Range-Based Set Reconciliation

Aljoscha Meyer

***Abstract*—Range-based set reconciliation is a simple approach to efficiently computing the union of two sets over a network, based on recursively partitioning the sets and comparing fingerprints of the partitions to probabilistically detect whether a partition requires further work. Whereas prior presentations of this approach focus on specific fingerprinting schemes for specific use-cases, we give a more generic description and analysis in the broader context of set reconciliation. Precisely capturing the design space for fingerprinting schemes allows us to survey for cryptographically secure schemes. Furthermore, we reduce the time complexity of local computations by a logarithmic factor compared to previous publications.**

I. Introduction

Set reconciliation is the problem of computing the union of two sets that are located at two different nodes in a network; both nodes should hold the union of the two sets afterward. Exchanging the full sets redundantly transmits their intersection. Hence, we are interested in (probabilistic) solutions whose communication complexity is bounded by the size of their symmetric difference.

A classic use case for set reconciliation are epidemic [DGH+87] peer-to-peer systems for information sharing. Nodes continuously connect to randomly chosen other nodes, and reconcile their data. Over time, the ratio of fresh to old data decreases, so the size of the sets usually eclipses the size of the symmetric difference.

Another use case is that of replication and mirroring in distributed database systems. Key-value stores, for example, are simply sets of pairs of keys and values. Mirroring the state of one store to another is related to computing the union of the two mappings — the range-based set reconciliation approach can be adapted to perform mirroring instead.

There exist sophisticated protocols [EGUV11] that solve set reconciliation in a constant number of communication rounds and with communication complexity linear in the size of the symmetric difference. These impressive bounds are provably optimal [MTZ03], but computing the necessary messages requires time and space linear in the size of the local set, even if the symmetric difference is small.

These computational costs can be prohibitive for large sets; also such approaches are conceptually complex and do not discuss how maliciously crafted sets can lead to faulty reconciliation. For reliable practical deployment, we hence look for a conceptualy simple solution with low computational complexity and protection against malicious inputs. We believe that optimizing these metrics can be more important than over-optimizing communication complexity.

Range-based set reconciliation has space complexity linear in the size of the symmetric difference of the sets, and time complexity linear in the size of the symmetric difference or the logarithm of the size of the local set, whichever is greater. This comes at the cost of a logarithmic number of communication rounds, as the procedure follows a straightforward divide-and-conquer approach: the sets are sorted according to some total order, and nodes initiate reconciliation by sending the fingerprint of all their local items within a certain range. Upon receiving a pair of range delimiters and a fingerprint, a node computes the fingerprint of all of its own local items within that range. If the fingerprints match, the range has been successfully reconciled. Otherwise the node splits the range into smaller subranges, and initiates reconciliation for these new ranges. Whenever a node receives the fingerprint of the empty set, it transmits all its local items within that range to its peer.

If the fingerprint of a set can be computed by associatively combining the fingerprints of its members, e.g., the exclusive or of hashes of all members, we can compute it efficiently. We store the set as a balanced search tree, every vertex labeled by the fingerprint of its subtree. Maintaining the labels takes no more time asymptotically than maintaining the tree structure itself, and we can compute the fingerprint for any subrange by traversing the tree in logarithmic time.

This approach appears in the literature as a building block for larger projects ([CEG+99] Section 3.6, [SYW+17] Section II.A[1]), but neither studies it as a viable approach to set reconciliation in its own right. To the best of our knowledge, there is no literature specifically dedicated to the range-based approach. We fill this gap with a comprehensive overview.

Beyond a more general and precise formulation and a more detailed complexity analysis than prior work, we make several new contributions:

- we reduce the time complexity of successive fingerprint computations by a logarithmic factor, while using only a constant amount of space,
- we give an algebraic characterization of suitable fingerprint functions, and
- we survey suitable cryptographic fingerprints.

The organization of this article is as follows. We review related work in section II. We state the protocol for range-based reconciliation in section III and analyze its complexity. In section IV we examine possible choices of fingerprints and show how to compute them efficiently. In section V we examine how malicious actors can influence reconciliation and survey secure fingerprint functions that can protect against this, before concluding in section VI.

[1]We cite a survey because there is no standalone publication on CCNx 0.8 Sync. The survey refers to online documentation at https://github.com/ProjectCCNx/ccnx/blob/master/doc/technical/SynchronizationProtocol.txt

II. Related Work

Prior mentions of range-based set reconciliation [CEG+99][SYW+17] discuss the algorithm only superficially. Our work improves time- and space complexities, captures the full space of possible fingerprint functions, considers collision resistance, and embeds it in the larger context of set reconciliation.

Most reconciliation literature focuses on reconciliation in a single communication round, at the price of high computational costs. None of the prior work considers maliciously crafted inputs. In the following discussion, we assume nodes $\mathcal{X}_0, \mathcal{X}_1$ holding sets $X_0, X_1 \subseteq U$ respectively. $n_\triangle$ is the size of the symmetric difference of X_0 and X_1.

The seminal work on set reconciliation introduces *characteristic polynomial interpolation* (CPI) [MTZ03]. Given an approximation of $n_\triangle$, the total number of transmitted bits is proportional to $n_\triangle$, which is more efficient than the range-based approach. The required interpolation of polynomials is reduced to performing Gaussian elimination however, which takes $\mathcal{O}(n_\triangle^3)$ time.

The authors further propose a strategy for approximating $n_\triangle$ over a logarithmic number of communication rounds. CPI then requires the same number of roundtrips as range-based reconciliation, but at higher computational complexity.

Bloom filters [Blo70] are a probabilistic data structures for set membership queries. *Invertible bloom lookup tables* [GM11] (IBLTs) further support listing all items stored in the data structure. By allowing for difference computations on IBLTs, the *Difference Digest* [EGUV11] enables their use for set reconciliation. Creating the required IBLT requires $\mathcal{O}(|X_i|)$ time for node $\mathcal{X}_i$ and $\mathcal{O}(n_\triangle)$ space.

Bounding the error probabilities of the IBLT operations requires a prior estimate of $n_\triangle$. The authors present a single-message estimation protocol based on IBLTs. The size of the message is in $\mathcal{O}(\log(|U|))$. Both creating and processing the message requires $\mathcal{O}(X_i)$ time and $\mathcal{O}(\log(|U|))$ space.

Overall, the IBLT approach achieves set reconciliation in a single round trip, using only $\mathcal{O}(n_\triangle + \log(|U|))$ bits. The computational cost is however linear in the size of the sets, and the space requirements for the computation are in $\mathcal{O}(n_\triangle)$.

This work has spawned several other approaches with a constant number of roundtrips and small message size at the cost of at least linear computation time and computation space requirements: transmitting the nodes of a patricia tree in a bloom filter [BCM02], estimating $n_\triangle$ with bloom filters prior to CPI [TZX+11], using counting bloom filters [GL12], using cuckoo filters [LGR+19], or combining IPLTs with regular bloom filters to reduce the message size [OAL+19].

Partition reconciliation [MT02] reconciles in a logarithmic number of rounds to reduce computational load. It attempts CPI for successively smaller subsets, succeeding once the difference between two subsets is sufficiently small.

This approach eliminates CPI's cubic scaling of the computation time in $n_\triangle$. Reconciliation messages are precomputed in a *partition tree* where a parent node includes subranges as children. Given a balanced partition tree, the reconciliation procedure has the same asymptotic worst-case complexity bounds as ours. The tree is not self-balancing however, so as reconcilliation adds more items to the set, the time complexity of local computations can degrade to $\mathcal{O}(n)$. Furthermore, the tree is specific to a particular choice of evaluation points and control points for the characteristic polynomial. When reusing the tree across multiple reconciliation sessions, these points have to be fixed in advance. This could allow an attacker to craft sets for which failed reconciliation is not detected.

The *Merkle Search Tree CRDT* [AT19] is similar to our approach, but using a pseudorandom tree construction. This enforces a rigid set representation and can degrade to a linear number of communication rounds for maliciously crafted sets. Our approach allows for free choice of search tree structure and number of recursion steps, guaranteeing a logarithmic number of communication rounds in the worst case.

III. Recursive Reconciliation

We now describe the communication side of a range-based set reconciliation session, while deferring the details of fingerprint computations to section IV.

We consider two nodes $\mathcal{X}_0$ and $\mathcal{X}_1$, connected via a bidirectional, reliable, ordered communication channel. They can send an arbitrary number of bits in a single, unit-length communication round. The nodes initially hold sets X_0 and X_1 respectively. After reconciliation, both will hold $X_0 \cup X_1$.

