TBD
Internet-Draft
Intended status: Standards Track
Expires: 23 December 2025

R. Bryce
Datatrails

21 June 2025

COSE Receipts for MMRs
draft-bryce-cose-receipts-mmr-profile-00

Abstract

This document defines a new verifiable data structure profile for the COSE Receipts document [I-D.ietf-cose-merkle-tree-proofs] specifically for use with ledgers based on post-order traversal binary Merkle trees and which are designed for high throughput, ease of replication and compatibility with commodity cloud storage.

Post-order traversal binary Merkle trees, also known as history trees, are more commonly known as Merkle Mountain Ranges.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 23 December 2025.

Copyright Notice

Table of Contents

1. Introduction

A post ordered binary merkle tree is, logically, the unique series of perfect binary merkle trees required to commit its leaves.

Example,

```
   6
 2   5
0 1 3 4 7
```

This illustrates MMR(8), which is comprised of two perfect trees rooted at 6 and 7. 7 is the root of a tree comprised of a single element.

The peaks of the perfect trees form the accumulator.

The storage of a tree maintained in this way is addressed as a linear array, and additions to the tree are always appends.

Proving and verifying are defined in terms of the cryptographic asynchronous accumulator described by ReyzinYakoubov (https://eprint.iacr.org/2015/718.pdf). The technical advantages of post-order traversal binary Merkle trees are discussed in CrosbyWallachStorage (https://static.usenix.org/event/sec09/tech/full_papers/crosby.pdf) and PostOrderTlog (https://research.swtch.com/tlog#appendix_a).

2. Conventions and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

* A complete MMR(n) defines an mmr with n nodes where no equal height sibling trees exist.

* i shall be the index of any node, including leaf nodes, in the MMR

* g shall be the zero based height of a node in the tree.

* H(x) shall be the SHA-256 digest of any value x

* || shall mean concatenation of raw byte representations of the referenced values.

In this specification, all numbers are unsigned 64 bit integers. The maximum height of a single tree is 64 (which will have g=63 for its peak).

3. Description of the Verifiable Data Structure

This documents extends the verifiable data structure registry of [I-D.ietf-cose-merkle-tree-proofs] with the following value:

Name	Value	Description	Reference
MMR_SHA256	TBD_1 (requested assignment 3)	Linearly addressed, position committing, MMR implementations, such as the MMR ledger	This document

Table 1: Verifiable Data Structure Algorithms

4. Inclusion Proofs

The CBOR representation of an inclusion proof is

```
inclusion-proof = bstr .cbor [

  ; zero based index of a tree node
  index: uint

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

Note that the inclusion path for the index leads to a single permanent node in the tree. This node will initially be a peak in the accumulator, as the tree grows it will eventually be "buried" by a new peak.

4.1. inclusion_proof_path

inclusion_proof_path(i, c) is used to produce the verification paths for inclusion proofs and consistency proofs.

Given:

* c the index of the last node in any tree which contains i.

* i the index of the mmr node whose verification path is required.

And the methods:

* index_height (Section 9.1) which obtains the zero based height g of any node.

And the constraints:

* i <= c

We define inclusion_proof_path as

```
     def inclusion_proof_path(i, c):

       path = []

       g = index_height(i)

       while True:

         # The sibling of i is at i +/- 2^(g+1)
         siblingoffset = (2 << g)

         # If the index after i is higher, it is the left parent,
         # and i is the right sibling
         if index_height(i+1) > g:

           # The witness to the right sibling is offset behind i
           isibling = i - siblingoffset + 1

           # The parent of a right sibling is stored immediately
           # after
           i += 1
         else:

           # The witness to a left sibling is offset ahead of i
           isibling = i + siblingoffset - 1

           # The parent of a left sibling is stored immediately after
           # its right sibling
           i += siblingoffset

         # When the computed sibling exceeds the range of MMR(C+1),
         # we have completed the path
         if isibling > c:
             return path

         path.append(isibling)

