Treehash Equivalence: Recursive and Iterative Definitions

Wrenna Robson[*]

February 2026

Status: draft—all proof bodies filled in; under review.

1 Introduction

The XMSS treehash computation (RFC 8391, Algorithm 9) is specified iteratively, using a stack to avoid storing the whole tree in memory. Security proofs, however, are most naturally stated against a *recursive* definition of the tree, where the root is defined by structural induction on height.

This document establishes a formal equivalence between the two formulations, for both single-tree XMSS and multi-layer XMSS-MT. The equivalence is not merely about output values: because the hash function takes an *address* as input (for domain separation), the proof must also show that every internal hash call uses the correct address. This address-correctness is the main technical difficulty.

The document is intended to be self-contained and to serve as a basis for a future mechanised proof (e.g. in EasyCrypt), so definitions are stated precisely and with an eye toward program logic.

2 Notation and Preliminaries

Fix a parameter set with:

- n: hash output length in bytes.
- h: tree height (number of levels above the leaves; the tree has 2^h leaves).
- w: Winternitz parameter (relevant for WOTS+, mentioned for completeness).

We write $[a, b)$ for the integer interval $\{a, a + 1, \ldots, b - 1\}$.

2.1 Addresses

An *address* adrs is a structured 32-byte value with fields including:

- layer: the hypertree layer (0 for XMSS; $0, \ldots, d - 1$ for XMSS-MT).
- tree: the tree index within a layer (XMSS-MT only).
- type: distinguishes OTS, L-tree, and hash-tree computations.
- $\mathsf{treeHeight}$, $\mathsf{treeIndex}$: position within the hash tree.

We say an address is *outer-fixed* at (ℓ, τ) if its layer and tree fields are fixed to ℓ and τ respectively.

[*]With Claude (Anthropic), which co-authored the document drafting and LaTeX conversion.

Address convention (RFC 8391). This is a critical point that must be pinned precisely. When the RFC computes an internal node at height h (i.e., the parent of two nodes at height $h-1$), it sets:

$$\begin{aligned}\mathsf{treeHeight} &= h-1 \quad \text{(the height of the \textit{children}, not the parent)}\\ \mathsf{treeIndex} &= j \quad \text{(the index of the \textit{parent} node among nodes at height } h\text{)}\end{aligned}$$

This is confirmed by Algorithm 9 (treeHash), which increments treeHeight *after* calling `RAND_HASH`, and by Algorithm 13 (rootFromSig), which sets treeHeight $= k$ when computing the node at height $k+1$.

We adopt this convention throughout. Define the *canonical hash-tree address* for computing a node at height h (with index j) within a tree with outer fields (ℓ, τ) as:

$$\mathsf{addr}(\ell, \tau, h, j) := (\text{type=HASH, layer=}\ell\text{, tree=}\tau\text{, treeHeight=}h{-}1\text{, treeIndex=}j)$$

That is, $\mathsf{addr}(\ell, \tau, h, j)$ is the address used as input to H when *producing* the node at height h, index j. The treeHeight field carries the value $h-1$.

2.2 Primitive hash functions

We treat the following as black-box primitives:

- $\mathsf{leaf}(i;\ \mathsf{SK},\ \mathsf{PK\_SEED},\ \mathsf{adrs})$: the n-byte leaf value at index i, computed as $\mathsf{lTree}(\mathsf{WOTS{+}genPK}(\ldots))$. The address adrs must have type OTS (for genPK) and type LTREE (for lTree); we suppress the internal address details here.
- $H(\mathsf{adrs}, L, R)$: the n-byte hash of two n-byte nodes L and R under address adrs.

3 Two Definitions of the XMSS Tree

3.1 Recursive definition

Fix outer fields (ℓ, τ), secret seed SK, public seed PK_SEED, and starting leaf index s (a multiple of 2^h).

Define $\mathsf{Tree}(h, s)$ by induction on h:

$$\begin{aligned}\mathsf{Tree}(0, i) &:= \mathsf{leaf}(i;\ \mathsf{SK},\ \mathsf{PK\_SEED},\ \cdot)\\ \mathsf{Tree}(h, s) &:= H\Big(\mathsf{addr}(\ell, \tau, h, s/2^h),\ \ \mathsf{Tree}(h-1, s),\ \ \mathsf{Tree}(h-1, s+2^{h-1})\Big)\end{aligned}$$

The *root* of the XMSS tree is $\mathsf{Tree}(h, 0)$.

