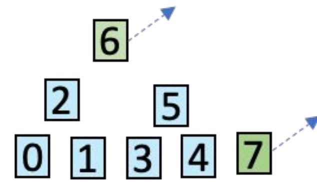
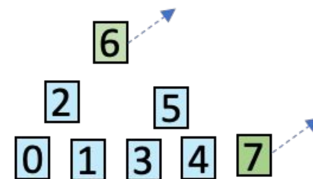


COSE Merkle Mountain Range Proofs

https://datatracker.ietf.org/doc/draft-bryce-cose-merkle-mountain-range-proofs/

Some context



Early last year I had a re-think of our transparency layer. Our wish list was:

* Fast appends

* Co-ordination free writers for HA & uptime

* Easy replication for auditors

* Simple operations for pruning of old data

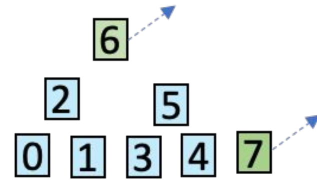
I picked a Merkle Mountain Range based log.

- Looked at loads of things. MMR's have everything on the left + some cool extras. But no standard ….
- I found > 5 implementations with various differences

When I saw COSE Receipts it look like a great fit

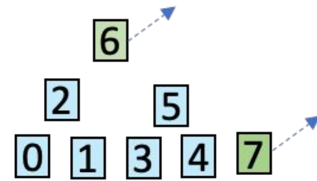
- A lot of the heavy lifting for interop is done already by COSE/CBOR encoding.
- The accommodations of protected and unprotected headers are specifically useful to how receipts are "pre signable" and "self serviceable" with this approach.

COSE Receipts ?



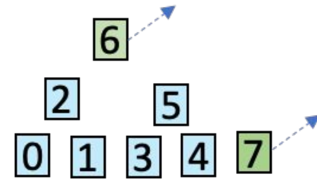
- A draft by Orie Steel
  https://datatracker.ietf.org/doc/draft-steele-cose-merkle-tree-proofs/
- Establishes a registry for encoding verifiable data structure proofs using COSE

What are Merkle Mountain Ranges ?



- The crop up in a few places with a few names: post order traversal tree, history trees, binary numeral trees.
- Cited in the crosby wallach paper and elsewhere.
- They have specific technical advantages:
  - *Fast appends and compatible with optimistic concurrency*
  - *linear storage layout*
  - *easy to prune*
  - *receipts can be pre-signed and "self serve"*
- For formal definitions treating the "peaks" as an accumulator see: *"Efficient Asynchronous Accumulators for Distributed PKI"* – *https://eprint.iacr.org/2015/718* *(Leonid Reyzin and Sophia Yakoubov)*

Receipt of inclusion




```
protected-header = {
  &(alg: 1) => int
  &(vds: 395) => 2
  * cose-label => cose-value
}
```

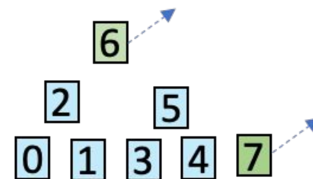
Payload: accumulator peak (detached)

```
unprotected-header = {
  &(vdp: 396) => verifiable-proofs
  * cose-label => cose-value
}
verifiable-proofs = {
  &(inclusion-proof: -1) => inclusion-proofs
}
inclusion-proofs = [ + inclusion-proof ]
```

```
inclusion-proof = bstr .cbor [

  ; zero based index of a tree node
  index: uint

  ; path proving the node's inclusion
  inclusion-path: [ + bstr ]
]
```

- Any node may have a receipt (including interiors)
- The node index + the node height + the length of the path identifies the peak
- Because receipts are rooted at peaks, we can pre-sign once
- We can attach the inclusion paths later for one or more nodes.
- The relying party can attach *offline* based on having a checkpoint and some log data.

