Range-Based Set Reconciliation Without Homomorphic Hashing

Aljoscha Meyer
Technical University Berlin
Email: research@aljoscha-meyer.de

Konstantin Scherer
Unaffiliated

***Abstract*—Range-based set reconciliation is a set reconciliation algorithm whose complexity tradeoffs excel in settings where nodes of limited computational capabilities reconcile sets with a large number of possibly malicious peers, such as in as peer-to-peer systems. Previous presentations of range-based reconciliation have relied on non-standard cryptographic hash functions with certain homomorphic properties. We remove this reliance on non-standard hashing, allowing to use range-based reconciliation with traditional hash functions such as the SHA family. We argue that range-based reconciliation strictly outperforms reconciliation based on merkle-search-trees, another algorithm which was developed for peer-to-peer systems and which does not require homomorphic hash functions. By eliminating the need for homomorphic hashing from range-based reconciliation, we effectively render merkle-search-tree reconciliation obsolete.**

I. Introduction

The problem of set reconciliation asks two nodes holding a set each to both efficiently obtain the union of the two sets. There exist several probabilistic algorithms with different complexity profiles; we focus on *range-based set reconciliation* (*RBSR*) [1]. Prior presentations of RBSR rely on non-standard homomorphic hash functions for efficient implementation. We present an alternate implementation technique that can make use of arbitrary hash functions.

In evaluating set reconciliation algorithms, there are several criteria of interest. The most obvious ones are the number of roundtrips and the number of transmitted bits. The latter is typically not measured against the total size of the union, but against the size of the symmetric difference of the two reconciled sets. When reconciling large but similar sets, an algorithm should send only small amounts of data.

While there are many algorithms optimizing for solely these two complexity criteria, there are several other criteria that are relevant in practice. Computing the messages that the nodes are sending must be efficient (or at least feasible). When a node updates its set, it must be able to efficiently update any auxiliary datastructures it requires for running the protocol.

Beyond traditional complexity criteria, another important criterium is *censorship resistence*: a malicious node should be unable to generate sets such that two honest nodes that attempt reconciliation of these two sets do *not* end up with the union. Ideally, censorship resistence should be obtained while maintaining *universality*: the most significant part of the computations that a node must perform should be independent of the other node's set; this allows nodes to efficiently reconcile with a large number of other nodes.

Two algorithms that factor in these additional criteria are RBSR, and the MST[1] reconciliation of Auvolat and Taïani [2]. Both are recursive algorithms where the nodes transmit sets of fingerprints of contiguous subranges which partition the original set. When receiving a fingerprint that a node already has, that node knows the corresponding subrange to be fully reconciled. For mismatching subranges fingerprints, the node partitions the subrange and sends fingerprints for each resulting subsubrange in the next communication round. For sufficiently small subranges, a node simply sends the items directly, ending the recursion.

MST reconciliation is driven by the unique representation of a node's set as a particular data structure, the MST. When a node splits a range, it may only split it into ranges such that the set of items in the range is exactly the set of items stored in a subtree of the node's MST. The MST is a Merkle-tree, and a fingerprint for a range is simply the label of the root of the corresponding subtree. Thus, the recursive splitting is guided by a pseudorandomly determined tree shape rather than by optimal choices.

RBSR, in contrast, has the nodes perform optimal partitioning by splitting each range into subranges of equal sizes. This guarantees a logarithmic number of communication rounds, but comes at the cost of more complicated fingerprint computation — Merkle tree labels do not suffice, since ranges do not necessarily correspond to subtrees. Fingerprint computation still relies on maintaining a search tree that labels each vertex with the fingerprint of its corresponding subtree (called a *monoid tree*). The fingerprint function must further satisfy an algebraic property: there must be an associative function that computes the fingerprint of the union of two non-overlapping ranges from their individual fingerprints. Given this function and a monoid tree, the fingerprint of an arbitrary subrange can be computed via a tree traversal that runs in time proportional to the height of the tree. This requirement precludes usage of conventional hash functions such as the SHA family.

