Compressing Digital Assets with Concurrent Merkle Trees

Jarry Xiao[1], Noah Gundotra[2], Austin Adams[3], Anatoly Yakovenko[4]

Abstract

New innovations in blockchain technology have spearheaded the creation of digital assets, whose custodial ownership cannot be censored nor revoked without explicit authorization from the owner. However, there are practical issues around the utility of these assets due to network speed and storage costs. For non-fungible digital assets to be truly ubiquitous, the marginal storage cost per unit must be as close to zero as possible. The natural solution is to store a compressed fingerprint of the digital asset data on the blockchain while maintaining the actual data with traditional database solutions. The off-chain data cannot be maliciously tampered with as it would be easy to verify that the on-chain fingerprint is mismatched. Additionally, it should always be possible to reconstruct the full uncompressed state of the world by processing the ledger sequentially. This paper attempts to provide a solution to data fingerprinting by storing hashes of digital asset metadata in the leaves of a Merkle tree. In order to make this process robust, it introduces a new primitive data structure that supports concurrent write requests to the same Merkle tree without failing due to proof collision.

[1]Engineer at Solana Labs: jarry@solana.com
[2]Engineer at Solana Labs: noah.gundotra@solana.com
[3]Lead Engineer at Metaplex Studios: austbot@metaplex.com
[4]Co-founder at Solana Labs: anatoly@solana.com

1 Merkle Trees

A Merkle tree is a data structure that encodes cryptographic information in the nodes of a tree. This paper will only discuss full binary Merkle trees. The binary Merkle tree of depth 3 in the diagram below has nodes labeled as X_i, where i represents the respective node's index in a level-order traversal of the binary tree.

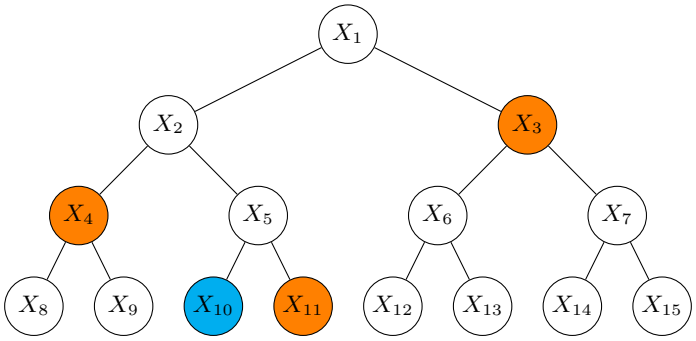

Figure 1: A depth 3 Merkle tree. The Merkle proof for the leaf node in cyan X_{10} would consist of the orange nodes $\{X_{11}, X_4, X_3\}$

Suppose the value stored at each node is a 256-bit (32 byte) string, which we can represent as a binary vector $X_i \in \{0,1\}^{256}$. We can define a *hash* function $H : \{0,1\}^{256} \times \{0,1\}^{256} \to \{0,1\}^{256}$, which takes 2 256-bit strings and outputs another 256-bit string. It is desirable for H to have unique outputs given unique inputs. In practice, the **SHA-256** algorithm is often used for H, but other choices exist.

A Merkle tree T will have the following properties:

1. T must be a full balanced binary tree of depth D
2. $X_i = H(X_{2i}, X_{2i+1})$ for all $i < 2^D$
3. Leaf nodes are the values X_i for $i \geq 2^D$. These nodes contain the fingerprinted data

These constraints allow the leaf data to be compressed into a single 256-bit string: the value of the *root* node at X_1. Given the root node X_1, it becomes possible to verify whether or not a leaf node belongs to that root. Suppose one were to claim that X_{10} belongs to the tree rooted at X_1. Based on the constraints on T, it would be sufficient to verify that $X_1 = H(H(X_4, H(X_{10}, X_{11})), X_3)$. The nodes $\{X_{11}, X_4, X_3\}$ are commonly referred to as the Merkle proof of X_{10}.