X_0 and X_1 are drawn from some universe U, which is ordered by a total order $\preceq$. To allow meaningful statements about communication complexity, we require encodings of bounded size for the members of U, i.e., we require U to be finite. This can always be achieved in practice by reconciling hashes of items rather than items themselves.

We finally fix a fingerprinting function $\mathrm{fp} : \mathcal{P}(U) \to H$ that maps subsets of U into some finite codomain H with negligible probability of collisions.

We use the following notation and terminology for *ranges*:

Definition 1. Let $S \subseteq U$ and $x, y \in U$.

The *range from x to y in S*, denoted by $[x, y)_S$, is the set $\{s \in S \mid x \preceq s \prec y\}$ if $x \prec y$, or $S \setminus [y, x)_S$ if $y \prec x$, or simply S if $x = y$. We call x the *lower boundary* and y the *upper boundary* of the range (even if $y \prec x$).

A. Protocol Description

In a given communication round, a node receives information about some subranges of the sets to be reconciled. For each such subrange, it receives either a fingerprint of the items the other node has in that range, or it receives those items directly. The node answers with information about new ranges; partitioning ranges into subranges if neither the received fingerprint matches the fingerprint of the local items within that range nor the range contains few enough items to transmit them directly. We precisely specify the vocabulary by which nodes exchange information in definition 2:

Definition 2. Let $\mathcal{X}_i$ be a node that holds a set $X_i \subseteq U$.

A *range fingerprint* is a triplet $\big(x, y, \mathrm{fp}([x,y)_{X_i})\big)$ for $x, y \in U$. It conveys the fingerprint over the range from x to y in X_i.

A *range item set* is a four-tuple (x, y, S, b) for $x, y \in U$, $S \subseteq [x,y)_{X_i}$, and $b \in \{0,1\}$. It transmits items within the range from x to y in X_i. The boolean signals whether the other node should respond with its local items from x to y ($b = 0$), or whether these have already been received ($b = 1$).

A *message part* is either a range fingerprint or a range item set. A *message* is a nonempty sequence of message parts.

A node initiates reconciliation by sending a message containing a single range fingerprint $\big(x, x, \mathrm{fp}([x,x)_{X_i})\big)$ for some $x \in U$. The nodes then run protocol 1:

Protocol 1 (Range-Based Set Reconciliation)**.** Let $\mathcal{X}_i$ be a node that holds a set $X_i \subseteq U$ and that has just received a message. It then performs the following actions:

1) Initialize an empty response.
2) For every range item set (x, y, S, b) in the message, add S to X_i. Add the range item set $(x, y, [x,y)_{X_i} \setminus S, 1)$ to the response unless that set is empty or $b = 0$.
3) For every range fingerprint $\big(x, y, \mathrm{fp}([x,y)_{X_j})\big)$ in the message, do one of the following:
   Case 1, Equal Fingerprints:
   If $\mathrm{fp}([x,y)_{X_j}) = \mathrm{fp}([x,y)_{X_i})$, do nothing.
   Case 2, Recursion Anchor: You may add the range items set $(x, y, [x,y)_{X_i} 0)$ to the response. If $|[x,y)_{X_i}| \leq 1$ or $\mathrm{fp}([x,y)_{X_j}) = \mathrm{fp}(\emptyset)$, always do so.
   Case 3, Recurse: Otherwise, select $m_0 := x \prec m_1 \prec \ldots \prec m_k := y$ from U, $k \geq 2$, such that $|[m_l, m_{l+1})_{X_i}| < |[x,y)_{X_i}|$ for all $0 \leq l < k$. For all $0 \leq l < k$ add either the range fingerprint $\big(m_l, m_{l+1}, \mathrm{fp}([m_l, m_{l+1})_{X_i})\big)$ or the range item set $(m_l, m_{l+1}, [m_l, m_{l+1})_{X_i} 0)$ to the response.
4) If the accumulated response is nonempty, send it. Otherwise terminate successfully.

Figure 1 visualizes an example run of the protocol.

The recursion anchor essentially runs the trivial reconciliation protocol of simply exchanging sets. One could use more sophisticated single-roundtrip protocols instead. For simplicity, we only discuss protocol 1 as presented, but range-based reconciliation can essentially function as a general preprocessing mechanism for bounding the computational complexity of arbitrary reconciliation protocols.

B. Range-Base Set Mirroring

Set mirroring is the problem of efficiently transferring a set from a *primary* node to a *replica* node, utilizing similarities between the primary's set and the outdated version on the replica. To do so, the primary node simply runs protocol 1 as-is, and the replica node runs a slightly modified version: whenever it receives a range item set (x, y, S, b), it removes from its local set X_i all items in $[x,y)_{X_i} \setminus S$; and whenever it sends a range item set itself, it sends the empty set.

We restrict our presentation to reconciliation only, but all our results apply to this mirroring technique as well.

C. Protocol Properties

Protocol 1 leaves open whether and into how many subranges to split large range fingerprints. In particular, two nodes can reconcile a set even when using different strategies for deciding when and how to recurse.

1) Termination and Correctness: Termination of protocol 1 follow from an inductive argument. Small ranges and ranges with matching fingerprints are handled within a constant number of communication rounds, and the largest subrange in round i strictly smaller than that of round $i-1$.

Correctness follows inductively as well. If fingerprints do not collide, ranges with equal fingerprints are reconciled correctly. Sending a range item set and receiving the response leads to both nodes storing the union of all items within that range. The subranges in the recursive case are reconciled correctly by induction hypothesis. And because the subranges cover the original range, this reconciles the original range.

2) Complexity: The protocol allows responding to range fingerprints with a range item set, even if that set is arbitrarily large. For a meaningful complexity analysis, we restrict nodes to send a range item set only if the number of its items in the range is less than or equal to some threshold $t \in \mathbb{N}^+$. Large t reduce the number of roundtrips, but increase the probability of sending an item the other node already has.

We similarly assume that nodes split ranges into at most $b \in \mathbb{N}, b \geq 2$ subranges when recursing. Large b reduce the number of roundtrips but transmit more fingerprints.

In the following complexity analyses, n_0 and n_1 denote the number of items held by nodes $\mathcal{X}_0$ and $\mathcal{X}_1$ respectively. We let $n := n_0 + n_1$, $n_{\min} := \min(n_0, n_1)$ and $n_\triangle := |(X_0 \cup X_1) \setminus (X_0 \cap X_1|$.

Observe that the range fingerprints of a protocol run form a rooted, b-ary *communication tree*, compare fig. 1. When a leaf of the tree is reached, an exchange of range item sets follows.

Node $\mathcal{X}_i$ can perform at most $\lceil \log_b(n_i) \rceil$ recursion steps, so the overall height of the communication tree is bounded by $2 \cdot \lceil \log_b(n_{\min}) \rceil$.

The parameter t influences the height of the tree. For $t = 1$, the protocol recurses as far as possible. For $t = b$, the last level of recursion is cut off, for $t = b^2$ the last two levels, and so on. Overall, the height of the tree is reduced by $\lfloor \log_b(t) \rfloor$.

The total number of communication rounds is bounded by the maximum number of times that nodes recurse, followed by two rounds of exchanging range items sets. This corresponds to two plus the height of the tree, so $2 + 2 \cdot \lceil \log_b(n_{\min}) \rceil - \lfloor \log_b(t) \rfloor \in \mathcal{O}(\log(n))$.

This number cannot be bounded by $n_\triangle$, as witnessed, e.g., by problem instances where one node is missing exactly one item compared to the other node (fig. 2). In such an instance, $b-1$ branches in each recursion step result in equal fingerprints, but the one branch that does continue reaches the recursion anchor only after the full number of rounds.

