Hash-based Signatures: An Outline for a New Standard

Andreas Hülsing*, Stefan-Lukas Gazdag†, Denis Butin‡ and Johannes Buchmann‡
*Department of Mathematics and Computer Science
Technische Universiteit Eindhoven
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
andreas.huelsing@googlemail.com
†genua mbh
Domagkstrasse 7, 85551 Kirchheim bei Muenchen, Germany
stefan-lukas_gazdag@genua.eu
†TU Darmstadt
Hochschulstrasse 10, 64289 Darmstadt, Germany
{dbutin,buchmann}@cdc.informatik.tu-darmstadt.de

Abstract. Hash-based signatures are quantum-safe and well understood. This paper presents an outline for a new standard extending the recent Internet-Draft by McGrew & Curcio. The goals of this new standard are twofold:

- To extend the existing draft to include XMSS and its multi-tree version;
- To prepare for possible extensions to cover stateless schemes.

Keywords: Hash-based signatures, Standardization, Merkle trees, XMSS.

I. Introduction

Hash-based signatures recently gained a lot of attention as a potential replacement for today's signature schemes when large-scale quantum computers are built. The reasons for this are manifold. The main reason probably consists in the reliable security estimates — also for security against attacks aided by quantum computers. This distinguishes hash-based signatures from other post-quantum signature schemes. Additionally, hash-based signatures need no computationally expensive mathematical operations like big integer arithmetic. The only requirement is a secure cryptographic hash function.

Hash-based signatures were initially proposed by Merkle [20] in the late 1970s and regained a lot of attention over the last decade [1], [3]–[12], [14]–[17], [21], [22]. These new schemes improve parameter sizes and runtimes, present security reductions, present implementations, and finally lower the security assumptions on the used hash function, i.e. provide collision resilience.

Recently, McGrew and Curcio published an Internet-Draft [19] for a hash-based signature scheme. Their draft essentially covers the scheme proposed by Merkle at the end of the 1970s. The advantage of this scheme over newer ones is that Merkle was granted a patent on this basic scheme that already expired. Hence, there cannot be any IPR claims for this scheme. However, to the best of our knowledge, there are no active or pending patents on XMSS and its variants. On the downside, the scheme in [19] has large signatures, relatively slow runtimes compared to other alternatives (especially for key generation), an unnecessary limit on the number of signatures that can be generated with one key pair and requires too strong assumptions about the security of the used hash-function.

This paper outlines a standard that includes the scheme from McGrew and Curcio's draft as a special case but otherwise covers the most efficient hash-based signature scheme XMSS [4] and its multi-tree variant [16]. Thereby, the draft also covers most basic building blocks used by the recently proposed stateless hash-based signature scheme SPHINCS [2]. That way, it prepares for a later extension.

Merkle's hash-based signatures. A hash-based signature scheme starts from a one-time signature scheme (OTS) — a signature scheme where each key pair must only be used to sign one message. If an OTS key pair is used to sign two different messages, an attacker can easily forge signatures. Merkle used Lamport's scheme [18] and variants thereof. To construct a many-time signature scheme, Merkle [20] proposed to use a binary hash tree later called Merkle tree. In a Merkle tree, the leaves are the hash values of OTS public keys. Each inner node is computed as the hash of the concatenation of its two child nodes. If a collision resistant hash function is used, this means that the root node can be used to authenticate all the leaf nodes, i.e. all the OTS public keys.

In a Merkle signature scheme (MSS) the root node of the Merkle tree becomes the public key, the set of all OTS secret keys becomes the secret key. For hash-based OTS the secret keys are random bit strings. Hence, instead of storing all OTS secret keys, one can store a short seed and (re-)generate the OTS secret keys using a cryptographically secure pseudorandom generator. To prevent reuse of OTS key pairs, they are used according to the order of the leaves, starting with the leftmost leaf. To do this, the scheme keeps as an internal state the index of the last used OTS key pair.

The signature of the ith message is $\Sigma = (i, \sigma_{\text{OTS}}, \mathsf{pk}_{\text{OTS},i}, \mathsf{Auth}_i)$, containing the index i, the OTS signature σ_{OTS} on the message using the ith OTS secret key, the ith OTS public key $\mathsf{pk}_{\text{OTS},i}$, and the so called *authentication path* Auth_i of the ith OTS public key. The authentication path Auth_i consists of all the sibling nodes of those nodes on the path from the ith leaf to the root.

To verify the signature Σ on message M, the verifier first validates the OTS signature on the message, using $\mathsf{pk}_{\text{OTS},i}$. If this verification succeeds, the OTS public key is verified. Towards this end, the ith leaf is computed as the hash of $\mathsf{pk}_{\text{OTS},i}$. Then, a root value is computed, using the nodes in Auth_i. If this root value matches the one given as public key, the signature is accepted, otherwise it is rejected.

Recall that a binary tree of height h has 2^h leaves. Hence, a MSS with a tree of height h can be used to sign $N = 2^h$ messages. For runtimes, the determining parameter is N. Key generation requires about 2^h hash function calls and is hence linear in N. Signing consists of one OTS signature and the authentication path computation. This can be done in time logarithmic in N using e.g. the BDS tree traversal algorithm from [6] to compute the authentication path. Verification time is also logarithmic in N.

For sizes, the important parameter is the output length of the hash function n. The public key is a n bit hash value. The secret key consists of a n bit seed (assuming pseudorandom key generation; and a public state for the BDS algorithm in the order of $n \log N$ if BDS is used). The signature size of the classical MSS using Lamport's scheme is $\approx 2n^2 + n \log_2 N$, i.e. quadratic in n, where the $2n^2$ is caused by the OTS and the $n \log_2 N$ by the authentication path[1]. Typical values for n and N are $n = 256$ and $N = 2^{20}$.

For a more detailed overview, also describing tree traversal algorithms, see [7].

The eXtended Merkle Signature Scheme (XMSS). The improvements of the last decade led to XMSS and its multi-tree version XMSS^{MT}. The main differences compared to the basic MSS are smaller signatures and collision resilience, which are actually closely related as we will show below. In addition, XMSS^{MT} allows to speed-up key and signature generation times at the cost of slightly larger signatures. These improvements go along with the following changes:

First, XMSS uses the Winternitz OTS (WOTS). The main advantage of WOTS is that to verify a signature a public key is computed and compared to the given one. When used in a MSS, this means that the MSS signature does not need to contain $\mathsf{pk}_{\text{OTS},i}$. Instead, verification first computes $\mathsf{pk}_{\text{OTS},i}$ from the OTS signature and then uses this OTS public key to compute a root value. If the computed root value matches the one in the public key, the OTS signature was valid and the used OTS key pair was authentic. This reduces signature size from $\approx 2n^2 + n \log_2 N$ to $\approx n^2 + n \log_2 N$, which is roughly a factor of 2 improvement. Furthermore, WOTS provides a trade-off between signature size and runtime, controlled by the Winternitz parameter $w \in \mathbb{N}$. Signatures shrink

[1]Here we consider a straightforward optimization where not the whole OTS public key is sent but those nodes not computable from the signature.

logarithmically in w (i.e. $\approx n^2/\log_2 w + n \log N$) while runtimes grow sub-linear in w, i.e. by a factor of $w/\log_2 w$.

