Juicebox Merkle-Radix Tree

Revision 1 — June 29, 2023

Simon Fell
Juicebox Systems, Inc

Diego Ongaro
Juicebox Systems, Inc

Abstract

Juicebox uses a combined Merkle and radix tree to allow Hardware Security Modules (HSMs) to securely store and access a much larger amount of data than what is available in local storage. The tree combines a binary Merkle tree with the path compression approach used by radix trees. The tree's depth depends on the number of records stored rather than the length of the key, which improves performance. The tree supports split and merge operations to scale the number of HSMs up and down elastically. The tree nodes are stored in a non-transactional storage system to reduce costs, and reads are done in parallel with ongoing writes for high performance. The tree enables a system that is practical, cost-effective, and scalable.

Contents

1 Introduction

Juicebox has implemented a secure and scalable online service that uses Hardware Security Modules (HSMs) to serve requests on sensitive user data. This service implements the Juicebox Protocol[1] to allow users to store secrets and later recover them using a short PIN. The service uses a novel *Merkle-Radix Tree* to store sensitive data outside the HSMs and scale up to billions of secrets, while maintaining the HSM's security properties.

We use HSMs that are tamper-resistant, have persistent memory, and are programmable with custom code. They aim to prevent both online and physical attackers from accessing HSM state or rolling back that state to earlier versions. The Juicebox Protocol requires a secret to self-destruct after a fixed number of attempts, so it is critical that the state cannot be rolled back.

Building a scalable service using HSMs is challenging because of their limited hardware capabilities. HSMs on the market today have little persistent memory, so storing a large dataset entirely within the HSMs would not be practical. On the other hand, storing sensitive data in a traditional database would negate the benefits of the HSMs. Even if the records were stored in encrypted form, an HSM could not reliably distinguish an old version of a record from the latest version, thereby allowing an adversary to roll back the state.

Juicebox uses its Merkle-radix tree to allow the service to scale to billions of secret records, while maintaining the HSM's security properties. The records are organized into trees, and the tree nodes are stored in an external, untrusted storage system. The HSMs verify during each request that they are operating on the latest version of each record. The tree design and Juicebox's implementation support the following features:

- HSMs with limited capacity can efficiently serve requests on very large trees (with billions of records), without sacrificing security. The HSMs maintain confidentiality of the data and verify its integrity and freshness before operating on it.
- The performance of tree operations is logarithmic in the number of records in the tree. This is a significant improvement over other approaches with performance proportional to the length of the lookup key.
- The overall dataset can be dynamically partitioned across any number of trees, and repartitioning the trees to scale up and down is fast and cheap.
- Reads from each tree are fully parallelizable, even while the tree is undergoing writes. HSMs can process requests to mutate a tree continuously, without waiting for the results to reach storage.
- The tree nodes are stored on an untrusted, non-transactional storage system and cached freely, thereby reducing storage costs. Even within a single tree read or write operation, requests to the storage for the entire path is done concurrently, further improving performance.

While this paper focuses on the tree itself, Juicebox uses the Authenticated Consensus Protocol[2] to recover from hardware failures. The ability to authenticate an entire tree with a single hash allows the consensus protocol to reach agreement on just a small amount of metadata as the tree evolves, making operations simpler and more efficient.

2 Overview

In this paper, we assume a collection of independent HSMs are all equally trusted to manage sensitive data. Each HSM has been provisioned with a copy of an encryption key that is used to encrypt and decrypt records stored in the trees. The HSMs have some persistent memory but with limited capacity. The HSMs process and respond to incoming requests; they do not issue requests or communicate with each other directly.

Outside of the trust boundary of the HSMs, the service runs on commodity hardware and leverages cloud computing. See Figure 1 for an overview. There are one or more processes called *agents* that:

- issue requests to the HSMs,
- issue requests to a storage system (which stores the tree nodes), and
- issue requests to other agents (for replication and repartitioning).

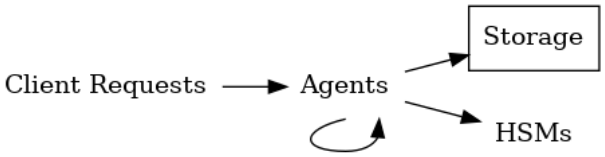


Figure 1: An overview of the system components. The HSMs are the only trusted components.

The security of the data depends solely on the HSMs functioning correctly. It does not depend on the agents, the storage system, or the network functioning correctly. However, these components must be available and must produce the correct data for the overall service to be available. For example, a single-bit error in the storage of a tree node would make that node and its subtree unusable.

A typical operation begins with an external client issuing a request that needs a record from a tree. The request identifies the record's key, which maps onto a particular partition and tree. The client's request is routed to an agent. The agent looks up the record in the tree in the storage system and produces a *proof*. The agent sends this proof, along with the client's request, to the HSM that's currently responsible for this tree. The HSM validates the proof, consults its *tree overlay* for recent updates, and then executes the request. If the request mutates the tree, the HSM returns a set of new nodes to write and existing nodes to delete, and the agent updates the external store.

The rest of the paper is organized as follows:

- Section 3 describes the tree data structure and its basic operations.
- Section 4 describes the format of the proofs that the agents produce when reading the tree. HSMs validate the proofs and mutate the trees using only the information contained in the proofs.
- Section 5 describes the tree split and merge operations. These are used when scaling the cluster up and down to repartition the data across a dynamic number of trees.
- Section 6 describes the tree overlay data structure that each HSM maintains. The overlay improves performance by allowing agents to read from storage while ongoing mutations are taking place, and allowing agents to submit multiple requests to mutate the tree concurrently to the HSM.
- Section 7 describes how agents organize data in the storage system for efficient access on cheap, non-transactional storage.
- Section 8 discusses important security considerations.

3 Tree Structure

A Merkle tree[3] stores records in leaf nodes and cascades node hashes up the tree to a single root hash. While Merkle trees have a variety of use cases, we use them to offload storage of data from HSMs to an untrusted system. By remembering the root hash, an HSM can later verify that its data hasn't been tampered with. The structure of the tree allows the HSM to verify that a particular record is correct using just the path of nodes up to the root.

In a binary Merkle tree, there is one level of the tree for each bit of the key. A tree using 64-bit keys will be 64 nodes deep. Each level of the tree requires a hash calculation to be performed when verifying or mutating the tree, so a tree that is 64 nodes deep requires 64 hash calculations to verify the integrity of a record. Designs that would benefit from larger keys have to be tempered by the cost of the additional hash calculations.

In a radix tree[4], each branch of the tree contains a substring of the key. The length of the substring depends on how many other keys share some prefix of the key. The size and shape of the tree depends on the number of keys rather than the length of the key. The tree is not balanced; its structure will depend on the distribution of the keys used.

Juicebox uses a combination Merkle and radix tree to allow the HSMs to securely offload their storage to an external, untrusted system. The tree combines a binary Merkle tree with the path compression approach used by radix trees. This results in a tree whose depth depends on the number of records stored rather than the size of the key. This allows for larger keys, making it easier to map other identifiers into keys. For a 256-bit key (as we use) this results in an order of magnitude fewer hashes to calculate and less data to move. This savings is significant when using HSMs, which typically have much less performance than today's commodity hardware.

Like a regular Merkle tree, the Merkle-radix tree consists of leaf nodes that store data and interior nodes that form the hash tree. Each interior node has a hash based on the child hashes and key path. These hashes eventually roll up to the root node with a final single hash value that captures the state of the entire tree. Like a radix tree, the edges between nodes include a substring of the key, and the concatenation of these substrings on a given path forms the key to the leaf. Also like a radix tree, the tree is not balanced; the shape and depth of the tree depend on the distribution of the keys used.

Figure 2 shows a small Merkle-radix tree with 4 leaves using 8-bit keys. (All the examples use 8-bit keys and small hashes for ease of reading.) The branch paths leading to a leaf are concatenated to make the key. For example, the path to the leaf with the value `C` starts at the root and takes the left branch. The branch has a path of `0000`, so our key so far is `0000`. The next branch is to the right with a path of `10`, so now our key is `000010`. Finally, the last branch is also to the right with a path of `11`; we've reached the leaf and the key is `00001011`.

