Update-Optimal Authenticated Structures Based on Lattices

Charalampos Papamanthou* Roberto Tamassia†

March 7, 2010

Abstract

We study the problem of authenticating an n-index *dynamic table* in the authenticated data structures model, which is related to memory checking. We present the first *lattice-based* authenticated structure for this problem, which is *update-optimal*. In specific, the update time is $O(1)$, improving in this way the "a priori" logarithmic limit (in n) of Merkle tree constructions. Moreover, the space is maintained to be $O(n)$, while other logarithmic bounds for other complexities (e.g., proof size) are still in place. To achieve this result, we exploit the "linearity" of lattice-based hash functions and show how necessary properties —for security—of lattice-based digests can be guaranteed under updates. This is the first construction achieving constant update bounds without causing other time complexities to increase beyond logarithmic. All previous solutions enjoying such an update complexity have (sub)linear proof or query bounds. As an application of our lattice-based authenticated structure, we provide the first construction of an authenticated Bloom filter, an update-intensive data structure that falls into our model.

Keywords: Authenticated data structures, lattice-based cryptography.

1 Introduction

Increasing interest in online data storage and processing has recently led to the establishment of the field of *cloud computing* [20]. Files can be outsourced to service providers that offer huge capacity and fast network connections (e.g., Amazon S3) as a means of mitigating maintenance and storage costs. In this way, clients create virtual hard drives consisting of online storage units that are operated by remote and geographically dispersed servers. In such settings, the ability to check the integrity of remotely stored data is an important security property, or otherwise a faulty or malicious server can lose or tamper with the client's data (e.g., deleting or modifying a file). In order to solve the problem of efficiently checking the integrity of outsourced data, the model of *authenticated data structures* (see, e.g., [24, 36]) has been developed, which is closely related to memory checking [7]. In an authenticated data structure, untrusted servers answer queries on a data structure on behalf of a trusted source and provide a proof of validity of each answer to the user.

In specific, in an authenticated data structure, there are three participating entities. The owner of the data, called *source*, outsources its data to multiple untrusted sites, called *servers*. The *clients*, due to scalability issues can only send queries to the *servers* and wish to verify answers received by the servers, based only on the trust they have to the source. This trust from the source to the clients is usually conveyed through a time-stamped signature on the data structure *digest*, a collision resistant succinct representation of the data structure (e.g., the root hash of a Merkle tree).

In the study of authenticated data structures, apart from achieving provable security under a well-accepted assumption (e.g., strong RSA assumption), it is important to achieve small asymptotic bounds for relevant complexity measures, which are listed in the first column of Table 1 (also explained in more detail at the end of Section 2). Therefore, there is typically a challenging trade-off between security and

*Department of Computer Science, Brown University. Email: `cpap@cs.brown.edu`.
†Department of Computer Science, Brown University. Email: `rt@cs.brown.edu`.

Table 1 Asymptotic time and space complexity of previous solutions and of our work for the problem of authenticating a dynamic table of size n. We denote with k the security parameter. Parameter $0 < \epsilon < 1$ is a constant that can be arbitrarily chosen. Also, "D. Log" stands for "Discrete Logarithm", "Generic CR" stands for "Generic Collision Resistance", GAPSVP_γ is the gap version of the shortest vector problem in lattices (see Definition 1), where $\gamma = O(nk\sqrt{k})$.

	[7, 27]	**[4]**	**[29]**	**[10, 35]**	**[18]**	**[30]**	**this work**
source update	$\log n$	1	1	1	n^ϵ	1	1
server update	$\log n$	1	n	$n \log n$	n^ϵ	1	$\log^2 n$
server query	$\log n$	n	1	1	n^ϵ	n^ϵ	$\log n$
verification	$\log n$	n	1	1	1	1	$\log n$
source space	n	n	n	n	n	n	n
server space	n	n	n	n	n	n	n
proof size	$\log n$	n	1	1	1	1	$\log n$
update info.	1	1	1	1	n^ϵ	1	1
assumption	Generic CR	D. Log	Strong DH	Strong RSA			GAPSVP_γ is hard

efficiency. In this work we show that the cryptographic primitive used can have a significant impact on the efficiency of the structure. Towards this goal, we employ *lattices*, a mathematical tool that was shown to have many applications in cryptography after Ajtai's seminal result [1] and we provide the first constant complexity bounds for the update time of a lattice-based authenticated structure.