Second, while MSS requires a collision resistant hash function, XMSS reduces this requirement to weaker security assumptions. This requires changing WOTS and the tree construction. Essentially, bitmasks are introduced and used to mask the inputs to the hash function before every hash computation. For a scheme with tree-height h, this comes at the price of slightly more than h additional $2n$ bit values in the public key to publish the bitmasks. The most important impact of this is that to achieve a security level of b bits, one can use a $n = b$ bits hash function. For MSS, a $n = 2b$ bit hash function is required to protect against birthday attacks. This reduces the signature size by another factor of two. Moreover, looking for example at MD5 and SHA1, there exist 'practical' collision attacks while the weaker properties XMSS uses (like second-preimage resistance) are still unbroken and there is not much progress in this direction so far.

Finally, XMSS^{MT} introduces a trade-off that allows to reduce key generation time from $\mathcal{O}(N)$ to $\mathcal{O}(d\sqrt[d]{N})$ at the cost of increasing signature size by a factor of d. It also slightly improves the worst-case signature generation time. XMSS^{MT} uses a certification tree of XMSS key pairs. This means, d layers of XMSS key pairs are used. The one on the top layer is used to sign the public keys (i.e. root nodes) of the key pairs on the layer below. These key pairs are used in turn to sign the public keys of the key pairs on the layer below, and so on. Finally, the key pairs on the lowest layer are used to sign messages. During key generation only the first key pair on each layer has to be generated, the generation of the remaining ones is distributed over signature generations.

This multi-tree construction is necessary for two cases. On the one hand, it allows to generate key pairs that can be used for a virtually unlimited number of signatures, i.e. $N = 2^{50}$. This was not possible with a single-tree scheme due to the key generation time linear in N. On the other hand, this improvement allows to implement key generation also on resource-constrained devices like smart cards. Previous implementations of single-tree schemes had to perform key generation on a more powerful device like a PC and transfer the key pair to the resource-constrained device afterwards.

For a more detailed overview that also covers provable security in detail see [13].

McGrew & Curcio's draft. The draft in [19] covers MSS with one change; WOTS is used as an OTS instead of Lamport's scheme. The used hash function has to be collision resistant. Hence, for a security level of b bits one needs $n = 2b$. Therefore, the performance figures look as follows. Key generation time is linear in N, signing and verification times logarithmic in N. The public key has $2b$ bits. The secret key consists of a $2b$ bit seed and the BDS state of $2b \log_2 N$ bits. The signature size is $\approx 4b^2 + N2b$. Please note that the draft does not dictate a specific algorithm for authentication path computation. We assume that the BDS algorithm is used. However, the comparison is independent of this choice.

Our draft. The draft proposed in this paper uses WOTS$^+$ from [14] as OTS. As tree construction we use the XMSS tree construction first proposed in [10]. These two ingredients already define single-tree XMSS. As mentioned above, the main technical difference is that these two constructions use bitmasks to mask the inputs to the hash function. Hence, the scheme described in [19] might be viewed as single-tree XMSS with empty bitmasks. Moreover, we specify a multi-tree setting.

The used hash function only has to guarantee weak security properties[2]. Hence, for a security level of b bits one needs $n = b$. Performance figures look as follows. Key generation time is $\mathcal{O}(d\sqrt[d]{N})$, signing and verification times logarithmic in N. The public key has $\approx (2\log_2 N + 1)b$ bits. The secret key consists of a b bit seed and the BDS state of $b \log_2 N$ bits. The signature size is $\approx db^2 + Nb$.

Allowing for a single-tree scheme with empty bitmasks includes the scheme from [19] as one case. Furthermore, this prepares for the extension to stateless schemes, as SPHINCS uses XMSS^{MT} as a building block. As outlined above the specified scheme has several security and performance advantages over the scheme specified in [19]. In addition, the multi-tree version allows for easy delegation of signing rights.

[2]To be specific, we require a second-preimage resistant undetectable one-way function. Actually, in both cases we also need a PRF and/or a PRG for the pseudorandom key generation and derandomization of random message hashes.

This is important as it removes the requirement to keep a synchronized state which becomes necessary for example if the same key should be shared between several devices or threads (consider for example TLS load balancing).

Organization. We start in Sec. II with the draft for WOTS$^+$. In Sec. III we describe XMSS, and in Sec. IV its multi-tree variant. In Sec. V, we discuss parameter choices. We conclude in Sec. VI with a synthesis and a request for feedback.

Notation. Throughout this paper we use the following notation. The data type $\texttt{bits}_x$ represents x bit strings, i.e. elements of $\{0,1\}^x$. The function `xor`(a, b) returns a $\texttt{bits}_x$ which holds the bitwise exclusive or of the two inputs a, b that also have type $\texttt{bits}_x$. Similarly, the function `concatenate`(a, b) on input of a $\texttt{bits}_x$ type a and a $\texttt{bits}_y$ type b returns a $\texttt{bits}_{(a+b)}$ type holding the string concatenation $a \| b$. For an array we write `array[i]` to address its $i+1$th element (i.e. the first element has address 0). We use the functions `ceil`(x) that for a real number x returns the smallest integer greater x and a mod b that returns the remainder of a/b as an integer between 0 and $b-1$.

II. WOTS$^+$

The main building block of a hash-based signature scheme is the OTS. We use WOTS$^+$ from [14]. In the following we specify three functions: `WOTS_genPK`, `WOTS_sign`, and `WOTS_pkFromSig`. These are the functions used within XMSS. In addition we specify the function `chain`, used by WOTS$^+$ internally. We start with parameter definitions. For a discussion on concrete parameters, see Section V.

