Merkle Signatures with Virtually Unlimited Signature Capacity

Johannes Buchmann[1], Erik Dahmen[1], Elena Klintsevich[1], Katsuyuki Okeya[2], and Camille Vuillaume[2]

[1] Technische Universität Darmstadt
Department of Computer Science
Hochschulstraße 10, 64289 Darmstadt, Germany
{buchmann,dahmen,klintsev}@cdc.informatik.tu-darmstadt.de
[2] Hitachi, Ltd., Systems Development Laboratory,
1099, Ohzenji, Asao-ku, Kawasaki-shi, Kanagawa-ken, 215-0013, Japan
{katsuyuki.okeya.ue,camille.vuillaume.ch}@hitachi.com

Abstract. We propose GMSS, a new variant of the Merkle signature scheme. GMSS is the first Merkle-type signature scheme that allows a *cryptographically unlimited* (2^{80}) number of documents to be signed with one key pair. Compared to recent improvements of the Merkle signature scheme, GMSS reduces the signature size as well as the signature generation cost.

Keywords: Merkle signatures, post-quantum cryptography, SSL.

1 Introduction

Digital signatures are one of the most important applications of public key cryptography. For example, they are an essential part of the SSL/TLS protocol for authenticating websites. If large scale quantum computers are built, all popular digital signature schemes like RSA, DSA and ECDSA will become insecure [12]. In addition, significant progresses have been made for solving the underlying number theoretic problems for RSA or DSA, and therefore, the key lengths required to provide sufficient security have been steadily increasing [10]. As a consequence, it is urgent to look for alternative signature schemes, thoroughly analyze and understand their security, and see how they behave in real-life applications.

A promising alternative to common digital signature schemes is the Merkle signature scheme (MSS) proposed by Merkle in [11]. The security of MSS solely relies on the existence of cryptographic hash functions. According to [7], the required properties of the hash function are one-wayness and collision resistance. Using MSS, it is possible to remove the requirement for number theoretic assumptions from digital signatures. In recent years, many improvements to the original MSS were proposed. The inefficient key generation and the resulting limited signature capacity of MSS is addressed in [3]. The authors proposed CMSS, a variant of MSS that increases the signature capacity from 2^{20} to 2^{40} and provides

J. Katz and M. Yung (Eds.): ACNS 2007, LNCS 4521, pp. 31–45, 2007.

competitive timings compared to common signature schemes. Other important contributions are efficient tree traversal algorithms [8,13,14], which play a crucial role for fast signature generation.

First of all, it is unclear whether a signature capacity of 2^{40} is sufficient for practical applications. Consider for example webservers using SSL/TLS or electronic archives. Second, the original MSS as well as CMSS suffer from quite large signature sizes. Further, there may be time and space requirements in specific application environments, such as smart cards, that are not satisfied by current constructions.

In this paper we present the generalized Merkle signature scheme (GMSS). GMSS is a highly flexible variant of CMSS that can be adjusted to the requirements and constraints of a particular environment. GMSS drastically reduces the signing time by distributing the costs for one signature generation across several previous signatures and the key generation. This in turn makes it possible to choose parameters that provide smaller signatures. Using GMSS, it is possible to achieve a signature capacity of 2^{80} with competitive timings and reasonable signature sizes, i.e. 26.1 ms for signing, 18.1 ms for verifying, and 3,620 bytes for the signature. In case of a signature capacity of 2^{40} it is possible to achieve 26 ms for signing, 19.6 ms for verifying, and only 1,860 bytes for the signature. For both signature capacities, it is also possible to achieve signing and verification times of only 10 ms at the expense of larger signatures, i.e. 2,340 bytes for 2^{40} and 4,240 bytes for 2^{80}. We also propose a client-server protocol for webservers using SSL/TLS that minimizes the latency and improves resistance to denial of service attacks.

This paper is organized as follows: Section 2 introduces GMSS. Section 3 shows how to choose appropriate parameters and how to exchange signatures in the SSL/TLS protocol. Finally, Section 4 states our conclusion.

2 GMSS in Theory

This section at first shows the general construction of GMSS. Then the key generation, signature generation and verification are explained in detail and the costs and memory requirements are estimated.

2.1 General Construction

The basic construction of GMSS consists of a tree with T layers. The nodes of this tree are in turn Merkle trees [11]. A brief description of Merkle trees can be found in Appendix A. The height of all Merkle trees in a certain layer i is denoted by h_i. Trees in different layers may have different heights. Each Merkle tree in layer $i = 1, \ldots, T-1$ is parent to 2^{h_i} Merkle trees. $\mathcal{T}_{1,0}$ denotes the single Merkle tree in layer 1. The $2^{h_1+\ldots+h_{i-1}}$ Merkle trees in layers $i = 2, \ldots, T$ are denoted by $\mathcal{T}_{i,j}$, $j = 0, \ldots, 2^{h_1+\ldots+h_{i-1}} - 1$ according to their position from left to right.

Parents and children Merkle trees have the following relationship: the root of a child tree is signed with the one-time signature key corresponding to a certain

leaf of his parent tree. In the following, when talking about leaves in the context of signing, we mean the corresponding one-time signature key. $\text{ROOT}_{\mathcal{T}}$ denotes the root of tree $\mathcal{T}$. $\text{SIG}_{\mathcal{T}}$ denotes the one-time signature of $\text{ROOT}_{\mathcal{T}}$, which is generated using leaf l of $\mathcal{T}$'s parent. Message digests d are signed using the leaves of the Merkle trees on the deepest layer T and we call their one-time signature SIG_d. Thanks to the layer hierarchy, the number of message digests that can be signed using one GMSS key pair is $S = 2^{h_1+\ldots+h_T}$. The general construction of GMSS is depicted in Figure 1.