If the value of X_{10} were to change to X'_{10}, the path from X_{10} back to the root (X_1) would also need to be modified (see Figure 2).

$$X'_5 = H(X'_{10}, X_{11}) \tag{1}$$
$$X'_2 = H(X_4, X'_5) \tag{2}$$
$$X'_1 = H(X'_2, X_3) \tag{3}$$

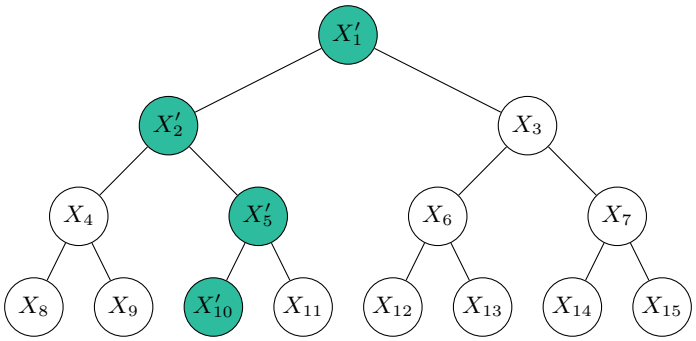

Figure 2: The green nodes represent the nodes (i.e. the path) that were modified after the value of X_{10} was updated to X'_{10}

2 Concurrent Leaf Replacement

Complications arise when considering concurrent writes (i.e. leaf replacements) to the same tree. Realistically, there can be multiple requests to modify leaf nodes of a tree that use the same root node as reference. Every time a leaf node is modified, all in-flight proofs that use the same root node will be invalid. A concrete example is illustrated below. Suppose that there is a request to modify the value of X_{10} to X_{10}^c and another request to modify the value of X_8 to X_8^o.

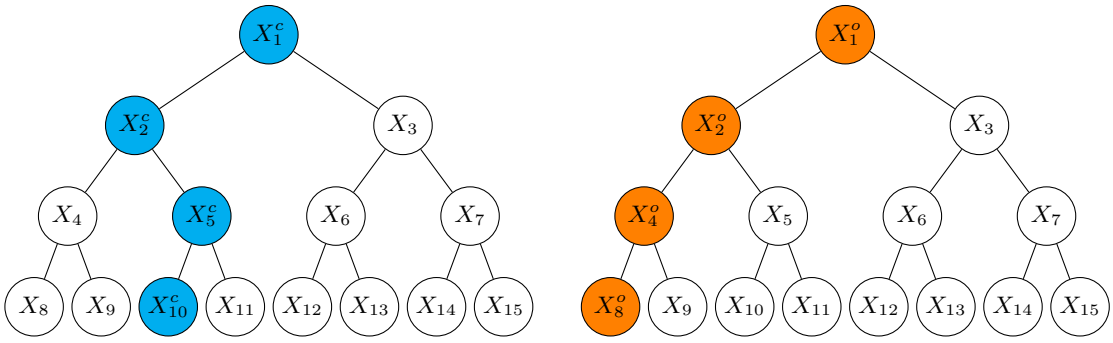

Figure 3: The cyan path and the orange path modify the same tree, but once one of the changes is locked in, the proof to the other change will be invalid.

Assume the cyan path is processed first using the proof $\{X_{11}, X_4, X_3\}$. The full path is updated and the new root node becomes X_1^c. The orange path is then processed with a proof $\{X_9, X_5, X_3\}$ to the root X_1. However, because the new root value is X_1^c, the proof is no longer valid because $X_1^c \neq H(H(H(X_8, X_9), X_5), X_3)$. For an arbitrary number of concurrent write requests that reference the same root, only 1 request would succeed while all others would fail.

If the program that controls write access to the tree kept a history of past updates, it could amend the proof of the orange path to match the current state of the root. The following would produce a valid hash to the root: $X_1^c = H(H(H(X_8, X_9), X_5^c), X_3)$. The cyan path modified one of the proof nodes of the orange path, which invalidated the original proof. If the program could modify the orange proof by replacing X_5 with X_5^c, both write requests could succeed while using proofs the same root.

