Optimal Parameters for XMSS^{MT}

Andreas Hülsing*, Lea Rausch, and Johannes Buchmann

Cryptography and Computeralgebra
Department of Computer Science
TU Darmstadt, Germany
huelsing@cdc.informatik.tu-darmstadt.de

Abstract. We introduce Multi Tree XMSS (XMSS^{MT}), a hash-based signature scheme that can be used to sign a virtually unlimited number of messages. It is provably forward and hence EU-CMA secure in the standard model and improves key and signature generation times compared to previous schemes. XMSS^{MT} has — like all practical hash-based signature schemes — a lot of parameters that control different trade-offs between security, runtimes and sizes. Using linear optimization, we show how to select provably optimal parameter sets for different use cases.

Keywords: hash-based signatures, parameter selection, linear optimization, forward secure signatures, implementation.

1 Introduction

Digital signatures are among the most important cryptographic primitives in practice. They are used to secure communication protocols like SSL/TLS and SSH, software updates, or to replace handwritten signatures. Hash-based signature schemes are an interesting alternative to the signature schemes used today. Not only because they are assumed to resist quantum computer aided attacks, but also because of their fast signature generation and verification times as well as their strong security guarantees. The latest hash-based signature schemes [3,9] come with a standard model security proof and outperform RSA in many settings regarding runtimes.

Practical hash-based signature schemes have a lot of parameters that control several trade-offs between runtimes and sizes. Hence, an important open problem on the way of making hash-based signatures practical is parameter selection. In previous works [3, 4, 6, 7, 9] it was shown how to select secure parameters using the exact security reductions, but this still leaves too many possible parameter choices to apply a brute-force search. Even applying reasonable restrictions on the parameters, the search space can easily grow to the order of 2^{80} and more. In this work we introduce Multi Tree XMSS (XMSS^{MT}), a new hash-based signature scheme that contains the latest hash-based signature schemes XMSS

* Supported by grant no. BU 630/19-1 of the German Research Foundation (www.dfg.de).

A. Cuzzocrea et al. (Eds.): CD-ARES 2013 Workshops, LNCS 8128, pp. 194–208, 2013.

and XMSS^{+} as special cases and propose a systematic way to select the optimal parameter set for a given scenario for XMSS^{MT}.

XMSS^{MT} improves XMSS in a way, that it allows for a virtually unlimited number of signatures, while it preserves its desirable properties. XMSS^{MT} has minimal security requirements, i.e. it can be proven EU-CMA secure in the standard model, when instantiated with a pseudorandom function family and a second preimage resistant hash function. Moreover, XMSS^{MT} is forward secure [1] and introduces a new trade-off between signature and key generation time on the one hand and signature size on the other hand. XMSS^{MT} can be viewed as applying the tree chaining idea introduced in [6] to XMSS and combine it with the improved distributed signature generation proposed in [9].

For XMSS^{MT}, we show how to model the parameter selection problem as a linear optimization problem. A straight forward approach would lead many non-linear constraints. To overcome this problem, we use the generalized lambda method [13] for linearization. The resulting model can then be solved using the Simplex algorithm [8]. As the general lambda method is exact, the Simplex algorithm outputs a provably optimal solution under the given constraints. We made the model flexible, such that it can be used for different use cases with different requirements using different inputs. We present results for demonstrative use cases and compare them to possible parameter choices from previous work. However, the code is available on the corresponding authors home page and can be used to select optimal parameters for other use cases. Linear optimization was used before in [4] to find optimal parameters. Unfortunately, the results given there do not carry over to the new hash-based signature schemes, as less parameters were taken into account and the authors of [4] do not provide details about how to model the problem.

Organization. We start with an introduction to XMSS^{MT} in Section 2, where we also discuss theoretical runtimes and parameter sizes as well as security and correctness of the scheme. In Section 3 we show how to model the parameter selection problem for XMSS^{MT} as a linear optimization problem and present optimal parameters for exemplary use cases in Section 4. We discuss our results and draw a conclusion in Section 5.

2 Multi Tree XMSS - XMSS^{MT}

In this section we introduce XMSS^{MT}. For a better understanding, we first give a brief description of the scheme. A hash-based signature scheme starts with a one-time signature scheme (OTS). To obtain a many time signature scheme, many instances of the OTS are used and their public key are authenticated using a binary hash tree, called Merkle-Tree. The root of the tree is the public key of the scheme and the secret key is the seed of a pseudorandom generator that is used to generate the OTS secret keys. A signature contains an index, the OTS signature, and the so called authentication path which contains the siblings of the nodes on the way from the used OTS key pair to the root. For XMSS^{MT} we extend this using many layers of trees. The trees on the lowest layer are used to

sign the messages and the trees on higher layers are used to sign the roots of the trees on the layer below. The public key contains only the root of the tree on the top layer. A signature consists of all the signatures on the way to the highest tree. A graphical representation of the scheme can be found in Appendix E. In the following we describe the construction in detail, starting with the description of some building blocks. Afterwards we describe the algorithms of the scheme and present an theoretical analysis of its performance, security and correctness.

Parameters. For security parameter $n \in \mathbb{N}$, the basic building blocks of XMSS^{MT} are a pseudorandom function family $\mathcal{F}_n = \{\mathrm{F}_K : \{0,1\}^n \to \{0,1\}^n | K \in \{0,1\}^n\}$, and a second preimage resistant hash function $\mathrm{H} : \{0,1\}^{2n} \to \{0,1\}^n$. Further parameters are the number of layers $d \in \mathbb{N}$, the binary message length m and one parameter set per layer. For a layer $0 \leq i \leq d-1$ a parameter set contains the tree height $h_i \in \mathbb{N}$, the so called BDS parameter $k_i \in \mathbb{N}$ with the restrictions $k_i < h_i$ and $h_i - k_i$ is even and the Winternitz parameter $w_i \in \mathbb{N}, w_i \geq 2$. To enable improved distributed signature generation we require $(h_{i+1} - k_{i+1} + 6)/2 \leq 2^{h_i - k_i + 1}$ for $0 \leq i < d-1$, as well as $(h_0 - k_0)/2 \geq d-1$. A XMSS^{MT} key pair can be used to sign $h = \sum_{i=0}^{d-1} h_i$ messages of m bits. The remaining parameters define the various trade-offs explained below. These parameters are publicly known.