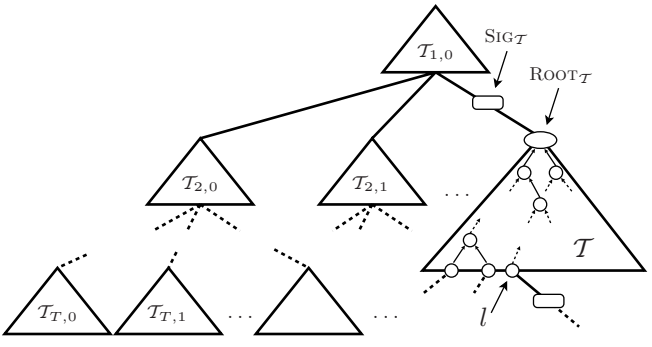

Fig. 1. General construction of GMSS

For any given signature $s \in \{0, \ldots, 2^{h_1+\ldots+h_T} - 1\}$, there is a unique path p from the Merkle tree on the lowest layer T, which is used to sign the digest, to the single Merkle tree on the top layer 1 ($\mathcal{T}_{1,0}$). This path involves one Merkle tree at each layer $i = 1, \ldots, T$. A GMSS signature of a message digest d contains the one-time signature SIG_d of d and the one-time signatures $\text{SIG}_{\mathcal{T}}$ of the roots of all Merkle trees on path p, except for $\mathcal{T}_{1,0}$. For all trees $\mathcal{T}$ on path p, a GMSS signature also contains the authentication path $\text{AUTH}_{\mathcal{T},l}$ of the leaf l that is used to sign either the child of $\mathcal{T}$, or the digest d. An authentication path is defined as the sequence of the siblings of all nodes on the path from leaf l to the root of $\mathcal{T}$. GMSS uses the Winternitz one-time signature scheme [4,11] for signing digests d and $\text{ROOT}_{\mathcal{T}}$ (see Appendix B for for a brief introduction to the Winternitz one-time signature scheme). w_i denotes the Winternitz parameter used in layer $i = 1, \ldots, T$. Different layers are allowed to use different Winternitz parameters. GMSS is specified by a GMSS parameter set

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

Remark 1. The Merkle variant CMSS proposed in [3] is a special case of GMSS. CMSS uses only two layers, where both layers use the same Winternitz parameter and all trees in both layers have the same height, i.e. $\mathcal{P} = \big(2, (h, h), (w, w)\big)$.

During the key generation, GMSS computes $\text{SIG}_{\mathcal{T}}$ and $\text{AUTH}_{\mathcal{T},l}$ for the Merkle trees on the path p used by the first signature (Section 2.3). $\text{SIG}_{\mathcal{T}}$ and $\text{AUTH}_{\mathcal{T},l}$

required by succeeding signatures are precomputed (Section 2.4). Since those values change less frequently for upper layers, the precomputation can be distributed over many steps. On the one hand, this results in a significant improvement of the signing speed. On the other hand, this enables us to choose large Winternitz parameters w_i, which results in smaller signatures. In Section 3.1, we formulate this trade-off as an optimization problem to find an optimal parameter set.

2.2 Winternitz One-Time Signature Key Generation

First of all, we describe our strategy for generating random data required by one-time signature keys. Let $H : \{0,1\}^* \to \{0,1\}^n$ be a cryptographic hash function with output length n bits. We adopt the approach for the generation of the Winternitz OTS signature keys proposed in [3] and use a pseudo random number generator (PRNG) $f : \{0,1\}^n \to \{0,1\}^n \times \{0,1\}^n$, $\text{SEED}_{\text{in}} \mapsto (\text{SEED}_{\text{out}}, \text{RAND})$ for generating secrets. In the following, we call c_{HASH} the cost evaluating one hash (in our case, the input size is small and fixed) and c_{PRNG} the cost for calling the PRNG. To assure interoperability we selected the PRNG described in [6], Appendix 3.1, which requires one single call to the hash function H.

$$\text{RAND} \leftarrow H(\text{SEED}_{\text{in}}), \text{SEED}_{\text{out}} \leftarrow (1 + \text{SEED}_{\text{in}} + \text{RAND}) \bmod 2^n \tag{1}$$

We use an initial seed $\text{SEED}_{\mathcal{T}_{i,j},l}$ to generate the seed for the lth Winternitz OTS signature key of Merkle tree $\mathcal{T}_{i,j}$, i.e.

$$\begin{aligned}(\text{SEED}_{\mathcal{T}_{i,j},l+1}, \text{SEED}_{\text{OTS}}) &\leftarrow f(\text{SEED}_{\mathcal{T}_{i,j},l}) \\ (\text{SEED}_{\text{OTS}}, x_k) &\leftarrow f(\text{SEED}_{\text{OTS}}), k = 1, \ldots, t_{w_i}\end{aligned}$$

and $t_{w_i} = \lceil n/w_i \rceil + \lceil (\lfloor \log_2(\lceil n/w_i \rceil) \rfloor + 1 + w_i)/w_i \rceil$. The lth one-time signature key and the lth leaf of Merkle tree $\mathcal{T}_{i,j}$ are given as $X = (x_1, \ldots, x_{t_{w_i}})$ and $Y = H\big(H^{2^{w_i}-1}(x_1) \| \ldots \| H^{2^{w_i}-1}(x_{t_w})\big)$, respectively. Note that Y is also the Winternitz one-time verification key that corresponds to X. $H^k(x)$ denotes the hash function applied k times and $\|$ the concatenation of two strings. The updated seed $\text{SEED}_{\mathcal{T}_{i,j},l+1}$ is stored and used to generate the $(l+1)$th signature key. If the current signature key is associated with the last leaf of tree $\mathcal{T}_{i,j}$, i.e. $l = 2^{h_i} - 1$, the updated seed is used as initial seed for the next Merkle tree $\mathcal{T}_{i,j+1}$, i.e. $(\text{SEED}_{\mathcal{T}_{i,j+1},0}, \text{SEED}_{\text{OTS}}) \leftarrow f(\text{SEED}_{\mathcal{T}_{i,j},2^{h_i}-1})$.

