How to hash a Merkle Tree

Potuz
Prysmatic Labs

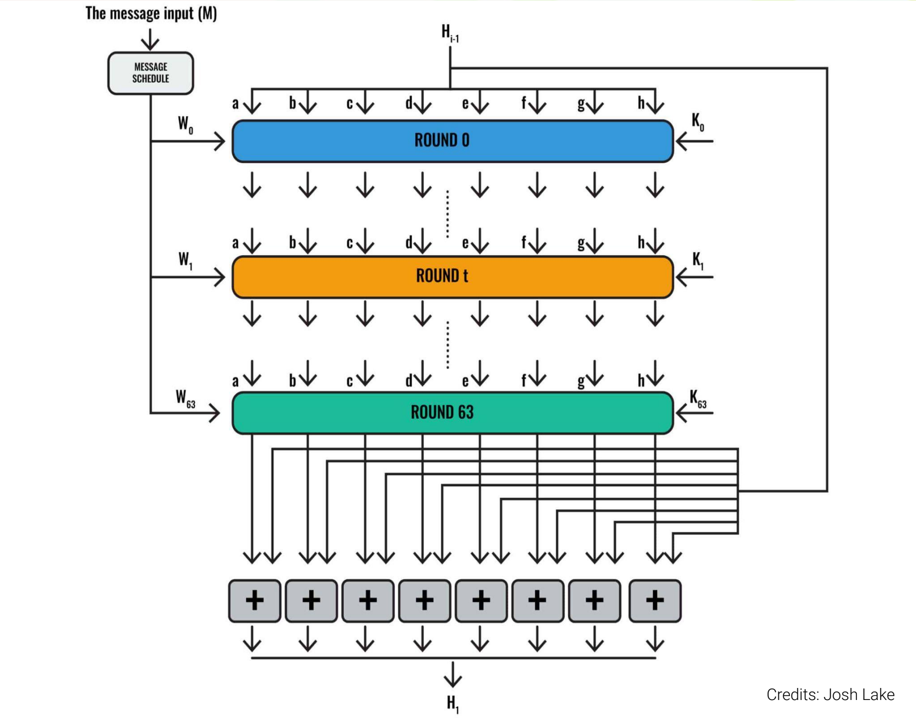
```
$ cd gohashtree
$ go test . -bench=.
goos: linux
goarch: amd64
pkg: github.com/prysmaticlabs/gohashtree
cpu: Intel(R) Xeon(R) CPU @ 2.80GHz
BenchmarkHash_1_minio-2                  2462506                473.1 ns/op
BenchmarkHash_1-2                        3040208                391.3 ns/op
BenchmarkHash_4_minio-2                   577078               1959 ns/op
BenchmarkHash_4-2                        1954473                604.9 ns/op
BenchmarkHash_8_minio-2                   298208               3896 ns/op
BenchmarkHash_8-2                        1882191                624.8 ns/op
BenchmarkHash_16_minio-2                  147230               7933 ns/op
BenchmarkHash_16-2                        557485               1988 ns/op
BenchmarkHashLargeList_minio-2                10          105404666 ns/op
BenchmarkHashList-2                           45           25368532 ns/op
PASS
ok      github.com/prysmaticlabs/gohashtree     13.969s
```

