Sparse Merkle Trees

Introducing the concept and benchmarking libraries available in Rust

Table of contents

Introduction

What is the problem

In decentralization where we rely on untrusted parties, exchanges requires **integrity** and **authenticity**, at a very **big scale**.

Introduction

What is the problem

In decentralization where we rely on untrusted parties, exchanges requires **integrity** and **authenticity**, at a very **big scale**.

> **Integrity**: The property that data or information have not been altered or destroyed.

Solved using a **hash function**, changing a single bit of data will change the hash.

Introduction

What is the problem

In decentralization where we rely on untrusted parties, exchanges requires **integrity** and **authenticity**, at a very **big scale**.

> **Integrity**: The property that data or information have not been altered or destroyed.

Solved using a **hash function**, changing a single bit of data will change the hash.

> **Authenticity**: The property that data originated from its purported source.

Solved using **cryptographic signatures** (RSA, ECDSA), generated using a *secret key* only the owner has, can be verified by anyone using the *public key* associated to the secret key.

Introduction

Let's hash everything ! ... No ?

Bitcoin: 2500 to 4600 transactions per block

Introduction

Let's hash everything ! ... No ?

Bitcoin: 2500 to 4600 transactions per block

Situation

Alice paid a pizza 10 BTC in "Hacker Pizza" using her smartphone, want to check if transaction accepted.
Hacker Pizza's WIFI **alters the data** she receives

Introduction

Let's hash everything ! ... No ?

Bitcoin: 2500 to 4600 transactions per block

Situation

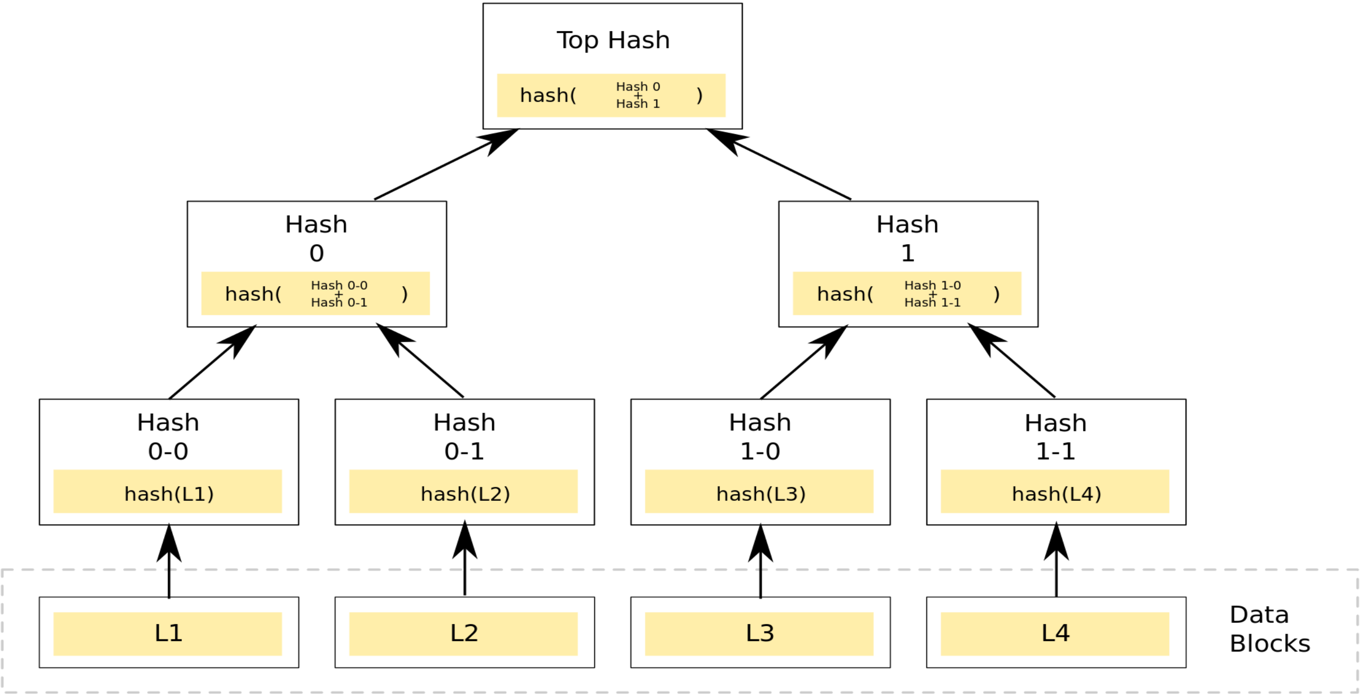
Alice paid a pizza 10 BTC in "Hacker Pizza" using her smartphone, want to check if transaction accepted.
Hacker Pizza's WIFI **alters the data** she receives

Only solution to protect herself

Hash all the transactions, compare with hash given in block header, then verify the block header's signature is correct.

