Darmstadt University of Technology
Department of Computer Science
Cryptography and Computeralgebra

Diploma Thesis

Improved Authentication Path Computation For Merkle Trees

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Michael Schneider
Darmstadt University of Technology
Department of Mathematics

March 2008

Supervisor: Prof. Dr. Johannes Buchmann
Erik Dahmen

Contents

List of Abbreviations

RSA	Cypher and Signature Algorithm of Rivest, Shamir and Adleman
DSA	Digital Signature Algorithm
ECDSA	Elliptic Curve Digital Signature Algorithm
PQC	Post Quantum Cryptography
MD5	Message Digest Algorithm 5
SHA	Secure Hash Algorithm
MAC	Message Authentication Code
DL problem	Discrete Logarithm problem
OTS	One Time Signature
OTSS	One Time Signature Scheme
PRNG	Pseudo Random Number Generator
MSS	Merkle Signature Scheme
CMSS	Coronado Merkle Signature Scheme
GMSS	Generalized Merkle Signature Scheme
JCA	Java Cryptography Architecture
JCE	Java Cryptography Extension
API	Application Programming Interface

List of Figures

List of Tables

1 Introduction

1.1 Outline

Digital signatures are one of the most popular applications of cryptographical technics, besides encryption. The concern is to allow identification, authentication, integrity, and liability in electronic applications. Digital signatures are used for secure interaction over the internet by signing emails or protecting web browser communications by SSL/TLS. They are necessary for procedures like digital voting or bureaucratic solutions over the internet.

Today digital signatures are mostly implemented using asymmetric, also called public key cryptography. Famous examples are the RSA, ECDSA, or DSA signature schemes. In the majority of cases the security of these methods is based on mathematical, number theoretical assumptions, like the factoring of big numbers or the discrete logarithm problem. Today all of these algorithms and schemes can be considered as sufficiently secure. However, new algorithms already exist to solve these number theoretical problems on quantum computers [1, 2]. The established signature algorithms can be used without worries, as long as no practically useable quantum computers exist. But additionally, new techniques must be found to be prepared for the case of working quantum computers. This field of work is called post quantum computing (PQC). Another weakness of the established techniques is the increasing key size. Because today's computers performance develops rapidly, the key size of the used public key schemes must be raised to assure security [3]. This procedure of rising key lengths ends, if algorithms independent of number theoretical assumptions are found.

An alternative way is to use so called one time signatures (OTS). These signature schemes are considered to be secure also on quantum computers. Their security relies on the security of hash functions. A hash function is a mathematical function which is easy to compute but hard to invert. While the keys of a usual signature algorithm can be used more often, the keys of a one time signature must not be used more than once. Otherwise the security of the signature scheme would be reduced,

as an OTS signature reveals parts of the security of the scheme. The problem with one time signatures is that the number of keys that have to be stored and delivered increases enormously. This is a well known problem by symmetric cryptography. To solve this key management problem, Merkle proposed his idea of using binary trees for authentication of big amounts of OTS public keys in 1989 [4]. Using this new idea, it is possible to authenticate up to 2^{20} (and even more) OTS private keys with one single public key. This leads to efficiency in storage concerns, as only one key has to be permanently stored instead of many. Merkle's idea offered the possibility to create a multi-time signature scheme, called Merkle signature scheme (MSS), based on any one time signature scheme. Extending this idea of MSS some extensions and andvancements were proposed: CMSS [5, 6] and GMSS [7], which is a generalization of CMSS. The advantage of GMSS (Generalized Merkle signature scheme) compared to the original merkle scheme is the smaller size of the signatures and a better scheduling of the signature generation. Furthermore the GMSS scheme is parameterized. This feature allows to customize the scheme for different applications, like usage on smartcards or comparable low computation devices where low storage space plays an important role. One important part of the Merkle signature scheme is the traversal of the authentication tree. Whereas simple traversal algorithms arrest the signature generation, a fast traversal algorithm enhances the whole scheme. Thus it is important to develop good authentication path algorithms.

1.2 About This Thesis

The subject of this thesis is the introduction of a new traversal algorithm for Merkle trees and the integration of this algorithm in GMSS, including an implementation in Java for the FlexiProvider. Section 2 gives the background information needed, while section 3 describes former known traversal algorithms. In section 4 the new traversal algorithm is introduced. Correctness and efficiency proofs complete this section. The Java implementation for the FlexiProvider is considered in section 5. In section 6 the comparison of GMSS using the new authentication path algorithm with other established signature algorithms is drawn. Section 7 finally gives a conclusion of the thesis.

The reader of this thesis is supposed to be familiar with fundamental mathematical notations of cryptographic considerations like signing or encryption functions, as well as simple mathematical principles like geometric series. Understanding of basic complexity theoretical ideas (like the $\mathcal{O}$-notation) and algorithm notation might also be necessary to understand the main parts of this thesis.

2 Background

This section informs about the basic mathematical and cryptographical principles and techniques needed for the considered applications. First an introduction of digital signatures is given. Then the principles of hash functions and one time signatures are explained, followed by an example one time signature scheme, the Winternitz OTS scheme, which will be used for the implementation of the new algorithm. After that the idea of Merkle trees and the Merkle signatures are illustrated. Finally, a short explanation of the GMSS extension is given.

2.1 Digital Signatures

The purpose of a digital signature is to offer special security purposes like identification, authentication, integrity or liability. It can, in some parts, be compared to a handwritten signature: only one person can create its own signature, every forgery can be determined. One big difference is that the digital exponent is a mathematical function of the message. If the document changes, the signature changes as well. The digital signature could otherwise be moved from one document to another, as all digital data can be easily copied.

Not only documents are signed digitally. Digital signatures are also used for package transport security in transport protocols. In principle every kind of digital data can be signed. In most applications not the data or document itself is signed but a *message digest* of it. That is a kind of fingerprint of the data. The principles of message digests are explained in section 2.2.

Digital signatures are always based on asymmetric cryptography. Such a system was first introduced by Diffie and Hellman in 1976 [8], which was one of the greatest advances in modern cryptology. For such a digital signature, two different types of keys are needed: a private key for signature generation and a public key for verification. The private key in this purpose is also called the *signing* key and the public key is also called the *verification* key. As one could guess from the name, the private key has to be kept secret, whereas the public key can be spread widely.

Everyone knowing this public key can verify the signature, but only the owner of the private key is able to create one. For one time signature schemes, these keys are generated newly for every signature. In contrast, for multi-time signature schemes both keys are used for bigger amounts of signatures. Some of the commonly used signature algorithms are also used for encryption (like RSA), whereas some of the systems are only applicable to signatures (like DSA and ECDSA). Some attributes of a digital signature scheme are:

Authenticity: Everyone should be able to control that the signer really is the originator of a signature. This is possible because everyone can use the public verification key. Nobody else shall be able to sign a document in the signers name. For this purpose the private signing key must be kept secret.

Non-Repudiation: This property means that the signer can not successfully deny the fact of having signed a document. Everyone possessing the signature and the original document can prove that the signature was really created with the signer's private key.

Since a signature is also a function of the private key and no one besides the signer knows this signing key, then nobody is able to construct signatures which can be verified by the corresponding public key. The signer can never deny having signed a message if a verifiable signature exists.

When the liability has to be proved, a third person (for example a court) has to control if a signature really belongs to the person it should. The non-reusability property of a signature in this concern means that this action can be performed without revealing the private key, so that it can be used again by the user.

Integrity: If a document changes or is manipulated, the signature of the original document (a contract for example) will not match this forged document and will be refused. Therefore changes in data can be proved using digital signatures.

Mathematically, digital signatures are based on one-way functions with trapdoor. A one-way function is a mathematical function which is easy to compute in one direction. However to compute the inversion of the function is hard. If $y = f(x)$ (with a one-way function f) it is easy to compute y given x and f, but it is hard to get x, if only y and f are known. A trapdoor means a secret (e.g. a secret number) which allows to apply the inverse function easily by knowing the secret. In a signature scheme the private key can be considered as the trapdoor. Creation of the signature is the inverse function, which is hard or impossible to compute when the signing key is unknown. When it is said that a function is hard to invert it is meant in today's context: it is possible that in a few years (when the performance of computers has raised furthermore or even quantum computers exist) today's one-way functions will be invertible without problems.

A digital signature *scheme* consists of three parts: a key generation algorithm, the signature construction, and the verification phase. As the name implies, the first part serves for the creation of the private and the public key. The second part is the use of the private signing key for creating the signature of a message. Finally, using the public key and the original message the autenticity of the signature is revised. The individual phases will be described later in the introduction of particular signature schemes.

2.2 Hash Functions

Most of the known multi-signature schemes are based on mathematical assumptions like factoring of big numbers or the discrete logarithm problem. However, one time signatures are mostly based on cryptographic hash functions. For this reason, those message digest principles are illustrated in this section.

A hash function maps any kind of digital data to a shorter, random looking sequence of numbers called the hash value or message digest of the data, which can be seen as kind of a 'fingerprint.' It is mostly represented by a hexadecimal depiction. As an example, the hexadecimal depiction of the 160 bit long SHA1 hash value of the

string 'Improved Authentication Path Computation' is

$$'fa072597154f81ba39b841f265acc8fa2d47d937'$$

Changing only one letter in the original data will change the whole message digest: the SHA1-hash of 'improved Authentication Path Computation' is

$$'70e053246a5e9f591bcae5b47173295899e62cba'$$

More mathematically, a hash function can be denoted as the following:

$$\text{HASH} : X = \{0,1\}^* \rightarrow Y = \{0,1\}^n$$

where the domain X includes all bitstrings with arbitrary length and the codomain Y consists of all n bit strings. An important attitude of a hash function is it's ability to only go one way. This means that it is not possible to generate the original data out of its hash value. A hash function can be considered secure if it assures the following assumptions:

- pre-image resistance
  Given the hash function HASH and a value y, it is not possible to find an x with $\text{HASH}(x) = y$.
- second-pre-image resistance
  Given HASH and x, it is not possible to find an x' (with $x \neq x'$) and $\text{HASH}(x) = \text{HASH}(x')$.
- collision resistance
  Given HASH, it is not possible to find x, x' (with $x \neq x'$) and $\text{HASH}(x) = \text{HASH}(x')$. As the size of the co-domain Y is smaller than the domain size it is clear that there are collisions between different messages out of X. Collision resistance means the impossibility of finding such a collision with non random propability.

In [6] Coronado shows that, for the security of the Merkle signature scheme, one-way-ness and collision resistance of the integrated hash function are sufficient.

Hash functions have different applications in cryptography. They are used for fingerprinting or message authentication codes (MAC) to securely identify data. In most signature algorithms, the message is hashed before it is signed, so that the security increases. For example, without application of a hash function to the message the RSA scheme is not secure against chosen message attacks [9]. Most famous representatives of hash functions are the SHA-family [10] and the Message Digest Algorithm 5 (MD5) [11].

In this thesis $\textsc{Hash} : \{0,1\}^* \rightarrow \{0,1\}^n$ is always an arbitrary hash function. The consecutive application of this function is denoted with superscript numbers: $\textsc{Hash}^2(m)$ stands for $\textsc{Hash}(\textsc{Hash}(m))$.

2.3 One Time Signatures

As mentioned in the introduction, one time signature (OTS) schemes are special kinds of signature algorithms where the signing key must not be used more than once, as every further use of these keys would reveal information which could weaken the security of the signature. Most OTS schemes are based on hash functions [12]. The security does not rely on mathematical problems, but only on the security of the hash function. As mentioned above this is dependent on properties like collision resistance. The security of most algorithms used today for multi-time signatures can only be increased by raising the length of the used keys. In the last 20 years, the key lengths of algorithms like RSA or ECDSA have been constantly increasing [3]. Furthermore if large scale quantum computers exist, the search for collisions of hash functions is hard, whereas the underlying problems of ECDSA and RSA can be computed in linear time. These schemes can be broken on quantum computers, while one time signature schemes based on hash functions remain secure.

As the computation of hash functions is fast, one time signatures are very efficient. Their application is possible on low computation complexity devices like smart cards.

2.3.1 The Winternitz One Time Signature Scheme

In this thesis, as in the actual GMSS, the Winternitz One Time Signature Scheme is used [4] [12]. The usage of other one time signature schemes like the BiBa scheme [13] would be possible as well. The Winternitz scheme uses a parameter w, which is typically chosen a small power of two. This parameter w allows a trade-off between generation cost and signature size. It defines the bit length of the single parts of the private key, whereas t_w is the count of components. With n as length of a hash, we define

$$t_w = \left\lceil n/w \right\rceil + \left\lceil \Big(\lfloor log_2(\lceil n/w \rceil) \rfloor + 1 + w \Big)/w \right\rceil$$

The private signature key is $X = (x_1, \ldots, x_{t_w})$, where $x_1 \ldots x_{t_w}$ are random values. For the generation of random data, a pseudo random number generator (PRNG) is used:

$$\text{PRNG}: \quad \{0,1\}^n \mapsto \{0,1\}^n \times \{0,1\}^n : \quad \text{SEED}_{in} \mapsto (\text{SEED}_{out}, \text{RAND})$$

It uses a value SEED_{in} to generate two random looking values SEED_{out} and RAND. If SEED_{out} is again used as input for the same PRNG we get a chain of values RAND_i which can always be reproduced by knowledge of only the first SEED_{in}. In this thesis the used PRNG is always the one described in the Digital Signature Standard (Appendix 3.1) [14] which requires only one call to a hash function HASH:

$$\text{RAND} \leftarrow \text{HASH}(\text{SEED}_{in}), \qquad \text{SEED}_{out} \leftarrow (1 + \text{SEED}_{in} + \text{RAND}) \mod 2^n$$

Key Generation. For the public key we apply the hash function $2^w - 1$ times to each x_i, i.e. we calculate $y_i = \text{HASH}^{2^w-1}(x_i)$ for $i = 1 \ldots t_w$. The verification key is then created out of the concatenation of the y_i-values:

$$Y = \text{HASH}(y_1 \| \ldots \| y_{t_w})$$

Signature Generation. For generation of the signature of a message first of all the n-bit message digest of this message is created. The digest md is then split

into $\lceil n/w \rceil$ parts $md_1 \dots md_{\lceil n/w \rceil}$, each with a length of w (if necessary zeros are padded first). Then the checksum $C = \sum_{i=1}^{\lceil n/w \rceil} 2^w - md_i$ is built. This checksum is also divided into blocks of length w, namely $md_{\lceil n/w \rceil + 1} \dots md_{t_w}$. The final signature is created by concatenating the hash-values $s_i = \text{HASH}^{md_i}(x_i)$ for $i = 1 \dots t_w$. The signature is then

$$\text{SIG} = (s_1 \| \dots \| s_{t_w})$$

Verification. For verifying the message digest, the signature and the verification key are needed. First the values md_i are computed in the same manner as in the signing process. Then $v_i = \text{HASH}^{2^w - md_i - 1}(s_i)$ is generated. Now the vector $V = \text{HASH}(v_1 \| \dots \| v_{t_w})$ can be compared to the verification key. Each of the x_i values should now have been hashed $2^w - 1$ times. The signature is declared to be verified if and only if $V = Y$. Example 1 explains a Winternitz OTS sample instance.

Without using the checksum an attacker could hash again some of the s_i values. The result would be a valid signature which could not be verified by the original public key. Therefore the scheme would not be secure against known signature attacks leading to existential forgery. For this, the checksum is appended to the signature, so that every additional hash to one of the s_i can be detected.

Example 1. Consider a 15 bit message digest to be signed: $md = 101100000010010$. Choose $w = 4$.
Key Generation

$$\text{PRNG} \Rightarrow X = (\underbrace{0101}_{x_1}\ \underbrace{1100}_{x_2}\ \underbrace{1010}_{x_3}\ \underbrace{1110}_{x_4}\ \underbrace{0011}_{x_5}\ \underbrace{1111}_{x_6}) \quad \text{(Private key)}$$

$$\begin{aligned} t_w &= \left\lceil 15/4 \right\rceil + \left\lceil \left(\lfloor log_2(\lceil 15/4 \rceil) \rfloor + 1 + 4 \right) / 4 \right\rceil \\ &= 4 + \left\lceil (2 + 1 + 4) / 4 \right\rceil = 6 \end{aligned}$$

$$\Rightarrow \text{ Public key:} \qquad Y = (\text{HASH}^{15}(x_1), \dots, \text{HASH}^{15}(x_6))$$

Signature Generation

$$md = \underbrace{\mathbf{0}101}_{b_1}\ \underbrace{1000}_{b_2}\ \underbrace{0001}_{b_3}\ \underbrace{0010}_{b_4}$$

$$\begin{aligned} C &= (10000 - 0101) + (10000 - 1000) + (10000 - 0001) + (10000 - 0010) \\ &= 1011 + 1000 + 1111 + 1110 = 110000 \end{aligned}$$

$$\Rightarrow b_5 = \mathbf{00}11 \text{ and } b_6 = 0000$$

$$\begin{aligned} \text{SIG} &= \Big(\text{HASH}^5(x_1) \,\|\, \text{HASH}^8(x_2) \,\|\, \text{HASH}(x_3) \,\|\, \text{HASH}^2(x_4) \,\|\, \text{HASH}^3(x_5) \,\|\, x_6\Big) \\ &= (s_1 \| \ldots \| s_6) \end{aligned}$$

Verification (b_i the same as above)

$$\begin{aligned} V &= \Big(\text{HASH}^{10}(s_1) \,\|\, \text{HASH}^7(s_2) \,\|\, \text{HASH}^{14}(s_3) \,\| \\ &\qquad\qquad \text{HASH}^{13}(s_4) \,\|\, \text{HASH}^{12}(s_5) \,\|\, \text{HASH}^{15}(s_6)\Big) \\ &= \Big(\text{HASH}^{15}(x_1) \,\|\, \ldots \,\|\, \text{HASH}^{15}(x_6)\Big) \stackrel{!}{=} Y \end{aligned}$$

□

2.4 Merkle Trees

A problem which occurs by usage of one time signatures is well known from symmetric cryptography applications: the space needed to store all involved keys rises too fast. For every message a user A wants to send to another user B, a private key must be created for user A. Additionally, user B has to store one public key for every message.