3.2 Iterative definition (Algorithm 9)

The iterative algorithm maintains a stack σ of pairs (v, k) where v is an n-byte node value and $k \geq 0$ is its height.

There are two variants of interest, which differ only in how they compute the address for each merge. Their equivalence is established by straightforward arithmetic (Lemma 1).

The formula for j computes the global index of the parent node directly from the current loop variable `idx`, the subtree start s, and the merge height `node_h`, without reference to any previous iteration's address state.

```
Stack sigma := []
for i = 0 to 2^h - 1:
    idx := s + i
    leaf_val := leaf(idx)
    ADRS.setTreeHeight(0)
    ADRS.setTreeIndex(idx)           // start: leaf's own index
    push(sigma, (leaf_val, 0))
    while |sigma| >= 2 and height(sigma[-2]) = height(sigma[-1]):
        ADRS.setTreeIndex((ADRS.getTreeIndex() - 1) / 2)   // parent index
        (L, _) := pop(sigma[-2]);  (R, _) := pop(sigma[-1])
        v := RAND_HASH(L, R, SEED, ADRS)
        ADRS.setTreeHeight(ADRS.getTreeHeight() + 1)        // after the hash
        push(sigma, (v, ADRS.getTreeHeight()))
return pop(sigma)
```

Figure 1: RFC 8391 Algorithm 9 (stateful address).

```
Stack sigma := []
for idx = s to s + 2^h - 1:
    leaf_val := leaf(idx)
    push(sigma, (leaf_val, 0))
    while |sigma| >= 2 and height(sigma[-2]) = height(sigma[-1]):
        node_h := height(sigma[-2])                              // children's height
        (L, _) := pop(sigma[-2]);  (R, _) := pop(sigma[-1])
        j := (s >> (node_h + 1)) + ((idx - s) >> (node_h + 1))
        adrs := addr(l, t, node_h + 1, j)                       // using our notation
        v := H(adrs, L, R)
        push(sigma, (v, node_h + 1))
return value(sigma[0])
```

Figure 2: Closed-form address variant.

4 Key Lemmas

4.1 Address Arithmetic

The two variants in Figures 1 and 2 differ only in how they compute treeIndex at each merge. The following lemma collects the arithmetic facts needed to show the two formulas always agree.

Lemma 1 (Address arithmetic)**.** *Let s, i be non-negative integers with $2^h \mid s$ for some $h \geq 1$, and let $1 \leq m \leq h$.*

(a) ***Shift decomposition.***

$$\left\lfloor \frac{s+i}{2^m} \right\rfloor = \frac{s}{2^m} + \left\lfloor \frac{i}{2^m} \right\rfloor$$

(b) ***Iterated halving.*** *If bits 0 through $m-1$ of a non-negative integer x are all 1, then applying the map $x \mapsto (x-1)/2$ exactly m times yields $\lfloor x/2^m \rfloor$.*

Proof. *(a)* Since $2^m \mid s$ (because $m \leq h$ and $2^h \mid s$), $s/2^m$ is an integer. For any integer a and real r, $\lfloor a + r \rfloor = a + \lfloor r \rfloor$. Apply with $a = s/2^m$ and $r = i/2^m$. □

(b) By induction on m:

- *Base ($m = 0$)*: zero applications; $\lfloor x/1 \rfloor = x$. ✓
- *Step ($m \to m + 1$)*: Suppose bits 0 through m of x are all 1. Bit 0 is 1, so x is odd, say $x = 2q + 1$. One application: $(x - 1)/2 = q = \lfloor x/2 \rfloor$. Bits 0 through $m-1$ of q: bit j of q equals bit $j+1$ of x, which is 1 for $j = 0, \ldots, m-1$. By the induction hypothesis, m more applications to q give $\lfloor q/2^m \rfloor = \lfloor x/2^{m+1} \rfloor$. ✓

□

Remark (Alignment). The condition $2^h \mid s$ is what makes part (a) work: it ensures $s/2^m$ is exact for all $m \le h$. Algorithm 9 checks this explicitly: `if (s % (1 « t) != 0) return -1`.