What is a Merkle Tree

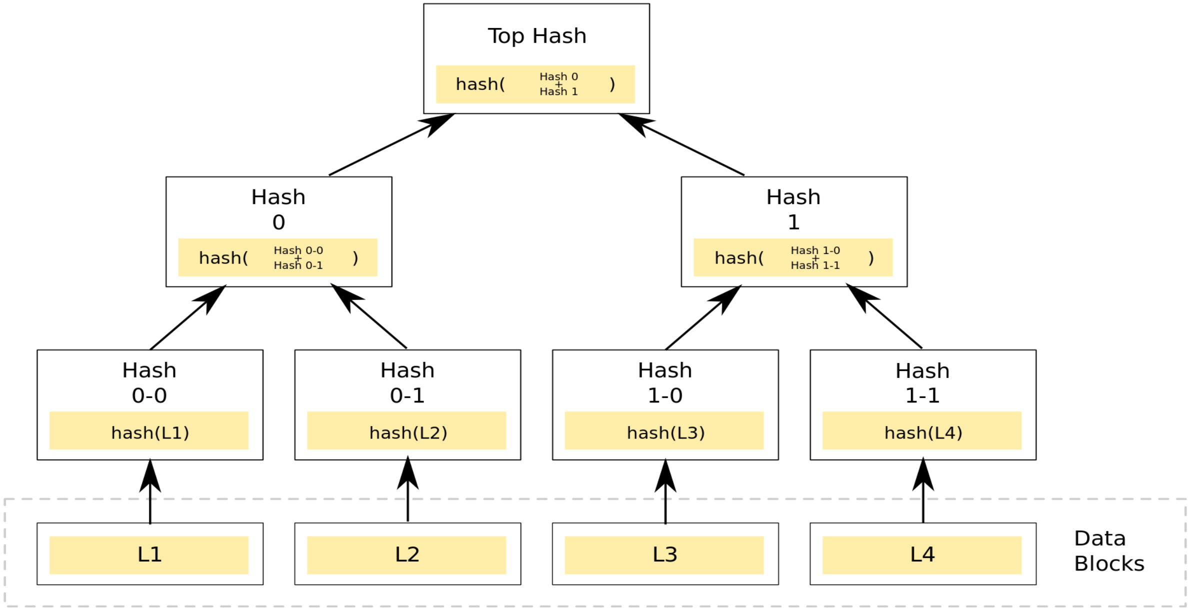
Hash large data, pieces by pieces, without compromises on integrity.



What is a Merkle Tree

Alice wants to verify the transaction L2

- Hash the transaction: `H0-1`
- Ask `H1` and `H0-0`, verify signatures
- `H0 = Hash(H0-0, H0-1)`
- `TopHash = Hash(H0, H1)`
- Compare `Top hash` to the one in block header
- Verify signature of the hash in block header



What is a Merkle Tree

> Merkle Proof: All the data needed to verify a leaf of the tree

For Alice's tx: `Hash(L2)` + `H0-0` + `H1` (concatenated)

What is a Merkle Tree

> Merkle Proof: All the data needed to verify a leaf of the tree

For Alice's tx: `Hash(L2)` + `H0-0` + `H1` (concatenated)

To compute hash of tree with 256 leaves, 8 hashes needed

What is a Merkle Tree

> Merkle Proof: All the data needed to verify a leaf of the tree

For Alice's tx: `Hash(L2)` + `H0-0` + `H1` (concatenated)

To compute hash of tree with 256 leaves, 8 hashes needed

Simplifies a `O(N)` operation to `O(log(N))`

What is a Merkle Tree

> Merkle Proof: All the data needed to verify a leaf of the tree

For Alice's tx: `Hash(L2)` + `H0-0` + `H1` (concatenated)

To compute hash of tree with 256 leaves, 8 hashes needed

Simplifies a `O(N)` operation to `O(log(N))`

Have hash of a modified datastore with N elements requires `O(log(N))` hashes operations.