To efficiently determine which proof nodes need to be switched out, one can use the **leaf indices** of the leaf node being modified and the leaf node in the historical path. The leaf index corresponds to the position of each leaf node in the tree ordered from left to right

$$\begin{bmatrix} \overset{0}{X_8} & \overset{1}{X_9} & \overset{2}{X_{10}} & \overset{3}{X_{11}} & \overset{4}{X_{12}} & \overset{5}{X_{13}} & \overset{6}{X_{14}} & \overset{7}{X_{15}} \end{bmatrix}$$

We can seamlessly convert from a **node index** i to a leaf index given the depth of the tree D with the equation $i - 2^D$ (e.g. the node index of X_{10} is 10 which translates to a leaf index of $10 - 2^3 = 2$)

The stored updated path (change log) and current proof can be listed in array form (the column corresponds to tree depth):

$$\text{change log} = \begin{bmatrix} \overset{3}{X_{10}^c} & \overset{2}{X_5^c} & \overset{1}{X_2^c} \end{bmatrix}$$

$$\text{proof} = \begin{bmatrix} \overset{3}{X_9} & \overset{2}{X_5} & \overset{1}{X_3} \end{bmatrix}$$

One can identify the intersection node of the two paths by taking an XOR of the leaf indices of the change log path and the new leaf (in their base 2 representation).

$$2 \otimes 0 = 010_2 \otimes 000_2 = 010_2$$

The number of leading zeros in the base 2 result corresponds to the depth (from the root) where the paths intersect. The level below the intersection point corresponds to the proof node that needs to be replaced. The above result yielded 1 leading zero, so the proof node at **depth 2** (this is 1 greater than the number of leading zeros because the critical node is 1 level below the intersection point) will be switched with the change log node at the same depth:

$$\text{change log} = \begin{bmatrix} \overset{3}{X_{10}^c} & \overset{2}{X_5^c} & \overset{1}{X_2^c} \end{bmatrix}$$

$$\text{proof} = \begin{bmatrix} \overset{3}{X_9} & \overset{2}{X_5^c} & \overset{1}{X_3} \end{bmatrix}$$

This trick works because the leaf index in base 2 encodes the path from the root to the leaf. Every 0 corresponds to a left traversal, and every 1 corresponds to a right traversal. If the XOR of two leaf indices has leading zeros, this indicates that the two paths from the root are identical up to the intersection index. The level at which the paths diverge is the same level where the original proof is corrupted.

It is not difficult to see that this approach generalizes to an arbitrary depth D and an arbitrary history size N. The high level idea is to search the historical change log buffer for the root that the supplied proof corresponds to. The proof is then updated from the matched root index to the latest change log by finding the path intersections. Finally, the updated proof is reevaluated against the current root and leaf node.

2.1 Algorithm

Below is pseudocode for the concurrent leaf replacement algorithm (the **FindCritbitIndex** function is unimplemented, but the procedure is described above).

Algorithm 1 Attempts to replace *leaf* with *newLeaf* at index i given a past *root* and nodes found in the *proof*

```
procedure ComputeParentNode(node, sibling, i, j)
    if i >> j & 1 == 0 then
        return H(node, sibling)
    end if
    return H(sibling, node)
end procedure
procedure ValidProof(root, leaf, i, proof)
    node ← leaf
    for j ← 0 to Length(proof) do
        node ← ComputeParentNode(node, proof[j], i, j)
    end for
    return node == root
end procedure
procedure UpdatePathToLeaf(tree, leaf, i, proof)
    changeLog ← NewChangeLog(i)
    node ← leaf
    for j ← 0 to Length(proof) do
        PushBack(changeLog.path, node)
        node ← ComputeParentNode(node, proof[j], i, j)
    end for
    PushFront(tree.changeLogs, changeLog)
    PushFront(tree.rootBuffer, node)
end procedure
procedure ReplaceLeaf(tree, root, leaf, newLeaf, i, proof)
    for j ← 0 to Length(tree.rootBuffer) do
        prevRoot ← tree.rootBuffer[j]
        if root == prevRoot then
            updatedLeaf ← leaf
            for k ← j to Length(tree.changeLogs) do
                changeLog ← tree.changeLogs[k]
                if i ≠ changeLog.index then
                    critbit ← FindCritbitIndex(i ⊗ changeLog.index, tree.depth)
                    proof[critbit] ← changeLog.path[critbit]
                else
                    updatedLeaf ← changeLog.path[0]
                end if
            end for
            if ValidProof(tree.rootBuffer[0], leaf, i, proof) and updatedLeaf == leaf then
                UpdatePathToLeaf(tree, newLeaf, i, updatedProof)
            end if
        end if
    end for
end procedure
```