         # Set g to the height of the next item in the path.
         g += 1
```

5. COSE Receipt of Inclusion

The cbor representation of an inclusion proof is:

```
protected-header-map = {
  &(alg: 1) => int
  &(vds: 395) => 3
  * cose-label => cose-value
}
```

* alg (label: 1): REQUIRED. Signature algorithm identifier. Value type: int.

* vds (label: 395): REQUIRED. verifiable data structure algorithm identifier. Value type: int.

The unprotected header for an inclusion proof signature is:

```
inclusion-proofs = [ + inclusion-proof ]

verifiable-proofs = {
  &(inclusion-proof: -1) => inclusion-proofs
}

unprotected-header-map = {
  &(vdp: 396) => verifiable-proofs
  * cose-label => cose-value
}
```

The payload of an inclusion proof signature is the tree peak committing to the nodes inclusion, or the node itself where the proof path is empty. The algorithm included_root (Section 5.2) obtains this value.

The payload MUST be detached. Detaching the payload forces verifiers to recompute the root from the inclusion proof, this protects against implementation errors where the signature is verified but the payload merkle root does not match the inclusion proof.

5.1. Verifying the Receipt of inclusion

The inclusion proof and signature are verified in order. First the verifiers applies the inclusion proof to a possible entry (set member) bytes. The result is the merkle root implied by the inclusion proof path for the candidate value. The COSE Sign1 payload MUST be set to this value. Second the verifier checks the signature of the COSE Sign1. If the resulting signature verifies, the Receipt has proved inclusion of the entry in the verifiable data structure. If the resulting signature does not verify, the signature may have been tampered with.

It is recommended that implementations return a single boolean result for Receipt verification operations, to reduce the chance of accepting a valid signature over an invalid inclusion proof.

As the proof must be processed prior to signature verification the implementation SHOULD check the lengths of the proof paths are appropriate for the provided tree sizes.

5.2. included_root

The algorithm included_root calculates the accumulator peak for the provided proof and node value.

Given:

* i is the index the nodeHash is to be shown at

* nodehash the value whose inclusion is to be shown

* proof is the path of sibling values committing i.

And the methods:

* index_height (Section 9.1) which obtains the zero based height g of any node.

* hash_pospair64 (Section 8.3.1) which applies H to the new node position and its children.

We define included_root as

```
    def included_root(i, nodehash, proof):

      root = nodehash

      g = index_height(i)

      for sibling in proof:

        # If the index after i is higher, it is the left parent,
        # and i is the right sibling

        if index_height(i + 1) > g:

          # The parent of a right sibling is stored immediately after

          i = i + 1

          # Set ‘root‘ to ‘H(i+1 || sibling || root)‘
          root = hash_pospair64(i + 1, sibling, root)
        else:

          # The parent of a left sibling is stored immediately after
          # its right sibling.

          i = i + (2 << g)

          # Set ‘root‘ to ‘H(i+1 || root || sibling)‘
          root = hash_pospair64(i + 1, root, sibling)

        # Set g to the height of the next item in the path.
        g = g + 1

      # If the path length was zero, the original nodehash is returned
      return root
```

6. Consistency Proof

A consistency proof shows that the accumulator, defined in ReyzinYakoubov (https://eprint.iacr.org/2015/718.pdf), for tree-size-1 is a prefix of the accumulator for tree-size-2.

The signature is over the complete accumulator for tree-size-2 obtained using the proof and the, supplied, possibly empty, list of right-peaks which complete the accumulator for tree-size-2.

The receipt of consistency is defined so that a chain of cumulative consistency proofs can be verified together.

The cbor representation of a consistency proof is:

```
consistency-path = [ * bstr ]

consistency-proof =  bstr .cbor [

  ; previous tree size
  tree-size-1: uint

  ; latest tree size
  tree-size-2: uint

  ; the inclusion path from each accumulator peak in
  ; tree-size-1 to its new peak in tree-size-2.
  consistency-paths: [ + consistency-path ]