Winternitz OTS. XMSS^{MT} uses the Winternitz-OTS (W-OTS) from [2]. W-OTS uses the function family $\mathcal{F}_n$ and a value $X \in \{0,1\}^n$ that is chosen during XMSS key generation. Besides message length m and Winternitz parameter w it has another parameter $\ell = \ell(m, w)$ that is a function of m and w. The W-OTS secret key, public key and signatures consist of ℓ bit strings of length n. An important property of W-OTS is that the public key can be computed from a valid signature. The sizes of signature, public, and secret key are ℓn bits. The runtimes for key generation, signature generation and signature verification are all bounded by $\ell(w-1)$ evaluations of elements from $\mathcal{F}_n$. The Winternitz parameter w controls a time - space trade-off, as ℓ shrinks logarithmically in w. For more detailed information see Appendix A.

XMSS Tree. XMSS^{MT} uses the XMSS tree construction, originally proposed in [7]. An XMSS tree of height h is a binary tree with $h+1$ levels. The leaves are on level 0, the root on level h. The nodes on level j, are denoted by $N_{i,j}$, for $0 \leq i < 2^{h-j}, 0 \leq j \leq h$. To construct the tree, h bit masks $B_j \in \{0,1\}^{2n}$, $0 < j \leq h$ and the hash function H, are used. $N_{i,j}$, for $0 < j \leq h$, is computed as $N_{i,j} = \mathrm{H}((N_{2i,j-1} || N_{2i+1,j-1}) \oplus B_j)$.

Leaf Construction. The leaves of a XMSS tree are the hash values of W-OTS public keys. To avoid the need of a collision resistant hash function, another XMSS tree is used, called L-tree. The ℓ leaves of an L-tree are the ℓ bit strings $(\mathsf{pk}_0, \ldots, \mathsf{pk}_\ell)$ from the corresponding public key. If ℓ is not a power of 2, a left node that has no right sibling is lifted to a higher level of the L-tree until it becomes the right sibling of another node. For the L-tree, the same hash function

as above but $\lceil \log \ell \rceil$ new bitmasks are used. These bitmasks are the same for all L-trees.

Pseudorandom Generator. The W-OTS key pairs belonging to one XMSS tree are generated using two pseudorandom generators (PRG) build using $\mathcal{F}$. The stateful forward secure PRG FsGen : $\{0,1\}^n \to \{0,1\}^n \times \{0,1\}^n$ is used to generate one seed value per W-OTS keypair. Then the seed is expanded to the ℓ W-OTS secret key bit strings using $\mathcal{F}_n$. FsGen starts from a uniformly random state $S_0 \xleftarrow{\$} \{0,1\}^n$. On input of a state S_i, FsGen generates a new state S_{i+1} and a pseudorandom output R_i: $\text{FsGen}(S_i) = (S_{i+1}||R_i) = (\text{F}_{S_i}(0)||\text{F}_{S_i}(1))$. The output R_i is used to generate the ith W-OTS secret key $(\mathsf{sk}_1, \ldots, \mathsf{sk}_\ell)$ as $\mathsf{sk}_j = \text{F}_{R_i}(j-1), 1 \leq j \leq \ell$.

Tree Traversal - The BDS Algorithm. To compute the authentication paths for the W-OTS key pairs of one XMSS tree, XMSS^{MT} uses the BDS algorithm [5]. The choice of this algorithm is already a time - memory trade-off. Instead of computing every node of the authentication path when needed, BDS uses some storage to decrease the worst case runtime. The BDS algorithm uses the TreeHash algorithm (see Algorithm 1) as a subroutine. The BDS algorithm reduces the worst case signing time from $2^h - 1$ to $(h-k)/2$ leaf computations and evaluations of TreeHash. More specifically, the BDS algorithm does three things. First, it uses the fact that a node that is a left child can be computed from values that occurred in an authentication path before, spending only one evaluation of H. Second, it stores the right nodes from the top k levels of the tree during key generation. Third, it distributes the computations for right child nodes among previous authentication path computations. It schedules one instance of TreeHash per tree level. The computation of the next right node on a level starts, when the last computed right node becomes part of the authentication path. BDS uses a state $\mathsf{State}_{\text{BDS}}$ of $2(h-k)$ states of FsGen and at most $\left(3h + \left\lfloor \frac{h}{2} \right\rfloor - 3k - 2 + 2^k\right)$ tree nodes that is initialized during key generation. To compute the authentication paths, the BDS algorithm spends $(h-k)/2$ leaf computations and evaluations of TreeHash as well as $(h-k)$ calls to F to update its state per signature. For more details see [5].

