Streaming Authenticated Data Structures

Charalampos Papamanthou[1], Elaine Shi[2], Roberto Tamassia[3], and Ke Yi[4]

[1] UC Berkeley, cpap@cs.berkeley.edu
[2] University of Maryland, elaine@cs.umd.edu
[3] Brown University, rt@cs.brown.edu
[4] The Hong Kong University of Science and Technology, yike@cse.ust.hk

Abstract. We consider the problem of streaming verifiable computation, where both a verifier and a prover observe a stream of n elements $x_1, x_2, \dots, x_n$ and the verifier can later delegate some computation over the stream to the prover. The prover must return the output of the computation, along with a cryptographic proof to be used for verifying the correctness of the output. Due to the nature of the streaming setting, the verifier can only keep *small local state* (e.g., logarithmic) which must be updatable in a streaming manner and with *no interaction* with the prover. Such constraints make the problem particularly challenging and rule out applying existing verifiable computation schemes.

We propose *streaming authenticated data structures*, a model that enables efficient verification of *data structure queries* on a stream. Compared to previous work, we achieve an *exponential improvement* in the prover's running time: While previous solutions have linear prover complexity (in the size of the stream), even for queries executing in sublinear time (e.g., set membership), we propose a scheme with $O(\log M \log n)$ prover complexity, where n is the size of the stream and M is the size of the universe of elements. Our schemes support a series of expressive queries, such as (non-)membership, successor, range search and frequency queries, over an ordered universe and even in higher dimensions. The central idea of our construction is a new authentication tree, called *generalized hash tree*. We instantiate our generalized hash tree with a hash function based on lattices assumptions, showing that it enjoys suitable algebraic properties that traditional Merkle trees lack. We exploit such properties to achieve our results.

1 Introduction

With the growing market of cloud computing, it is crucial to construct protocols that enable the verification of computation performed by untrusted servers. For example, when searching over our remotely stored Gmail inbox, it would be desirable if the search results could be accompanied by a cryptographic proof vouching for their correctness, e.g., that no email was omitted (either deliberately or not) from the answer.

We consider verifiable computation in a *streaming* setting, where the dataset outsourced is rapidly evolving (e.g., stock quotes, network flows, sensor streams), and the verifier can only store a small state as the stream goes by (which should be efficiently updatable). Many prior verifiable computation schemes are unsuitable in the streaming setting: Some schemes require that the verifier (client) has access to all the data ahead of time, and performs some preprocessing (e.g., [17,40]) before outsourcing it to the prover (server). Other existing schemes allow a client to update the dataset through an interactive protocol between the client and the server (e.g., [35]). Particularly, since the client does not have sufficient local storage to store all the data, the client needs the server's help to update its local state. Unfortunately, requiring an interactive protocol for every update may be too expensive in a streaming setting. For example, consider a network traffic accounting application [14], where an ISP charges a customer based on the type and duration of its network flows. To enforce that the ISP is performing the accounting correctly, the ISP logs a customer's network flows such that the customer can later make queries to the logs to perform auditing (typically the customer does not have sufficient local storage to log all the flows). In such high link-speed settings, performing an interactive protocol with every packet or flow sent is very expensive.

Our contributions. We introduce *streaming authenticated data structures*, and design expressive and efficient schemes that *do not require any interaction between the client and server while the stream is observed*. In our scenario, streaming elements $x_1, x_2, \ldots, x_n$ are inserted into a data structure that is stored by the prover and the verifier stores and updates small local state of size $O(\log n)$. At any point of time, the verifier can send a data structure query to the prover, e.g., "return the predecessor of y" or "return the elements in the range $[a, b]$". Subsequently the prover can compute an answer and a proof in time $O(\log M \log n)$, where n is the size of the stream and M is the size of the ordered universe from where the elements are drawn. Our protocols are based on the difficulty of solving the *small integer solution problem* [30], and are the first ones to achieve the following properties simultaneously:

1. **Independence of prover and verifier**. While elements $x_1, x_2, \ldots, x_n$ are streaming, the verifier and the prover update their states *independently* and with *no interaction* (there is not even unidirectional communication[5]). The only eligible interaction occurs in the querying phase (which is inherent anyways). Such interaction consists only of one round as opposed to existing schemes in the statistical setting that can have up to a logarithmic number of interactions (e.g., [11,12]).
2. **Efficiency**. The running times of the verifier and the prover (for both updates and queries) as well as the proof size are all logarithmic in n and M, where n is the size of the stream and M is the size of the elements universe. In comparison, existing schemes in this setting (e.g., [10,11,12]) incur a linear proof generation overhead on the server, even for sublinear computations (see Section 1.1). We thus achieve an exponential improvement for many common queries in the prover's running time.
3. **Expressiveness**. Our construction supports a wide range of queries over an ordered universe, such as (non-)membership, successor, range search and frequencies. Our results can also be extended for d-dimensional elements, by applying well-known techniques from authenticated data structures [27]. To the best of our knowledge, our construction is the first streaming verifiable protocol to support such an extensive suite of queries with logarithmic prover *and* verifier complexity.

1.1 Related work

The research community has introduced *streaming verifiable computation*, both in the statistical [11,12] and the cryptographic setting [10,42], where a verifier and a prover observe a stream of n elements $x_1, x_2, \ldots, x_n$ and the verifier can delegate some computation over the stream to the prover. The protocols in [11,12] are probabilistic and

[5] Such a property is not achieved in the recent work of Schröder and Schröder [42].

use multiple interactions for verification, which reveal the secret randomness. Thus they support one-shot computation tasks, whereas we allow any number of queries.

Although most of the existing streaming verifiable protocols [10,11,12] are particularly efficient in terms of verifier complexity (e.g., (poly-)logarithmic in the size of the stream), the main shortcoming of all previous work (except for the work of Schröder and Schröder [42], see next paragraph) is the fact that the prover complexity is *linear* in the size of the stream, *even for sublinear (logarithmic) computations*, e.g., membership and range search queries on a stream of elements drawn from an ordered universe.[6] This significantly limits the applicability of these protocols since such an overhead introduces a large amount of latency, making them impractical for real-world deployment. Indeed, as Cormode *et al.* [12] point out in their experimental results,*"the chief bottleneck of these protocols seems to be $\mathcal{P}$'s time to make the proof"*. In this paper, we address the prover's complexity bottleneck for queries of practical importance (e.g., range search) and we design constructions supporting logarithmic prover *and* verifier complexity. Apart from a major theoretical improvement, we believe our protocols comprise a significant step towards practical verifiable streaming protocols.

The only efficient verifiable streaming protocol (with logarithmic prover complexity) was recently introduced by Schröder and Schröder [42]. Their construction can be applied only to *sequential* streams and hence it does not support data structures like dictionaries, where the relative order is decided depending on the element that is being streamed. Therefore it cannot be used to verify range queries efficiently. Moreover, there is unidirectional communication from the verifier to the prover per stream update.

Practical streaming verifiable computation has been also studied by the database community, with often increased worst-case complexities. Li *et al.* [25] considered verifying queries on a data stream with sliding windows, hence the verifier's space is proportional to the window size. The protocol of Papadopoulos *et al.* [33] verifies continuous queries over a stream, again requiring linear verifier space in the worst case.

Other related works such as *verifiable computation* [2,5,15,16,17,18,34,40] and *authenticated data structures* [13,20,21,31,35,36,37,38] are not directly applicable to the streaming setting or their application yields high complexities or interactive protocols.

Why common solutions fail. Traditional Merkle trees [28] (using collision resistant hash functions like SHA-2) can be used to provide very efficient proofs for membership and range search queries in logarithmic time. However, since the client cannot keep linear amount of local state, in order to update the digest when a new item is streamed, the client needs to interact with the server, where the server returns and proves the correctness of the path of the Merkle tree that is "touched" by the update (e.g., see [35]).

To avoid interaction with the prover, one could use accumulator-based solutions (e.g., [9]). Indeed, accumulators have the attractive property that a set of elements can be represented with a small digest that can be updated in a very straightforward way, e.g., by performing an exponentiation and with no interaction. However, this property comes at some significant cost, since proofs of membership can be computed in linear time (or time $O(n^\epsilon)$, e.g., see [36]) which translates into increased prover complexity.

Our work combines the merits of the above paradigms, enabling flexible updates with no interaction and at the same time achieving logarithmic prover complexity.

[6] For linear-time computations, linear complexity at the prover is acceptable, e.g., see [11].