2.3 GMSS Key Generation

Next, we explain how to generate a GMSS keypair, establish the size of the public and private keys and the cost for computing them. From the parameter set $\mathcal{P}$ and initial seeds $\text{SEED}_{\mathcal{T}_{1,0},0}, \ldots, \text{SEED}_{\mathcal{T}_{T,0},0}$, the key generation step computes the GMSS public key $\text{ROOT}_{\mathcal{T}_{1,0}}$ and the GMSS private key which consists of the following entries.

$$\begin{cases} \text{SEED}_{\mathcal{T}_{i,0},0}\ , i = 1, \ldots, T\ , & \text{SEED}_{\mathcal{T}_{i,2},0}\ , i = 2, \ldots, T \\ \text{SIG}_{\mathcal{T}_{i,0}}\ , i = 2, \ldots, T\ , & \text{ROOT}_{\mathcal{T}_{i,1}}\ , i = 2, \ldots, T \\ \text{AUTH}_{\mathcal{T}_{i,0},0}\ , i = 1, \ldots, T\ , & \text{AUTH}_{\mathcal{T}_{i,1},0}\ , i = 2, \ldots, T \end{cases} \tag{2}$$

At first, the key generation computes the root of the first tree in each layer $\text{ROOT}_{\mathcal{T}_{i,0}}, i = 1, \ldots, T$. This can be done efficiently using a classical algorithm also referred to as treehash [13], shown in Algorithm 1. This algorithm uses a stack $\mathcal{S}$ of nodes, where each node knows its height in the tree. In this paper, we use a slightly modified version of treehash, which allows us to easily distribute costs by incrementally computing the root. In Algorithm 1, the leaf l is the lth verification key, computed using $\text{SEED}_{\mathcal{T}_{i,j},l}$ as described above. For a tree of height h_i, Algorithm 1 must be called 2^{h_i} times, where the 2^{h_i} leaves are supplied in sequential order, from left to right. For each call of Algorithm 1, the inner while loop might compute from 0 to h_i iterations, but in total, the 2^{h_i} calls will result in exactly $2^{h_i} - 1$ iterations. After the 2^{h_i} calls, the stack $\mathcal{S}$ contains a single node, the root of the tree. Note that during the computation of the root $\text{ROOT}_{\mathcal{T}_{i,0}}$, the authentication path for the 0th leaf of tree $\mathcal{T}_{i,0}$, i.e. $\text{AUTH}_{\mathcal{T}_{i,0},0}$ is generated for free, since all nodes of the tree are parsed by the algorithm.

Algorithm 1. Treehash

Input: Leaf l, stack $\mathcal{S}$
Output: updated stack $\mathcal{S}$

1. push l to $\mathcal{S}$
2. **while** top two nodes of $\mathcal{S}$ have the same height **do**
   2.1. pop n_1 from $\mathcal{S}$; pop n_2 from $\mathcal{S}$
   2.2. push $H(n_2||n_1)$ to $\mathcal{S}$
3. **return** $\mathcal{S}$

Next, the roots $\text{ROOT}_{\mathcal{T}_{i,1}}$ and authentication paths $\text{AUTH}_{\mathcal{T}_{i,1},0}$ of of the succeeding trees $\mathcal{T}_{i,1}$, $i = 2, \ldots, T$ are computed with Algorithm 1. As explained above, the initial seeds $\text{SEED}_{\mathcal{T}_{i,1},0}$ related to the trees $\mathcal{T}_{i,1}$ are now available. Finally, after generating the second tree in each layer, the seeds $\text{SEED}_{\mathcal{T}_{i,2},0}$ are available, which are stored as part of the private key to allow an efficient generation of trees $\mathcal{T}_{i,2}$ during the signing process. Note that $\text{SIG}_{\mathcal{T}_{i,0}}$, $i = 2, \ldots, T$ does not have to be computed explicitly. It is an intermediate value during the computation of the 0th leaf of tree $\mathcal{T}_{i-1,0}$, $i = 2, \ldots, T$.

Lemma 1 (Key Generation). *The total cost for the key generation is*

$$c_{\text{keygen}} = \sum_{i=1}^{T} c_{\text{tree}}(i) + \sum_{i=2}^{T} c_{\text{tree}}(i) \tag{3}$$

where $c_{\text{tree}}(i) = \left(2^{h_i}\left(t_{w_i}(2^{w_i}-1)+1\right) + 2^{h_i} - 1\right) c_{\text{HASH}} + 2^{h_i}\left(t_{w_i}+1\right) c_{\text{PRNG}}$. *The memory requirements for the keys are*

$$\begin{aligned} m_{\text{pubkey}} &= n \text{ bits} \\ m_{\text{privkey}} &= \left(\sum_{i=1}^{T}(h_i+1) + \sum_{i=2}^{T}(h_i + t_{w_{i-1}} + 2)\right) n \text{ bits.} \end{aligned} \tag{4}$$

Proof. A tree on layer i requires the computation of 2^{h_i} leaves. Each leaf computation requires $(2^{w_i}-1)\cdot t_{w_i}+1$ hash function evaluations and $t_{w_i}+1$ calls to the PRNG: one PRNG to obtain the seed and t_{w_i} to obtain the signature key. The 2^{h_i} applications of treehash require $2^{h_i}-1 c_{\text{HASH}}$. Therefore, the total cost for one tree on layer i is given as $c_{\text{tree}}(i) = \left(2^{h_i}\left((2^{w_i}-1)\cdot t_{w_i}+1\right)+2^{h_i}-1\right)c_{\text{HASH}} + 2^{h_i}\left(t_{w_i}+1\right)c_{\text{PRNG}}$. Since we construct two trees on layers $i = 2,\dots T$ and one on layer $i = 1$, the total cost for the key generation is given by Equation (3). The memory requirements of the keys depend on the output size of the hash function n. A root $\text{ROOT}_{\mathcal{T}_{i,j}}$ is a single hash value and requires n bits, which explains $m_{\text{pubkey}} = n$ bits. A seed $\text{SEED}_{\mathcal{T}_{i,j},l}$ requires n bits. A one-time signature $\text{SIG}_{\mathcal{T}_{i,j}}$ requires $n \cdot t_{w_{i-1}}$ bits. An authentication path requires $h_i \cdot n$ bits. For each layer $i = 2,\dots,T$, we store two seeds, two authentication paths, one root and one one-time signature. For layer $i = 1$, we store one seed and one authentication path. Hence, the size of the pivate key is m_{privkey} as in Equation (4). □

