Merkle Trees and Signatures

Chris Peikert, Idan Meshita, and Riad Wahby

Algorand, Inc.

February 8, 2022

For a non-negative integer k, let $[k] := \{0, 1, \ldots, k-1\}$. Recall that a *collision* in a function H is a pair of distinct inputs $m \neq m'$ such that $H(m) = H(m')$, and a *claw* in a pair of functions H, H' is a pair of (not necessarily distinct) inputs m, m' such that $H(m) = H'(m')$.

1 Merkle Trees and Vector Commitments

This section defines:

1. a version of Merkle trees, which allows for concisely committing to a possibly large, implicitly *unordered* set of data entries;
2. a version of vector commitments, which allows for concisely committing to an *ordered* (indexed) vector of data entries, based on Merkle trees.

Binary trees. A *complete* (also known as *almost complete*) binary tree is one in which every level is completely filled, except for possibly the leaf level, in which the leaves are filled from the left. Such a tree T can be:

- *empty*, meaning it has zero nodes (not even a root node);
- a *leaf*, meaning it has exactly one node (which is its root);
- or neither of these, in which case it has a non-empty left child tree denoted T.left, and a (possibly empty) right child tree denoted T.right.

A *perfect* binary tree is a complete one where every layer, including the leaf layer, is completely filled.

Merkle tree overview. In the present version of Merkle trees, the data entries are placed at the leaves of an (almost) complete binary tree, with one entry per leaf. Every binary tree T, even the empty one, has an attribute T.digest, which holds a hash digest representing the contents of the entire tree:

- The digest of the empty tree is always defined to be $T.\text{digest} = \mathbf{0}$, a canonical (but arbitrary) digest value.
- The digest of a leaf tree is defined to be the hash of its associated data entry, using a specified function for hashing entries.

- The digest of a non-empty, non-leaf tree is defined to be the hash of the digests of its left and right child trees, using a specified function for hashing pairs of digests.

The two hash functions should satisfy certain collision-resistance properties individually and together; in particular, they must be different functions, but they can be defined using a single collision-resistant hash function via appropriate domain separation. See **??** and **??** for further details.

The abstract syntax (interface) for Merkle trees is as follows (see **??** for the formal definitions):

- $\mathsf{MT.Build}$ takes an (almost) complete tree with a data entry at each one of its leaves, and populates the digests of the tree and all its subtrees, also returning the digest of the entire tree.
- $\mathsf{MT.Prove}$ takes a built tree (i.e., one whose digests have been populated by $\mathsf{MT.Build}$) and a root-to-leaf path specified by a binary string (where 0 and 1 respectively denote left and right), and returns a proof consisting of one hash digest per bit of the string. If the provided path does not lead from the root to a leaf (because the path is either too short or too long), then the output is an error. In particular, there is no valid path for an empty tree.
- $\mathsf{MT.Verify}$ takes a digest representing a commitment to an entire tree (of arbitrary size and topology), a claimed data entry, a root-to-leaf path specified by a binary string of some non-negative length (possibly zero), and a purported proof consisting of one hash digest per bit of the string. It either accepts or rejects, indicating whether these inputs represent a valid proof that the claimed entry is located at the leaf specified by the claimed path.

Vector commitment overview. The abstract syntax (interface) for vector commitments is as follows (see **??** for the formal definitions):

- $\mathsf{VC.Commit}$ takes an ordered vector of any non-negative number (possibly zero) of data entries, and returns a commitment along with some auxiliary data that is later used for proving individual entries.
- $\mathsf{VC.Prove}$ takes the auxiliary data (as returned by $\mathsf{VC.Commit}$) and a valid index into the committed vector, and returns a proof. (Concretely, the proof is an ordered tuple of hash digests, as returned by $\mathsf{MT.Prove}$.)
- $\mathsf{VC.Verify}$ takes a commitment (concretely, a hash digest) to some vector of any valid dimension, a vector index, a claimed data entry, and a purported proof (concretely, an ordered tuple of some non-negative number of hash digests). It either accepts or rejects, indicating whether these inputs represent a valid proof that the claimed entry is located at the specified index of the committed vector. The provided index must be in $[2^d]$, where d is the number of hash digests in the proof; otherwise $\mathsf{VC.Verify}$ immediately rejects.