Receipt of consistency




```
protected-header = {
  &(alg: 1) => int
  &(vds: 395) => 2
  * cose-label => cose-value
}
```

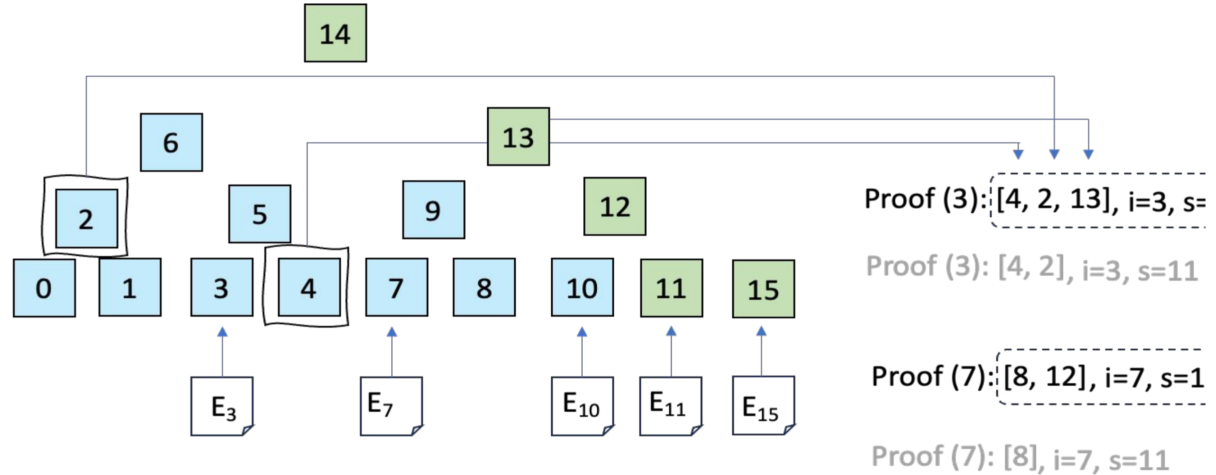
Payload: array of accumulator peaks (detached)

```
unprotected-header = {
  &(vdp: 396) => verifiable-proofs
  * cose-label => cose-value
}
verifiable-proofs = {
  &(consistency-proof: -2) => inclusion-proofs
}
consistency-proofs = [ + consistency-proof ]
```

```
consistency-path = [ * bstr ]

consistency-proof =  bstr .cbor [

  tree-size-1: uint

  tree-size-2: uint

  ; the inclusion path from each accumulator peak in tree-size-1
  consistency-paths: [ + consistency-path ]

  ; the additional peaks for accumulator at tree-size-2
  right-peaks: [ *bstr ]
]
```

- Log implementations can pre-sign receipts in the same operation as they generate their checkpoints
- As consistency proofs are just inclusion proofs, much of the implementation is shared.
- Including a series of consistency proofs allows for a “catchup” proof from an archived (possibly long term) checkpoint