Merkle's idea was to use a complete binary tree for verification of one time signatures. With this approach many signatures can be verified by one single public key. The storage needed for the verification key is extremely small (only one key has to be stored). Every one time signature scheme can be extended to a multi-time one by using such an authentication tree.

A complete binary tree of height H consists of 2^H leaves and $2^H - 1$ inner nodes. The height of a leaf is defined to be 0, whereas the height of inner nodes denotes the length of a path down to a leaf. Thus, the root node has height H. The leaves are numbered consecutively from left to right, starting with 0. An example tree can be seen in Figure 1.

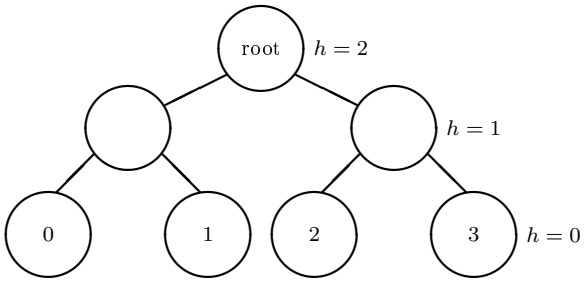

Figure 1: A complete binary tree of height H = 2. The values $0 \ldots 4$ are the leaf indices, h denotes the nodes' height

Merkle trees were first introduced by Merkle in 1989 [4]. A Merkle tree is a complete binary tree equipped with a hash function HASH. The values $\Phi(n)$ of a leaf can be chosen arbitrarily, whereas the values of inner nodes are calculated by the following: for each inner node n_{parent} the value $\Phi(n_{parent})$ is defined to be the hash of the concatenation of the left and right child nodes n_{left} and n_{right}:

$$\Phi(n_{parent}) = \text{HASH}(n_{left} \,\|\, n_{right})$$

By this construction the Merkle tree is completely determined by the leaf values. A sample tree is shown in Figure 2.

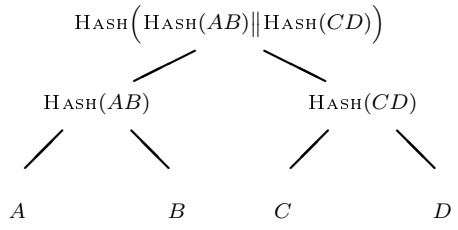

Figure 2: A Merkle tree with leaf values A, B, C, D

Merkle trees are used for authenticating the leaf data using the root value. For this purpose additional data is required, called the *authentication data*. For authenticating leaf i, on each height h ($h = 0 \dots H-1$) one node value AUTH_h is stored, namely the sibling of the nodes on the path from leaf i up to the root. An example for the authentication path is illustrated in Figure 3. For authenticating leaf i, one starts at the bottom of the tree. Using the leaf value and the authentication data on each height by concatenating and hashing the root value can be computed. If the original stored root value is identical to the newly calculated one, the leaf value is truely authenticated.

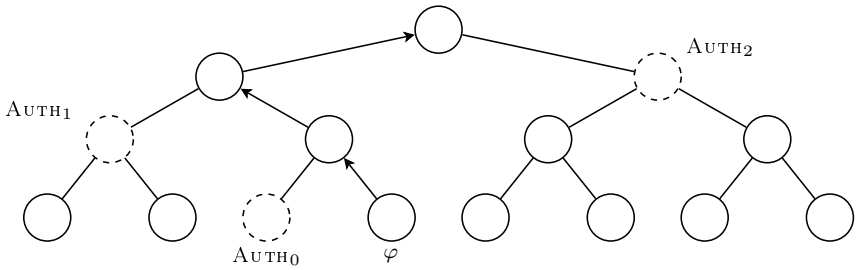

Figure 3: Authentication data of leaf φ. Hashing the concatenation of AUTH_0 and $\Phi(\varphi)$ gives the upper node, continuing up the root finally gives the root value. The dashed nodes denote the authentication path for leaf φ. The arrows indicate the path from leaf φ to the root.

Besides digital signatures Merkle trees have been implemented for other useful applications like wireless security [15]. As authentication is the real purpose of the Merkle tree and not signature verification, lots of other applications are imaginable. However, this thesis will only focus on the application of digital signatures.

2.5 The Merkle Signature Scheme

2.5.1 MSS - Merkle Signature Scheme

The Merkle Signature Scheme (MSS) proposed in [4] consists of a one time signature scheme like the Winternitz OTSS and a Merkle tree. A Merkle tree of height H can be used to authenticate 2^H OTS keys (one for each leaf of the tree). The leaf values

of the tree are formed by the OTS public keys Y_i. More precisely the three signature steps are:

MSS Key Generation. The MSS private key is the set of OTS private keys $(Y_1, \ldots, Y_{2^H})$ which are computed as usual, depending on the used scheme (for the Winternitz scheme e.g. see section 2.3.1). The OTS public keys are hashed and stored as the tree's leaf values. By concatenating and hashing each two child nodes, the node labels of the tree can be computed from bottom up to the root. The root value of the tree forms the MSS public key for verification.

The key pair generation uses an algorithm called *treehash* (Algorithm 1) [16]. This algorithm is used to compute the root of a Merkle tree using a stack structure equipped with the usual push and pop operations [1]. It consecutively computes the 2^H leaf values consisting of the OTS verification keys Y_j from left to right and pushs them on the stack. When two nodes of the same height lie on the stack, they are concatenated and hashed to the next upper node. After complete 2^H leaf calculations and $2^H - 1$ hash evaluations, the root of the Merkle tree is the upper node on the stack. Figure 4 illustrates an example.

Algorithm 1 Treehash

Input: Leaf l, stack STACK
Output: updated stack STACK

1. push l to STACK
2. **while** top two nodes of STACK have same height **do**
   (a) pop n_1 from STACK, pop n_2 from STACK
   (b) push HASH$(n_1 \| n_2)$ to STACK
3. **return** STACK.

MSS Signature Generation. Complete 2^H signatures can be created using one Merkle Tree. For each new signature the next OTS key is used so that each OTS

[1] A stack is a data structure using a *'first in - first out'* strategy: `push` stores a node on top of the stack, `pop` delivers the top node of the stack.

D	height 0
C	height 0
HASH(AB)	height 1

Stack before hashing

HASH(HASH($AB | $HASH($CD$)))	height 2

Stack after hashing

Figure 4: Sample of the treehash algorithm: value 'D' is pushed on the stack. Then 'C' and 'D' are hashed to a height 1 node which is then again hashed with the bottom node to a height 2 node.

key is only used once. The MSS signature consists of the index φ that appoints which OTS key is used for the current signature. Furthermore the OTS signature, the OTS verification key Y_φ and the authentication data of leaf φ are components of the MSS signature: $\text{SIG}_{MSS} = (\varphi, \text{SIG}_{OTS}, Y_\varphi, \{\text{AUTH}_\varphi\})$.

MSS Verification. The first step of verification is the control of the OTS signature using the key Y_φ. If this phase fails, the whole MSS signature is rejected as invalid. Otherwise the authentication of this key is necessary. This happens by calculating the root value of the tree using the value Y_φ and the authentication data stored in the MSS signature. First Y_φ is concatenated and hashed with AUTH_0 on the lowest level, then the result is hashed again with AUTH_1 and so on up to the root. If the thus computed root is equal to the public MSS key, the signature is considered to be valid.

SEED calculation. Every leaf of the Merkle tree requires a random value SEED_{OTS} for the generation of the x_k values needed for generation of the Winternitz OTS keys. This random data is calculated using the PRNG as described in section 2.3.1:

$$(1) \qquad (\text{SEED}_{\varphi+1}, \text{SEED}_{OTS}) \leftarrow \text{PRNG}(\text{SEED}_\varphi)$$

$$(2) \qquad (\text{SEED}_{OTS}, x_k) \leftarrow \text{PRNG}(\text{SEED}_{OTS}),\ k = 1 \ldots t_{w_i}$$

As input a random value SEED_0 is required. Formula (1) generates the seeds needed for the leaves. Formula (2) delivers the random data x_k. This seed calculation can

be seen as a lattice of seed values, as Figure 5 illustrates: the upper line shows the consecutive calculation of the SEED_{OTS} values, whereas the downside lines show the generation of the x_k values.

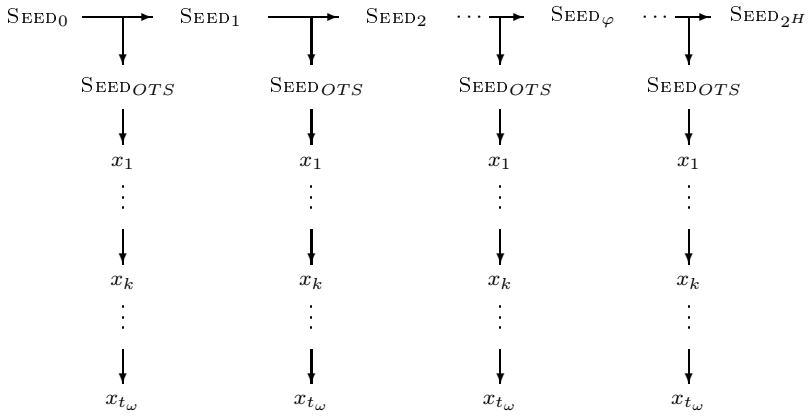

Figure 5: Seed generation for a single Merkle tree. Each array indicates one call to the PRNG.

With this construction of the seed values we get a value SEED_{2^H} as output of the seed calculation. This output will be used in GMSS later on. Here we conclude that for the generation of all private and public keys only one initial seed value is required.

Security Of MSS. The security of MSS was regarded in [6]. It can be proved that the Merkle signature scheme resists any adaptive chosen message attack if

1. a secure, that means collision resistant hash function exists and
2. the underlying one time signature scheme resists any forgery.

A chosen message attack is an attack where the adversary has the possibility to get a valid signature to every chosen message. He can use this message/signature pairs either to forge a signature or to break the private key. Adaptive in this concern means that the attackers messages can be chosen dependent on further message/signature

pairs. As Coronado shows in [6], this attack will fail if the above mentioned assumptions hold. The Merkle signature scheme can be constructed using an arbitrary hash function. So if a hash function should get insecure, it can be easily substituted by a secure one. The MSS remains secure.

2.5.2 GMSS - Generalized Merkle Signature Scheme

As mentioned above GMSS is an expansion of the Merkle signature scheme. GMSS stands for *generalized Merkle signature scheme* and was proposed in 2007 [7]. One instance of GMSS is CMSS, which was proposed in 2006 [5]. When MSS and CMSS have relatively large sized signatures, GMSS is adressed to allow smaller signatures, and faster generation and verification. Additionally with GMSS it is possible to sign up to 2^{80} and even more messages, while with MSS this number is only applicable up to 2^{20}. This attribute is helpful considering practical appliances like web server applications, where big amounts of signatures are necessary. The parameterization of GMSS allows the choice of either fast runtime, small signatures or a trade-off between both depending on the application. This section introduces the main characteristics and gives an overview about GMSS. A more detailed description can be found in [7] and [17].

General Construction. The general GMSS construction is made up of a tree with height T. The nodes of this tree are again Merkle trees. Each of the Merkle trees on layer i of the basic tree has height h_i and is parent of 2^{h_i} Merkle trees on the layer $i+1$. The Merkle trees are labeled $\mathcal{T}_{i,j}$, where i is the level in the basic tree and j is the number of the node on height i, consecutively numbered from left to right with $0 \dots 2^{h_1+h_2+\dots+h_i-1} - 1$. The root tree is labeled $\mathcal{T}_{1,0}$.

Again the Winternitz OTS scheme is used for the signatures in the Merkle trees. For each layer a different parameter w_i, $i = 1 \dots T$ is allowed. GMSS is parameterized by the height of the basic tree, the heights of the trees on each layer and the Winternitz parameters. Altogether the parameter set $\mathcal{P}$ of GMSS is

$$\mathcal{P} = (T, (h_1, \dots, h_T), (w_1, \dots, w_T))$$

CMSS is the variant defined by the parameters $\mathcal{P} = (2, (h, h), (w, w))$.

The root of each Merkle tree $T_{i,j}$ is labeled $\text{ROOT}_{T_{i,j}}$. It gets signed with the OTS key of the corresponding parent leaf: the root of tree $\mathcal{T}_{i,j}$ is signed using the signature key of a leaf of the parent tree on height $i-1$. The signature of tree $\mathcal{T}$'s root is called $\text{SIG}_{\mathcal{T}}$. To sign a message digest the signature keys of the Merkle trees on the deepest layer T are used. These signatures are denoted with SIG_d. Following this construction the number of message digests that can be signed is $2^{h_1+h_2 \ldots h_T}$. The general construction of GMSS is illustrated in figure 6.

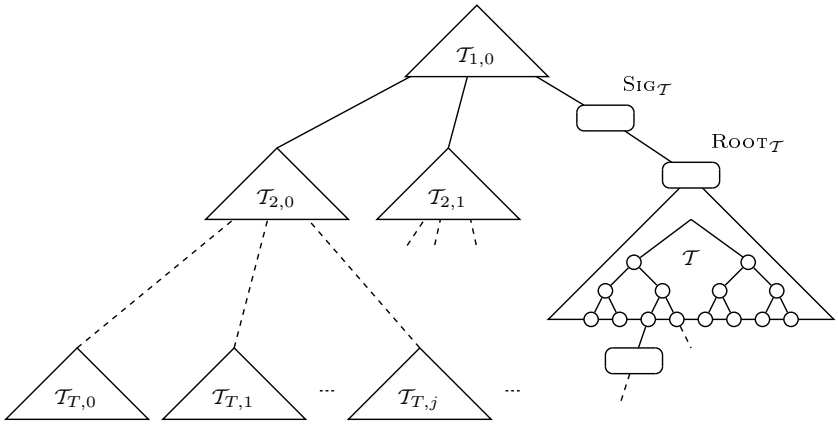

Figure 6: Basic construction of GMSS. Only the leaves on the lowest layer are used for GMSS signatures.

As on upper layers the leaves advance less frequently, the precomputation of these trees can be distributed over many steps. This property allows an advance in signature generation time. As well it allows the choice of higher parameters w_i for the OTS scheme, which leads to a smaller signature size in total.

A GMSS signature. As known from MSS for each signature there is a unique path from the leaf φ up to the root. Here this path contains one Merkle tree on each layer. Additional to the one time signature of the message digest, the one time signatures of the root values of these trees are stored in the GMSS signature. Also the authentication data on the path existing of $\text{AUTH}_{\mathcal{T},l}$ for each tree $\mathcal{T}$ is appended to the GMSS signature. Hereby l is the index of the leaf of tree $\mathcal{T}$ used for signing

the root of the tree on the layer below. On the deepest layer the authentication data of the leaf used to sign the message digest is appended. An example of this process is depicted in figure 7.