The vector commitment scheme VC is defined essentially as a "thin wrapper" around the Merkle tree scheme MT, with two pieces involved in the translation. First, VC uses a suitable correspondence between vector indices and root-to-leaf paths. Namely, index i corresponds to a path specified by the binary representation of i, from *least*- to *most*-significant bit. This ordering (which is the "bit-reversed" version of the usual indexing of leaves in a binary tree) is *essential* for the position-binding security property; see **??** and the discussion following it. Note that a given path uniquely defines an index (e.g., path 0111 corresponds to index 14), but a given index can yield multiple paths (e.g., index 13 corresponds to paths 1011, 10110, 101100, etc.). However, the VC functions always have additional context that determines the proper path length to use (called d in the pseudocode), and with this context, an index uniquely defines a path.

Second, to ensure that VC provides MT with a complete tree (which is not immediate, due to the above-described indexing), VC.Commit also pads its input vector with special, distinguished 'nothing' values to make the vector have a power-of-two length. (These 'nothing' values must be hashed appropriately, which can be done by suitable domain separation.) This ensures that the resulting tree is perfect, and hence complete.

Algorithm 1 Merkle tree scheme (MT).

- Let $\mathcal{M} = \{0,1\}^* \cup \{\bot\}$, the set of all binary strings together with a distinguished 'nothing' value $\bot$, denote the space of valid vector entries.
- Let $\mathsf{HashE}\colon \mathcal{M} \to \mathcal{H}$ be a hash function for vector entries, and let $\mathsf{HashC}\colon \mathcal{H} \times \mathcal{H} \to \mathcal{H}$ be a hash function for children of internal nodes. Let $\mathbf{0} \in \mathcal{H}$ denote a special hash digest, which may be arbitrary.

  These hash functions can be instantiated using a single function with appropriate domain separation.

```
1: function MT.Build($T$)
       ▷ in a complete binary tree, compute all digests from its leaf entries (if any); return the tree's digest
2:     if $T$ is empty then do nothing                                   ▷ $T$.digest $= \mathbf{0}$ already
3:     else if $T$ is a leaf then $T$.digest $=$ HashE($T$.entry)
4:     else $T$.digest $=$ HashC(MT.Build($T$.left), MT.Build($T$.right))
5:     return $T$.digest
6: function MT.Prove($T, p = p_1 \cdots p_d \in \{0,1\}^d$) ▷ prove entry at the root-to-leaf path $p$ of a built tree $T$
7:     if $d = 0$ then                                                    ▷ path is empty...
8:         if $T$ is a leaf then return () else return error              ▷ ... so tree must be a leaf
9:     if $T$ is empty or a leaf then return error          ▷ path is non-empty, so children are required
10:    if $p_1 = 0$ then $T' = T$.left, $\pi = T$.right.digest else $T' = T$.right, $\pi = T$.left.digest
11:    return $\pi$, MT.Prove($T', p_2 \cdots p_d$)                  ▷ prepend sibling digest to proof for subtree
12: function MT.Verify($C \in \mathcal{H}, m \in \mathcal{M}, p = p_1 \cdots p_d \in \{0,1\}^d, \pi = (\pi_1, \ldots, \pi_d) \in \mathcal{H}^d$)
       ▷ for commitment $C$, verify a proof $\pi$ that the entry at root-to-leaf path $p$ (both of length $d \geq 0$) is $m$
13:    let $h_d =$ HashE($m$) $\in \mathcal{H}$
14:    for $j = d, \ldots, 1$ do                             ▷ hash from leaf to root (doing nothing if $d = 0$)
15:        if $p_j = 0$ then $h_{j-1} =$ HashC($h_j, \pi_j$) else $h_{j-1} =$ HashC($\pi_j, h_j$)     ▷ hash left then right
16:    if $h_0 = C$ then return accept else return reject
```

Theorem 1.1. *If both* HashE *and* HashC *are collision resistant, and the pair* HashE, HashC *is claw resistant, then the vector-commitment scheme from* **??** *is position binding.*