Our main contribution stems from a single observation: if the auxiliary tree structure belongs to a certain family of history-independent search trees, then the tree traversal

[1] "MST" is short for "Merkle Search Tree". MSTs are *not* arbitrary Merkle-trees that are also search trees, but a particular kind of history-independent data structure introduced by Auvolat and Taïani.

developed for RBSR produces unique results even for non-associative fingerprint functions. This gives us an alternate family of fingerprinting schemes suitable for RBSR which can make use of conventional hash functions. Conversely, we can frame this result as augmenting MST-like reconciliation with the ability to compute fingerprints for arbitrary ranges, reducing the asymptotic worst-case complexity and the practical average-case complexity in the process. In both views, we significantly improve a state-of-the-art, practically applicable algorithm.

The remainder of this paper is structured as follows. We review related work in Section II, and define our required preliminaries in Section III. In Section IV, we define the core principles behind non-homomorphic RBSR, precisely characterize the data structures that can be used to define and compute the fingerprints, and provide an optimized algorithm for the computation. We conclude in Section V.

II. Related Work

The two classic approaches to set reconciliation — *characteristic polynomial interpolation* (CPI) [3] and *invertable bloom lookup tables* (IBLTs) [4] — and their numerous variants rigorously minimize communication complexity and roundtrips at the cost of high computational loads.

Our work falls into a different family of reconciliation work: approaches which reduce the computational complexity below $\mathcal{O}(n)$ for nodes holding a set of size n by accepting a logarithmic number of communication rounds.

The first of these approaches is partition reconciliation [5], which organizes a set as a *partition tree* of subranges, and precomputes CPI messages for each subrange in the tree. The peers exchange these messages layer by layer; reconciliation of each tree vertex succeeds when the size of the symmetric difference between the peers' items within a range falls below a configurable threshold. Unfortunately, this approach is not censorship-resistant, malicious actors can craft mismatching sets for which the peers believe that reconciliation succeeded without actually reconciling any differences. Furthermore, the partition tree can become unbalanced over time, causing a degenerate number of roundtrips.

MST-based reconciliation [2] partitions sets into a (particular, history-independent) tree of subranges, and precomputes a label for each vertex that fingerprints the set stored in the subtree. The peers exchange these labels layer by layer, using label equality to determine when a range has been fully reconciled. The tree structure is pseudo-randomized, so the average-case complexity is worse than if a perfectly balanced tree could be used. Further, malicious data sources can compute trees of n items in $\mathcal{O}(n)$ time with high probability that degenerate to a single, large array. For these degenerate trees, one peer ends up sending its complete set, irrespective of how small the symmetric difference might be.

Most crucially however, MST-based reconciliation is asymmetric. One peer repeatedly sends the labels of its vertices — one layer of the tree per communication round — and the other peer passively responds which labels matched and which differed. Our approach (just like traditional RBSR based on homomorphic fingerprinting) has peers directly reply with labels of their own. By having both peers actively send labels, RBSR performs twice as many subdivisions per communication round compared to MST-based reconciliation, thus halving the number of roundtrips.

Range-based set reconciliation [1] also has the peers recursively exchange fingerprints for subranges. Unlike MST-based reconciliation, these subranges can be chosen arbitrarily, with the optimal choice being splits into subranges of equal sizes (assuming uniformly distributed differences between the sets). This increases the burden on fingerprint computation; prior presentations of RBSR require usage of homomorphic hash functions for label computation to achieve computation times of $\mathcal{O}(\log(n))$. Our approach to fingerprint computation lifts this restriction, at the cost of increasing the local computation times to $\mathcal{O}(n)$ for specific, maliciously crafted sets. Communication complexity, roundtrips, and censorship resistance remain immune to malicious input.

RBSR is the only algorithm we are aware of that can operate with a static, well-known instantiation (the secure fingerprinting function) while staying resistant to maliciously crafted or modified data sets, both in terms of worst-case complexities and in terms of censorship resistance. MST-reconciliation can protect against malicious input only by randomizing the tree construction for each reconciliation session, which would cause an $\mathcal{O}(n)$ computational overhead per session. Similarly, traditional reconciliation approaches such as CPI and IBLTs require per-session randomization to remain secure against adversarial data sets (which does not impact their complexities, because they *already* require $\mathcal{O}(n)$ computation steps for each session).