Every item in the symmetric difference can be responsible for only one such a path from the root to a leaf, a fact we can use to bound the number of transmitted bits. Range fingerprints and range item sets can be encoded using $\mathcal{O}(1)$ bits (because

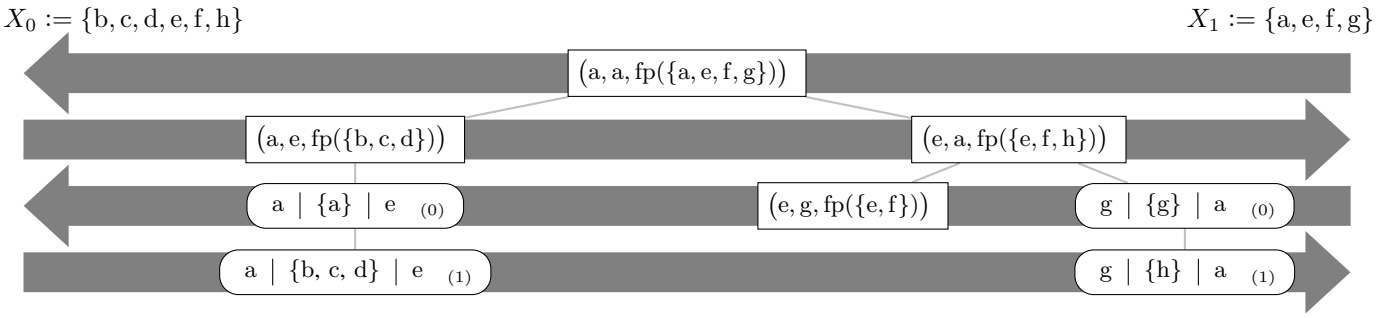

Fig. 1. An example run of protocol 1. In this and further examples, $U := \{\mathrm{a,b,c,d,e,f,g,h}\}$, and $\preceq$ orders the universe alphabetically. Range fingerprints have sharp corners, range item sets have rounded corners. The arrows in the background indicate sending and receiving node.
$\mathcal{X}_1$ initiates reconciliation over the full universe, transmitting the fingerprint of X_1.
Upon receiving this range fingerprint, $\mathcal{X}_0$ locally computes $\mathrm{fp}([\mathrm{a,a})_{X_0})$. Because the result does not match the received fingerprint, $\mathcal{X}_0$ splits X_0 into two parts of equal size and transmits range fingerprints for these subranges.
In the third round, $\mathcal{X}_1$ locally computes fingerprints for the two received ranges, but neither matches. Because $|[\mathrm{a,e})_{X_1}| \leq 1$, $\mathcal{X}_1$ transmits the corresponding range items set $(\mathrm{a,e,a},0)$. For the other range, $|[\mathrm{e,a})_{X_1}| > 1$ however, so another recursion step can be performed. After splitting the range, the lower range contains enough items to send another range fingerprint. The upper range however only contains one item, thus $\mathcal{X}_1$ handles it by sending a range item set.
In the final communication round, $\mathcal{X}_0$ receives the two range item sets and answers with the items it holds within those ranges. For the range fingerprint $(\mathrm{e,g,fp}([\mathrm{e,g})_{X_1}))$, it computes an equal fingerprint $\mathrm{fp}([\mathrm{e,g})_{X_0})$, so no response is required.

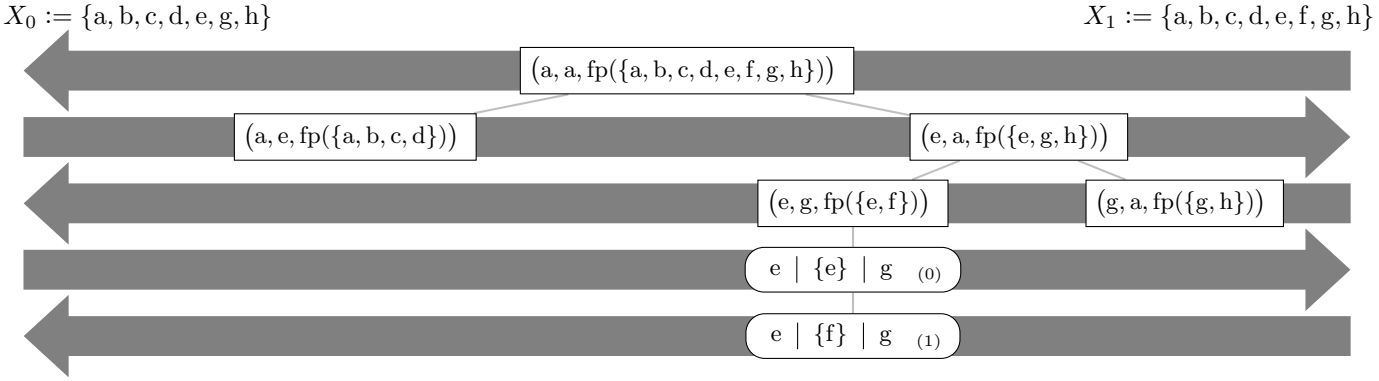

Fig. 2. An example run of the protocol that takes the greatest possible number of rounds, even though $n_\triangle = 1$. $b := 2, t := 1$.

we assume U to be finite and limit the size of range item sets to t). As the height of the tree is in $\mathcal{O}(\log(n))$, we get an overall bound of $\mathcal{O}(n_\triangle \cdot \log(n))$ bits.

These paths to $n_\triangle$ many leaves overlap however, and every vertex of the communication tree only contributes $\mathcal{O}(1)$ bits to the reconciliation session. As we have at most $\mathcal{O}(n)$ nodes in the tree, the overall number of bits is at most $\mathcal{O}(\min(n_\triangle \cdot \log(n), n))$. The case of transmitting $\mathcal{O}(n)$ bits occurs if one node is lacking every second item of the other node, see fig. 3.

In terms of bits per item, this is efficient however: since $n_\triangle$ is within a constant factor of n, we transmit $\mathcal{O}(1)$ bits per item that needs synchronization. The least efficient scenario from this point of view is that of $n_\triangle = 1$, where we send $\mathcal{O}(\log(n))$ bits per item.

The size of each message is proportional to the number of vertices of the the corresponding depth in the communication tree, which is at most $n_\triangle$. This affects the space complexity for the participating nodes. If each node stores full messages in memory, the space complexity then is in $\mathcal{O}(n_\triangle)$ — we will carefully choose fingerprints such that nodes can successively process each message part in $\mathcal{O}(1)$ space.

Alternatively, nodes can split messages and transmit only a bounded number of message parts at a time. Once a message fragment has been processed and the corresponding, newly computed response message parts have been sent, the other node can transmit the next message parts.

If both nodes follow this strategy but allocate space for only a constant number of message parts, the protocol can deadlock. The number of message parts in a response can increase across rounds. When it exceeds the total space capacity of both nodes, no node is able to receive or transmit more data.

If however one node can allocate $\mathcal{O}(n_\triangle)$ space to buffer both an incoming and an outgoing message, the other node can operate within constant space. This does lead to a higher number of communication rounds, as the buffering node needs to split messages into chunks of constant size and wait for confirmation before transmitting the next one. The number of communication rounds becomes proportional to the number of bits to transmit, so it is in $\mathcal{O}(\min(n_\triangle \cdot \log(n), n))$.

This analysis seems unfavorable, but note that only a constant number of bits needs to be transmitted in every single communication round. Our analysis of the setting with

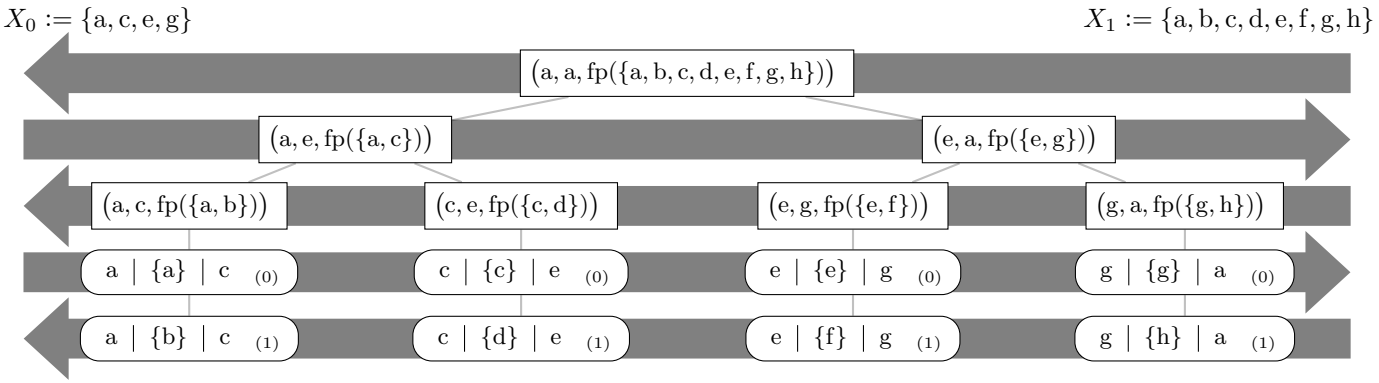

Fig. 3. An example run of the protocol that requires transmitting the maximum amount of bytes. $b := 2, t := 1$.

unbounded buffering capabilities assumes that messages of arbitrary size can be transmitted in a single communication round. In a more realistic networking model with limited bandwidth, *every* protocols requires rounds proportional to the number of bits it sends. All set reconciliation protocols transmit at least $\Omega(n_\triangle)$ bits [MTZ03], so range-based set reconciliation is within a logarithmic factor of the optimal number of communication rounds under this model, whether with bounded or unbounded memory.