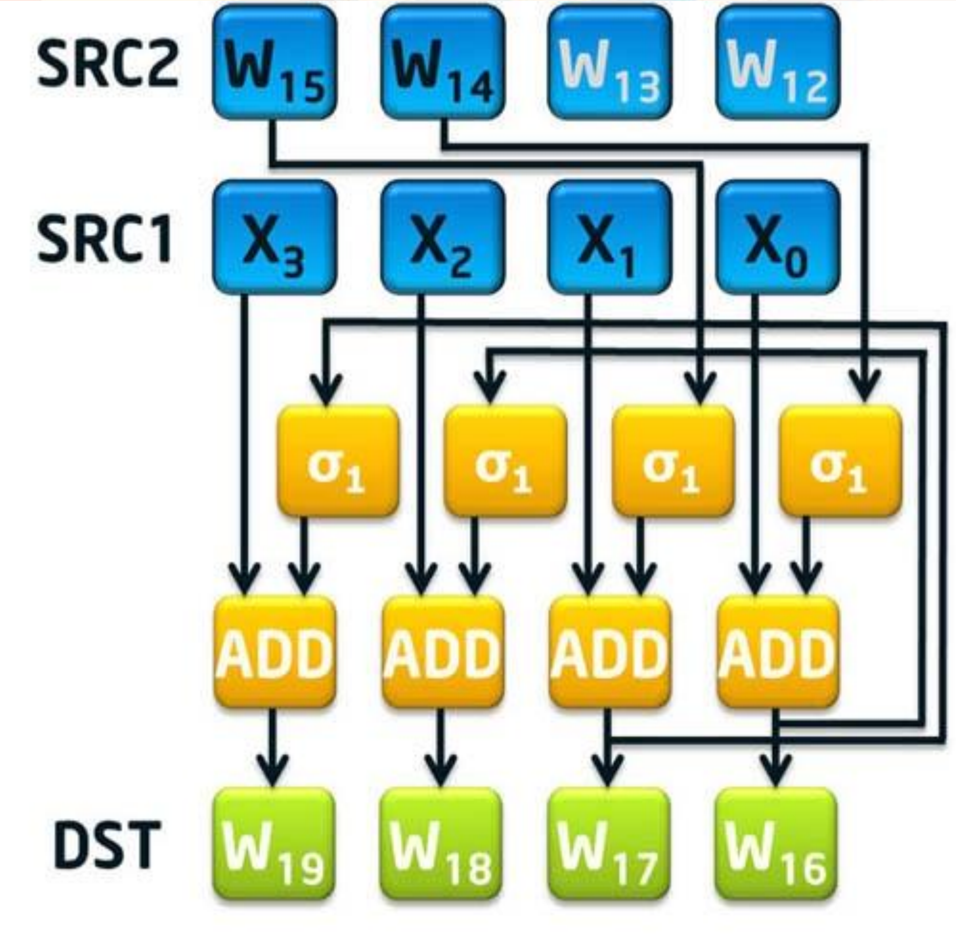
Section 1

SHA 256 Basics

SHA 256 Basics

- Break into 64 bytes chunks
- Schedule 64 dwords (4 bytes)
  - $W_0,\dots,W_{15}$ are the message
  - $W_i,\dots,W_{i+15}$ are computed in terms of $W_{i-16},\dots,W_{i-1}$.
- Start with an incoming digest of 8 dwords $(a_0, \dots, h_0)$
- Round_i takes 10 dwords $(a_i, \dots, h_i;\ W_i, K_i)$ and returns $(a_{i+1}, \dots, h_{i+1})$.
- incoming digest for next chunk: $(a_0, \dots, h_0) + (a_{63}, \dots, h_{63})$