Why we want it in Massa

Want to have a fingerprint of the Ledger
Check its integrity

Why we want it in Massa

Want to have a fingerprint of the Ledger
Check its integrity

Ledger of Massa: 1TB of data

Why we want it in Massa

Want to have a fingerprint of the Ledger
Check its integrity

Ledger of Massa: 1TB of data

Current implementation

Add new data to the ledger: `F' = F XOR Hash(new_data)`
Remove data off the ledger: `F' = F XOR Hash(rm_data)`

Why we want it in Massa

Want to have a fingerprint of the Ledger
Check its integrity

Ledger of Massa: 1TB of data

Current implementation

Add new data to the ledger: `F' = F XOR Hash(new_data)`
Remove data off the ledger: `F' = F XOR Hash(rm_data)`

Simple, fast, incremental

Why we want it in Massa

Want to have a fingerprint of the Ledger
Check its integrity

Ledger of Massa: 1TB of data

Current implementation

Add new data to the ledger: `F' = F XOR Hash(new_data)`
Remove data off the ledger: `F' = F XOR Hash(rm_data)`

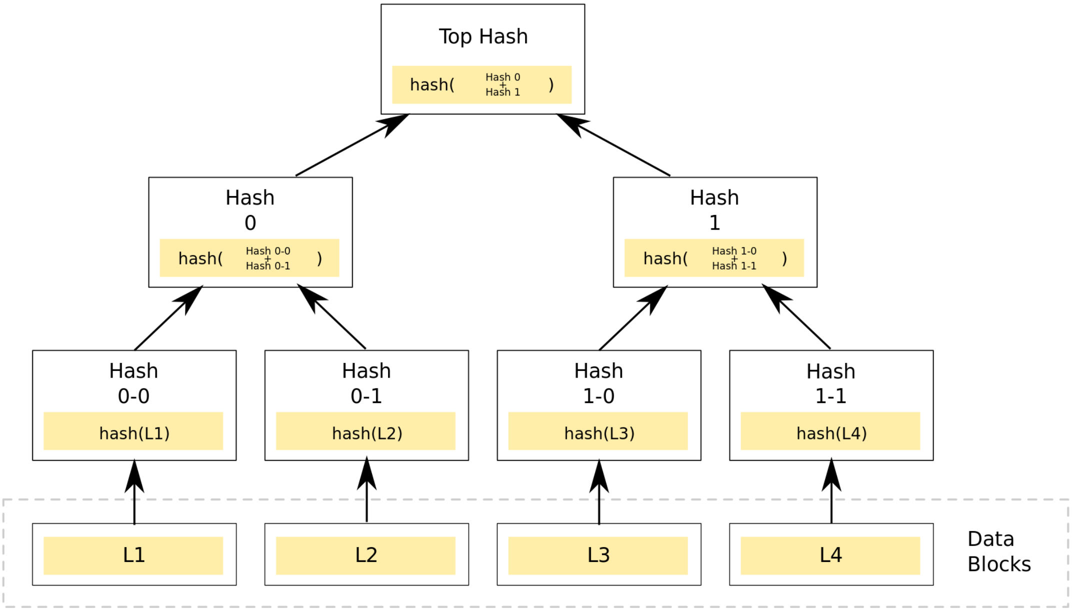
Simple, fast, incremental

Not a hash function, not suited for integrity checks
If `A XOR B = 0`, then `C XOR (A XOR B) = C`

Why we want it in Massa

So let's use Merkle Trees ! Why Sparse ?

What if we want to add a new data between L1 and L2 ?



Why Sparse Merkle Tree is what we need

Additions to Merkle Trees

Allows for `null` leaves (with `Hash(null)` a known constant)

Why Sparse Merkle Tree is what we need

Additions to Merkle Trees

Allows for `null` leaves (with `Hash(null)` a known constant)

Populate all the possible keys with a leaf of value `null`

Can know if a key is not present, go through the tree, check if value is `null`

Why Sparse Merkle Tree is what we need

Additions to Merkle Trees

Allows for `null` leaves (with `Hash(null)` a known constant)

Populate all the possible keys with a leaf of value `null`

Can know if a key is not present, go through the tree, check if value is `null`

Proof of **non-inclusion**

Why Sparse Merkle Tree is what we need

Proof of inclusion

Proof of non-inclusion

Integrity check over the whole data

No compromises

Why Sparse Merkle Tree is what we need

Proof of inclusion

Proof of non-inclusion

Integrity check over the whole data

No compromises

> Note: Incremental hash was also a different posibility, look at the discussion on Github to find more details on why SMT was chosen.

Why Sparse Merkle Tree is what we need

Proof of inclusion

Proof of non-inclusion

Integrity check over the whole data

No compromises

> Note: Incremental hash was also a different posibility, look at the discussion on Github to find more details on why SMT was chosen.
> Or ask to Varun, he explains it very well :-)

The frameworks for SMT in Rust

The frameworks for SMT in Rust

```rust
struct SparseMerkleTree<H: Hasher, D: Database> { ... }
```

The frameworks for SMT in Rust

```rust
struct SparseMerkleTree<H: Hasher, D: Database> { ... }
```

```rust
trait Database {
  fn get(...)
  fn put(...)
  fn remove(...)
}
```

The frameworks for SMT in Rust

```
struct SparseMerkleTree<H: Hasher, D: Database> { ... }
```

```
trait Database {
  fn get(...)
  fn put(...)
  fn remove(...)
}
```

```
trait Hasher {
  fn new(...)
  fn update(...)
  fn finalize(...)
}
```

The frameworks for SMT in Rust

Why benchmark ?