2.4 Signature Generation

In the following, we discuss the signature generation stage. We introduce the components of a GMSS signature and estimate formulas for the signature size and costs. We also explain how the signature generation costs are distributed. For the sth GMSS signature ($s \in \{0,\dots,2^{h_1+\dots+h_T}-1\}$), let

$$\begin{cases} j_T = \lfloor s/2^{h_T} \rfloor, & l_T = s \bmod 2^{h_T} \\ j_i = \lfloor j_{i+1}/2^{h_i} \rfloor, & l_i = j_{i+1} \bmod 2^{h_i}, i = 1,\dots,T-1. \end{cases} \tag{5}$$

The path from the lowest tree $\mathcal{T}_{T,j_T}$ to the top tree $\mathcal{T}_{1,0}$ used by the sth signature is given as $(\mathcal{T}_{T,j_T}, \mathcal{T}_{T-1,j_{T-1}}, \dots, \mathcal{T}_{2,j_2}, \mathcal{T}_{1,0})$. The one-time signature SIG_d of the message digest d is generated using the l_Tth leaf of tree $\mathcal{T}_{T,j_T}$. The one-time signature $\text{SIG}_{\mathcal{T}_{i,j_i}}$ of the root of tree $\mathcal{T}_{i,j_i}$ is generated using the l_{i-1}th leaf of tree $\mathcal{T}_{i-1,j_{i-1}}$, $i = 2,\dots,T$. The sth GMSS signature consists of

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

Lemma 2 (Signature Size). *The size of a signature is*

$$m_{\text{signature}} = \sum_{i=1}^{T}(h_i + t_{w_i}) \cdot n \text{ bits.} \tag{6}$$

Proof. A signature consists of T authentication paths ($h_i \cdot n$ bits) and T one-time signatures ($t_{w_i} \cdot n$ bits), one for each layer $i = 1,\dots,T$. Adding up yields $m_{\text{signature}}$ as shown by Equation (6) as the size of a signature. □

Following the framework of [5], we split the signature generation into two parts. The first part is the online part which computes SIG_d and outputs the signature.

The second part is the offline part that precomputes the authentication paths and one-time signatures of the roots required for upcoming signatures. In fact, the offline part performs an update of the private key and GMSS is therefore a key-evolving signature scheme [2]. Note that for the first signature $s = 0$ the offline part was done during the key generation.

Lemma 3 (Online Signature Cost). *The average cost for the online signing part is*

$$c_{\text{online}} = (2^{w_T} - 1)t_{w_T}/2 \cdot c_{\text{HASH}} + (t_{w_T} + 1)c_{\text{PRNG}}. \tag{7}$$

Proof. The generation of the one-time signature key requires one call to the PRNG to obtain the seed and t_{w_T} are necessary to obtain the secrets. The average signing costs of the Winternitz one-time signature scheme are $\big((2^{w_T} - 1)t_{w_T}\big)/2 \cdot c_{\text{HASH}}$. This leads to Equation (7). □

Next, we explain how to efficiently compute the offline signature part by distributing its cost. Our idea is based on the observation that trees in upper layers do not change frequently from one signature to the other. In the following, the values l_i, j_i correspond to the current signature s.

We begin with the precomputation of $\text{SIG}_{\mathcal{T}_{i,j_i+1}}$, $i = 2, \ldots, T$, which must be ready when the next signature uses tree $\mathcal{T}_{i,j_i+1}$. $\text{SIG}_{\mathcal{T}_{i,j_i+1}}$ is generated using the one-time signature key that corresponds to either the $(l_{i-1} + 1)$th leaf of tree $\mathcal{T}_{i-1,j_{i-1}}$ or the 0th leaf of tree $\mathcal{T}_{i-1,j_{i-1}+1}$. The latter case appears if $(l_{i-1}+1) = 0 \bmod 2^{h_{i-1}}$, i.e. we have to use the next tree in the next upper layer $i-1$. For now we assume that $\text{ROOT}_{\mathcal{T}_{i,j_i+1}}$ is known when tree $\mathcal{T}_{i,j_i}$ is used for the first time, i.e. $l_i = 0$. This certainly holds if $j_i = 0$, since $\text{ROOT}_{\mathcal{T}_{i,1}}$ was computed during the key generation. We distribute the computation of $\text{SIG}_{\mathcal{T}_{i,j_i+1}}$ over the 2^{h_i} leaves (or steps) of tree $\mathcal{T}_{i,j_i}$. If $l_i = 0$ we use $\text{ROOT}_{\mathcal{T}_{i,j_i+1}}$ and perform the initialization steps of the Winternitz one-time signature scheme. Then we compute $\text{SIG}_{\mathcal{T}_{i,j_i+1}}$ step-by-step each time we advance one leaf in tree $\mathcal{T}_{i,j_i}$. The generation of $\text{SIG}_{\mathcal{T}_{i,j_i+1}}$ is completed if $l_i = 2^{h_i} - 1$.

Lemma 4. *On average, we require*

$$c_{\text{sig}}(i) = \left\lceil \frac{(2^{w_{i-1}}-1)t_{w_{i-1}}}{2^{h_i+1}} \right\rceil c_{\text{HASH}} + \left\lceil \frac{t_{w_{i-1}}+1}{2^{h_i}} \right\rceil c_{\text{PRNG}} \tag{8}$$

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

Proof. The one-time signature $\text{SIG}_{\mathcal{T}_{i,j_i+1}}$ is generated using the Winternitz parameter of layer $i-1$ (w_{i-1}), and on average requires $(2^{w_{i-1}} - 1)t_{w_{i-1}}/2$ hash evaluations and $t_{w_{i-1}} + 1$ calls to the PRNG, see Lemma 3. Since there are 2^{h_i} leaves in the tree on layer i, the computation of $\text{SIG}_{\mathcal{T}_{i,j_i+1}}$ can be distributed over 2^{h_i} steps which yields Equation (8) as costs per step. □

Above, we assumed that $\text{ROOT}_{\mathcal{T}_{i,j_i+1}}$ is known when we first use tree $\mathcal{T}_{i,j_i}$. Hence we must precompute $\text{ROOT}_{\mathcal{T}_{i,j_i+2}}$ while using tree $\mathcal{T}_{i,j_i}$, such that it is ready when we switch to tree $\mathcal{T}_{i,j_i+1}$ and want to start generating $\text{SIG}_{\mathcal{T}_{i,j_i+2}}$. This is