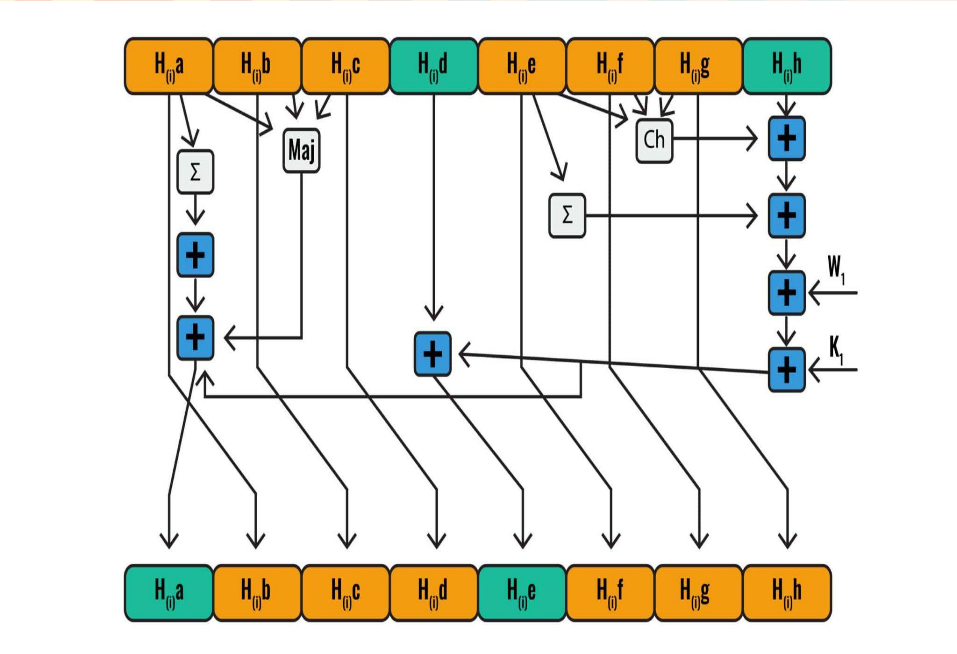
Message scheduling

$\sigma_0(W) = ROR_7(W) \wedge ROR_{18}(W) \wedge SHR_3(W)$

$\sigma_1(W) = ROR_{17}(W) \wedge ROR_{19}(W)$
$\wedge SHR_{10}(W)$

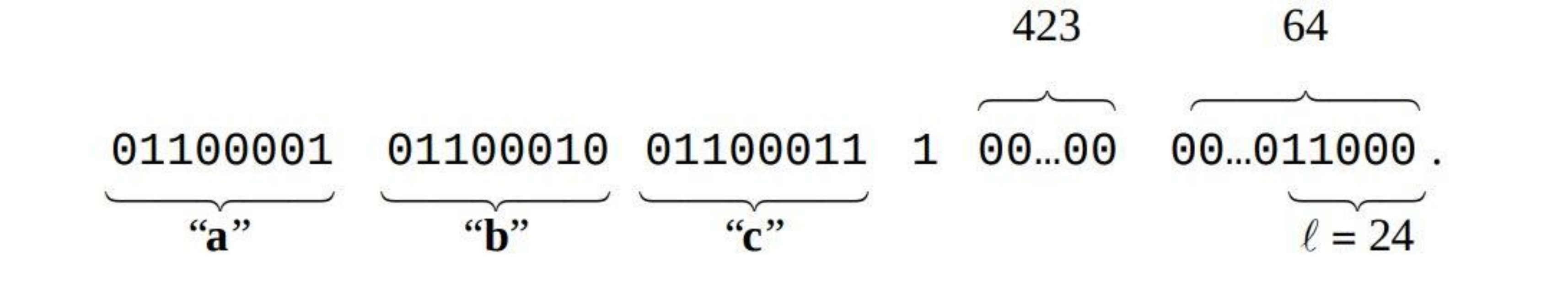
- Compute 4 words at a time
- Can be done in parallel to rounds
- Does not depend on previously processed chunks.

Rounds



- Incoming:
  - Status 8 dwords ($a_n, b_n, \ldots, g_n, h_n$)
  - Constant K_n
  - Scheduled word W_n
- **Outcoming:**
  - **Status 8 dwords ($h_{n+1}, a_{n+1} \ldots, f_{n+1}, g_{n+1}$)**
- Depends on previous steps.
- Depends on Scheduled words