Algorithm 1. TreeHash

Input: Stack Stack, node N_1
Output: Updated stack Stack

1. **While** top node on Stack has same height as N_1 **do**
   (a) $t \longleftarrow N_1.height() + 1$
   (b) $N_1 \longleftarrow \text{H}\left((\mathsf{Stack}.pop()||N_1) \oplus B_t\right)$
2. $\mathsf{Stack}.push(N_1)$
3. **Return** Stack

Key Generation. The XMSS^{MT} key generation algorithm takes as input all of the above parameters. First, the $\max_{0 \leq i \leq d-1}\{h_i + \lceil \log \ell_i \rceil\}$ bitmasks and X are chosen uniformly at random. The same bitmasks and X are used for all layers. Then, the root of the first XMSS tree on each layer is computed. This is done in an ordered way, starting from layer 0. For the tree TREE_i on layer i the initial state of FsGen, $S_{0,i}$ is chosen uniformly at random and a copy of it is stored as part of the secret key SK. The tree is constructed using the TreeHash algorithm above. Starting with an empty stack Stack_i and $S_{0,i}$, all 2^{h_i} leaves are successively generated and used as input to the TreeHash algorithm to update Stack_i. This is done updating S_i and generating the W-OTS key pairs using its output. The W-OTS public key is then used to compute the corresponding leaf using an L-tree which in turn is used to update the current Stack_i using TreeHash. Then the W-OTS key pair is deleted. After all 2^{h_i} leaves were processed by TreeHash, the only value on Stack_i is the root ROOT_i of TREE_i. When ROOT_i $0 \leq i < d-1$, is computed, it is signed using the first W-OTS key pair of TREE_{i+1}, which is computed next. This signature σ_{i+1} can be extracted while TREE_i is generated and hence does not need any additional computation. Then σ_{i+1} is stored as part of SK. If the highest layer $d-1$ is reached, ROOT_{d-1} is stored in the public key PK. During the computation of ROOT_i, the state of the BDS algorithm $\mathsf{State}_{\text{BDS},i}$ is initialized. For details see [5].

Finally, the data structures for the next trees are initialized: For the next tree NEXT_i on each layer $0 \leq i < d-1$ a FsGen state $S_{n,i}$ is chosen uniformly at random and a new TreeHash stack $\mathsf{Stack}_{next,i}$ is initialized. Summing up, SK consists of the states $(S_{0,i}, \mathsf{State}_{\text{BDS},i}), 0 \leq i \leq d-1$ and the $d-1$ signatures σ_i, $0 < i \leq d-1$. Additionally, it contains $d-1$ FsGen states $S_{n,i}$, $d-1$ TreeHash stacks $\mathsf{Stack}_{next,i}$ and $d-1$ BDS states $\mathsf{State}_{\text{BDS},n,i}$ for the next trees NEXT_i on layer $0 \leq i < d-1$. The public key PK consists of the $\max_{0 \leq i \leq d-1}\{h_i + \lceil \log \ell_i \rceil\}$ bitmasks, the value X and ROOT_{d-1}.

Signature generation. The signature generation algorithm takes as input a m bit message M, the secret key SK, and the index i, indicating that this is the ith message signed with this keypair. The signature generation algorithm consists of two phases. First, M is signed. A XMSS^{MT} signature $\Sigma = (i, \sigma_0, \mathsf{Auth}_0, \sigma_1, \mathsf{Auth}_1, \ldots, \sigma_d, \mathsf{Auth}_d)$ contains the index i, the W-OTS signature σ_0 on the message M, the corresponding authentication path for TREE_0 and the W-OTS signatures on the roots of the currently used trees together with the corresponding authentication paths. The only thing that has to be computed is σ_0 — the W-OTS signature on message M using the ith W-OTS key pair on the lowest layer. All authentications paths and the W-OTS signatures on higher layers are already part of the current secret key.

The second phase is used to update the secret key. Therefor BDS is initialized with $\mathsf{State}_{\text{BDS},0}$ and receives $(h_0 - k_0)/2$ updates. If not all of these updates are needed to update $\mathsf{State}_{\text{BDS},0}$, i.e. all scheduled TreeHash instances are finished and there are still updates left, the remaining updates are used for the upper trees. On the upper layers, not only the BDS state has to be updated, but while one leaf is used, one leaf in the next tree must be computed and $h_i - k_i$

FsGen states must be updated. This means that remaining updates from layer zero are first used to update $\mathsf{State}_{\mathrm{BDS},1}$. If all scheduled TreeHash instances in $\mathsf{State}_{\mathrm{BDS},1}$ are finished, one update is used to compute a new leaf for Next_1 and to update $\mathsf{Stack}_{next,1}$ afterwards. The next update is used to for the FsGen states. If layer 1 does not need anymore updates, the remaining updates are forwarded to layer 2 and so on, until either all updates are used or all tasks are done. Finally, one leaf of the next tree is computed and used as input for TreeHash to update $\mathsf{Stack}_{next,0}$.

A special case occurs if $i \mod 2^{h_0} = 2^{h_0} - 1$. In this case, the last W-OTS key pair of the current Tree_0 was used. This means that for the next signature a new tree is needed on every layer j with $i \mod 2^{h_j} = 2^{h_j} - 1$. For all these layers, $\mathsf{Stack}_{next,j}$ already contains the root of Next_j. So, Tree_{j+1} is used to sign Root_j. Each signature is counted as one update. In case not all updates are needed, remaining updates are forwarded to the first layer that did not get a new tree. In SK, $\mathsf{State}_{\mathrm{BDS},j}$, S_j, and Σ_j are replaced by the newly computed data. Afterwards, new data structures for the next tree on layer j are initialized and used to replace the ones in SK. Finally, the signature SIG, the updated secret key SK and $i + 1$ are returned.