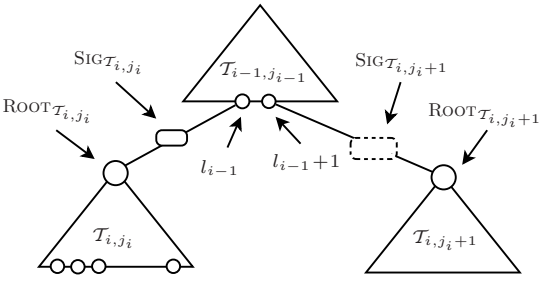

Fig. 2. $\text{SIG}_{\mathcal{T}_{i,j_i+1}}$ is precomputed from $\text{ROOT}_{\mathcal{T}_{i,j_i+1}}$ while using tree $\mathcal{T}_{i,j_i}$

done by successively computing the leaves of tree $\mathcal{T}_{i,j_i+2}$ and passing them to Algorithm 1. While using the l_ith leaf of $\mathcal{T}_{i,j_i}$ we compute the l_ith leaf of $\mathcal{T}_{i,j_i+2}$. Its computation is distributed over the $2^{h_{i+1}}$ leaves (or steps) of $\mathcal{T}_{i+1,j_{i+1}}$, the current tree on the next lower layer $i+1$. Once the leaf is generated, it is passed to treehash which partially constructs $\text{ROOT}_{\mathcal{T}_{i,j_i+2}}$. Since treehash must be called 2^{h_i} times to construct the root of a tree of height h_i, the construction of $\text{ROOT}_{\mathcal{T}_{i,j_i+2}}$ is completed once we switch from tree $\mathcal{T}_{i,j_i}$ to tree $\mathcal{T}_{i,j_i+1}$. Note that $\text{SEED}_{\mathcal{T}_{i,j_i+2},0}$, the seed required to compute the 0th leaf $\mathcal{T}_{i,j_i+2}$ was obtained during the generation of $\text{ROOT}_{\mathcal{T}_{i,j_i+1}}$ and is part of the private key. If $j_i = 0$ the seed was computed during the key generation. For the lowest layer $i = T$ the computation of the leaves has to be done at once. We also store $\text{AUTH}_{\mathcal{T}_{i,j_i+2},0}$ during the preparation of $\text{ROOT}_{\mathcal{T}_{i,j_i+2}}$.

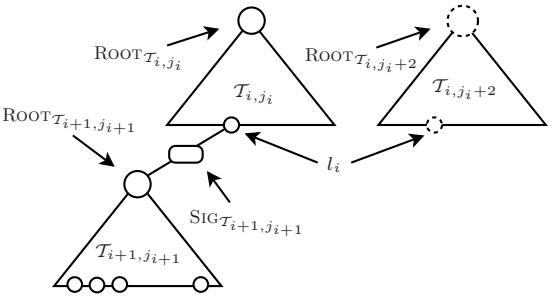

Fig. 3. Leaf l_i of tree $\mathcal{T}_{i,j_i+2}$ is precomputed while using tree $\mathcal{T}_{i+1,j_{i+1}}$

Lemma 5. *We require*

$$\begin{cases} c^1_{\text{leaf}}(i) = \left\lceil \frac{(2^{w_i}-1)t_{w_i}+1}{2^{h_{i+1}}} \right\rceil c_{\text{HASH}} + \left\lceil \frac{t_{w_i}+1}{2^{h_{i+1}}} \right\rceil c_{\text{PRNG}} \\ c^2_{\text{leaf}}(i) = h_i \cdot c_{\text{HASH}} \ \textit{(at most)} \end{cases} \tag{9}$$

operations each time we advance one leaf in $\mathcal{T}_{i+1,j_{i+1}}$ and $\mathcal{T}_{i,j_i}$, respectively, to compute $\text{ROOT}_{\mathcal{T}_{i,j_i+2}}$.

Proof. The generation of the l_ith leaf of tree $\mathcal{T}_{i,j_i+2}$ requires $((2^{w_i}-1)t_{w_i}+1)c_{\text{HASH}}$ and $(t_{w_i}+1)c_{\text{PRNG}}$. Since there are $2^{h_{i+1}}$ leaves in the tree on layer $i+1$, the computation of the l_ith leaf can be distributed over $2^{h_{i+1}}$ steps which yields $c^1_{\text{leaf}}(i)$. The while-loop of treehash requires at most h_i hash function evaluations which yields $c^2_{\text{leaf}}(i)$. □

Next, we describe the precomputation of $\text{AUTH}_{\mathcal{T}_{i,j_i},l_i+1}$, the authentication path of the next leaf of tree $\mathcal{T}_{i,j_i}$. We use an algorithm proposed by Szydlo in [13]. This algorithm uses $h_i - 1$ instances of treehash to compute the upcoming authentication nodes. Given a leaf index l_i, Szydlo's algorithm firstly checks if a new instance of treehash must be generated. Then it spends at most h_i computational units, which are either hash function evaluations for the while-loop of treehash or leaf calculations. Again, the computation of the required leaves is distributed over the $2^{h_{i+1}}$ leaves of $\mathcal{T}_{i+1,j_{i+1}}$. When using the leaf $l_{i+1}=0$ of tree $\mathcal{T}_{i+1,j_{i+1}}$, we perform the initialization steps of Szyldo's algorithm and decide (1) if a new instance of treehash must be generated and (2) how many new leaves are required by the active treehash instances. Those leaves are computed step-by-step as explained above. If $l_{i+1} = 2^{h_{i+1}} - 1$ the calculation of the required leaves is completed and we pass them to Szydlo's algorithm which updates the treehash instances and outputs $\text{AUTH}_{\mathcal{T}_{i,j_i},l_i+1}$. Note that $\text{AUTH}_{\mathcal{T}_{i,j_i},0}$ is stored during the construction of $\text{ROOT}_{\mathcal{T}_{i,j_i}}$ and therefore already available if $l_i = 0$. Also, $\text{AUTH}_{\mathcal{T}_{T,j_T},l_T+1}$ must be computed at once.