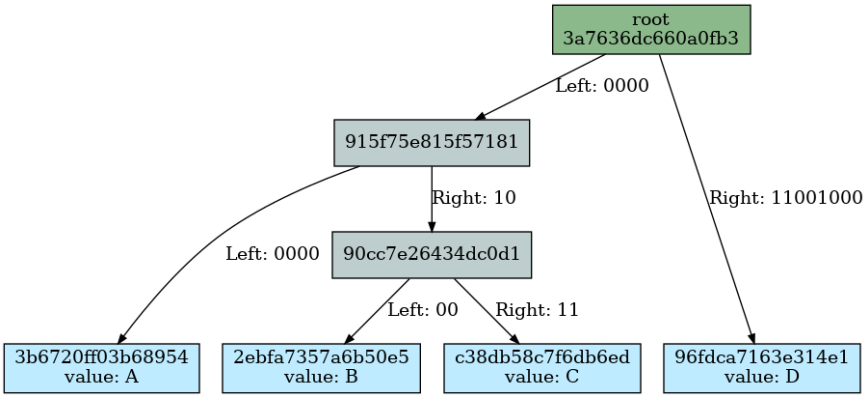


Figure 2: A Merkle-radix tree with 8-bit keys and 4 leaves. The root node is colored green, other interior nodes are gray, and leaves are shown in blue. The left and right labels on the arrows indicate the path for that branch in binary.

The concatenation of the path at each branch forms the keys. The tree in Figure 2 contains four keys, as shown in Table 1.

Steps	Key	Value
root ⟶ left ⟶ left	`00000000`	`A`
root ⟶ left ⟶ right ⟶ left	`00001000`	`B`
root ⟶ left ⟶ right ⟶ right	`00001011`	`C`
root ⟶ right	`11001000`	`D`

Table 1: The keys obtained by traversing the branches of the tree in Figure 2.

The tree is constructed with the following invariants:

- Only the root node may have empty child branches.
- The left path on an interior node always begins with a `0` bit.
- The right path on an interior node always begins with a `1` bit.
- Every branch encodes a non-empty substring of the key.

A tree with a specific set of keys will have the same structure regardless of the order in which the keys were added.

We designed the tree to store keys of a fixed length. This assumption simplifies the tree's design and implementation. To keep the tree well-balanced, we use the Blake2s-256[5] hash function to generate pseudorandom 256-bit keys from other identifiers.

3.1 Leaves

The hash of a leaf node is calculated as follows, where `leaf.key` is its location in the tree:

```
def hash_leaf(leaf):
    leaf_value = encrypt_record(leaf.record)
    return hash(concat('leaf', leaf.key, be64(len(leaf_value)), leaf_value))
```

`hash` is a cryptographically secure hash function. We use Blake2s-256[5], which performs well on 32-bit CPUs (as used by some HSMs).

`be64` encodes an unsigned 64-bit number in big-endian format.

To preserve confidentiality, data is encrypted at the leaf nodes of the tree. The encrypted `leaf_value` is built up from the application-level record as follows:

```
def encrypt_record(record):
    nonce = random(24)
    ciphertext = encrypt(leaf_encryption_key, record, nonce)
    return concat(ciphertext, nonce)
```

The `random` function returns bytes from a cryptographically-secure pseudorandom number generator. The `encrypt` function performs authenticated encryption with the given nonce. The `nonce` must be large enough that randomly generated nonces do not repeat. We use XChaCha20-Poly1305[6] with a 192-bit nonce. The encryption key (`leaf_encryption_key`) is the same for all leaves in the tree.

3.2 Interior Nodes

All non-leaf nodes, including the root node, are considered interior nodes. Interior nodes contain a left and right branch. The branch consists of the child node's hash and a key path containing one or more bits. The path is the substring of the key that leads from this node to the child node. The left branch always has a path that starts with bit `0`, and the right branch always has a path that starts with bit `1`. Only the root node may have empty branches.

The hash of an interior node is based on the hash and path of its two branches. The path used is just the substring of the key relative to the parent node, not the entire prefix of the key to reach the node. To calculate the hash, the path first needs to be converted to bytes. Every eight path bits are encoded into a byte, with the first path bit being the most significant bit in the byte. If the length of the path is not a multiple of eight, the final bits in the last byte are padded with `0`. For example, the path `0100` is encoded into a byte as value `01000000`. The path `010011111001` is encoded into two bytes with values `01001111` and `10010000`. The `b.num_path_bits` is a 16-bit unsigned integer in big-endian format. The lengths of the encoded branches are encoded as a single byte.

```
def branch(b):
    return concat(b.num_path_bits, b.path_bytes, b.child_hash)
def hash_interior(node):
    b_left = branch(node.left)
    b_right = branch(node.right)
    return hash(concat('interior', len(b_left), len(b_right), b_left, b_right))
```

The hash of a root node has two differences. First, to support partitioning (see Section 5), it includes additional information about the key range that the tree contains. Second, the left and right branches of the root node may be empty. For the purpose of hashing, empty branches are considered to have a zero-length path and a child hash of all zero bytes. Assuming a 256-bit hash function, this results in 34 bytes with value `0`.

```
def hash_root(node):
    b_left = branch(node.left) if node.left else [0] * 34
    b_right = branch(node.right) if node.right else [0] * 34
    return hash(concat(
        'root',
        partition_start, partition_end,
        len(b_left), len(b_right), b_left, b_right))
```

3.3 Empty Tree

Every tree starts as an empty tree, which consists of just the root node with empty left and right branches. Its hash will depend on the partition (key range) the tree is configured to store. See Section 5 for more information on partitioning.

3.4 Insert and Update

This section describes how to insert and update keys in an existing tree. While it would be possible to distinguish these operations, they are closely related. The implementation and this section combine insert and update into a single operation.

The following steps set a new value in the tree for a new or existing key:

1. First, traverse the tree from the root based on the key until either reaching the existing leaf or reaching the interior node where the leaf should be connected.
2. There are three cases to handle to connect the new leaf:
    - If there's an existing leaf for the key, update its parent to reference the new leaf.
    - If the branch from the root node for the key is empty, update the root node to branch to the new leaf.
    - Otherwise, the bottommost interior node currently branches to another key. Create a new interior node that connects the existing child and the new leaf. Update the branch to reference this new interior node.
3. Work up from the new leaf to the root re-calculating the node hashes.

Figure 3 through Figure 6 show the process of going from an empty tree to a tree with 4 leaves, by inserting 4 keys. This sequence illustrates the interesting cases in the algorithm above. (An update of an existing key is not shown, as it would only cause new hashes to propagate up to the root.)

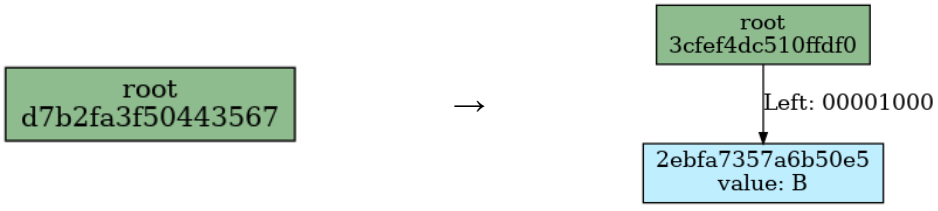


Figure 3: The value `B` is inserted for key `00001000`. The root node is updated with a branch to the new leaf.

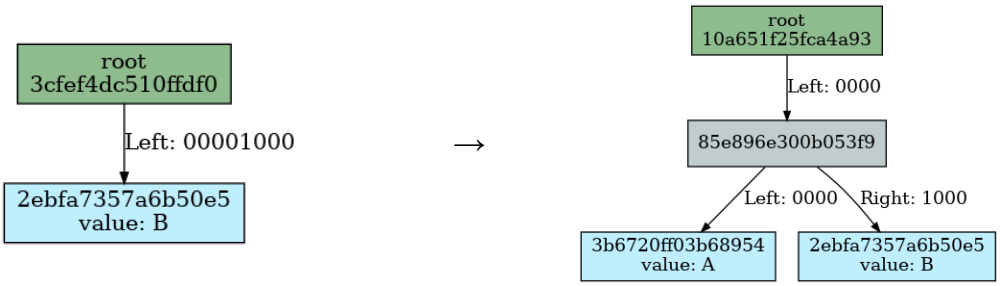


Figure 4: The value `A` is inserted for key `00000000`. The root node is the only interior node on the path. A new child interior node is added, connecting to the existing child and the new leaf.