Signature verification. The signature verification algorithm takes as input a signature $\Sigma = (i, \sigma_0, \mathsf{Auth}_0, \sigma_1, \mathsf{Auth}_1, \ldots, \sigma_d, \mathsf{Auth}_d)$, the message M and the public key PK. To verify the signature, σ_0 and M are used to compute the corresponding W-OTS public key. The corresponding leaf $N_{0,j}$ of Tree_0 is constructed and used together with Auth_0 to compute the path $(P_0, \ldots, P_{h_0})$ to the root of Tree_0, where $P_0 = N_{0,j}$, $j = i \mod 2^{h_0}$ and

$$P_c = \begin{cases} \mathrm{H}((P_{c-1} || \mathsf{Auth}_{c-1,0}) \oplus B_c), & \text{if } \lfloor j/2^c \rfloor \equiv 0 \mod 2 \\ \mathrm{H}((\mathsf{Auth}_{c-1,0} || P_{c-1}) \oplus B_c), & \text{if } \lfloor j/2^c \rfloor \equiv 1 \mod 2 \end{cases}$$

for $0 \leq c \leq h_0$. This process is then iterated for $1 \leq a \leq d-1$, using the output of the last iteration $P_{h_{a-1}} = \mathrm{Root}_{a-1}$ as message, σ_a, Auth_a and $j = \left\lfloor i/2^{\sum_{b=0}^{a-1} h_b} \right\rfloor \mod 2^{h_a}$. If the output of the last iteration $P_{h_{d-1}}$ equals the root value contained in PK, Root_{d-1}, the signature is assumed to be valid and the algorithm returns 1. In any other case it returns 0.

2.1 Analysis

In the following we provide an analysis of XMSS^{MT}. A discussion of correctness and security can be found in appendices B, C and D. In the following we discuss key and signature sizes and the runtimes of the algorithms.

First we look at the sizes. A signature contains the index and d pairs of W-OTS signature and authentication path. Hence a signature takes $24 + n \cdot \sum_{i=0}^{d-1} (\ell_i + h_i)$ bits, assuming we reserve three bytes for the index. The public key contains the bitmasks, X and Root_{d-1}. Thus, the public key size is $n \cdot (\max_{0 \leq i \leq d-1} \{h_i + \lceil \log \ell_i \rceil\} + 2)$ bits. The secret key contains one FsGen state and one BDS state consisting of $2(h_i - k_i)$ FsGen states and no more than $(3h_i + \lfloor \frac{h_i}{2} \rfloor - 3k_i - 2 + 2^{k_i})$

tree nodes [5] per currently used tree TREE_i. In addition it contains the $d-1$ W-OTS signatures $\sigma_1, \ldots, \sigma_{d-1}$ which have a total size of $n \cdot \sum_{i=1}^{d-1} \ell_i$ bits and the structures for upcoming trees. As observed by the authors of [9], these structures do not require a full BDS state, as some of the structures are not filled during initialization and the space to store the k top levels of nodes can be shared with the current tree. Thus, these structures require only $(h_i - k_i + 1)$ FsGen states (one for building the tree and the remaining as storage for the BDS state) and no more than $3h_i - k_i + 1$ tree nodes per TREE_i, $0 \leq i < d-1$. The secret key size is

$$n \cdot \left(\sum_{i=0}^{d-1} \left[\left(5h_i + \left\lfloor \frac{h_i}{2} \right\rfloor - 5k_i - 2 + 2^{k_i} \right) + 1 \right] + \sum_{i=0}^{d-2} (\ell_{i+1} + 4h_i - 2k_i + 2) \right)$$

bits. For the runtimes we only look at the worst case times and get the following. During key generation, the first tree on each layer has to be computed. This means, that each of the 2^h W-OTS key pairs has to be generated, including the execution of the PRGs. Further, the leaves of the trees have to be computed and all internal nodes of the tree, to obtain the root. If key generation generates the trees in order, starting from the first one, the W-OTS signatures on the roots of lower trees need no additional computation, as the signature can be extracted while the corresponding W-OTS key pair is generated. The key generation time is

$$\mathsf{t}_{\mathsf{Kg}} \leq t_{\text{H}} \left(\sum_{i=0}^{d-1} \left(2^{h_i} (\ell_i + 1) \right) \right) + t_{\text{F}} \left(\sum_{i=1}^{d-1} \left(2^{h_i} (2 + \ell_i (w_i + 1)) \right) \right),$$

where t_{H} and t_{F} denote the runtimes of one evaluation of H and F, respectively. During one call to Sign, a W-OTS signature on the message must be generated, including generation of the key $(t_{\text{F}}(2 + \ell_0(w_0 + 1)))$, the BDS algorithm receives $(h_0 - k_0)/2$ updates, one leaf of the next tree on layer 0 must be computed and the BDS algorithm updates $h_0 - k_0$ upcoming seeds $(t_{\text{F}}(h_0 - k_0))$. The worst case signing time is bounded by

$$\mathsf{t}_{\mathsf{Sign}} \leq \max_{i \in [0, d-1]} \left\{ \begin{array}{l} t_{\text{H}} \left(\frac{h_0 - k_0 + 2}{2} \cdot (h_i - k_i + \ell_i) + h_0 \right) \\ + t_{\text{F}} \left(\frac{h_0 - k_0 + 4}{2} \cdot (\ell_i (w_i + 1)) + h_0 - k_0 \right) \end{array} \right\}$$

Signature verification consists of computing d W-OTS public keys and the corresponding leafs plus hashing to the root. Summing up verification takes $\mathsf{t}_{\mathsf{Vf}} \leq \sum_{i=0}^{d-1} \left(t_{\text{H}} \left(\ell_i + h_i \right) + t_{\text{F}} \left(\ell_i w_i \right) \right)$ in the worst case.

3 Optimization

Given the theoretical formulas for runtimes and sizes from the last section, we now show how to use them to model the parameter selection problem as linear optimization problem. There are parameters which control different trade-offs. The BDS parameters $k_i \in \mathbb{N}$ control a trade-off between signature time and secret key size, the Winternitz parameters $w_i \in \mathbb{N}$ control a trade-off between

runtimes and signature size and the number of layers d determines a trade-off between key generation and signature time on the one hand and signature size on the other hand. Moreover there are the different tree heights h_i that do not define any obvious trade-off, but influence the security as well as the performance of the scheme. The goal of the optimization is to choose these parameters. The function families $\mathcal{F}_n$ and H can be instantiated, either using a cryptographic hash function or a block cipher. Hence the security parameter n is restricted to the output size of such functions. We choose 128 and 256 bit corresponding to AES and SHA-2 for our optimization, respectively.