IV. Fingerprint Computation

We now examine the time and space complexity of the computations each node performs during a reconciliation session. We consider a model where each node, in addition to working memory for performing computations, maintains an auxiliary data structure across computations. The node updates its auxiliary data structure whenever its set changes, and it can read from this data structure during its fingerprint computations.

This model is motivated by the fact that each node already has to update an external data structure — its set — between message computations. Overall, we are interested in the time and memory it takes to update the auxiliary datastructure to reflect changes to the set, the space consumed by the auxiliary data structure, the time it takes to compute each message during a reconciliation session, and the space this requires.

Assuming the set is stored as a balanced search tree, it consumes a linear amount of space, and adding or removing individual items requires $\mathcal{O}(\log(n))$ time. This gives us a free complexity budget to work with; if our auxiliary data structure requires the same amount of time and space, it does not impact the asymptotic performance of our approach. We will, in fact, extend the tree representation of the set by storing additional data in each vertex.

A. Monoid Trees

When computing messages, a node must efficiently compute the fingerprint of all items it holds within arbitrary ranges. We now consider a general family of functions that map ranges within a set to some codomain, and that can be efficiently computed with an auxiliary tree structure. These functions reduce a finite set to a single value according to a monoid.

Definition 3. Let M be a set, $\oplus : M \times M \to M$, and $\mathbb{0} \in M$.

We call $(M, \oplus, \mathbb{0})$ a *monoid* if it satisfies two properties:

associativity: for all $x, y, z \in M$: $(x \oplus y) \oplus z = x \oplus (y \oplus z)$,
neutral element: for all $x \in M$: $\mathbb{0} \oplus x = x = x \oplus \mathbb{0}$.

Definition 4 (Lifted Function)**.** Let U be a set, $\preceq$ a linear order on U, $\mathcal{M} = (M, \oplus, \mathbb{0})$ a monoid, and $\mathrm{f} : U \to M$.

We *lift* f *to finite sets via* $\mathcal{M}$ to obtain $\mathrm{lift}_\mathrm{f}^\mathcal{M} : \mathcal{P}(U) \rightharpoonup M$ with:

$$\mathrm{lift}_\mathrm{f}^\mathcal{M}(\emptyset) := \mathbb{0},$$
$$\mathrm{lift}_\mathrm{f}^\mathcal{M}(S) := \mathrm{f}\big(\min(S)\big) \oplus \mathrm{lift}_\mathrm{f}^\mathcal{M}\big(S \setminus \{\min(S)\}\big).$$

In other words, if $S = \{s_1, s_2, \ldots, s_{|S|}\}$ with $s_1 \prec s_2 \prec \cdots \prec s_{|S|}$, then $\mathrm{lift}_\mathrm{f}^\mathcal{M}(S) = \mathrm{f}(s_1) \oplus \mathrm{f}(s_2) \oplus \cdots \oplus \mathrm{f}(s_{|S|})$.

Let, for example, U be an arbitrary set, $\mathcal{N}$ be the monoid of natural numbers under addition, and let $\lambda x.1$ map any x to the number 1, then $\mathrm{lift}_{\lambda x.1}^\mathcal{N}(S) = |S|$ for every finite $S \subseteq U$.

We can efficiently compute lifted functions by maintaining a labeled tree.

Definition 5. A *binary tree* t over a universe U is either the empty tree `nil`, or a triplet of a left subtree $t.\mathtt{l}$, a value $t.\mathtt{v} \in U$, and a right subtree $t.\mathtt{r}$.

We say t is a *vertex* if $t \neq \mathtt{nil}$; we denote the set of all vertices in a tree t by $\mathrm{V}(t)$. We say a vertex t is a *leaf* if $t.\mathtt{l} = \mathtt{nil} = t.\mathtt{r}$, otherwise, t is *internal*.

Let $\preceq$ be a total order. We say t is a *search tree* (with respect to $\preceq$) if $t = \mathtt{nil}$, or if $t.\mathtt{v}$ is greater than the greatest value in $t.\mathtt{l}$, $t.\mathtt{v}$ is less than the least value in $t.\mathtt{r}$, and every subtree of t is also a search tree.

Toward efficient computation of functions of the form $\mathrm{lift}_\mathrm{f}^\mathcal{M}$, we label a binary search tree t:

Definition 6. Let U be a set, $S \subset U$ a finite set, $\preceq$ a linear order on U, $\mathcal{M} := (M, \oplus, \mathbb{0})$ a monoid, $\mathrm{f} : U \to M$, and let t be a binary search tree on S.

We define a *monoid labeling function* $\mathrm{label}_{\mathrm{f}}^{\mathcal{M}} : \mathrm{V}(t) \to M$:
$\mathrm{label}_{\mathrm{f}}^{\mathcal{M}}(t) := \mathbb{0}$ if $t = \mathtt{nil}$,
$\mathrm{label}_{\mathrm{f}}^{\mathcal{M}}(t) := \mathrm{label}_{\mathrm{f}}^{\mathcal{M}}(t.\mathtt{l}) \oplus \mathrm{f}(t.\mathtt{v}) \oplus \mathrm{label}_{\mathrm{f}}^{\mathcal{M}}(t.\mathtt{r})$ otherwise.
We call a tree labeled by $\mathrm{label}_{\mathrm{f}}^{\mathcal{M}} : \mathrm{V}(t) \to M$ a *monoid tree*.

Observe that $\mathrm{label}_{\mathrm{f}}^{\mathcal{M}}(t) = \mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}(\mathrm{V}(t))$ for every binary search tree t. The exact shape of the tree dictates the grouping of how to apply $\oplus$ to several values; different groupings yield the same result, as $\oplus$ is associative. Because we label a search tree, $\oplus$ is always applied to the items in ascending order, regardless of the tree shape.

Returning to our previous example, labeling a tree with $\mathrm{label}_{\lambda \mathrm{x}.1}^{\mathcal{N}}$ annotates each subtree with its size, i.e., this yields the order statistic trees [CLRS22]. The labels can be kept updated in a self-balancing search tree implementation without changing the asymptotic complexity of insertion and deletion for both $\mathrm{label}_{\lambda \mathrm{x}.1}^{\mathcal{N}}$ in particular and for arbitrary $\mathrm{label}_{\mathrm{f}}^{\mathcal{M}}$ functions in general [CLRS22].

Every monoid labeling function can be used for efficiently maintaining labels in the tree, but are these the only such functions? To answer this, we give a homomorphism-flavored characterization of candidate functions: given the images of two sets, one containing only items strictly less than those in the other, the image of the union of these sets should be the same as combining the original images in some monoid. This ensures that vertex labels can be updated by considering only the labels of their children and the image of their value.

Definition 7 (Tree-Friendly Function)**.** Let U be a set, $\preceq$ a linear order on U, $\mathcal{M} := (M, \oplus, \mathbb{0})$ a monoid, and $\mathrm{f} : \mathcal{P}(U) \rightharpoonup M$ a partial function mapping all finite subsets of U into M.

We call f a *tree-friendly function* if for all finite sets $S_0, S_1 \in \mathcal{P}(U)$ such that $\max(S_0) \prec \min(S_1)$, we have $\mathrm{f}(S_0 \cup S_1) = \mathrm{f}(S_0) \oplus \mathrm{f}(S_1)$.

This definition captures exactly the functions of form $\mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}$, as shown in the following propositions:

Proposition 1. Let U be a set, $\preceq$ a linear order on U, $\mathcal{M} := (M, \oplus, \mathbb{0})$ a monoid, and $\mathrm{f} : U \to M$.

Then $\mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}$ is a tree-friendly function.

Proof. Let $S_0, S_1 \in \mathcal{P}(U)$ be finite sets such that $\max(S_0) \prec \min(S_1)$. Then:

$$\begin{aligned}\mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}(S_0 \cup S_1) &= \bigoplus_{\substack{s_i \in S_0 \cup S_1, \\ \text{ascending}}} \mathrm{f}(s_i) \\ &= \bigoplus_{\substack{s_i \in S_0, \\ \text{ascending}}} \mathrm{f}(s_i) \oplus \bigoplus_{\substack{s_i \in S_1, \\ \text{ascending}}} \mathrm{f}(s_i) \\ &= \mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}(S_0) \oplus \mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}(S_1)\end{aligned}$$

□

Proposition 2. Let U be a set, $\preceq$ a linear order on U, $\mathcal{M} := (M, \oplus, \mathbb{0})$ a monoid, and $\mathrm{g} : \mathcal{P}(U) \rightharpoonup M$ a tree-friendly function.