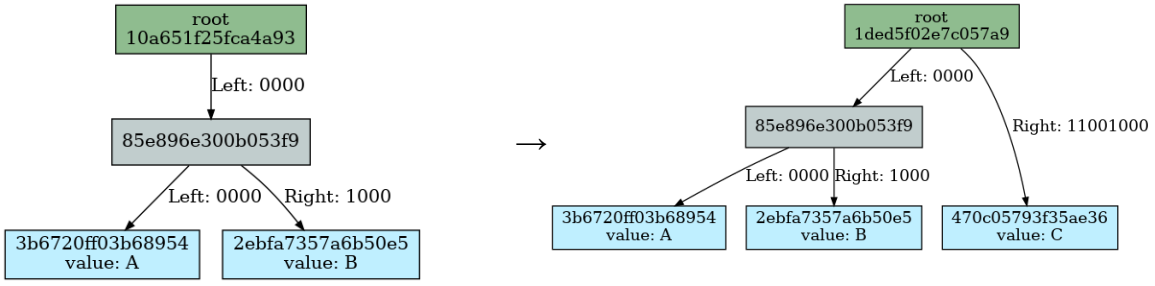


Figure 5: The value `C` is inserted for key `11001000`. The root node has an empty branch for this key. It is updated to connect to the new leaf.

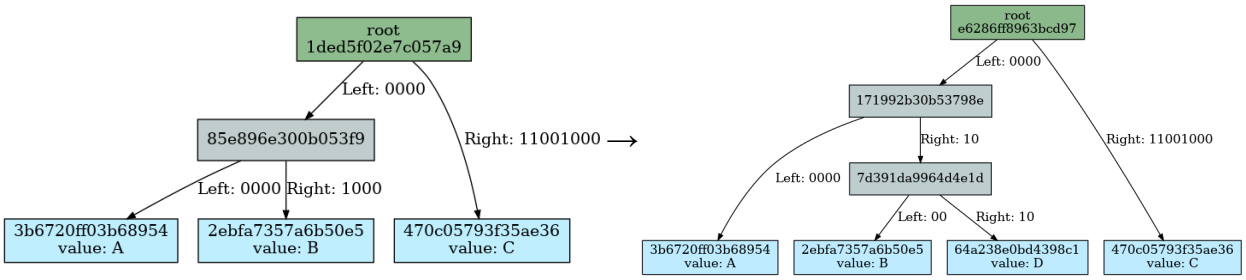


Figure 6: The value `D` is inserted for key `00001010`. The interior node with hash `85e`... is the bottommost interior node on the path. A new child interior node is added, connecting to the existing leaf (with value `B`) and the new leaf.

3.5 Delete

The tree does not currently support deleting leaves. Our initial use case did not require this operation, since we wanted to preserve a portion of the record as an audit log in the event of a "deletion". Implementing delete correctly would require some modifications to the tree design. We recommend using a soft-delete approach instead (updating the record to mark it as obsolete).

Conceptually, to fully remove a leaf, one could perform the inverse of an insert operation. However, this would be problematic with the exact way we have defined the node hashes and storage operations. Removing the leaf can cause its ancestor node hashes to revert to their former values. Node hashes are used to identify nodes in the storage system, so this can cause a catastrophic ambiguity: a delayed delete of an ancestor node hash in the storage system could remove a newly resurrected and necessary node.

4 Proofs

In our model, the HSMs do not have direct access to the tree storage. An HSM receives a request to read or update/insert a key, along with the tree nodes from the root following the path to that key. This path is called a *proof*. The HSMs verify that each proof is internally consistent and that its root hash is trusted before processing the corresponding request. They then process the request, without any further information from the tree storage. The HSMs execute the insert and update operations described in Section 3.4 using proofs.

This section describes the two types of proofs: inclusion proofs (for keys that exist in the tree) and non-inclusion proofs (for keys that do not exist in the tree). The two types of proofs are similar.

4.1 Inclusion Proofs

A proof that a specific leaf value exists for a key can be constructed from the path of nodes from the root to the leaf. It includes the leaf value. Figure 7 shows an example.

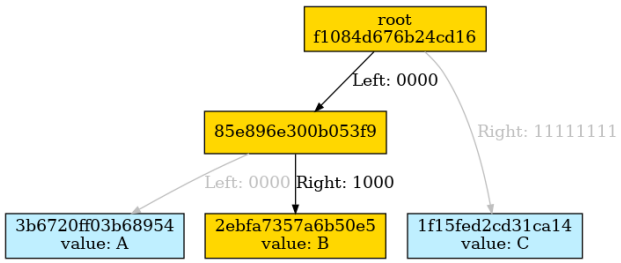


Figure 7: The yellow highlights show an inclusion proof for the key `00001000`.

The HSM checks an inclusion proof by verifying:

- That the root hash is trusted and current;
- The leaf hash;
- The interior node hashes, working up the path from the leaf to the root; and
- That the key described by the path from the root to the leaf matches the expected key.

4.2 Non-Inclusion Proofs

A proof that a specific key does not exist can be constructed and verified in a similar manner. The proof contains the path of interior nodes from the root that should would to the key, if it existed. It does not contain a leaf. Figure 8 shows an example.

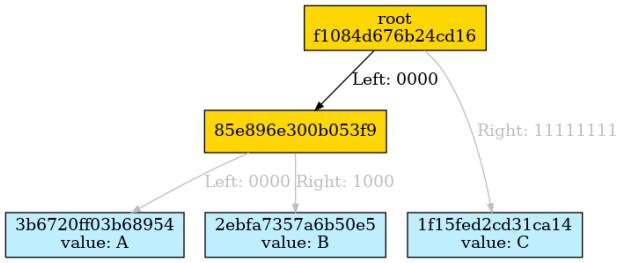


Figure 8: The yellow highlights show a non-inclusion proof for the key `00000001`.

The HSM checks a non-inclusion proof by verifying:

- That the root hash is trusted and current;
- The interior node hashes, working up to the root;
- That the key described by the path matches a prefix of the expected key; and
- That the terminal branches can not lead to the key because either:
    - they lead to a different key, or
    - they are empty (which should only occur for the root node).

Attempts to claim that a leaf does not exist by truncating the interior node path can be detected by the last verification step, which will see that there is a potential path to the leaf that does not lead to a different key.

As the key path is encoded into the interior node hashes, attempts to alter the path will be detected as a hash mismatch.

5 Partitioning

Updates to a single Merkle tree cannot scale to multiple HSMs, since every change to the tree affects at least the root node. For higher performance than one tree and one HSM can offer, the overall service needs to update multiple trees in parallel.

A simple approach would be to pre-partition the space into a fixed number of trees from the beginning, when the service has no data. Each tree would be assigned a fixed portion of the key-space and would be initialized with an empty root node. However, this approach is inflexible, as it would be impossible to create new partitions to scale up.

Juicebox's Merkle-radix tree supports dynamic repartitioning, so the number of trees can scale up and down. A tree *split* operation converts one tree into two trees with adjacent key-spaces. A tree *merge* operation is the inverse; it converts two trees with adjacent key-spaces into one tree. A series of split and merge operations can be used to scale clusters up and down and rebalance load.

Juicebox's approach supports fully elastic partitioning, allowing a cluster to grow or shrink by one HSM at a time. The key-space is partitioned into arbitrary ranges of keys, where each tree "owns" one contiguous range. We previously considered a simpler approach of only splitting trees in two at the root node, but that would restrict cluster sizes to powers of two. For example, to go past 4 HSMs, you would need 8 HSMs, then 16 HSMs, etc. We felt the restrictions could be both challenging and expensive in practical deployments. The approach described in this section is more complex, but it allows for much more elasticity and operational flexibility.

The split and merge operations are cheap, so they can be done frequently with minimal service impact. Only up to $tree\ depth * 2$ nodes need to be read and updated to split or merge arbitrary partitions.

5.1 Tree Split

Split takes a single tree and produces two adjacent trees. To split the tree, we need to choose a *split key*, which is any key inside the current partition where we desire to split the tree. All keys less than the split key will end up in the left tree. All keys equal to or larger than the split key will end up in the right tree.