Totally the GMSS signature consists of the following:

- the index φ of the leaf used
- the one time signature SIG_d of the document d signed with the key corresponding to a leaf of the lowest layer
- the one time signatures $\text{SIG}_{\mathcal{T}_{i,j}}$ of the roots
- the authentication paths $\text{AUTH}_{\mathcal{T}_{i,j}}$ of each tree on the path from the bottom leaf φ to the GMSS root

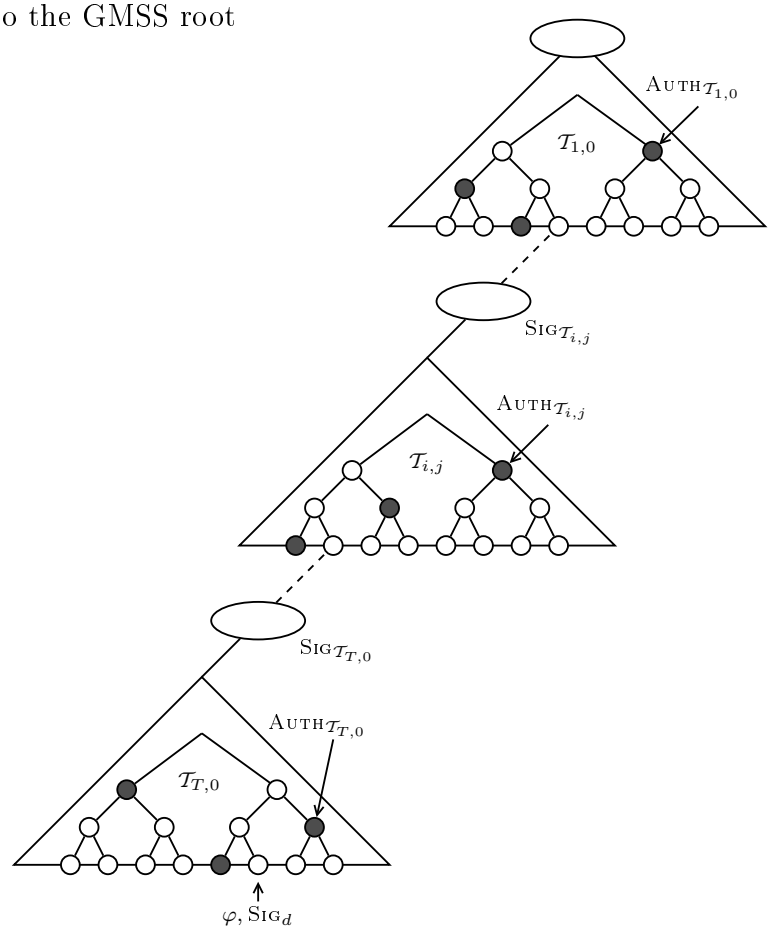

Figure 7: Example of a GMSS signature

Seed calculation in GMSS. For every single Merkle tree of the GMSS construct the seed generation procedure described on page 25 is used. There an initial seed for every tree is needed. For every tree of the GMSS structure this is an initial seed value $\text{SEED}_{in_{\mathcal{T}_{i,j}}}$. The SEED_{in} for the first tree in each layer ($\text{SEED}_{in_{\mathcal{T}_{i,0}}}$) is required as input. The following $seed_{in}$ values are computed as the output of the last leaf of the previous tree:

$$(\text{SEED}_{in_{\mathcal{T}_{i,j+1}}}, \text{SEED}_{OTS}) \leftarrow \text{PRNG}(\text{SEED}_{2^{h_i}})$$

Here $\text{SEED}_{2^{h_i}}$ is the seed of the last leaf of tree $\mathcal{T}_{i,j}$. Hence using one initial seed for each layer all required seed values can be constucted.

GMSS Key Generation. This phase uses the initial seed values for constructing the public and private keys needed for GMSS. The GMSS public key is the root of the top Merkle tree: $\text{ROOT}_{\mathcal{T}_{1,0}}$. The private key is built by the following:

$$\begin{array}{rcl@{\qquad\qquad}rcl} \text{SEED}_{in_{\mathcal{T}_{i,0}}},\ i & = & 1 \ldots T & \text{SEED}_{in_{\mathcal{T}_{i,2}}},\ i & = & 2 \ldots T \\ \text{SIG}_{\mathcal{T}_{i,0}},\ i & = & 2 \ldots T & \text{ROOT}_{\mathcal{T}_{i,1}},\ i & = & 2 \ldots T \\ \text{AUTH}_{\mathcal{T}_{i,0},0},\ i & = & 1 \ldots T & \text{AUTH}_{\mathcal{T}_{i,1},0},\ i & = & 2 \ldots T \end{array}$$

Using the *treehash* algorithm (Algorithm 1) the root values of the first Merkle tree on each layer $\mathcal{T}_{i,0}$ (including the GMSS public key $\text{ROOT}_{\mathcal{T}_{1,0}}$) are built. For this the initial seed values $\text{SEED}_{in_{\mathcal{T}_{i,0}}}$ are needed. While calculating these roots the authentication data of the first tree $\text{AUTH}_{\mathcal{T}_{i,0},0}$ of each layer can be stored, so that the AUTH values for these trees are obtained for free. The initial seeds for the second trees are now available. The same as above the root values of the second trees $\text{ROOT}_{\mathcal{T}_{i,1}}$ and the corresponding authentication data $\text{AUTH}_{\mathcal{T}_{i,1},0}$ are generated with Algorithm 1. After this the initial seed values for the third tree of each layer $\text{SEED}_{in_{\mathcal{T}_{i,2}}}$ is ready and can be stored in the private key. The signatures $\text{SIG}_{\mathcal{T}_{i,0}}$ are the one time signatures of the root values already known.

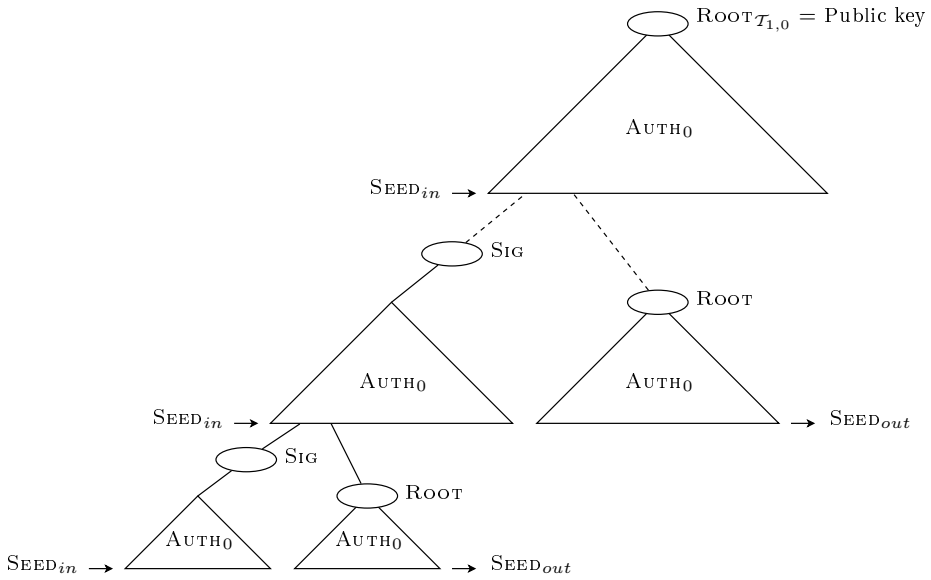

Figure 8: Example GMSS keys. The private key consists of the authentication path for the first leaf of the first two trees on each layer, the SEED$_{in}$ for the first and the third tree on each layer, the root signatures SIG of the first trees and the root values ROOT of the second trees. The public key is the uppermost root value ROOT$_{\mathcal{T}_{1,0}}$.

This private GMSS key is the key for the first signature. Having created this signature the key is updated and so changes for every new signature. Therefore the GMSS scheme is called a *key evolving signature scheme* [18]. As the private signing key changes (*evolves*) frequently, this leads to a special security feature of GMSS, so called *forward security*. Also if an adversary compromises the actual signing key, it is impossible to forge signatures belonging to former signing keys. Using the introduced seed scheduling, MSS does contain this security feature as well [6].

GMSS Signature Generation. The signature generation is distributed in an online and an offline part. Such a separated framework is described in [19]. The offline part can be seen as preparation of the next online part. This online part can not be done until the message is known. It is a fast process, so that the signature can be generated rapidly, when the offline part has already been done. The offline part belonging to the first signature was done during the key generation phase. Later during the offline phase the private key has to be updated (as mentioned above, *key*

evolving scheme). The online part only consists of the generation of the signature. All parts needed for this signature were created and provided by the previous offline part. A detailled description of both phases can be found in [7, 17].

The offline part distributes the computation of the needed ROOT, leaf and SIG values, so that for each signature the time to spend is not too different. If a ROOT or a SIG value is computed at once, the actual round lasts longer than previous rounds where no such time expensive operations were done. Therefore the computation of those values is distributed over the calculation of the leaves of the underlying layer, i.e. over $2^{h_{i+1}}$ steps. Figure 9 illustrates the precomputation of those values.

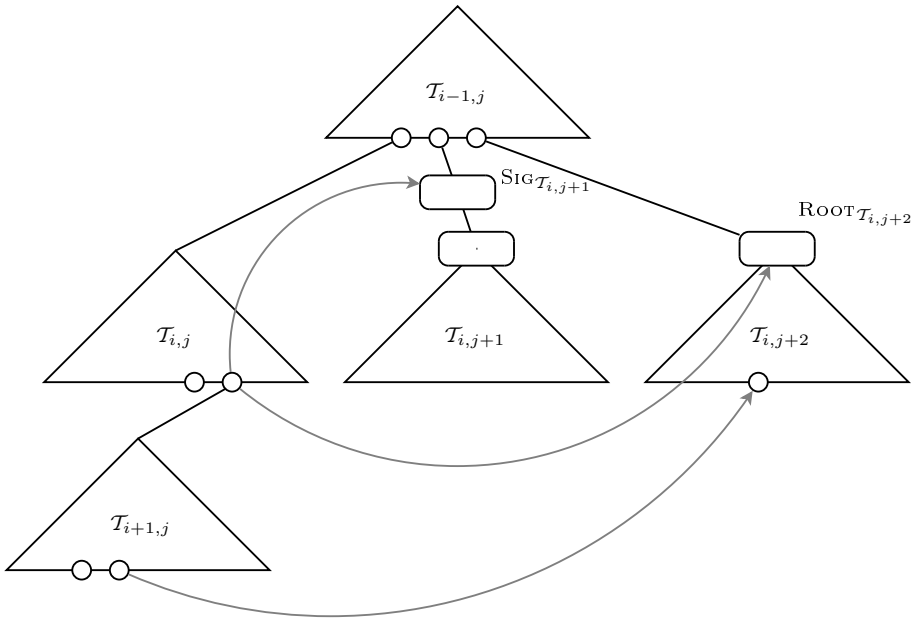

Figure 9: While advancing a leaf in tree $\mathcal{T}_{i,j}$, the values $\text{SIG}_{\mathcal{T}_{i,j+1}}$ and $\text{ROOT}_{\mathcal{T}_{i,j+2}}$ are updated, so that the computation of those values is distributed over all 2^{h_i} steps of tree $\mathcal{T}_{i,j}$. While doing one step in $\mathcal{T}_{i+1,j}$ the leaf of tree $\mathcal{T}_{i,j+2}$ is partly computed.

GMSS Verification. The GMSS verification is nearly the same as in the original Merkle scheme. The first part is the verification of the one time signature of the original data. If this already fails, the verification can be stopped. Next the authentication starts with the tree on the lowest layer. Using the corresponding authentication data the root value of all trees can be calculated. The one time sig-

nature of the roots are compared to the values SIG in the signature. Also if one of these signatures cannot be verified truly, the GMSS verification fails with a negative result. Ending up at the root $\text{ROOT}_{\mathcal{T}_{1,0}}$ of the GMSS construction, this can be compared to the GMSS public key. Only if this comparison is successful the whole signature is accepted.

Needed Storage. Following [7] the size of the keys and the signature is:

$$\begin{aligned} m_{\text{pubkey}} &= n \text{ bits} \\ m_{\text{privkey}} &= \Big(\sum_{i=1}^{T} (h_i + 1) + \sum_{i=2}^{T} (h_i + t_{w_{i-1}} + 2) \Big) \cdot n \text{ bits} \\ m_{\text{signature}} &= \sum_{i=1}^{T} (h_i + t_{w_i}) \cdot n \text{ bits} \end{aligned}$$

The variable n again denotes the length of the output of the hash function HASH. The public key is only one single hash value, that's why its bit length is n. The size of the private key and the signature can easily be derived from the listings above.

In practice these numbers will not hold. Some additional data has to be stored, for example the parameters $\mathcal{P}$ must be added to the public key as they are needed for the verification process. So these numbers are more theoretical, but they give an idea of the overall sizes of signatures and keys. A comparison of the needed storage capacity can be found in section 6.

3 Common Traversal Algorithms

3.1 Overview

The Merkle tree traversal problem is the challenge of computing the authentication paths of consecutive leaves of one single Merkle tree. This is one of the most crucial steps in the Merkle signature scheme and its derivatives. Today MSS and its descendants are not often used in practice, because they are too slow or the signature size is too big. Better traversal techniques may speed up the signature generation (as well as better implementations like GMSS shall make the system more useful for practical considerations). As consecutive leaves mostly share a lot of authentication nodes, only the changes have to be computed from one leaf to the following. Good scheduling algorithms use this fact to speed up the computation of new authentication data.

With digital signatures a tree traversal algorithm for authentication data consist of three phases: *key generation*, *output* and *verification*.

During the *key generation* phase the root of the Merkle tree is constructed and the first authentication path is stored. Some additional authentication data can be stored used as input for the traversal algorithm as well.

The *output* phase consists of 2^H rounds. In each round the leaf value $\Phi(\varphi)$ and the authentication data $\{\text{AUTH}_h\}$ of leaf φ is output and then updated for the next round. This is the main part, requiring good scheduling ideas.

The *verification* phase is always the same as for the original Merkle tree.

In his original paper Merkle introduced a simple traversal algorithm [4]. Jakobsson et. al. proposed an algorithm using subtrees in [20]. This algorithm allows a trade-off between storage and computation time. It needs a maximum of $2H \,/\, \log(H)$ hash function evaluations and maximum storage of $1.5H^2 \,/\, \log(H)$ hash values per round. An implementation of the Merkle signature scheme using Jakobsson's ideas can be found in [21].

Szydlo presented a log-time and log-space algorithm in [22] and a slightly different version in a preprint in [16]. An algorithm is called logarithmic if its time per round respectively the maximum memory capacity needed is logarithmic in the total number of signatures N. He also proves that these bounds are optimal for the authentication path computation, i.e. that it is not possible to create an algorithm that in both time and space complexity is better than $\mathcal{O}(\log N)$. Other work considering authentication path computation can be found in [23]. The new algorithm presented in this thesis is an improvement of Szydlo's algorithms. For this reason the outline of this section is the introduction of Szydlo's traversal algorithm (the more efficient preprint version of [16], not the more simple, published version of [22]). The description of Merkle's classical algorithm leads to Szydlo's improved algorithm version (Algorithm 2). Finally some drawbacks of Szydlo's algorithm are presented to motivate the improved algorithm presented in the main part of this thesis.

3.1.1 Notation

For authentication data the notation already known is used: AUTH_h is the height h sibling on the path from the current leaf φ to the root. Further on for each level h of the tree one instance of the *treehash* algorithm (Algorithm 1) called STACK_h is used. For practical considerations two methods `initialize()` and `update()` exist for these instances. The first method only sets the start node index and the desired height of the instance. The `update()` method either computes a node and pushs it on the stack or it once hashes the stack's top nodes if possible (if top nodes have same height). Temporarily stored nodes on a stack are called *tail* nodes. If STACK_h is completed, the top node is stored in an array NEED_h. There all upcoming right nodes are stored until they are needed for an authentication path.

Some computed nodes are later on again helpful for speeding up the computation of higher left nodes. For each height h at most one such additional node can be kept. For this the set KEEP_h is used. The height of the tree is denoted H, hence the number of nodes is $N = 2^H$, numbered from 0 to $N - 1$ from left to right. All papers [4, 16, 20, 22, 23] do not consider the complexity of the calculation of one leaf. They use an oracle $\text{LEAFCALC}(\varphi)$ which computes the leaf value $\Phi(\varphi)$. The

call of this oracle is counted as one computation unit for the complexity analysis, as well as hash function evaluations are counted one unit each.

3.2 Szydlo's Algorithm

3.2.1 Motivation

The classical algorithm introduced by Merkle in his original paper distinguishes between computation of left and right authentication nodes. It uses one treehash instance for each height, as described above. Using these, new upcoming *right* authentication nodes are precomputed, for that they are ready when they are needed for AUTH values. In each round $\varphi \in [0 \dots 2^H - 1]$ every treehash instance gets one update, if it was not already completed. This leads to the following problem: in the worst case all H treehash instances are active at the same time. So the maximum number of required space units is $0.5(\log(N))^2$. Szydlo's idea was to change the scheduling strategy for the treehash instances to save memory.

The generation of left nodes is quite easy, because their child nodes have already been computed. Saving those child nodes only one hash operation is required for computation of a left authentication node.

As Merkle did in his original algorithm, Szydlo distinguishes between the computation of left and right authentication nodes. The computation of left nodes is quite the same as in Merkle's paper.

3.2.2 The Algorithm

As input Algorithm 2 needs the authentication path of the first leaf of the Merkle tree. These values $\{\text{AUTH}_h, h = 0 \dots H\}$ can be stored during the key generation phase when computing the root of the tree. So the first authentication path is obtained for free. Every round of the authentication path algorithm of Szydlo the same steps are executed:

Generating an output. Every round starts with the output of the previous computed authentication path. This will always be completed when it is needed. Additionally the current leaf value $\Phi(\varphi)$ is output. If the leaf index is even, this value must be computed using one LEAFCALC operation, otherwise it is always available.