Then there exists $\mathrm{f} : U \to M$ such that $\mathrm{g} = \mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}$.

Proof. Define $\mathrm{f} : U \to M$ as $\mathrm{f}(u) := \mathrm{g}(\{u\})$. We show by induction on the size of $S \subseteq U$ that $\mathrm{g}(S) = \mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}(S)$.

IB: If $S = \emptyset$, then $\mathrm{g}(S) = \mathbb{0} = \mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}(S)$. Suppose that $\mathrm{g}(\emptyset) \neq \mathbb{0}$, this would contradict the fact that for all $x \in U$ we have $\mathrm{g}(\{x\}) = \mathrm{g}(\{x\}) \oplus \mathrm{g}(\emptyset) = \mathrm{g}(\emptyset) \oplus \mathrm{g}(\{x\})$, which holds because $\{x\} = \{x\} \cup \emptyset = \emptyset \cup \{x\}$ and g is a tree-friendly function. If $S = \{x\}$, then $\mathrm{g}(S) = \mathrm{f}(x) = \mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}(S)$.

IH: For all sets T with $|T| = n$ it holds that $\mathrm{g}(T) = \mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}(T)$.

IS: Let $S \subseteq U$ with $|S| = n + 1$, then:

$$\begin{aligned}\mathrm{g}(S) &= \mathrm{g}(\{\min(S)\}) \oplus \mathrm{g}(S \setminus \{\min(S)\}) \\ &\overset{\text{IH}}{=} \mathrm{g}(\{\min(S)\}) \oplus \mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}(S \setminus \{\min(S)\}) \\ &= \mathrm{f}(\min(S)) \oplus \mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}(S \setminus \{\min(S)\}) \\ &= \mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}(S)\end{aligned}$$

As g is only defined over finite inputs, we thus have $\mathrm{g} = \mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}$. □

B. Range Computations

Given a monoid tree t, we can compute $\mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}\big([x, y)_{\mathrm{V}(t)}\big)$ efficiently for *any* $x, y \in U$. Without loss of generality, we can assume that $x \prec y$, as we can otherwise compute $\mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}\big([\min(\mathrm{V}(t)), y)_{\mathrm{V}(t)}\big) \oplus \mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}\big([x, \max(\mathrm{V}(t)))_{\mathrm{V}(t)}\big)$.

Intuitively speaking, we trace paths from (the root of) t to x and y, and then we need to combine all values in the "area between those paths". The labels of the children of the vertices along these paths which lie within that area summarize this information, so it suffices to combine information from the out-neighborhood of the paths. If the tree is balanced, we thus only need to combine a logarithmic number of values.

Algorithm 1 gives the precise definition of the algorithm. First, we search for the first vertex reachable from the root that lies within the range (the procedure FIND_INITIAL). If no such vertex exists, the set contains no items within the range. If such an initial vertex exists however, it is necessarily unique. Assume toward a contradiction that there are two distinct such vertices $a \prec b$. Because t is a search tree, the least common ancestor of a and b is also in the range, and it is closer to the root than both a and b, a contradiction. Consequently, all items within the range are descendents of the initial vertex, which we name $init$.

Because all items within the range are descendents of $init$ and $x \preceq init \prec y$, we have that

$$\begin{aligned}[x, y)_{\mathrm{V}(t)} = \{z \in \mathrm{V}(init.\mathtt{l}) \mid z \succeq x\} \dot\cup \\ \{init.\mathtt{v}\} \dot\cup \{z \in \mathrm{V}(init.\mathtt{r}) \mid z \prec y\},\end{aligned}$$

and hence

$$\begin{aligned}\mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}\big([x, y)_{\mathrm{V}(t)}\big) = \mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}\big(\{z \in \mathrm{V}(init.\mathtt{l}) \mid z \succeq x\}\big) \oplus \\ \mathrm{f}(init.\mathtt{v}) \oplus \mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}\big(\{z \in \mathrm{V}(init.\mathtt{r}) \mid z \prec y\}\big).\end{aligned}$$

Procedure AGGREGATE_LEFT demonstrates how to compute $\mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}\big(\{z \in \mathrm{V}(init.\mathtt{l}) \mid z \succeq x\}\big)$: starting from the initial vertex, we search for x, accumulating the labels of

all right children, as well as the monoid values that correspond to those vertices on the search path that are greater than or equal to x. Similarly, procedure AGGREGATE_RIGHT computes $\mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}\big(\{z \in \mathrm{V}(init.\mathtt{r}) \mid z \prec y\}\big)$. Figure 4 depicts an example run.

Overall, algorithm 1 searches for two items in a search tree, along with some constant-time computations in each search step. If t is balanced, the time complexity is thus in $\mathcal{O}(\log(|\mathrm{V}(t)|))$. As the algorithm requires no dynamic memory allocation and is not recursive, its space complexity is in $\mathcal{O}(1)$.

Because associativity guarantees equal results regardless of the precise shape of the tree, implementations need not restrict themselves to binary trees. The algorithm can be extended to the practically more efficient B-trees [BM02], for example.

C. Monoidal Fingerprints

Now that we have characterized a general family of functions that admit efficient computation on ranges, we can turn back to the range-based set reconciliation approach. Protocol 1 works by recursively testing fingerprints for equality. For our purposes, we can define a fingerprint or hash function as follows:

Definition 8. A *hash function* is a function $\mathrm{h} : U \to D$ with a finite codomain, such that, for randomly chosen $u \in U$ and $d \in D$, the probability that $\mathrm{h}(u) = d$ is roughly[2] $\frac{1}{|D|}$. $\mathrm{h}(u)$ is called the *hash of* u, *fingerprint of* u or *digest of* u.

To efficiently compute fingerprints for arbitrary ranges, we use tree-friendly functions $\mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}$ that serve as hash functions from $\mathcal{P}(U)$. As $\mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}(\{u\})$ is equal to $\mathrm{f}(u)$, f must itself already be a hash function. Typical hash functions map values to bit strings of a certain length, i.e., the codomain is $\{0,1\}^k$ for some $k \in \mathbb{N}$. We will thus consider monoids whose elements can be represented by such bit strings.

A natural choice of the monoid universe is then $[0, 2^k)_{\mathbb{N}}$, some simple monoidal operations on this universe include bitwise xor, addition modulo 2^k, and multiplication modulo 2^k. Of these three options, multiplication is the least suitable, because multiplying any number by 0 yields 0. Consequently, for every set containing an item u with $\mathrm{f}(u) = 0$, the fingerprint of the set is 0, which clearly violates the criterion that all possible values for fingerprints occur with equal probability.

The monoid operation preserves a good distribution of fingerprints if any given fingerprint can be obtained from any particular fingerprint by combining it with some third one, i.e., if, for every $x \in M$, $\lambda y.x \oplus y$ is a bijection. Addition and xor satisfy this criterium, as does in fact every finite commutative group $\mathcal{G} = (G, \oplus, -)$: for every $x, z \in M$ there exists $y \in M$ such that $x \oplus y = z$, by choosing $y := z \oplus -x$, because then $x \oplus y = x \oplus z \oplus (-x) = x \oplus (-x) \oplus z = z$. Hence, $\lambda y.x \oplus y$ is surjective, and, because G is finite, the function is also injective.

[2]To keep our focus on set reconciliation rather than cryptography, we keep arguments about probabilities qualitative rather than quantitative at this point.

Algorithm 1 Computing $\mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}\big([x, y)_{\mathrm{V}(t)}\big)$.

