Compact Sparse Merkle Trees

Faraz Haider

October 6, 2018

Abstract

A Sparse Merkle tree is based on the idea of a complete Merkle tree of an intractable size. The assumption here is that as the size of the tree is intractable, there would only be a few leaf nodes with valid data blocks relative to the tree size, rendering the tree as sparse. We present a novel approach called *Minimum distance path algorithm* to simulate this Merkle tree of intractable size which gives us efficient space-time trade-offs. We provide the algorithms for insertion, deletion and (non -) membership proof for a leaf in this Compact Sparse Merkle tree.

1 Introduction

Verifiable Log [ELC15] is a data structure which provides an append only list of values, the integrity of which can be determined by a cryptographic proof for each value. These logs can be implemented using Binary Merkle trees [Mer87], and they have been successfully used to solve the problem of Certificate Transparency [LLK13]. These verifiable logs have to be public in order to be effective. There is one caveat though, as these logs are append only, if a certificate gets somehow compromised there is no way to remove it from the log.

Verifiable Maps [ELC15] is a data structure which helps in solving the above issue of removing/revoking this compromised certificate. It provides for insertion and removal of values and provides a cryptographic proof of membership or non-membership of each value. Verifiable Maps are based on an extension of Merkle Trees known as Sparse Merkle trees. A Sparse Merkle tree (SMT) is based on the idea of a complete Merkle tree of an intractable size. The assumption here is that as the size of the tree is intractable, there would only be a few leaf nodes with valid data blocks relative to the tree size, rendering the tree as sparse. An SMT provides efficient non-membership proofs which are central to certificate revocation.

We have devised a novel approach to construct a SMT and the resulting data structure is referred to as Compact Sparse Merkle Trees(C-SMT). SMTs can be used to create efficient Blockchains, Secure Key-Value stores, Secure IM etc.

Before delving into our approach we'll first discuss some preliminaries and previous approaches to SMTs.

2 Preliminaries

2.1 Merkle Trees

A hash tree or a Merkle tree is an authenticated data structure where every leaf node of the tree contains the cryptographic hash of a data block and every non leaf node contains the concatenated hashes of it's child nodes.

2.2 Audit path

A Merkle audit path for a leaf in a Merkle tree is the shortest list of additional nodes in the tree required to compute the root hash for that tree. In other words, the audit path consists of the list of missing nodes required to compute the nodes leading from a leaf to the root of the tree.

2.3 Membership Proof

If the root computed from the audit path matches the true root, then the audit path is a proof of membership for that leaf in the tree.

2.4 History Independence

A unique set of keys produce a deterministic root hash, regardless of the order in which keys have been inserted or removed.

3 Compact Sparse Merkle Trees

3.1 Earlier Proposals

Different approaches ha e been proposed in the past to represent/simulate an SMT.

Bauer [Bau04] proposed an explicit tree structure where all the non-empty attributes are elevated upwards through their ancestors.The elevation stops when the root of a subtree containing a single non-empty leaf is reached, and all descendants to such roots are discarded. The original SMT can be reconstructed by recording indices for the non-empty leaves in each subtree, but will require excessive amounts of memory unless they are evenly spread out.

Laurie and Kasper in their work on Certificate revocation transparency [LK12], a Sparse Merkle tree is simulated on a collection of keys.It also utilizes the concept of empty hashes, and a non-membership proof a key is given by providing an audit path which leads to an empty hash value. Non-membership proofs are inefficient to calculate in this approach as hash-values are re-calculated on every request.

Their proposal is improved upon in Efficient Sparse Merkle Trees by Dahlberg [DPP16], the crux of their idea is that they use caching mechanisms to speed up proof generation and insertion. Due to this caching, the hash-values aren't explicitly calculated again on every request, they can be looked up through a cache.