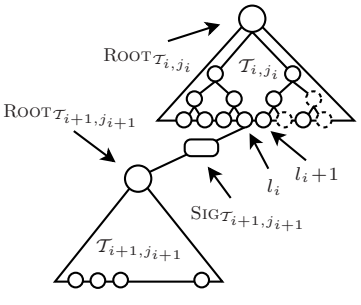

Fig. 4. Leaves required for $\text{AUTH}_{\mathcal{T}_{i,j_i},l_i+1}$ are precomputed while using tree $\mathcal{T}_{i+1,j_{i+1}}$

Lemma 6. *We require at most*

$$\begin{cases} c^1_{\text{auth}}(i) = h_i \cdot c^1_{\text{leaf}}(i) + \left\lceil \frac{2^{h_i-2}}{2^{h_{i+1}}} \right\rceil c_{\text{PRNG}} \\ c^2_{\text{auth}}(i) = h_i \cdot c_{\text{HASH}} \end{cases} \tag{10}$$

operations each time we advance one leaf in $\mathcal{T}_{i+1,j_{i+1}}$ and $\mathcal{T}_{i,j_i}$, respectively, to compute $\text{AUTH}_{\mathcal{T}_{i,j_i},l_i+1}$.

Proof. Szydlo's algorithm initializes at most one instance of treehash in each iteration. We need at most 2^{h_i-2} calls to the PRNG to obtain the initial seed for

the leaf required by this instance from the current seed. In the worst case, the active treehash instances require the computation of h_i leaves. The computation of those h_i leaves and the 2^{h_i-2} calls to the PRNG are distributed over the $2^{h_{i+1}}$ steps in the tree on layer $i+1$. This yields $c^1_{\text{auth}}(i)$. At most $h_i - 1$ hash evaluations are spend on the while-loops of the active treehash instances. One hash evaluation is required by the initialization steps of Szydlo's algorithm. This yields $c^2_{\text{auth}}(i)$. □

The following lemma considers the worst case costs of the offline part. It assumes that for the next signature, we have to advance one leaf in each tree $\mathcal{T}_{i,j_i}$, $i = 1, \ldots, T-1$. Note that this is equivalent to advancing from tree $\mathcal{T}_{i,j_i}$ to $\mathcal{T}_{i,j_i+1}$ in all layers $i = 2, \ldots, T$.

Lemma 7 (Offline Signature Cost and Memory). *The worst case costs and the memory for the offline part are*

$$\begin{aligned} c_{\text{offline}} &= \sum_{i=2}^{T} \left(c_{\text{sig}}(i) + c^1_{\text{leaf}}(i) + c^2_{\text{leaf}}(i)\right) + \sum_{i=1}^{T} \left(c^1_{\text{auth}}(i) + c^2_{\text{auth}}(i)\right) \\ m_{\text{offline}} &= \left(\sum_{i=2}^{T} (t_{w_{i-1}} + 4h_i) + 3h_1\right) \cdot n \text{ bits.} \end{aligned} \tag{11}$$

Proof. In the worst case, we have to advance one leaf in the current tree on all layers $i = 1, \ldots, T-1$. The cost for this case are obtained by adding the costs for the precomputation of $\text{SIG}_{\mathcal{T}_{i,j_i+1}}$ and $\text{ROOT}_{\mathcal{T}_{i,j_i+2}}$ for all layers $i = 2, \ldots, T$ and $\text{AUTH}_{\mathcal{T}_{i,j_i},l_i+1}$ for all layers $i = 1, \ldots, T$. This yields c_{offline}. During the offline part, we have to store $\text{SIG}_{\mathcal{T}_{i,j_i+1}}$ which requires $t_{w_{i-1}} \cdot n$ bits and the stack required by treehash to construct $\text{ROOT}_{\mathcal{T}_{i,j_i+2}}$ which requires $h_i \cdot n$ bits, $i = 2, \ldots, T$. Further, we have to store $\text{AUTH}_{\mathcal{T}_{i,j_i},l_i+1}$ and some temporary nodes required by Szydlo's algorithm which require $3h_i \cdot n$ bits, $i = 1, \ldots, T$. This yields m_{offline}. □

2.5 Verification

The verification process of GMSS is essentially the same as for MSS and CMSS. The verifier knows the public key $\text{ROOT}_{\mathcal{T}_{1,0}}$ and the parameter set $\mathcal{P}$ used by the signer. At first, he verifies the one-time signature SIG_d of the digest d using the Winternitz parameter w_T. Doing so, he obtains the verification key for this signature, i.e. leaf l_T of tree $\mathcal{T}_{T,j_T}$. Then, the verifier repeats the following steps for $i = T, \ldots, 2$. (1) use l_i and $\text{AUTH}_{\mathcal{T}_{i,j_i},l_i}$ to compute $\text{ROOT}_{\mathcal{T}_{i,j_i}}$. (2) use $\text{ROOT}_{\mathcal{T}_{i,j_i}}$ and verify $\text{SIG}_{\mathcal{T}_{i,j_i}}$ and obtain l_{i-1}. Finally, the verifier uses l_1 and $\text{AUTH}_{\mathcal{T}_{1,j_1},l_1}$ to obtain $\text{ROOT}_{\mathcal{T}_{1,0}}$. If $\text{ROOT}_{\mathcal{T}_{1,0}}$ matches the signers public key, the signature is accepted.

Lemma 8 (Verification Cost). *The average cost for the verification is*

$$c_{\text{verify}} = \sum_{i=1}^{T} \left((2^{w_i} - 1)t_{w_i}/2 + h_i\right) c_{\text{HASH}}. \tag{12}$$

Proof. On average, $\big((2^{w_i}-1)t_{w_i}\big)/2$ hash evaluations are required to verify an one-time signature. Using a leaf and an authentication path to construct a root requires h_i hash evaluations. Since there is a one-time signature and an authentication path for each layer the average costs for the verification are c_{verify}. □

3 GMSS in Practice

In this section, we give practical GMSS parameters that simultaneously allow for a large signature capacity, good efficiency and small bandwidth, and describe how to integrate GMSS in a protocol such as SSL with a client/server architecture.