1.2 Our techniques

The core idea of our scheme is a new primitive called *generalized hash tree*, which is a generalization of traditional Merkle trees [28]. Generalized hash trees can be instantiated with various collision resistant hash functions. In our construction we choose the hash function $h_n(\mathbf{x}, \mathbf{y}) = \mathbf{Lx} + \mathbf{Ry} \mod q$ (originally introduced in [1]), where $\mathbf{L}, \mathbf{R}$ are picked at random from $\mathbb{Z}_q^{k \times m}$ and $\mathbf{x}, \mathbf{y}$ are m-dimensional vectors with entries bounded by $n < q$. As in Merkle trees, we hierarchically apply this hash function over a binary tree—however this creates a problem, since the output of the above hash function is a vector of different dimensions and larger entries than its inputs. To overcome this problem, we devise a way to map the outputs back to the input domain. Although many mappings could be used, we choose one that maintains the function's homomorphic properties (Figure 4.2), allowing us to express the label of the root (i.e., the roothash) as the sum of well-defined functions of the leaves called *partial labels* (see Definition 16).

For example if the stream contains elements $\{3, 4, 6, 7\}$, we can simply express the label of the root of our hash tree as $\mathcal{L}(3) + \mathcal{L}(4) + \mathcal{L}(6) + \mathcal{L}(7)$, where $\mathcal{L}(x)$ is the partial label that depends only on x and on the public matrices $\mathbf{L}$ and $\mathbf{R}$, and which can be computed in logarithmic time. Clearly, such representation allows for efficient streaming updates of the verifier's state (label of the root), just like accumulators constructions [9]. More importantly (and unlike accumulators constructions), a proof for any element can still be computed in logarithmic time, by having the prover maintain an appropriate Merkle-tree-like authenticated data structure.

2 Definitions

We now present definitions for streaming authenticated data structures (SADS[7]). Our definitions are similar to the ones given by Chung *et al.* [10] for streaming delegation, adjusted to the data structures setting. We denote with k the security parameter and with $n = \mathsf{poly}(k)$ an upper bound on the size of the stream.[8] PPT stands for *probabilistic polynomial-time* and $\mathsf{neg}(k)$ is a negligible function, i.e., a function less than $1/p(k)$, for all polynomials $p(k)$. Finally we define $[n] = \{0, 1, \ldots, n\}$.

Definition 1 (SADS scheme). *Let D be* any *data structure that supports queries q and updates* upd. *An SADS (streaming authenticated data structure) scheme $\mathcal{A}$ is a collection of the following six PPT algorithms:*

1. $\mathsf{pk} \leftarrow \mathsf{genkey}(1^k, n)$*: On input the security parameter k and an upper bound n on the size of the stream, it outputs a public key* pk*;*
2. $\{\mathsf{auth}(D_0), d_0\} \leftarrow \mathsf{initialize}(D_0, \mathsf{pk})$*: On input an empty data structure D_0 and the public key* pk*, it computes the authenticated data structure* $\mathsf{auth}(D_0)$ *and the respective state d_0 of it;*
3. $d_{h+1} \leftarrow \mathsf{updateVerifier}(\mathsf{upd}, d_h, \mathsf{pk})$*: On input an update* upd *to data structure D_h, the current state d_h and the public key* pk*, it outputs the updated state d_{h+1} (run by verifier);*
4. $\{D_{h+1}, \mathsf{auth}(D_{h+1})\} \leftarrow \mathsf{updateProver}(\mathsf{upd}, D_h, \mathsf{auth}(D_h), \mathsf{pk})$*: On input an update* upd *to data structure D_h, the authenticated data structure* $\mathsf{auth}(D_h)$ *and the*

[7] This acronym has also been used by Pappas *et al.* [39] to denote a private search system.
[8] Otherwise (i.e., if n is not $\mathsf{poly}(k)$) the server might need exponential space.

public key pk*, it outputs the updated data structure* D_{h+1} *along with the updated authenticated data structure* $\mathsf{auth}(D_{h+1})$ *(run by prover);*

5. $\{\alpha(q), \Pi(q)\} \leftarrow \mathsf{query}(q, D_h, \mathsf{auth}(D_h), \mathsf{pk})$*: On input a query* q *on data structure* D_h*, the authenticated data structure* $\mathsf{auth}(D_h)$ *and the public key* pk*, it returns the answer* $\alpha(q)$ *to the query, along with a proof* $\Pi(q)$ *(run by prover);*
6. $\{1, 0\} \leftarrow \mathsf{verify}(q, \alpha(q), \Pi(q), d_h, \mathsf{pk})$*: On input a query* q*, an answer* $\alpha(q)$*, a proof* $\Pi(q)$ *for query* q*, a digest* d_h *and the public key* pk*, it outputs either* 1 *(accepts) or* 0 *(rejects) (run by verifier);*

As part of the data structure specification (and not of the above definition), we also define the algorithm $\{0, 1\} \leftarrow \mathsf{check}(q, \alpha, D_h)$ such that it outputs 1 if and only if α is the correct answer to query q on data structure D_h (otherwise it outputs 0).

Note that there is *no secret key in our definition*, supporting in this way a stronger definition with *public verifiability*, as opposed to other verifiable streaming constructions that appear in the literature [10,42], where the verifier's state needs to be secret.

There are two properties that an SADS scheme should satisfy, namely *correctness* and *security* (as in signature schemes definitions).

Definition 2 (Correctness). *Let* $\mathcal{A}$ *be an SADS scheme consisting of the set of algorithms* $\{\mathsf{genkey}, \mathsf{initialize}, \mathsf{updateVerifier}, \mathsf{updateProver}, \mathsf{query}, \mathsf{verify}\}$*. We say that the SADS scheme* $\mathcal{A}$ *is* correct *if, for all* $k \in \mathbb{N}$*, for all* pk *output by algorithm* genkey*, for all* $D_h, \mathsf{auth}(D_h), d_h$ *output by one invocation of* $\mathsf{initialize}$ *followed by polynomially-many invocations of* $\mathsf{updateVerifier}$ *and* $\mathsf{updateProver}$*, where* $h \geq 0$*, for all queries* q *and for all* $\Pi(q), \alpha(q)$ *output by* $\mathsf{query}(q, D_h, \mathsf{auth}(D_h), \mathsf{pk})$*, with all but negligible probability* $\mathsf{neg}(k)$*, it holds that* $1 \leftarrow \mathsf{verify}(q, \Pi(q), \alpha(q), d_h, \mathsf{pk})$.

Definition 3 (Security). *Let* $\mathcal{A}$ *be an SADS scheme consisting of the set of algorithms* $\{\mathsf{genkey}, \mathsf{initialize}, \mathsf{updateVerifier}, \mathsf{updateProver}, \mathsf{query}, \mathsf{verify}\}$*,* k *be the security parameter,* D_0 *be the empty data structure and* $\mathsf{pk} \leftarrow \mathsf{genkey}(1^k)$*. Let also* Adv *be a PPT adversary and let* d_0 *be the state output by* $\mathsf{initialize}(D_0, \mathsf{pk})$.

- *(Update) For* $i = 0, \ldots, h-1 = \mathsf{poly}(k)$*,* Adv *picks the update* upd_i *to data structure* D_i*. Let* $d_{i+1} \leftarrow \mathsf{updateVerifier}(\mathsf{upd}_i, d_i, \mathsf{pk})$ *be the new state corresponding to the updated data structure* D_{i+1}.
- *(Forge)* Adv *outputs a query* q*, an answer* α *and a proof* Π.

We say that the SADS scheme $\mathcal{A}$ *is* secure *if for all* $k \in \mathbb{N}$*, for all* pk *output by algorithm* genkey*, and for any PPT adversary* Adv *it holds that*

$$\Pr\left[\begin{array}{r}\{q, \Pi, \alpha\} \leftarrow \mathsf{Adv}(1^k, \mathsf{pk});\ 1 \leftarrow \mathsf{verify}(q, \alpha, \Pi, d_h, \mathsf{pk});\\ 0 \leftarrow \mathsf{check}(q, \alpha, D_h).\end{array}\right] \leq \mathsf{neg}(k)\,. \qquad (2.1)$$

3 Small integer solution problem

The security of our constructions is based on the hardness of the *small integer solution* problem, as given in the following definition:

Definition 4 (Problem $\mathsf{SIS}_{q,\mu,\beta}$). *Given an integer* q*, a matrix* $\mathbf{M} \in \mathbb{Z}_q^{k \times \mu}$ *picked uniformly at random (where* $\mu \geq k$*) and a real* β*, find an integer vector* $\mathbf{z} \in \mathbb{Z}^{\mu} \backslash \{\mathbf{0}\}$ *such that* $\mathbf{Mz} = \mathbf{0} \mod q$ *and* $\|\mathbf{z}\| \leq \beta$.