A more recent approach to set reconciliation proposes *rateless invertable bloom lookup tables* (RIBLTs) [6], and claims to combine sublinear computational complexities, communication complexity and roundtrips comparable to computationally more expensive approaches, and suitability for adversarial settings. Their solution for adversarial settings is to randomize a hash function per session, thus incurring full computational overhead in every single session instead of a single precomputation, thus still leaving RBSR as the only algorithm to handle adversarial inputs without resorting to per-session randomization.

More gravely, however, we disagree with their assessment of the communication complexity of their algorithm. Their algorithm consists of one peer streaming a conceptually infinite sequence of coded symbols to the other peer. Once the other peer has received enough symbols, it notifies the streamer to stop. The expected length of the stream is linear in the size of the symmetric difference. While this seems efficient at first glance, there is a hidden factor: the latency between sending the stop signal, and the streamer receiving it and halting transmission. During the time between the receiver decoding the difference and sending a stop signal, and the arrival of the stop signal at the sender, the sender continues transmitting. This overhead is completely unrelated to the size

of the symmetric difference. If the symmetric difference is small, but the latency of the stop signal is high, the analysis turns unfavourable very quickly.

A proper assessment of the RIBLT algorithm requires a networking model that takes into account bandwidth and latency. And it should be compared not against reconciliation approaches designed for a different networking model, but ones designed for the same model. Partition reconciliation, for example, can easily be adapted to also use the stream-until-stop-signal technique:

Classic partition reconciliation sends the messages for a single layer of the partition tree, and then waits for a response by the other peer. It could just as easily be adapted to continuously stream the messages of all vertices of the partition tree, in layer-ordering, and stop once the other peer tells it to. Similarly, both MST reconciliation and RBSR could have both peers stream the fingerprints of the internal trees in layer-ordering as well, and thin out the streams whenever a peer receives a matching fingerprint from the other peer. It is these variations that would have to be compared against the RIBLTs aproach, in an appropriate evaluation framework. Simply declaring the RIBLT approach to be more efficient by ignoring latency is incorrect, and actually diminishes the contribution of the authors: the impact of the intricate and elegant rateless IBLT is dwarfed by the comparatively primitive technique of spamming data until receiving a stop signal.

III. Preliminaries

We now introduce the precise definitions and notation we use for some well-known concepts.

A. Trees

A *tree* over some universe U is either the *empty tree*, or a pair of

- a finite sequence $keys$ of elements of U, individually denoted as key_i, and
- a sequence $children$ of trees, containing exactly one more element than $keys$, individually denoted as $children_i$.

For any $k \in \mathbb{N}, k \geq 2$, a tree is called *$k$-ary* if it is the empty tree, or if $children$ has length at most k and all children are also k-ary.

The set of *items* of a tree is the empty set if the tree is the empty tree, or the union of all $keys$ of the tree and the items of its children. A tree over a totally ordered universe is a *search tree* if it is the empty tree, or if it has l many $keys$ and

- the greatest item of $children_i$ is strictly less than key_i for all $0 \leq i < l$,
- the least item of $children_i$ is strictly greater than key_{i-1} for all $1 \leq i \leq l$, and
- all its children are search trees.

B. Merkle Trees

A *hash function* is a function from the set of finite bitstrings into the set of bitstrings of some length l. It is called *pre-image resistant* if it is computationally infeasible to find an input that maps to any given output. It is called *collision-resistant*, if it is computationally infeasible to find two inuts that map to the same output. It is called *secure* if it is both pre-image-resistant and collision-resistant.

Let t be a search tree over the set of finite bitstrings (ordered, say, lexicographically), and let h be a secure hash function. Let $empty\_label$ be the result of applying h to the empty string. We then define a *Merkle-tree* as a search tree together with a labeling function label that maps the empty tree to $empty\_label$, and any non-empty tree of l many $keys$ to $h(h(children_0) \cdot h(key_0) \cdot \ldots \cdot h(key_{l-1}) \cdot h(children_l))$, where $\cdot$ denotes concatenation of bitstrings.

This definition diverges from the classic Merkle-tree, which is a binary tree that stores items only in its leaves [7]. Our variant is a k-ary generalization of the *authenticated search tree* of Buldas, Laud, and Lipmaa[8]. At the end of the day, the precise definition is less important than the resulting properties: it must be computationally infeasible to produce two non-equal trees with the same root label (collision resistence), or any tree with an arbitrary given root label (pre-image resistence).