Left node computation. For each leaf φ exactly one new left authentication node L must be added. The height of this node is the height of the first parent node of leaf φ that is a left node. This height is denoted τ. If the current leaf is a left node itself, τ is set to 0. Figure 10 shows an example. The new node on height τ is stored as AUTH_τ. If $\tau > 0$, both child nodes of the new authentication node have already been computed and stored in $\text{AUTH}_{\tau-1}$ and $\text{KEEP}_{\tau-1}$. Out of these two child nodes the parent node L can be computed (by concatenating and hashing), so the new node requires only one hash calculation. All nodes AUTH_i with index $i < \tau$ are reset with values from completed treehash stacks (NEED_i).

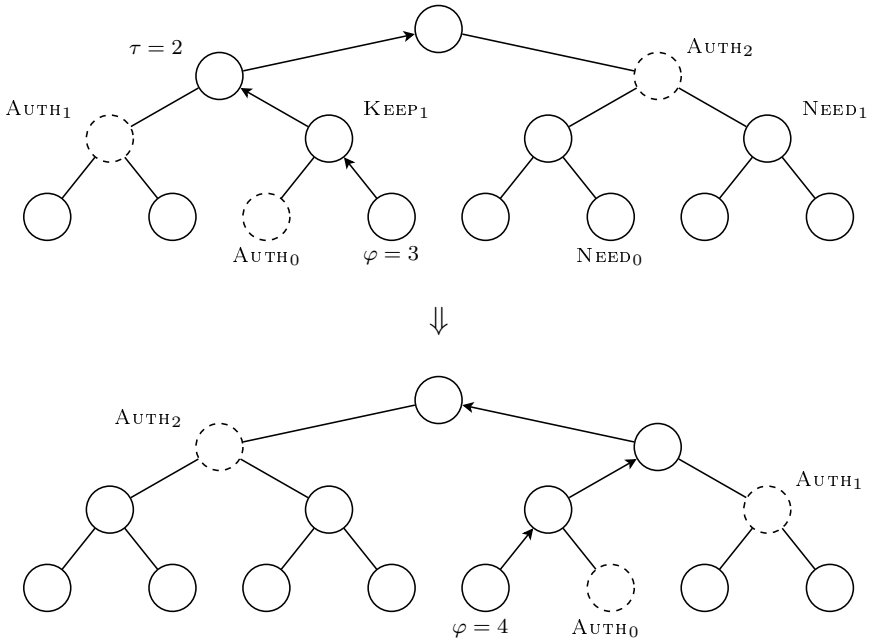

Figure 10: Left node computation: A Merkle tree of height 4 in rounds $\varphi = 3$ and $\varphi = 4$. In the upper tree the height of the first parent of leaf φ that is a left node is $\tau = 2$. The lower figure shows the authentication data of leaf $\varphi = 4$. All lower authentication nodes (AUTH_0 and AUTH_1) are pushed from the stacks an reset in round $\varphi = 3$.

Releasing space. Some previously stored nodes are no more needed after the computation of the new left node. Therefore some memory spaces can be freed by deleting the values AUTH_i for $i < \tau$ and $\text{KEEP}_{\tau-1}$. The former value AUTH_τ is stored in KEEP_τ, for possibly this node is needed for a new left node creation one layer above.

Stack creation. Every round one stack is initialized anew: the stack belonging to height τ. The new left node L has replaced AUTH_τ in this round. Then 2^τ rounds later again this authentication node will change to a right node. This right node is created by STACK_τ, the one which is initialized anew. The starting index for this stack is $\varphi + 1 + 2^{\tau+1}$.

Building needed future nodes. In total, exactly H operations shall be performed in one round. One is already spent either in step 2 (if φ is even) or in step 4 (if $\tau = 0$ which is equivalent to φ is odd) of Algorithm 2. So still $H - 1$ operations are to perform in step 6. Here the main improvement to Merkle's classical algorithm takes place: the scheduling for choosing which stack gets an update. Szydlo always chooses the stack with the lowest top node. One update (either LEAFCALC or hash operation) is performed to this stack. This happens $H - 1$ times, so that exactly H computation units are spent in each round.

The whole algorithm description is depicted in Algorithm 2.

Algorithm 2 Logarithmic Merkle Tree Traversal

Input: First authentication path $\{\text{AUTH}_h\}$
Output: Auththentication paths for leaves $\varphi + 1$

1. Let $\varphi = 0$
2. **Output** If φ is even, compute $\Phi(\varphi) = \text{LEAFCALC}(\varphi)$. Output $\Phi(\varphi)$, for each $h \in [0, H-1]$ output AUTH_h
3. **Release nodes** Let L be the current leaf if φ is even, or its first ancestor which is a left node. Let τ be the height of L (equal to the highest τ with $2^\tau | (\varphi + 1)$). Remove certain expired nodes below L:
   - Remove all node values AUTH_i for $i < (\tau - 1)$
   - **if** $\tau = 0$ record $\Phi(\varphi + 1) = \text{AUTH}_0$
   - **if** L's parent is a right node, remove L's sibling, AUTH_τ
   - **if** L's parent is a left node, set $\text{KEEP}_\tau = \text{AUTH}_\tau$
4. **Add left node**
   - **if** $\tau = 0$ set $\text{AUTH}_0 = \Phi(\varphi)$
   - **if** $\tau > 0$ compute $\text{AUTH}_\tau = \text{HASH}(\text{AUTH}_{\tau-1} \| \text{KEEP}_{\tau-1})$
   - Remove node values $\text{AUTH}_{\tau-1}$ and $\text{KEEP}_{\tau-1}$.
   - Copy new lower right nodes: **for** $i < \tau$ set $\text{AUTH}_i = \text{NEED}_i$
5. **Add stack** Create STACK_τ at height τ, with starting value $\varphi + 1 + 2^{\tau+1}$
6. **Building needed nodes**
   Repeat $H - 1$ times
   - set *active* to be the stack with the lowest node (choose the lowest of such index in case of a tie)
   - **if** there is no such active stack, break and go to step 7
   - Spend one unit building STACK_{active}, as in TREEHASH
   - **if** STACK_{active} is complete, put result in NEED_{active} and destroy STACK_{active}
7. **Loop to next round**
   - Set $\varphi = \varphi + 1$
   - **if** $\varphi < 2^H$ go to Step 2

It is an important task to show that every right authentication node is completed when it is needed by the traversal algorithm. The proof of correctness of the presented authentication path algorithm can be found in [16]. Exactly H computation units are spent in each round of the algorithm, so the computing time is in $\mathcal{O}(H)$. Szydlo shows that the maximum space needed with $3H - 2$ is likewise algorithmic in the total number of signatures (since $H = \log_2(N)$).

As an interesting concern, Szydlo proves that the bounds of $\mathcal{O}(H)$ for both time and space complexity he found are optimal. It is impossible to find an authentication path algorithm that is in both better than $\mathcal{O}(\log_2(N))$. It is clear that at least $H-2$

nodes have to be stored. So it suffices to show that if an algorithm needs a storage capacity of $\mathcal{O}(\log_2(N))$, then at least $\mathcal{O}(\log_2(N))$ computation units per round are required. A trade-off between time and space bounds can always be found, as no constants are given. But the complexity bounds of $\mathcal{O}(\log_2(N))$ for both at the same time are hard.

3.3 Drawbacks of Former Algorithms

All known work on traversal algorithms consider the leaf-calculation and the hash-function evaluation to require the same amount of computation. Both operations are counted as one computation unit each. When applying a one time signature scheme for the leaf calculation, many hash value computations are needed to generate a single leaf, i.e. up to thousands. One can expect that leaf-calculation is much more expensive considering the computation time needed than a single hash-function evaluation. This leads to the problem that one cannot predict the number of hashes really needed during one step of the authentication path algorithms. So the generation time of a signature varies enormously from round to round.

Szydlo's algorithm is the one that provably allows the best time and memory properties. Using H stacks which store at most H nodes each, the maximum number of nodes stored is in $\mathcal{O}(H^2)$. He shows that the memory needed for the stacks is at most H, so that all other memory spaces are not needed at once. But implementing this algorithm on a platform without dynamic memory allocation would need the complete space of $\mathcal{O}(H^2)$, as space for all H^2 nodes has to be reserved.

These ideas were included constructing the new traversal algorithm (Algorithm 3) which is presented in the next section. Counting the number of hashes and the number of leaf calculations separately leads to more balanced timings. Furthermore we show that it is possible that all treehash instances share one single stack, so that the storage needed is bounded linearly in H even on system without dynamic memory allocation.

4 A New Authentication Path Algorithm

This section introduces a new algorithm for Merkle tree authentication path computation. It is an improvement of Szydlo's preprint algorithm [16] and addresses it's drawbacks mentioned in section 3. The correctness of the new algorithm will be proved below. Further, some calculations on runtime and storage requirements are made for comparison with former algorithms. This section presents the theoretical results, whereas practical results are given later in section 6.

Our new algorithm will allow a time-memory trade-off. In the key generation phase the whole Merkle tree has to be computed completely once. In this phase the first authentication path was stored as input for Merkle's and Szydlo's scheduling algorithms. Now we are going to store some more nodes: as the computation of right nodes near to the root is most expensive, the idea is to store those right nodes from the beginning, so that the time to compute these nodes is saved later. The parameter K denotes the number of top layers in the tree where all right nodes are stored. K is chosen so that $H - K$ is even (we perform $(H - K)/2$ steps per round).

As mentioned above, our new algorithm yields to a more balanced signature generation time and also a moderate space requirement. Clearly the logarithmic bounds in space and time complexity shall be maintained. We will show that an amount of less than $H/2$ leaf calculations per round are sufficient to compute authentication paths and that storage is also bounded logarithmically in the number of leaves.

We use stacks that are slightly different from the ones used by Szydlo. For each height we apply a structure TREEHASH, which computes the upcoming right nodes for the authentication path (again using Algorithm 1). All these instances share one single stack, whereas in former algorithms every instance had its own stack to store nodes on. We achieve a logarithmic total number of nodes stored at once, also on devices without dynamic memory allocation. We will show that sharing a single stack for all TREEHASHs works well. Further on we are using a slightly modified scheduling of the computation of right nodes, so that the amount of $(H - K)/2$ leaf calculations per round are sufficient. The computation of right nodes changes, whereas left nodes are computed in the same manner as with Szydlo's algorithm.

4.1 Notation

The main part of the notation is already known from previous sections. H denotes again the height of the Merkle tree. With $y_h[j]$ the jth node on height h ($i = 0 \dots H, j = 0 \dots 2^{H-h} - 1$) is referred. The authentication path of the current leaf φ is again $\text{AUTH}_0, \dots, \text{AUTH}_{H-1}$. The values $\text{KEEP}_0, \dots, \text{KEEP}_{H-2}$ are the same certain nodes to quickly compute left children. With τ we denote the height of the first parent of the actual leaf φ which is a left node. The stacks to store the right nodes near to the root (on the upper K levels, $K \geq 2$) are called $\text{RETAIN}_h (h = H - 2 \dots H - K)$. They are filled from left to right, so that the top node of a RETAIN stack is always the next one needed for the authentication path.

Again we use an oracle LEAFCALC which computes the leaf value of the leaf with the omitted index. Using the Merkle tree for digital signatures this method computes the verification key of the underlying one time signature scheme. In difference to former algorithms we do not just count it as one computation unit, we will distinguish between the computation of leaves and single hash evaluations.

4.1.1 Treehash Stacks

With $\text{TREEHASH}_0 \dots \text{TREEHASH}_{H-K-1}$ we denote the structures to compute right children. Each such instance stores the first node itself, further nodes are pushed on the commonly shared stack. A node stored on a TREEHASH stack is called a *tail* node. Additionally to the push and pop operations each treehash stack is equipped with three methods: initialize(), height() and update(). The method TREEHASH_h.initialize() sets the start node which tells the TREEHASH with which leaf the computation of the stack has to begin. TREEHASH_h.height() returns the height of the lowest node stored in the instance, either on the stack or in TREEHASH_h itself. This method is required for the scheduling of the $(H - K)/2$ LEAFCALC operations, which are always assigned to the one instance with the lowest tail node. If there is more than one instance with same lowest node height, the one with the lowest index is chosen for the update. In order to skip instances that are already finished or not yet initialized, TREEHASH_h.height() is set to infinity in these cases. When the

treehash stack was initialized, the first call of the last method TREEHASH_h.update() calculates the leaf with the start index. It is stored in the instance itself. The next updates work in analogy to Algorithm 1: the next leaf is calculated and stored on the stack. If the top two nodes have the same height they are hashed together and the result is pushed on the stack. If the top node on the stack and the first node in the treehash have same height, the result replaces the first node of the treehash.

4.2 Algorithm Description

4.2.1 Initialization

During the key generation, we store certain nodes of the Merkle tree. First we store the authentication path for the first leaf $\varphi = 0$:

$$\text{AUTH}_h = y_h[1], \qquad h = 0 \dots H-1$$

We also store the next right authentication node for each height $h = 0 \dots H-K-1$ on the stacks

$$\text{TREEHASH}_h.\text{push}(y_h[3]), \qquad h = 0 \dots H-K-1$$

Depending on K, we store all right authentication nodes on the retain stacks:

$$\text{RETAIN}_h.\text{push}(y_h[2j+3]), \qquad h = H-K \dots H-2, \qquad j = 2^{H-h-1}-2 \dots 0$$

Figure 11 illustrates the initialization process.

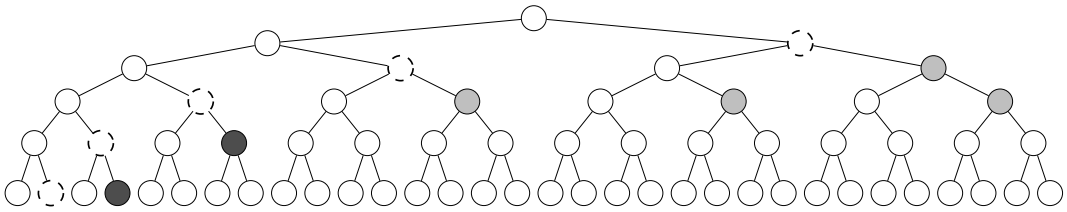

Figure 11: Values of the initialization, $H = 5, K = 3$. The dashed nodes are authentication nodes, the black ones are stored in treehash, the grey nodes are kept in retain stacks.

4.2.2 Authentication Path Computation

As input, Algorithm 3 requires a node index φ between 0 and $2^H - 2$ and the actual algorithm state (which means the TREEHASH instances, the stack and the authentication path of the current leaf). As output, it returns the authentication path of the next leaf $\varphi + 1$.

Left node computation. The first steps are again handling the left node computation. The value τ is the height of the first parent node of leaf φ which is a left node, remember $\tau = 0$ if the current leaf itself is a right node. In formula we have $\tau = \max\{h : 2^h \,|\, (\varphi + 1)\}$. For left node computation the current AUTH node on height τ is stored in KEEP_τ if $\lfloor \varphi / 2^{\tau+1} \rfloor$ is even (which means that the parent of φ on height $\tau + 1$ is a left node). This node is required in round $\varphi + 2^\tau$ for the next authentication node on height $\tau + 1$, which can then be computed as $\text{HASH}(\text{AUTH}_\tau \| \text{KEEP}_\tau)$. See figure 12 of an example for the left node computation.

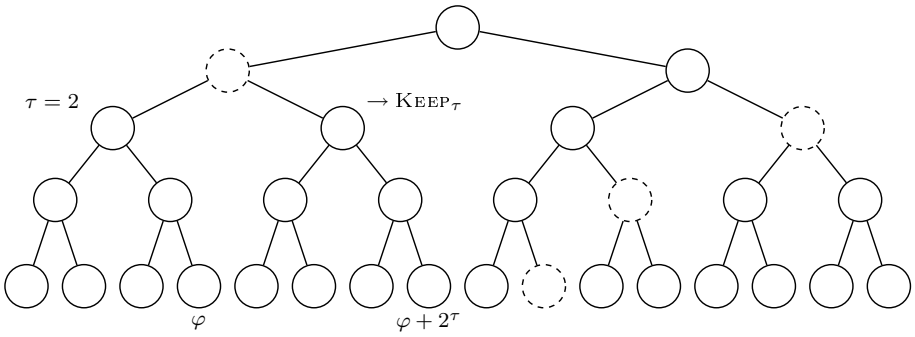

Figure 12: In round φ the node AUTH_τ is stored in KEEP_τ. This node is needed in round $\varphi + 2^\tau$ for the computation of its parent node, which is part of the dashed authentication path computed in round $\varphi + 2^\tau$.

If $\tau = 0$, which occurs in the rounds where leaf φ is a left node itself, we use $\text{LEAFCALC}(\varphi)$ to compute the leaf value itself. This node is the lowest authentication node for the next round, i.e. $\text{AUTH}_0 = \text{LEAFCALC}(\varphi)$.

Considering this, the computation of the left authentication node requires either one hash function call (if φ is a right node) or one LEAFCALC operation (in case that φ is even).