For certain parameters, Micciancio and Peikert [29] proved that SIVP_γ (shortest independent vector problem [41]), a hard problem in lattices, reduces to $\mathsf{SIS}_{q,\mu,\beta}$ for $\gamma = \mathsf{poly}(k)$. In the following we state an immediate corollary of Theorem 1.1 in [29]:

Corollary 1 (Reducing SIVP_γ to $\mathsf{SIS}_{q,\mu,\beta}$ [29]). *Let $\mathsf{SIS}_{q,\mu,\beta}$ be an instance of the small integer solution problem. Let also β, μ, q be $\mathsf{poly}(k)$, where q is a prime such that $q \geq \beta \cdot k^\delta$ for some $\delta > 0$. $\mathsf{SIS}_{q,\mu,\beta}$ is as hard as approximating the problem SIVP_γ in the worst case to within certain $\gamma = \frac{\beta}{k^\delta} \cdot O(\beta\sqrt{k} \cdot \mathsf{poly}(\log k))$.*

For exponential values of γ, i.e., $\gamma = 2^{O(k)}$, one can use the LLL algorithm [24] and solve the SIVP_γ problem in polynomial time. However, for polynomial γ, no efficient algorithm is known to date, even for factors slightly smaller than exponential [41]. Therefore, for the parameters of Corollary 1, $\mathsf{SIS}_{q,\mu,\beta}$ is also hard, leading to the following assumption:

Assumption 1 (Hardness of $\mathsf{SIS}_{q,\mu,\beta}$) *Let k be the security parameter and $\mathsf{SIS}_{q,\mu,\beta}$ be an instance of the small integer solution problem. Let also β, μ, q be $\mathsf{poly}(k)$, where q is a prime such that $q \geq \beta \cdot k^\delta$ for some $\delta > 0$. There is no PPT algorithm for solving $\mathsf{SIS}_{q,\mu,\beta}$, except with negligible probability $\mathsf{neg}(k)$.*

3.1 Setting the parameters q, μ and β

For the application we are considering in this paper, we are using an instance of the problem $\mathsf{SIS}_{q,\mu,\beta}$ where β (i.e., the norm of the solution vector) takes polynomially-large values depending on a polynomially-bounded application parameter n (n will be the size of the stream). Specifically we are going to use the parameters q, μ, β as set by the algorithm $\mathsf{parameters}(1^k, n)$ in Figure 3.1.

We note here that the parameters in Figure 3.1 comply with Corollary 1: First, as $n = \mathsf{poly}(k)$, all q, μ, β are $\mathsf{poly}(k)$. Second, $q/\sqrt{\lceil \log q \rceil} \geq \sqrt{2} \cdot n \cdot k^{0.5+\delta} \Leftrightarrow q \geq \beta \cdot k^\delta$, since $\beta = n\sqrt{\mu}$ and $\mu = 2k\lceil \log q \rceil$. Also note that there is always a prime $q = \Theta(n \cdot k^{0.5+\delta} \cdot \sqrt{\log k})$ satisfying the inequality above, for some $\delta > 0$.

3.2 The hash function

Our construction uses a hash function that is a syntactic modification (it accepts two inputs instead of one) of the collision resistant hash function presented by Micciancio and Regev [30], following seminal work by Ajtai [1] and Goldreich *et al.* [19]. The security of our function is based on the hardness of $\mathsf{SIS}_{q,\mu,\beta}$, using the parameters by Micciancio and Peikert [29], as shown above. We note here that a similar two-input hash function was also used to build a string commitment scheme by Kawachi *et al.* [23].

Definition 5 (Hash function [29,30]). *Let k be the security parameter, $n = \mathsf{poly}(k)$ and q, μ, β be the parameters output by algorithm $\mathsf{parameters}(1^k, n)$. Set $m = \frac{\mu}{2}$. Let also $\mathbf{L}, \mathbf{R} \in \mathbb{Z}_q^{k \times m}$ be two $k \times m$ matrices picked uniformly at random. We define the function $h_n : [n]^m \times [n]^m \to \mathbb{Z}_q^k$ as $h_n(\mathbf{x}, \mathbf{y}) = \mathbf{L} \cdot \mathbf{x} + \mathbf{R} \cdot \mathbf{y} \mod q$.*

Theorem 1 (Collision resistance [29,30]). *Let k be the security parameter, $n = \mathsf{poly}(k)$ and $\{q, \mu, \beta\} \leftarrow \mathsf{parameters}(1^k, n)$. Set $m = \frac{\mu}{2}$. Let also $\mathbf{L}, \mathbf{R} \in \mathbb{Z}_q^{k \times m}$ be matrices picked uniformly at random. Assuming hardness of $\mathsf{SIS}_{q,\mu,\beta}$ (see Assumption 1), there*

Algorithm $\{q, \mu, \beta\} \leftarrow \mathsf{parameters}(1^k, n)$**:** For $n = \mathsf{poly}(k)$, let q be the smallest prime satisfying $q/\sqrt{\lceil \log q \rceil} \geq \sqrt{2} \cdot n \cdot k^{0.5+\delta}$ for some $\delta > 0$. Set $\mu = 2k\lceil \log q \rceil$ and $\beta = n\sqrt{\mu}$.

Fig. 3.1. Setting the parameters of $\mathsf{SIS}_{q,\mu,\beta}$ as a function of the application parameter n.

is no PPT algorithm that outputs two distinct pairs of vectors $(\mathbf{x}_1, \mathbf{y}_1) \in [n]^m \times [n]^m$ *and* $(\mathbf{x}_2, \mathbf{y}_2) \in [n]^m \times [n]^m$ *such that* $\mathbf{L} \cdot \mathbf{x}_1 + \mathbf{R} \cdot \mathbf{y}_1 = \mathbf{L} \cdot \mathbf{x}_2 + \mathbf{R} \cdot \mathbf{y}_2 \mod q$*, except with negligible probability* $\mathsf{neg}(k)$.

3.3 Binary representations

For our constructions, we are going to need *binary representations* of vectors:

Definition 6 (Binary representation of scalars). *Let* $\tau = \lceil \log q \rceil$*. Denote with* $\mathbf{b}(a) = [\mathbf{b}_0, \mathbf{b}_1, \ldots, \mathbf{b}_{\tau-1}]^\intercal \in \{0,1\}^\tau$ *the binary representation of* $a \in \mathbb{Z}_q$*, i.e.,* $a = \sum_{i=0}^{\tau-1} \mathbf{b}_i 2^i$.

Note now that Definition 6 can be naturally extended for vectors $\mathbf{a} \in \mathbb{Z}_q^k$: For $i = 0, \ldots, k-1$, $\mathbf{a}_i$ is mapped to the respective τ entries $\mathbf{b}(\mathbf{a}_i)$ in the resulting vector $\mathbf{b}(\mathbf{a})$:

Definition 7 (Binary representation of vectors). *Let* $\mathbf{a} = [\mathbf{a}_0, \mathbf{a}_1, \ldots, \mathbf{a}_{k-1}]^\intercal \in \mathbb{Z}_q^k$*. We denote with* $\mathbf{b}(\mathbf{a}) = [\mathbf{b}(\mathbf{a}_0), \mathbf{b}(\mathbf{a}_1), \ldots, \mathbf{b}(\mathbf{a}_{k-1})]^\intercal \in \{0,1\}^{k \cdot \tau}$ *(*$\tau = \lceil \log q \rceil$*) the binary representation of* $\mathbf{a} \in \mathbb{Z}_q^k$*, where* $\mathbf{b}(\mathbf{a}_i)$ *is defined in Definition 6.*

For example, if $k = 2$, $q = 8$ and $\mathbf{a} = [6, 3]^\intercal \in \mathbb{Z}_8^2$, then $\mathbf{b}(\mathbf{a}) = [0, 1, 1, 1, 1, 0]^\intercal$, since $\mathbf{b}(6) = [0, 1, 1]^\intercal$ and $\mathbf{b}(3) = [1, 1, 0]^\intercal$.

4 Generalized hash trees

The main primitive of our construction is what we call a *generalized hash tree*. A generalized hash tree has several differences from the traditional Merkle hash tree [28].

First we recall that a Merkle hash tree is a labeled binary tree T where the label $\lambda(w)$ of every node w is the collision resistant hash (e.g., a SHA-2 hash) of the labels $\lambda(u)$ and $\lambda(v)$ and of its children u and v, i.e., $\lambda(w) = h(\lambda(u), \lambda(v))$. When function h is applied recursively on all the nodes of the tree, the label $\lambda(r)$ of the root r has the following property: A PPT adversary cannot find two different data sets at the leaves that produce the same label at the root of a Merkle tree.

In our work, instead of using a hash function such as SHA-2 that lacks algebraic structure, we employ the hash function h_n described in Section 3. However, we cannot directly apply this function since its domain (vectors in $[n]^m$) is different from its range (vectors in $\mathbb{Z}_q^k$). Generalized hash trees, introduced in the next section, provide a way to overcome this domain-range discrepancy problem.