C. History-Independence

The algorithms we discuss require the reconciling nodes to represent equal sets as equal trees, in order to compute matching hashes. To get a more formal grasp on the relation between abstract data types and concrete representations, we employ Pugh's representation functions [9]: A *representation function* is a function from some concrete data type σ to an abstract data type τ.

Diverging from Pugh's original definition, we define a *representation scheme* for τ via σ as a *surjective* representation function from σ to τ. That is, we require every possible instance of τ to have at least one concrete representation in σ. As an example, the search trees over a universe U represent the abstract type of all finite subsets of U. The surjective representation function to witness this claim maps each tree to the set of its items.

We call a representation scheme *history-independent* if its representation function is *injective* (and thus, bijective). The bijection corresponds to the intuitive notion of a one-to-one correspondence between data structures and represented data that sits at the heart of other authors' definitions of this concept (*unique representation* [10], *structural unicity* [2], *confluent persistence* [11], *anti-persistence* [12]).

History-independent set data structures which guarantee access in logarithmic time must necessarily take superlogarithmic time for insertions and deletions [10]. In practice, most history-independent data structures employ randomization to achieve logarithmic time complexities for both access and mutation with high probability only. Conveniently, many randomized set data structures can be converted into history-independent data structures by using the items as sources of pseudorandomness. History-independent data structures in this category include treaps [13], skip-lists [14],

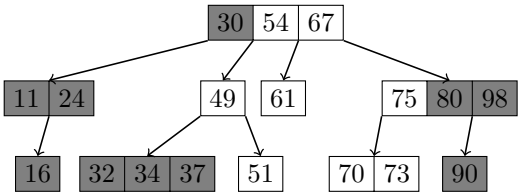

Fig. 1. An example Merkle-tree, highlighting the items in the range $[41, 77)$. None of the labels in the tree corresponds exactly to the items in the range. Some subtrees are neither fully contained in nor fully outside the range.

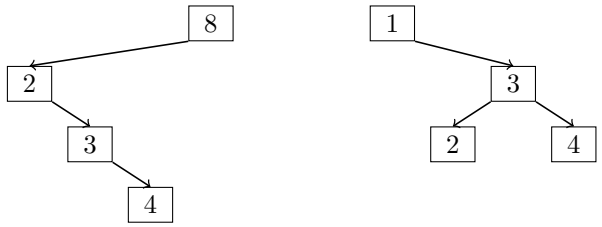

Fig. 2. Two example Merkle-tree, both containing a subtree on the items $\{2, 3, 4\}$. The subtrees have different shapes. History-independence does not forbid this, it is too weak a property to state anything about subtrees.

zip-trees [15], zip-zip-trees [16], B-treaps [17], B-skip-lists [18], randomized-block-search-trees [19], the external-memory, history-independent B-tree and skip-list versions of Bender et al. [20], skip-trees [21], dense skip-trees [22], MSTs [2], prolly-trees [23], and G-trees [24].

IV. Merkle-Tree-Based Fingerprinting

In order to perform range-based set reconciliation, nodes must efficiently compute fingerprints for any subranges of their set. History-independent Merkle-trees suggest an obvious definition for the fingerprint of each set: the root label of the unique Merkle-tree for that set. The tricky part is being able to efficiently compute that label for *arbitrary* subranges of a *given* set; this is the main challenge we solve.

The basic idea is for nodes to always store the Merkle tree of their set. When attempting to compute the fingerprint for a subrange of the set, two main problems can prevent the reuse of the precomputed labels in the Merkle tree: range boundaries might pass through larger nodes (Fig. 1), and equal subsets might be stored in mismatched tree shapes (Fig. 2).

To overcome these problems, we define the notion of *clamping* a given tree down to a given range; intuitively, this means considering the subtree that remains after ignoring all items outside the range. After giving a formal definition of clamping, we can define a tree traversal that computes the root label of the result of any clamping application inside a larger Merkle tree (without explicitly constructing the subtree); this procedure solves the issue of inconvenient range boundaries (Fig. 1), and is how nodes compute their fingerprint. Finally, we can define families of trees in which all larger trees clamp down to equal subtrees for equal ranges; this solves the problem of mismatched tree shapes (Fig. 2), and ensures that all nodes compute equal fingerprints for equal subsets.