  ; the additional peaks that
  ; complete the accumulator for tree-size-2,
  ; when appended to those produced by the consistency paths
  right-peaks: [ *bstr ]
]
```

6.1. consistency_proof_path

Produces the verification paths for inclusion of the peaks of tree-size-1 under the peaks of tree-size-2.

right-peaks are obtained by invoking peaks(tree-size-2 - 1), and discarding length(proofs) from the left.

Given:

* ifrom is the last index of tree-size-1

* ito is the last index of tree-size-2

And the methods:

* inclusion_proof_path (Section 4.1)

* Section 9.2

And the constraints:

* ifrom <= ito

We define consistency_proof_paths as

```
    def consistency_proof_paths(ifrom, ito):

      proof = []

      for i in peaks(ifrom):
        proof.append(inclusion_proof_path(i, ito))

      return proof
```

7. COSE Receipt of Consistency

The cbor representation of an inclusion proof is:

```
protected-header-map = {
  &(alg: 1) => int
  &(vds: 395) => 3
  * cose-label => cose-value
}
```

* alg (label: 1): REQUIRED. Signature algorithm identifier. Value type: int.

* vds (label: 395): REQUIRED. verifiable data structure algorithm identifier. Value type: int.

The unprotected header for an inclusion proof signature is:

```
consistency-proofs = [ + consistency-proof ]

verifiable-proofs = {
  &(consistency-proof: -2) => consistency-proof
}

unprotected-header-map = {
  &(vdp: 396) => verifiable-proofs
  * cose-label => cose-value
}
```

The payload MUST be detached. Detaching the payload forces verifiers to recompute the roots from the consistency proofs. This protects against implementation errors where the signature is verified but the payload is not genuinely produced by the included proof.

7.1. Verifying the Receipt of consistency

Verification accommodates verifying the result of a cumulative series of consistency proofs.

Perform the following for each consistency-proof in the list, verifying the signature with the output of the last.

1. Initialize current proof as the first consistency-proof.

2. Initialize accumulatorfrom to the peaks of tree-size-1 in the current proof.

3. Initialize ifrom to tree-size-1 - 1 from the current proof.

4. Initialize proofs to the consistency-paths from the current proof.

5. Apply the algorithm consistent_roots (Section 7.1.1)

6. Apply the peaks algorithm to obtain the accumulator for tree-size-2

7. From the peaks for tres-size-2, discard from the left the number of roots returned by consistent_roots.

8. Create the consistent accumulator by appending the remaining peaks to the consistent roots.

9. If there are no remaining proofs, use the consistent accumulator as the detached payload and verify the signature of the COSE Sign1.

It is recommended that implementations return a single boolean result for Receipt verification operations, to reduce the chance of accepting a valid signature over an invalid consistency proof.

As the proof must be processed prior to signature verification the implementation SHOULD check the lengths of the proof paths are appropriate for the provided tree sizes.

7.1.1. consistent_roots

consistent_roots returns the descending height ordered list of elements from the accumulator for the consistent future state.

Implementations MUST require that the number of peaks returned by Section 9.2(ifrom) equals the number of entries in accumulatorfrom.

Given:

* ifrom the last index in the complete MMR from which consistency was proven.

* accumulatorfrom the node values corresponding to the peaks of the accumulator for tree-size-1

* proofs the inclusion proofs for each node in accumulatorfrom for tree-size-2

And the methods:

* included_root (Section 5.2)

* Section 9.2

We define consistent_roots as

```
def consistent_roots(ifrom, accumulatorfrom, proofs):

    frompeaks = peaks(ifrom)