The motivation for this work stems from the absence in the literature of an authenticated structure where an efficient update (e.g., in $O(1)$ or $O(\log \log n)$ time) does not cause other complexity measures to "blow up" to sublinear or linear. For example, although updates are optimally performed in $O(1)$ time in [4], the size of the proof implied with such an authenticated structure is $O(n)$, i.e., to prove an element that has been accumulated to an *optimally* updatable digest, all the elements have to be communicated. Similarly, in [30], while more optimal bounds due to the use of accumulators are achieved, there is a sublinear complexity $O(n^\epsilon)$ for query or update time, a trade-off that was also observed in [14]. Therefore, if one wishes to avoid (sub)linear complexities, one has to resort to the extensively used, both in theory and practice, Merkle tree [25], or various alternations of it [7, 19]. However, all solutions based on Merkle trees and use "generic collision resistance[1]" (see Table 1) as a hardness assumption, inherently enforce logarithmic complexities on all the complexity measures. In this paper we combine the merits of a Merkle tree (a binary tree is used in our construction) and the convenient "linearity" of lattice-based hash functions [17] towards constructing a constant-update authenticated structure, while keeping other complexity measures logarithmic.

In this work, we use a model similar to that of memory checking [7]. The structure we wish to authenticate is a *dynamic* table of size n, accessed through indices $1, \ldots, n$. The table is dynamic and each index can take one of C different values, e.g., for $C = 2$ we have a *boolean* table. The value C is not dependent on n, i.e., $C = O(1)$ (it can also be $\mathsf{poly}(k)$, where k is the security parameter).

Related work. Lattice-based cryptography began with Ajtai's first construction of an one-way hash function based on hard lattice problems [1]. This function was shown to be collision resistant by Goldreich [17] and further generalizations of it were given by Micciancio [26]. Other hash functions based on lattices with reduced public key size are due to Micciancio [23] and Peikert [31]. Recently, *trapdoor* functions based on lattices were introduced in [16].

In the field of data integrity checking, several authenticated data structures based on cryptographic hashing have been developed, beginning with the well-known Merkle trees [7, 25, 27] and modifications of

[1] We call *generic collision resistant functions* (Generic CR in Table 1) those functions that are believed to be collision resistant in practice (e.g., SHA-256).

it [19]. Lower bounds for hashing-based methods in the authenticated data structures model are shown in [37], and in the context of memory checking in [14, 28]. Authenticated data structures using other cryptographic primitives, such as *one-way accumulators* [2, 5, 10] are presented in [18], achieving $O(n^{\epsilon})$ bounds. Bilinear pairings accumulators, the security of which is based on the strong Diffie-Hellman assumption, are introduced in [29]. Authenticated hash tables proved secure under the strong RSA assumption or the strong Diffie-Hellman assumption are introduced in [30], where, however, the update or query time is sublinear.

We observe that all of the above constructions belong to one of the following two categories: either (a) they have logarithmic source update complexity, with all the other complexity measures being also logarithmic, e.g., [7, 19, 27]; or (b) they have sublogarithmic source update complexity (e.g., constant) but at least one of the other complexities is (sub)linear, e.g., [4, 29, 30]. A summary and comparison of our work with previous constructions in the literature can be found in Table 1. We note that, to our knowledge, this is the first construction that enjoys a constant update time complexity, without an increase in other complexity measures. We are able to achieve these bounds by exploiting the "linearity" of lattice-based hash functions, which other primitives such as *generic collision resistant functions* (used in [7, 19, 27]) and *exponentiation functions* (used in [4, 29, 30]) lack.