Optimization Model. To find good parameter choices, we use linear optimization. In the following we discuss how we model the problem of optimal parameter choices as a linear optimization problem. As objective function of our problem we chose a weighted sum of all runtimes and sizes that should be minimized. Using the weights it is possible to control the importance of minimizing a certain parameter and thereby using the model for different scenarios. We further allow to apply absolute bounds on the runtimes and sizes. The formulas for runtimes and sizes are modeled as constraints as well as the parameter restrictions and the formula for bit security (see Appendix D). The input to the model are the runtimes of $\mathcal{F}$ and $\mathcal{H}$ for $n = 128$ and $n = 256$, the overall height h, the message length m and a value b as lower bound on the bit security.

As many of our initial constraints are not linear, we have to linearize all functions and restrictions. This is done using the generalized lambda method [13]. In addition, we split the problem into sub problems each having some decision variables fixed. The optimization problem contains the parameters of the scheme (d, n, the h_i, k_i, w_i for all layer) as decisions variables which are determined by solving the optimization. Further the message length m_i on each layer has to be modelled as a decision variable. Since solving the optimization problem takes much time and memory, we split the problem into sub problems by fixing the decision variables n and d. Therefore, we receive one sub problem for each combination of possible values of n and d. The resulting sub problems are solved independently and the best of their solutions is chosen as global solution of the original optimization problem.

The next step is to linearize the remaining sub problems by using the generalized lambda method. Therefore, we introduce a grid point for each possible combination of the remaining variables h, k, w and m on each layer i. For each grid point we have a binary variable $\lambda_{h,k,w,m,i}$ which takes value 1 if the combination of h, k, w, m is chosen on layer i. Otherwise, it takes value 0. Since we need one choice of h, k, w, m for each $i \in [0, d-1]$, d λ's must be chosen.

We use those lambdas to calculate the functions describing the problem. Thus, before optimizing we determine the values of the functions for each possible values of their variables. To make this feasible, we have to introduce bounds on the decision variables. We bound the tree height per layer layer by 24. As $k \leq h$ this bound also applies to k. For w we chose 255. These bounds are reasonable for the scenarios of the next section. For different scenarios they might have to

be changed. Then, to model the needed space of signatures $\sum_{i=0}^{d-1}(\ell_i + h_i) \cdot n$, we formulate the constraint

$$SpaceSig == \sum_{i=0}^{d-1}\sum_{h=1}^{24}\sum_{k=1}^{24}\sum_{w=2}^{512}\sum_{m\in\{128,256\}} \lambda_{h,k,w,m,i} \cdot \underbrace{f_{SpaceSig}(w,h,m)}_{pre-calculated}$$

in the optimizing model, where $f_{SpaceSig}(w,h,m) = (\ell + h)n$. Thus, $SpaceSig$ gives the exact value of the needed space of signatures for the choice of lambda's and can be used in constraints and objective function.

To linearize a condition containing the maximum of some terms, such as the public key size $n \cdot (\max_{0\leq i\leq d-1}\{h_i + \lceil \log \ell_i \rceil\} + 2)$, we write the following constraint:

$$\forall i \in \{1, ..., T\}$$
$$SpacePK \geq \sum_{h=1}^{24}\sum_{k=1}^{24}\sum_{w=2}^{512}\sum_{m\in\{128,256\}} \lambda_{h,k,w,m,i} \cdot (\underbrace{f_{\lceil \log \ell \rceil}(w,m)}_{pre-calculated} + h + 2)n$$

Hence, $SpacePK$ gives the public key size for the choice of lambda's. This constraint pushes the value of $SpacePK$ up high and due to the objective function the value will be pushed down as low as possible, so that in the end it takes the exact value.

4 Optimization Results

In this section we present optimal parameters for two exemplary use cases. More results will be included in the full version of this paper. To solve the optimization problem, we used the IBM Cplex solver [10], that implements the Simplex algorithm [8] with some improvements. The linearization described in the last section is exact. Thus, there is no loss of information or error. Therefore, it can be proven that the solution found by linear optimization based on the Simplex algorithm is the best possible solution. In the following we present the results and compare them with the results for parameter sets proposed in [3] and [9]. We choose a message length of 256 bits for all use cases assuming that the message is the output of a collision resistant hash function. Moreover we use 80 bits as lower bound for the provable bit security. This seems reasonable, as the used bit security represents a provable lower bound on the security of the scheme and is not related to any known attacks. We used the instantiations for $\mathcal{F}$ and $\mathcal{H}$ proposed in [3] with AES and SHA2 for $n = 128$ and $n = 256$, respectively and measured the resulting runtimes on a Laptop with Intel(R) Core(TM) i5-2520M CPU @2.5 GHz and 8 GB RAM. We got $t_{\mathrm{F}} = 0.000225ms$ and $t_{\mathrm{H}} = 0.00045ms$ for $n = 128$ as well as $t_{\mathrm{F}} = 0.00169ms$ and $t_{\mathrm{H}} = 0.000845ms$ for $n = 256$.

