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 epicdemic [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. We hence look for a solution that bounds computational complexity by the size of the symmetric difference.

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 first review related work in section II. We then 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.

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. To that end, we give an overview

[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

of the set reconciliation literature in section II-A, and of the existing literature on set fingerprinting in section II-B.

A. Set Reconcilliation

Most reconciliation literature focuses on reconciliation in a single communication round, at the price of high computational costs. In the following discussion, we consider 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 filter 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].

[MT02] introduces *partition reconciliation*,which reconciles in a logarithmic number of rounds in order to reduce computational load. It builds upon the CPI approach, and attempts reconciliation for successively smaller subsets until it succeeds once the subsets becomes small enough.

This approach eliminates CPI's cubic scaling of the computation time in $n_\triangle$. Similar to our auxiliary tree, the authors propose a tree structure of partitions where a parent node includes subranges as children. The reconciliation message for each of these ranges is precomputed and stored within the tree.

Given such a partition tree, the reconciliation procedure has the same asymptotic worst-case complexity bounds as ours. It has performs better on average however: whereas our approach recurses whenever the two ranges that are being compared are unequal, i.e., whenever the size of the symmetric difference is greater than zero, their approach only recurses whenever the size of the symmetric difference is greater than some arbitrary, fixed constant.

The tree can be updated over insertions and deletions in time logarithmic in the height of the tree, but these updates are not balancing. In the worst case, maintaining this auxiliary data structure can thus take time linear in the size of the set per update. Furthermore, the tree is specific to a particular choice of evaluation points and control points for the characteristic polynomial. If the tree is to be reused across multiple reconciliation sessions, these points have to be fixed in advance. This could allow an attacker to craft malicious sets for which a failed reconciliation is not detected. As a consequence, this approach can only be used with the producers of the sets are trusted.

[AT19] also considers reconciliation in a logarithmic number of rounds, they use pseudorandom Merkle trees, comparable to our **??**. Their approach is far less flexible than ours however: reconciliation messages are guided by the tree shape, so the number of recursion steps in each round is fixed, and the maximum number of rounds can degrade to $\mathcal{O}(n)$ in the worst case. Our approach leaves complete freedom for the number of recursion steps and guarantees a logarithmic number of communication rounds in the worst case, even when using a pseudorandom tree for fingerprint computations. Their promising experimental evaluation provides a strong indicator that our more efficient approach would also perform well in practice.

B. Fingerprinting Sets

As range-based synchronization relies on fingerprinting sets, we give a small overview of work on the topic.

Merkle trees [Mer89] introduced the idea of maintaining hashes in a tree to efficiently compute and update a fingerprint. Since the exact shape of the tree determines the root label, unique representations of sets as trees are of interest. [Sny77] proves the important negative result that unique representations require superlogarithmic time to maintain under insertions and deletions in general. [ST94] gives logarithmic solutions for sparse or dense sets and points to further deterministic solutions.

Unique representations with probabilistic time complexities that are logarithmic in the expected case have been studied in [PT89], suggesting hash-tries as a set representation for computing hashes. Pugh also developed skip lists [Pug90], a probabilistic set representation not based on trees. [SA96] offers treaps, another randomized tree structure. Further study of uniquely represented data structures has been done under the moniker of *history-independent data structures*, including IO-efficient solutions for treaps [Gol09] and skip list [BBJ+16] in an *external memory* computational model. Using history-independent data structures for defining the fingerprints in a range-based reconciliation scheme is possible in principle, but malicious actors can craft degenerate sets that incur super-logarithmic computation times. Furthermore, such an approach cannot leave the exact choice of set data structure as an implementation detail.

Beyond the comparison of root labels for set equality testing, Merkle trees and their technique of hashing the concatenation of hashes form the basis of many authenticated data structures, ranging from simple balanced trees or treaps [NN00], authenticated maps [BLL00][AT19] and skip lists [GT00] to more general DAG-based data structures [MND+04].

A different line of authenticated data structures utilizes dynamic accumulators [CL02], small digests for sets that can be efficiently updated under insertion and deletion of items, and which allow computing small certificates that some item has been ingested. [PTT16] and [PTT11] use accumulators to provide authenticated set data structures that are more efficient than their Merkle-tree-based counterparts. Accumulators are stronger than necessary for the range-based synchronization approach, so they are not discussed in our main text.

Orthogonal to these lines of research are algebraic approaches. The earliest mention of hashing into a group and performing computations on hashes to compactly store information about sets that we could find is in [WC81]. They provide probabilistic set equality testing, but without maintaining any tree structure, effectively anticipating (cryptographically insecure) homomorphic set fingerprinting.

The first investigation of the cryptographic security of this approach was done in the seminal [BM97], albeit in the slightly less natural context of sequences rather than sets, with the additive hash being broken in [Wag02] and [Lyu05]. Multiset homomorphic terminology is due to [CDVD+03]; [CNQ09], [MSTA17] and [LKMW19] give further constructions.

The generalized hash tree of [PSTY13] extends the idea of combining hashes via a binary operation to operations that are not closed, instead the output is mapped back into the original domain by a separate function. These "compressed" outputs are then arranged in a tree; this technique allows using the algebraic approach for authenticated data structures.

