TLDR

: We suggest a sparse Merkle key-value tree with improved simplicity, efficiency and security compared to this recent construction.

Construction

Given 32-byte values l

and r

let h(l, r) = \text{sha256}(l, r) | 0b1

(i.e. hardcode the least significant bit to 1) if l, r \ne 0

and h(l, r) = \max(l, r)

otherwise. Build a sparse Merkle tree where the leaf at position k

has value \text{sha256}(k, v) \& 0b0

(i.e. hardcode the least significant bit to 0) for every key-value pairs (k, v)

, and use h

to accumulate the leaves into a root.

Whenever a key-value pair (k, v)

is authenticated (in particular, when modifying the tree statelessly) it suffices to validate the corresponding Merkle path from leaf to root, as well as

check that the leaf position k

is consistent with the leaf value \text{sha256}(k, v)

.

Security argument

Notice that h

is not collision resistant precisely when l = 0

or r = 0

, and h(l, r) = \max(l, r) = h(r, l)

. (Notice also that leaf values—and their accumulations with zero leaves—do not collide with h

when l, r \ne 0

because of the hardcoded bit.) We argue that the construction is nevertheless secure.

Notice the collisions when l = 0

or r = 0

allow for a non-zero leaf in an otherwise zero subtree to “move” within the subtree while preserving the Merkle root. Conversely, it is easy to see (e.g. with a recursive argument) that two equal-depth Merkle trees with the same root are equivalent modulo moving non-zero leaves in otherwise zero subtrees.

As such, validating a Merkle path from leaf to root authenticates the leaf within an otherwise zero subtree. The exact leaf position is then disambiguated by checking that the key k

is properly “mixed in” the leaf value \text{sha256}(k, v)

.

Discussion

The construction is comparable to Vitalik’s. The main difference is the disambiguation of non-zero leaf positions within zero subtrees. Vitalik’s construction zeroes out 16 bits from the sha256 outputs and uses the extra space as “position bits”. This construction keys the leaves instead, with several benefits:

- simplicity

: The accumulating function h

is cleaner and simpler.

- efficiency

: Vitalik's construction has an overhead of about 256/(256-240) = 16 hashes per Merkle path due to the overflowing of position bits. There is no such overhead here.

- security

: Vitalik's construction reduces preimage resistance by 16 bits due to the zeroing of hash output bits. Only a single bit of preimage security is lost in this construction.