The algorithm for the split is:[1]

1. Walk down the tree following the path for the split key. Stop when both branches out of the node are on the same side of the key, or upon reaching a leaf. Go back up one level, this is the split node.
2. Split this node into two, connecting one child branch to each.
3. Work back up to the root node, splitting each parent node.
    - One side of the split will only have a single child. Which side of the split that is matches the direction that was traversed from the parent.
4. At this point, there are two distinct trees. However, there may be interior nodes with only a single child. These must be pruned away.
    - For the node that has a single child, take its child branch path and append that to the parent branch's path. Update the parent branch's child hash with the child branch's hash.

[1]Implementation note: Steps 2, 3, and 4 can be done in a single pass, and the current implementation does this.

Let's walk through the example step by step. Figure 9 shows an initial tree to be split at key `00001011` (a key that is not present in the tree). The node with hash `e1a`… (highlighted in yellow) is the first node on the key path where both children are on the same side of the split key. The keys for the node's two children are `00001000` and `00001010`, which are both less than the split key of `00001011`.

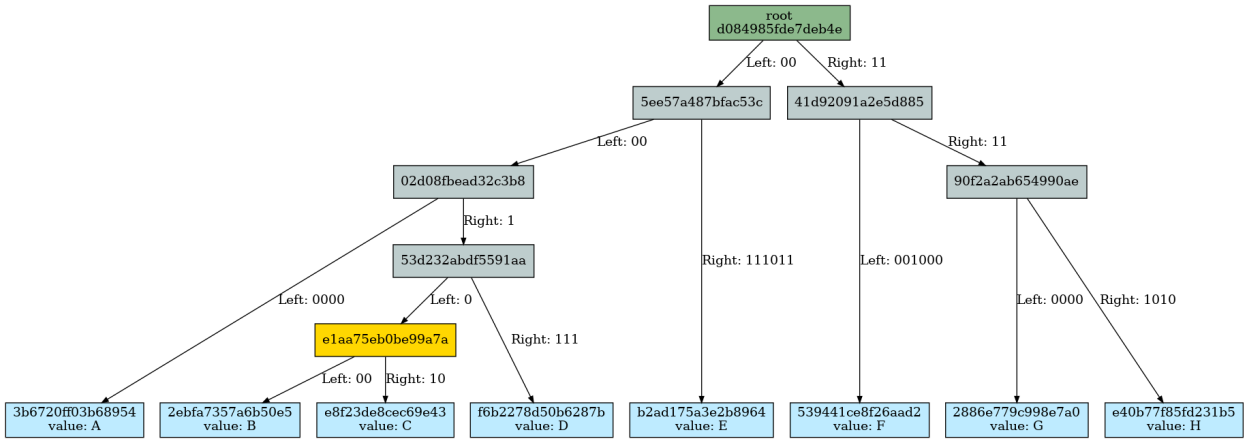


Figure 9: The starting tree, to be split at key `00001011` into two trees. The node highlighted in yellow is the point where both its child branches are on the same side of the split key.

Go back up one level. This is where the split starts. Figure 10 shows this node highlighted in yellow.

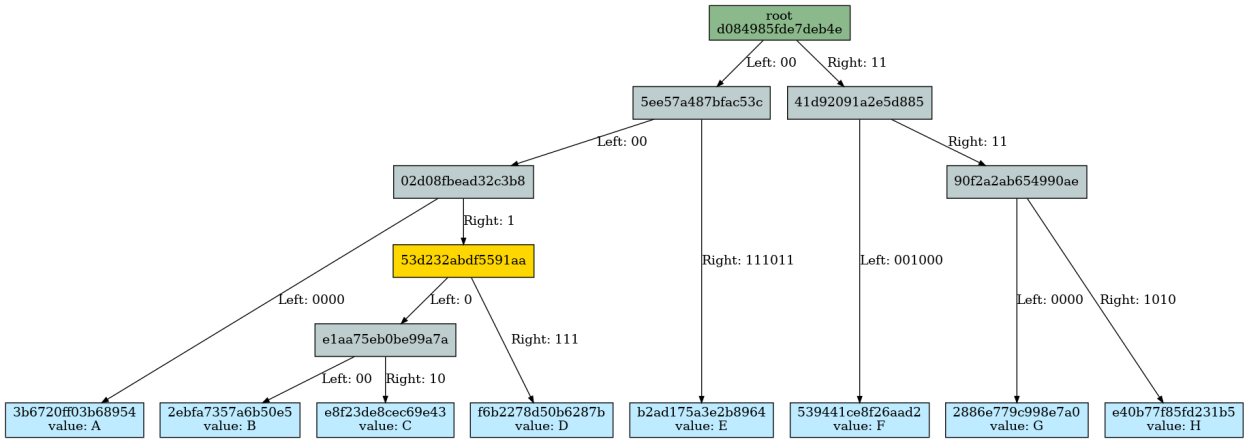


Figure 10: The starting tree, to be split at key `00001011` into two trees. The node highlighted in yellow is the point where we start the split operation.

In Figure 11 this node has been split into two, with a single child branch connected to each.

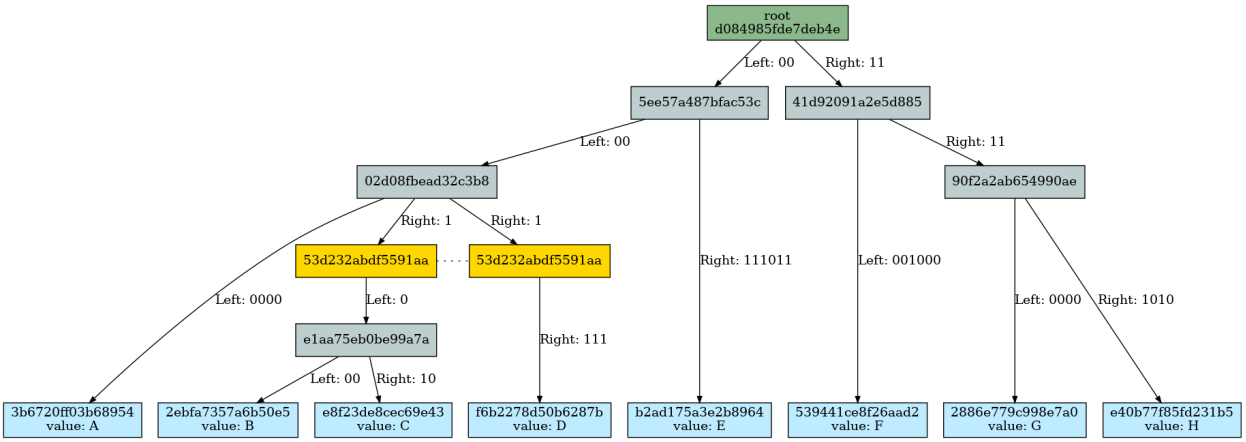


Figure 11: The tree is being split at key `00001011` into two trees. The node highlighted in yellow was split into two. The hashes shown are no longer valid, but are included for continuity.

Next, we walk back up towards the root, splitting each node in two. This is shown in Figure 12.