III. Recursive Reconciliation

In this section, we describe the communication that goes into a range-based set reconciliation session, while deferring the details of fingerprint computations to section IV.

We consider the setting of two nodes $\mathcal{X}_0$ and $\mathcal{X}_1$ that are connected via a bidirectional, reliable, ordered communication channel. We assume that an arbitrary number of bits can be sent in a single, unit-length communication round. The nodes initially hold sets X_0 and X_1 respectively, and 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 every element from U to be transmittable using a bounded number of bits, that is, we require U to be finite. This can always be achieved in practice by reconciling hashes of items rather than items themselves.

We finally assume that there is a fingerprinting function $\mathrm{fp} : \mathcal{P}(U) \to H$ that maps arbitrary subsets of U (and hence also of X_0 and X_1) into some finite codomain H with negligible probability of collisions. We discuss such functions in detail in section IV.

Before we can define the message exchange, we need some formal definitions and terminology for talking about ranges:

Definition 1 (Range)**.** 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 $y \preceq x$).

Note that x and y need not be elements of S themselves.

A. Protocol Description

In a given communication round, a node receives information about some subranges of the sets to be reconciled: for each subrange in question, it receives either a fingerprint which depends exactly on which items the other node holds within that subrange, or it receives specific items within that subrange that need to be reconciled. The node then answers by supplying its own information, further partitioning ranges into arbitrarily many 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 (Message)**.** 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 flag signals whether the other node should respond with its local items from x to y as well ($b = 0$), 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.

In order to initiate reconciliation, a node sends a message consisting solely of a range fingerprint $\big(x, x, \mathrm{fp}([x, x)_{X_i})\big)$ for some $x \in U$. The nodes can then reconcile their sets by following 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. If $b = 0$ and $[x, y)_{X_i} \setminus S \neq \emptyset$, add the range item set $(x, y, [x, y)_{X_i} \setminus S, 1)$ to the response.
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 $\mathrm{fp}([x, y)_{X_j}) = \mathrm{fp}(\emptyset)$, you must 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.

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 an older version that is stored on the replica. Protocol 1 can be used with very little modification in order to give a set mirroring protocol. 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.

For simplicity, we restrict our presentation to set reconciliation only for the remainder of this article, but all our results also apply to this mirroring technique.

C. Protocol Properties

Protocol 1 gives some leeway for nodes to decide whether and into how many subranges to split any range fingerprint they receive. The precise choices do not impact the correctness of the protocol. In particular, two nodes can reconcile a set even when using different strategies for deciding when and how to recurse. In a decentralized setting, different implementations do not need to coordinate and can make decisions based on their available resources.

1) Termination and Correctness: Termination of protocol 1 follow from an inductive argument. The number of items in the largest subrange of the current message is strictly decreasing with each round. Small ranges and ranges with matching fingerprints are handled within a constant number of communication rounds, so the protocol always terminates, even if the nodes prefer recursing over sending range item whenever allowed to do so.

Correctness can also be argued inductively. Under the assumption that fingerprints do not collide, ranges with equal fingerprints are reconciled correctly. Sending a range item set and receiving the response also leads to both nodes storing the union of all items within that range. The subranges created by the recursive case are reconciled correctly by induction hypothesis. And because the subranges partition the original range[2], this completely reconciles the original range.

[2]In fact, for correctness it already suffices that the subranges *cover* the original range. We limit our discussion to *partitions* because they minimally cover the original range without containing any duplicate items.

2) Complexity: The protocol gives nodes the freedom to respond to any range fingerprint with a range item set, even if the range fingerprint is arbitrarily large. For a meaningful complexity analysis we need to restrict the behavior of nodes. A realistic modus operandi is for a node to send a range item set only if the number of its items in the range is less than or equal to some threshold $t \in \mathbb{N}^+$. Higher choices for t reduce the number of roundtrips, but increase the probability that an item is being sent even though the other node already holds it.

A node is similarly given freedom over the number of subranges into which to split a range when recursing. We will assume that nodes always split into at most $b \in \mathbb{N}, b \geq 2$ subranges. As with t, higher numbers reduce the number of roundtrips, at the cost of potentially sending fingerprints that did not need sending.

Transmitting all available items within a range is a very simple choice of recursion anchor, but becomes wasteful for large choices of t. A more sophisticated approach is to run a more efficient constant-round set reconciliation protocol such as [MTZ03] once a range contains few enough items for both nodes. As that protocol would only be run on small sets, its time and space complexities would be in $\mathcal{O}(1)$, so the overall time and space complexities would be the same as for unmodified range-based reconciliation. Hence, range-based set reconciliation can be regarded as a mechanism for reducing the computational complexity of arbitrary reconciliation protocols. In our further analysis, we stick to protocol 1 as presented however.

Because we want to analyze not only the worst-case complexity but also the complexity depending on the similarity between the two sets held by the participating nodes, we define some fine-grained instance size parameters: n_0 and n_1 denote the number of items held by $\mathcal{X}_0$ and $\mathcal{X}_1$ respectively. We let $n := n_0 + n_1$, $n_{\min} := \min(n_0, n_1)$ and $n_\triangle := |(X_0 \cup X_1) \setminus (X_0 \cap X_1|$ (the size of the symmetric difference of X_0 and X_1).