A. Clamping a Search Tree

Let t be a search tree over some universe U, and let $x, y \in U$ with $x < y$. We now define how to clamp t to the range $[x, y)$. We say an item a is *included* in the range $[x, y)$ if $x \leq a < y$.

Intuitively, we wish to simply discard all items outside the range (Fig. 3). Special care needs to be taken with vertices that contain no keys within the range at all, to not disconnect the tree (Fig. 4).

We write $\mathrm{clamp}(t, x, y)$ to denote the result of clamping the tree t to the range $[x, y)$.

If t is the empty tree, $\mathrm{clamp}(t, x, y)$ is the empty tree again.

For any non-empty tree t of l keys, let $start$ be the least index such that $key_{start} \geq x$ (or l if $key_{l-1} < x$), and let end be the greatest index such that $key_{end} < y$ (or -1 if $key_0 \geq y$).

If $start > end$, i.e., if no key_i of t is included in $[x, y)$, then $\mathrm{clamp}(t, x, y) := \mathrm{clamp}(children_{start}, x, y)$. Intuitively, we recurse in the one child that might contain items in the range.

Otherwise, we have $start \leq end$, i.e., t has at least one key in the range. Then $\mathrm{clamp}(t, x, y)$ is the tree whose $keys$ are $key_{start}, \ldots, key_{end}$, and whose $children$ are $\mathrm{clamp}(children_{start}, x, y), children_{start+1}, \ldots, children_{end}, \mathrm{clamp}(children_{end+1}, x, y)$.

B. Clamping-Invariant Representation Schemes

We call a history-independent representation scheme for the finite subsets of some totally ordered universe U via search trees over U *clamping-invariant* if, for all trees t, u, and all items $x, y \in U$ with $x < y$, we have that $\mathrm{clamp}(t, x, y) = \mathrm{clamp}(u, x, y)$ if t and u contain the same items in the range $[x, y)$.

This definition precisely enables Merkle-trees for usage with RBSR: when two peers store non-equal items in a range, their clamped subtrees are non-equal and thus have different root hashes. If they do store the same set in a range, their clamped subtrees will be equal, and hence have equal root hashes.

Most tree-based, efficient, history-independent set datastructures are clamping-invariant. Intuitively, this is not surprising: updates are efficient when modifications only have local effects but leave most of the tree unchanged, and clamping-invariance is a very specific expression of changes outside some area of the tree leaving that area itself unaffected.

To prove that some history-independent data structure is clamping-invariant, it suffices to show that the data structure is closed under application of clamp: since the data structure is history-independent, and clamping reduces any search tree down to a tree representing the intersection of range and the set of items in the starting tree, the resulting trees will always be equal **if** they conform to the constraints of the data structure.

We give an exemplary proof for treaps. Treaps [13] (pseudorandomly) assign to each item a numeric rank; the treap on a set of items is the unique tree that is both a search tree for the items and a heap for the ranks. We can show that clamp preserves both the search tree property and the heap property by induction on the height of the treap. For the empty treap

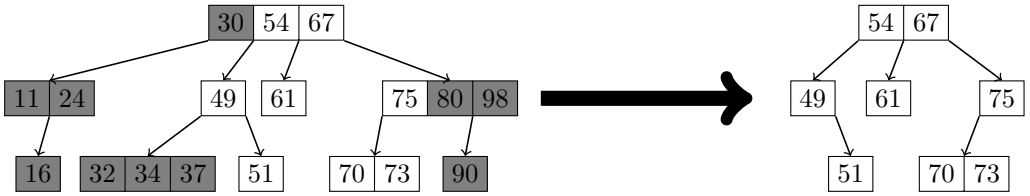

Fig. 3. An example of clamping a Merkle-tree to the range $[41, 77)$. All items outside the range are discarded.