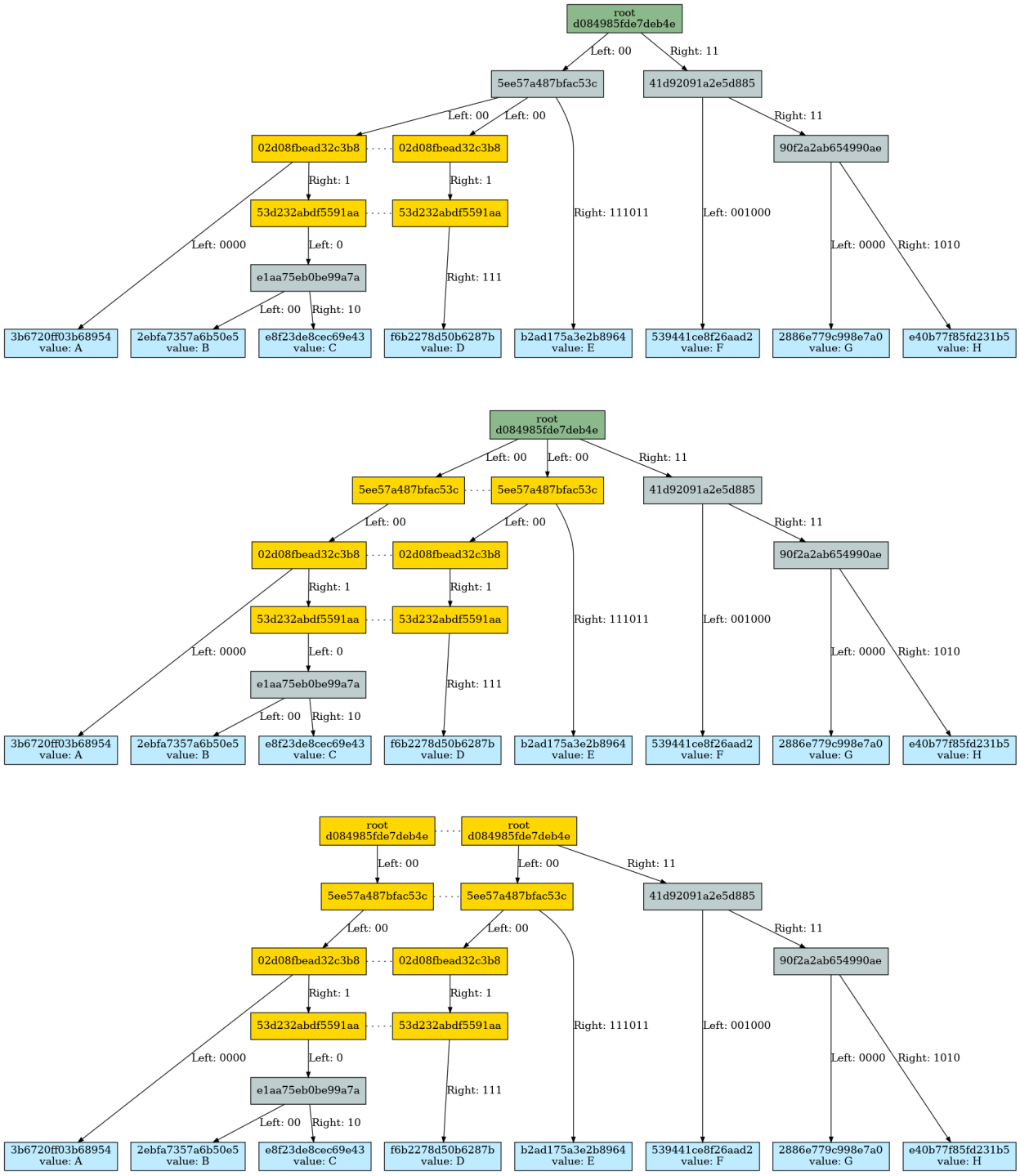


Figure 12: The tree is being split at key `00001011` into two trees. The sequence shows splitting each node up through the root. The nodes highlighted in yellow were split into two.

After reaching and splitting the root, you now have two independent trees. However, the trees don't satisfy the tree invariants because they have interior nodes below the root with only one child. These nodes are highlighted in purple in Figure 13. Each such node must be compressed into its parent by extending the parent branch's path.

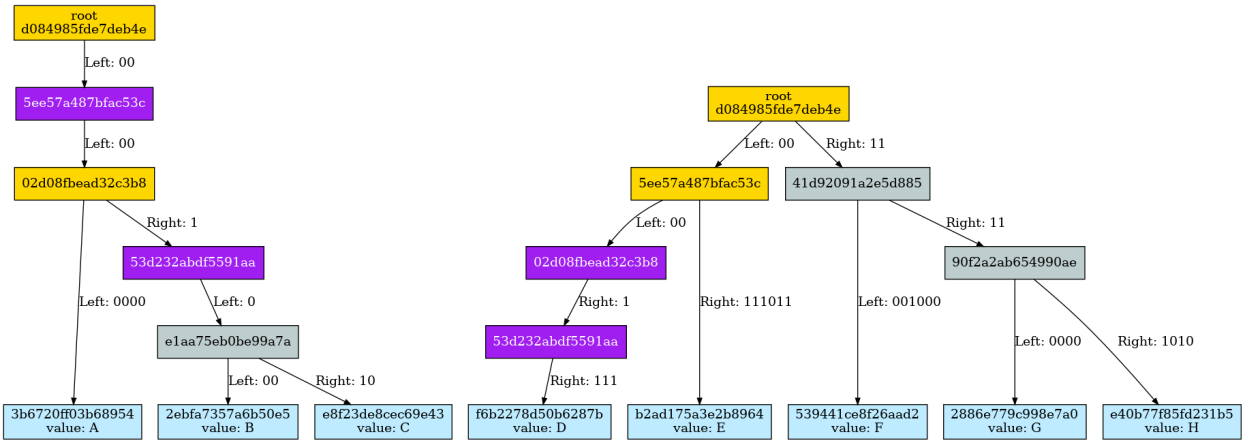


Figure 13: The tree is being split at key `00001011` into two trees. The nodes highlighted in purple have only a single child and must be compressed.

After compressing the nodes, recalculate the hashes. This restores the tree invariants and completes the split operation, leaving the two independent trees shown in Figure 14.

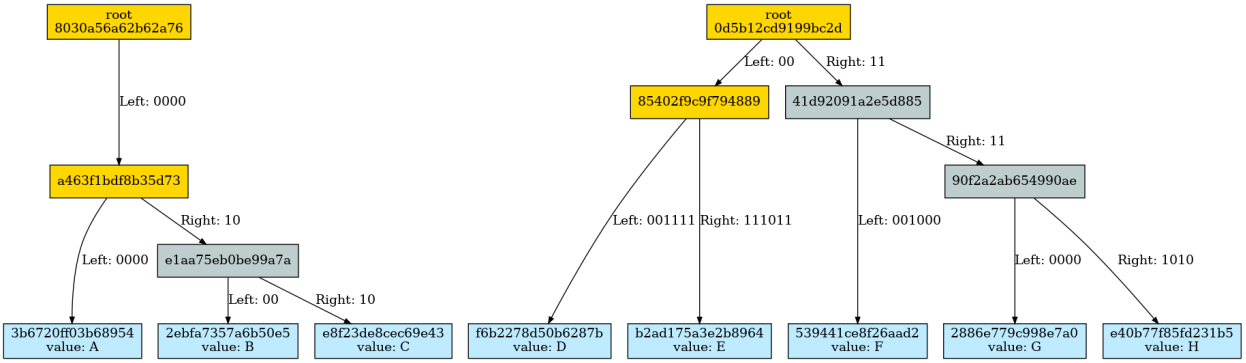


Figure 14: The original tree was split at key `00001011` into these two trees. The nodes highlighted in yellow were involved in the split.

5.2 Tree Merge

When scaling down or re-balancing load, you may want to reduce the number of trees. Merge is the inverse of split. Split takes a tree for a key range and makes two adjacent trees. Merge takes two trees for adjacent key ranges and turns them back into one tree. Figure 15 shows an example of two trees that we'd like to merge back into one (these are the trees resulting from the previous split).

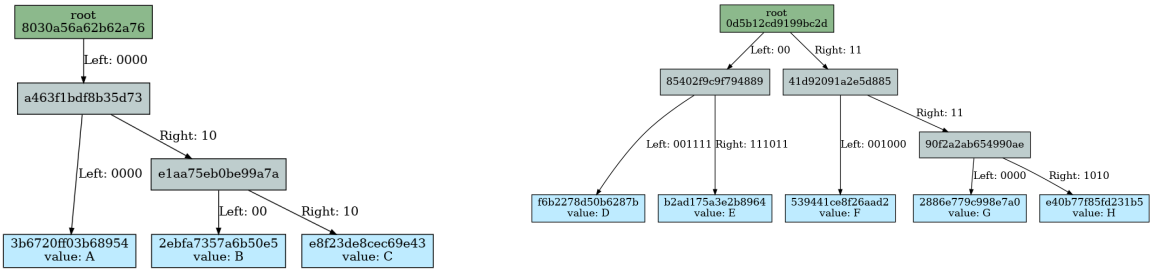


Figure 15: The trees for two adjacent key ranges that we want to merge.