In the following section we introduce a novel approach which creates an explicit tree structure without utilizing empty hashes. The minimum distance path algorithm utilized for insertion of a value forms the tree in such a way that it is inherently sorted.Due to this feature of the tree, we get rid of empty hashes and utilize the window approach for non-membership proofs wherein the closest pair of values which bound the given value form the non-membership proof. Insertion and deletion and (non-) membership proofs all are done in logarithmic time.

3.2 Our Approach

The idea is to place every unique key in its correct subtree. This would ensure history independence. In this approach we are not going to simulate a Sparse Merkle Tree, we are going to create it, albeit a very compact one. This C-SMT contains only the keys inserted till now. There are no empty hashes utilized, and we get the root hash in constant time.

To achieve this C-SMT, we need to augment the tree nodes to contain a parameter called max-key. So for every non-leaf node, it's two children will have max-key values representing the maximum key in their respective subtree. An incoming key to be inserted in this SMT would get put in the subtree for which it's binary distance is closest. And it will recursively go down the tree following this approach until it reaches a leaf node. The key gets inserted at this level and the hashes and max-key values are adjusted as we recurse back. The path which the key follows down the tree is called *minimum distance path* and we call this approach for inserting a key as *Minimum distance path algorithm*

- **Max-Key**: Every non-leaf node contains the maximum key under its subtrees called max-key.
- **Distance**: Distance is defined as the binary separation between two keys. It is calculated by taking bitwise XOR and subsequently calculating the log. The distance parameter helps in determining which subtree a key is closer too.

Listing 1: Binary distance between two keys

```
def distance(x, y) do
   result = bxor(x, y)
   :math.log2(result)
end
```

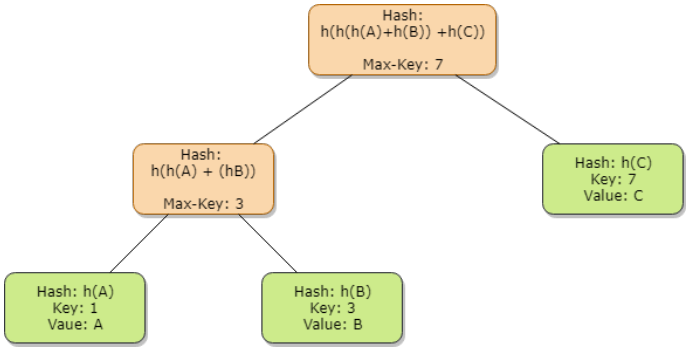

Figure 1: Compact sparse Merkle tree

4 Tree Operations

The algorithms below have been provided in a pseudo-code format. A few functional programming constructs have also been used to make the pseudo-code look succinct.

4.1 Inserting a key

The insert algorithm recursively walks down the tree along the *minimum distance path* until it reaches a leaf node. The key to be inserted is made the left or right sibling depending upon whether the current leaf node key is greater or smaller than the key to be inserted. On recursing back, on the way up all the hashes and max-key values along the minimum distance path are readjusted.

Listing 2: Inserting a key

```
def insert(root, k, v) do
  left = root.left
  right = root.right

  l_dist = distance(k, left.key)
  r_dist = distance(k, right.key)

  # checks if the key to be inserted falls in the left subtree or the
      right subtree
  cond do
    l_dist == r_dist ->
      new_leaf = make_node(k, v)

      min_key = min(left.key, right.key)

      if k < min_key do
        # make new leaf as left child at the new level
        make_node(new_leaf, root)
      else
        # make new leaf as right child at the new level
        make_node(root, new_leaf)
      end


    l_dist < r_dist ->
```

```
      # Going towards left subtree
      left = insert(tree, left, k, v)
      make_node(left, right)

    l_dist > r_dist ->
      # Going towards right subtree
      right = insert(tree, right, k, v)
      make_node(left, right)
  end
end

def insert(leaf, k, v) do
  new_leaf = make_node(k, v)

  cond do
    k == key ->
      raise "key exists"

    k > key ->
      # new key will be right child
      make_node(tree, leaf, new_leaf)

    k < key ->
      # new key will be left child
      make_node(tree, new_leaf, leaf)
  end
end
```

4.2 (Non-) Membership proof for a key

The membership proof part of this algorithm is similar to the insert algorithm mentioned above. The algorithm recursively walks down a minimum distance path for the key. If we successfully reach the leaf node, the membership proof is given by recursing back and adding the sibling of the current node to the proof.