Right node computation. If the new left node for the authentication path was computed, the values AUTH_h for $h = 0 \dots \tau - 1$ must change as well (compare figure 10 on page 37). The required nodes were either stored on the RETAIN stacks (for all $h \geq H - K$) or they were precomputed in the TREEHASH instances. So the call RETAIN_h.pop respective the call TREEHASH_h.pop delivers the newly needed authentication nodes on heights lower than τ. In section 4.3 we will show that every treehash in fact is completed when its top node is needed.

If an AUTH node was taken from a TREEHASH instance, it must be reinitialized for the precomputation of the next right nodes. The instance with height h must again be completed in round $\varphi + 2^{h+1}$. For that the instance is initialized with start index $\varphi + 1 + 3 \cdot 2^h$. An illustration of this process can be seen in Figure 13.

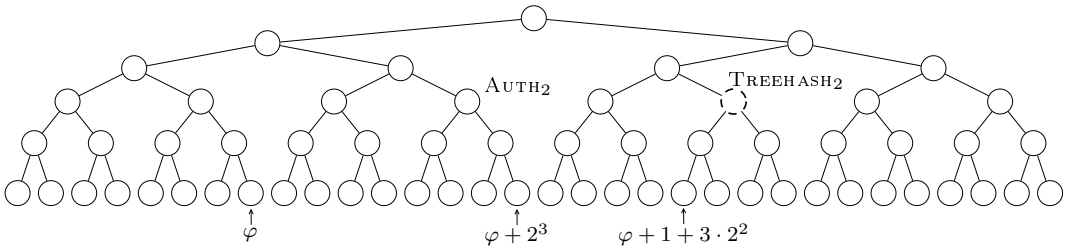

Figure 13: In round φ the node AUTH_2 is popped from TREEHASH_2. This instance is then initialized anew with start index $\varphi + 1 + 3 \cdot 2^2$ and computes the declared right node on height 2. This node is needed in round $\varphi + 2^3$.

The next step is the scheduled computation of the remaining $(H - K)/2$ computations of LEAFCALC operations. We use the same scheduling as Szydlo did: the TREEHASH instance with the lowest tail node on his top gets the current update. If more than one instances have tail nodes at the same minimal height we choose the one which has the lowest index.

The last step of the algorithm is the output of the authentication path $\{\text{AUTH}_h : h = 0 \dots H - 1\}$. Algorithm 3 shows the complete algorithm listing.

Algorithm 3 Improved Authentication Path Computation

Input: $\varphi \in \{0, \ldots, 2^H - 2\}$, H, K and the algorithm state.
Output: Auththentication path for leaf $\varphi + 1$

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

4.3 Correctness of the Algorithm

This subsection proves that the new authentication path algorithm works correctly. First we will show that the amount of $(H-K)/2$ LEAFCALC operations per round is sufficient for computation of the right authentication nodes, which means that each treehash instance is ready when needed.

Lemma 1. *In Algorithm 3 every right node is completed in time.*

Proof. In this proof we show that every TREEHASH instance is definitely completed when its top node is required for the authentication path.
On height h we need 2^h LEAFCALC-operations and $2^h - 1$ hash value operations to complete TREEHASH_h. When TREEHASH_h is initialized in round φ, the authentication node on height h computed by this instance is needed in round $\varphi + 2^{h+1}$. So there is an amount of 2^{h+1} rounds until TREEHASH_h must be completed. In each round we perform $(H-K)/2$ LEAFCALC-operations. Our total is $\frac{H-K}{2} \cdot (2^{h+1}) = (H-K) \cdot 2^h$ operations to spend before the treehashs top node is required. The chart of Table 1 shows which TREEHASH instances can be computed during the computation of TREEHASH_h and what costs they need.

TREEHASH	Quantity	LEAFCALC-ops each
TREEHASH_{H-K-1}	1	$\max 2^h$
$\vdots$	$\vdots$	$\vdots$
TREEHASH_{h+1}	1	$\max 2^h$
TREEHASH_h	1	2^h
TREEHASH_{h-1}	2	2^{h-1}
$\vdots$	$\vdots$	$\vdots$
TREEHASH_{h-j}	2^j	2^{h-j}
$\vdots$	$\vdots$	$\vdots$
TREEHASH_0	2^h	1

Table 1: Number of LEAFCALC operations

As shown in the Table 1, active TREEHASH instances on higher levels than h can apply at most 2^h LEAFCALC operations each (the total cost of completing a stack on height h). Before they were continued on higher levels, TREEHASH_h must have been completed. There are $H - K - 1 - h$ exemplars of higher instances (indices $h+1 \ldots H-K-1$). The computation of a lower instance TREEHASH_{h-j} with index $j \in \{1 \ldots h\}$ requires 2^{h-j} LEAFCALC operations. During the available 2^{h+1} rounds TREEHASH_{h-j} is initialized 2^j times.

Summing up the number of the maximal count of LEAFCALC operations, we get less than $(H - K - 1 - h) \cdot 2^h$ for the stacks with index higher than h and

$$(h+1) \cdot 2^j \cdot 2^{h-j} = (h+1) \cdot 2^h$$

for the stacks with index less or equal h (down to 0). Totally we get at most

$$(H - K - 1 - h) \cdot 2^h + (h+1) \cdot 2^h = (H - K) \cdot 2^h$$

This is an upper bound for the maximum number of LEAFCALC operations performed until TREEHASH_h must be completed. As we have seen above we have a total amount of $(H - K) \cdot 2^h$ LEAFCALC operations. So we determine that every stack is completed when its top node is needed in the algorithm. The upper bound is tight for $h = H - K - 1$. □

In his algorithm Szydlo uses one stack for each height $h = 0 \ldots H - 1$. In our new algorithm all TREEHASH instances share one single stack. For the correctness of the algorithm we have to show that sharing one single stack really works.

Lemma 2. *In Algorithm 3 it is sufficient to share one single stack for all* TREEHASH *instances.*

Proof. We have to show that tail nodes belonging to different TREEHASH instances do not interfere on the stack. If TREEHASH_h gets an update and has previously stored nodes on the stack, we have to show that these nodes lie on top of the stack.

First we consider TREEHASH instances with index greater than h. When TREEHASH_h receives its first update, the lowest tail node of higher TREEHASH instances has a

height of at least h. That implies that TREEHASH$_h$ is completed before those instances get another update. So TREEHASH$_h$ and instances on higher levels never interfere on the shared stack.

Let us now examine lower TREEHASH instances. It is possible that TREEHASH$_i$ with index $i < h$ gets updates and stores nodes on the stack while TREEHASH$_h$ is not completed and stores tail nodes on the stack. This can happen only if the lowest tail node of TREEHASH$_h$ has height greater or eqal i. But in this case TREEHASH$_i$ is completed before TREEHASH$_h$ gets another update, and the top nodes on the stack are again the tail nodes of TREEHASH$_h$. We have shown that lower TREEHASH instances do not interfere with the tail nodes of TREEHASH$_h$, and so the proof is completed. □

4.4 Computational Bounds

Lemma 3. *Algorithm 3 needs* $(H - K)/2 + 1$ *many* LEAFCALC *operations per round. The number of performed hash value evaluations per round is bounded by* $\frac{3}{2}(H - K - 1) + 1$. *Therefore the total computation cost of Algorithm 3 lies in* $\mathcal{O}(\log_2 N)$.

Proof. **LEAFCALC operations.** In step 3 of our algorithm one LEAFCALC operation is performed, if φ is a left node. In step 5 at most $(H - K)/2$ calculations are executed. Totally we have at most $(H - K)/2 + 1$ LEAFCALC operations.

Hash operations. Now we give an upper bound for the number of hash calculations performed in one round. Let $u = \frac{H-K}{2}$. We will show that the maximum number of hash evaluations is performed in the following case: the instance TREEHASH$_{H-K-1}$ receives all u updates and is completed by the last one of these updates.
We will now give an upper bound for the number of hashes required in this worst case. On height 0 every second round a hash is required. Every fourth round one additional hash is required on height 1. Generally on height h every 2^{h+1}th round an additional hash is performed $(h = 0 \dots \lceil u/2^h \rceil - 1)$.
Since we have at all u updates to perform, on height 0 we get totally $\lceil u/2 \rceil$ hashes,

on height 1 there are additional $\lceil u/4 \rceil$ hashes and generally for height h we have to add $\lceil u/2^{h+1} \rceil$ hashes, which makes totally

$$\sum_{h=0}^{\lceil \log_2 u \rceil - 1} \left\lceil \frac{u}{2^{h+1}} \right\rceil = \sum_{h=1}^{\lceil \log_2 u \rceil} \left\lceil \frac{u}{2^h} \right\rceil$$

The last update requires $H - K - 1 = 2u - 1$ hashes to complete TREEHASH_{H-K-1} up to height $H - K - 1$. So far only $\lceil \log_2 u \rceil$ of these hashes were considered, so we have to add $2u - 1 - \lceil \log_2 u \rceil$ hash value evaluations. In total for the worst case we get the following upper bound for the number of hashes required for one round:

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

Using the geometric series it is

$$\sum_{h=1}^{\lceil \log_2 u \rceil} \frac{1}{2^h} = \frac{(1/2)^{\lceil \log_2 u \rceil + 1} - (1/2)}{1/2 - 1} = -2 \cdot ((1/2)^{\lceil \log_2 u \rceil + 1} - 1/2) = 1 - \frac{1}{2^{\lceil \log_2 u \rceil}}$$

Additionally it is

$$-\frac{1}{2^{\lceil \log_2 u \rceil}} \leq -\frac{1}{2^{\log_2 u + 1}} = -\frac{1}{2 \cdot 2^{\log_2 u}} = -\frac{1}{2u} = -\frac{1}{H - K}$$

Including this we get

$$(3) \leq (1 - (\frac{1}{H-K})) \frac{H-K}{2} + H - K - 1 = \frac{H-K}{2} - \frac{1}{2} + H - K - 1 = \frac{3}{2}(H - K - 1)$$

One additional hash is performed in step 4a of Algorithm 3. This leads to the maximum of $\frac{3}{2}(H - K - 1) + 1$ hashs per round. What remains now is to show

that there is no other case that requires more hash evaluations, so that the above mentioned case is indeed the worst case.

If a treehash instance on height less than $H - K - 1$ receives all updates and is completed in this round, less than (3) hashes are required. The same holds if the treehash instance receives all updates but is not completed in this round.
The last case to consider is the one where the u available updates are spent on treehash instances on different heights. If the active treehash instance has a tail node on height j, it will receive updates until it has a tail node on height $j + 1$, which requires 2^j updates and 2^j hashes (so $2^j < u$, otherwise again only one treehash instance would receive updates). First consider the case that the active treehash instance is not completed by the u updates. Additional to the 2^j hashes there can be $t \in \{0 \dots H - K - j - 2\}$ hashes which take nodes from the stack, as on the stack nodes on heights $j + 1 \dots H - K - 3$ could be stored. Then the next treehash instance worked on has a tail node on heights j or $j + 1$ ($> j + 1$ is not possible, otherwise the old treehash instance would get the next updates again, $< j$ is not possible because then this treehash instance would have gotten the updates earlier) and it cannot store nodes on the stack on heights $\leq j + t$ (on each height at most one node is stored on the stack). But this is the same case which appears in the above mentioned worst case if it computes a node on height j and gets the next updates for the same instance. The last case to consider is the case where the active treehash instance is completed by the first 2^j updates and hashes. Again it is possible that $t \in \{0 \dots H - K - j - 2\}$ hashes are additionally needed for nodes on the stack. Then the next active treehash instance has a tail node on height $\geq j$, and on the stack there can only be nodes with height at least $j + t + 1$. Again this case appears in our worst case scenario, as it makes no difference if the same instance receives the next update or another one. So we could show that all other cases can be reduced to the worst case and this bound was given above.

Considering the bounds of $(H - K)/2 + 1$ LEAFCALC operations and $\frac{3}{2}(H - K - 1) + 1$ hash evaluations per round it is easy to see that the computation costs of our algorithm is bounded linearly in H. Since $H = \log_2 N$ the cost is logarithmically in the number of leaves N, so that it lies in $\mathcal{O}(\log_2 N)$. □

4.5 Storage Efficiency

Lemma 4. KEEP *consists of at most* $\lfloor H/2 \rfloor + 1$ *node values. For the upper* $K-1$ RETAIN *stacks* $2^K - K - 1$ *nodes are stored. On the shared stack at most* $H - K - 1$ *nodes are stored. Therefore the total space required by Algorithm 3 is bounded by* $3H + \lfloor H/2 \rfloor + 2^K - 3K - 1$.

Proof. **Space requirements for KEEP nodes.** Consider that in step 2 of Algorithm 3 a node gets stored in KEEP_h ($h = 1 \dots H-2$). Then the node in KEEP_{h-1} is removed in the same round in step 4a.

Next we will show that if a node is stored in KEEP_h, $h = 0, \dots, H-3$, then KEEP_{h+1} is empty. A node is stored in KEEP_{h+1} in rounds $\varphi \in A_a = \{2^{h+1} - 1 + a \cdot 2^{h+3}, \dots, 2^{h+2} - 1 + a \cdot 2^{h+3}\}, a \in \mathbb{N}_0$. In rounds $\varphi' = 2^h - 1 + b \cdot 2^{h+2}, b \in \mathbb{N}_0$, a node gets stored in KEEP_h. We will show that $\varphi' \notin A_a$. Assume

$$(4) \qquad \varphi' \in A_a \quad \Leftrightarrow \quad \underbrace{\varphi' \geq 2^{h+1} - 1 + a \cdot 2^{h+3}}_{(4.1)} \text{ and } \underbrace{\varphi' \leq 2^{h+2} - 1 + a \cdot 2^{h+3}}_{(4.2)}$$

$$\begin{aligned}
(4.1) \qquad & \varphi' && \geq && 2^{h+1} - 1 + a \cdot 2^{h+3} \\
\Leftrightarrow \quad & 2^h - 1 + b \cdot 2^{h+2} && \geq && 2^{h+1} - 1 + a \cdot 2^{h+3} \\
\Leftrightarrow \quad & 1 + b \cdot 2^2 && \geq && 2^1 + a \cdot 2^3 \\
\Leftrightarrow \quad & 4b && \geq && 1 + 8a \\
\Leftrightarrow \quad & b && \geq && 1/4 + 2a
\end{aligned}$$

$$\begin{aligned}
(4.2) \qquad & \varphi' && \leq && 2^{h+2} - 1 + a \cdot 2^{h+3} \\
\Leftrightarrow \quad & 2^h - 1 + b \cdot 2^{h+2} && \leq && 2^{h+2} - 1 + a \cdot 2^{h+3} \\
\Leftrightarrow \quad & 1 + b \cdot 2^2 && \leq && 2^2 + a \cdot 2^3 \\
\Leftrightarrow \quad & 4b && \leq && 3 + 8a \\
\Leftrightarrow \quad & b && \leq && 3/4 + 2a
\end{aligned}$$

So $\varphi' \in A_a$ is equivalent to $1/4 + 2a \leq b \leq 3/4 + 2a$. Since $a \in \mathbb{N}_0$ this is a contradiction to $b \in \mathbb{N}_0$. That shows that KEEP_{h+1} is always empty when KEEP_h gets a node to store.

We have shown that if a node gets stored in KEEP_h, then KEEP_{h+1} is empty and KEEP_{h-1} gets removed in the same round. So at most every second KEEP stores a node at the same time, totally we have to store less then $\lfloor H/2 \rfloor$ nodes. Between steps 2 and 4a of Algorithm 3 we need to store one temporary node, what gives us a total space requirement of $\lfloor H/2 \rfloor + 1$ for the KEEP nodes.

Nodes stored in RETAIN. In the highest $K-1$ RETAIN stacks all right nodes are stored. During the initialization, for heights $H-K, \ldots, H-2$, the nodes $y_i[2j+3]$ for $j = 2^{H-i-1} - 2$ down to 0 are stored. This makes totally

$$\sum_{i=H-K}^{H-2} (2^{H-i-1} - 1) = \sum_{i=-K}^{-2} (2^{-i-1} - 1) = \sum_{i=2}^{K} (2^{i-1} - 1) = \sum_{i=0}^{K-2} (2^{i+1}) - (K-1)$$

Using the geometric series we have

$$\sum_{i=0}^{K-2} 2^i = \frac{2^{K-1} - 1}{2 - 1} = 2^{K-1} - 1$$

Including this, we get

$$2 \cdot (2^{K-1} - 1) - K + 1 = 2^K - K - 1$$

This is the storage needed for the highest $K-1$ RETAIN stacks where all right nodes are stored.

Nodes stored on the stack. We will show that at most one tail node can be stored on each height $h = 0 \ldots H-K-3$. An instance TREEHASH_h stores at most h tailnodes. While the first one is stored in TREEHASH_h itself, the remaining $h-1$ nodes are pushed on the stack. Additionally one temporary node could be stored short before the top nodes on the stack are hashed together to a higher node.

When TREEHASH_h gets active and receives its first update, all lower instances with height less than h are either completed or not initialized. Otherwise the height of such an instance would be less than h and it would have received updates before TREEHASH_h did. For the same reason, instances with index $> h$ can only store nodes on height greater than h, or they are as well either completed or empty. Consider the case that an instance on height i stores a node on the stack. Then all other

TREEHASH instances on heights $> i$ can only store nodes on height $\geq i$, because otherwise TREEHASH$_i$ would not have received updates. And since TREEHASH$_i$ can only store nodes up to height $i-1$ on the stack we have seen that there can never be two nodes with the same height stored on the stack.