Remark (Special case $s = 0$). When $s = 0$, part (a) simplifies to $\lfloor i/2^m \rfloor$, and the closed-form formula becomes just $\mathsf{idx} \gg m$. The general case $s \neq 0$ arises in XMSS-MT (subtrees at non-zero offsets) and in naive auth-path computation.

4.2 The Stack Invariant

The key to the equivalence proof is the following invariant, which holds after each outer loop iteration.

Lemma 2 (Stack invariant)**.** *After processing leaves $s, s+1, \ldots, s+k-1$ (i.e. after k iterations of the outer loop, $1 \le k \le 2^h$), the stack σ satisfies:*

1. ***Heights**: the heights of stack entries, read from bottom to top, are exactly the positions of the set bits in k, in decreasing order. In particular, $|\sigma| = \mathsf{popcount}(k)$.*
2. ***Values**: if the i-th stack entry (from the bottom, 0-indexed) has height h_i, then its value is $\mathsf{Tree}(h_i, s_i)$, where $s_i = s + \sum_{j<i} 2^{h_j}$ is the starting leaf of the corresponding canonical subtree. (That is, s_i is s plus the total number of leaves covered by entries below entry i.)*
3. ***Addresses**: every hash-tree merge that produced a node $\mathsf{Tree}(h', s')$ (in this or any previous iteration) used the address $\mathsf{addr}(\ell, \tau, h', s'/2^{h'})$, matching the recursive definition.*

Proof. By induction on k.

Base case ($k = 1$). After processing leaf s (the first iteration, $\mathsf{idx} = s$, $i = 0$):

The leaf is pushed at height 0. The merge condition checks whether the top two entries have equal height; since the stack has only one entry, no merge occurs.

- *Heights*: $k = 1 = (1)_2$, whose only set bit is at position 0. The stack has one entry at height 0. ✓
- *Values*: The entry is $(\mathsf{leaf}(s), 0)$. By the recursive definition, $\mathsf{Tree}(0, s) = \mathsf{leaf}(s)$. ✓
- *Addresses*: No internal hash calls were made (the leaf computation uses OTS and L-tree addresses, which are not hash-tree addresses and are outside the scope of this invariant). Vacuously true. ✓

Inductive step ($k \to k + 1$, for $1 \le k < 2^h$). Assume the invariant holds after k iterations. We process leaf $\mathsf{idx} = s + k$ (the $(k + 1)$-th leaf, with $i = k$).

Step 1: State before the push. By the induction hypothesis, the stack heights are the set-bit positions of k in decreasing order (bottom to top), and each entry at height h_j holds $\mathsf{Tree}(h_j, s_j)$ for the appropriate starting index s_j.

Step 2: Push. The new leaf $\mathsf{leaf}(s + k)$ is pushed at height 0. The stack now has the entries for the binary decomposition of k, plus a height-0 entry on top.

Step 3: Merge loop (carry propagation). Let $r \geq 0$ be the number of trailing 1-bits of k (i.e., $k \bmod 2^r = 2^r - 1$ and either $r = h$ or bit r of k is 0).

By the induction hypothesis (Part 1), the stack from iteration k has entries at heights corresponding to the set bits of k. In particular, bits 0 through $r{-}1$ of k are all 1, so the stack has entries at heights $0, 1, \ldots, r{-}1$ (among others). After the push in Step 2, the top of the stack has two height-0 entries (if $r \geq 1$), triggering the first merge. Each merge at height t produces a height-($t{+}1$) entry, which pairs with the existing height-($t{+}1$) entry from the stack (if $t + 1 < r$), triggering the next merge. The loop performs exactly r merges, at children's heights $0, 1, \ldots, r{-}1$, and stops because the stack has no entry at height r (bit r of k is 0, or we have consumed all entries). This is the binary carry propagation for the increment $k \to k + 1$.

Merge t (at children's height t, for $t = 0, 1, \ldots, r{-}1$):

The top two stack entries both have height t: one from the binary decomposition of k (the induction hypothesis entry at bit position t), the other from the previous merge (or the pushed leaf, if $t = 0$).