If we are not able to reach the leaf node or at leaf node, the key does not match with the leaf key, then the non-membership proof part of the algorithm takes over. The crux of this algorithm is to supply proofs of the closest two keys which bound the key for which non-membership proof is to be given.

Listing 3: Membership proof for a key

```
  def membership_proof(root, k) do

      #  root has no sibling or direction so nil is used
      result = membership_proof(nil, nil, root, k)

      case result do
          # membership proof case
          # when the key is present in the tree
          [{_, _} | _] ->
              [{value, hash} | proof] = List.reverse(result)
              %{key: k, value: value, hash: hash, proof: proof}

          # edge case 1 for non-membership proof
          # When the key is greater than the largest element in the
              tree
          [key, :MINRS] -> [membership_proof(tree, key), nil]

          # edge case 2 for non-membership proof
          # when the key is smaller than the smallest element in the
              tree
          [:MAXLS, key] -> [nil, membership_proof(tree, key)]
```

```
        # when a key is bounded by two keys in case of non-
            membership proof
        [key1, key2] ->
            [membership_proof(tree, key1),
            membership_proof(tree,key2)]
    end
end

def membership_proof(sibling, direction, leaf, k) do
    # key is present
    if key == k do
        [{sibling.hash, reverse(direction)},
        {leaf.value, leaf.hash}]
    else
        # Find the non membership proof otherwise
        non_membership_proof(k, key, direction, sibling)
    end
end

def membership_proof(sibling, direction, root, k) do
    left = root.left
    right = root.right

    l_dist = distance(k, left.key)
    r_dist = distance(k, right.key)

    cond do
        l_dist == r_dist ->
            # Find the non membership proof otherwise
            non_membership_proof({k, root.key, direction, sibling})

        l_dist < r_dist ->
            # Going towards left child
            result = membership_proof(right, "L", left, k)

            case { result, direction } do
                # membership proof case
                {[{_, _} | _], _} ->
                    [{sibling.hash, reverse(direction)} | result]
                {[key, :MINRS],"L"} ->
                    [key, min_in_subtree(sibling)]
                {[:MAXLS, key], "R"} ->
                    [max_in_subtree(sibling), key]
                _ -> result
            end

        l_dist > r_dist ->
            # Going towards right child
            result = membership_proof(left, "R", right, k)

            case { result, direction } do
                # membership proof case
                {[{_, _} | _], _} ->
                    [{sibling.hash, reverse(direction)} | result]
                {[key, :MINRS],"L"} ->
                    [key, min_in_subtree(tree, sibling)]
                {[:MAXLS, key], "R"} ->
                    [max_in_subtree(sibling), key]
                _ -> result
            end
    end
end
```

```
  def non_membership_proof(k, key, direction, sibling) do
      case [k > key, direction] do
          [true, "L"] ->  [key, min_in_subtree(sibling)]
          [true, "R"] ->  [key, :MINRS]
          [false, "L"] -> [:MAXLS, key]
          [false, "R"] -> [max_in_subtree(sibling), key]
      end
  end

  def min_in_subtree(root) do
      min_in_subtree(root.left)
  end

  def min_in_subtree(leaf) do
      leaf.key
  end

  def max_in_subtree(root) do
      root.key
  end

  def reverse(direction) do
      case direction do
          "R" -> "L"
          "L" -> "R"
      end
  end
```

4.3 Deleting a key

The delete algorithm recursively walks down the tree until it reaches the leaf node with the key to delete. On recursing back, it attaches the sibling of the deleted node with the parent's sibling and the tree hashes get updated.