Merge requires proofs for the edges of the trees that are joining. It requires a proof for the highest key in the tree from the lower-ranged tree and a proof for the lowest key in the tree for the higher-ranged tree. These proofs can be built without knowing the keys in advance by generally following the right branch for lower range and the left branch for the higher range. However, as shown in Figure 16, this always-left or always-right direction does not apply to the root node if the root node is missing that branch (in that case, its other branch is needed, if any).

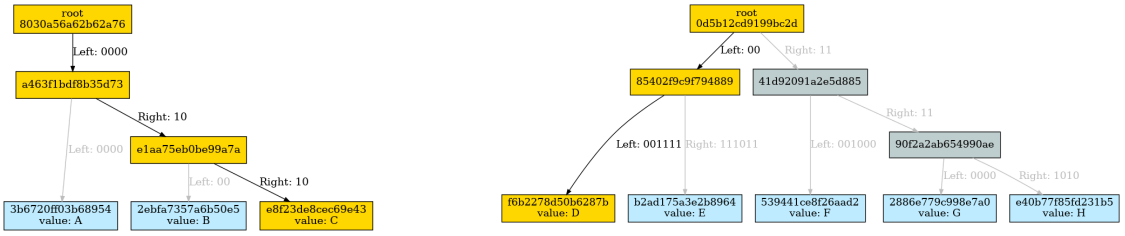


Figure 16: Highlighted in yellow are the edge proofs for the two trees we're merging.

Intuitively, there's a version of merge that works by joining the two proofs into one as they are traversed. However, we found the implementation of this approach to have too many complications. Instead, we take an alternative approach for merge. It's not quite as efficient, but the resulting code is straightforward to understand. As merge operates on a small number of nodes, more understandability for a small efficiency loss is a good trade-off.

To convert the two trees into one, we need to effectively remove the proof path nodes at the edges and connect the remaining branches into a single tree. To collect the relevant branches, traverse each proof and record the full key prefix and hash to each branch that does not lead to another interior node in that proof. We will keep these branches and discard the rest.

Continuing the previous example, Figure 17 shows the branches to collect in green. Table 2 shows the resulting list of prefixes and hashes that were collected.

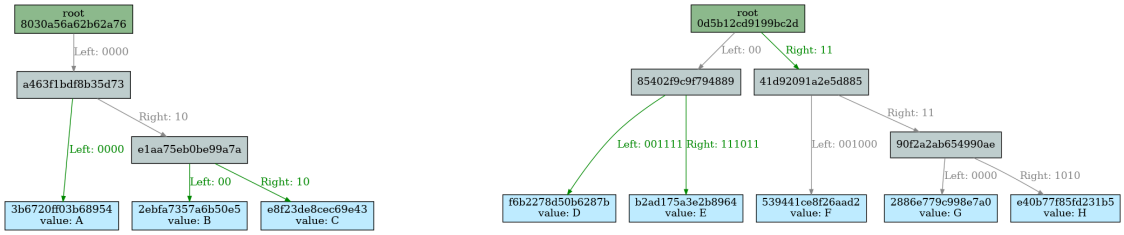


Figure 17: Highlighted in green are the branches of interest.

Key prefix	Hash
00000000	3b6720ff03b68954
00001000	2ebfa7357a6b50e5
00001010	e8f23de8cec69e43
00001111	f6b2278d50b6287b
00111011	b2ad175a3e2b8964
11	41d92091a2e5d885

Table 2: Key prefixes and hashes there were collected from the trees shown in Figure 17.

Now that we've collected all the items that need to be in our final tree, we can work on building a tree for them. We use a recursive function that walks down the prefix bits, looking for where some branches start with `0` and others start with `1`. These groups of branches are processed recursively and joined with a new interior node. If the branches all start with `0` or all start with `1`, that path bit does not result in a new node (it will be compressed by the radix encoding instead).

```
def build_tree(branches, start):
    if len(branches) == 1:
        return branches[0]
    bit = start
    while True:
        zeros = [b for b in branches if b.prefix[bit] == 0]
        ones = [b for b in branches if b.prefix[bit] == 1]
        if len(zeros) > 0 and len(ones) > 0:
            left = build_tree(zeros, bit + 1)
            right = build_tree(ones, bit + 1)
            return join_nodes(left, right, bit)
        bit += 1
```

In Table 3 the background colors indicate where the joining operations occur.

Key prefix	Hash
0 0 0 0 0 0 0 0	3b6720ff03b68954
0 0 0 0 1 0 0 0	2ebfa7357a6b50e5
0 0 0 0 1 0 1 0	e8f23de8cec69e43
0 0 0 0 1 1 1 1	f6b2278d50b6287b
0 0 1 1 1 0 1 1	b2ad175a3e2b8964
1 1	41d92091a2e5d885

Table 3: Groups of branches from Table 2 that share a common prefix. Each colored group becomes a new interior node in the resulting tree.

Once the function has recursed down and back up, it returns a single key prefix and a hash. If the key prefix is empty, then this is the new root node. If the key prefix is not empty, then an additional node needs to be inserted above this one to be the new root node.

We've completed rebuilding the tree and have our new single root node. The highlighted interior nodes in Figure 18 are those built by the `build_tree` function above.

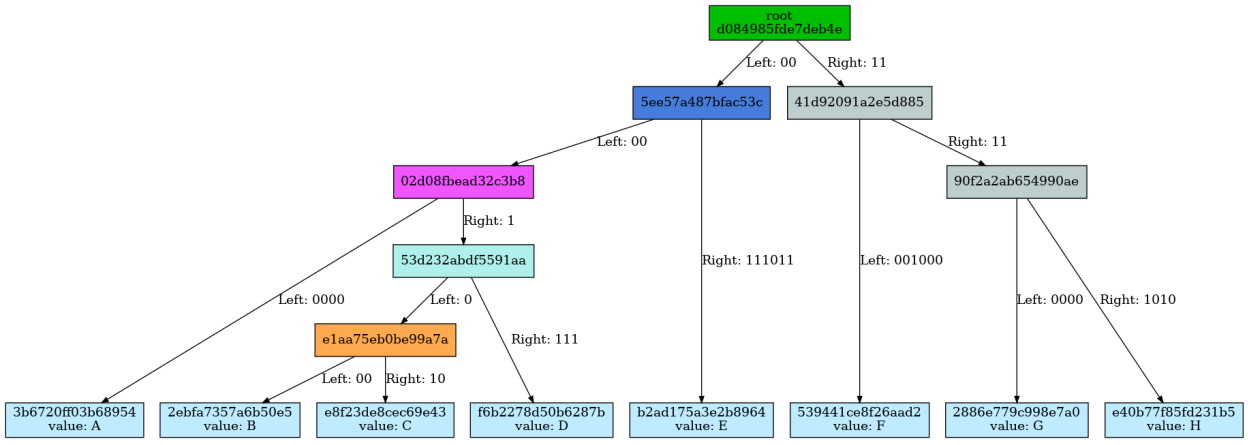


Figure 18: The final merged tree. Interior nodes created by the merge process are highlighted in colors matching Table 3.

6 Tree Overlay

A mutation to a standard Merkle tree must start with the latest state of the tree, using the latest root hash. However, this approach would be highly inefficient in our case, as an insert/update request would have to completely finish before the next request could start reading the tree.

To enable high throughput, the Merkle-radix tree allows reading from the tree storage in parallel with ongoing insert and update operations. These reads result in *stale proofs* – proofs with a root hash that is no longer the latest version of the tree. The HSMs accept stale proofs as long as the proofs are internally valid and use a known recent root hash. The HSMs use a local data structure called the *tree overlay* to "refresh" stale proofs and allow operations to take place as if the agents had sent a fresh proof.

The overlay allows the agent to issue multiple concurrent requests to modify a tree to an HSM. Because each request can be bundled with a stale proof, a "later" requests' proof need not reflect the root hash

produced by an “earlier” request. The design avoids waiting for the relatively long latency of writing to storage, and it results in huge throughput benefits (even for HSMs that process only one request at a time).