We write $s_L^{(t)}$ for the starting leaf of the left (lower) entry at this merge. The left entry holds $\mathsf{Tree}(t, s_L^{(t)})$ (from the induction hypothesis) and the right entry holds $\mathsf{Tree}(t, s_R^{(t)})$ where $s_R^{(t)} = s_L^{(t)} + 2^t$ (from the induction hypothesis and construction).

Identifying $s_L^{(t)}$. We claim $s_L^{(t)} = \mathsf{idx} - 2^{t+1} + 1$, i.e. the two children together cover the 2^{t+1} leaves ending at $\mathsf{idx} = s + k$.

For $t = 0$: the left entry is the induction hypothesis's topmost entry (height 0), which by the induction hypothesis has starting leaf $s + \sum_{j<|\sigma|-1} 2^{h_j} = s + (k - 1) = \mathsf{idx} - 1$. The right entry is the pushed leaf at idx. So $s_L^{(0)} = \mathsf{idx} - 1 = \mathsf{idx} - 2^1 + 1$. ✓

For $t > 0$: the right entry is the result of merge $t{-}1$, which (by the case $t{-}1$ of this analysis) covers 2^t leaves ending at idx, starting at $\mathsf{idx} - 2^t + 1$. The left entry is the induction hypothesis entry at height t, which by contiguity of the induction hypothesis's subtree decomposition ends where the right child begins: its range is $[s_L^{(t)},\ \mathsf{idx} - 2^t + 1)$, of size 2^t. So $s_L^{(t)} = \mathsf{idx} - 2^t + 1 - 2^t = \mathsf{idx} - 2^{t+1} + 1$. ✓

Bit condition. Since $t < r$, bits 0 through t of $k = \mathsf{idx} - s$ are all 1. Because $2^h \mid s$ (with $h > t$), the low $t + 1$ bits of s are 0, so bits 0 through t of $\mathsf{idx} = s + k$ are also all 1. In particular, $\mathsf{idx} \bmod 2^{t+1} = 2^{t+1} - 1$.

Address correctness. The merged node is at height $t + 1$; its canonical address requires $\mathsf{treeIndex} = s_L^{(t)}/2^{t+1}$.

By the bit condition:

$$\left\lfloor \frac{\mathsf{idx}}{2^{t+1}} \right\rfloor \cdot 2^{t+1} = \mathsf{idx} - (\mathsf{idx} \bmod 2^{t+1}) = \mathsf{idx} - (2^{t+1} - 1) = s_L^{(t)}.$$

So $\lfloor \mathsf{idx}/2^{t+1} \rfloor = s_L^{(t)}/2^{t+1}$. ✓

The closed-form formula computes $(s \gg (t{+}1)) + ((\mathsf{idx} - s) \gg (t{+}1))$, which equals $\lfloor \mathsf{idx}/2^{t+1} \rfloor$ by Lemma 1(a). ✓

The stateful formula (Figure 1) reaches the same value: the map $x \mapsto (x{-}1)/2$ has been applied $t + 1$ times to idx, and the bit condition gives $\lfloor \mathsf{idx}/2^{t+1} \rfloor$ by Lemma 1(b). ✓

Value correctness. By the recursive definition:

$$\mathsf{Tree}(t+1, s_L^{(t)}) = H\Big(\mathsf{addr}(\ell, \tau, t+1, s_L^{(t)}/2^{t+1}),\ \mathsf{Tree}(t, s_L^{(t)}),\ \mathsf{Tree}(t, s_L^{(t)} + 2^t)\Big)$$

The merge uses the correct address ($s_L^{(t)}/2^{t+1}$ as shown above), the correct left child, and the correct right child ($s_R^{(t)} = s_L^{(t)} + 2^t$). So the result is $\mathsf{Tree}(t+1, s_L^{(t)})$. ✓

Step 4: Resulting stack. After r merges, the r induction hypothesis entries at heights $0, 1, \ldots, r-1$ and the pushed leaf have been replaced by a single entry at height r, holding $\mathsf{Tree}(r, s')$ where $s' = s_L^{(r-1)} = \mathsf{idx} - 2^r + 1 = s + k - 2^r + 1$.