3 Concurrent Appends

When a tree is initialized, all of its leaves will be empty. By convention, an empty leaf is defined as the 256-bit string with all zeros (0_{256}). Define an **empty tree** as a full binary Merkle tree with all empty leaves. Every node in an empty tree can be determined without any information. A natural way to populate an empty tree is to fill in the leaves from leaf index 0 to 2^D (left to right).

Define the **rightmost leaf** as the leaf node at the first empty leaf index. Like with leaf replacement, filling the rightmost leaf has the problem of running into proof conflicts in the case of concurrent requests. In fact, updating the rightmost leaf is equivalent to performing a replacement from the empty leaf to a non-empty leaf.

It is possible to support concurrent appends by keeping track of the rightmost proof, leaf, and index of the Merkle tree at all times. The implication is that the program will need to update the rightmost proof, leaf, and index every time any leaf node is modified. The previous section demonstrated that every 2 paths in the a tree will intersect at exactly 1 critical point, so the same algorithm can be applied between the updated path and the rightmost path to keep the rightmost proof and leaf up to date at all times. This approach has 2 main benefits:

1. Proofs are no long required to be passed into the program in order to verify whether the append operation is permitted. The diagram in Figure 4 explains the basic intuition behind this process.

2. Because no buffer is required to store the rightmost proof, an arbitrary number of concurrent requests (up to the size of the tree) are supported for appends.

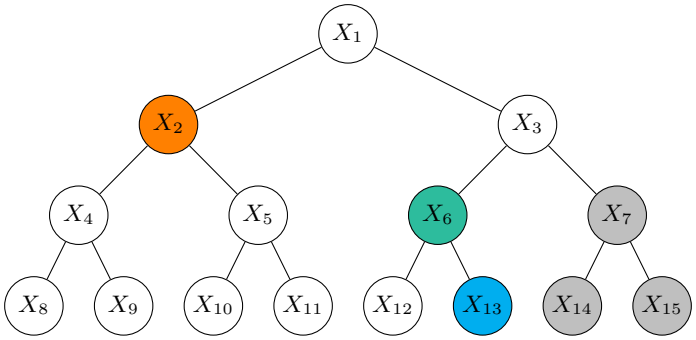

Figure 4: The cyan node at X_{13} is the current rightmost leaf. The gray nodes are nodes of empty sub-trees (the hashed values are deterministic). X_{14} would be updated on an append operation. The proof for X_{14} consists of $\{X_{15}, X_6, X_2\}$. X_{15} is empty, X_6 is the critical node between X_{13} and X_{14}, and X_2 is part of the current rightmost proof. Because the current rightmost proof and leaf are known, there is enough information in the program to fetch or compute the values of X_6 and X_2. Additionally, the value of X_{15} can be computed because it is in an empty sub-tree.

The appended leaf must be the leftmost leaf node of a sub-tree with all empty leaves. When the leaf index is odd, that sub-tree is just the leaf itself. In the above example, the empty sub-tree's root is at node X_7.