Require: $x \preceq y, t \neq \mathtt{nil}$

procedure AGGREGATE_RANGE(t, x, y)
  if $x = y$ **then**
    return $\mathrm{label}_{\mathrm{f}}^{\mathcal{M}}(t)$
  end if
  $init \leftarrow$ FIND_INITIAL(t, x, y)
  if $init = \mathtt{nil}$ **then**
    return $\mathbb{0}$
  else
    $acc_l \leftarrow$ AGGREGATE_LEFT($init.\mathtt{l}, x$)
    $acc_r \leftarrow$ AGGREGATE_RIGHT($init.\mathtt{r}, y$)
    return $acc_l \oplus \mathrm{f}(init.\mathtt{v}) \oplus acc_r$
  end if
end procedure
procedure FIND_INITIAL(t, x, y)
  while `true` **do**
    if $t = \mathtt{nil}$ **then**
      return t
    else if $t.\mathtt{v} \prec x$ **then**
      $t \leftarrow t.\mathtt{r}$
    else if $t.\mathtt{v} \succeq y$ **then**
      $t \leftarrow t.\mathtt{l}$
    else
      return t
    end if
  end while
end procedure
procedure AGGREGATE_LEFT(t, x)
  $acc \leftarrow \mathbb{0}$
  while `true` **do**
    if $t = \mathtt{nil}$ **then**
      return acc
    else if $t.\mathtt{v} \prec x$ **then**
      $t \leftarrow t.\mathtt{r}$
    else if $t.\mathtt{v} = x$ **then**
      return $\mathrm{f}(t.\mathtt{v}) \oplus \mathrm{label}_{\mathrm{f}}^{\mathcal{M}}(t.\mathtt{r}) \oplus acc$
    else
      $acc \leftarrow \mathrm{f}(t.\mathtt{v}) \oplus \mathrm{label}_{\mathrm{f}}^{\mathcal{M}}(t.\mathtt{r}) \oplus acc$
      $t \leftarrow t.\mathtt{l}$
    end if
  end while
end procedure
procedure AGGREGATE_RIGHT(t, y)
  $acc \leftarrow \mathbb{0}$
  while `true` **do**
    if $t = \mathtt{nil}$ **then**
      return acc
    else if $t.\mathtt{v} \prec y$ **then**
      $acc \leftarrow acc \oplus \mathrm{label}_{\mathrm{f}}^{\mathcal{M}}(t.\mathtt{l}) \oplus \mathrm{f}(t.\mathtt{v})$
      $t \leftarrow t.\mathtt{r}$
    else if $t.\mathtt{v} = x$ **then**
      return $acc \oplus \mathrm{label}_{\mathrm{f}}^{\mathcal{M}}(t.\mathtt{l})$
    else
      $t \leftarrow t.\mathtt{l}$
    end if
  end while
end procedure

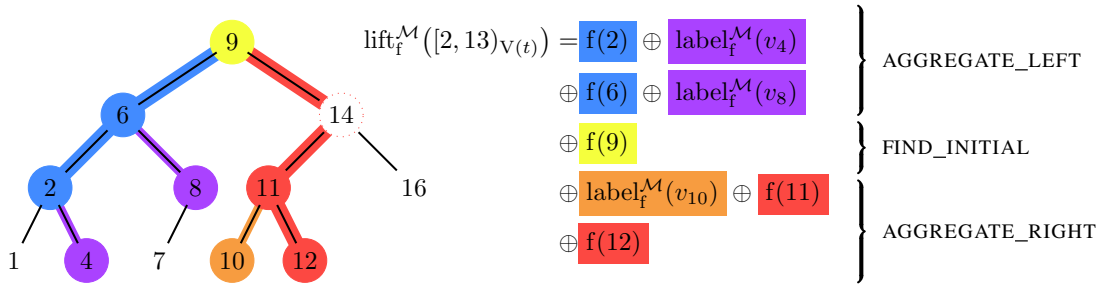

Fig. 4. Visualization of an exemplary tree traversal as performed by algorithm 1 to compute $\mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}([2,13)_{\mathrm{V}(t)})$. Notice that v_7 need not be visited, as its contribution to the accumulated value is already part of $\mathrm{label}_{\mathrm{f}}^{\mathcal{M}}(v_8)$. Notice further that the traversal visits v_{14} but ignores it, as 14 lies outside the range.

By using such a tree-friendly function $\mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}$, we can efficiently implement range-based set reconciliation. A node stores its set in a monoid tree labeled by both $\mathrm{label}_{\mathrm{f}}^{\mathcal{M}}$ and $\mathrm{label}_{\lambda \mathrm{x}.1}^{\mathcal{N}}$. On receiving a range fingerprint $\big(x, y, \mathrm{fp}([x,y)_{X_j})\big)$, the node $\mathcal{X}_i$ efficiently computes $\mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}\big([x,y)_{X_i}\big)$. If the fingerprints do not match, it computes $\mathrm{lift}_{\lambda \mathrm{x}.1}^{\mathcal{N}}\big([x,y)_{X_i}\big)$ to determine the number of items it has in the range, and uses this information for determining the sizes of the subranges to create. Finding the boundaries of those subranges amounts to looking up items by index in an order-statistic tree, and thus takes logarithmic time. All of these operations require only $\mathcal{O}(1)$ space.

Overall, the computations for processing a single range fingerprint for a local set of size n_i thus take $\mathcal{O}(\log(n_i))$ time. As a single message can contain $\mathcal{O}(n_\triangle)$ many range fingerprints, where $n_\triangle$ is the size of the symmetric difference of the sets to reconcile, the overall time complexity per communication round is in $\mathcal{O}(n_\triangle \cdot \log(n_i))$.

D. Ascending Intervals

When computing fingerprints for several ranges, we can reduce the overall time complexity if the ranges are sorted by their lower boundaries in ascending order. We can accumulate labels while traversing from the lower boundary of each range to its upper boundary; then we traverse to the lower boundary of the next range, ready to process it. In this traversal, any edge is traversed at most twice, giving an upper bound for processing a single message of $\mathcal{O}(n)$.

Processing any individual range this way still requires $\mathcal{O}(\log(n))$ time, since the maximum distance between two vertices in a balanced tree on n vertices is in $\mathcal{O}(\log(n))$. Overall, the time complexity for a single communication round is hence in $\mathcal{O}(\min(n, n_\triangle \cdot \log(n)))$. This can result in a logarithmic speed-up compared to prior discussion of range-based set reconciliation ([CEG+99][SYW+17]).

AGGREGATE_UNTIL (algorithm 2) implements this traversal as a procedure that takes the boundaries of a single range and the vertex that stores the lower boundary as arguments, and returns both the aggregated monoidal value of the range, and the vertex that stores the least value that is greater than the upper boundary. This vertex can be used as the starting point for the next invocation of the procedure to find the lower boundary of the next range. If no such vertex exists, the procedure returns `nil` in its place, and the aggregated value for all following ranges is known to be $\mathbb{0}$.

The path from some lower boundary x to some upper boundary y consists of some (possibly zero) upward steps from x, and then some (possibly zero) downward steps toward y. In order to compute this path in constant space and time per step, we add to each vertex v a reference $v.\mathtt{p}$ to its parent (nil for the root), and the maximal value stored in its subtree, denoted as $v.\mathtt{m}$. The traversal begins by following parent references until reaching the root of a subtree t that contains a value greater than or equal to y (AGGREGATE_UP), which we can efficiently detect by comparing $t.\mathtt{m}$ against y. The successive traversal (AGGREGATE_DOWN) for finding the least value above the range terminates upon reaching a vertex t with $t.\mathtt{v} \succeq y$ whose left subtree is fully contained within the range, i.e., with $t.\mathtt{l}.\mathtt{m} \prec y$.

V. Adversarial Environments

Protocol 1 requires that sets with equal fingerprints are actually equal. Reconciliation becomes faulty if it involves unequal sets with equal fingerprints. If sets map into randomly distributed hashes from a sufficiently large universe, the probability of collisions becomes negligible for randomly distributed input sets. Random distribution of input sets is a strong assumption however. In this section, we examine how to protect reconciliation against adversarially chosen sets.

A. Impact of Hash Collisions

We distinguish between *active* adversaries who can who can select the sets to be reconciled, and *passive* adversaries who needs to find and cause collisions in existing sets. If fingerprints are bit strings of length k, every set of size at least $k+1$ contains two subsets with the same fingerprint. We primarily focus on active adversaries, as they are more powerful.

Fingerprint collisions result in parts of the set not being synchronized, so information is being withheld from one or both of the nodes. When a malicious node synchronizes with an honest one, the malicious node can withhold arbitrary information by simply pretending not to have certain data, which does not require finding collisions at all. So the actually

Algorithm 2

Require: $x \prec y$

procedure AGGREGATE_UNTIL(t, x, y)
  $(acc, t) \leftarrow$ AGGREGATE_UP(t, x, y)
  if $t = \texttt{nil} \vee t.\texttt{v} \succeq y$ **then**
    return (acc, t)
  else
    return AGGREGATE_DOWN$(t.\texttt{r}, y, acc \oplus \mathrm{f}(t.\texttt{v}))$
  end if
end procedure
procedure AGGREGATE_UP(t, x, y)
  $acc \leftarrow \mathbb{0}$
  while $t.\texttt{m} \prec y$ **do**
    if $t.\texttt{v} \succeq x$ **then**
      $acc \leftarrow acc \oplus \mathrm{f}(t.\texttt{v}) \oplus \mathrm{label}_{\mathrm{f}}^{\mathcal{M}}(t.\texttt{r})$
    end if
    if $t.\texttt{p} = \texttt{nil}$ **then**
      return $(acc, \texttt{nil})$
    else
      $t \leftarrow t.\texttt{p}$
    end if
  end while
  return (acc, t)
end procedure
procedure AGGREGATE_DOWN(t, y, acc)
  while $t \neq \texttt{nil}$ **do**
    if $t.\texttt{v} \prec y$ **then**
      $acc \leftarrow acc \oplus \mathrm{label}_{\mathrm{f}}^{\mathcal{M}}(t.\texttt{l}) \oplus \mathrm{f}(t.\texttt{v})$
      $t \leftarrow t.\texttt{r}$
    else if $t.\texttt{l} = \texttt{nil} \vee t.\texttt{l}.\texttt{m} \prec y$ **then**
      return $(acc \oplus \mathrm{label}_{\mathrm{f}}^{\mathcal{M}}(t.\texttt{l}), t)$
    else
      $t \leftarrow t.\texttt{l}$
    end if
  end while
  return $(acc, \texttt{nil})$
end procedure

interesting cases are those where a malicious node can cause honest nodes to incompletely reconcile amongst themselves.