Parameters. The scheme is instantiated with a hash function `hash_n_n` that, given a $\texttt{bits}_n$ type, outputs a $\texttt{bits}_n$ type holding the message digest. In practice, `hash_n_n` might be implemented using any cryptographically secure hash function with n bit outputs. As parameters, the scheme takes the message length m (m will be one of the typical message digest lengths) and the Winternitz parameter $w \in \{4, 8, 16\}$. The scheme can deal with any integer $w \geq 2$, but we decided to limit the choices as these values give the best trade-offs and allow for easy implementations. These two parameters determine the value ℓ computed as: $\ell_1 = \left\lceil \frac{m}{\log_2(w)} \right\rceil$, $\ell_2 = \left\lfloor \frac{\log_2(\ell_1(w-1))}{\log_2(w)} \right\rfloor + 1$, $\ell = \ell_1 + \ell_2$.

Function `chain`. The function `chain` (Alg. 1) computes an iteration of `hash_n_n` on an n bit input using a vector of random n bit *bitmasks*. In each iteration a bitmask is first XORed to the intermediate result before it is processed by `hash_n_n`. Please note that these bitmasks are part of the XMSS (or XMSS^{MT}) public key. Hence, we simply assume they are given here. In the following, `bm` is a length $w-2$ array of $\texttt{bits}_n$ types (that will contain the bitmasks).

Algorithm 1: $\texttt{bits}_n$ `chain` ($\texttt{bits}_n$ X, int i, int s, $\texttt{bits}_n$ [w-2] `bm`)

Input: Input value X, start index i, steps s, bitmasks `bm`.
Output: The value obtained by iterating `hash_n_n` s times on input X using the bitmasks starting at index i.

```
1 if $s == 0$ then
2     return $X$
3 end
4 if $(i + s) \geq w$ then
5     return $NULL$
6 end
7 $\texttt{bits}_n$ $T$ = chain( $X, i, s-1, \texttt{bm}$ );
8 $T$ = hash_n_n (xor( $T, \texttt{bm}[i+s-1]$ ));
9 return $T$
```

Function `WOTS_genPK`. A WOTS$^+$ secret key `sk` is a ℓ element vector of random n bit strings. At this point we consider this value as given. It can either be sampled uniformly or generated using a cryptographically secure pseudorandom generator. A WOTS$^+$ key pair defines a virtual structure that consists of ℓ *hash chains* of length w. The n bit strings in the secret key define the start node for one hash chain, each. The public key (Alg. 2) consists of the end nodes of these hash chains. To compute the hash chains the function `chain` (Alg. 1) is used. Please note that the same bitmasks are used for all chains.

Function `WOTS_sign`. A WOTS$^+$ signature σ is a ℓ element vector of n bit strings. A message is mapped to ℓ integers between 0 and $w-1$, taking a base w representation and appending a checksum. Each of

Algorithm 2: $\texttt{bits}_n[\ell]$ `WOTS_genPK` ($\texttt{bits}_n[\ell]$ `sk`, $\texttt{bits}_n[w-2]$ `bm`)

```
Input: Secret key vector sk, bitmasks bm.
Output: Public key vector pk.
bits_n [ℓ] pk;
for int $i = 0; i < \ell; i++$ do
    pk [i] = chain( sk[$i$], 0, $w-1$, bm );
end
return pk
```

these integers is used to select a node from a different hash chain. These selected nodes form the signature (Alg. 3). Please recall that all values that we allow for w are powers of two. Hence, $\log_2 w$ is an integer. Assuming $\log_2 w$ divides m, we can view a $\texttt{bits}_m$ type as a vector of unsigned integers between 0 and $w-1$, each described by $\log_2 w$ bits. Similar to [19] we use a function `coef` ($\texttt{bits}_m$ X, int i, int b) that returns the unsigned integer represented by the ith b bits of X. For instance, `coef` ((011010110100), i, 3) returns $011_2 = 3$ ($010_2 = 2$, $110_2 = 6$) for $i = 0$ ($i = 1$, $i = 2$, respectively).

Function `WOTS_pkFromSig`. Given a signature σ, a message M, and the bitmasks `bm`, `WOTS_pkFromSig` (Alg. 4) computes the corresponding public key value `pk`. This is done by recomputing the message mapping and continuing the hash chains from the signature values. To verify the signature, this public key value has to be verified. In XMSS, this is done by checking the authenticity of this public key value.

Choices made. We picked WOTS$^+$ out of the different available WOTS schemes. Our reasons are the mild security assumptions made on the hash function, the tight security reduction which gives greater exact security, and the performance benefits (see [13] for a comparison of the different WOTS variants). We fixed w to be a power of 2 on the one hand, which allows for a simpler, more efficient implementation of the message mapping. On the other hand, we restrict w to the set $\{4, 8, 16\}$. We did not include bigger values, as for these values, signature size decreases significantly while the decrease becomes decreasingly significant for greater values of w. We did not include $w = 2$ as $w = 4$ leads to roughly the same runtimes while shrinking the signature size by a factor of 2. The reason is that while the chains

Algorithm 3: $\texttt{bits}_n[\ell]$ `WOTS_sign` ($\texttt{bits}_n[\ell]$ `sk`, $\texttt{bits}_m$ M, $\texttt{bits}_n[w-2]$ `bm`)

```
Input: Secret key vector sk, message $M$,
       bitmasks bm.
Output: Signature $\sigma$ on $M$.
bits_n [ℓ] $\sigma$;
unsigned int csum = 0;
unsigned int[ℓ] msg;
unsigned int $\ell_3$ = ceil($\log_2 \ell_1 (w-1)$);
Append ($m \mod \log_2 w$) 0 bits to $M$;
for int $i = 0; i < \ell_1; i++$ do
    msg[$i$] = coef($M, i, \log_2 w$);
end
for int $i = 0; i < \ell_1; i++$ do
    csum += $w - 1 -$ msg[$i$];
end
Convert csum to a bits_{ℓ3} and append ($\ell_3 \mod \log_2 w$) 0 bits;
for int $i = 0; i < \ell_2; i++$ do
    msg[$i + \ell_1$] = coef($csum, i, \log_2 w$);
end
for int $i = 0; i < \ell; i++$ do
    $\sigma$[i] = chain( sk[$i$], 0, $msg[i]$, bm );
end
return $\sigma$
```

get twice as long and hence increase runtime, the number of chains is roughly halved. E.g., for the most important setting of $m = n = 256$ we get $\ell = 265$ for $w = 2$ and $\ell = 133$ for $w = 4$ (Bear in mind that signature size is ℓn bits and runtime is ℓw hash function calls).