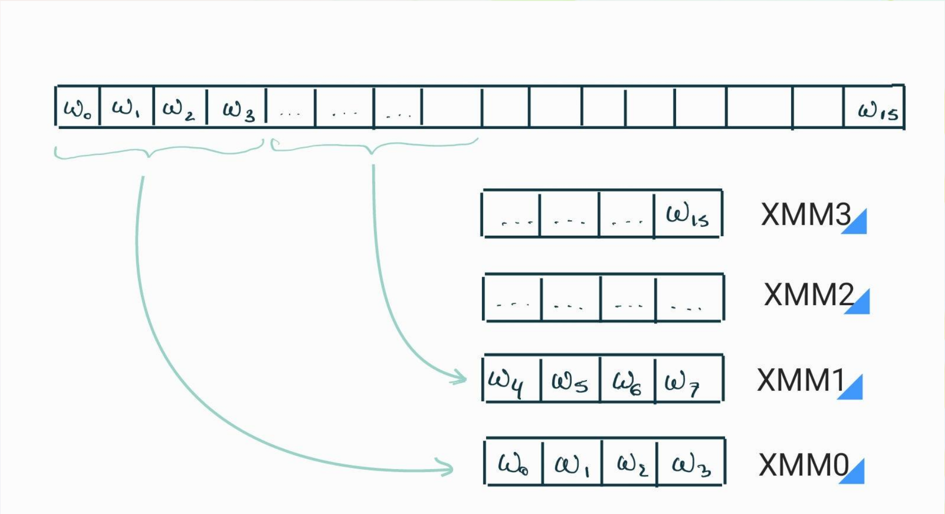
The padding Block



The last block contains the length of the message as a little endian `uint64`. This length occupies the last 64 bits of the last `512bits` (64 bytes block). A bit `1` is added after the last bit of the message, to signal its end.

Vectorization

- Word scheduling can be done in parallel
- AVX can schedule 4 dwords at a time
- AVX2 can schedule 8 dwords at a time
- AVX-512 can schedule 16 dwords at a time
- AVX-1024 …
- SIMD instructions can be interleaved with arithmetic ones for better pipelining
- **Rounds have to be scalar**



Hasher signature

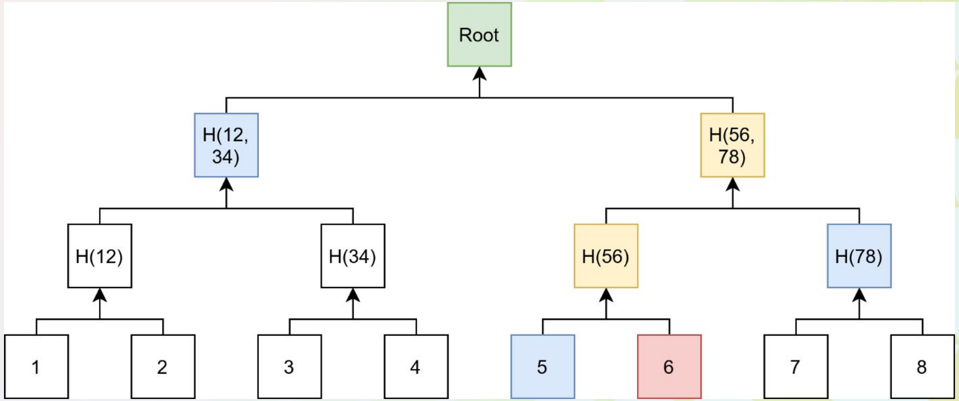
```
func hash(message []byte) [32]byte

def hash(data:bytes) -> Bytes32

pub fn hash(input: &[u8]) -> [u8; HASH_LEN]
```

Section 2

Merkle Trees

Parallelization + Fixed Size blocks

- Each node is a `256bit` hash
- Each node is the digest of hashing the concatenation of its two children (`512bits`)
- Siblings can be computed in parallel



Implementations

```
def merkle_root(self) -> Root:
    if self._root is not None:
        return self._root
    self._root = merkle_hash(self.left.merkle_root(), self.right.merkle_root())
    return self._root
```

```
def merkle_tree(leaves: Sequence[Bytes32]) -> Sequence[Bytes32]:
    bottom_length = get_power_of_two_ceil(len(leaves))
    o = [Bytes32()] * bottom_length + list(leaves) + [Bytes32()] * (bottom_length - len(leaves))
    for i in range(bottom_length - 1, 0, -1):
        o[i] = hash(o[i * 2] + o[i * 2 + 1])
    return o
```