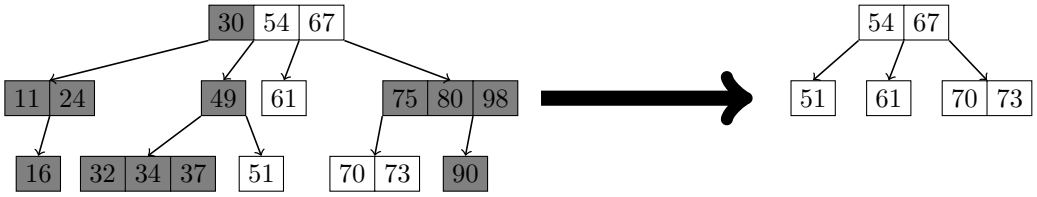

Fig. 4. An example of clamping a Merkle-tree to the range $[50, 74)$. Simply discarding items would leave the tree disconnected, so the subtrees need to be reattached closer to the root.

of height zero, clamp yields the empty tree again, which is a treap. Otherwise, let t be a treap of height h. There are two possible cases when applying clamp to t: if no key of t is in the range, clamp is applied to a single subtree. That subtree is a treap of height $h-1$, so clamp yields a treap by the induction hypothesis. If there are keys of t in the range, clamp is applied to several children, each of which results in a treap by the induction hypothesis. The maximum rank of the roots of each of these treaps is bounded by the rank of the original subtree, hence the heap property is preserved. Similarly, all items in these treaps were items in the original subtree, which are bounded by the parent keys; hence, the search-tree property is preserved. □

Zip-trees [15] are treaps with a tie-breaking rule for ranks. This tie-breaking can also be considered as refining the ordering of ranks, so the argument for preserving the heap property remains unchanged. Dense skip-trees [22], MSTs [2], and G-trees [24] admit simple inductive proofs analogous to that for treaps. The zip-zip-trees [16] are special cases of the G-trees, hence are also clamping-invariant, and the skip-trees [21] are special cases of MSTs.

Prolly-trees [23] are *not* clamping-invariant: their tree shapes depend on applying a rolling hash function to runs of consecutive items, and clamping items can change how many items the rolling hash function is applied to, influencing the required tree shape.

The remaining data structures listed in Section III-C are sufficiently complex that we consider proofs of clamping-invariance (or the finding of counterexamples) to be out of scope.

C. Clamped Fingerprint Computation

We now give an algorithm that takes a search-Merkle-tree t and a range $[x, y)$ as arguments and returns the root label of $\mathrm{clamp}(t, x, y)$. In the pseudocode, we write `nil` for the empty tree, [] for the empty string, $t.label$ for the (precomputed) label of the root vertex of a non-empty tree t, $t.keys[i]$ for the i-th key of t, $t.children[i]$ for the i-th child of t, and $t.len$ for the length of $t.keys$. Crucially, this algorithm does not need to construct $\mathrm{clamp}(t, x, y)$ itself to do so. Instead, the algorithm mimics the definition of clamp and only feeds those keys and subtrees into the label computation that are not clamped away.

Reusing precomputed labels of subtrees that are known to be unaffected by clamping is crucial for the efficiency of the algorithm. Algorithm 1 provides a naive translation from the definition of clamp into an algorithm. This algorithm is inefficient, however: there are up to two recursive procedure calls per step. In a binary tree that is fully included in the range, the algorithm visits every single vertex. But we are interested in a running time of $\mathcal{O}(\log(n))$, not $\mathcal{O}(n)$.

Fortunately, a relatively simple observation suffices to speed up the algorithm: for all but one vertex that contains keys in the range, one of the two recursive applications of clamp has no effect. To see why, consider the first such vertex; this is the one vertex where both recursive applications might result in clamping off some items. Now consider the clamping of the greater of the two clamped subtrees. We know all its items to be greater than or equal to the lower boundary of the range. For any vertex in that subtree that has keys in the range, we thus know that clamping its least subtree leaves that subtree unchanged. Analogously, clamping the greatest subtree of descendents of the lesser of the two first clamped subtrees has no effect either.

We can leverage this insight in clamped label computation by passing two boolean flags to the algorithm, initially set to $\bot$. When the algorithm encounters its first vertex that has keys in the range, it sets one of the flags to $\top$ in each of its recursive applications. When a flag is $\top$, the algorithm reads precomputed labels — rather than recursing — on the corresponding end of the range. Algorithm 2 gives the optimized procedure.

The first time the optimized algorithm encounters a vertex with keys in the range, it performs two recursive calls. In all other cases, it performs only a single recursive call for non-

Algorithm 1 Naively computing root labels of clamped Merkle-trees.