The instance TREEHASH$_{H-K-1}$ is the one with the highest index. It stores nodes up to height $H-K-2$, where nodes on height $0 \dots H-K-3$ can be stored on the stack (the first one is stored in TREEHASH$_{H-K-1}$ itself). This is the case in round $\varphi = 2^{H-K+1}-2$, the round where the update that completes TREEHASH$_{H-K-1}$ is performed. Considering the temporary node created by LEAFCALC we get a total bound of $H-K-1$ of the nodes stored on the stack.

Space requirements in total. On each height $h \in \{0 \dots H-1\}$ there is always one authentication node stored. For this reason, the space needed for authentication nodes is H. Each of the $H-K$ TREEHASHs saves one node. Summing up gives us totally

$$\begin{aligned} &H + (\lfloor H/2 \rfloor + 1) + (2^K - K - 1) + (H - K - 1) + (H - K) \\ &= 3H + \lfloor H/2 \rfloor + 2^K - 3K - 1 \end{aligned}$$

□

Since the space requirements are exponential in K, this parameter should be chosen small. The following chart shows the size of the RETAIN stacks corresponding to the value K. The number of 720 nodes is a big amount and should never be needed for the retain nodes. So K could be chosen 2 if H is even or 3 if H is odd.

K	2	3	4	5	6	7
Size of RETAIN	1	4	11	26	57	120

Table 2: Total number of node stored in RETAIN ($2^K - K - 1$)

4.6 Computing Leaves using a PRNG

For the computation of each leaf of the Merkle tree a random number SEED_φ is required. Out of this seed the keys of the one time signature are created. Remember that these SEEDs are computed consecutively using the forward secure PRNG:

$$\text{SEED}_{\varphi+1} \leftarrow \text{PRNG}(\text{SEED}_\varphi)$$

In the flow of authentication path computation not only consecutive leaves are computed: for upcoming right nodes computed by the TREEHASHs some future nodes are required as well. It would be very inefficient to compute every SEED value in time when it is needed: the maximum number of PRNG calls would be $3 \cdot 2^{H-K-1}$ which occurs when TREEHASH_{H-K-1} gets its first update. A special scheduling for these seeds has to be implemented to distribute the calls to the PRNG.

Our proposed scheduling strategy requires $H-K$ calls to the PRNG each round. We have to store two seeds for each height $h = 0, \ldots, H-K-1$. The first (SEEDACTIVE) is used to successively compute the leaves for the authentication node currently constructed by TREEHASH_h and the second (SEEDNEXT) is used for upcoming right nodes on this height. SEEDNEXT is updated using the PRNG in each round. During the initialization, we set $\text{SEEDNEXT}_h = \text{SEED}_{3 \cdot 2^h}$ for $h = 0, \ldots, H-K-1$. In each round, at first all seeds SEEDNEXT_h are updated using the PRNG. If in round φ a new treehash instance is initialized on height h, we copy SEEDNEXT_h to SEEDACTIVE_h. In that case $\text{SEEDNEXT}_h = \text{SEED}_{\varphi+1+3 \cdot 2^h}$ holds and thus is the correct seed to begin computing the next authentication node on height h.

4.7 Comparison of Theoretical Bounds

This section is terminated with a comparison of the theoretical bounds of the former discussed authentication path algorithms. Table 3 shows the composition in short.

Comparing our new algorithm to Szydlo's, the computation time needed per round seems to increase. The difference is the distinction between leaf calculations and simple hashes. As the number of LEAFCALC operations with Szydlo's algorithm

could grow up to H, at most $(H-K)/2+1$ are done with our scheduling. With this the maximum number of hashes to perform per round is more balanced and in total lower using the new algorithm.

Algorithm	Computation Time	Space
Merkle	$2\log_2(N)-2$	$1/2(\log_2(N))^2$
Jakobsson et. al.	$2\log_2(N)/\log_2(\log_2(N))$	$1.5(\log_2(N))^2/\log_2(\log_2(N))$
Szydlo	$\log_2(N)$	$3\log_2(N)-2$
Algorithm 3	$\frac{3}{2}(\log_2(N)-K-1)+1$ $+(\log_2(N)-K)/2+1$	$3.5\log_2(N)+2^K-3K-1$

Table 3: Comparison of complexity bounds. In concern of computation time, Algorithm 3 distinguishes between hash function evaluations (first row) and leaf calculations (second row)

Concerning the memory, we first saved half of the KEEP nodes, as only every second has to be stored at once. The parameter K provides a time-memory trade-off for our algorithm. For the stacks the bound we found is really tight, whereas for Szydlo's algorithm this is only true if dynamic memory allocation is possible. Otherwise the space needed could grow quadraticly in H.

These theoretical results are analyzed in section 6 of this thesis by some practical work.

5 Java Implementation

5.1 Overview

As one part of this thesis, the GMSS signature scheme was implemented using the Java programming language. Responsible for the use of cryptographic algorithms in Java applications is the Java Cryptography Architecture (JCA) [24]. It is the Java security API, providing standardized programming interfaces for message digests, digital signatures, key exchange or cyphers for use with all Java applications. As it is an API, it strictly separates the implementation of algorithms from their usage. Some interfaces are required out of a Java Cryptography Extension (JCE), which is a part of the the Java Platform.

GMSS was integrated into the FlexiProvider package [25], which is an open source cryptographic service provider for the JCA. A provider for the JCA has two functions: it administrates the implementation of the cryptographic algorithms and it is responsible for the assignment of algorithms to their names. The FlexiProvider contains modules for integration into any application built on top of the JCA. As GMSS is topic of the post quantum computing research, it was implemented as part of the FlexiPQCProvider, which contains algorithms secure against quantum computer attacks. The FlexiProvider includes established algorithms like RSA or DSA as well as algorithms that are still research topics, like GMSS.

As the JCA provides interfaces, it allows the simple exchange of cryptographic algorithms. For this GMSS can easily be integrated into other applications based on the JCA. GMSS was implemented so that the underlying message digest algorithm (used for the OTSS and the Merkle tree) can be exchanged easily. So the FlexiProvider implementation will stay secure if a message digest algorithm drops out. Some predefined versions of GMSS (using the hash functions SHA1, SHA224, SHA256, SHA384 and SHA512) can be integrated into applications by using some predefinded object identifiers (OIDs). Those OIDs assigned to GMSS can be found in Appendix D. The complete source code can be found as download on the website of the FlexiProvider project [25].

For the implementation of GMSS the JCE of Fraunhofer Gesellschaft (FhG) was used. For encoding and decoding of an ASN.1 representation of the GMSS keys, the ASN.1 codec package provided by sourcefourge.net[2] was imported. Both packages can also be found via the FlexiProvider website. ASN.1 stands for *Abstract Syntax Notation One*. It is a description language for the definition of data structures, standardized by the ITU-T [26]. It is used for interoperability with other applications. Using this notation it is for example commonly possible to use the GMSS keys for X.509 certificates.

A former Java implementation of GMSS already existed [17]. The main drawback of this work was the implementation of the authentication path algorithm. It used the Szydlo algorithm for the scheduling of the authentication path computation. The absence of a seed scheduling was the first fact slowing down the computation. But even worse was the fact that each stack was computed at once, which meant the computation of 2^h leaves at once. The distributed computation of these leaves in combination with the more balanced authentication path algorithm of section 4 balances the whole signature generation time.

The GMSS parameter set was upgraded: now it contains additionally the K values for each layer of the GMSS structure. In summary the GMSS parameter set $\mathcal{P}$ is now

$$\mathcal{P} = (T, (h_1, \ldots, h_T), (w_1, \ldots, w_T), (K_1, \ldots, K_T))$$

For the application of the new authentication path algorithm with GMSS a data structure for the treehash instances is required. For this the class `Treehash` was implemented. It stores the first node itself and uses a shared stack for the storage of additional tail nodes. The update method of this class executes the treehash algorithm (Algorithm 1) once.

There are more nodes being precomputed in the new implementation. Not only the leaves of the tree after the following are precomputed, but as well those nodes needed for the actual tree. This additional distributed leaf computation is shown in the next section.

[2]SourceForge.net is one of the most famous Open Source software collection, available at `http://sourceforge.net`

5.2 Distributed Node Computation

The former GMSS implementation distributed the calculation of the next leaf in $\mathcal{T}_{i,j+2}$, the tree after the following of the currently processed tree $\mathcal{T}_{i,j}$. For the actual tree, every leaf is calculated at once. The idea is now to distribute the computation of those leaves as well over the pass of the underneath tree $T_{i+1,j}$. This tree consists of $2^{h_{i+1}}$ leaves, thus the computation of upper nodes is distributed over $2^{h_{i+1}}$ steps. Distributed generation of a leaf means the computation of the OTS key corresponding to the leaf. Using the Winternitz OTS scheme each random value x_i is hashed $2^w - 1$ times ($i = 1 \dots t_w$) to get the values y_i. The concatenation of these values is hashed once to get the OTS public key Y. For creation of every random value x_i one hash is required. So the total number of hash function calls is $(2^w - 1) \cdot t_w + 1 + t_w$. For each of the $2^{h_{i+1}}$ leaves of the underneath tree we get an amount of

$$\left\lceil \left((2^w - 1) \cdot t_w + 1 + t_w\right) \;/\; 2^{h_{i+1}} \right\rceil$$

This is the number of hashes performed per round, so that after $2^{h_{i+1}}$ rounds the leaf is completed. In the implementation, the class `GMSSLeaf` already existing adopts this distributed computation. A detailed description of this class can be found in [17].

Actual Processed Nodes. The first leaf to precompute is the following leaf on the actual layer i which is needed when in the layer beneath a next tree is begun. The next leaf of tree $\mathcal{T}_{i,j}$ is partly computed when advancing a leaf in the lower tree $\mathcal{T}_{i+1,j}$ (see Figure 14). On the lowest layer $H - 1$ each leaf has to be computed at once, no distribution is possible (as no lower tree exists).

Another way to compute this leaf would be the verification of the signature belonging to the root of the following tree on the lower level. This one time signature is already known, it was precomputed out of its root value. For the next leaf of tree $\mathcal{T}_{i,j}$, the OTS public key belonging to the signature is required. It can be computed by just verifying the precomputed signature. On average half of the hashes for the leaf could be saved using this approach.

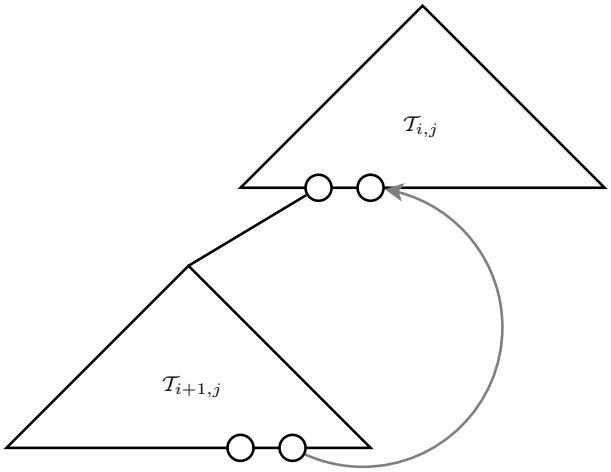

Figure 14: While advancing a leaf in tree $\mathcal{T}_{i+1,j}$, the next leaf of tree $\mathcal{T}_{i,j}$ is partly computed.

Treehash Nodes. Secondly the leaves needed for the authentication path algorithm, used for the computation of upcoming right nodes, can be precomputed. In round φ of tree $\mathcal{T}_{i,j}$ at maximum $(H-K)/2$ LEAFCALC operations have to be done for the authentication path algorithm. The leaves are directly passed to the treehash updates. On each layer besides the lowest one the calculation of these $(H-K)/2$ leaves is distributed equally over the flow of the $2^{h_{i+1}}$ leaves of the underlying tree $\mathcal{T}_{i+1,j}$. So while advancing

$$\frac{2^{h_{i+1}}}{(H-K)/2} = \frac{2^{h_{i+1}+1}}{H-K}$$

steps in the lower tree, one single leaf of the upper tree $\mathcal{T}_{i,j}$ is computed. Since every leaf requires an amount of $(2^w-1)\cdot t_w+1+t_w$ the total number of hashs to perform while advancing a leaf in $\mathcal{T}_{i+1,j}$ is

$$((2^w-1)\cdot t_w+1+t_w)\cdot(H-K) \;\;/\;\; 2^{h_{i+1}+1}$$

When all $2^{h_{i+1}}$ leaves of the lower tree were passed, all $(H-K)/2$ leaves needed for the treehash updates have been computed. Figure 15 depicts this precomputation process. The implementation of the treehash update process is described below, when handling the implementation of the authentication algorithm.

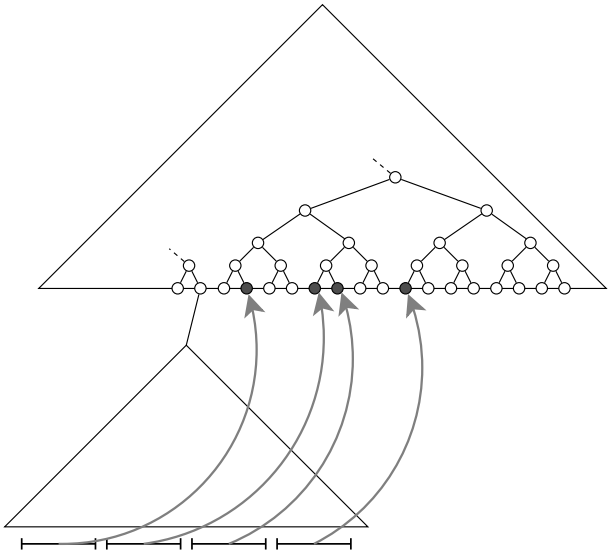

Figure 15: Suppose $(H - K)/2 = 4$, so that the four dark leaves of the upper tree are required for treehash updates. They are computed while advancing leaves in the lower tree.

Distributed Root Calculation. The implementation of the distributed precomputation of the root of tree $\mathcal{T}_{i,j+2}$ was changed as well. For the new authentication path algorithm additional values have to be computed, not only the root of the next but one tree is neccessary. The authentication path of the first leaf of tree $\mathcal{T}_{i,j+2}$ is stored in $\text{AUTH}_{\mathcal{T}_{i,j+2}}$. This is the standard approach in the MSS key generation, where the value of the first leaf of each height of the tree is stored. The third leaf of each height is again stored in $\text{TREEHASH}_{\mathcal{T}_{i,j+2}}$ and the upper nodes close to the root are stored in $\text{RETAIN}_{\mathcal{T}_{i,j+2}}$. All treehash instances share one single stack $\text{STACK}_{\mathcal{T}_{i,j+2}}$, which is stored as well. When advancing to the first leaf of $\mathcal{T}_{i,j+2}$ the authentication path computation will start with those stored values. The value $\text{ROOT}_{\mathcal{T}_{i,j+2}}$ is applied for the distributed signature generation, like it was described in section 2.5.2.

In Java the class `GMSSRootCalc` is responsible for the precomputation of the next but one tree; this class is also used in the keypair generator of GMSS for the computation of the first two trees of each GMSS layer. For this reason the implementation of the key pair generator was rewritten (and shortened because of the re-use of this part of code) in the new implementation.

5.3 Implementation of the Authentication Path Algorithm

Most of Algorithm 3 was implemented exactly following the algorithm description. The computation of τ, the storage of nodes in KEEP if necessary, the computation of left nodes and so on. The KEEP array was implemented in a way that each two consecutive levels share one entry of the array: nodes on layer h and $h-1$ are both stored in $\text{KEEP}_{\lfloor h/2 \rfloor}$. Temporarily in step 2 of Algorithm 3 the higher node is stored until step 4a was performed and the shared keep entry is surely empty. The LEAFCALC operation in the third step was replaced on upper layers: the leaf was already precalculated and must only be copied. The initialization of the treehash instances can be performed without committing the start index: the seed scheduling described at the end of the last section makes sure that the SEEDACTIVE is always the right seed belonging to the leaf $\varphi + 1 + 3 \cdot 2^h$ when restarting a treehash.

The most crucial part of the implementation is the update of the treehashs. As mentioned above, the computation of each of the $(H-K)/2$ leaves is distributed. This fact conditions that the update of the treehash is paused until all $\frac{2^{h_{i+1}+1}}{H-K}$ leaves of the lower tree $\mathcal{T}_{i+1,j}$ have been finished. So step 5a is computed partly for layer i when advancing one leaf in layer $i+1$.

Conclusion

As a result of these improvements, the new implementation provides more balanced time characteristics. The divergence in time needed for the generation of a signature is essentially smaller than before. This is achieved by application of the new, more balanced authentication path algorithm as well as by spending more attention to the distribution of upper tree computation. The next section shows some practical results which shall state this pronouncement.