III. XMSS

XMSS is currently the most efficient hash-based signature scheme. It subsumes different improvements from the last decade. Compared to the classical MSS, there are three main differences. First, XMSS uses a collision resilient WOTS variant. Here we decided to use WOTS$^+$ described in the last Section. Second, regarding tree construction, the computation of inner nodes was changed to achieve collision resilience. The change consists in XORing bitmasks with the inputs to the hash function. Third, leaf computation is altered to achieve collision resilience. Instead of applying a collision resistant hash function, another binary hash tree is used to compress the WOTS public

Algorithm 4: $\texttt{bits}_n[\ell]$ `WOTS_pkFromSig` ($\texttt{bits}_n[\ell]$ σ, $\texttt{bits}_m$ M, $\texttt{bits}_n[w-2]$ `bm`)

```
Input: Signature $\sigma$, message $M$.
Output: Public key vector pk.
$\texttt{bits}_n$ $[\ell]$ pk;
unsigned int csum = 0;
unsigned int$[\ell]$ msg;
unsigned int $\ell_3 = \texttt{ceil}(\log_2 \ell_1(w-1))$;
Append ($m \mod \log_2 w$) 0 bits to $M$;
for int $i = 0$; $i < \ell_1$; $i++$ do
    msg$[i]$ = coef$(M, i, \log_2 w)$;
end
for int $i = 0$; $i < \ell_1$; $i++$ do
    csum += $w - 1-$ msg$[i]$;
end
Convert csum to a $\texttt{bits}_{\ell_3}$ and append ($\ell_3 \mod \log_2 w$) 0 bits;
for int $i = 0$; $i < \ell_2$; $i++$ do
    msg$[i + \ell_1]$ = coef$(csum, i, \log_2 w)$;
end
for int $i = 0$; $i < \ell$; $i++$ do
    pk [i]=chain ($\sigma$[i], msg[i], w-1-msg[i], bm);
end
return pk
```

keys. This tree is called L-tree. Within the L-tree, bitmasks are used, like for the XMSS tree.

The research paper introducing XMSS [4] also describes a pseudorandom key generation algorithm. As this is not relevant for interoperability, we do not discuss it here. However, we suggest to use pseudorandom key generation and refer to [4]. The only requirement is that the used method provides the same security level as the remaining scheme. Moreover, for tree and authentication path calculation, a tree traversal algorithm is needed. We only present a very simple algorithm since this is irrelevant for interoperability, too. Note that much more efficient alternatives — like the BDS algorithm [6] — exist, and should be used in practice.

We start by listing the parameters for XMSS. Some required subroutines called `ltree` and `treeHash` are then defined. Afterwards, the key and signature generation as well as the verification algorithms are defined.

Parameters. Besides the hash function `hash_n_n` needed for WOTS$^+$, XMSS uses a second hash function `hash_2n_n` that compresses a $\texttt{bits}_{2n}$ type to a $\texttt{bits}_n$ type. In addition, a hash function `hash_m` which can handle arbitrary length bit strings ($\texttt{bits}_*$) as inputs and outputs a $\texttt{bits}_m$ type is used. Moreover, a pseudorandom function family `prf_m` is needed. It takes arbitrary length bit strings ($\texttt{bits}_*$) and a $\texttt{bits}_n$ type key as input, and outputs a $\texttt{bits}_m$ type. All these functions can be implemented using a cryptographic hash function; see Section V for details. Further parameters are the tree height $h \in \mathbb{N}$ that determines the number of signatures $N = 2^h$ per key pair and the Winternitz parameter w defined in the last section. The scheme uses $a = \max\{2(h + \lceil \log \ell \rceil), w - 2\}$ bitmasks, produced during key generation.

Function `ltree`. One of the main differences between XMSS and the MSS is the way the OTS public keys are compressed to leaves. XMSS uses a separate binary hash tree for this, called L-tree. The following function `ltree` implements this. It takes a WOTS$^+$ public key and compresses it to a single $\texttt{bits}_n$ type using a binary tree with bitmasks. Again, we assume that the bitmasks are externally given.

Algorithm 5: $\texttt{bits}_n$ `ltree` ($\texttt{bits}_n$ $[\ell]$ `pk`, $\texttt{bits}_n$ $[a]$ `bm`)

```
Input: WOTS$^+$ public key pk, and bitmasks bm.
Output: The $\texttt{bits}_n$ type root node of a binary
        hash tree built on top of pk.
unsigned int $\ell' = \ell$;
unsigned int $j = 0$;
while $\ell' > 1$ do
    for $i = 0; i < \lfloor \ell'/2 \rfloor; i++$ do
        pk$[i]$ = hash_2n_n (
        concatenate(xor(pk$[2i]$, bm$[j]$),
              xor(pk$[2i+1]$, bm$[j+1]$)));
    end
    if $l' == 1 \mod 2$ then
        $\texttt{pk}_{\lfloor l'/2 \rfloor + 1} = \texttt{pk}_{l'}$;
    end
    $l' = \lceil l'/2 \rceil$;
    $j = j + 2$;
end
return $\texttt{pk}_1$
```

Function `treeHash`. One of the main computations required for XMSS key and signature generation is computing nodes inside the tree. This can be done with the `treeHash` algorithm proposed by Merkle. The algorithm computes leaves one by one and, with each new leaf, tries to finish the computation of as many inner nodes as possible. We assume the algorithm takes the whole XMSS secret key SK. If pseudorandom key generation is used, the seed together with access to the pseudorandom generator is enough. The algorithm uses a $\texttt{bits}_n[h-1]$ type stack for which we assume that typical stack functions `push` and `pop` are available. To improve readability, we assume the existence of a function $\texttt{select}(\mathsf{SK}, x)$ that outputs the $\texttt{bits}_n[\ell]$ type containing the secret key of the WOTS$^+$ key pair that belongs to the xth leaf.