That is, it is infeasible for an adversary to output a commitment C, an index i, two distinct *messages $m, m' \in \mathcal{M}$, and two proofs π, π' (of any, possibly different lengths) such that both* VC.Verify(C, i, m, π) *and* VC.Verify(C, i, m', π') *accept.*

Note that this theorem crucially relies on the fact that the root-to-leaf path to entry i follows the *LSB-to-MSB* binary representation of i. By contrast, using the MSB-to-LSB representation (as is done in some Merkle tree implementations) *makes it trivially easy to break the position binding property*. (For example, an adversary can easily create a commitment and open its entry $i = 1$ in two different ways, using proofs of different 'depths' d.) Therefore, for security it is *vital* to use a proper correspondence between indices and paths.

Algorithm 2 Vector-commitment scheme (VC) based on Merkle trees.

```
1: function $\mathsf{VC.Commit}(m_0, m_1, \ldots, m_{k-1} \in \mathcal{M})$        ▷ commit to a vector of any dimension $k \geq 0$
2:     let $d \geq 0$ be the smallest integer such that $2^d \geq k$
3:     for $i = k, \ldots, 2^d - 1$ do let $m_i = \perp$
4:     construct a perfect depth-$d$ binary tree $T$, where:
         for each $i \in [2^d]$, the entry given by the root-to-leaf path $\mathsf{RootToLeafPath}(d, i)$ is $m_i$.
5:     $C = \mathsf{MT.Build}(T)$
6:     return commitment $C$ and auxiliary proving data $(T, k, d)$
7: function $\mathsf{VC.Prove}((T, k, d), i \in [k])$      ▷ open entry $i$ of a commitment using auxiliary data $(T, k, d)$
8:     return $\mathsf{MT.Prove}(T, \mathsf{RootToLeafPath}(d, i))$
9: function $\mathsf{VC.Verify}(C \in \mathcal{H}, i \in [2^d], m \in \mathcal{M}; \pi = (\pi_1, \ldots, \pi_d) \in \mathcal{H}^d)$
                 ▷ for commitment $C$, verify a proof $\pi$ that entry $i$ is $m$ (where $d$ is determined by $\pi$)
10:    if $i \notin [2^d]$ then return false                              ▷ explicitly reject out-of-bounds indices
11:    return $\mathsf{MT.Verify}(C, m, \mathsf{RootToLeafPath}(d, i), \pi)$          ▷ use the path to the leaf storing entry $i$
12: function $\mathsf{RootToLeafPath}(d \geq 0, i \in [2^d])$ ▷ the root-to-leaf path $p \in \{0,1\}^d$ for index $i$, LSB to MSB
13:    if $d = 0$ then return $()$ else return $(i \bmod 2, \mathsf{RootToLeafPath}(d-1, \lfloor i/2 \rfloor) \in \{0,1\}^d)$
```

Proof. Consider an attacker against the position binding of VC. Suppose that it successfully outputs some $C \in \mathcal{H}, i \in [2^d], m \in \mathcal{M}, \pi = (\pi_1, \ldots, \pi_d) \in \mathcal{H}^d, m' \in \mathcal{M}, \pi' = (\pi'_1, \ldots, \pi'_{d'}) \in \mathcal{H}^{d'}$ where $d \leq d'$ without loss of generality, $m \neq m'$, and both $\mathsf{VC.Verify}(C, i, m, \pi)$ and $\mathsf{VC.Verify}(C, i, m', \pi')$ accept. From such an output we show how to efficiently extract a collision in HashE or HashC, or a claw in $\mathsf{HashE}, \mathsf{HashC}$.

Let

$$p = p_1 p_1 \cdots p_d = \mathsf{RootToLeafPath}(d, i) \in \{0,1\}^d$$
$$p' = p'_1 p'_1 \cdots p'_d = \mathsf{RootToLeafPath}(d', i) \in \{0,1\}^{d'}.$$

By definition of $\mathsf{RootToLeafPath}$, p is a prefix of p', i.e., $p_j = p'_j$ for all $j \in [d]$. Note that this is where we use the fact that $\mathsf{RootToLeafPath}$ outputs the LSB-to-MSB binary representation of i. (Under the MSB-to-LSB representation, this prefix property would not hold.)