The overlay is an in-memory copy of the recent changes the HSM has made to the tree, along with the related root hashes. When the HSM needs to verify a proof, it checks the proof’s hash against its list of root hashes, rather than just the latest hash. Once the HSM has validated the proof, it rebuilds the path from the latest root to the leaf using a combination of the overlay and the nodes in the original proof. It now has the most recent view of the tree for that key. For a key that had been recently written, this may mean that the value for the key is taken from the overlay rather than the proof. If the request requires mutation of the tree, the HSM can safely do so from this most recent view. Once the HSM has completed its tree operation, it applies the changes to the tree overlay and externalizes the changes to be persisted in the external store.

Figure 19 shows an example of the overlay and how it is used. The overlay contains portions of the tree that the HSM has mutated recently (for example, after inserting some values). It contains a history of the 5 most recent root hashes; these mutations are included in the overlay nodes. The figure also shows a valid but stale proof from a recent root hash (not the latest). In this case, the overlay contains a more recent change to the key. The HSM constructs a fresh proof for that key entirely from the overlay nodes, as highlighted. Note that the overlay contains a newer leaf value than in the original proof.

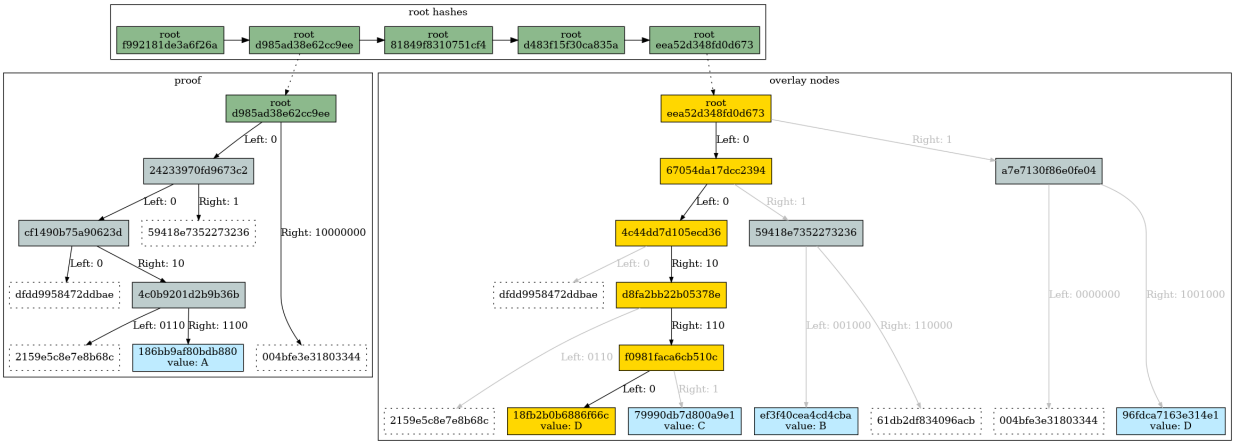


Figure 19: A stale but valid proof is shown for key `00101100`. The nodes highlighted in yellow represent the most recent path for the same key from the overlay.

Figure 20 shows a different scenario, where the key is not present in the overlay. The fresh proof begins with the latest root in the overlay and follows the path to the key. After the third node (highlighted in yellow), the overlay branches to a hash (`4c0`…) that’s not found in the overlay but is found in the stale proof. Thus, the upper portion of the fresh proof comes from the tree overlay, and the lower portion comes from the stale proof. This combination represents the latest proof because, if those nodes had changed in the interim, they would have been present in the overlay.

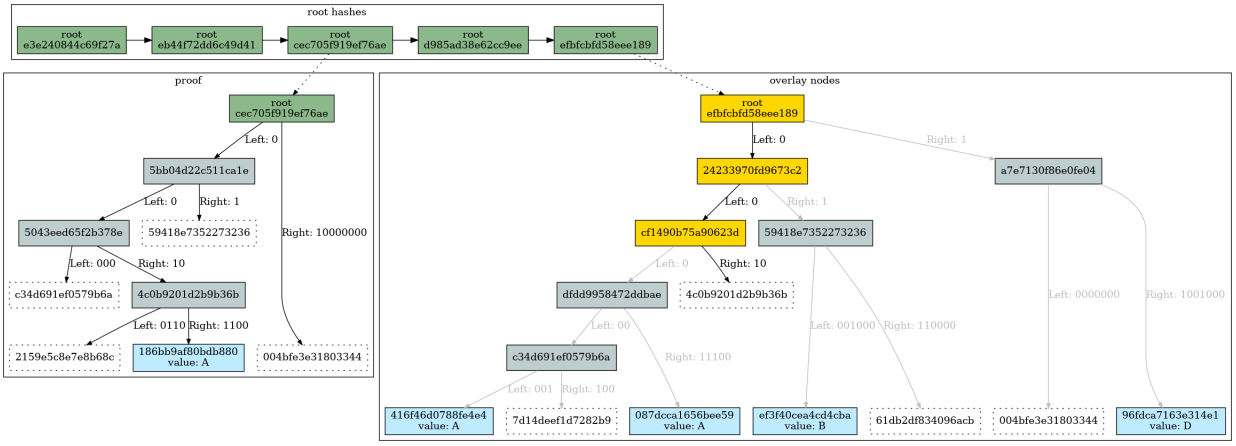


Figure 20: A stale but valid proof is shown for key `00101100`. The nodes highlighted in yellow represent the most recent partial path for the same key from the overlay. The full path can be completed with the final two nodes from the proof.

The tree overlay is configured to keep the last N mutations and acts as a cache in some ways. When the oldest mutation is evicted from the overlay, any tree nodes that it added are removed from the overlay. The top of the tree is updated often (the root is updated with every mutation), so generally the evictions remove leaves and interior nodes deep in the tree, as well as nodes higher in the tree that are no longer reachable from the latest root.

In the Juicebox application, the Merkle tree is managed in conjunction with a log that tracks the history of changes to the tree. The log allows for other HSMs to take over in the event of a failure. This is described in more detail in Authenticated Consensus Protocol[2].

7 Storage

The agent is responsible for reading and writing tree nodes to the external storage system. How the service interacts with the storage is a significant factor in its overall cost and performance. The radix compression helps significantly, but trees with many records are still quite deep. For example, a binary tree storing 1 billion nodes will have a depth of about 30 levels, so every incoming request will require reading 30 tree nodes from storage, then possibly writing 30 new nodes back to storage.

Juicebox currently uses Google Cloud Bigtable[7] to store the tree nodes. Bigtable is scalable and relatively cheap. It does not support multi-row transactions or snapshot reads, and storage operations incur significant latency. All tree nodes are stored in one table in Bigtable with a single column, indexed by a primary key. Storing the nodes for all trees in the same table allows for faster repartitioning, as the tree nodes do not have to be copied.

7.1 Non-Transactional Storage

A transactional storage system would allow executing read requests on a snapshot of the tree and allow modifying the tree in a single atomic and isolated request. This would be straightforward. Unfortunately, many common storage systems, including Bigtable, do not provide such transactional features. Systems that do may be more expensive or slower than non-transactional systems (or just the transactional operations may be more expensive or slower than non-transactional operations on the same systems).

The Juicebox Merkle-radix tree is designed to be stored on a non-transactional storage system. This introduces some challenges. The agent must present a valid proof to the HSM, so it cannot collect a mishmash of nodes from the tree while the tree is being modified concurrently. Writes to the tree must leave the tree in a readable state in the event the agent fails.

The tree nodes are written to the store with a primary key that includes the node's hash. Mutations to the tree always generate new hashes, so they never overwrite an existing store key. This allows readers to read a consistent snapshot of the tree, even while mutations are taking place. The tree nodes can be written to the store in batch or in parallel, as long as readers only use the new root hash after all the writes complete.

Mutations cause old nodes to be deleted. This includes deleting the previous root node with every mutation. The agents delay the deletes by a few seconds so that concurrent readers can finish reading their snapshot of the tree. A very slow reader may find that a tree node it needs has been deleted; in this case, it can retry from the latest root node. Delete nodes can be done in batch or in parallel, even for a single update to the tree.

To summarize, an agent applies a mutation to the store as follows:

1. Write the new Merkle nodes.
2. Publish the new root hash for readers to use.
3. After some delay, delete the Merkle nodes that were to be removed.

This approach has one limitation: if the agent fails before it's able to complete these steps, it will leave behind cruft—tree nodes that are not reachable from the latest root hash. We expect these events to be rare and such cruft to be a relatively small or even negligible fraction of the overall storage. While we have not yet implemented a solution, a garbage collector could identify and delete these unnecessary nodes, without needing transactions.