Algorithm 6: $\texttt{bits}_n$ `treeHash` ($\texttt{bits}_n[N\ell+2]$ SK, unsigned int s, unsigned int h, $\texttt{bits}_n[a]$ bm)

Input: Secret key SK, start leaf index s, target node height h, bitmasks bm.
Output: Root node of tree of height h with left most leaf being the hash of the sth OTS pk.

```
1  $\texttt{bits}_n[h-1]$Stack;
2  $\texttt{bits}_n$node;
3  $\texttt{bits}_n[\ell]$ pk;
4  for $i = 0; i < 2^h; i{+}{+}$ do
5      pk = WOTS_genPK (select(SK, $i$), bm);
6      node = ltree (pk, bm);
7      while Top node on Stack has same height $h'$ as node do
8          node = hash_2n_n (concatenate(
9              xor(Stack.pop(), bm[$2\ell + 2h'$]),
10             xor(node, bm[$2\ell + 2h' + 1$])));
11     end
12     Stack.push(node);
13 end
14 return Stack.pop()
```

Function `XMSS_genPK`**.** The XMSS secret key consists of $N\ell+1$ uniformly random $\texttt{bits}_n$ types, where the last one is used as a PRF key, followed by one $\texttt{bits}_n$ type that is reserved to store the index of the last used WOTS$^+$ key pair and initialized with 0[3]. Here, we assume it is given to make the algorithm independent of the secret key generation procedure. A complete XMSS key generation algorithm must first sample these secret key elements, or generate them using a cryptographically secure pseudorandom generation procedure. The public key generation algorithm `XMSS_genPK` takes the secret key SK as input. Then a bitmasks are chosen uniformly at random. Here, we assume the existence of a randomness source $\texttt{rand}(x)$ that returns x uniformly random $\texttt{bits}_n$ types. The root node is constructed using `TreeHash` (Alg. 6). The public key is a data structure consisting of $a+1$ $\texttt{bits}_n$: $\mathsf{PK} = [root, \texttt{bm}[0], \texttt{bm}[1], \ldots, \texttt{bm}[a-1]]$.

The same bitmasks are used for WOTS$^+$ on the one hand and the XMSS tree and L-tree on the other hand. Moreover, if a tree traversal algorithm like the BDS algorithm is used, the algorithms state also must be initialized during `XMSS_genPK`. For more details, see the respective algorithm descriptions [6].

Algorithm 7: $\texttt{bits}_{a+1}$ `XMSS_genPK` ($\texttt{bits}_n[N\ell+2]$ SK)

Input: Secret key SK.
Output: Public key SK.

```
1  $\texttt{bits}_n[a]$ bm = rand($a$);
2  $\texttt{bits}_n$ root;
3  root = treeHash (SK, 0, $h$, bm);
4  $\texttt{bits}_n[a]$ PK = concatenate( root , bm );
5  return PK
```

Function `XMSS_sign`**.** The signature algorithm signs a message M given as an arbitrary length bit string[4]. The function `XMSS_sign` takes the message $\texttt{bits}_*$ M, the secret key SK and the bitmasks bm. During signature generation, the secret key is updated, i.e. the index is incremented by one. The algorithm outputs the evolved key and a signature on M. An XMSS signature is a bit string of length $h + n + \ell n + (h-1)n = h + (\ell + h)n$. The first h bits keep the index of the used WOTS$^+$ key pair. The next n bits store the randomness used for

[3]We stick to a $\texttt{bits}_n$ type to ease notation. To save space, one can restrict this last element to a $\texttt{bits}_h$ type.

[4]Actually, the maximum input length for `hash_m` will determine the limit here.

randomized hashing. The following ℓn bits store the WOTS$^+$ signature and the last $(h-1)n$ bits store the authentication path. We assume the existence of a subroutine $\texttt{bits}_n[(h-1)]$ `buildAuth`$(\texttt{bits}_n[N\ell+2]\ \mathsf{SK}, \texttt{bits}_n[a]\ \texttt{bm}, \texttt{bits}_h\ i)$ that, on input of the secret key, the bitmasks and an index, outputs the authentication path for this index. We emphasise again that this should be replaced by one's favorite tree traversal algorithm.

```
Algorithm 8: $\texttt{bits}_{h+(N\ell+\ell+h+2)n}$ XMSS_sign ($\texttt{bits}_*\ M$, $\texttt{bits}_{N\ell+2}$ SK, $\texttt{bits}_n[a]$ bm)
    Input: Message $M$, secret key $SK$, bitmasks bm.
    Output: Updated secret key SK followed by XMSS signature $\Sigma$.
1 $\texttt{bits}_h\ i$ = (unsigned int) SK[$N\ell$];
2 SK[$N\ell$] = SK[$N\ell$] + 1;
3 $\texttt{bits}_n[h-1]$ Auth = buildAuth(SK, bm, $i$);
4 $\texttt{bits}_n\ r$ = prf_m(SK[$N\ell+1$], $M$);
5 $\texttt{bits}_m\ M'$ = hash_m(concatenate($r, M$));
6 $\texttt{bits}_n[\ell]\ \sigma$ = WOTS_sign(select(SK, $i$), $M'$, bm);
7 $\texttt{bits}_{h+(\ell+h)n}\ \Sigma = (i, r, \sigma, \mathsf{Auth})$;
8 return concatenate( SK, $\Sigma$ )
```

Function `XMSS_verify`. XMSS signature verification entails computing the WOTS$^+$ public key from the WOTS$^+$ signature, which is achieved using `WOTS_pkFromSig`. The computed WOTS$^+$ public key is then used together with the authentication path to compute a root node. This root node is compared to the first value of the XMSS public key. If they are identical, the outcome of the signature verification is successful.

Choices made. Like for WOTS$^+$, the main reason to choose XMSS is collision resilience and the implied smaller signature size. While [4] does not deal with the message hash, we propose randomized hashing to achieve collision resilience for this part as well. Based on the presented scheme, it is possible to build a forward secure construction. To that end, the use of a forward secure PRG is necessary. This aspect is not describe here, but is to be included in future stages. For further information about the requirements and the construction, see [4].

```
Algorithm 9: Boolean XMSS_verify ($\texttt{bits}_{h+(\ell+h)n}\ \Sigma$, $\texttt{bits}_m\ M$, $\texttt{bits}_n[a+1]$ PK)
    Input: XMSS signature $\Sigma = (i, r, \sigma, \mathsf{Auth})$, message $M$, and XMSS public key PK.
    Output: true if signature is valid, false otherwise.
1 $\texttt{bits}_m\ M'$ = hash_m(concatenate($r, M$));
2 $\texttt{bits}_n\ [\ell]$ pk = WOTS_pkFromSig ($\sigma$, $M'$, bm);
3 $\texttt{bits}_n\ [2]$ node;
4 node [0] = ltree (pk, bm);
5 for $k = 1; k < h; k{+}{+}$ do
6     if $\lfloor i/2^k \rfloor \mod 2 == 0$ then
7         node[1] = hash_2n_n (concatenate(
          xor(node[0], bm[$2\ell+2k$]),
          xor(Auth[$k-1$], bm[$2\ell+2k+1$])));
8     else
9         node[1] = hash_2n_n(concatenate(
          xor(Auth[$k-1$], bm[$2\ell+2k$]),
          xor(node[0], bm[$2\ell+2k+1$])));
10    end
11    node[0] = node[1];
12 end
13 if node[0] == PK[0] then
14     return true
15 else
16     return false
17 end
```

IV. XMSS^{MT}

XMSS^{MT} [16] is a generalization of XMSS allowing faster key generation and, as a consequence, more signatures per key pair. A single XMSS^{MT} tree consists of several layers of XMSS trees. The trees on the top and intermediate layers are used to sign the roots of the trees on the layer below. The trees on the lowest layer are used to sign the actual messages. Therefore, the public key still only needs the root of the top tree (and the bitmasks), while an XMSS^{MT} signature has to provide all intermediate signatures and authentication paths on the way to the top tree's root. In the following we define the three functions 10, 11 and 12 of XMSS^{MT}. They use the XMSS algorithms from the last section with small modifications. For instance, randomized message hashing is only needed on the lowest layer. On the other layers, the messages to be signed are the root nodes of other trees that only have n bits. Hence, we add descriptions of those modified versions, too.