A helpful observation for the coming analyses is that the range fingerprints of a protocol run form a rooted, b-ary 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 tree is bounded by $2 \cdot \lceil \log_b(n_{\min}) \rceil$. The number of vertices of a b-ary tree of height h is less than twice the number of leaves, so the overall number of vertices is in $\mathcal{O}(n)$.

The parameter t determines when recursion is cut off, and thus 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 required for reconciliation is bounded by the number of times that each node can split ranges without transmitting items, followed by two rounds of exchanging range items sets. This corresponds to two plus the height of the tree, so $2 + 2 \cdot \lceil \log_b(n_{\min}) \rceil - \lfloor \log_b(t) \rfloor \in \mathcal{O}(\log(n))$.

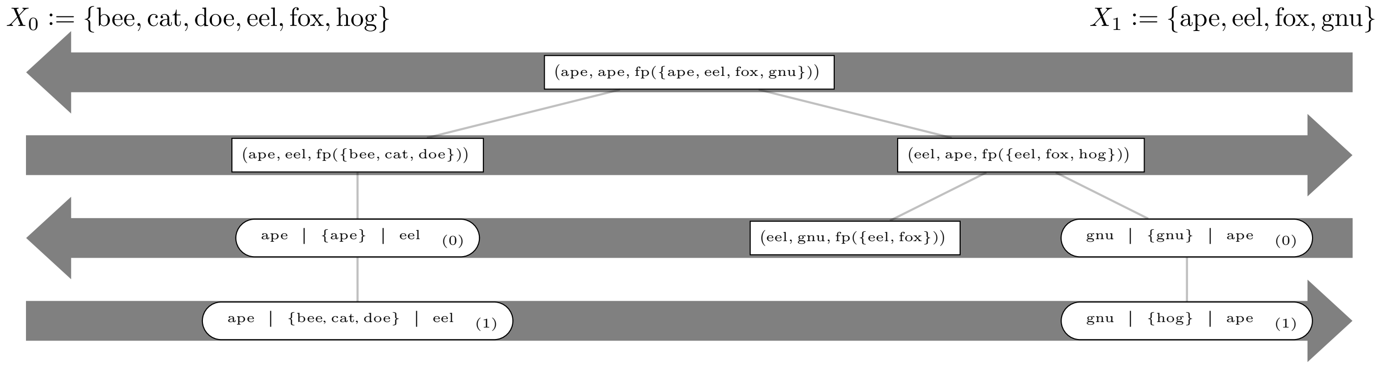

Fig. 1. An example run of protocol 1. In this and further examples, $U := \{\mathrm{ape}, \mathrm{bee}, \mathrm{cat}, \mathrm{doe}, \mathrm{eel}, \mathrm{fox}, \mathrm{gnu}, \mathrm{hog}\}$, and $\preceq$ orders the universe alphabetically. Range fingerprints have sharp corners, range item sets have rounded corners. The arrows in the background indicate which node is sending and which node is receiving.
$\mathcal{X}_1$ initiates reconciliation over the full universe, transmitting the fingerprint of X_1.
Upon receiving this range fingerprint, $\mathcal{X}_0$ locally computes $\mathrm{fp}([\mathrm{ape}, \mathrm{ape})_{X_0})$. 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 $|[\mathrm{ape}, \mathrm{eel})_{X_1}| \leq 1$, $\mathcal{X}_1$ transmits the corresponding range items set $(\mathrm{ape}, \mathrm{eel}, \mathrm{ape}, 0)$. For the other range, we have that $|[\mathrm{eel}, \mathrm{ape})_{X_1}| > 1$ 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 fourth and 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 $\big(\mathrm{eel}, \mathrm{gnu}, \mathrm{fp}([\mathrm{eel}, \mathrm{gnu})_{X_1})\big)$, it computes an equal fingerprint $\mathrm{fp}([\mathrm{eel}, \mathrm{gnu})_{X_0})$, so that range can be considered successfully reconciled without transmitting any more data.

This number cannot be bounded by $n_\triangle$, as witnessed by problem instances where one node is missing exactly one item compared to the other node. 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. See fig. 2 for an example.

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 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 overlap however, and every vertex of the tree only contributes $\mathcal{O}(1)$ bits to the reconciliation session, no matter how many items made it necessary to transmit that fingerprint. 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 more overlap between the paths, the fewer bits per item in the symmetric difference have to be transmitted. 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 very efficient however: in this scenario, $n_\triangle$ is within a constant factor of n, so 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.

Finally, we can observe that the largest possible size of a single message is proportional to the highest number of vertices in a single layer of the tree, i.e., it is proportional to $n_\triangle$. This affects the space complexity for the participating nodes. We will choose fingerprints such that nodes can process each message part within $\mathcal{O}(1)$ space. A simple way of implementing protocol 1 is then for each node to store the full messages of size $\mathcal{O}(n_\triangle)$ in memory; the space requirement then also becomes $\mathcal{O}(n_\triangle)$.