4.1 Defining generalized hash trees

Let $h : \mathcal{D} \times \mathcal{D} \rightarrow \mathcal{R}$ be a collision resistant hash function accepting two inputs that take values from domain $\mathcal{D}$ and outputting a value in a *different* range $\mathcal{R}$. Generalized hash trees solve the domain-range discrepancy problem (and at the same time maintain the authentication and algorithmic properties of traditional Merkle hash trees [28]) as follows: They require that the labels $\lambda(u) \in \mathcal{D}$ and $\lambda(v) \in \mathcal{D}$ of the children u and v hash to a *deterministic* and *easily computable* projection function $f : \mathcal{D} \to \mathcal{R}$ of the label $\lambda(w) \in \mathcal{D}$ of the parent w, i.e., $f(\lambda(w)) = h(\lambda(u), \lambda(v))$.

An immediate implication of this property is that the labels of a generalized hash tree are generally *not* uniquely determined by the labels of the leaves: In the above example, $\lambda(w)$ can be any f-preimage of $h(\lambda(u), \lambda(v))$. However, the collision resistant property of Merkle trees is still true: Any two valid hash trees representing different data sets at the leaves but with the same root label yield a collision to the underlying hash function. We now continue with defining generalized hash trees formally. We first need the following definition for representing binary trees.

Definition 8 (Full binary tree). *A* full binary tree T *is a non-empty tree where every internal node has two children. It is represented with set of binary strings, where ϵ is the empty string representing the root of T and $w0$ and $w1$ are the string representations of the left and right children of a node having string representation w.*

For example, a full binary tree with five nodes is $T = \{\epsilon, 0, 1, 00, 01\}$. Note that full binary trees need not be complete, i.e., not all leaves must lie at the same level.

Definition 9 (Labeled binary tree). *A* labeled binary tree (T, λ) *is a full binary tree T along with labels $\lambda(w)$ for all $w \in T$.*

Definition 10 (Generalized hash tree). *Given functions $h : \mathcal{D} \times \mathcal{D} \to \mathcal{R}$ and $f : \mathcal{D} \to \mathcal{R}$, a* generalized hash tree (T, λ, f, h) *is a labeled binary tree (T, λ) such that **(a)** for all $w \in T$, $\lambda(w) \in \mathcal{D}$; **(b)** for all internal nodes $w \in T$, $f(\lambda(w)) = h(\lambda(w0), \lambda(w1))$, where $w0$ and $w1$ are the left and right children of w respectively.*

Definition 11 (Tree collision). *A* tree collision *is a pair of two distinct generalized hash trees (T, λ, f, h) and (T, l, f, h) such that $\lambda(\epsilon) = l(\epsilon)$.*

We now give our main security theorem, establishing collision resistance for generalized hash trees. The proof is in the Appendix (see Section 7.1).

Theorem 2 (Collision resistance). *Let k be the security parameter, T be a full binary tree of $\mathsf{poly}(k)$ depth. If h is collision resistant, there is no PPT algorithm that can output a tree collision (T, λ, f, h) and (T, l, f, h), except with probability $\mathsf{neg}(k)$.*

4.2 An instantiation of generalized hash trees

In our application setting, we are using a *structured binary tree* which is a special case of the full binary tree from Definition 8:

Definition 12 (Structured binary tree). *Let M be a power of two. A* structured binary tree $T_\mathcal{C}$ *is a full binary tree T of $\log M$ levels where all the leaves lie at the last level of the tree, storing values $\mathcal{C} = [c_0, c_1, \ldots, c_{M-1}]$, where $c_i \in \mathbb{Z}_q$.*

Definition 13 (Range of a node). *Let w be a node of a structured binary tree $T_\mathcal{C}$. The set $\mathsf{range}(w)$ contains the leaves of the subtree of $T_\mathcal{C}$ rooted on w.*

In the following sections, we instantiate the generalized hash tree for a structured binary tree using the lattice-based hash function $h_n(\mathbf{x}, \mathbf{y}) = \mathbf{L} \cdot \mathbf{x} + \mathbf{R} \cdot \mathbf{y}$ from Definition 5, where $\mathcal{D} = [n]^m$ and $\mathcal{R} = \mathbb{Z}_q^k$—see Section 3 for the definition of all parameters k, n, m, q. We will also show which projection function f to use and how to compute the labels λ so that Definition 10 is satisfied.

Function $\mathbf{y} = f(\mathbf{x})$: Let $\tau = \lceil \log q \rceil$. On input a vector $\mathbf{x} \in [n]^m$, where $m = k \cdot \tau$, output a vector $\mathbf{y}$ of k entries such that each $\mathbf{y}_i$ $(i = 0, \ldots, k-1)$ is the number in $\mathbb{Z}_q$ represented by the radix-2 representation $[\mathbf{x}_{i\tau}, \mathbf{x}_{i\tau+1}, \ldots, \mathbf{x}_{(i+1)\tau-1}]^\mathsf{T}$, namely

$$\mathbf{y}_i = \sum_{j=0}^{\tau-1} \mathbf{x}_{i\tau+j} 2^j \mod q\,, \text{ for } i = 0, \ldots, k-1\,.$$

Fig. 4.2. The projection function f. It parses the input $\mathbf{x}$ as a vector of radix-2 representations and convers it to a vector $\mathbf{y}$ (of smaller dimension) storing the respective numbers in $\mathbb{Z}_q$.

4.3 The projection function f

The projection function $f : [n]^m \to \mathbb{Z}_q^k$ we use is very simple. It *parses* the input vector $\mathbf{x}$ as a radix-2 representation (i.e., a base-2 representation but not necessarily of binary coefficients) and converts it to the respective vector in $\mathbb{Z}_q^k$. We give the code of the function in Figure 4.2. We now have the following corollary for function f:

Corollary 2 (Applying function f to binary representations). *Let $\mathbf{a} \in \mathbb{Z}_q^k$. Then $f(\mathbf{b}(\mathbf{a})) = \mathbf{a}$, where $\mathbf{b}(\mathbf{a})$ is the binary representation of $\mathbf{a}$ defined in Definition 6.*

Clearly, function f is a linear function. This property (stated below) is crucial for proving that the labels (defined in Section 4.4) comply with Definition 10:

Corollary 3 (Linearity of function f). *Let $\mathbf{x} \in [n]^m$ and $\mathbf{y} \in [n]^m$ such that $\mathbf{x} + \mathbf{y} \in [n]^m$. Then $f(\mathbf{x} + \mathbf{y}) = f(\mathbf{x}) + f(\mathbf{y})$.*

4.4 Computing the labels

We now continue with defining the labels of the generalized hash tree (see Definition 16). Before that, we give some necessary definitions:

Definition 14. *Define the functions $g_0 : [n]^m \to [n]^m$ and $g_1 : [n]^m \to [n]^m$ such that $g_0(\mathbf{x}) = \mathbf{b}(\mathbf{L} \cdot \mathbf{x})$ and $g_1(\mathbf{x}) = \mathbf{b}(\mathbf{R} \cdot \mathbf{x})$. Also, for a bitstring $w = b_1 b_2 \ldots b_e$, define the function $g_w : [n]^m \to [n]^m$ as the composition $g_w(\mathbf{x}) = g_{b_1} \circ g_{b_2} \circ \ldots \circ g_{b_e}(\mathbf{x})$.*

Definition 15 (Partial labels of a node w). *Let $T_\mathcal{C}$ be a structured binary tree. The* partial label *of a leaf node v with respect to itself is defined as $\mathcal{L}_v(v) = \mathbf{1}$, where $\mathbf{1} = [1, 1, \ldots, 1]^\mathsf{T} \in [n]^m$. For every other node w of $T_\mathcal{C}$, and for every leaf $v \in \mathsf{range}(w)$, the* partial label *$\mathcal{L}_w(v)$ of w with respect to v is defined as $\mathcal{L}_w(v) = g_{v-w}(\mathbf{1})$, where $v - w$ is the result of removing prefix w from bitstring v.*

E.g., for a structured binary tree of 8 leaves, the partial label of the root wrt leaves 2 and 3 are $\mathcal{L}_\epsilon(2) = \mathbf{b}(\mathbf{L} \cdot \mathbf{b}(\mathbf{R} \cdot \mathbf{b}(\mathbf{L} \cdot \mathbf{1})))$ and $\mathcal{L}_\epsilon(3) = \mathbf{b}(\mathbf{L} \cdot \mathbf{b}(\mathbf{R} \cdot \mathbf{b}(\mathbf{R} \cdot \mathbf{1})))$ respectively.