Let digests $h_j, h'_j \in \mathcal{H}$ be as in the execution of $\mathsf{MT.Verify}(C, m, p, \pi)$ and $\mathsf{Verify}(C, m', p', \pi')$, respectively. By definition, we have $h_0 = C = h'_0$. Let $j \in \{1, 2, \ldots, d\}$ be the smallest integer such that $(h_j, \pi_j) \neq (h'_j, \pi'_j)$, if such an integer exists. We proceed with a case analysis based on whether j exists and whether $d = d'$.

First suppose that j exists. Then if $p_j = p'_j = 0$, we have

$$\mathsf{HashC}(h_j, \pi_j) = h_{j-1} = h'_{j-1} = \mathsf{HashC}(h'_j, \pi'_j),$$

so $(h_j, \pi_j) \neq (h'_j, \pi'_j)$ is a collision in HashC, as needed. The same reasoning holds if $p_j = p'_j = 1$, but with the arguments to HashC swapped.

If no such j exists, then $h_d = h'_d$. If $d = d'$, then we have

$$\mathsf{HashE}(m) = h_d = h'_d = \mathsf{HashE}(m'),$$

so $m \neq m'$ is a collision in HashE, as needed. Otherwise $d < d'$, and if $p'_{d+1} = 0$ we have

$$\mathsf{HashE}(m) = h_d = h'_d = \mathsf{HashC}(h'_{d+1}, \pi'_{d+1}),$$

so $m, (h'_{d+1}, \pi'_{d+1})$ is a claw in the pair $\mathsf{HashE}, \mathsf{HashC}$, as needed. The same reasoning holds if $p'_d = 1$, but with the arguments to HashC swapped. This completes the proof. $\square$

2 Merkle Signatures

This section specifies the *Merkle signature scheme* (MSS), which generically combines any vector-commitment scheme and a signature scheme $\mathsf{Sig} = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ to get another signature scheme. The key property of MSS is that it generates several *ephemeral* public and secret keys, each of which is only useful for a limited time, and the secret keys can be deleted once they have been used (or expire). This mitigates against the risk of a cryptanalytic attack on a long-lived public key.

To implement limited-time keys, MSS extends the usual syntax of a signature scheme via the concept of a *round*. The current value of the round is maintained externally (e.g., by a clock or a blockchain) and is publicly known. The set of rounds for which a given MSS key can sign is chosen in advance by the signing party, at the time of MSS key generation. The MSS signing algorithm is given the current round value, along with (as usual) the secret key and message. Finally, the MSS verifier is also given a specified round along with (as usual) a public key, message and purported signature, and checks that the signature is valid for that specific round.

Algorithm 3 Merkle signature scheme (MSS).

```
function MSS.Gen($r_0, r_1, \ldots, r_{k-1}$)                              ▷ generate keys for signing at each round $r_i$
    for $i \in [k]$ do $(pk_i, sk_i) \leftarrow$ Sig.Gen()
    let $(C, aux) =$ VC.Commit($(pk_0, r_0), \ldots, (pk_{k-1}, r_{k-1})$)     ▷ commit to all key-round pairs
    return public key $pk_{\mathsf{MSS}} = C$, non-secret $aux$, and secret key $sk_{\mathsf{MSS}} = (sk_0, \ldots, sk_{k-1})$
function MSS.Sign($sk_{\mathsf{MSS}}, aux, r, M \in \{0,1\}^*$)                       ▷ sign $M$ in round $r$
    in $aux$, find an index $i$ for which $r_i = r$ (and fail if no such $i$ exists)
    let $\pi =$ VC.Prove($aux, i$)
    let $\sigma \leftarrow$ Sig.Sign($sk_i, M$)
    return signature $\sigma_{\mathsf{MSS}} = (i, pk_i, \pi, \sigma)$
function MSS.Verify($pk_{\mathsf{MSS}}, r, M \in \{0,1\}^*, \sigma_{\mathsf{MSS}} = (i^*, pk^*, \pi^*, \sigma^*)$)
                                                            ▷ verify a signature on $M$ for round $r$
    return VC.Verify($pk_{\mathsf{MSS}}, i^*, (pk^*, r), \pi^*$) AND Sig.Verify($pk^*, M, \sigma^*$)
```