There exists a point where the empty sub-tree containing the appended leaf will intersect with the existing tree. We can find the intersection node by traversing the rightmost path to the correct level. Conveniently, this level is equivalent to D minus the number of trailing zeros in the leaf index (special logic is needed for index 0) minus 1. In the above example, the leaf index of X_{14} is $14 - 2^3 = 6 = 110_2$, which has 1 trailing 0. The intersection level is $3 - 1 - 1 = 1$. This corresponds to X_3 when comparing the paths to X_{13} and X_{14} at depth 1.

$$\text{current rightmost path} = \begin{bmatrix} \overset{3}{X_{13}} & \overset{2}{X_6} & \overset{1}{X_3} \end{bmatrix}$$

$$\text{new rightmost path} = \begin{bmatrix} \overset{3}{X_{14}} & \overset{2}{X_7} & \overset{1}{X_3} \end{bmatrix}$$

This indicates that the critical node at level 2 (this node is one level below the intersection point) of the rightmost path, X_6, will be included in the new proof for X_{14}. X_6 is derivable from the rightmost leaf X_{13} and the rightmost proof:

$$X_6 = H(X_{12}, X_{13})$$

Beyond the intersection node, the rest of the proof for X_{14} is equivalent to the proof for X_{13}, so those proof values do not need to be updated. This naturally breaks the algorithm for performing the rightmost append into 4 steps.

1. Compute the proof nodes from the new rightmost leaf that are in the empty sub-tree.
2. Compute the proof node from the critical node (below the intersections point) of the current rightmost leaf and the new rightmost leaf.
3. Keep all remaining proof nodes above the critical node.
4. Use the new rightmost proof to update the path to the root as well as the change log buffer. (It is possible to do this step while computing steps 1-3.)

3.1 Algorithm

Below is pseudocode for the concurrent append algorithm. Note that the previous algorithm for concurrent leaf replacement must also be modified to reflect updates to the rightmost proof, leaf, and index in order to support both features.

Algorithm 2 Appends $leaf$ to the tree while updating the rightmost proof

```
procedure EmptyNode(i)
    if i == 0 then
        return 0_256
    end if
    child ← EmptyNode(i − 1)
    return H(child, child)
end procedure
procedure Append(tree, leaf)
    node ← leaf
    rightmostProof ← tree.rightmostProof
    rightmostIndex ← tree.rightmostIndex
    rightmostLeaf ← tree.rightmostLeaf
    intersectionNode ← rightmostLeaf
    intersectionIndex ← TrailingZeros(rightmostIndex)
    changeLog ← NewChangeLog(rightmostIndex)
    for j ← 0 to tree.depth do
        PushBack(changeLog.path, node)
        if j < intersectionIndex then
            node ← H(node, EmptyNode(j))
            intersectionNode ← ComputeParentNode(intersectionNode, rightmostProof[j], rightmostIndex−1, j)
            rightmostProof[j] ← EmptyNode(j)
        else if j == intersectionIndex then
            node ← H(intersectionNode, node)
            rightmostProof[j] ← intersectionNode
        else
            node = ComputeParentNode(node, rightmostProof[j], rightmostIndex − 1, j)
        end if
    end for
    tree.rightmostIndex ← rightmostIndex + 1
    tree.rightmostLeaf ← leaf
    PushFront(tree.changeLogs, changeLog)
    PushFront(tree.rootBuffer, node)
end procedure
```

4 Compressing Digital Assets

The replace and append operations on a Merkle tree can be mapped to actions taken when interacting with digital assets. The creation (or "minting") of non-fungible assets can be represented by appending new non-empty leaves to a tree. Transferring, delegating, freezing, and destroying assets map to replacing existing non-empty leaf nodes.

Compressed Asset Operations		
Action	Tree Operation	Authority
Mint	Append	Mint Authority
Transfer	Replace Leaf	Owner + Delegate
Delegate	Replace Leaf	Owner
Burn	Replace Leaf	Owner + Delegate

Figure 5: A list of actions and corresponding tree operations for compressed assets.