Definition 16. *Let $T_\mathcal{C}$ be a structured binary tree, where $\mathcal{C} = [c_0, c_1, \ldots, c_{M-1}]$. For every node $w \in T_\mathcal{C}$ we define a function $\lambda(w) = \sum_{v \in \mathsf{range}(w)} c_v \cdot \mathcal{L}_w(v)$.*

Lemma 1. *Let $T_\mathcal{C}$ be a structured binary tree. If $\sum_{i=0}^{M-1} c_i \leq n$, then for all nodes $w \in T_\mathcal{C}$ it holds that $\lambda(w) \in [n]^m$, where $\lambda(w)$ is the function defined in Definition 16.*

Proof. Write $\lambda(w)$ as in Definition 16. Since $\sum_{i=0}^{M-1} c_i \leq n$ and the entries of each partial label $\mathcal{L}_w(v)$ are in $\{0,1\}$, it follows that $\lambda(w) \in [n]^m$. □

Lemma 2. *Let $T_\mathcal{C}$ be a structured binary tree. If $\sum_{i=0}^{M-1} c_i \leq n$, then $f(\lambda(w)) = \mathbf{L} \cdot \lambda(w0) + \mathbf{R} \cdot \lambda(w1)$, where $\lambda(w)$ is the function defined in Definition 16 and w is any internal node of $T_\mathcal{C}$.*

Proof. Let w be an internal node of the structured binary tree $T_\mathcal{C}$. Let $w0$ be its left child and $w1$ be its right child. Since $\sum c_i \leq n$, by Lemma 1, it is $\lambda(w) \in [n]^m$, so we can apply function f. Therefore we have

$$f(\lambda(w)) = f\left(\sum_{v \in \mathsf{range}(w)} c_v \cdot \mathcal{L}_w(v)\right) \quad \text{(Def. 16)}$$

$$= \sum_{v \in \mathsf{range}(w0)} c_v \cdot f\left(\mathcal{L}_w(v)\right) + \sum_{v \in \mathsf{range}(w1)} c_v \cdot f\left(\mathcal{L}_w(v)\right) \quad \text{(Cor. 3)}$$

$$= \sum_{v \in \mathsf{range}(w0)} c_v \cdot f\left(g_{w-v}(\mathbf{1})\right) + \sum_{v \in \mathsf{range}(w1)} c_v \cdot f\left(g_{w-v}(\mathbf{1})\right) \quad \text{(Def. 15)}$$

$$= \sum_{v \in \mathsf{range}(w0)} c_v \cdot f\left(g_0(g_{w0-v}(\mathbf{1}))\right) + \sum_{v \in \mathsf{range}(w1)} c_v \cdot f\left(g_1(g_{w1-v}(\mathbf{1}))\right) \quad \text{(Def. 14)}$$

$$= \sum_{v \in \mathsf{range}(w0)} c_v \cdot f\left(g_0(\mathcal{L}_{w0}(v))\right) + \sum_{v \in \mathsf{range}(w1)} c_v \cdot f\left(g_1(\mathcal{L}_{w1}(v))\right) \quad \text{(Def. 15)}$$

$$= \sum_{v \in \mathsf{range}(w0)} c_v \cdot f\left(\mathbf{b}(\mathbf{L} \cdot \mathcal{L}_{w0}(v))\right) + \sum_{v \in \mathsf{range}(w1)} c_v \cdot f\left(\mathbf{b}(\mathbf{R} \cdot \mathcal{L}_{w1}(v))\right) \quad \text{(Def. 14)}$$

$$= \sum_{v \in \mathsf{range}(w0)} c_v \cdot \mathbf{L} \cdot \mathcal{L}_{w0}(v) + \sum_{v \in \mathsf{range}(w1)} c_v \cdot \mathbf{R} \cdot \mathcal{L}_{w1}(v) \quad \text{(Cor. 2)}$$

$= \mathbf{L} \cdot \lambda(w0) + \mathbf{R} \cdot \lambda(w1)$ (Def. 16). This completes the proof. □

Theorem 3. *Let $T_\mathcal{C}$ be a structured binary tree. If $\sum_{i=0}^{M-1} c_i \leq n$, then $(T_\mathcal{C}, \lambda, f, h_n)$ is a generalized hash tree, where $h_n(\mathbf{x}, \mathbf{y}) = \mathbf{L} \cdot \mathbf{x} + \mathbf{R} \cdot \mathbf{y}$ is the function from Definition 5, λ is defined in Definition 16 and f is the function in Figure 4.2.*

Proof. It follows from Lemmas 1 and 2 and by Definition 10. □

4.5 Efficient updates of the labels

Note that Definition 16 enables very efficient updates of the label of any node, whenever a leaf value changes. For example, if $\lambda(\epsilon)$ is the label of the root of a generalized hash tree $(T_\mathcal{C}, \lambda, f, h_n)$ with eight leaves $\{0, 1, 2, \ldots, 7\}$ where $c_3 = 2$, $c_4 = c_6 = c_7 = 1$ and $c_0 = c_1 = c_2 = c_5 = 0$, then the root label $\lambda(\epsilon)$ can be expressed as $2\mathcal{L}_\epsilon(3) + \mathcal{L}_\epsilon(4) + \mathcal{L}_\epsilon(6) + \mathcal{L}_\epsilon(7)$. Particularly, each occurrence of an element i contributes $\mathcal{L}_\epsilon(i)$ (i.e., partial label of the root ϵ with respect to i) to the root label. Adding (or removing) an element x to the set is equivalent to adding $\mathcal{L}_\epsilon(x)$ (or $-\mathcal{L}_\epsilon(x)$) to $\lambda(\epsilon)$. It is also important to note that the partial labels (defined in Definition 15) required for such updates can be easily computed in polylogarithmic time:

Lemma 3. *The partial label $\mathcal{L}_w(v)$ can be computed in time $O(\log M \log^2 n)$.*

Proof. Computing $\mathcal{L}_w(v)$, by Definition 15, requires $O(\log M)$ recursive calls, each one of which involves: (a) computing a binary representation of k $O(\log q)$-bit numbers, which takes time $O(k \log q)$; (b) multiplying a $k \times O(k \log q)$ matrix with a vector of $O(k \log q)$ bits, which takes time $O(k \log^2 q)$. This completes the proof. □

5 Our SADS construction

Let $T_\mathcal{C}$ be a structured binary tree with M leaves corresponding to the universe of integer values $\mathcal{U} = \{0, 1, \ldots, M-1\}$. For our construction, we are using a generalized hash tree $(T_\mathcal{C}, \lambda, f, h_n)$ as described in the previous section, where λ is defined in Definition 16, f is the function in Figure 4.2, h_n is the hash function from Definition 5 and $\{c_0, c_1, \ldots, c_{M-1}\}$ correspond to the frequency of elements $\{0, 1, \ldots, M-1\}$ appearing in the stream. Note that even for an exponential value of M, the condition $\sum_{i=0}^{M-1} c_i \leq n$ of Theorem 3 still holds since for the elements x that do not appear in the stream it is $c_x = 0$. To store the generalized hash tree, we store only the labels that are defined on the paths from non-zero leaves to the root (all other labels are zero). This requires space proportional to $O(\nu \log M)$, where ν is the number of distinct element appearing in the stream. In this way, we avoid storing $O(M)$ space, which is prohibitive given the potential exponential universe size M.

Figure 5.3 presents our SADS scheme for frequency queries. We note that algorithms query and verify are the same for all generalized hash trees, unlike the update algorithms that are specific for the algebraic hash function h_n.

5.1 Range search queries

In this section we show how to support range search queries. The proof for a range search query $[x, y]$ simply contains the two proofs $\Pi(x)$ and $\Pi(y)$ as output by algorithms $\mathsf{query}(x, D_h, \mathsf{auth}(D_h), \mathsf{pk})$ and $\mathsf{query}(y, D_h, \mathsf{auth}(D_h), \mathsf{pk})$ respectively from Figure 5.3. It also contains the frequencies $\mathcal{C}_{xy} = \{c_{a_1}, c_{a_2}, \ldots, c_{a_s}\}$ of the reported range as an answer. Let now $\mathcal{R}_{xy} = \{a_1, a_2, \ldots, a_s\}$ denote the respective reported range that corresponds to $\mathcal{C}_{xy}$.

For verification, the proofs $\Pi(x)$ and $\Pi(y)$ are verified first by using algorithm verify from Figure 5.3. If this verification is successful, perform the following test (else reject): If for all labels $\lambda(v) \in \Pi(x) \cup \Pi(y)$ such that $\mathsf{range}(v) \cap \mathcal{R}_{xy}$ is not empty, the following relation (as in Definition 16)

$$\lambda(v) = \sum_{i \in \mathsf{range}(v) \cap \mathcal{R}_{xy}} c_i \cdot \mathcal{L}_v(i) \tag{5.3}$$

is true, output 1 (i.e., accept), else output 0 (i.e., reject). The above relation ensures that all the range (with the correct frequencies) has been reported, or otherwise, the adversary could find a collision. The above technique can be also used for verifying successor queries, where the reported range is empty.

We now give our final result stating the formal security guarantee of our algorithms, along with their detailed asymptotic performance. The correctness of our scheme follows easily by inspecting the algorithms, therefore its proof is omitted. The security proof and the proof of asymptotic performance are in the Appendix.