Require: $x < y, t \neq \texttt{nil}$

```
1:  procedure CLAMPED_LBL(t, x, y)
2:      if t = nil then
3:          return H([])
4:      else
5:          start ← min({i : t.keys[i] ≥ x}, default t.len)
6:          end ← max({i : t.keys[i] < y}, default −1)
7:          if start > end then
8:              return CLAMPED_LBL(t.children[start], x, y)
9:          else
10:             s ← CLAMPED_LBL(t.children[start], x, y)
11:             s ← s · H(t.keys[start])
12:             for i = start + 1; i ≤ end; i++ do
13:                 s ← s · t.children[start + i].label
14:                 s ← s · H(t.keys[start + i])
15:             end for
16:             s ← s· CLAMPED_LBL(
                    t.children[end + 1], x, y
                )
17:             return H(s)
18:         end if
19:     end if
20: end procedure
```

empty trees. Hence, the number of function calls is upper-bounded by twice the height of the tree. This puts the running time and space complexity in $\mathcal{O}(\log(n))$ with high probability (assuming a data structure whose height is logarithmic in n with high probability).

V. Conclusion

We have described how to efficiently compute fingerprints for arbitary ranges in a set when given a Merkle-tree representation of that set. This allows for efficient implementation of range-based set reconciliation without relying on non-standard, homomorphic hash functions. We have further argued that this approach to reconciliation is strictly more efficient than reconciliation based on the merkle-search-trees of Auvolat and Taïani.

We have characterized the families of data structures that result in unique range-fingerprints as the history-independent, clamping-invariant search trees. Most practically useful history-independent set data structures fall into this category.

Degenerate instances of these data structures affect the complexity of fingerprint computation, but both the communication complexity and the number of roundtrips in set reconciliation remain unaffected. Range-based set reconciliation backed by homomorphic fingerprint computations, in contrast, can use self-balancing set data structures and is thus completely impervious to malicious input data. It also affords nodes to freely select their backing data structure. Weighing these advantages against the overhead induced by homomorphic hashing is the

Algorithm 2 Efficiently computing root labels of clamped Merkle-trees.

Require: $x < y, t \neq \texttt{nil}$

```
1:  procedure CLAMPED_LBL(
        t, x, y, skipLeast, skipGreatest
    )
2:      if t = nil then
3:          return H([])
4:      else
5:          start ← min({i : t.keys[i] ≥ x}, default t.len)
6:          end ← max({i : t.keys[i] < y}, default −1)
7:          if start > end then
8:              return CLAMPED_LBL(
                    t.children[start], x, y,
                    skipLeast, skipGreatest,
                )
9:          else
10:             if skipLeast = skipGreatest = ⊥ then
11:                 s ←CLAMPED_LBL(
                        t.children[start], x, y, ⊤, ⊤
                    )
12:             else if skipLeast = skipGreatest = ⊤ then
13:                 s ← t.children[start].label
14:             else
15:                 s ←CLAMPED_LBL(
                        t.children[start], x, y,
                        skipLeast, skipGreatest
                    )
16:             end if
17:             s ← s · H(t.keys[start])
18:             for i = start + 1; i ≤ end; i++ do
19:                 s ← s · t.children[start + i].label
20:                 s ← s · H(t.keys[start + i])
21:             end for
22:             if skipLeast = skipGreatest = ⊥ then
23:                 s ←CLAMPED_LBL(
                        t.children[end + 1], x, y, ⊤, ⊤
                    )
24:             else if skipGreatest = ⊥ then
25:                 s ←CLAMPED_LBL(
                        t.children[end + 1], x, y, ⊤, ⊥
                    )
26:             else
27:                 s ← t.children[end + 1].label
28:             end if
29:             return H(s)
30:         end if
31:     end if
32: end procedure
```

main decision that any system employing range-based set reconciliation needs to make; neither option is strictly superior to the other.

References

[1] A. Meyer, "Range-based set reconciliation," in *2023 42nd International Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2023, pp. 59–69.

[2] A. Auvolat and F. Taïani, "Merkle search trees: Efficient state-based crdts in open networks," in *2019 38th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2019, pp. 221–22 109.

[3] Y. Minsky, A. Trachtenberg, and R. Zippel, "Set reconciliation with nearly optimal communication complexity," *IEEE Transactions on Information Theory*, vol. 49, no. 9, pp. 2213–2218, 2003.