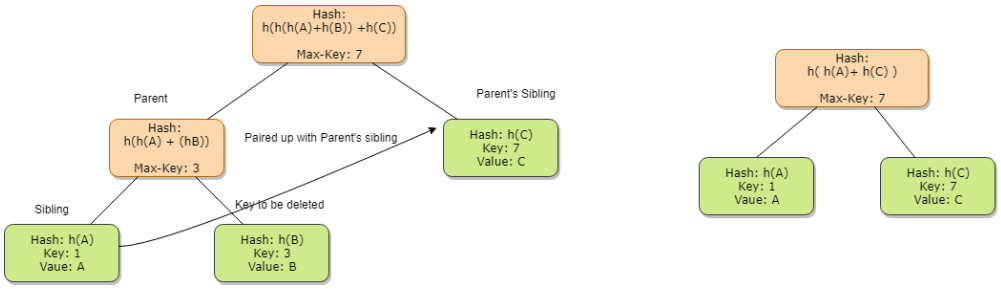

Figure 2: Deleting a key from SMT

Listing 4: Deleting a key

```
def delete(root, k) do
  left = root.left
  right = root.right

  if check_for_leaf(left, right, k) do
    if left.key == k do
      # deletes the target key
      delete_node(left)
      right
    else
      # deletes the target key
      delete_node(right)
```

```
      left
    end

  else
    l_dist = distance(k, left.key)
    r_dist = distance(k, right.key)

    cond do
      l_dist == r_dist ->
        raise "key does not exist"

      l_dist < r_dist ->
        # Going towards left subtree
        left = delete(left, k)
        make_node(left, right)

      l_dist > r_dist ->
        # Going towards right subtree
        right = delete(right, k)
        make_node(tree, left, right)
    end
  end
end

def check_for_leaf(left, right, k) do
  (left.size == 1 && left.key == k) ||
  (right.size == 1 && right.key == k)
end
```

5 Properties

As in the case of Bauer [Bau04], we assume that our hash function behaves as an ideal hash function, an assumption in cryptography called Random Oracle Model. Under this assumption the following properties are implied.

5.1 Structure

The SMT is nearly balanced. If there are n entries in the tree, we can reach to any particular entry in $\log_2(n)$ steps.

5.2 Space

Space for the SMT is bounded by twice the number of keys in the tree. n keys are stored at n leaves and each pair of leaf nodes have a parent.

$$\begin{aligned} &\sum_{i=0}^{\lceil \log_2(n) \rceil} 2^i \\ &= 2^{\lceil \log_2(n) \rceil + 1} - 1 \\ &< 2 \cdot n \end{aligned}$$

5.3 Max Proof Size

As the SMT is nearly balanced the max proof size would be $2 \cdot 256 \cdot \log_2(n)$ bits if SHA256 is used as the hashing algorithm.

5.4 Differences from other approaches

We get an explicit tree structure which is inherently sorted due to the minimum distance path algorithm utilized. The advantages of this approach are that we can have an explicit tree structure without compromising on space as the the max space required is roughly $2 \cdot n$. We need not have a cache as insertions are also possible in logarithmic time. The proof size does not contain the empty hashes so it becomes compact and is only $log_2(n)$ hashes long.

References

[Bau04] Matthias Bauer. Proofs of zero knowledge. *CoRR*, cs.CR/0406058, 2004.

[DPP16] Rasmus Dahlberg, Tobias Pulls, and Roel Peeters. Efficient sparse merkle trees: Caching strategies and secure (non-)membership proofs. 2016.

[ELC15] Adam Eijdenberg, Ben Laurie, and Al Cutter. Verifiable data structures. *Google Research*, 2015.

[LK12] Ben Laurie and Emilia Kasper. Revocation transparency. *Google Research*, 2012.

[LLK13] Ben Laurie, Adam Langley, and Emilia Käsper. Certificate transparency. *ACM Queue*, 12:10–19, 2013.

[Mer87] Ralph C Merkle. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*, pages 369–378. Springer, 1987.