Implementations

```
func NewUsing(data [][]byte, hash HashType, salt bool) (*MerkleTree, error) {
	…
	for i := len(data) + branchesLen; i < len(nodes); i++ {
		nodes[i] = make([]byte, hash.HashLength())
	}
	// Branches
	for i := branchesLen - 1; i > 0; i-- {
		nodes[i] = hash.Hash(nodes[i*2], nodes[i*2+1])
	}

	tree := &MerkleTree{
		salt:  salt,
		hash:  hash,
		nodes: nodes,
		data:  data,
	}

	return tree, nil
}
```

Section 3

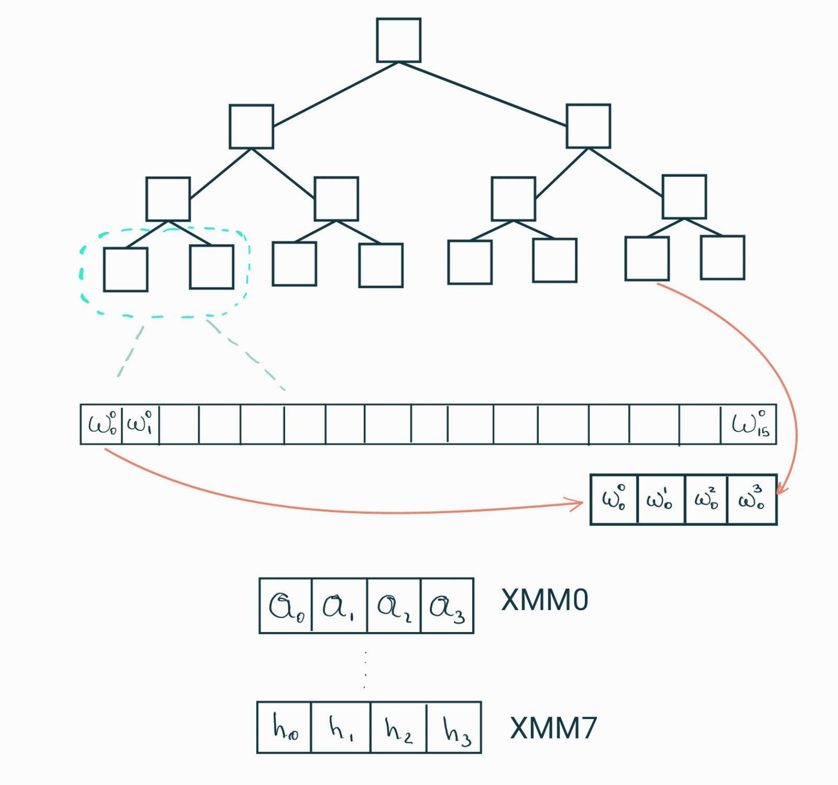
The right way to hash a Merkle Tree

The padding block is known, so we can hardcode the scheduled words $W_0, \dots, W_{63}$

~20%-30% gain.

```
0b100000000000000000000000000000000
00000000000000000000000000000000000
00000000000000000000000000000000000
00000000000000000000000000000000000
00000000000000000000000000000000000
00000000000000000000000000000000000
00000000000000000000000000000000000
00000000000000000000000000000000000
00000000000000000000000000000000000
00000000000000000000000000000000000
00000000000000000000000000000000000
00000000000000000000000000000000000
00000000000000000000000000000000000
00000000000000000000000000000000000
00000000000000000000000000000000000
000000000000000001000000
```

Vectorization



- AVX can hash 4 blocks at a time (128bit)
- AVX2 can hash 8 blocks at a time (256bit)
- AVX-512 can hash 16 blocks at a time
- AVX-1024…

- ARM NEON is faster than scalar hashing
- AVX-512 is faster than crypto extensions

Hasher signature

```
func hash(message []byte) [][32]byte

def hash(data:bytes) -> Sequence[Bytes32]

pub fn hash(input: &[u8]) -> Vec<[u8; HASH_LEN]>

void hash( unsigned char* out,
           const unsigned char* in,
           uint64_t count)
```

Thank you!

Potuz

Prysmaticlabs

potuz@prysmaticlabs.com

https://github.com/prysmaticlabs/hashtree

https://github.com/prysmaticlabs/gohashtree