The first use case we look at meets the requirements of a document or code signature. We assume that the most important parameters are signature

size and verification time and try to minimize them, while keeping reasonable bounds on the remaining parameters. We used the bounds $\mathsf{t_{Sign}} < 1000ms, \mathsf{t_{Vf}} < 1000ms, \mathsf{t_{Kg}} < 60s, \mathsf{sk} < 25kB, \mathsf{pk} < 1.25kB, \sigma < 100kB$ and the weights $\mathsf{t_{Sign}} = 0.00000001, \mathsf{t_{Vf}} = 0.00090000$, $\mathsf{t_{Kg}} = 0.00000001, \mathsf{sk} = 0.00000001, \sigma = 0.99909996, \mathsf{pk} = 0.00000001$ for $h = 20$. We chose different weights for $\mathsf{t_{Vf}}$ and σ, because the optimization internally counts in bits and milliseconds. We set the weights such that $1ms$ costs the same as 1000 bit. The remaining weights are not set to zero but to $1.0e{-}8$, the smallest possible value that we allow. This is necessary to ensure that our implementation of inequalities in the model works. This also ensures that within the optimal solutions regarding $\mathsf{t_{Vf}}$ and σ, the best one regarding the remaining parameters is chosen. It turns out, that the optimization can be solved for $d \geq 2$. For $d = 1$ the bound on the key generation time cannot be achieved for the required height. If we relax this bound to be $\mathsf{t_{Kg}} < 600s$, i.e. 10 minutes, the problem can be solved for $n = 128$ using AES. For $d \geq 2$ this relaxation does not change the results. The optimal parameters for this setting are $n = 128, d = 2$, $h_0 = 17, k_0 = 5, w_0 = 5$ and $h_1 = 3, k_1 = 3, w_1 = 22$. For comparison we use a parameter set from [9] that matches the bound on the bit security ($n = 128, d = 2, h_0 = h_1 = 10, k_0 = k_1 = 4, w_0 = w_1 = 4$). The resulting runtimes and sizes are shown in Table 1. The results show that it is possible to reduce the signature size by almost one kilo byte, changing the other parameters within their bounds and increasing the second important parameter, the verification time, by 0.08 milliseconds.

As a second use case we take a total tree height of 80 and aim for a balanced performance over all parameters. This use cases corresponds to the use in a communication protocol. Again, we choose the weights such that $1ms$ costs the same as 1000 bit, but this time we use the same weights for all runtimes and for all sizes. For comparison we use parameters from [4] ($d = 4$, $h_0 = h_1 = h_2 = h_3 = 20$, $w_0 = 5$, $w_1 = w_2 = w_3 = 8$, $k_0 = k_1 = k_2 = k_3 = 4$). As they do not use a BDS parameter, we choose $k = 4$ on all layers. To make a fair comparison, we limited our optimization also to four layers. The optimal parameters returned by the optimization are $h_0 = h_1 = h_2 = h_3 = 20$, $w_0 = 14$, $w_1 = w_2 = w_3 = 24$ and $k_0 = k_1 = k_2 = 4$, $k_3 = 2$. The results are shown in Table 1. It turns out, that again, by trading some runtime, the signature size can be significantly reduced.

Table 1. Runtimes and sizes for optimized parameters and parameters proposed in previous works

	Runtimes (ms)			Sizes (bit)		
Use case	$\mathsf{t_{Kg}}$	$\mathsf{t_{Sign}}$	$\mathsf{t_{Vf}}$	σ	PK	SK
$UC1$ optimal	27251	1.65	0.36	21376	6144	25472
$UC1$ from [9]	326	1.00	0.28	28288	4608	25856
$UC2$ optimal	166min	25.55	9.13	83968	13824	209152
$UC2$ from [4]	98min	14.53	5.01	119040	13824	233472

5 Conclusion

With XMSS^{MT} we presented a new hash-based signature scheme, that allows to use a key pair for 2^{80} signatures, which is a virtually unlimited number of signatures in practice. The new scheme is highly flexible and can be parameterized to meet the requirements of any use case for digital signatures. In order to get the maximum benefit from this parameterization, we showed how to use linear optimization to obtain provably optimal parameter sets. We have shown the strength and functionality of our approach by presenting the parameter sets for two different use cases. Comparing our results to parameter sets from other works, it turns out that there is a lot of space for optimization.

References

1. Bellare, M., Miner, S.: A forward-secure digital signature scheme. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 431–448. Springer, Heidelberg (1999)
2. Buchmann, J., Dahmen, E., Ereth, S., Hülsing, A., Rückert, M.: On the security of the Winternitz one-time signature scheme. In: Nitaj, A., Pointcheval, D. (eds.) AFRICACRYPT 2011. LNCS, vol. 6737, pp. 363–378. Springer, Heidelberg (2011)
3. Buchmann, J., Dahmen, E., Hülsing, A.: XMSS - A practical forward secure signature scheme based on minimal security assumptions. In: Yang, B.-Y. (ed.) PQCrypto 2011. LNCS, vol. 7071, pp. 117–129. Springer, Heidelberg (2011)
4. Buchmann, J., Dahmen, E., Klintsevich, E., Okeya, K., Vuillaume, C.: Merkle signatures with virtually unlimited signature capacity. In: Katz, J., Yung, M. (eds.) ACNS 2007. LNCS, vol. 4521, pp. 31–45. Springer, Heidelberg (2007)
5. Buchmann, J., Dahmen, E., Schneider, M.: Merkle tree traversal revisited. In: Buchmann, J., Ding, J. (eds.) PQCrypto 2008. LNCS, vol. 5299, pp. 63–78. Springer, Heidelberg (2008)
6. Buchmann, J., Coronado García, L.C., Dahmen, E., Döring, M., Klintsevich, E.: CMSS – an improved merkle signature scheme. In: Barua, R., Lange, T. (eds.) INDOCRYPT 2006. LNCS, vol. 4329, pp. 349–363. Springer, Heidelberg (2006)
7. Dahmen, E., Okeya, K., Takagi, T., Vuillaume, C.: Digital signatures out of second-preimage resistant hash functions. In: Buchmann, J., Ding, J. (eds.) PQCrypto 2008. LNCS, vol. 5299, pp. 109–123. Springer, Heidelberg (2008)
8. Dantzig, G.B.: Linear Programming And Extensions. Princeton University Press (1963)
9. Hülsing, A., Busold, C., Buchmann, J.: Forward secure signatures on smart cards. In: Knudsen, L.R., Wu, H. (eds.) SAC 2012. LNCS, vol. 7707, pp. 66–80. Springer, Heidelberg (2013)
10. IBM. IBM ILOG CPLEX Optimizer, `http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/` (accessed Januray 2013)
11. Lenstra, A.K.: Key lengths. Contribution to The Handbook of Information Security (2004)
12. Malkin, T., Micciancio, D., Miner, S.: Efficient generic forward-secure signatures with an unbounded number of time periods. In: Knudsen, L.R. (ed.) EUROCRYPT 2002. LNCS, vol. 2332, pp. 400–417. Springer, Heidelberg (2002)
13. Moritz, S.: A Mixed Integer Approach for the Transient Case of Gas Network Optimization. PhD thesis, TU Darmstadt (February 2007)