Alternatively, nodes can split messages and transmit only a bounded number of message parts at a time, since it is not necessary to know the full message in order to process a single message part. 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 and only allocate space for a constant number of message parts, the protocol can reach a deadlock. The number of message parts in a response can be greater than the number of message parts from which they were generated. When the number of message parts that would have to be transmitted exceeds the space capacity of both nodes added together, no node is able to receive or transmit more data.

It is however possible for only one of the two nodes to use a constant amount of space, whereas the other node then has to buffer up to a full outgoing and incoming message simultaneously, for a total space complexity of $\mathcal{O}(n_\triangle)$. This setup leads to a higher number of communication rounds. The node with unbounded memory has to wait for confirmation before transmitting a new set of message parts, which adds a round trip for every set of message parts that needs to be transmitted; transmitting a message of size $\mathcal{O}(n_\triangle)$ takes $\mathcal{O}(n_\triangle)$ communication rounds. The overall number of communication rounds for the protocol then becomes $\mathcal{O}(n_\triangle \cdot \log(n))$.

This analysis seems unfavorable, but note that only a constant number of bits needs to be transmitted in every single communication round. The lower number of communication rounds in the setting where both nodes can store arbitrarily large messages can only be achieved if arbitrarily large messages can be transmitted in a single communication round. In

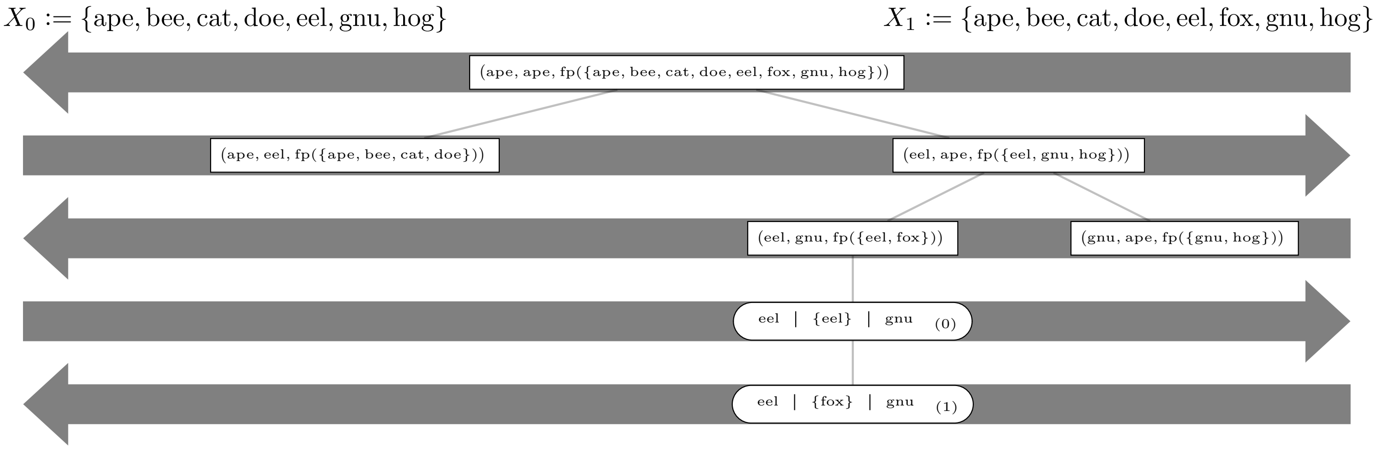

Fig. 2. An example run of the protocol that takes the greatest possible number of rounds, even though $n_\triangle = 1$. $b := 2, t := 1$.