7.2 Concurrent Path Lookups

If agents stored tree nodes indexed by their hashes alone, the agents would have to read $tree\ depth$ nodes sequentially to form a proof, starting from the root node. This would incur high latency, especially since Bigtable does not offer stored procedures to express these dependent reads. The agent would have to serially read a node from storage, pick a branch based on the key, then go back to storage and read the next node, etc.

Juicebox's design allows for reading all the tree nodes in batch or in parallel for a single path lookup. This requires the storage system to provide prefix reads. Bigtable supports range reads, which are a strictly more general form of prefix reads. The prefix reads allow the agent to look up a node by a prefix of the tree key, without knowing that node's hash. The storage system will return the node that the agent requested, though it might also return some extra nodes that have been modified recently.

The agent encodes the store keys as the node's key prefix followed by the node hash. This allows a prefix read to find the nodes that start with a particular key prefix, without knowing their hashes. A leaf is encoded as the full key and the leaf's hash. A root node is encoded as an empty key and the root hash. Other interior nodes are encoded with their position in the tree and that node's hash.

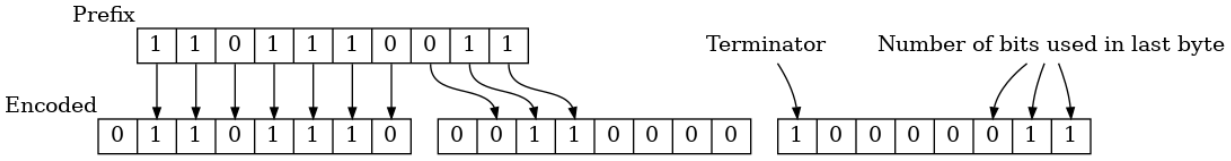
The node's key prefix is encoded with a trailing delimiter, and this delimiter is included in the prefix reads. The delimiter prevents a shorter key prefix from containing a longer key prefix. For example, it prevents a lookup for the root node (with an empty prefix) from retrieving every node in the storage. The trailing

delimiter is more efficient than encoding a length before the key prefix because it improves Bigtable's locality. Bigtable stores data in sorted order (roughly speaking), so using the delimiter should allow for nodes lower in the tree to be stored near each other and have better cache locality.

The agent issues concurrent prefix read requests to the store based on just the node's tree key prefixes, not their hashes. To read the nodes forming a path to a n-bit tree key:

1. The agent issues $n + 1$ prefix reads in batch or in parallel, for every possible prefix of that key. The prefix reads will collectively return all the nodes needed for the path (and likely some extraneous nodes). Most of the reads will not return any nodes, since the radix compression eliminates most intermediate nodes. We believe the cache locality makes these operations relatively cheap.
2. The agent determines a recent root hash, finds that node in its result set, and follows the branches down the tree in its result set. The resulting nodes form the proof, and any remaining nodes are ignored.

The exact key encoding is more complicated, because it's ambiguous to append a delimiter to binary data, and the number of bits in the tree key prefix isn't necessarily a multiple of 8. We use a 7-bit encoding of the tree key prefix, where the high-order bit is set to `0`. That's followed by a single byte that encodes the number of bits used in the previous byte, with a high-order bit set to `1`. Figure 21 shows some example encodings.



Prefix Bits	Encoded Bytes
(empty)	`10000000`
`1`	`01000000 10000001`
`011`	`00110000 10000011`
`0101010`	`00101010 10000111`
`11111111`	`01111111 01000000 10000001`

Figure 21: Tree key prefix encoding and examples.

This encoding satisfies the property that adding a wildcard after an encoded prefix will never match another encoded prefix. Therefore, reading nodes at one level of depth in the tree will never return nodes from another level of depth in the tree. Table 4 shows the first 10 prefixes of the tree key with all `1`s to illustrate this property.

Prefix Bits	Encoded Bytes - Binary	Hex
(empty)	10000000	80
1	01000000 10000001	40 81
11	01100000 10000010	60 82
111	01110000 10000011	70 83
1111	01111000 10000100	78 84
11111	01111100 10000101	7B 85
111111	01111110 10000110	7E 86
1111111	01111111 10000111	7F 87
11111111	01111111 01000000 10000001	7F 40 81
111111111	01111111 01100000 10000010	7F 60 82

Table 4: The first 10 encoded prefixes for a tree key that consists of all 1s. A wildcard following any encoded prefix will never match another encoded prefix.

7.3 Agent Cache

The previous subsection showed how to read all the nodes from the store for a single tree lookup concurrently. Even so, read operations are still frequent and relatively expensive. To improve performance and reduce load on the store, the agents maintain an LRU cache of tree nodes, indexed by their store keys. They update this cache when they read nodes from the store or write nodes to the store.

Agents walk the nodes in the cache from the root hash when looking up a key in the tree. The nodes at the top of the tree will normally be found in the cache. If the next node is not found in the cache, the agent then issues reads for the remaining prefixes to the store, as described previously, in parallel or in batch.

We have found that even a modestly-sized cache helps significantly:

- The same exact leaf is often needed more than once in different requests. Juicebox's application exhibits temporal locality, where recovering a secret requires a client to make 3 sequential requests for the same record.
- When the tree is undergoing high levels of mutations, the delaying of deletes to the store causes a high ratio of extraneous nodes to be returned in the prefix lookups for short key prefixes, towards the top of the tree. Without the cache, these nodes are read and returned to the agent just to be discarded.
- The top of the tree is very cheap to cache. For example, the first 8 levels of the tree require just 255 cache entries and reduce lookups by over 25%, even for very large trees.

8 Security Considerations

The clients make requests to the HSMs via the untrusted agents. As these requests are assumed to be sensitive, the requests into and responses out of the HSM should be encrypted.

The agents may observe several side channels: which records are read and modified in the tree and when, how large the records are, and how long the HSMs take to process requests. To address the latter two, the HSM can pad the records to obscure their size and can use constant-time algorithms to evaluate requests.

The tree overlay could conceivably allow an adversary to issue requests for the same record in rapid succession, which could amplify a timing side-channel. As a mitigation, the HSM can rate-limit requests for a record or reject subsequent requests until earlier changes for that record have fully completed.

The Merkle-radix tree is an unbalanced tree. An adversary in control of the tree keys could create a purposely imbalanced tree to slow down the tree operations (similar to "Hash DoS" attacks[8]). To mitigate this, the agents could use the output of a keyed cryptographic hash function as the tree keys.

9 Conclusion

This paper has described the design of Juicebox's Merkle-Radix Tree, a core building block that enables our scalable and secure service to use HSMs with limited capacity. The tree combines a wide variety of techniques and optimizations to arrive at a practical and cost-effective system. We believe the design will scale up to storing billions of records, while also being practical and cost-effective for small deployments.

10 References

[1] N. Trapp and D. Ongaro, "Juicebox Protocol". [Online]. Available: https://juicebox.xyz/papers/TODO.pdf

[2] D. Ongaro and S. Fell, "Authenticated Consensus Protocol". [Online]. Available: https://juicebox.xyz/papers/TODO.pdf

[3] R. C. Merkle, "A Digital Signature Based on a Conventional Encryption Function", C. Pomerance, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 369–378.

[4] D. R. Morrison, "PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric", *J. ACM*, no. 4, p. 514, Oct. 1968, doi: 10.1145/321479.321481.

[5] M.-J. O. Saarinen and J.-P. Aumasson, "RFC 7693: The blake2 cryptographic hash and message authentication code (MAC)". [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7693

[6] S. Arciszewski, "XChaCha: eXtended-nonce ChaCha and AEAD_XChaCha20_Poly1305", Internet Engineering Task Force, Jan. 2020. Available: https://datatracker.ietf.org/doc/draft-irtf-cfrg-xchacha/03/

[7] F. Chang *et al.*, "Bigtable: A Distributed Storage System for Structured Data", 2006, pp. 205–218.

[8] S. A. Crosby and D. S. Wallach, "Denial of Service via Algorithmic Complexity Attacks", Washington, D.C.: USENIX Association, Aug. 2003. Available: https://www.usenix.org/conference/12th-usenix-security-symposium/denial-service-algorithmic-complexity-attacks