6 Results

This section presents some results obtained using the new authentication path algorithm. First Algorithm 3 is compared to Szydlo's scheduling algorithm. The theoretical improvements of section 4 shall be confirmed using practical results. The second part gives some results of the revised GMSS scheme, compared to the previous GMSS implementation [7, 17] as well as other known schemes for digital signatures like DSA or ECDSA. At this juncture the size of keys and signatures as well as the time needed for key pair generation, signature generation, and verification respectively, are considered.

6.1 Comparison: Authentication Path Algorithm

Both authentication path algorithms were used for the pass of one single Merkle tree. The graphs of Figures 16 and 17 illustrate the number of hash evaluations needed for each round; the blue line is the result using Szydlo's algorithm, the red line used Algorithm 3. The leaves and hashes for left node computation were not considered, because both algorithms use the same procedure here. For the comparison a Winternitz parameter $w = 2$ and a 160 bit hash function was chosen. This leads to the cost of 256 hash function evaluations for one leaf calculation ($t_w = 85$ and we need $(2^2 - 1) \cdot t_w + 1 = 256$ hashes). Table 4 shows the statistical data belonging to the tests. H denotes the height of the Merkle tree.

	Mean Value		Standard Deviation	
H	Algorithm 3	Szydlo	Algorithm 3	Szydlo
5	214.9	405.4	95.8	263.0
10	899.9	1028	314.0	452.1

Table 4: Statistic data of the number of hashes required per round

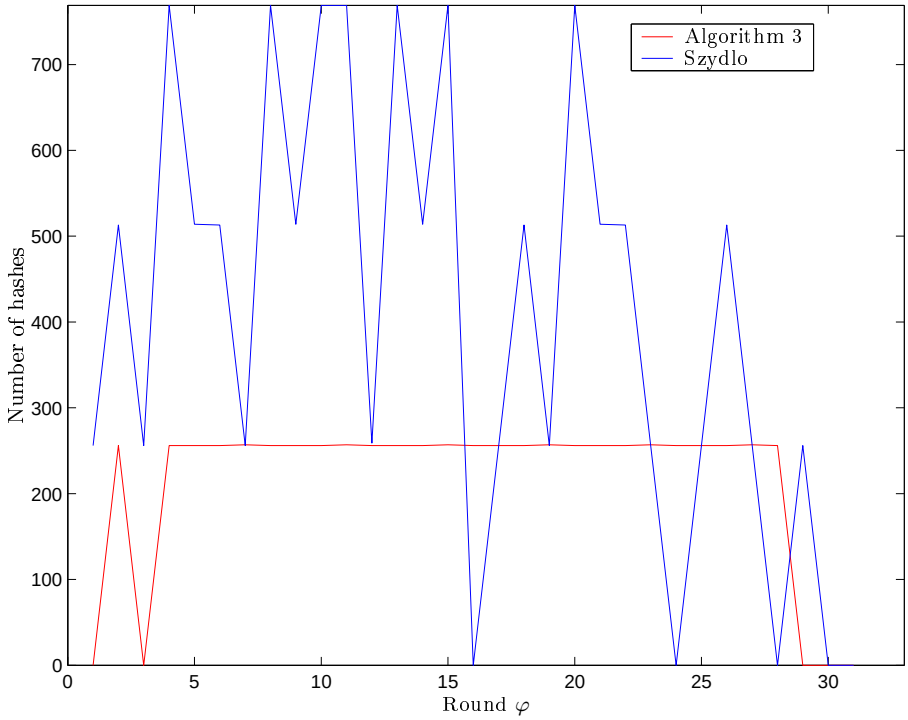

Figure 16: Number of hashes needed for right nodes per round while advancing one Merkle tree. On the x-axis the single rounds are assigned (tree height $H = 5 \Longrightarrow 2^5 = 32$ rounds), the y-axis shows the number of needed hash function evaluations.

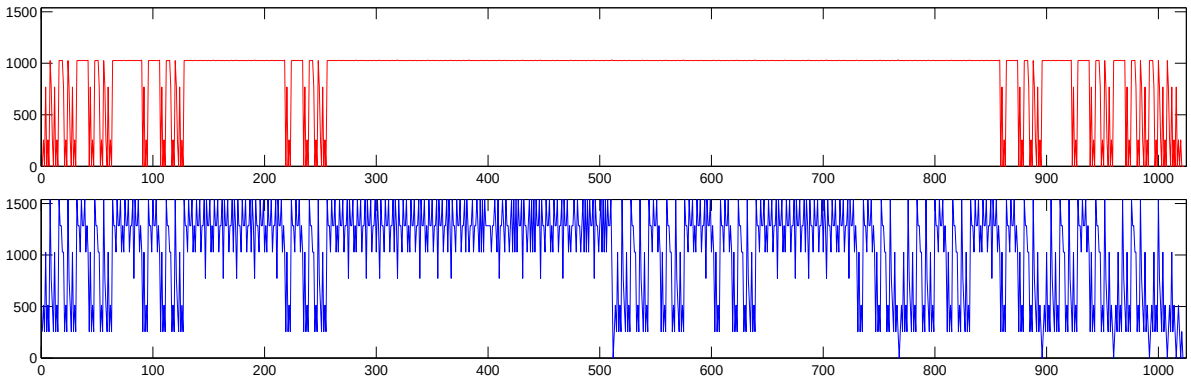

Figure 17: Number of hashes per round. The upper graph shows the result of Algorithm 3, the lower graph belongs to Szydlo's algorithm ($H = 10 \Longrightarrow 1024$ rounds).

It is evidently noticeable that the scheduling used within the new algorithm leads to a more balanced authentication path computation and with this to a more balanced signature generation. Also the total number of hash values was reduced. As one can see in Table 4 the mean value of the number of hashes per round is reduced compared to Szydlo's algorithm. This shows that the new algorithm actually gets along with less hashes per round. The standard deviation, which indicates the balancing, decreases drastically. By this we assert the better balancing of our new algorithm.

Visually these improvements are recognizable by the fact that in Figures 16 and 17 the red line proceeds mostly below the blue one and the blue graph shows much more oszillating properties. For higher values of H we get related results.

Worst Case Values. We are going to compare the results gained theoretically in section 4 with some measured practical values. The following table presents some results obtained with a practical implementation of both hash tree traversal algorithms. It presents the worst case number of hashes and leaves required per round. Again a Winternitz parameter $w = 2$ and a 160 bit hash function are chosen, so that the number of hashes for one leaf calculation is 256. The values in brackets are the theoretically forecasted costs of lemma 3 and 4.

	Our Algorithm			Szydlo's Algorithm		
H	leaves	hashes	hashes total	leaves	hashes	hashes total
5	1 (1)	1 (2.5)	257	3	1	769
10	4 (4)	8 (11.5)	1032	6	3	1539
15	6 (6)	14 (17.5)	1551	9	5	2309
20	9 (9)	24 (26.5)	2328	12	7	3079

Table 5: Comparison of the number of hashes required in the worst case.

It is considerable that the precomputed bounds hold. The number of leaves computed per round is tight. Even in the worst case, our new algorithm needs less hashes than Szydlo's former algorithm.

6.2 Comparison: GMSS

In this part the GMSS implementation is analyzed. As hash function, all tests used the SHA1 version out of the FlexiCoreProvider. As pseudo random number generator the Sha1PRNG of the Sun provider was used. All tests were performed on an Intel Core 2 Duo T7200 2GHz processor with 1 GB RAM. As runtime environment the Sun JRE 1.3 was deployed.

The time needed for generation and verification of a single signature is quite small. For this it is essential to measure timings in microseconds. Following [27] we use the *hrtlib.dll* library, which provides a timer to exactly measure time differences in those spheres. Just creating a signature more than once and computing the mean value would not be a solution: the private key changes with every signature, so it is not easy to create the same signature more than once.

Nearly all parametersets $\mathcal{P}$ used for the testings are characterized by the fact that the Winternitz parameter belonging to the lowest layer is smaller than all others. Smaller parameter w allows faster signature generation, but is responsible for bigger signatures. As on the lowest layer the public keys for the leaf values have to be computed at once and cannot be distributed, a smaller parameter on this layer speeds up the whole process, even more than the parameters on upper layers would do. For this the Winternitz parameter on the lowest layer is mostly chosen smaller than the others.

Balancing. First a comparison between the old GMSS implementation of [17] is compared to the new one. Using the parameterset $\mathcal{P} = (4, (4, 4, 4, 4), (8, 8, 8, 3))$ the signature generation lasts arbitrarily four milliseconds, whatever implementation is used. But among different signatures the duration varies more or less, because the offline part does not always compute the same parts. Figure 18 depicts the resulting timings for both implementations for 200 signatures. The red line indicates the timings of the new implementation, the blue line belongs the old one. The parameter K is set to 2 on each layer.

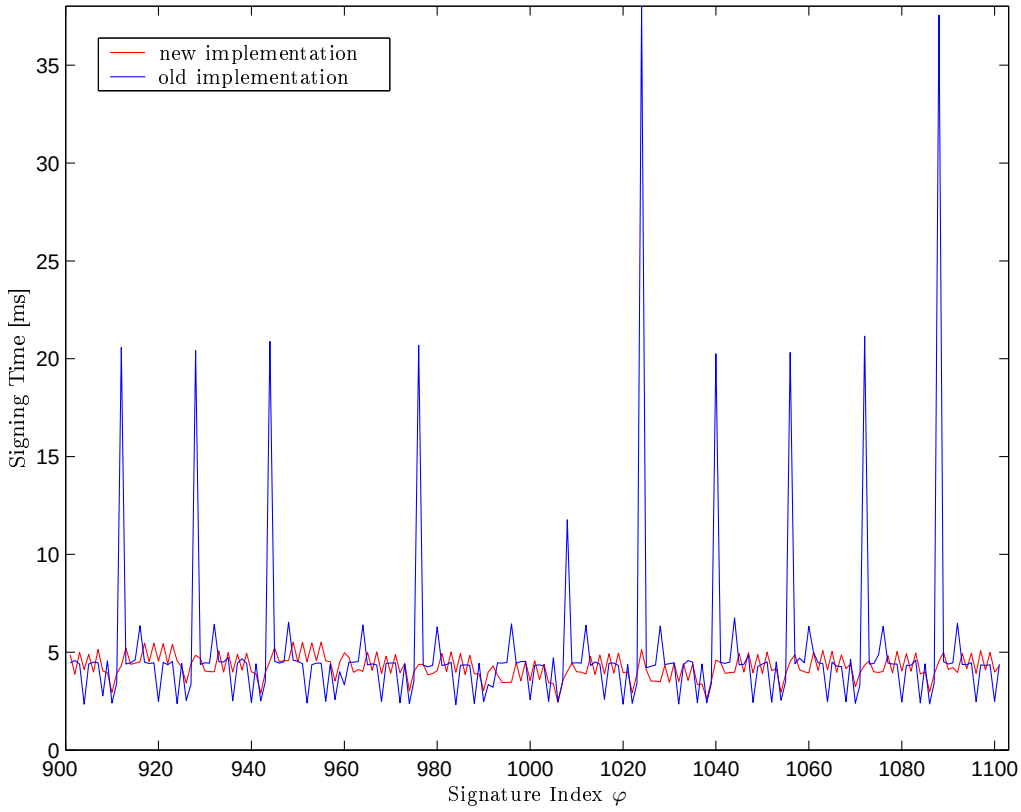

Figure 18: Time needed for signing with GMSS. The red line shows the timings using the new GMSS implementation, the blue line belongs to the old implementation. The used parameterset is $\mathcal{P} = (4, (4, 4, 4, 4), (8, 8, 8, 3))$, K is set 2 on each layer.

Figure 18 illustrates that the time needed for signing is much more balanced using the new GMSS implementation. The edges within the blue graph come up every 16 signatures. Using a bottom tree of height 4 (which means $2^4 = 16$ leaves), the old implementation needs much time for advancing a leaf on the second lowest layer. This is the situation where the new implementation uses the better balanced authentication path algorithm. Furthermore the precomputation of the actual and the coming (treehash) leaves of this tree saves time. Those leaves are computed completely within the old implementation, whereas in the other case they can be simply copied. So the applied changes really affect the timings the way it was supposed.

The statistical analysis of the data emphasizes the better balancing of the new implementation: whereas the mean value remains nearly the same (5.0 ms (old) to 4.2 ms (new)), the standard deviation of the timings was reduced to more than a seventh part: it decreases from 4.6 ms to 0.6 ms using the new GMSS implementation. This evidently shows that the scheduling of the nodes in the upper tree really leads to better balancing attributes for the signature generation.

Greater Amounts Of Signatures. In [7] some linear optimization was used to find optimal GMSS parameter sets, allowing modular key and signature sizes besides applicable timings. The optimal sets for an amount of 2^{40} and 2^{80} signatures were adopted and the parameter K was included. So we get the following parameter sets for our test:

$$\mathcal{P}_{40} = (2, (20, 20), (10, 5), (2, 2)) \qquad \mathcal{P}'_{40} = (2, (20, 20), (9, 3), (2, 2))$$

$$\mathcal{P}'_{80} = (4, (20, 20, 20, 20), (7, 7, 7, 3), (2, 2, 2, 2))$$

The following tabular shows the resulting timings and memory requirements. The key size always denotes the byte length of the ASN.1 encoded keys. The timings were obtained on the above mentioned platform as mean value of the first 2^{12} signatures. With a tree of height 20 on the lowest layer, for comparison the first 2^{21} or even more signatures should have been created, so that an advance on upper

layers was considered. But this test would take too long, so only the first 2^{12} were constructed. To show how this effects the final results, we compared the timings and key sizes of a GMSS structure with lowest layer height 10 with 2^{10} and 2^{15} signatures: the difference in the private key size is 0.2%, whereas in timings no difference is recognizable. So we adopt that for our parametersets it is adequate to compare only the first 2^{12} signatures.

The values in the tables represent the following: m values are memory requirements for the keys and the signature. The time needed for key pair generation, signature construction or verification, respectively, is denoted by t values.

	$m_{\text{public key}}$	$m_{\text{private key}}$	$m_{\text{signature}}$	t_{keygen}	t_{sign}	t_{verify}
$\mathcal{P}_{40}$	75 bytes	12341 bytes	1868 bytes	539 min	13.4 ms	13.1 ms
$\mathcal{P}'_{40}$	75 bytes	12501 bytes	2348 bytes	299 min	6.6 ms	8.1 ms
$\mathcal{P}'_{80}$	93 bytes	30372 bytes	4256 bytes	464 min	7.4 ms	8.4 ms

Table 6: Measured values for the new GMSS implementation

For comparing these numbers with the old GMSS implementation, we adopt the results from [17] measured on an Asus V6J (1.83GHz CPU).

	$m_{\text{public key}}$	$m_{\text{private key}}$	$m_{\text{signature}}$	t_{keygen}	t_{sign}	t_{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	14731 bytes	4256 bytes	498 min	11.6 ms	9.5 ms

Table 7: Measured values for the old GMSS implementation, from [17]

The timings are quite the same using both implementations, the discrepancies are mostly caused by the different platforms. The signature size remains exactly the same, it was not touched by the revision of GMSS. The public key rises few, as the K parameters for each layer have to be stored additionally. The private key size nearly doubles. For the better authentication path computation, more upcoming data has to be stored, like the treehash instances or the stacks of the following trees

on each height. This data is stored in the private key, and that is why its size grows. However, the sizes of up to 30 kilobytes are still useable in practice. The table only shows a mean value of the private key sizes: for $\mathcal{P}'_{40}$ it ranges from 10541 to 12731 bytes, for $\mathcal{P}'_{80}$ it differs between 28413 and 30602 bytes. The balancing of the timings cannot be seen in this tables, the achievements in this concern have been shown in the last section.

Some more measures are depicted in Appendix A. Therefrom we get some more information of the affects of the GMSS parameters: if the parameters K raise, the private key size rises as well. Higher K makes sure that more upper nodes are permanently stored in the private key, so it is clear that its size increases. Simultaneously the signing time declines, as the upper nodes must no more be computed chosing higher K values. The signature size is not affected by this parameter.
The impacts of the parameter w are the same as before in GMSS: choosing bigger w values, the signature and the private key sizes decline, whereas the timings grow a bit. Smaller w's have exactly the contrary impact.

It is concluded that GMSS is ready to use in practical applications. The timings are comparable to other signature schemes that are used widely today, like ECDSA, DSA or RSA. For measured results of these schemes see for example [5]. Even if the key sizes, especially of the private signing keys, are relatively big, GMSS is still applicable. We have created up to 2^{80} signature keys with reasonable effort and costs. This amount should be adequate for todays use, even in online applications like packet signing in broadcast protocols.

7 Conclusion and Further Work

Merkle Tree Traversal. This thesis presented a new algorithm for the computation of consecutive authentication paths in Merkle trees. Compared to the best formerly known, the new algorithm features a better balancing concerning the real number of hash function calls per round. This property could be obtained theoretically, and it could be approved by practical results as well. The worst case number of leaves calculated per round was reduced to $(H - K)/2 + 1$, while the maximum number of hashes to perform is bounded linearly in H.