Huge tree with `null` values everywhere

The frameworks for SMT in Rust

Why benchmark ?

Huge tree with `null` values everywhere

Can be heavily optimized in **space** and in **computation**

The frameworks for SMT in Rust

Why benchmark ?

Huge tree with `null` values everywhere

Can be heavily optimized in **space** and in **computation**

Underlying database calls can also be optimized

The frameworks for SMT in Rust

Why benchmark ?

Huge tree with `null` values everywhere

Can be heavily optimized in **space** and in **computation**

Underlying database calls can also be optimized

Very implementation-dependant

The frameworks for SMT in Rust

Benchmarks

Implemented `Blake3Hasher`, `MemoryStorage`, `RockSdbStorage`

The frameworks for SMT in Rust

Benchmarks

Implemented `Blake3Hasher`, `MemoryStorage`, `RockSdbStorage`

Framework	Last updated	Stars on Github
Monotree	Dec. 2021	32
Sparse-Merkle-Tree	6 months ago	25
lsmtree	9 months ago	15

The frameworks for SMT in Rust

Benchmarks

Implemented `Blake3Hasher`, `MemoryStorage`, `RockSdbStorage`

Framework	Last updated	Stars on Github
Monotree	Dec. 2021	32
Sparse-Merkle-Tree	6 months ago	25
lsmtree	9 months ago	15

> `cw-merkle-tree` was ignored as it's too tied to CosmWasm smart contract framework

The frameworks for SMT in Rust

Benchmarks

On `MemoryStore` (storage in RAM)

```
monotree/memstore+blake3
  time:   [17.473 µs 17.619 µs 17.770 µs]

sparse-merkle-tree/memstore+blake3
  time:   [119.37 µs 120.63 µs 122.11 µs]

lsmtree/memstore+blake3
  time:   [25.587 µs 25.768 µs 25.952 µs]
```

The frameworks for SMT in Rust

Benchmarks

On `RocksDB` (storage on the disk)

```
monotree/rocksdb+blake3
  time:   [153.27 µs 155.79 µs 158.37 µs]

sparse-merkle-tree/rocksdb+blake3
  time:   [1.3083 ms 1.3135 ms 1.3190 ms]

lsmtree/rocksdb+blake3
  time:   [248.28 µs 249.85 µs 251.47 µs]
```

The frameworks for SMT in Rust

Read / Write operations benchmark

On `MemoryStore` (storage in RAM)

```
monotree/memstore+blake3/read
  time:   [2.7194 µs 2.7374 µs 2.7564 µs]

lsmtree/memstore+blake3/read
  time:   [174.66 ns 178.09 ns 181.79 ns]

monotree/memstore+blake3/write
  time:   [14.213 µs 14.318 µs 14.423 µs]

lsmtree/memstore+blake3/write
  time:   [24.849 µs 25.431 µs 26.049 µs]
```

The frameworks for SMT in Rust

Read / Write operations benchmark

On `RocksDB` (storage on the disk)

```
monotree/rocksdb+blake3/read
  time:   [10.370 µs 10.646 µs 10.938 µs]

lsmtree/rocksdb+blake3/read
  time:   [581.90 ns 609.57 ns 638.72 ns]

monotree/rocksdb+blake3/write
  time:   [150.39 µs 161.96 µs 172.92 µs]

lsmtree/rocksdb+blake3/write
  time:   [233.95 µs 239.78 µs 245.64 µs]
```

What framework to choose ?

I recommend `Monotree`

What framework to choose ?

I recommend `Monotree`

Optimization on database access, `N -> log2(N)`

What framework to choose ?

I recommend `Monotree`

Optimization on database access, `N -> log2(N)`

Very simple `Database` and `Hasher` traits

What framework to choose ?

I recommend `Monotree`

Optimization on database access, `N -> log2(N)`

Very simple `Database` and `Hasher` traits

Fully featured already

Simple but efficient

Can be maintained by our own means

Some links

Benchmark code

Article on a performance-oriented SMT implementation

How Merkle trees is used in Bitcoin

Github discussions about implementing SMT in Massa

Why use binary trees over trees with more children

Libra whitepaper, contains optimizations for SMT