Please note that the full performance potential of XMSS^{MT} can only be achieved using distributed key and signature generation. In this case, the key generation algorithm generates the first tree on each layer. The generation of remaining trees is distributed among signature generations. We leave this out here as it is not linked to interoperability. For details see [16]. Throughout this section we assume the idea of pseudorandom key generation is understood and omit the discussion. We consider a uniformly random secret key as given. We now start with the parameter definition for XMSS^{MT}.

Parameters. As XMSS is a building block of XMSS^{MT}, it uses all the functions required by XMSS. Moreover, a total tree height h (a key pair can be used to sign $N = 2^h$ messages) and a Winternitz parameter $w \in \{4, 8, 16\}$ is needed. The only XMSS^{MT}-specific parameter is the number of layers $d \in \mathbb{N}^*$. In contrast to [16], we use the same tree height h/d and the same Winternitz parameter w for all tree layers. When $n \neq m$, the parameter ℓ of WOTS$^+$ key pairs on the lowest layer differs from the one of WOTS$^+$ key pairs on higher layers, as only n bit values have to be signed. We stick to ℓ for the 'ℓ'-value on the lowest layer, computed using w and m. We use ℓ_n for the 'ℓ'-value on all other layers, computed using w and n. For further information about parameters, see Sec. V.

Function `XMSSMT_genPK`**.** The XMSS^{MT} secret key consists of $s = 1 + N\ell + \sum_{i=1}^{d-1} 2^{i(h/d)}\ell_n$ uniformly random $\texttt{bits}_n$ types. The last one is used as PRF key, and one $\texttt{bits}_n$ type, set to zero, is reserved for the index of the last used WOTS$^+$ key pair on the bottom layer. Like for XMSS, we assume the existence of a function $\texttt{select}(\mathsf{SK}, x, y)$ that outputs the $\texttt{bits}_n[N\ell_n]$ ($\texttt{bits}_n[N\ell]$, resp. for layer 0) type containing the secret key of the XMSS key pair that belongs to the xth tree on the yth layer without the PRF key and the index. XMSS^{MT} key generation (Alg. 10) begins with the random generation of $a = \max\{2(h + \lceil \log \ell \rceil), 2(h + \lceil \log \ell_n \rceil), w - 2\}$ bitmasks `bm`. The same bitmasks are used on each layer. Then, the root node of the top layer tree is generated using a slightly modified variant of `XMSS_genPK` (Alg. 7) denoted by `XMSS_genPK'`. The algorithm `XMSS_genPK'` equals `XMSS_genPK` without the first line. as it takes the bitmasks as input. The XMSS^{MT} public key PK^{MT} is a $\texttt{bits}_n[a+1]$ type that contains the root of the top layer tree followed by `bm`, exactly like an XMSS public key.

Algorithm 10: $\texttt{bits}_n[a+1]$ `XMSSMT_genPK(` $\texttt{bits}_n[s+1]$ SK^{MT}`)`

```
Input: Secret key $\mathsf{SK}^{MT}$.
Output: Public key $\mathsf{PK}^{MT}$.
1 $\texttt{bits}_n[a]$ bm = rand($a$);
2 $\texttt{bits}_n$ root =
    XMSS_genPK'(select($\mathsf{SK}^{MT}, 0, d-1$), bm);
3 $\texttt{bits}_n[a+1]$ $\mathsf{PK}^{MT}$ = concatenate( root, bm );
4 return $\mathsf{PK}^{MT}$
```

Function `XMSSMT_sign`**.** The XMSS^{MT} signature generation algorithm (Alg. 11) consists of the signature generation proper and of the secret key update. The latter only consists of increasing the index stored in the last $\texttt{bits}_n$ of the secret key. The signing itself involves signing M using a tree on the bottom layer, generating all its ancestor trees, and signing the roots of each of these trees using the respective parent tree. When using distributed signature generation as described in [16], this is reduced to signing the message on the bottom layer and a few operations to update a state. `XMSSMT_sign` uses `XMSS_sign` and a variant we denote by `XMSS_sign'`. The latter is equal to `XMSS_sign` but it omits hashing the message (lines 4 and 5 are missing, $M' = M$) and the output signature does not contain index i and randomness r. An XMSS^{MT} signature is a $t = h + (\ell + (d-1)\ell_n + h)n$ bit string. The first h bits take the index i of the used WOTS$^+$ key pair on the lowest layer, followed by n bits storing the randomness for the message hash. Next, there are the d XMSS signatures (without indices and randomness) starting with the message signature on layer 0 followed by the signatures on the root nodes ordered by increasing layer number.

Function `XMSSMT_verify`**.** XMSS^{MT} signature verification (Alg. 12) roughly consists of d XMSS signature verifications. As a subroutine, it uses `XMSS_verify'` which equals `XMSS_verify` but returns the computed root node `node`$[0]$ from line 13 instead of a boolean. Moreover, it uses `XMSS_verify''` which equals `XMSS_verify'` but also omits the message hash