Parameterization allows a trade-off between computation time and memory demands. This allows the application of the algorithm on different kinds of devices, for example on smart cards and similar low computation appliances. The storage needed for the flow of the algorithm is bounded logarithmically in the number of leaves, which is the best complexity to reach. Even on hardware which does not allow dynamic memory allocation, the new algorithm does only need linear space. This results in the utilization of one single stack shared by all treehash instances.

For heights H greater than twenty the advantages of Algorithm 3 decline. But in practice Merkle trees with heights $H > 20$ should not be applied. The key pair generation, which must always compute the whole tree at once, lasts too long in this case. It is much more comfortable to use the extensions of MSS, if greater amounts of signatures are demanded.

Practical Part: GMSS. The second part of this thesis was the implementation of the new algorithm into an existing GMSS implementation for the FlexiProvider. The construction and use of the JCA assures maximal flexibility. The generalized Merkle signature scheme can be plugged into every application based on the JCA. As an example there exists a MS Outlook plugin for signing emails with any algorithm of the FlexiPovider [5]. The Winternitz one time signature scheme can easily be replaced by any other OTS scheme. As a first further work the BiBa OTS scheme [13] shall be integrated into GMSS, as it allows smaller signatures than the Winternitz scheme. For the hash function, used for the construction of the Merkle trees, different

variants have been implemented, e.g. SHA1 or SHA512. But even if the SHA-family should turn out insecure, the message digest function could be exchanged easily. The same occurs to the used pseudo random number generator. While we used the one described in [14], another one could be made use of.

Still one drawback of GMSS is the long key generation time. As an amount of 2^{80} keys can be regarded as cryptographically unlimited, in practice this problem can be disregarded, because it only must be run once before all signatures are created. So this part can be done offline, before the creation of the first signature.

The Merkle signature schemes are characterized by an enormous flexibility. Equipped with so many parameters these schemes can be used on nearly every imaginable platform. The size of the keys and the signatures can be adjusted as well as the timings for signature generation or verification, respectively. This makes GMSS (as actual the best implementation of the Merkle schemes) applicable on all hardware devices.

The timings for signature generation and verification, respectively, are comparable to the widely used schemes like RSA, DSA or ECDSA. The GMSS public key is even smaller than former keys. The private key is relatively big, but for today's practical usage still reasonable. Therefore, a conclusion is that today there are digital signature schemes that exist out of the post quantum computing field with possible practicable use.

References

[1] Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *IEEE Symposium on Foundations of Computer Science*, pages 124–134, 1994.

[2] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *STOC '96: Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 212–219, New York, NY, USA, 1996. ACM.

[3] Arjen K. Lenstra and Eric R. Verheul. Selecting cryptographic key sizes. *Journal of Cryptology*, 14(4):255–293, 2001. Updated version from 2004 available at `http://plan9.bell-labs.com/who/akl/index.html`.

[4] 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.

[5] Johannes Buchmann, Luis Carlos Coronado Garcia, Erik Dahmen, Martin Döring, and Elena Klintsevich. CMSS – an improved Merkle signature scheme. In *Proc. Progress in Cryptology (Indocrypt'06)*, volume 4329 of *Lecture Notes in Computer Science*, pages 349–363. Springer-Verlag, 2006.

[6] Luis Carlos Coronado García. On the security and the efficiency of the Merkle signature scheme. Cryptology ePrint Archive, Report 2005/192, 2005.

[7] Johannes Buchmann, Erik Dahmen, Elena Klintsevich, Katsuyuki Okeya, and Camille Vuillaume. Merkle signatures with virtually unlimited signature capacity. 5th International Conference on Applied Cryptography and Network Security - ACNS'07, LNCS 4521, Springer, 2007, pp. 31-45.

[8] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[9] Alexander May. Skript zur Vorlesung Public Key Kryptanalyse, TU Darmstadt, 2005/2006.

[10] Digital signature standard. FIPS PUB 180-2, 2002. Available at `http://csrc.nist.gov/publications/PubsFIPS.html`.

[11] Ron Rivest. The MD5 Message-Digest Algorithm, 1992.

[12] 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.

[13] Adrian Perrig. The BiBa one-time signature and broadcast authentication protocol. In *ACM Conference on Computer and Communications Security*, pages 28–37, 2001.

[14] Digital signature standard. FIPS PUB 186-2, 2000. Available at `http://csrc.nist.gov/publications/PubsFIPS.html`.

[15] Y. Hu, A. Perrig, and D. Johnson. Packet leashes: A defense against wormhole attacks in wireless ad hoc networks. Technical report, Department of Computer Science, Rice University, 2001.

[16] Michael Szydlo. Merkle tree traversal in log space and time (preprint version), 2003. Available at `http://www.szydlo.com`.

[17] Sebastian Blume. Efficient Java implementation of GMSS, diploma thesis, 2007.

[18] Mihir Bellare and Sara K. Miner. A forward-secure digital signature scheme. *Lecture Notes in Computer Science*, 1666:431–448, 1999.

[19] Shimon Even, Oded Goldreich, and Silvio Micali. On-line/off-line digital signatures. In *CRYPTO '89: Proceedings on Advances in cryptology*, pages 263–275, New York, NY, USA, 1989. Springer-Verlag New York, Inc.

[20] Markus Jakobsson, Tom Leighton, Silvio Micali, and Michael Szydlo. Fractal Merkle tree representation and traversal. In *Proc. Cryptographer's Track at RSA Conference (CT-RSA'03)*, volume 2612 of *Lecture Notes in Computer Science*, pages 314–326. Springer-Verlag, 2003.

[21] Dalit Naor, Amir Shenhav, and Avishai Wool. One-time signatures revisited: Have they become practical. Cryptology ePrint Archive, Report 2005/442, 2005.

[22] Michael Szydlo. 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.

[23] Piotr Berman, Marek Karpinski, and Yakov Nekrich. Optimal trade-off for Merkle tree traversal. *El. Coll. on Comp. Complexity*, 49, 2004.

[24] Sun Microsystems. JavaTM Cryptography Architecture - API Specification and Reference, 2004. Available at `http://java.sun.com/j2se/1.5.0/docs/guide/security/CryptoSpec.html`.

[25] FlexiProvider research group at Technische Universität Darmstadt. Flexi - provider - an open source java cryptographic service provider, 2001 - 2008. Available at `http://www.flexiprovider.de`.

[26] International Telecommunication Union Telecommunication Standardization Sector (ITU-T). Abstract Syntax Notation One (ASN.1) X.680: Specification of basic notation, ITU Standard, 2002.

[27] Vladimir Roubtsov. My kingdom for a good timer! Reach submillisecond timing precision in Java. JavaWorld.com, January 2003, `http://www.javaworld.com/javaworld/javaqa/2003-01/01-qa-0110-timing.html`.

[28] Don Johnson and Alfred Menezes. The elliptic curve digital signature algorithm ECDSA, 1999.

[29] Ron Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[30] 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.

[31] Boris Ederov. Merkle tree traversal techniques, bachelor thesis, 2007.

[32] S. Micali. Efficient certificate revocation. Technical Report MIT/LCS/TM-542b, 1996.

[33] A. Perrig, R. Canetti, D. Tygar, and D. Song. The tesla broadcast authentication protocol, 2002.

[34] Charanjit Jutla and Moti Yung. Paytree: 'amortized-signature' for flexible micropayments. In *2nd Workshop on Electronic Commerce*, pages 213–221. USENIX, 1996.

[35] Ronald L. Rivest and Adi Shamir. Payword and micromint: Two simple micropayment schemes. In *Security Protocols Workshop*, pages 69–87, 1996.

A Practical Results

$m_{\text{public key}}$	$m_{\text{private key}}$	$m_{\text{signature}}$	t_{keygen}	t_{sign}	t_{verify}
$\mathcal{P} = (2, (8, 8), (10, 5), (2, 2))$					
75 bytes	5852 bytes	1388 bytes	8.0 sec	8.9 ms	15.1 ms
$\mathcal{P} = (2, (8, 8), (10, 5), (6, 6))$					
75 bytes	7780 bytes	1388 bytes	8.1 sec	5.8 ms	14.6 ms
$\mathcal{P} = (4, (8, 8, 8, 8), (3, 3, 3, 3), (2, 2, 2, 2))$					
93 bytes	26261 bytes	5216 bytes	1.9 sec	4.2 ms	2.1 ms
$\mathcal{P} = (4, (8, 8, 8, 8), (8, 8, 8, 3), (2, 2, 2, 2))$					
93 bytes	16464 bytes	3116 bytes	11.6 sec	4.1 ms	13.9 ms
$\mathcal{P} = (4, (8, 8, 8, 8), (8, 8, 8, 3), (6, 6, 6, 6))$					
93 bytes	21315 bytes	3116 bytes	11.8 sec	2.5 ms	14.3 ms
$\mathcal{P} = (4, (10, 10, 10, 10), (9, 9, 9, 3), (2, 2, 2, 2))$					
93 bytes	18205 bytes	3156 bytes	80.2 sec	5.2 ms	24.8 ms
$\mathcal{P} = (4, (12, 12, 12, 12), (9, 9, 9, 3), (2, 2, 2, 2))$					
93 bytes	20585 bytes	3256 bytes	136 sec	12.7 ms	11.4 ms
$\mathcal{P} = (4, (16, 16, 16, 16), (8, 8, 8, 3), (2, 2, 2, 2))$					
93 bytes	20000 bytes	3316 bytes	322.9 sec	63 ms	22.1 ms
$\mathcal{P} = (2, (10, 10), (5, 4), (2, 2))$					
75 bytes	8335 bytes	1968 bytes	4.8 sec	6.9 ms	1.2 ms
$\mathcal{P} = (2, (10, 10), (10, 5), (2, 2))$					
75 bytes	6977 bytes	1468 bytes	32.6 sec	10.6 ms	15.1 ms
$\mathcal{P} = (2, (15, 15), (5, 4), (3, 3))$					
75 bytes	10873 bytes	2168 bytes	149 sec	9.3 ms	2.1 ms
$\mathcal{P} = (2, (15, 15), (8, 5), (3, 3))$					
75 bytes	9834 bytes	1748 bytes	409 sec	13.8 ms	5.1 ms

Continues on next page...

$m_{\text{public key}}$	$m_{\text{private key}}$	$m_{\text{signature}}$	t_{keygen}	t_{sign}	t_{verify}
$\mathcal{P} = (3, (15, 15, 10), (5, 5, 4), (3, 3, 2))$					
84 bytes	17982 bytes	3072 bytes	193 sec	7.4 ms	2.9 ms
$\mathcal{P} = (3, (15, 15, 10), (8, 8, 5), (3, 3, 2))$					
84 bytes	15644 bytes	2392 bytes	849 sec	11.1 ms	10.4 ms
$\mathcal{P}'_{40} = (2, (20, 20), (9, 3), (2, 2))$					
75 bytes	12501 bytes	2348 bytes	299 min	6.6 ms	8.1 ms
$\mathcal{P}_{40} = (2, (20, 20), (10, 5), (2, 2))$					
75 bytes	12341 bytes	1868 bytes	539 min	13.4 ms	13.1 ms
$\mathcal{P}'_{80} = (4, (20, 20, 20, 20), (7, 7, 7, 3), (2, 2, 2, 2))$					
93 bytes	30372 bytes	4256 bytes	464 min	7.4 ms	8.4 ms

Table 8: Results of the new GMSS implementation: time and memory requirements of selected parameter sets. For the average timings, in each case the mean value of the first 2^{12} signatures were considered.

B Code Examples

This section presents an example code extract that shows how to use the FlexiProvider implementation of GMSS. It is divided into three steps: Generating a key pair, generating a signature and verifying the signature.

Generating a Key Pair.
Input: Parameterset, Output: ASN.1 encoded keys

1. Add Providers
   ```
   Security.addProvider(new FlexiCoreProvider());
   Security.addProvider(new FlexiPQCProvider());
   ```
2. Get KPG instance
   ```
   KeyPairGenerator kpg = KeyPairGenerator.getInstance("GMSSwithSHA1");
   ```
3. Set the required Parameterset, create corresponding Parameterspec
   ```
   GMSSParameterset gps = new GMSSParameterset(3, {10, 10, 10}, {2, 4,
   3}, {2, 2, 2});
   GMSSParameterSpec gpsp = new GMSSParameterSpec(gps);
   ```
4. Initializing Key Pair Generator
   ```
   kpg.initialize(gpsp);
   ```
5. Generating key pair
   ```
   KeyPair GMSSkeyPair = kpg.generateKeyPair();
   GMSSPrivateKey privateKey = (GMSSPrivateKey)GMSSkeyPair.getPrivate();
   GMSSPublicKey publicKey = (GMSSPublicKey)GMSSkeyPair.getPublic();
   byte[] privKey = privateKey.getEncoded();
   byte[] pubKey = publicKey.getEncoded();
   ```

Generating a Signature.
Input: encoded keys, message, Output: signature

1. Get the private key
   ```
   KeySpec privKeySpec = new PKCS8EncodedKeySpec(privKey);
   KeyFactory kf = KeyFactory.getInstance("GMSS", "FlexiPQC");
   privateKey = (GMSSPrivateKey)kf.generatePrivate(privKeySpec);
   ```
2. Initialize the signature generation phase
   ```
   Signature Sig = Signature.getInstance("GMSSwithSHA1","FlexiPQC");
   Sig.initSign(privateKey);
   ```
3. Create the signature
   ```
   Sig.update(message.getBytes());
   byte[] sigBytes = Sig.sign();
   ```

Verifying the Signature.
Input: signature, message, encoded public key

1. decode public key
   ```
   KeySpec pubKeySpec = new X509EncodedKeySpec(pubKey);
   publicKey = (GMSSPublicKey)kf.generatePublic(pubKeySpec);
   ```
2. Initialize Verification
   ```
   Sig.initVerify(publicKey);
   ```
3. Verification Process, returns either `true` or `false`
   ```
   Sig.update(message.getBytes());
   Sig.verify(sigBytes);
   ```

C ASN.1 Encoding

This part presents the ASN.1 encoding [26] of the GMSS keys. The public key encoding was modified only marginally: the ParameterSet was extended by the sequence of the parameter K for each layer. This is the new ASN.1 definition of the GMSS public key:

```
GMSSPublicKey    ::= SEQUENCE {
    publicKey         SEQUENCE OF OCTET STRING
    heightOfTrees     SEQUENCE OF INTEGER
    Parameterset      ParSet
}
ParSet           ::= SEQUENCE {
    T                 INTEGER
    h                 SEQUENCE OF INTEGER
    w                 SEQUENCE OF INTEGER
    K                 SEQUENCE OF INTEGER
}
```

The private key ASN.1 definition was enlarged with the treehash, stack and retain parts. `DistrRoot` and `TreehashStack` were added as well. The whole ASN.1 definition of the GMSS private key is the following:

```
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
    curTreehash       SEQUENCE OF TreehashStack
    nextTreehash      SEQUENCE OF TreehashStack
    StackKeep         SEQUENCE OF Stack
    curStack          SEQUENCE OF Stack
    nextStack         SEQUENCE OF Stack
    curRetain         SEQUENCE OF Retain
    nextRetain        SEQUENCE OF Retain
    nextNextLeaf      SEQUENCE OF DistrLeaf
```

```
    upperLeaf         SEQUENCE OF DistrLeaf
    upperTHLeaf       SEQUENCE OF DistrLeaf
    minTreehash       SEQUENCE OF INTEGER
    nextRoot          SEQUENCE OF OCTET STRING
    nextNextRoot      SEQUENCE OF DistrRoot
    curRootSig        SEQUENCE OF OCTET STRING
    nextRootSig       SEQUENCE OF DistrRootSig
    Parameterset      ParSet
    names             SEQUENCE OF ASN1IA5String
}
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
}
DistrRoot         ::= SEQUENCE {
    name              SEQUENCE OF ASN1IA5String
    statBytes         SEQUENCE OF OCTET STRING
    statInts          SEQUENCE OF INTEGER
    treeH             SEQUENCE OF Treehash
    ret               SEQUENCE OF Retain
}
TreehashStack     ::= SEQUENCE OF Treehash
Treehash          ::= 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
    K                 SEQUENCE OF INTEGER
}
Retain            ::= SEQUENCE OF Stack
AuthPath          ::= SEQUENCE OF OCTET STRING
Stack             ::= SEQUENCE OF OCTET STRING
```

D Object Identifiers

The following table shows the object identifiers of some predefined GMSS implementations. Those use the given hash function for the OTS scheme as well as for the Merkle tree construction. For all cases the hash functions are taken out of the FlexiCoreProvider.

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 9: Object Identifiers for GMSS

The different number groups of the above given object identifiers signify the following:

1.3.6.1.4.1.8301	Darmstadt University of Technology
1.3.6.1.4.1.8301.3	Cryptography and Computer Algebra Research Group
1.3.6.1.4.1.8301.3.1	Cryptographic Algorithms
1.3.6.1.4.1.8301.3.1.3	Post Quantum Cryptography
1.3.6.1.4.1.8301.3.1.3.3	GMSS