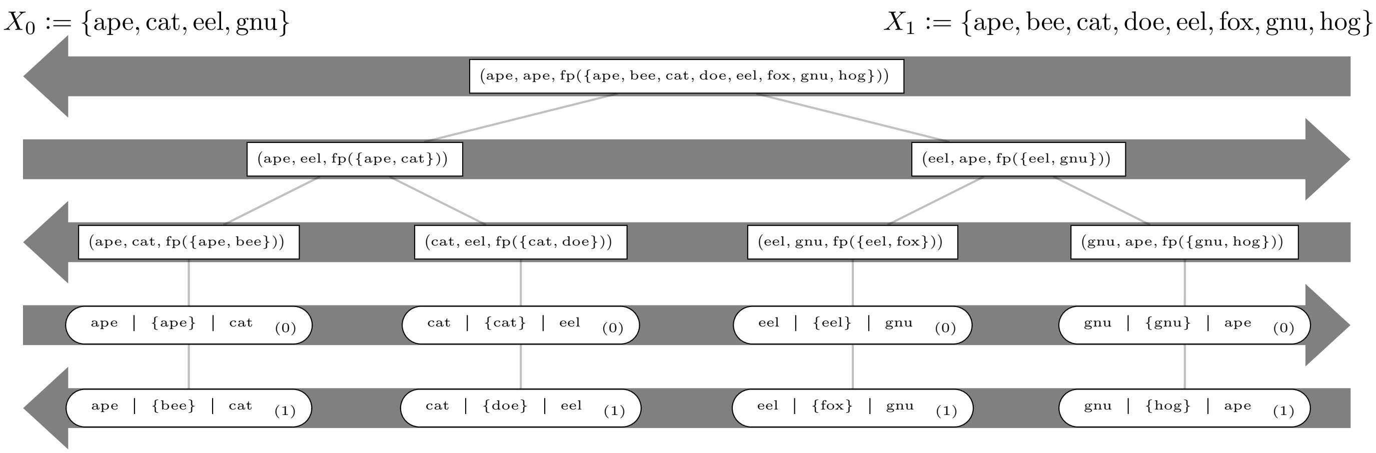

Fig. 3. An example run of the protocol that requires transmitting the maximum amount of bytes. $b := 2, t := 1$.

a more realistic networking model, a limited bandwidth of k bits per communication round implies that transmitting n bits requires $\frac{n}{k} \in \mathcal{O}(n)$ rounds.

Under such a model, transmitting a single message of size $\mathcal{O}(n_\triangle)$ requires $\mathcal{O}(n_\triangle)$ rounds, hence, running range-based set reconciliation with unbounded memory takes $\mathcal{O}(n_\triangle \cdot \log(n))$ rounds as well. So when taking finite bandwidth into account, one of the two nodes can operate with a constant amount of memory without affecting the asymptotic complexity of the protocol. Every possible set reconciliation protocol needs to transmit $\Omega(n_\triangle)$ bits [MTZ03], so range-based set reconciliation is within a logarithmic factor of the optimal number of communication rounds under this model.

IV. Fingerprint Computation

We now examine the time and space complexity of the computations each node must perform 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 and in secondary storage rather than main memory. Overall, we are interested in the time 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. *Labeled Trees*

When computing messages, a node must be able to 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 (Monoid)**.** 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 tree structure. We define trees inductively to simplify the presentation of algorithms:

Definition 5 (Binary Tree)**.** A *binary tree* t over a universe U is either the empty tree $\mathtt{nil}$, or a triplet of a left subtree $t.\mathtt{l}$, a value $t.\mathrm{v} \in U$, 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 (Monoid Tree)**.** 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 *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 such a labeling function 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 yields the same result, as $\oplus$ is associative. Because we label a search tree, $\oplus$ is always applied to the items in ascending order. If we were labeling arbitrary, not necessarily sorted binary trees, $\oplus$ would have to be commutative for the equality of labels and lifted functions to hold.

Returning to our previous example, labeling a tree with $\mathrm{label}_{\lambda x.1}^{\mathcal{N}}$ 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 $\mathrm{label}_{\lambda x.1}^{\mathcal{N}}$ in particular and for arbitrary $\mathrm{label}_{\mathrm{f}}^{\mathcal{M}}$ functions in general [CLRS22].

Every function of the form $\mathrm{label}_{\mathrm{f}}^{\mathcal{M}}$ for some function f and some monoid $\mathcal{M}$ can be used for maintaining labels in the tree, but do there exist other such functions as well? To answer this, we give a homomomorphism-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 that is available locally around 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 $x \preceq \mathit{init} \prec y$, we have that

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

and hence

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

Procedure AGGREGATE_LEFT demonstrates how to compute $\mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}\big(\{z \in \mathrm{V}(\mathit{init}.\mathtt{l}) \mid z \succeq x\}\big)$: 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 $\mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}\big(\{z \in \mathrm{V}(\mathit{init}.\mathtt{r}) \mid z \prec y\}\big)$. Figure 4 depicts an example run.

Overall, algorithm 1 performs 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

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

```
Require: x ⪯ y, t ≠ nil
procedure AGGREGATE_RANGE(t, x, y)
    if x = y then
        return label_f^M(t)
    end if
    init ← FIND_INITIAL(t, x, y)
    if init = nil then
        return 𝟘
    else
        acc_l ← AGGREGATE_LEFT(init.l, x)
        acc_r ← AGGREGATE_RIGHT(init.r, y)
        return acc_l ⊕ f(init.v) ⊕ acc_r
    end if
end procedure
procedure FIND_INITIAL(t, x, y)
    while true do
        if t = nil then
            return t
        else if t.v ≺ x then
            t ← t.r
        else if t.v ⪰ y then
            t ← t.l
        else
            return t
        end if
    end while
end procedure
procedure AGGREGATE_LEFT(t, x)
    acc ← 𝟘
    while true do
        if t = nil then
            return acc
        else if t.v ≺ x then
            t ← t.r
        else if t.v = x then
            return f(t.v) ⊕ label_f^M(t.r) ⊕ acc
        else
            acc ← f(t.v) ⊕ label_f^M(t.r) ⊕ acc
            t ← t.l
        end if
    end while
end procedure
procedure AGGREGATE_RIGHT(t, y)
    acc ← 𝟘
    while true do
        if t = nil then
            return acc
        else if t.v ≺ y then
            acc ← acc ⊕ label_f^M(t.l) ⊕ f(t.v)
            t ← t.r
        else if t.v = x then
            return acc ⊕ label_f^M(t.l)
        else
            t ← t.l
        end if
    end while
end procedure
```