The remaining entries (at heights corresponding to bits $> r$ of k, which are the same as bits $> r$ of $k+1$) are unchanged. The resulting stack heights are the set-bit positions of $k+1$, in decreasing order.

Verification of starting indices. We must check Part 2 of the invariant for $k+1$. The starting leaf of the new height-r entry is $s' = s + k - 2^r + 1$. By the Part 2 formula, it should equal $s + \sum_{j<i'} 2^{h_j}$, where the sum is over entries below it in the new stack. These are exactly the unchanged induction hypothesis entries at bit positions $> r$ of k, whose sizes sum to $k - (2^0 + 2^1 + \cdots + 2^{r-1}) = k - (2^r - 1)$. So the formula gives $s + k - 2^r + 1 = s'$. ✓

The unchanged entries retain their induction hypothesis starting leaves (the entries below them are also unchanged). ✓

Address correctness. Every hash call in the merge loop uses a canonical address by the argument in Step 3 (via Lemma 1). Hash calls from previous iterations are canonical by the induction hypothesis. ✓

This completes the induction. □

Corollary 1 (Termination and correctness)**.** *At $k = 2^h$, $|\sigma| = 1$ and $\sigma[0] = (\mathsf{Tree}(h, s), h)$. The algorithm returns $\mathsf{Tree}(h, s)$ with all internal hashes at canonical addresses.*

5 Main Theorem: XMSS

Theorem 1 (Iterative–recursive equivalence, XMSS)**.** *Let* SK*,* PK_SEED *be fixed. Then:*

$$\mathsf{treehash}(\mathsf{SK}, \mathsf{PK\_SEED}, 0, 2^h) \ = \ \mathsf{Tree}(h, 0)$$

and every call to H inside treehash *uses the canonical address for the node it computes.*

Proof. Immediate from Lemma 2 at $k = 2^h$. □

6 Extension to XMSS-MT

XMSS-MT composes d layers of XMSS trees, each of height $h' = h/d$. A tree at layer ℓ and tree-index τ is computed by calling treehash with outer address fields fixed to (ℓ, τ).

The additional difficulty over the single-tree case is:

- **Shared hash**: all layers use the same hash function H, with domain separation achieved entirely through the address fields layer and tree. The proof must track that these outer fields are consistently set for every hash call throughout the computation.

- **Offset starts ($s \neq 0$)**: while the full tree at each layer uses $s = 0$, subtree computations (e.g. for auth paths) use $s \neq 0$. Lemma 2 already handles arbitrary aligned s, so this requires no additional argument beyond the outer-field tracking.

Lemma 3 (Outer-field consistency)**.** *For any call to* treehash *with outer fields* (ℓ, τ) *and start* s*, every hash call inside uses an address with* layer $= \ell$ *and* tree $= \tau$.

Proof. By inspection of Algorithm 9 (Figures 1 and 2).

Every address used inside treehash is constructed by copying the caller-provided address adrs and then modifying only *inner* fields: type, and whichever type-specific fields apply (OTS, LTree, treeHeight, treeIndex, etc.).

By the address structure (Appendix A), the outer fields layer (word 0) and tree (words 1–2) are distinct from the type word (word 3) and the type-specific fields (words 4–7). The setType operation zeroes words 4–7 (per RFC 8391, §2.5) but does not touch words 0–2. No other operation in Algorithm 9 modifies words 0–2.

Therefore, every address used inside treehash—whether for OTS key generation, L-tree compression, or hash-tree merging—carries layer $= \ell$ and tree $= \tau$, inherited from the caller's adrs. □

Theorem 2 (Iterative–recursive equivalence, XMSS-MT)**.** *For each layer* $\ell \in [0, d)$ *and tree-index* τ*, the iterative* treehash *with outer fields* (ℓ, τ) *computes* $\mathsf{Tree}_{\ell,\tau}(h', 0)$ *(the recursive tree for that layer and tree), with every internal hash call at the canonical address for that layer and tree.*

Proof. Fix a layer ℓ and tree index τ. The XMSS-MT construction calls treehash with parameters $s = 0$, $t = 2^{h'}$, and an address adrs with layer $= \ell$ and tree $= \tau$.