    # if length(frompeaks) != length(proofs) -> ERROR

    roots = []
    for i in range(len(accumulatorfrom)):
      root = included_root(
          frompeaks[i], accumulatorfrom[i], proofs[i])

      if roots and roots[-1] == root:
          continue
      roots.append(root)

    return roots
```

8. Appending a leaf

An algorithm for appending to a tree maintained in post order layout is provided. Implementation defined methods for interacting with storage are specified.

8.1. add_leaf_hash

When a new node is appended, if its height matches the height of its immediate predecessor, then the two equal height siblings MUST be merged. Merging is defined as the append of a new node which takes the adjacent peaks as its left and right children. This process MUST proceed until there are no more completable sub trees.

add_leaf_hash(f) adds the leaf hash value f to the tree.

Given:

* f the leaf value resulting from H(x) for the caller defined leaf value x

* db an interface supporting the Section 8.2.2 and Section 8.2.1 implementation defined storage methods.

And the methods:

* index_height (Section 9.1)

* Section 8.3.1

We define add_leaf_hash as

```
  def add_leaf_hash(db, f: bytes):

    # Set g to 0, the height of the leaf item f
    g = 0

    # Set i to the result of invoking Append(f)
    i = db.append(f)

    # If index_height(i) is greater than g (#looptarget)
    while index_height(i) > g:

      # Set ileft to the index of the left child of i,
      # which is i - 2^(g+1)

      ileft = i - (2 << g)

      # Set iright to the index of the the right child of i,
      # which is i - 1

      iright = i - 1

      # Set v to H(i + 1 || Get(ileft) || Get(iright))
      # Set i to the result of invoking Append(v)

      i = db.append(
        hash_pospair64(i+1, db.get(ileft), db.get(iright)))

      # Set g to the height of the new i, which is g + 1
      g += 1

    return i
```

8.2. Implementation defined storage methods

The following methods are assumed to be available to the implementation. Very minimal requirements are specified.

Informally, the storage must be array like and have no gaps.

8.2.1. Get

Reads the value from the tree at the supplied index.

The read MUST be consistent with any other calls to Append or Get within the same algorithm invocation.

Get MAY fail for transient reasons.

8.2.2. Append

Appends new node to storage and returns the index that will be occupied by the node provided to the next call to append.

The implementation MUST guarantee that the results of Append are immediately available to Get calls in the same invocation of the algorithm.

Append MUST return the node i identifying the node location which comes next.

The implementation MAY defer commitment to underlying persistent storage.

Append MAY fail for transient reasons.

8.3. Node values

Interior nodes in the MUST prefix the value provided to H(x) with pos.

The value v for any interior node MUST be H(pos || Get(LEFT_CHILD) || Get(RIGHT_CHILD))

The algorithm for leaf addition is provided the result of H(x) directly.

8.3.1. hash_pospair64

Returns H(pos || a || b), which is the value for the node identified by index pos - 1

Editors note: How this draft accommodates hash alg agility is tbd.

Given:

* pos the size of the MMR whose last node index is pos - 1

* a the first value to include in the hash after pos

* b the second value to include in the hash after pos

And the constraints:

* pos < 2^64

* a and b MUST be hashes produced by the appropriate hash alg.

We define hash_pospair64 as

```
  def hash_pospair64(pos, a, b):

    # Note: Hash algorithm agility is tbd, this example uses SHA-256
    h = hashlib.sha256()

    # Take the big endian representation of pos
    h.update(pos.to_bytes(8, byteorder="big", signed=False))
    h.update(a)
    h.update(b)
    return h.digest()
```

9. Essential supporting algorithms

9.1. index_height

index_height(i) returns the zero based height g of the node index i

Given:

* i the index of any mmr node.

We define index_height as

```
  def index_height(i) -> int:
    pos = i + 1
    while not all_ones(pos):
      pos = pos - most_sig_bit(pos) + 1

    return bit_length(pos) - 1
```

9.2. peaks

peaks(i) returns the peak indices for MMR(i+1), which is also its accumulator.

Assumes MMR(i+1) is complete, implementations can check for this condition by testing the height of i+1

Given:

* i the index of any mmr node.

We define peaks