Algorithm $\mathsf{pk} \leftarrow \mathsf{genkey}(1^k, n)$**:** Call $\{q, \mu, \beta\} \leftarrow \mathsf{parameters}(1^k, n)$ from Figure 3.1, on input the security parameter k and a bound n on the size of the stream. Set $\mathsf{pk} = \{\mathbf{L}, \mathbf{R}, q, \mathcal{U}\}$, where $\mathcal{U}$ is a universe such that $|\mathcal{U}| = M$ and $\mathbf{L}, \mathbf{R}$ are picked uniformly at random from $\mathbb{Z}_q^m$ for $m = \frac{\mu}{2}$.

Algorithm $\{\mathsf{auth}(D_0), d_0\} \leftarrow \mathsf{initialize}(D_0, \mathsf{pk})$**:** Let D_0 be a structured binary tree $T_{\mathcal{C}}$ where $c_i = 0$ ($i = 0, \ldots, M-1$). The algorithm outputs the generalized hash tree $(T_{\mathcal{C}}, \lambda, f, h_n)$ as $\mathsf{auth}(D_0)$, where $\lambda(v) = \mathbf{0} \in [n]^m$ for all nodes v in $T_{\mathcal{C}}$. Also it outputs $d_0 = \mathbf{0} \in [n]^m$.

Algorithm $d_{h+1} \leftarrow \mathsf{updateVerifier}(x, d_h, \mathsf{pk})$**:** Let $x \in \mathcal{U}$ be the current element of the stream. The algorithm updates the local state by setting $d_{h+1} = d_h + \mathcal{L}_\epsilon(x)$, where ϵ is the root of $T_{\mathcal{C}}$ and $\mathcal{L}_\epsilon(x)$ is defined in Definition 15.

Algorithm $\{D_{h+1}, \mathsf{auth}(D_{h+1})\} \leftarrow \mathsf{updateProver}(x, D_h, \mathsf{auth}(D_h), \mathsf{pk})$**:** Let $x \in \mathcal{U}$ be the current element of the stream. The algorithm sets $c_x = c_x + 1$, outputting the updated tree $T_{\mathcal{C}}$. Let $v_\ell, \ldots, v_1$ be the path in $T_{\mathcal{C}}$ from node v_ℓ (v_ℓ stores c_x) to the child v_1 of the root ϵ of $T_{\mathcal{C}}$. Set

$$\lambda(v_i) = \lambda(v_i) + \mathcal{L}_{v_i}(x) \text{ for } i = \ell, \ell-1, \ldots, 1\,, \tag{5.2}$$

where $\mathcal{L}_{v_i}(x)$ is defined in Definition 15. The new authenticated data structure $\mathsf{auth}(D_{h+1})$ is the new generalized hash tree with the updated labels as computed in Equation 5.2.

Algorithm $\{\alpha(q), \Pi(q)\} \leftarrow \mathsf{query}(q, D_h, \mathsf{auth}(D_h), \mathsf{pk})$**:** Let q be a frequency query for element $x \in \mathcal{U}$. Set $\alpha(q) = c_x$ (note that if $c_x = 0$, x is not contained in the collection). Let $v_\ell, \ldots, v_1$ be the path in the structured binary tree $T_{\mathcal{C}}$ from node v_ℓ (v_ℓ stores the value c_x) to the child v_1 of the root ϵ of $T_{\mathcal{C}}$. Let also $w_\ell, \ldots, w_1$ be the *sibling nodes* of $v_\ell, \ldots, v_1$. Proof $\Pi(q)$ contains the ordered sequence of the pairs of labels belonging to the tree path from leaf v_ℓ to the root ϵ of the tree, i.e., the pairs $\{(\lambda(v_\ell), \lambda(w_\ell)), (\lambda(v_{\ell-1}), \lambda(w_{\ell-1})), \ldots, (\lambda(v_1), \lambda(w_1))\}$.

Algorithm $\{1, 0\} \leftarrow \mathsf{verify}(q, \alpha(q), \Pi(q), d_h, \mathsf{pk})$**:** Let q be a frequency query for element $x \in \mathcal{U}$. Parse $\Pi(q)$ as $\{(\lambda(v_\ell), \lambda(w_\ell)), \ldots, (\lambda(v_1), \lambda(w_1))\}$ and $\alpha(q)$ as c_x.
If $\lambda(v_\ell) \neq c_x\mathbf{1}$ or $\lambda(v_\ell), \lambda(w_\ell) \neq [n]^m$, output $\mathsf{0}$. Compute values $y_{\ell-1}, y_{\ell-2}, \ldots, y_0$ as $y_i = \mathbf{L} \cdot \lambda(v_{i+1}) + \mathbf{R} \cdot \lambda(w_{i+1})$ (if v_{i+1} is v_i's left child) or $y_i = \mathbf{R} \cdot \lambda(v_{i+1}) + \mathbf{L} \cdot \lambda(w_{i+1})$ (if v_{i+1} is v_i's right child). For $i = \ell-1, \ldots, 1$, if $f(\lambda(v_i)) \neq y_i$ **or** $\lambda(v_i), \lambda(w_i) \notin [n]^m$ output $\mathsf{0}$. If $f(d_h) \neq y_0$, output $\mathsf{0}$. Output $\mathsf{1}$.

Fig. 5.3. Algorithms of the SADS scheme for verifying frequency queries.

Theorem 4 (Streaming authenticated frequency with range search). *Let k be the security parameter, $n = \mathsf{poly}(k)$ be an upper bound on the size of a stream containing elements from an ordered universe $\mathcal{U}$ of size M, $\{q, \mu, \beta\} \leftarrow \mathsf{parameters}(1^k, n)$ and ν be the number of unique elements that have appeared in the stream. There exists a streaming authenticated data structure scheme for one-dimensional frequency queries and one-dimensional range queries (outputting the respective frequencies) such that: **(a)** It is correct according to Definition 2 and secure according to Definition 3 and assuming hardness of $\mathsf{SIS}_{q,\mu,\beta}$ (Assumption 1); **(b)** Algorithms* updateVerifier *and* updateProver *run in $O(\log M \log^2 n)$ time; **(c)** Algorithm* query *(both for frequency and range search queries) runs in $O(\log M \log n)$ time, outputting a proof of size $O(\log M \log n)$; **(d)** A frequency query can be verified in $O(\log M \log^2 n)$ time and a range search query* *can be verified in $O(s \log M \log^2 n)$ time, where s is the size of the output range;* ***(e)*** *The space required at the verifier is $O(\log n)$ and the space required at the prover is $O(\nu \log M \log n)$.*

Our algorithms can be extended to two (or multiple) dimensions by leveraging existing methods for multidimensional range queries [27], carefully adjusted in our framework. Due to space limitations, we defer such extensions to the full version of our paper.

6 Applications

In this section we present three applications of our construction.

Cryptographic accumulator with efficient witness generation. A cryptographic accumulator [4,6] allows one to hash a set of inputs into one short *accumulation value*, such that there is a witness that a given input was incorporated into the accumulator, and at the same time, it is infeasible to find a witness for a value that was not accumulated. In CRYPTO 2002, Camenisch and Lysyanskaya [9] introduced *dynamic accumulators*, that enable updating the accumulation value when inputs are dynamically added or deleted, such that the cost of an update is independent of the number of accumulated inputs. However, all dynamic accumulator constructions that appeared since then (e.g., [3,8,9,26,32]) share one common limitation: Computing a witness, in absence of the trapdoor information (which has many practical applications, e.g., [36]), takes at least linear time. We observe that our construction comprises a dynamic accumulator that does not have this limitation: Specifically, for a set of elements $\mathcal{X} \subseteq \{0, 1, \ldots, M-1\}$, our accumulation value, from Definition 16, is $\mathsf{acc}(\mathcal{X}) = \sum_{i \in \mathcal{X}} \mathcal{L}_\epsilon(i)$. To update the accumulation value with element y, one has to set $\mathsf{acc}(\mathcal{X}) = \mathsf{acc}(\mathcal{X}) + \gamma \cdot \mathcal{L}_\epsilon(y)$, where $\gamma \in \{1, -1\}$ depending on whether we add or remove y from the set. Our construction satisfies basic accumulator properties such as quasi-commutativity and efficient updates [9]. Moreover, one can use the generalized hash tree and compute witnesses in *logarithmic* time (see Theorem 4), as opposed to *linear* time.