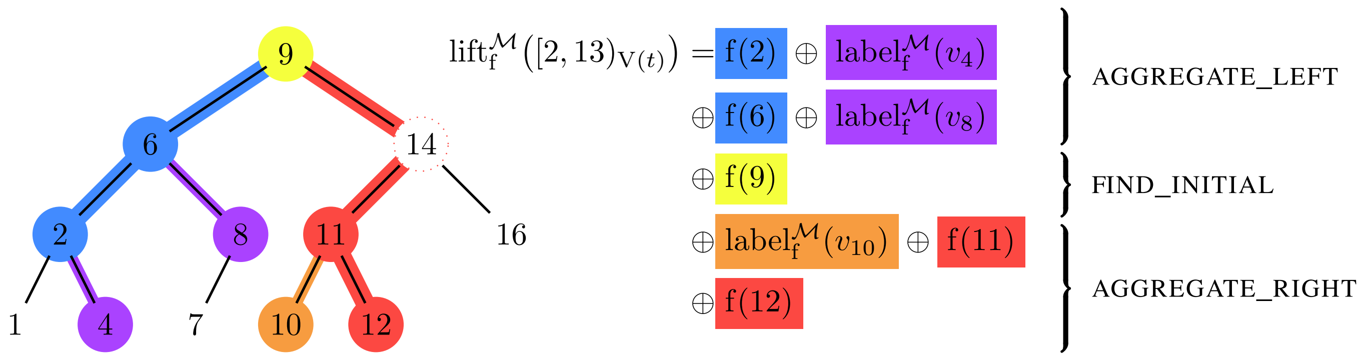

Fig. 4. Visualization of an exemplary tree traversal as performed by algorithm 1 to compute $\mathrm{lift}_{\mathrm{f}}^{\mathcal{M}}\big([2,13)_{\mathrm{V}(t)}\big)$. Notice that v_7 need not be visited, as its contribution to the accumulated value is already part of $\mathrm{label}_{\mathrm{f}}^{\mathcal{M}}(v_8)$.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

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 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[3] $\frac{1}{|D|}$. $\mathrm{h}(u)$ 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 monoidical 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.

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 $\mathrm{label}_{\lambda x.1}^{\mathcal{N}}$. 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 $\mathrm{lift}_{\lambda x.1}^{\mathcal{N}}\big([x,y)_{X_i}\big)$ 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.

The maximum distance between two vertices in a balanced tree on n vertices is in $\mathcal{O}(\log(n))$. Processing any individual range this way thus requires $\mathcal{O}(\log(n))$ time, just like our previous approach. Notice however that, when traversing the tree for successive sorted ranges, every edge is traversed at most twice. The time complexity for traversing multiple ranges in sequence is thus at most in $\mathcal{O}(n)$. Using this

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

approach, we can hence bound the time complexity for a single communication round by $\mathcal{O}(\min(n_i, n_\triangle \cdot \log(n_i)))$. 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 $\mathbb{0}$.

Observe that the path from some lower boundary x to some upper boundary y consists of some (possibly zero) steps from x toward the root, and then some (possibly zero) steps toward y. In order to compute this path in constant space and time per step, we augment the data structure by adding to each vertex v a reference $v.\mathtt{p}$ to its parent (nil for the root), and the largest value stored in its subtree $v.\mathtt{m}$. 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 downward 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 uses fingerprints of sets for probabilistic equality checking: we assume sets with equal fingerprints to be equal. Synchronization can thus become faulty if it involves unequal sets with equal fingerprints. If the universe of possible fingerprints is chosen large enough, and the distribution of fingerprints of randomly chosen sets is randomly distributed within that universe, the probability for this to occur becomes negligible.

Random distribution of input sets is however a very strong assumption. In this section, we examine how to mitigate a malicious adversary that can influence the sets to be fingerprinted, with the goal of causing fingerprint collisions and consequently triggering faulty behavior of the system.

A. Impact of Hash Collisions

We can generally distinguish between malicious actors in two different positions: those who can actively impact the contents of the data structure to be synchronized, and those who passively relay updates and need to search for a collision within the available data. As a set of size n has 2^n subsets, if fingerprints are bit strings of length k, then by the pigeonhole principle a fingerprint collision can be found within any set of size at least $k+1$.

An attack against the fingerprinting scheme by an active adversary can involve computing many fingerprints and adding the required items to the set once a collision has been found.

Algorithm 2

Require: $x \prec y$

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

Such an attack is not usable by the passive adversary. We will primarily focus on discussing active adversaries, as they are strictly more powerful than passive ones. Yet it should be kept in mind that passive adversaries can be more common in certain settings, particularly in peer-to-peer systems: if a node is interested in synchronizing a data structure, it probably trusts the source of the data, otherwise it would have little reason for expending resources on synchronization. The data may however be synchronized not with the original source but with completely untrusted nodes.

Fingerprint collisions result in parts of the data structure not being synchronized, so information is being withheld from one or both of the synchronizing 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 cases in which a malicious node can do actual damage by finding a collision are those where it supplies data to two honest nodes such that these two nodes perform faulty synchronization amongst each other. 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 run the synchronization protocol, 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.