Value correctness. Since $s = 0$ is trivially a multiple of $2^{h'}$, the alignment precondition of Lemma 2 is satisfied. By Lemma 2 at $k = 2^{h'}$, the algorithm returns $\mathsf{Tree}(h', 0)$. Parametrising by the outer fields, this is $\mathsf{Tree}_{\ell,\tau}(h', 0)$. ✓

Inner-field correctness. By Lemma 2(3), every hash-tree merge inside treehash uses the canonical address $\mathsf{addr}(\ell, \tau, h_{\text{node}}+1, j)$ for the node it computes, with correct treeHeight and treeIndex. ✓

Outer-field correctness. By Lemma 3, every address used inside treehash—including OTS, L-tree, and hash-tree addresses—carries layer $= \ell$ and tree $= \tau$, inherited from the caller. ✓

Combining these, the iterative computation produces $\mathsf{Tree}_{\ell,\tau}(h', 0)$ with every internal hash call at the canonical address for layer ℓ, tree τ. Since this holds for all (ℓ, τ), and the layers use the same hash function H with domain separation achieved entirely through these address fields, the full XMSS-MT tree is correctly computed. □

7 Remarks toward Formalisation

A mechanised proof in EasyCrypt (or similar) will need to:

- Represent the stack as a concrete data structure with a bounded-size invariant; the bound $h + 1$ follows from Lemma 2(1).
- State the loop invariant (Lemma 2) as a loop annotation in a program logic.
- Handle the address arithmetic (Lemma 1) and the carry-propagation argument (Lemma 2, Step 3) concretely in the proof assistant's integer/bitvector theory.

- Lift from the single-tree to the multi-tree case by parametrising over (ℓ, τ) and showing independence of inner-field computations from outer fields.

The fact that the iterative and recursive definitions are extensionally equivalent does *not* immediately give security: the security proof also needs the recursive structure to apply the collision-resistance and second-preimage-resistance of H at each level. The equivalence theorem here is a prerequisite for that reduction.

A Address Conventions (RFC 8391)

The treeHeight/treeIndex convention is stated in Section 2.1 and is pinned against two RFC algorithms:

- **Algorithm 9** (treeHash): sets `treeHeight` before the merge, increments it *after* `RAND_HASH` returns. At the point of the hash call, `treeHeight` holds the children's height.
- **Algorithm 13** (rootFromSig): sets treeHeight $= k$ in the loop header when computing the node at height $k + 1$.

Both are consistent: the treeHeight field in the address encodes the height of the *inputs* to the hash, not the output.

The ADRS structure is 32 bytes, consisting of 8 big-endian 32-bit words. Words 0–2 are shared across all address types ("outer fields"); word 3 is the type discriminator; words 4–7 are type-specific ("inner fields").

Common prefix (all types):

Word	Bytes	Field	Width
0	0–3	layer address	32 bits
1	4–7	tree address (high)	32 bits
2	8–11	tree address (low)	32 bits
3	12–15	type	32 bits

The tree address is a 64-bit value stored across words 1 (most significant) and 2 (least significant).

Type 0 — OTS hash address:

Word	Bytes	Field
4	16–19	OTS address
5	20–23	chain address
6	24–27	hash address
7	28–31	keyAndMask

Type 1 — L-tree address:

Word	Bytes	Field
4	16–19	L-tree address
5	20–23	tree height
6	24–27	tree index
7	28–31	keyAndMask

Type 2 — Hash tree address:

Word	Bytes	Field
4	16–19	padding (= 0)
5	20–23	tree height
6	24–27	tree index
7	28–31	keyAndMask

Domain separation rule (RFC 8391, §2.5). Whenever the type word (word 3) is changed, words 4–7 must be zeroed before any type-specific fields are set. This prevents stale field values from a previous type from leaking into the new address.

Serialisation. The 8 words are serialised in order, each in big-endian, producing the 32-byte value passed to hash functions. For a mechanised proof, the address is most conveniently modelled as an array of 8 unsigned 32-bit words with big-endian serialisation.

Errata 7900 note. Errata 7900 corrects the SK serialisation byte layout but does not affect the ADRS structure. The address layout above is unchanged by the errata.