3.1 Choosing $\mathcal{P}$

To find an optimal parameter set, we consider following optimization problem: given certain bounds on the signature capacity as well as the key generation, signature generation and verification time, find the parameter set which provides the smallest signatures. We formulated this optimization problem as mixed integer program (MIP) using Zimpl [9]. The constraints of this MIP are the equations for the key generation (3), signature generation (7),(11) and verification (12) time and the signature size (6). Note that the worst cast cost of Szydlo's algorithm for the authentication path computation shown in Equation (10) occurs only once per tree. To compute the parameter sets, we use the average costs of Szydlo's algorithm, which are

$$\Big(h_i/2 \cdot \big((2^{w_i}-1)t_{w_i}+1\big) + h_i/2 - 1\Big)c_{\text{HASH}} + \Big(h_i + t_{w_i}\Big)c_{\text{PRNG}}$$

for a tree on layer i. To solve the MIP, we used the free solver SCIP [1]. Using different bounds for the signature generation and verification time, we obtained the following parameter sets $\mathcal{P}_k = \big(T, (h_1, \ldots, h_T), (w_1, \ldots, w_T)\big)$ that allow up to 2^k signatures.

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

Table 1 shows timings (t) and memory requirements (m) for the parameter sets when using a 160 bit hash function. The size of the public key is $m_{\text{pubkey}} = 20$

Table 1. Timings and memory requirements

	m_{offline}	m_{privkey}	$m_{\text{signature}}$	t_{keygen}	t_{sign}	t_{verify}
$\mathcal{P}_{40}$	3160 bytes	1640 bytes	1860 bytes	723 min	26.0 ms	19.6 ms
$\mathcal{P}'_{40}$	3200 bytes	1680 bytes	2340 bytes	390 min	10.7 ms	10.7 ms
$\mathcal{P}_{80}$	7320 bytes	4320 bytes	3620 bytes	1063 min	26.1 ms	18.1 ms
$\mathcal{P}'_{80}$	7500 bytes	4500 bytes	4240 bytes	592 min	10.1 ms	10.1 ms

bytes for all parameter sets. To get timings, we use the ratio $c_{\text{HASH}} = c_{\text{PRNG}} = 0.002$ ms, which we obtained using a Java implementation of SHA1 on a Pentium dualcore 1.8GHz.

This table clarifies the flexibility of GMSS. $\mathcal{P}_{40}$ and $\mathcal{P}_{80}$ provide the shortest possible signatures. In case of $\mathcal{P}_{40}$, the signature size is reduced more that 26% compared to what was stated in [3]. $\mathcal{P}'_{40}$ and $\mathcal{P}'_{80}$ provide very fast signature generation and verification times, at the expense of larger signatures.

Note that a large portion of the signature cost is required for the precomputation of the upcoming authentication paths. One possibility to circumvent this is to use a tree of small height (≤ 5) in the lowest layer and to store it completely in memory. Then the authentication paths can be obtained for free. In case of a 160 bit hash function, storing a tree of height five requires 1,260 bytes.

3.2 Message Flow and Application to SSL/TLS

Finally, we describe how signatures should be transmitted when the signer and the verifier are connected during the signing process as in case of the SSL/TLS protocol. Thanks to the online/offline strategy [5], the server has all signature parts ready from the beginning of the transaction, except for the one-time signature of the message digest SIG_d. The server delays the generation of the online signature SIG_d and sends only the first offline signature part $\text{SIG}_{\mathcal{T}_{T,j_T}}$ and $\text{ROOT}_{\mathcal{T}_{T,j_T}}$ to the client. Then, the client demonstrates his honesty by sending leaf l_{T-1} (the verification key of $\text{SIG}_{\mathcal{T}_{T,j_T}}$) back to the server. The client has to spend some computational effort to obtain l_{T-1}, namely he has to verify $\text{SIG}_{\mathcal{T}_{T,j_T}}$, while the server does not have to do anything. While the server is waiting for the client to send l_{T-1}, he can also start with next the offline part of the signing process. When the server receives the correct leaf l_{T-1}, he sends the remaining parts of the signature and starts to compute SIG_d. In the same time, the client can verify the remaining one-time signatures $\text{SIG}_{\mathcal{T}_{i,j_i}}$, $i = 2, \ldots, T-1$

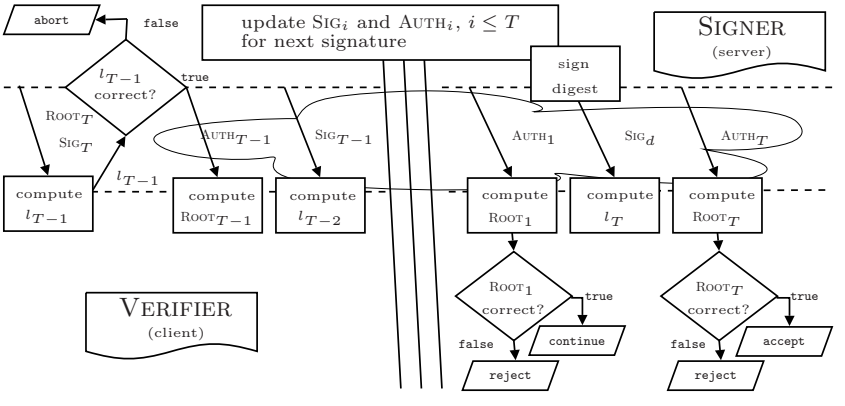

Fig. 5. Message Flow for SSL/TLS

that he receives from the server, and compares the root of the top tree to the server's public key. In the final step, the server sends SIG_d to the client who verifies its correctness.

The overhead of our protocol for the server is just $2n$ bits memory to store $\text{ROOT}_{\mathcal{T}_{T,j_T}}$ and l_{T-1}. The benefits are as follows: it minimizes bandwidth and server-side calculations in case of DoS attacks, and optimizes the latency of the transaction. Indeed, the protocol can be stopped early in case of DoS. In addition, the signature generation, transmission and verification stages are performed concurrently.