Specifically: let $\mathcal{M}$ be a malicious node, $\mathcal{A}$ and $\mathcal{B}$ be honest nodes, then a successful attack consists of $\mathcal{M}$ crafting sets X_A, X_B and sending these to $\mathcal{A}$ and $\mathcal{B}$ respectively, so that when $\mathcal{A}$ and $\mathcal{B}$ then reconcile, they end up with distinct sets. A passive adversary does not craft X_A, X_B but must find them as subsets of some set X supplied by an honest node.

Let $S_A \subseteq X_A$ and $S_B \subseteq X_B$ be nonequal sets with the same fingerprint. To have any impact on the correctness of a particular protocol run, their two fingerprints need to actually be compared during that run. For that to happen, there have to be $x, y \in U$ such that $S_A = [x, y)_{X_A}$ and $S_B = [x, y)_{X_B}$. This alone is not sufficient, as only a low number of pairs of such sets are actually compared in a single run.

Nodes can randomize the split points when determining subranges to bound the probability that a given pair of colliding sets is compared in a single protocol run. They can, e.g., split ranges into equally-sized subranges first, but then randomly shift the range boundaries by a small number of items. This preserves a logarithmic number of communication rounds in the worst case. They can even choose b subrange boundaries fully at random. The expected number of communication rounds is in $\mathcal{O}(\log_b(n)) = \mathcal{O}(\log(n))$ with high probability, as it corresponds to the height of a randomly chosen b-complete tree [Dev90].

This argument is however only qualitative and should be enjoyed with caution. A strong attacker might be able to find many pairs of sets of colliding fingerprints, or many sets that all share the same fingerprint.

B. Cryptographically Secure Fingerprints

For stronger guarantees, we thus look for cryptographically secure fingerprint functions that make it computationally infeasible for an adversary to find colliding fingerprints.

A typical definition of cryptographically secure hash functions is the following [MVOV18]:

Definition 9. A *secure hash function* is a hash function $\mathrm{h} : U \to D$ that satisfies three additional properties:

pre-image resistance: Given $d \in D$, it is computationally infeasible to find a $u \in U$ such that $\mathrm{h}(u) = d$.

second pre-image resistance: Given $u \in U$, it is computationally infeasible to find a $u' \in U, u' \neq u$ such that $\mathrm{h}(u) = \mathrm{h}(u')$.

collision resistance: It is computationally infeasible to find $u, v \in U, u \ = v$ such that $\mathrm{h}(u) = \mathrm{h}(v)$.

What do secure fingerprints for our sets look like? Since $\mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}(\{u\}) = \mathrm{f}(u)$, f must necessarily be a secure hash function if $\mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}$ is to be one. More interesting is the choice of monoid.

Bellare and Micciancio [BM97] propose incremental hashing of strings by first hashing substrings and then combining the hashes according to some commutative group operation — the groups they study are also possible candidates for our construction. After showing how to efficiently find collisions when using xor for combining hashes, they consider three more robust groups.

They unify parts of their discussion by relating the hardness of finding collisions to solving the balance problem: in a commutative group $(G, \oplus, \mathbb{0})$, given a set of group elements $S = \{s_1, s_2, \ldots, s_n\}$, find disjoint, nonempty subsets $S_0 = \{s_{0,0}, s_{0,1}, \ldots, s_{0,k}\} \subseteq S, S_1 = \{s_{1,0}, s_{1,1}, \ldots, s_{1,l}\} \subseteq S$ such that $s_{0,0} \oplus s_{0,1} \oplus \ldots \oplus s_{0,k} = s_{1,0} \oplus s_{1,1} \oplus \ldots \oplus s_{1,l}$. They then reduce the hardness of the balance problem to other problems, depending on the specific group.

One group operation they consider is addition modulo the group size. The balance problem is as hard as subset sum for this group, which was conjectured to be sufficiently hard. Wagner showed however how to solve the balance problem in subexponential time for addition [Wag02]. Lyubashevsky later strengthened the attack, finding collisions in $\mathcal{O}(2^{n^\epsilon})$ for arbitrary $\epsilon < 1$. A more recent proposal suggesting

addition for combining SHA-3 [Dwo15] digests hence proposes fingerprints of length between 2688 and 4160 or 6528 to 16512 bits to achieve security levels of 128 or 256 bit respectively [MGS15].

Another candidate group is $\mathbb{Z}_n^*$, the group yielded by multiplication modulo n on the set $\{x \in [0,n)_{\mathbb{N}} \mid x \text{ is coprime to } n\}$. The balance problem is as hard as the discrete logarithm problem in these groups; this is hard for groups of prime order and for $\mathbb{Z}_p^*$ where p is prime [BM97]. Multiplication is however less efficient to compute than addition, benchmarks show that the additive hash outperforms the multiplicative one by two orders of magnitude, even though the additive hash uses longer digests to account for Wagner's attack [SMBA10]. Fingerprints based on multiplication still need larger digests than traditional, non-incremental hash functions, Maitin-Shepard et al. [MSTA17] suggests fingerprints of 3200 bit to achieve 128 bit security.

The last candidate group partitions bitstrings into smaller strings and performs component-wise addition. Collision resistance is related to the hardness of the shortest lattice vector approximation problem [Ajt96]. A recent instantiation provides 200 bits of security with fingerprints of size $16 \cdot 1024 = 16384$ bit [LKMW19].

Bellare and Micciancio's hash functions are *multiset homomorphic* [CDVD$^+$03]:

Definition 10. Let $\mathcal{U}_0 := (U_0, \oplus_0, \mathbb{0}_0)$ and $\mathcal{U}_1 := (U_1, \oplus_1, \mathbb{0}_1)$ be monoids, and let $\mathrm{f} : U_0 \to U_1$.

We call f a *monoid homomorphism from $\mathcal{U}_0$ to $\mathcal{U}_1$* if for all $x, y \in U_0$ we have $\mathrm{f}(x \oplus_0 y) = \mathrm{f}(x) \oplus_1 \mathrm{f}(y)$.

Definition 11. Let $\mathcal{S} := (\mathbb{N}^U, \cup, \emptyset)$ be the monoid of multisets over the universe U under union, $\mathcal{M} := (M, \oplus, \mathbb{0})$ a monoid, and $\mathrm{f} : \mathbb{N}^U \to M$.

We call f a *multiset homomorphic hash function* if f is a hash function and a monoid homomorphism from $\mathcal{S}$ to $\mathcal{M}$.

Being a monoid homomorphism from $\mathcal{S}$ to $\mathcal{M}$ is a strictly stronger criterium than being a tree-friendly function. Hence, every multiset homomorphic hash function is suitable for our purposes. Additional multiset homomorphic hash functions are based on RSA [CNQ09] or on elliptic curves [MSTA17].

Because multiset union is commutative, so is necessarily any multiset homomorphic hash function. Tree-friendly functions do not require commutativity however. *Cayley hash functions* [Zém91][PQ$^+$11] are non-commutative hash functions based on multiplication of invertible matrices. While early schemes [TZ94] have been successfully attacked [GIMS11][PQ10], there are several modifications for which no attacks are known [Pet09][BSV17][Sos16]. Cayley hash functions are randomly self-reducible [MT16].

Aside from Cayley hashes, we are not aware of any non-commutative monoids used for hashing. Note that even cayley hashes have more more structures than we need, as we don't require existence of inverse elements.