Parallel online memory checking in the public key setting. Memory checking [7] studies the problem of cryptographically verifying the correctness of untrusted indexed storage by only storing small local memory. Many checkers with logarithmic *sequential* query complexity (number of reads and writes to the *untrusted* memory), e.g., [7,31,20], have appeared in the literature. However, *parallelizing* existing checker constructions can only be achieved in the secret key setting (e.g., see [22]). Checkers in the public key setting (e.g., [20]) cannot be naturally parallelized because they are traditionally implemented with Merkle trees [28]: Whenever a leaf value of the checker tree is written, the roothash can be updated only after the value of its child has been updated, which is an inherently sequential process. Our generalized hash tree can be used to overcome this barrier, yielding a *parallel* memory checker in the public key setting (recall we do not use any secret keys in our construction). This is because in our construction, whenever a leaf value i is written, changing its value from c to c', we can execute algorithm $\mathsf{updateProver}$ from Section 5 in parallel (note the loop described in Relation 5.2 is fully parallelizable) and by accessing *only* the old value c, thus issuing $O(1)$ queries to the untrusted memory. Therefore our construction yields the first parallel memory checker in the public key setting with $O(1)$ query complexity using $O(\log M)$ processors.

Authenticated data structure with logarithmic space at the trusted source. Our construction can be used for implementing an authenticated dictionary with improved space bounds in the three-party model—the traditional model of authenticated data structures [43]. Specifically, we can reduce the space of the trusted source from $O(n)$ to $O(\log n)$. This is because in the three-party model of authenticated data structures, the only goal of the trusted source is to update the publish the latest digest d_h, which, in our construction can be achieved in a streaming fashion (by storing only the previous digest d_{h-1}) and without having access to all the elements of the dictionary (only access to the element of the update is required, see Algorithm $\mathsf{updateVerifier}$). In previous implementations however (e.g., [20,36]), the source keeps all the Merkle tree locally (otherwise the digest cannot be updated), therefore requiring $O(n)$ local space.

Acknowledgments

This work was supported by the Center for Geometric Computing and the Kanellakis fellowship at Brown University, by a NetApp Faculty Fellowship, by Intel through the ISTC for Secure Computing and by the NSF grant CNS-1228485. Part of the work was performed while the first author was at Brown University.

We are grateful to Chris Peikert for his continuous feedback and support during the early stages of this work and to Daniele Micciancio for his insightful feedback on improving the presentation of our result. We particularly thank Bobby Bhattacharjee for suggesting the bandwidth sharing application and the idea of composing multiple streams. We also thank Aris Anagnostopoulos, Hubert Chan, Michael Goodrich, Christopher Imbriano, Jonathan Katz, Behzad Koosha, Dave Levin, Anna Lysyanskaya, Shay Mozes, Franco Preparata, Nikos Triandopoulos and Vinod Vaikuntanathan for useful discussions, and the EUROCRYPT 2013 program committee for their helpful reviews.

References

1. M. Ajtai. Generating hard instances of lattice problems. In *STOC*, pp. 99–108, 1996.
2. B. Applebaum, Y. Ishai, and E. Kushilevitz. From secrecy to soundness: Efficient verification via secure computation. In *ICALP*, pp. 152–163, 2010.
3. M. H. Au, P. P. Tsang, W. Susilo, and Y. Mu. Dynamic universal accumulators for DDH groups and their application to attribute-based anonymous credential systems. In *CT-RSA*, pp. 295–308, 2009.
4. N. Baric and B. Pfitzmann. Collision-free accumulators and fail-stop signature schemes without trees. In *EUROCRYPT*, pp. 480–494, 1997.
5. S. Benabbas, R. Gennaro, and Y. Vahlis. Verifiable delegation of computation over large datasets. In *CRYPTO*, pp. 111–131, 2011.
6. J. C. Benaloh and M. de Mare. One-way accumulators: A decentralized alternative to digital signatures. In *EUROCRYPT*, pp. 274–285, 1993.
7. M. Blum, W. S. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. *Algorithmica*, 12(2/3):225–244, 1994.
8. J. Camenisch, M. Kohlweiss, and C. Soriente. An accumulator based on bilinear maps and efficient revocation for anonymous credentials. In *PKC*, pp. 481–500, 2009.
9. J. Camenisch and A. Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In *CRYPTO*, pp. 61–76, 2002.

10. K.-M. Chung, Y. T. Kalai, F.-H. Liu, and R. Raz. Memory delegation. In *CRYPTO*, pp. 151–168, 2011.
11. G. Cormode, M. Mitzenmacher, and J. Thaler. Practical verified computation with streaming interactive proofs. In *ITCS*, pp. 90–112, 2012.
12. G. Cormode, J. Thaler, and K. Yi. Verifying computations with streaming interactive proofs. *PVLDB*, 5(1):25–36, 2011.
13. P. Devanbu, M. Gertz, A. Kwong, C. Martel, G. Nuckolls, and S. Stubblebine. Flexible authentication of XML documents. *Journal of Computer Security*, 6:841–864, 2004.
14. C. Estan and G. Varghese. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Trans. Comput. Syst.*, 21(3):270–313, 2003.
15. D. Fiore and R. Gennaro. Improved publicly verifiable delegation of large polynomials and matrix computations. Cryptology ePrint Archive, Report 2012/434.
16. D. Fiore and R. Gennaro. Publicly verifiable delegation of large polynomials and matrix computations, with applications. In *CCS*, pp. 501–512, 2012.
17. R. Gennaro, C. Gentry, and B. Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *CRYPTO*, pp. 465–482, 2010.
18. R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *EUROCRYPT*, 2013.
19. O. Goldreich, S. Goldwasser, and S. Halevi. Collision-free hashing from lattice problems. *Electronic Colloquium on Computational Complexity*, 3(42), 1996.
20. M. T. Goodrich, R. Tamassia, and A. Schwerin. Implementation of an authenticated dictionary with skip lists and commutative hashing. In *DISCEX II*, pp. 68–82, 2001.
21. M. T. Goodrich, R. Tamassia, and N. Triandopoulos. Efficient authenticated data structures for graph connectivity and geometric search problems. *Algorithmica*, 60(3):505–552, 2011.
22. E. Hall and C. S. Julta. Parallelizable authentication trees. In *SAC*, pp. 95–109, 2005.
23. A. Kawachi, K. Tanaka, and K. Xagawa. Concurrently secure identification schemes based on the worst-case hardness of lattice problems. In *ASIACRYPT*, pp. 372–389, 2008.
24. A. K. Lenstra, H. W. L. Jr, and L. Lovasz. Factoring polynomials with rational coefficients. *Math.Ann.*, (261):515–534, 1982.
25. F. Li, K. Yi, M. Hadjieleftheriou, and G. Kollios. Proof-infused streams: enabling authentication of sliding window queries on streams. In *VLDB*, pp. 147–158, 2007.
26. J. Li, N. Li, and R. Xue. Universal accumulators with efficient nonmembership proofs. In *ACNS*, pp. 253–269, 2007.
27. C. U. Martel, G. Nuckolls, P. T. Devanbu, M. Gertz, A. Kwong, and S. G. Stubblebine. A general model for authenticated data structures. *Algorithmica*, 39(1):21–41, 2004.
28. R. C. Merkle. A certified digital signature. In *CRYPTO*, pp. 218–238, 1989.
29. D. Micciancio and C. Peikert. Hardness of SIS and LWE with small parameters. Cryptology ePrint Archive, Report 2013/069.
30. D. Micciancio and O. Regev. Worst-case to average-case reductions based on gaussian measures. *SIAM J. Comput.*, 37(1):267–302, 2007.
31. M. Naor and K. Nissim. Certificate revocation and certificate update. In *USENIX Security*, pp. 217–228, 1998.
32. L. Nguyen. Accumulators from bilinear pairings and applications. In *CT-RSA*, pp. 275-292, 2005.
33. S. Papadopoulos, Y. Yang, and D. Papadias. Continuous authentication on relational streams. *VLDB Journal*, 19(2):161–180, 2010.
34. C. Papamanthou, E. Shi, and R. Tamassia. Signatures of correct computation. In *TCC*, pp. 222-242, 2013.
35. C. Papamanthou and R. Tamassia. Time and space efficient algorithms for two-party authenticated data structures. In *ICICS*, pp. 1–15, 2007.

36. C. Papamanthou, R. Tamassia, and N. Triandopoulos. Authenticated hash tables. In *CCS*, pp. 437–448, 2008.
37. C. Papamanthou, R. Tamassia, and N. Triandopoulos. Optimal authenticated data structures with multilinear forms. In *PAIRING*, pp. 246–264, 2010.
38. C. Papamanthou, R. Tamassia, and N. Triandopoulos. Optimal verification of operations on dynamic sets. In *CRYPTO*, pp. 91–110, 2011.
39. V. Pappas, M. Raykova, B. Vo, S. M. Bellovin, and T. Malkin. Private search in the real world. In *ACSAC*, pp. 83–92, 2011.
40. B. Parno, M. Raykova, and V. Vaikuntanathan. How to delegate and verify in public: Verifiable computation from attribute-based encryption. In *TCC*, pp. 422-439, 2012.
41. O. Regev. On the complexity of lattice problems with polynomial approximation factors. *The LLL algorithm*, pp. 475–496, 2010.
42. D. Shroeder and H. Shroeder. Verifiable data streaming. In *CCS*, pp. 953-964, 2012.
43. R. Tamassia. Authenticated data structures. In *ESA*, pp. 2–5, 2003.