There are some qualitative arguments that even if an adversary finds a fingerprint collision, the impact is rather low. 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 $S_B = [x, y)_{X_B}$.

If the adversary has found such sets, this provides still no guarantee that the range $[x, y)_{X_i}$ is being compared during the synchronization session of $\mathcal{A}$ and $\mathcal{B}$. In particular, there is no need for $\mathcal{A}$ and $\mathcal{B}$ to choose the range boundaries that occur in a protocol run deterministically. They can, for example, 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, while also making it impossible for an attacker to make sure that a given hash collision will impact a given synchronization session. In order to minimize the probability that a particulr collision affects a given protocol run, the b subrange boundaries can be chosen fully at random. The expected number of communication rounds is in $\mathcal{O}(\log(n))$ with high probability, as it corresponds to the height of a randomly chosen b-complete tree, which can be expected to be within a constant factor of $\log_b(n)$ [Dev90].

Another factor mitigating the impact of an adversary finding fingerprint collisions is the communication with other, non-colluding nodes. A fourth party could send some $u \in U, x \preceq u \prec y$ to $\mathcal{A}$ or $\mathcal{B}$ before $\mathcal{A}$ and $\mathcal{B}$ synchronize, disrupting the collision.

Finally, in systems where nodes repeatedly synchronize with different other nodes, a single fingerprint collision in a single synchronization session would merely delay propagation of information rather than stop it completely. Peer-to-peer systems communicating on a random overlay network in particular fall into this category. A malicious actor with enough control over the communication of other nodes to guarantee a tangible benefit from fingerprint collisions can likely disrupt operation of the network more effectively by exercising that control than by sabotaging synchronization.

All of these arguments are however purely qualitative and should as such be taken into account with caution, they are not a substitute for quantitative cryptographic analysis. A strong attacker might be able to find many pairs of sets of colliding fingerprints, or many sets that all share the same fingerprint, and none of the above arguments consider these cases.

B. Cryptographically Secure Fingerprints

For stronger guarantees, we thus look at cryptographically secure fingerprint functions that make it computationally infeasible for an adversary to find inputs that lead to faulty synchronization.

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. This alone is unfortunately not sufficient, as demonstrated by xor as the monoid function: [BM97] shows how to reduce the problem of finding a collision to that of solving a system of linear equations over the finite field on two elements (in which xor is the additive operation), which can be done in cubic time.

We now present further monoids that have been studied in the construction of secure hash functions. The seminal [BM97] considers secure hash functions for strings such that the hash of the concatenation of two strings can be efficiently computed from the hashes of the two strings. By considering sets as strings of their items in ascending order, the functions studied in [BM97] can also be applied to our sets. After demonstrating the unsuitability of xor, the authors consider addition modulo the total number of possible hashes, and the multiplicative group $\mathbb{Z}_n^*$, the group yielded by multiplication modulo n on the set $\{x \in [0, n)_{\mathbb{N}} \mid x \text{ is coprime to } n\}$.

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.

For addition, the balance problem is as hard as subset sum, which was at the time of publication conjectured to be sufficiently hard. Wagner showed however in [Wag02] how to solve the balance problem in subexponential time for addition. [MGS15] suggests addition for combining SHA-3 [Dwo15] digests, and proposes using fingerprints of length between 2688 and 4160 or 6528 to 16512 bits to achieve security levels of 128 or 256 bit respectively against Wagner's attack. [Lyu05] gives an improvement over Wagner's attack finding collisions in $\mathcal{O}(2^{n^{\epsilon}})$ for arbitrary $\epsilon < 1$, further weakening addition as a choice of monoid operation.

For multiplication, the balance problem is as hard as the discrete logarithm problem in the group. This is a more "traditional" hardness assumption than subset sum; there are groups for which no efficient algorithm is known. The main drawback is that multiplication is less efficient to compute than addition. [SMBA10] includes a comparison between the performance of addition and multiplication for incremental hashing; the additive hash outperforms the multiplicative one by two orders of magnitude, even though the additive hashes use longer digests to account for Wagner's attack.

When fingerprints are frequently sent over the network, longer computation times might be preferable over longer hashes. Fingerprints based on multiplication nevertheless need larger digests than traditional, non-incremental hash functions; [MSTA17] suggests fingerprints of 3200 bit to achieve 128 bit security.

[BM97] also proposes a fourth monoid based on lattices. [LKMW19] give a specific instantiation providing 200 bits of security with fingerprints of size $16 \cdot 1024 = 16384$ bit.

[CDVD+03] identifies the hash functions of [BM97] to be *multiset homomorphic*:

Definition 10 (Monoid Homomorphism)**.** 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 (Multiset Homomorphic Hash Function)**.** 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.

Beyond the multiset homomorphic hash functions introduced in [BM97], [CNQ09] provides a construction based on RSA, and [MSTA17] provides an efficient construction based on elliptic curves.