Algorithm 11: $\texttt{bits}_t$ `XMSSMT_sign` ($\texttt{bits}_* M$, $\texttt{bits}_n[s+1]$ SK^{MT}, $\texttt{bits}_n[a+1]$ `bm`)

```
Input: Message $M$, XMSS$^{MT}$ secret key $\mathsf{SK}^{MT}$, bitmasks bm.
Output: A $t+(s+1)n$ bit string holding the updated secret key, followed by the XMSS$^{MT}$ signature $\Sigma^{MT}$.
1  $\texttt{bits}_h$ $i$ = (unsigned int) $\mathsf{SK}^{MT}[s]$;
2  $\mathsf{SK}^{MT}[s] = \mathsf{SK}^{MT}[s] + 1$;
3  unsigned int tree = $h - h/d$ most significant bits of $i$;
4  unsigned int tree0 = tree;
5  unsigned int leaf = $h/d$ least significant bits of $i$;
6  $\texttt{bits}_{N\ell+2}$ SK = concatenate( select($\mathsf{SK}^{MT}, tree, 0$), $\mathsf{SK}^{MT}[s-1], leaf$);
7  $\texttt{bits}_n$ root = XMSS_genPK'(SK,bm);
8  $\texttt{bits}_{h+(\ell+h)n}$ $\sigma$= XMSS_sign (M, SK, bm));
9  $\texttt{bits}_{(\ell+h-1)n}[d-1]\sigma'$; for $j = 1; j < d; j$ ++ do
10     leaf = $h/d$ least significant bits of $tree$;
11     tree = $h - jh/d$ most significant bits of $tree$;
12     SK = concatenate( select($\mathsf{SK}^{MT}, tree, j$), $\mathsf{SK}^{MT}[s-1], leaf$);
13     $\texttt{bits}_n$ root = XMSS_genPK'(SK,bm);
14     $\sigma'[j]$ = XMSS_sign'(root, SK, bm));
15 end
16 $\texttt{bits}_t$ $\Sigma^{MT}$ = concatenate( $tree0, \sigma, \sigma'$ );
17 return concatenate( $\mathsf{SK}^{MT}, \Sigma^{MT}$ )
```

Algorithm 12: Bool `XMSSMT_verify` ($\texttt{bits}_t$ Σ^{MT}, $\texttt{bits}_* M$, $\texttt{bits}_n[a+1]\mathsf{PK}^{MT}$)

```
Input: Signature $\Sigma^{MT}$, message $M$, and public key $\mathsf{PK}^{MT}$.
Output: true if signature is valid, false otherwise.
1  unsigned int leaf;
2  unsigned int tree = $h - h/d$ most significant bits of $\Sigma^{MT}$;
3  $\texttt{bits}_{t-h+h/d}$ $\Sigma' = t - h$ least significant bits of $\Sigma^{MT}$;
4  $\texttt{bits}_{h+(\ell+h)n}$ $\sigma = h + (\ell+h)n$ most significant bits of $\Sigma'$;
5  $\Sigma' = \Sigma' \ll h + (\ell+h)n$;
6  $\texttt{bits}_n$ node = XMSS_verify'($\sigma$, M, $\mathsf{PK}^{MT}$);
7  for $j = 1; j < d; j$ ++ do
8      leaf = $h/d$ least significant bits of $tree$;
9      tree = $h - jh/d$ most significant bits of $tree$;
10     $\sigma$= concatenate(leaf, $(\ell_n + h - 1)n$ most significant bits of $\Sigma'$);
11     $\Sigma' = \Sigma' \ll (\ell_n + h - 1)n$;
12     node = XMSS_verify''($\sigma$, node, $\mathsf{PK}^{MT}$);
13 end
14 if node == $\mathsf{PK}^{MT}[0]$ then
15     return true
16 else
17     return false
18 end
```

(line 1, i.e. $M' = M$) and correspondingly takes a signature without randomness as input.

Choices made. We decided to restrict the parameters such that the same height and Winternitz parameter are used on each layer. While this restricts the trade-offs provided by the scheme, it also significantly simplifies implementation and makes it less error-prone. This might actually increase the practical security of the scheme more than the switch to collision resilience.

V. Parameters

Only some parameters (e.g. w) are limited to specific values in this standard outline. We intend to provide optimized parameter sets in the actual standard. Techniques to determine optimal parameters for the schemes presented here already exist and have been used before [16]. However, we will need further input from stakeholders regarding the requirements.

In general, we intend to propose parameter sets for three different classical security levels: 128, 256, and 512 bit. To achieve this, we set the function output sizes to $m = n = 128$ (256, 512, respectively). Considering quantum attacks, these correspond to 64, 128, and 256 bit post-quantum security. We suggest to support $n = 128$ as it might encourage adoption in the pre-quantum era. Similarly, we suggest $n = 512$ only when considering a post-quantum scenario.

For the $n = 128$ setting, AES-based hash functions seem to be the best choice as they can benefit from hardware acceleration on many platforms. For constructions of the required functions, see [4]. For the other two settings, the use of SHA-3 seems a reasonable choice. However, we can achieve better results using specially tailored short input length hash functions like those presented in [2].

For single-tree XMSS, the tree height h should be at most 20, otherwise key generation time gets too slow. For XMSS^{MT}, we suggest to keep the height per layer significantly smaller (e.g. between 10 and 16) to benefit from the trade-off. A total tree height of 50 appears to be sufficient for any current applications.

For single-tree schemes, we propose to define one parameter set where the bitmasks are implicitly set to 0. Implementations should support this, either replacing the bitmasks by a zero vector or removing the XOR steps in the code. We suggest that this be only supported for the $n = 256$ and $n = 512$ settings, as formal security reductions require the used hash functions to be collision resistant in this case. Hence, the estimated classical security levels are 128 and 256 bits.

VI. Conclusion

We outline plans for a new standard for hash-based signatures, including WOTS$^+$ as a Winternitz-type OTS and the hash-based signature schemes XMSS and XMSS^{MT}. This extends the current draft by McGrew and Curcio [19] with schemes that provide faster key and signature generation, smaller signatures and milder security requirements. The possibility of empty bitmasks allows us to include the scheme from [19] as a special case. We also prepare for the future, since the standardized schemes constitute building blocks of recent stateless hash-based signature schemes [2]. We describe which aspects of the schemes ought to be specified in a standard, and which ones are left as implementation choices. The main algorithms for WOTS$^+$, XMSS and XMSS^{MT} are defined in pseudo-code. An open point requiring feedback from stakeholders is parameter selection. We warmly welcome comments and suggestions on this proposal.

Acknowledgment

This work was supported by the Netherlands Organisation for Scientific Research (NWO) under grant 639.073.005, the Bavarian Ministry of Economic Affairs and Media, Energy and Technology (StMWi) and the German Research Foundation (DFG).