A Winternitz OTS

XMSS^{MT} uses the Winternitz-OTS (W-OTS) from [2]. W-OTS uses the function family $\mathcal{F}_n$ and a value $X \in \{0,1\}^n$ that is chosen during XMSS key generation. For $K, X \in \{0,1\}^n$, $e \in \mathbb{N}$, and $\mathrm{F}_K \in \mathcal{F}_n$ we define $\mathrm{F}_K^e(X)$ as follows. We set $\mathrm{F}_K^0(X) = K$ and for $e > 0$ we define $K' = \mathrm{F}_K^{e-1}(X)$ and $\mathrm{F}_K^e(X) = \mathrm{F}_{K'}(X)$. Also, define

$$\ell_1 = \left\lceil \frac{m}{\log(w)} \right\rceil, \quad \ell_2 = \left\lfloor \frac{\log(\ell_1(w-1))}{\log(w)} \right\rfloor + 1, \quad \ell = \ell_1 + \ell_2.$$

The secret signature key of W-OTS consists of ℓ n-bit strings sk_i, $1 \le i \le \ell$. The generation of the sk_i will be explained later. The public verification key is computed as

$$\mathsf{pk} = (\mathsf{pk}_1, \ldots, \mathsf{pk}_\ell) = (\mathrm{F}_{\mathsf{sk}_1}^{w-1}(X), \ldots, \mathrm{F}_{\mathsf{sk}_\ell}^{w-1}(X)),$$

with F^{w-1} as defined above. W-OTS signs messages of binary length m. They are processed in base w representation. They are of the form $M = (M_1 \ldots M_{\ell_1})$, $M_i \in \{0, \ldots, w-1\}$. The checksum $C = \sum_{i=1}^{\ell_1}(w - 1 - M_i)$ in base w representation is appended to M. It is of length ℓ_2. The result is a sequence of ℓ base w numbers, denoted by $(T_1, \ldots, T_\ell)$. The signature of M is

$$\sigma = (\sigma_1, \ldots, \sigma_\ell) = (\mathrm{F}_{\mathsf{sk}_1}^{T_1}(X), \ldots, \mathrm{F}_{\mathsf{sk}_\ell}^{T_\ell}(X)).$$

It is verified by constructing $(T_1 \ldots, T_\ell)$ and checking

$$(\mathrm{F}_{\sigma_1}^{w-1-T_1}(X), \ldots, \mathrm{F}_{\sigma_\ell}^{w-1-T_\ell}(X)) \stackrel{?}{=} (\mathsf{pk}_1, \ldots, \mathsf{pk}_\ell).$$

The sizes of signature, public, and secret key are ℓn bits. The runtimes for key generation, signature generation and signature verification are all bounded by $\ell(w-1)$ evaluations of elements from $\mathcal{F}_n$. The Winternitz parameter w controls a time - space trade-off, as ℓ shrinks logarithmically in w. For more detailed information see [2].

B Correctness

In the following we argue that the BDS state on every layer receives its $(h-k)/2$ updates between two signatures. The authors of [9] showed that there is a gap between the updates a XMSS key pair receives over its lifetime and the updates it really uses. Namely, for $2 \le k \le h$ there are 2^{h-k+1} unused updates. For every tree on a layer other than 0, $h - k + 3$ updates are needed. The first $h - k$ updates are needed to update the TreeHash instances in the BDS state. Further one update is needed to compute a node in the next tree, to update

the FsGen states in the BDS state and to sign the root of the next tree on the lower layer, respectively. Between two signatures of a tree on layer $i+1$, a whole tree on layer i is used, so if we ensure that $(h_{i+1} - k_{i+1} + 6)/2 \leq 2^{h_i - k_i + 1}$, as we do, the $(h_i - k_i)/2$ updates TREE_i receives over its lifetime leave enough unused updates such that TREE_{i+1} receives the required updates. There would still occur a problem, if the number of updates per signature would be smaller than $d-1$. The reason is, that this would mean that the roots of the new trees on different layers could not always be signed using the updates of the last signature before the change. In this case the private storage would grow, as we would need some intermediate storage for the new signatures. This is the reason why the second condition ($(h_0 - k_0)/2 \geq d - 1$) is needed.

C Security

For the security, the argument is similar to that of [9]. In [3], an exact proof is given which shows that XMSS is forward secure, if $\mathcal{F}$ is a pseudorandom function family and $\mathcal{H}$ a second preimage resistant hash function family. The tree chaining technique corresponds to the product composition from [12]. In [12] the authors give an exact proof for the forward security of the product composition if the underlying signature schemes are forward secure. It is straight forward to combine both security proofs to obtain an exact proof for the forward security of XMSS^{MT}. In contrast to the case of XMSS^{+}, for XMSS^{MT} the product composition is applied not only once, but $d-1$ times. As forward security implies EU-CMA security, this also shows that XMSS^{MT} is EU-CMA secure.