The leaf node of the tree will contain a fingerprint (hash) of the associated metadata of the asset. Figure 6 shows a potential schema for hashing metadata.

Metadata Hash Seeds			
Seed	Type	Size (B)	Description
Owner	PublicKey	32	Public key of the asset owner
Delegate	PublicKey	32	Public key of the asset delegate
Name	String	8	Name of the asset
URI	String	256	Link to the asset metadata
Asset ID	UUID	16	Unique asset identifier
Creator	PublicKey	32	Creator of the asset (entitled to royalties)
Royalty Percent	Integer	4	Percentage of sale transferred to the creator

Figure 6: An example schema for an asset leaf hash.

To compress this data into a leaf, one would simply need to hash all of the seeds together. A smart contract can also validate necessary authorization around specific actions by verifying signatures and recomputing the seed hash in the protocol. Higher level contracts that compose with an underlying concurrent Merkle tree give this design a lot of flexibility for a wide range of applications beyond digital assets.

4.1 Compression Ratio

If this full metadata were to be stored, it would require 380 bytes of data per asset. While this is a relatively tiny amount of data, a distinguishing feature of blockchains is that the data cost per byte is relatively high. For the following diagrams, assume the cost to store 1 byte on-chain is \$0.0003.

Figure 7 shows the data size of a (minimal) concurrent Merkle tree of depth 24 and a change log buffer size of 1024. Unsurprisingly, almost all of the cost is in storing the change log buffer. This tree has a maximum capacity of $2^{24} = 16777216$ leaves. A full tree would have marginal asset cost of \$0.0000148 per leaf.

Compressed Merkle Tree Size	
Data Field	Size
Max Buffer Size	4B
Max Depth	4B
Buffer Index	8B
Buffer Size	8B
Change Logs	827KB
Rightmost Proof	808B

Figure 7: Size of an on-chain concurrent Merkle tree.

Figure 8 compares the cost of storing the 380 byte example assets to compressing those assets in the concurrent Merkle tree described in Figure 7.

Storage Costs					
# of assets	Raw Bytes	Raw Cost	Compressed Bytes	Compressed Cost	Cost Per Asset
1000	380KB	$114	828KB	$248.39	$0.25
10000	3.8MB	$1140	828KB	$248.39	$0.025
100000	38MB	$11400	828KB	$248.39	$0.0025
1000000	380MB	$114000	828KB	$248.39	$0.00025
10000000	3.8GB	$1140000	828KB	$248.39	$0.000025

Figure 8: Marginal cost of storing on-chain data.

5 Conclusion

The cost saving of compression can solve a lot of issues around scaling the usage of digital assets. Building in support for concurrent write access allows this solution to be scalable for users. Without concurrent write access, it would be difficult to reliably support any type direct modification to compressed assets.

It is important to make a distinction between parallel write access and concurrency. Because all write requests are modifying the same underlying state (i.e. the tree), there is still a single resource that is in contention. However, because that resource is stored in a shared database, it is likely that many requests are sent based on the same root node. Ensuring that these requests are all able to be fulfilled is an important design constraint to consider. The resource is still modified one request at a time, but each subsequent request is still able to succeed.

Future research and development can be made by implementing ways to verifiably seed non-empty trees. Right now it takes a linear amount of work to populate an empty tree, but it would make a lot of sense to support creating new trees where leaf nodes are already populated.

Progress on the reference implementation of the algorithms and smart contracts described in the paper are found at this link: https://github.com/jarry-xiao/candyland

Disclaimer

This paper is for general information purposes only. It does not constitute investment advice or a recommendation or solicitation to buy or sell any investment and should not be used in the evaluation of the merits of making any investment decision. It should not be relied upon for accounting, legal or tax advice or investment recommendations. This paper reflects current opinions of the authors and is not made on behalf of Solana Labs, Metaplex Studios, or their affiliates and does not necessarily reflect the opinions of Solana Labs, Metaplex Studios, or their affiliates or individuals associated with them. The opinions reflected herein are subject to change without being updated.