Regardless of the choice of monoid, the reconciliation protocol can exchange (conventional) hashes of fingerprints rather than exchanging fingerprints directly. This allows us to use large fingerprints to achieve security (e.g., using bitwise additions on long bitstrings as the monoid), while still transmitting only a small number of bits over the wire. The large fingerprints do however increase the space consumption of the monoid tree. Whether a computationally expensive monoid operation on small bitstrings outperforms a cheaper operation on larger bitstrings is hence not obvious and requires benchmarking to make an informed choice.

VI. Conclusion

We consider range-based set reconciliation to be an important complement to the bulk of the related literature which focuses on achieving a constant number of communication rounds. While the logarithmic number of communication rounds is a significant drawback, no constant-roundtrip approach achieves computational complexity proportional to only the size of the symmetric difference; and range-based reconciliation is conceptually much simpler. Among other schemes that take a logarithmic number of rounds, ours is the only one to guarantee both logarithmic roundtrips and computational complexity in the worst case.

The asymmetric scenario where one node operates using only a constant amount of working memory is unique among all proposed schemes. The option of using a more sophisticated reconciliation procedure once ranges have become small enough can make it worth a consideration in almost any reconciliation scenario.

We have deliberately restricted our focus to the very core issues of the range-based approach. There is a large number of engineering issues and possible generalizations, for example,

- generalizing to multidimensional ranges,
- identifying other expressive partitioning/covering schemes, and data structures for efficiently computing the fingerprints for such partitions,
- generalizing to reconciliation/mirroring of maps (with proper conflict resolution when both peers map equal keys to different values),
- changing the monoid tree to a tree of higher degree with item storage in the leaves only, as would be typical for IO-efficient persistence on secondary storage,
- investigating how to efficiently relay changes to a partially reconciled set when running multiple reconciliation sessions concurrently or in parallel,
- generalizing from two-party reconciliation to multi-party reconciliation, or
- adapting the protocol to unordered and/or unreliable transports.

So above all, we hope to bring more attention to the range-based approach, as the sparse treatment it has received in the literature so far does not do justice to its practical applicability.

VII. Acknowledgments

Anonymized Person contributed the idea of compressing fingerprints with a hash function before transmission.

REFERENCES

[Ajt96] Miklós Ajtai. Generating hard instances of lattice problems. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 99–108, 1996.

[AT19] Alex Auvolat and François Taïani. Merkle search trees: Efficient state-based crdts in open networks. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, pages 221–22109. IEEE, 2019.

[BCM02] John Byers, Jeffrey Considine, and Michael Mitzenmacher. Fast approximate reconciliation of set differences. In *BU Computer Science TR*. Citeseer, 2002.

[Blo70] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[BM97] Mihir Bellare and Daniele Micciancio. A new paradigm for collision-free hashing: Incrementality at reduced cost. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 163–192. Springer, 1997.

[BM02] Rudolf Bayer and Edward McCreight. Organization and maintenance of large ordered indexes. In *Software pioneers*, pages 245–262. Springer, 2002.

[BSV17] Lisa Bromberg, Vladimir Shpilrain, and Alina Vdovina. Navigating in the cayley graph of $sl_2(f_p)$ and applications to hashing. In *Semigroup Forum*, volume 94, pages 314–324. Springer, 2017.

[CDVD+03] Dwaine Clarke, Srinivas Devadas, Marten Van Dijk, Blaise Gassend, and G Edward Suh. Incremental multiset hash functions and their application to memory integrity checking. In *International conference on the theory and application of cryptology and information security*, pages 188–207. Springer, 2003.

[CEG+99] Yuan Chen, Jan Edler, Andrew Goldberg, Allan Gottlieb, Sumeet Sobti, and Peter Yianilos. A prototype implementation of archival intermemory. In *Proceedings of the fourth ACM conference on Digital libraries*, pages 28–37, 1999.

[CLRS22] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

[CNQ09] Julien Cathalo, David Naccache, and Jean-Jacques Quisquater. Comparing with rsa. In *IMA International Conference on Cryptography and Coding*, pages 326–335. Springer, 2009.

[Dev90] Luc Devroye. On the height of random m-ary search trees. *Random Structures & Algorithms*, 1(2):191–203, 1990.

[DGH+87] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12, 1987.

[Dwo15] Morris J Dworkin. Sha-3 standard: Permutation-based hash and extendable-output functions. 2015.

[EGUV11] David Eppstein, Michael T Goodrich, Frank Uyeda, and George Varghese. What's the difference? efficient set reconciliation without prior context. *ACM SIGCOMM Computer Communication Review*, 41(4):218–229, 2011.

[GIMS11] Markus Grassl, Ivana Ilić, Spyros Magliveras, and Rainer Steinwandt. Cryptanalysis of the tillich–zúmor hash function. *Journal of cryptology*, 24(1):148–156, 2011.

[GL12] Deke Guo and Mo Li. Set reconciliation via counting bloom filters. *IEEE Transactions on Knowledge and Data Engineering*, 25(10):2367–2380, 2012.

[GM11] Michael T Goodrich and Michael Mitzenmacher. Invertible bloom lookup tables. In *2011 49th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 792–799. IEEE, 2011.

[LGR+19] Lailong Luo, Deke Guo, Ori Rottenstreich, Richard TB Ma, and Xueshan Luo. Set reconciliation with cuckoo filters. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, pages 2465–2468, 2019.

[LKMW19] Kevin Lewi, Wonho Kim, Ilya Maykov, and Stephen Weis. Securing update propagation with homomorphic hashing. *IACR Cryptol. ePrint Arch.*, 2019:227, 2019.

[MGS15] Hristina Mihajloska, Danilo Gligoroski, and Simona Samardjiska. Reviving the idea of incremental cryptography for the zettabyte era use case: Incremental hash functions based on sha-3. In *International Workshop on Open Problems in Network Security*, pages 97–111. Springer, 2015.

[MSTA17] Jeremy Maitin-Shepard, Mehdi Tibouchi, and Diego F Aranha. Elliptic curve multiset hash. *The Computer Journal*, 60(4):476–490, 2017.

[MT02] Yaron Minsky and Ari Trachtenberg. Practical set reconciliation. In *40th Annual Allerton Conference on Communication, Control, and Computing*, volume 248. Citeseer, 2002.

[MT16] Ciaran Mullan and Boaz Tsaban. sl_2 homomorphic hash functions: worst case to average case reduction and short collision search. *Designs, Codes and Cryptography*, 81(1):83–107, 2016.

[MTZ03] Yaron Minsky, Ari Trachtenberg, and Richard Zippel. Set reconciliation with nearly optimal communication complexity. *IEEE Transactions on Information Theory*, 49(9):2213–2218, 2003.

[MVOV18] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 2018.

[OAL+19] A Pinar Ozisik, Gavin Andresen, Brian N Levine, Darren Tapp, George Bissias, and Sunny Katkuri. Graphene: efficient interactive set reconciliation applied to blockchain propagation. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 303–317. 2019.

[Pet09] Christophe Petit. *On graph-based cryptographic hash functions.* PhD thesis, Catholic University of Louvain, Louvain-la-Neuve, Belgium, 2009.

[PQ10] Christophe Petit and Jean-Jacques Quisquater. Preimages for the tillich-zémor hash function. In *International Workshop on Selected Areas in Cryptography*, pages 282–301. Springer, 2010.

[PQ+11] Christophe Petit, Jean-Jacques Quisquater, et al. Rubik's for cryptographers. *IACR Cryptol. ePrint Arch.*, 2011:638, 2011.

[SMBA10] Paul T Stanton, Benjamin McKeown, Randal Burns, and Giuseppe Ateniese. Fastad: an authenticated directory for billions of objects. *ACM SIGOPS Operating Systems Review*, 44(1):45–49, 2010.

[Sos16] Bianca Sosnovski. Cayley graphs of semigroups and applications to hashing. 2016.

[SYW+17] Wentao Shang, Yingdi Yu, Lijing Wang, Alexander Afanasyev, and Lixia Zhang. A survey of distributed dataset synchronization in named data networking. *NDN, Technical Report NDN-0053*, 2017.

[TZ94] Jean-Pierre Tillich and Gilles Zémor. Hashing with sl 2. In *Annual International Cryptology Conference*, pages 40–49. Springer, 1994.

[TZX+11] Xiaomei Tian, Dafang Zhang, Kun Xie, Can Hu, Mengfan Wang, and Jinguo Deng. Exact set reconciliation based on bloom filters. In *Proceedings of 2011 International Conference on Computer Science and Network Technology*, volume 3, pages 2001–2009. IEEE, 2011.

[Wag02] David Wagner. A generalized birthday problem. In *Annual International Cryptology Conference*, pages 288–304. Springer, 2002.

[Zém91] Gilles Zémor. Hash functions and graphs with large girths. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 508–511. Springer, 1991.