References

[1] Piotr Berman, Marek Karpinski, and Yakov Nekrich. Optimal trade-off for Merkle tree traversal. In Joaquim Filipe, Helder Coelhas, and Monica Saramago, editors, *E-business and Telecommunication Networks*, volume 3 of *Communications in Computer and Information Science*, pages 150–162. Springer Berlin Heidelberg, 2007. 1

[2] Daniel J. Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Peter Schwabe, and Zooko Wilcox O'Hearn. Sphincs: practical stateless hash-based signatures. Cryptology ePrint Archive, Report 2014/795, 2014. http://eprint.iacr.org/. 1, 11

[3] Johannes Buchmann, Erik Dahmen, Sarah Ereth, Andreas Hülsing, and Markus Rückert. On the security of the Winternitz one-time signature scheme. In A. Nitaj and D. Pointcheval, editors, *Africacrypt 2011*, volume 6737 of *Lecture Notes in Computer Science*, pages 363–378. Springer Berlin / Heidelberg, 2011. 1

[4] Johannes Buchmann, Erik Dahmen, and Andreas Hülsing. XMSS — A Practical Forward Secure Signature Scheme Based on Minimal Security Assumptions. In Bo-Yin Yang, editor, *Post-Quantum Cryptography 2011*, volume 7071 of *Lecture Notes in Computer Science*, pages 117–129. Springer Berlin / Heidelberg, 2011. 1, 6, 8, 10

[5] Johannes Buchmann, Erik Dahmen, Elena Klintsevich, Katsuyuki Okeya, and Camille Vuillaume. Merkle signatures with virtually unlimited signature capacity. In Jonathan Katz and Moti Yung, editors, *ACNS 2007*, volume 4521 of *Lecture Notes in Computer Science*, pages 31–45. Springer Berlin / Heidelberg, 2007. 1

[6] Johannes Buchmann, Erik Dahmen, and Michael Schneider. Merkle tree traversal revisited. In Johannes Buchmann and Jintai Ding, editors, *Post-Quantum Cryptography*, volume 5299 of *Lecture Notes in Computer Science*, pages 63–78. Springer Berlin / Heidelberg, 2008. 1, 2, 6, 7

[7] Johannes Buchmann, Erik Dahmen, and Michael Szydlo. Hash-based digital signature schemes. In Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen, editors, *Post-Quantum Cryptography*, pages 35–93. Springer Berlin Heidelberg, 2009. 1, 2

[8] Johannes Buchmann, L. C. Coronado García, Erik Dahmen, Martin Döring, and Elena Klintsevich. CMSS — an improved Merkle signature scheme. In *Indocrypt 2006*, volume 4329 of *Lecture Notes in Computer Science*, pages 349–363. Springer, 2006. 1

[9] Erik Dahmen and Christoph Krauß. Short hash-based signatures for wireless sensor networks. In Juan Garay, Atsuko Miyaji, and Akira Otsuka, editors, *Cryptology and Network Security*, volume 5888 of *Lecture Notes in Computer Science*, pages 463–476. Springer Berlin / Heidelberg, 2009. 1

[10] Erik Dahmen, Katsuyuki Okeya, Tsuyoshi Takagi, and Camille Vuillaume. Digital signatures out of second-preimage resistant hash functions. In Johannes Buchmann and Jintai Ding, editors, *Post-Quantum Cryptography 2008*, volume 5299 of *Lecture Notes in Computer Science*, pages 109–123. Springer Berlin / Heidelberg, 2008. 1, 3

[11] Chris Dods, Nigel Smart, and Martijn Stam. Hash based digital signature schemes. In Nigel Smart, editor, *Cryptography and Coding*, volume 3796 of *Lecture Notes in Computer Science*, pages 96–115. Springer Berlin / Heidelberg, 2005. 1

[12] L. C. Coronado García. On the security and the efficiency of the Merkle signature scheme. Technical Report Report 2005/192, Cryptology ePrint Archive - Report 2005/192, 2005. Available at http://eprint.iacr.org/2005/192/. 1

[13] Andreas Hülsing. *Practical Forward Secure Signatures using Minimal Security Assumptions*. PhD thesis, TU Darmstadt, Darmstadt, August 2013. 3, 5

[14] Andreas Hülsing. W-OTS+ — shorter signatures for hash-based signature schemes. In Amr Youssef, Abderrahmane Nitaj, and AboulElla Hassanien, editors, *Progress in Cryptology – AFRICACRYPT 2013*, volume 7918 of *Lecture Notes in Computer Science*, pages 173–188. Springer Berlin Heidelberg, 2013. http://huelsing.files.wordpress.com/2013/05/wotsspr.pdf. 1, 3, 4

[15] Andreas Hülsing, Christoph Busold, and Johannes Buchmann. Forward secure signatures on smart cards. In Lars R. Knudsen and Huapeng Wu, editors, *Selected Areas in Cryptography*, volume 7707 of *Lecture Notes in Computer Science*, pages 66–80. Springer Berlin Heidelberg, 2013. 1

[16] Andreas Hülsing, Lea Rausch, and Johannes Buchmann. Optimal parameters for XMSS^{MT}. In Alfredo Cuzzocrea, Christian Kittl, Dimitris E. Simos, Edgar Weippl, and Lida Xu, editors, *Security Engineering and Intelligence Informatics*, volume 8128 of *Lecture Notes in Computer Science*, pages 194–208. Springer Berlin Heidelberg, 2013. 1, 8, 9, 10

[17] Markus Jakobsson, Tom Leighton, Silvio Micali, and Michael Szydlo. Fractal Merkle tree representation and traversal. In Marc Joye, editor, *Topics in Cryptology — CT-RSA 2003*, volume 2612 of *Lecture Notes in Computer Science*, pages 314–326. Springer Berlin Heidelberg, 2003. 1

[18] Leslie Lamport. Constructing digital signatures from a one way function. Technical Report SRI-CSL-98, SRI International Computer Science Laboratory, 1979. 2

[19] David McGrew and Michael Curcio. Hash-Based Signatures, 2014. 1, 3, 5, 11

[20] Ralph Merkle. A certified digital signature. In Gilles Brassard, editor, *Crypto'89*, volume 435 of *Lecture Notes in Computer Science*, pages 218–238. Springer Berlin / Heidelberg, 1990. 1, 2

[21] Sebastian Rohde, Thomas Eisenbarth, Erik Dahmen, Johannes Buchmann, and Christof Paar. Fast hash-based signatures on constrained devices. In Gilles Grimaud and François-Xavier Standaert, editors, *Smart Card Research and Advanced Applications*, volume 5189 of *Lecture Notes in Computer Science*, pages 104–117. Springer Berlin / Heidelberg, 2008. 1

[22] Michael Szydlo. Merkle tree traversal in log space and time. In Christian Cachin and Jan Camenisch, editors, *Advances in Cryptology — EUROCRYPT 2004*, volume 3027 of *Lecture Notes in Computer Science*, pages 541–554. Springer Berlin / Heidelberg, 2004. 1