4 Conclusion

We presented the generalized Merkle signature scheme (GMSS). GMSS is parameterized by the parameter set $\mathcal{P}$, that allows a great degree of freedom in choosing the signature capacity, the signature generation and verification timings, and the signature size. GMSS uses a scheduling strategy to precompute upcoming signatures. This drastically reduces the signature generation time and in turn allows to choose parameters that provide smaller signatures. For a signature capacity of 2^{40}, the signature size is 1,860 bytes, where signature generation and verification requires 26 ms and 19.6 ms, respectively. GMSS is the first Merkle-type signature scheme that maintains its efficiency even if the signature capacity *cryptographically unlimited*, i.e. 2^{80}. In that case, the signature size is 3,620 bytes, where signature generation and verification requires 26.1 ms and 18.1 ms, respectively. For both signature capacities $(2^{40}, 2^{80})$, it is also possible to find parameter sets such that the signature generation and verification time is only 10 ms. This makes GMSS a serious competitor to commonly used signature schemes such as RSA or ECDSA. Furthermore, GMSS does not rely on number theoretic problems and is not vulnerable to quantum computer attacks. Finally, we proposed a DoS-resilient protocol for SSL/TTL that minimizes the total latency of a signature exchange.

References

1. Tobias Achterberg. SCIP - a framework to integrate constraint and mixed integer programming. Technical Report 04-19, Zuse Institute Berlin, 2004.
2. Mihir Bellare and Sara K. Miner. A forward-secure digital signature scheme. In *Proc. Advances in Cryptology (Crypto'99)*, volume 1666 of *Lecture Notes in Computer Science*, pages 431–238. Springer-Verlag, 1999.
3. 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.
4. 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.

5. Shimon Even, Oded Goldreich, and Silvio Micali. On-line/off-line digital signatures. In *Proc. Advances in Cryptology (Crypto'89)*, volume 435 of *Lecture Notes in Computer Science*, pages 263–277. Springer-Verlag, 1990.
6. Digital signature standard (DSS). FIPS PUB 186-2, 2007. Available at `http://csrc.nist.gov/publications/fips/`.
7. Luis Carlos Coronado García. On the security and the efficiency of the Merkle signature scheme. Cryptology ePrint Archive, Report 2005/192, 2005. `http://eprint.iacr.org/`.
8. 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.
9. Thorsten Koch. Rapid mathematical programming. Technical Report 04-58, Zuse Institute Berlin, 2004.
10. Arjen K. Lenstra and Eric R. Verheul. Selecting cryptographic key sizes. *Journal of Cryptology*, 14(4):255–293, 2001.
11. 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.
12. Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proc. 35th Annual Symposium on Foundations of Computer Science*, pages 124–134. IEEE Comput. Soc. Press, 1994.
13. Michael Szydlo. Merkle tree traversal in log space and time (preprint). Available at `http://www.szydlo.com/`.
14. 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.

A Merkle's Tree Authentication Scheme

The tree authentication scheme was proposed by Merkle in [11] for using multiple one-time signature instances with a single "master" public key. Merkle's tree authentication scheme in conjunction with a one-time signature scheme is referred to as the Merkle signature scheme (MSS).

Keypair Generation: The signer first generates 2^h one-time key pairs. The one-time verification keys form the leaves of a binary hash tree of height h, called Merkle tree. The value of an inner node is obtained by hashing the concatenation of the values of its two children. Iterating yields the root of the tree which is the MSS public key, and the private key consists of the 2^h one-time signature keys.

Signature Generation: To authenticate the s-th leaf, i.e. the s-th verification key, the s-th authentication path is required. This authentication path consists of the $h-1$ siblings of the $h-1$ nodes on the path from the sth leaf to the root. The s-th Merkle signature consists of four parts: first the index $s \in \{0, \ldots, 2^h - 1\}$ of the selected one-time signature; second, the one-time signature of data d generated with the s-th signature key; third, the s-th verification key; and fourth, the authentication path for the s-th verification key.

Verification: After verifying the one-time signature of d, the verifier has to validate the authenticity of the supplied verification key. Using the index s and the authentication path of the s-th leaf, he re-computes the path from the sth leaf to the root. To do so, he hashes the concatenation of the s-th leaf and first node in the authentication path to obtain the first node on the path to the root (the order for concatenating is decided according to the index s). By successively concatenating the hashed nodes and authentication nodes, the verifier can eventually recover the root itself. If the root matches the signer's public key, the signature is valid.

B The Winternitz One-Time Signature Scheme

The Winternitz OTS [11], described in detail in [4], is parameterized by w, which allows a trade-off between the signature cost and size.

Keypair Generation: The keypair generation produces t_w random values x_1, x_2, ..., x_{t_w}, where $t_w = \lceil n/w \rceil + \lceil (\lfloor \log_2(\lceil n/w \rceil) \rfloor + 1 + w)/w \rceil$. Then, the one-time signature key is $X = (x_1, \ldots, x_{t_w})$, and the one-time verification key $Y = H\big(H^{2^w-1}(x_1) \| \ldots \| H^{2^w-1}(x_{t_w})\big)$, where $H^k(x)$ denotes the hash function applied k times and $\|$ the concatenation of two strings. The cost for the key pair generation is $c_{\text{OTSkeygen}} = \big((2^w - 1)t_w + 1\big)c_{\text{HASH}} + t_w \cdot c_{\text{PRNG}}$.

Signature Generation: For an n-bit message digest d, the OTS is computed as follows. The binary representation of d (possibly padded) is divided into $\lceil n/w \rceil$ blocks of length w: $b_1, \ldots, b_{\lceil n/w \rceil}$. Next, the checksum $C = \sum_{k=1}^{\lceil n/w \rceil} 2^w - b_k$ is calculated. The binary representation of C (possibly padded) is again divided into blocks of length w: $b_{\lceil n/w \rceil + 1}, \ldots, b_{t_w}$. Finally, the signature of d is $\text{SIG} = (\sigma_1, \ldots, \sigma_{t_w})$, where $\sigma_k = H^{b_k}(x_k)$, $k = 1, \ldots, t_w$. The signature size is $t_w \cdot n$ bits and the average cost for signing is $c_{\text{OTSsign}} = (2^w - 1)t_w/2 \cdot c_{\text{HASH}}$.

Verification: Given the digest d, the signature $\text{SIG} = (\sigma_1, \ldots, \sigma_{t_w})$, and the verification key Y, the verifier computes $b_1, \ldots, b_{t_w}$ just like the signer and then calculates $y_k = H^{2^w-1-b_k}(\sigma_k)$, $k = 1, \ldots, t_w$. The signature is accepted if $H(y_1 \| \ldots \| y_{t_w})$ equals the verification key Y. The average verification cost is exactly the same as the signature cost.