Because multiset union is commutative, so is necessarily any multiset homomorphic hash function. We do not require commutativity for our fingerprints however. A typical associative but not commutative operation is matrix multiplication. Study of a family of hash functions based on multiplication of invertible matrices was initiated in [Zém91]. The security of these hash functions is related to solving hard graph problems on the Cayley graph of the matrix multiplication group. [PQ+11] gives an overview about the general principles and the security aspects of Cayley hash functions.

While [TZ94], an improvement over the originally proposed scheme, has been successfully attacked in [GIMS11] and [PQ10], there are several modifications such as [Pet09][BSV17][Sos16] for which no attacks are known; and [MT16] shows random self-reducibility for Cayley hash functions.

Aside from Cayley hashes, we are not aware of any non-commutative monoids used for hashing. Notice that hash functions based on non-commutative groups (such as Cayley hashes) still have more structure than we need, as we don't require existence of inverse elements. Suitable hash functions can thus be located in a more general design space than studied in any literature we know of.

Regardless of the choice of monoid, the reconciliation protocol can exchange hashes of fingerprints rather than exchanging the 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. Despite its comparative simplicity, its profile of complexity guarantees is not strictly worse (nor better) than that of any other approach. While the logarithmic number of communication rounds is a significant drawback, no other approach achieves computational complexity proportional to only the size of the symmetric difference. The option of allowing one endpoint to perform reconciliation with a constant amount of working memory is unique as well. These two factors can allow reconciliation in resource-constrained settings in which no other approach can function. Furthermore, 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.

In this article, we have deliberately restricted our focus to providing a comprehensive overview of the very core issues of the range-based approach. There is a large number of engineering issues and possible generalizations, for example,

- generalizing to higher-dimensional ranges, backed by a kd-tree [Ben75],
- 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

Albus Dumbledore (anonymized for peer review) contributed the idea of hashing monoid values before transmission

in order to use large monoid universes for secure fingerprints without affecting the number of transmitted bits.

References

[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.

[BBJ+16] Michael A Bender, Jonathan W Berry, Rob Johnson, Thomas M Kroeger, Samuel McCauley, Cynthia A Phillips, Bertrand Simon, Shikha Singh, and David 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*, pages 289–302, 2016.

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

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[BLL00] Ahto Buldas, Peeter Laud, and Helger Lipmaa. Accountable certificate management using undeniable attestations. In *Proceedings of the 7th ACM Conference on Computer and Communications Security*, pages 9–17, 2000.

[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.

[CL02] Jan Camenisch and Anna Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In *Annual International Cryptology Conference*, pages 61–76. Springer, 2002.

[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.

[Gol09] Daniel Golovin. B-treaps: A uniquely represented alternative to b-trees. In *International Colloquium on Automata, Languages, and Programming*, pages 487–499. Springer, 2009.

[GT00] Michael T Goodrich and Roberto Tamassia. Efficient authenticated dictionaries with skip lists and commutative hashing. Technical report, Technical Report, Johns Hopkins Information Security Institute, 2000.

[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.

[Lyu05] Vadim Lyubashevsky. The parity problem in the presence of noise, decoding random linear codes, and the subset sum problem. In *Approximation, randomization and combinatorial optimization. Algorithms and techniques*, pages 378–389. Springer, 2005.

[Mer89] Ralph C Merkle. A certified digital signature. In *Conference on the Theory and Application of Cryptology*, pages 218–238. Springer, 1989.

[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.

[MND+04] Charles Martel, Glen Nuckolls, Premkumar Devanbu, Michael Gertz, April Kwong, and Stuart G Stubblebine. A general model for authenticated data structures. *Algorithmica*, 39(1):21–41, 2004.

[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.

[NN00] Moni Naor and Kobbi Nissim. Certificate revocation and certificate update. *IEEE Journal on selected areas in communications*, 18(4):561–570, 2000.

[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.

[PSTY13] Charalampos Papamanthou, Elaine Shi, Roberto Tamassia, and Ke Yi. Streaming authenticated data structures. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 353–370. Springer, 2013.

[PT89] William Pugh and Tim Teitelbaum. Incremental computation via function caching. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 315–328, 1989.

[PTT11] Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Optimal verification of operations on dynamic sets. In *Annual Cryptology Conference*, pages 91–110. Springer, 2011.

[PTT16] Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Authenticated hash tables based on cryptographic accumulators. *Algorithmica*, 74(2):664–712, 2016.

[Pug90] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[SA96] Raimund Seidel and Cecilia R Aragon. Randomized search trees. *Algorithmica*, 16(4):464–497, 1996.

[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.

[Sny77] L. Snyder. On uniquely represented data strauctures. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 142–146, Los Alamitos, CA, USA, oct 1977. IEEE Computer Society. URL: https://doi.ieeecomputersociety.org/10.1109/SFCS.1977.22, `doi:10.1109/SFCS.1977.22`.

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

[ST94] Rajamani Sundar and Robert E Tarjan. Unique binary-search-tree representations and equality testing of sets and sequences. *SIAM Journal on Computing*, 23(1):24–44, 1994.

[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.

[WC81] Mark N Wegman and J Lawrence Carter. New hash functions and their use in authentication and set equality. *Journal of computer and system sciences*, 22(3):265–279, 1981.

[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.