A visualisation

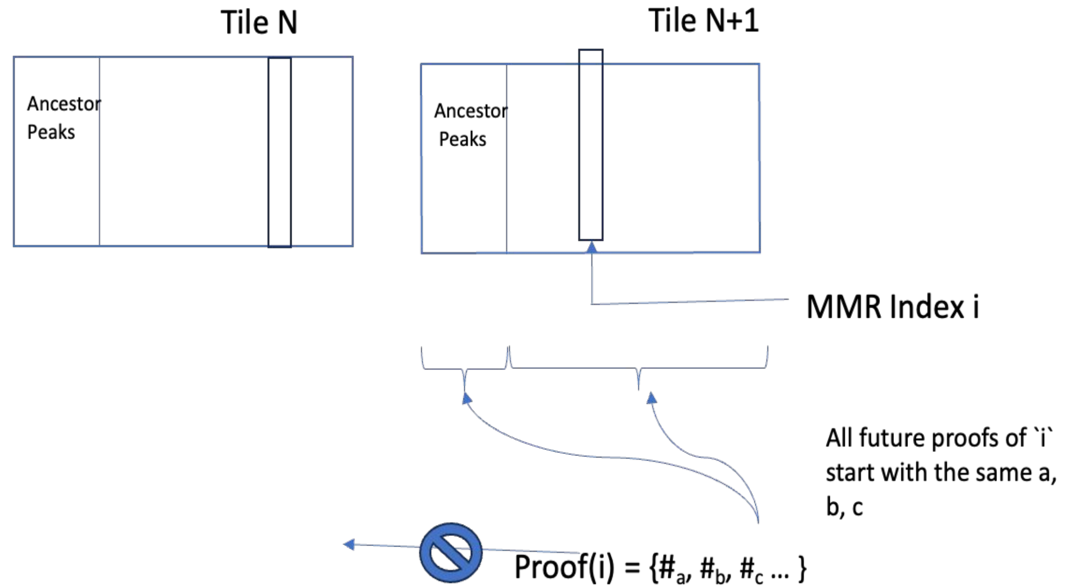


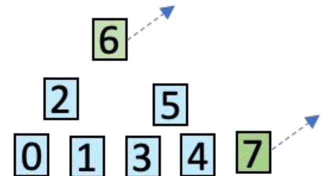
- The blue nodes are MMR(11). It has peaks 6, 9 and 10. Those are the accumulator
- The inclusion proofs for 0, 1, 3, 4 in MMR(11) all have 6 as the root
- Log checkpoints are always accumulator states (peak lists).
- The consistency proof is the inclusion proof for each old peak in the new accumulator (interiors can be proven)
- A single receipt for 0, 1, 3, 4 can be pre signed when the accumulator containing 6 is verified as consistent

- When the log grows to MMR (15) the updated witnesses MUST include the previous peaks as path elements else the log is inconsistent. And any receipt holder can prove this.
- The accumulator changes for each addition but the peaks in it change slower and slower. Hence “old accumulator compatibility” and “low update frequency” async strong accumulator.
- It is still “just a binary merkle tree”, lots of affordances are the same, same mental model.

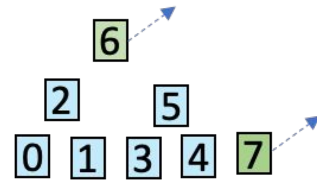
What about storage

- Storage is out of scope except for the “get” and “append” interface minimally defined.
- But it is very amenable to tiling in static “object storage” files. The linear organisation really helps here.
- The ancestor peaks are just the accumulator from the end of the last tile. Always < tree height. Which is max 64 in this draft



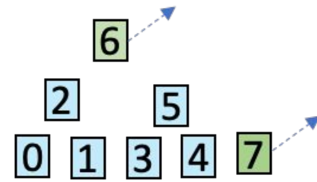


Working code ?

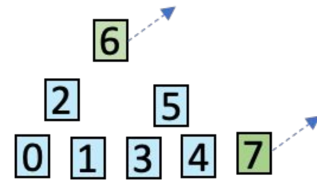


- > 5 implementations of MMR's
  - https://github.com/mimblewimble/grin/blob/master/doc/mmr.md
  - https://github.com/BeamMW/beam/blob/master/core/merkle.cpp
  - https://github.com/HerodotusDev/rust-accumulators
  - …
- Python, typescript and go lang for this draft
  - https://github.com/datatrails/go-datatrails-merklelog
  - https://github.com/robinbryce/mmriver-tiles-ts/blob/main/README.md
  - https://github.com/robinbryce/draft-bryce-cose-merkle-mountain-range-proofs/blob/main/algorithms.py (generates test vectors)

TODO’s



- If accepted
- Hash algorithm agility, only defined for SHA 256, but referenced implementations use BLAKE, pedersen, SHA3, Keccak …
- Test vectors - they are provided as ‘side content’ for now.
- Some “useful but not essential” algorithms may be worth adding back.



The MMRIVER draft defines a COSE Receipts VDS.

Questions