, Inc.
- [10] The FlexiProvider group at Technische Universität Darmstadt. Flexi-Provider, an open source Java Cryptographic Service Provider. Available at <http://www.flexiprovider.de/>.
- [11] D. Johnson and A. Menezes. The elliptic curve digital signature algorithm (ecdsa, 1999).
- [12] Leslie Lamport. Constructing digital signatures from a one way function. Technical Report SRI-CSL-98, SRI International Computer Science Laboratory, 1979.

## Bibliography

- [13] Ralph C. Merkle. A certified digital signature. In *Proc. Advances in Cryptology (Crypto'89)*, volume 435 of *Lecture Notes in Computer Science*, pages 218–238. Springer-Verlag, 1989.
- [14] Sun Microsystems. The Java Cryptography Architecture (JCA), API Specification and Reference. Available at: <http://java.sun.com/j2se/1.4.2/docs/guide/security/CryptoSpec.html>.
- [15] Sun Microsystems. The Java Cryptography Extension (JCE) Reference Guide. Available at: <http://java.sun.com/j2se/1.4.2/docs/guide/security/JCERefGuide.html>.
- [16] Ron Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. volume 21, pages 120–126. ACM Press, 1978.
- [17] Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proc. 35th Annual Symposium on Foundations of Computer Science*, pages 124–134. IEEE Comput. Soc. Press, 1994.
- [18] Michael Szydło. Merkle tree traversal in log space and time. In *Proc. Advances in Cryptology (Eurocrypt'04)*, volume 3027 of *Lecture Notes in Computer Science*, pages 541–554. Springer-Verlag, 2004.

## A. ASN.1 Encoding

This section describes the specification of the GMSS public and private keys and the GMSS signature using Abstract Syntax Notation number One (ASN.1).[1]

```
GMSSPublicKey ::= SEQUENCE{
    publicKey      SEQUENCE OF OCTET STRING
    heightOfTrees  SEQUENCE OF INTEGER
    Parameterset   ParSet
}

GMSSPrivateKey ::= SEQUENCE {
    algorithm      OBJECT IDENTIFIER
    index          SEQUENCE OF INTEGER
    curSeeds       SEQUENCE OF OCTET STRING
    nextNextSeeds SEQUENCE OF OCTET STRING
    curAuth        SEQUENCE OF AuthPath
    nextAuth       SEQUENCE OF AuthPath
    nextNextAuth   SEQUENCE OF AuthPath
    nextRoot       SEQUENCE OF OCTET STRING
    curRootSig     SEQUENCE OF OCTET STRING
    StackKeep      SEQUENCE OF Stack
    StackNeed0     SEQUENCE OF Stack
    StackNeed1     SEQUENCE OF Stack
    nextNextLeaf   SEQUENCE OF DistrLeaf
    nextNextRoot   SEQUENCE OF OCTET STRING
    nextRootSig    SEQUENCE OF DistrRootSig
    Parameterset   ParSet
    names          SEQUENCE OF ASN1IA5String
}

GMSSSignature ::= SEQUENCE{
    index          SEQUENCE OF INTEGER
    authpath       SEQUENCE OF AuthPath
    signature      SEQUENCE OF OCTET STRING
}

AuthPath ::= SEQUENCE OF OCTET STRING
Stack ::= SEQUENCE OF OCTET STRING
```

## A. ASN.1 Encoding

```
DistrLeaf ::= SEQUENCE {  
    name      SEQUENCE OF ASN1IA5String  
    statBytes SEQUENCE OF OCTET STRING  
    statInts  SEQUENCE OF INTEGER  
}
```

```
DistrRootSig ::= SEQUENCE {  
    name      SEQUENCE OF ASN1IA5String  
    statBytes SEQUENCE OF OCTET STRING  
    statInts  SEQUENCE OF INTEGER  
}
```

```
ParSet ::= SEQUENCE {  
    T      INTEGER  
    h      SEQUENCE OF INTEGER  
    w      SEQUENCE OF INTEGER  
}
```

## B. Object Identifiers

This section lists the object identifiers (OIDs) assigned to the GMSS implementation. The main OID for GMSS as well as the OID for the GMSSKeyFactory is

1.3.6.1.4.1.8301.3.1.3.3

The OIDs for GMSS are summarized in the following table, where the column "Hash function" denotes the hash function used in the OTSS and the authentication trees.

Hash function	Object Identifier (OID)
SHA1	1.3.6.1.4.1.8301.3.1.3.3.1
SHA224	1.3.6.1.4.1.8301.3.1.3.3.2
SHA256	1.3.6.1.4.1.8301.3.1.3.3.3
SHA384	1.3.6.1.4.1.8301.3.1.3.3.4
SHA512	1.3.6.1.4.1.8301.3.1.3.3.5

Table B.1.: OIDs assigned to GMSS

*B. Object Identifiers*

## C. Java Examples

### Key Pair Generation

The GMSS KeyPairGenerator can be used as follows:

1. Get instance of GMSS key pair generator:  

```
KeyPairGenerator kpg = KeyPairGenerator.getInstance("GMSSwithSHA1",  
"FlexiPQC");
```
2. Initialize the KPG with the desired Parameterset  

```
kpg.initialize(parameterset);
```
3. Create GMSS key pair:  

```
KeyPair keyPair = kpg.generateKeyPair();
```
4. Get the encoded private and public keys from the key pair:  

```
encodedPublicKey = keyPair.getPublic().getEncoded();  
encodedPrivateKey = keyPair.getPrivate().getEncoded();
```

### Signature generation

The GMSS Signature generation works as follows:

1. Generate KeySpec from encoded GMSS private key:  

```
KeySpec privateKeySpec = new PKCS8EncodedKeySpec(encPrivateKey);  
KeyFactory keyFactory = KeyFactory.getInstance("GMSS",  
"FlexiPQC");
```
3. Decode GMSS private key:  

```
PrivateKey privateKey = keyFactory.generatePrivate(privateKeySpec);
```
4. Get instance of a GMSS signature:  

```
Signature gmmsSig = Signature.getInstance("GMSSwithSHA1",  
"FlexiPQC");
```
5. Initialize signing:  

```
gmmsSig.initSign(privateKey);
```
6. Sign message:  

```
gmmsSig.update(message.getBytes());  
signature = gmmsSig.sign();  
return signature;
```

## Signature verification

The GMSS Signature verification works as follows:

1. Generate KeySpec from encoded GMSS public key:  
`KeySpec publicKeySpec = new X509EncodedKeySpec(encPublicKey);`
2. Decode GMSS public key:  
`PublicKey publicKey = keyFactory.generatePublic(publicKeySpec);`
3. Initialize verifying:  
`gmssSig.initVerify(publicKey);`
4. Verify the signature:  
`gmssSig.update(message.getBytes());`  
`return gmssSig.verify(signature);`