```
  def peaks(i):
    peak = 0
    peaks = []
    s = i+1
    while s != 0:
      # find the highest peak size in the current MMR(s)
      highest_size = (1 << log2floor(s+1)) - 1
      peak = peak + highest_size
      peaks.append(peak-1)
      s -= highest_size

    return peaks
```

10. Security Considerations

See the security considerations section of:

* [RFC9053]

10.1. Detection of improper inclusion

A receipt of inclusion shows only that the element is included in the ledger. Defining whether that inclusion was legitimate, or in some way valid, is out of scope for this document.

10.2. Misbehaving Ledgers

A ledger can misbehave in several ways. Examples include the following: failing to incorporate a leaf entry in the MMR; presenting different, conflicting views of the MMR at different times and/or to different parties.

Detection of a failure to include items in the first place is out of scope for this document.

Having included an element, ledger implementations using this draft MUST use consistency proofs as the basis for proving entries are not moved, modified or excluded in future states of the MMR. Similarly, consistency proofs MUST be the basis for proving the unequivocal history of additions.

11. IANA Considerations

Editors note: Hash agility is desired. We can start with SHA-256. Two of the referenced implementations use BLAKE2b-256, We would like to add support for SHA3-256, SHA3-512, and possibly Keccak and Pedersen.

11.1. Additions to Existing Registries

Editors note: todo registry requests

11.2. New Registries

12. Normative References

[I-D.ietf-cose-merkle-tree-proofs]
Steele, O., Birkholz, H., Delignat-Lavaud, A., and C. Fournet, "COSE Receipts", Work in Progress, Internet-Draft, draft-ietf-cose-merkle-tree-proofs-14, 11 May 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-cose-merkle-tree-proofs-14>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

[RFC9053] Schaad, J., "CBOR Object Signing and Encryption (COSE): Initial Algorithms", RFC 9053, DOI 10.17487/RFC9053, August 2022, <https://www.rfc-editor.org/rfc/rfc9053>.

Appendix A. References

A.1. Informative References

* ReyzinYakoubov (https://eprint.iacr.org/2015/718.pdf)

* CrosbyWallach (https://static.usenix.org/event/sec09/tech/full_papers/crosby.pdf)

* CrosbyWallachStorage (https://static.usenix.org/event/sec09/tech/full_papers/crosby.pdf) 3.3 Storing the log on secondary storage

* PostOrderTlog (https://research.swtch.com/tlog#appendix_a)

* PeterTodd (https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2016-May/012715.html)

* KnuthTBT (https://www-cs-faculty.stanford.edu/~knuth/taocp.html) 2.3.1 Traversing Binary Trees

* BNT (https://eprint.iacr.org/2021/038.pdf)

Appendix B. Assumed bit primitives

B.1. log2floor

Returns the floor of log base 2 x

```
def log2floor(x):
  return x.bit_length() - 1
```

B.2. most_sig_bit

Returns the mask for the the most significant bit in pos

```
def most_sig_bit(pos) -> int:
  return 1 << (pos.bit_length() - 1)
```

The following primitives are assumed for working with bits as they commonly have library or hardware support.

B.3. bit_length

The minimum number of bits to represent pos. b011 would be 2, b010 would be 2, and b001 would be 1.

```
def bit_length(pos):
  return pos.bit_length()
```

B.4. all_ones

Tests if all bits, from the most significant that is set, are 1, b0111 would be true, b0101 would be false.

```
def all_ones(pos) -> bool:
  msb = most_sig_bit(pos)
  mask = (1 << (msb + 1)) - 1
  return pos == mask
```

B.5. ones_count

Count of set bits. For example ones_count(b101) is 2

B.6. trailing_zeros

```
(v & -v).bit_length() - 1
```

Appendix C. Acknowledgments

Author’s Address

Robin Bryce
Datatrails
Email: robinbryce@gmail.com