Appendix

7.1 Proof of collision resistance (proof of Theorem 2)

Since generalized hash trees (T, λ, f, h) and (T, l, f, h) comprise a collision, it is $\lambda(\epsilon) = l(\epsilon)$ and there exists $v_\ell \in T$ such that $\lambda(v_\ell) \neq l(v_\ell)$—see Definition 11. Consider now the path of nodes $v_\ell, v_{\ell-1}, \ldots, v_1, v_0 = \epsilon$ from node v_ℓ to the root $v_0 = \epsilon$ of T. Let also $w_\ell, w_{\ell-1}, \ldots, w_1$ be the siblings of the nodes $v_\ell, v_{\ell-1}, \ldots, v_1$ respectively. We define the following events: **(1)** $\mathcal{E}_{\ell,0}$: $l(v_\ell) \neq \lambda(v_\ell)$; **(2)** $\mathcal{E}_{i,0}$: $l(v_i) \neq \lambda(v_i)$ for $i = \ell-1, \ldots, 1$; **(3)** $\mathcal{E}_{i,0}$: $l(v_i) = \lambda(v_i)$ for $i = \ell - 1, \ldots, 1$; **(4)**: $\mathcal{E}_{0,1}$: $\lambda(\epsilon) = l(\epsilon)$. The probability that a PPT algorithm can output a collision (T, λ, f, h) and (T, l, f, h) is at most

$$\begin{aligned} &\Pr[\mathcal{E}_{\ell,0} \cap (\mathcal{E}_{\ell-1,0} \cup \mathcal{E}_{\ell-1,1}) \cap (\mathcal{E}_{\ell-2,0} \cup \mathcal{E}_{\ell-2,1}) \cap \ldots \cap \mathcal{E}_{0,1}] \\ \leq\ &\Pr[\mathcal{E}_{\ell,0} \cap \mathcal{E}_{\ell-1,1}] + \ldots + \Pr[\mathcal{E}_{1,0} \cap \mathcal{E}_{0,1}] = \sum_{0 \leq i \leq \ell-1} \Pr[\mathcal{E}_{i+1,0} \cap \mathcal{E}_{i,1}]\,. \end{aligned}$$

Note that the event $\mathcal{E}_{i+1,0} \cap \mathcal{E}_{i,1}$ is equivalent to the set of conditions: **(i)** $l(v_{i+1}) \neq \lambda(v_{i+1})$; **(ii)** $l(v_i) = \lambda(v_i)$ (recall $v_0 = \epsilon$).

It is easy to see that if h is collision resistant, the probability $\Pr[\mathcal{E}_{i+1,0} \cap \mathcal{E}_{i,1}]$ is $\mathsf{neg}(k)$ since it is equivalent with outputting a collision to function h: Since both (T, λ, f, h) and (T, l, f, h) are generalized hash trees it is $f(l(v_i)) = h(l(v_{i+1}), l(w_{i+1}))$ and $f(\lambda(v_i)) = h(\lambda(v_{i+1}), \lambda(w_{i+1}))$. Since now $\lambda(v_i) = l(v_i)$ we have $f(\lambda(v_i)) = f(l(v_i))$. But $l(v_{i+1}) \neq \lambda(v_{i+1})$ and therefore $(\lambda(v_{i+1}), \lambda(w_{i+1}))$ is a collision with the pair $(l(v_{i+1}), l(w_{i+1}))$. Therefore the probability $\Pr[\mathcal{E}_{i,0}|\mathcal{E}_{i-1,1}]$ is $\mathsf{neg}(k)$, implying that the sum $\sum_{i=0}^{\ell-1} \Pr[\mathcal{E}_{i,0}|\mathcal{E}_{i-1,1}]$ is also $\mathsf{neg}(k)$, as T has polynomial depth and ℓ is no greater than T's depth. □

7.2 Proof of security (stated in Theorem 4)

Fix the security parameter k and output $\mathsf{pk} = (\mathbf{L}, \mathbf{R}, q, \mathcal{U})$ by calling algorithm genkey. Let Adv be a PPT adversary. Let D_0 an initial structured binary tree T_C where $c_i = 0$ for $i = 0, \ldots, M-1$ and let d_0 be the state output by $\mathsf{initialize}(D_0, \mathsf{pk})$.

1. *Update.* For $t = 1, \ldots, h \leq n$, the adversary Adv picks an element $x_t \in \mathcal{U}$. Let d_t be the final state output by calling updateVerifier for every element x_t and let

$T_{\mathcal{C}}$ be the final structured binary tree D_t after all the updates have been performed, where $\mathcal{C} = [c_0, c_1, \ldots, c_{M-1}]$.

2. *Forge.* Let $x \in \mathcal{U}$ be a query element and $\alpha \neq c_x$ be an *incorrect* value for index x picked by Adv (as in Definition 3). The adversary Adv outputs

$$\Pi(x) = \{(l(v_\ell), l(w_\ell)), (l(v_{\ell-1}), l(w_{\ell-1})), \ldots, (l(v_1), l(w_1))\}$$

as the proof for element x, where v_ℓ is the node corresponding to index x.

We prove that the probability that $1 \leftarrow \mathsf{verify}(x, \alpha, \Pi(x), d_h, \mathsf{pk})$ while $\alpha \neq c_x$ is negligible. To do that we consider the full binary tree T (see Definition 8) defined by the nodes $(v_\ell, w_\ell), (v_{\ell-1}, w_{\ell-1}), \ldots, (v_1, w_1)$ and the root node ϵ. It is easy to see that since the verification algorithm accepts, (T, l, f, h_n) is a generalized hash tree as defined in Definition 10, and where l is the labeling in $\Pi(x)$.

Consider now the structured binary tree $T_{\mathcal{C}}$ (where $\mathcal{C} = [c_0, c_1, \ldots, c_{M-1}]$) as defined in Definition 12. Let $T'_{\mathcal{C}}$ be the subtree of $T_{\mathcal{C}}$ that has the same nodes as T. By Theorem 3, the adversary can compute (T, λ, f, h_n), which is also a generalized hash tree. However, since $\alpha \neq c_x$ this means that $l(v_\ell) \neq \lambda(v_\ell)$. Note now that $\lambda(\epsilon) = l(\epsilon) = d_h$ and therefore the adversary has output a tree collision, which, by Theorem 2, happens with probability $\mathsf{neg}(k)$ since h_n is collision resistant (see Theorem 1). A same argument applies for the range search query. This completes the proof. □

7.3 Proof of asymptotic performance (stated in Theorem 4)

Algorithm $\mathsf{updateVerifier}$ requires computing the partial label $\mathcal{L}_\epsilon(x)$, where x is the element of the update and ϵ is the root of $T_{\mathcal{C}}$. Computing $\mathcal{L}_\epsilon(x)$ can be achieved in $O(\log M \log^2 n)$ time, by Lemma 3.

Algorithm $\mathsf{updateProver}$ needs to compute the partial labels $\mathcal{L}_{v_j}(x)$ $(j = \ell, \ldots, 0)$, where $v_\ell, v_{\ell-1}, \ldots, v_0$ are the nodes of the structured binary tree from element x to the root of the tree. By Definition 15, all these labels can be computed in $O(\log M \log^2 n)$ time during the computation of $\mathcal{L}_\epsilon(x)$ (i.e., in one pass). However, to update a label $\lambda(v_i)$, one needs to retrieve it from the underlying data structure that stores the "useful" portion of the generalized hash tree (and either store it back or delete if the label becomes $\mathbf{0}$). Therefore $\mathsf{updateProver}$ needs to spend an extra $O(\log \nu)$ time in the worst case, where ν is the number of the currently stored elements. Since however $\nu \leq n$, it follows that the time required is $O(\log M \log^2 n)$.

Algorithm query for membership and successor queries needs to retrieve $O(\log M)$ binary representations of $O(\log n)$ bits each, spending $O(\log \nu)$ time to retrieve each one of them. Since $\nu \leq n$, it follows that query runs in $O(\log M \log n)$ time. The proof has also size $O(\log M \log n)$, since it contains $O(\log M)$ binary representations of $O(\log n)$ bits each. Since range search is implemented via s successor queries, the same bounds apply multiplied with s, where s is the size of the output range.

Finally, for the space at the client, it is required that the client store d_h, which consists of k $O(\log n)$-bit numbers, therefore the space at the client is $O(\log n)$. For the space at the prover, we recall that we only store labels $\lambda(v)$ that lie on tree paths starting from leaves x such that $c_x > 0$ (all these labels are also non-zero and have $O(\log n)$ bits). Since at every point in time there are ν elements stored in the data structure, it follows that the space at the server is $O(\nu \log M \log n)$.