D Security Level

We compute the security level in the sense of [11]. This allows a comparison of the security of XMSS^{MT} with the security of a symmetric primitive like a block cipher for given security parameters. Following [11], we say that XMSS^{MT} has security level b if a successful attack on the scheme can be expected to require approximately 2^{b-1} evaluations of functions from $\mathcal{F}_n$ and $\mathcal{H}_n$. Following the reasoning in [11], we only take into account generic attacks on $\mathcal{H}_n$ and $\mathcal{F}_n$. A lower bound for the security level of XMSS was computed in [3]. For XMSS^{MT}, we combined the exact security proofs from [3] and [12]. Following the computation in [3], we can lower bound the security level b by

$$b \geq \min_{0 \leq i \leq d-1} \left\{ n - 4 - w_i - 2log(\ell_i w_i) - \sum_{j=i}^{d-1} h_j \right\}$$

for the used parameter sets.

E XMSS^{MT} in Graphics

Here we included some graphics for a better understanding of XMSS^{MT}. Figure 1 shows the construction of the XMSS tree. An authentication path is shown in Figure 2. And finally Figure 3 shows a schematic representation of a XMSS^{MT} instance with four layers.

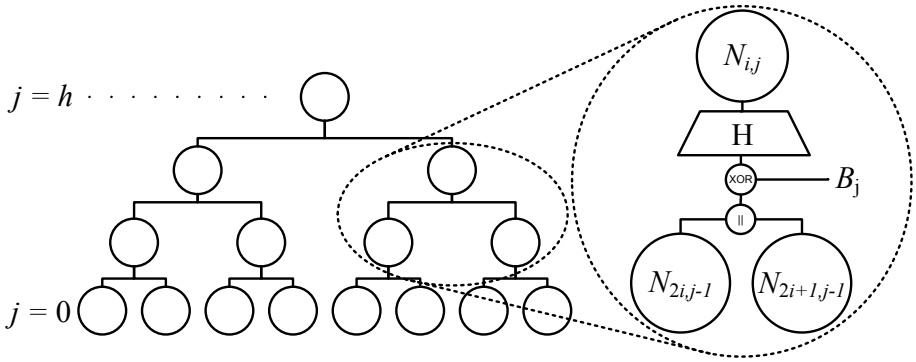

Fig. 1. The XMSS tree construction

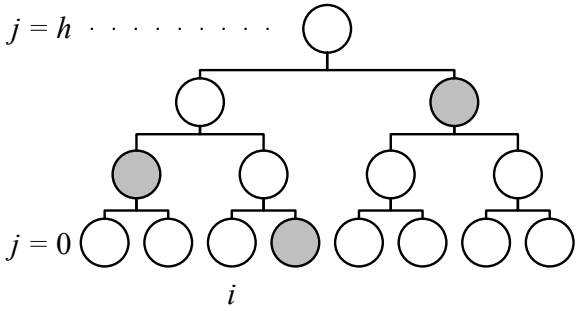

Fig. 2. The authentication path for a leaf i

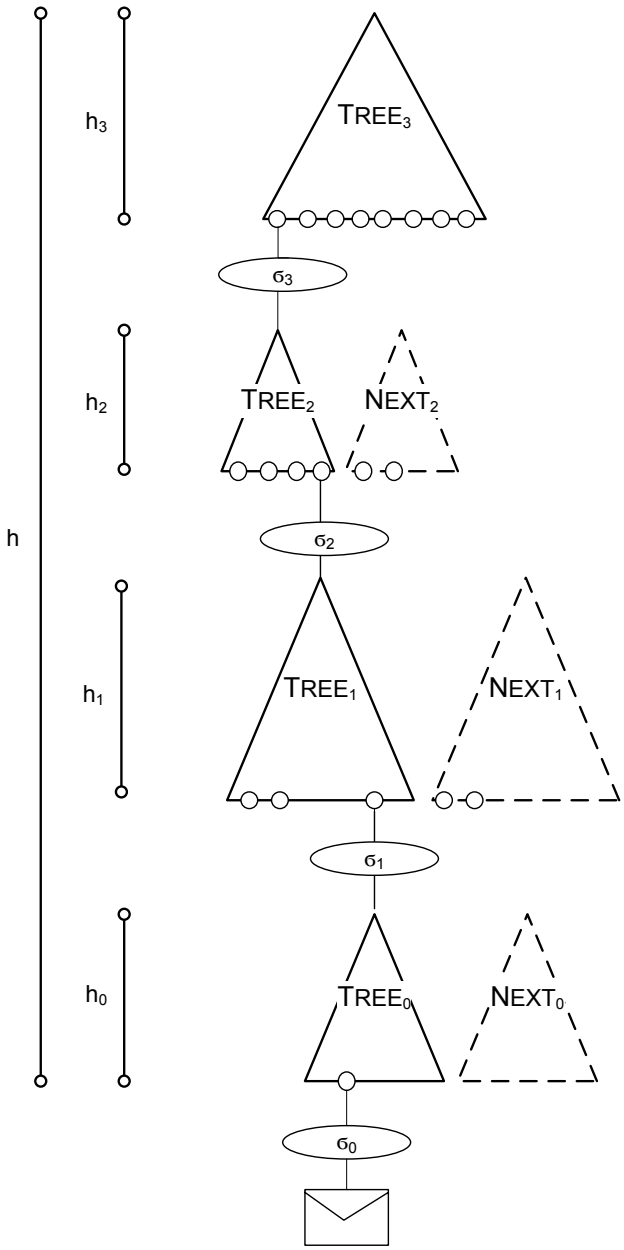

Fig. 3. A schematic representation of a XMSS^{MT} instance with four layers