[4] D. Eppstein, M. T. Goodrich, F. Uyeda, and G. Varghese, "What's the difference? efficient set reconciliation without prior context," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 218–229, 2011.

[5] Y. Minsky and A. Trachtenberg, "Practical set reconciliation," in *40th Annual Allerton Conference on Communication, Control, and Computing*, vol. 248. Citeseer, 2002.

[6] L. Yang, Y. Gilad, and M. Alizadeh, "Practical rateless set reconciliation," in *Proceedings of the ACM SIGCOMM 2024 Conference*, 2024, pp. 595–612.

[7] R. C. Merkle, "A certified digital signature," in *Conference on the Theory and Application of Cryptology*. Springer, 1989, pp. 218–238.

[8] A. Buldas, P. Laud, and H. Lipmaa, "Accountable certificate management using undeniable attestations," in *Proceedings of the 7th ACM Conference on Computer and Communications Security*, 2000, pp. 9–17.

[9] W. Pugh and T. Teitelbaum, "Incremental computation via function caching," in *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1989, pp. 315–328.

[10] L. Snyder, "On uniquely represented data strauctures," in *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*. Los Alamitos, CA, USA: IEEE Computer Society, oct 1977, pp. 142–146. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SFCS.1977.22

[11] J. R. Driscoll, D. D. Sleator, and R. E. Tarjan, "Fully persistent lists with catenation," *Journal of the ACM (JACM)*, vol. 41, no. 5, pp. 943–959, 1994.

[12] M. Naor and V. Teague, "Anti-persistence: History independent data structures," in *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, 2001, pp. 492–501.

[13] R. Seidel and C. R. Aragon, "Randomized search trees," *Algorithmica*, vol. 16, no. 4, pp. 464–497, 1996.

[14] W. Pugh, "Skip lists: a probabilistic alternative to balanced trees," *Communications of the ACM*, vol. 33, no. 6, pp. 668–676, 1990.

[15] R. E. Tarjan, C. Levy, and S. Timmel, "Zip trees," *ACM Transactions on Algorithms (TALG)*, vol. 17, no. 4, pp. 1–12, 2021.

[16] O. Gila, M. T. Goodrich, and R. E. Tarjan, "Zip-zip trees: Making zip trees more balanced, biased, compact, or persistent," in *Algorithms and Data Structures Symposium*. Springer, 2023, pp. 474–492.

[17] D. Golovin, "B-treaps: A uniquely represented alternative to b-trees," in *International Colloquium on Automata, Languages, and Programming*. Springer, 2009, pp. 487–499.

[18] ——, "The b-skip-list: A simpler uniquely represented alternative to b-trees," *arXiv preprint arXiv:1005.0662*, 2010.

[19] R. Safavi and M. P. Seybold, "B-treaps revised: Write efficient randomized block search trees with high load," *arXiv preprint arXiv:2303.04722*, 2023.

[20] M. A. Bender, J. W. Berry, R. Johnson, T. M. Kroeger, S. McCauley, C. A. Phillips, B. Simon, S. Singh, and D. Zage, "Anti-persistence on persistent storage: History-independent sparse tables and dictionaries," in *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, 2016, pp. 289–302.

[21] X. Messeguer, "Skip trees, an alternative data structure to skip lists in a concurrent approach," *RAIRO-Theoretical Informatics and Applications*, vol. 31, no. 3, pp. 251–269, 1997.

[22] M. Spiegel and P. F. Reynolds Jr, "The dense skip tree: A cache-conscious randomized data structure," 2009.

[23] A. Boodman, R. Weinstein, E. Arvidsson, C. Masone, D. Willhite, and B. Kalman, "Prolly trees: Probabilistic b-trees," Documentation, Attic Labs, 2016, https://github.com/attic-labs/noms/blob/master/doc/intro.md#prolly-trees-probabilistic-b-trees. [Online]. Available: https://github.com/attic-labs/noms/blob/master/doc/intro.md#prolly-trees-probabilistic-b-trees

[24] C. Farmer and A. Meyer, "Geometric search trees," 2023, https://g-trees.github.io/g_trees/. [Online]. Available: https://g-trees.github.io/g_trees/