Contributions. Our main contribution is the construction of an update-optimal authenticated data structure for an n-index table based on lattices. The update time of the authenticated structure is $O(1)$ per update and the space complexity is $O(n)$. Our authenticated structure is update-optimal: The update complexity is constant, i.e., not dependent on n, while logarithmic costs for other complexity measures are still in place. This is the first lattice-based authenticated structure and the first one to achieve constant update bounds. All previous solutions enjoying such an update complexity have (sub)linear proof or query bounds. As an application of our lattice-based structure, we provide the first construction of an authenticated Bloom filter.

2 Preliminaries

We start with some preliminary notions that are important in our main construction. In the following, we use k to denote the security parameter (we do not use n as is usually done in lattice-related bibliography) and n to denote the size of the table to be authenticated. We use upper case bold letters to denote matrices, e.g., $\mathbf{B}$, lower case bold letters to denote vectors, e.g., $\mathbf{b}$ and lower case italic letters to denote scalars. Finally, for a vector $\mathbf{x} = [\mathbf{x}_1\ \mathbf{x}_2\ \dots\ \mathbf{x}_k]^T$, $\|\mathbf{x}\|$ denotes the Euclidean norm of $\mathbf{x}$.

Lattices. Given the security parameter k, a full-rank k-dimensional lattice is the infinite-sized set of all vectors produced as the integer combinations $\{\sum_{i=1}^{k} x_i \mathbf{b}_i\ :\ x_i \in \mathbb{Z},\ 1 \leq i \leq k\}$, where $\mathbf{B} = \{\mathbf{b}_1, \mathbf{b}_2, \dots, \mathbf{b}_k\}$ is the *basis* of the lattice and $\mathbf{b}_1, \mathbf{b}_2, \dots, \mathbf{b}_k$ are linearly independent, all belonging to $\mathbb{R}^k$. We denote the lattice produced by $\mathbf{B}$ (i.e., the set of vectors) with $L(\mathbf{B})$.

A well-known difficult problem in lattices is the approximation within a polynomial factor of the *shortest* vector in a lattice (SVP problem). Namely, given a lattice $L(\mathbf{B})$ produced by a basis $\mathbf{B}$, approximate up to a polynomial factor in k the shortest (in an Euclidean sense) vector in $L(\mathbf{B})$, the length of which we denote with $\lambda(\mathbf{B})$. A similar problem in lattices is the "gap" version of the shortest vector problem (GAPSVP_γ), the difficulty of which is going to be useful in our context.

Definition 1 (Problem GAPSVP_γ) *Let k be the security parameter. An input to GAPSVP_γ is a k-dimensional lattice basis $\mathbf{B}$ and a number d. In YES inputs $\lambda(\mathbf{B}) \leq d$ and in NO inputs $\lambda(\mathbf{B}) > \gamma \times d$, where $\gamma \geq 1$.*

Concerning the complexity of the above problem, we note that, for exponential values of γ, i.e, $\gamma = 2^{O(k)}$, one can use the LLL algorithm [22] and produce a solution in polynomial time. Therefore the difficult version of the problem arises for polynomial γ, for which no efficient algorithm is known to date, even for factors slightly smaller than exponential [32], i.e., very big polynomials. Moreover, for polynomial factors, there is no proof that this problem is NP-hard[2], which makes the polynomial approximation cryptographically interesting as well.

Reductions. After Ajtai's seminal work [1] where an one-way function based on hard lattices problem is presented, Goldreich et al. [17] presented a variation of the function, providing at the same time collision resistance. Based on this collision resistant hash function, Micciancio [26] described a generalized version of it, a modification of which we are using in our construction. The security of the hash function is based on the difficulty of the *small integer solution* problem (SIS):

Definition 2 (Problem $\mathsf{SIS}_{p,m,\beta}$) *Let k be the security parameter. Given an integer p, a matrix $\mathbf{F} \in \mathbb{Z}_p^{k\times m}$ and a real β, find a non-zero integer vector $\mathbf{z} \in \mathbb{Z}^m \backslash \{\mathbf{0}\}$ such that $\mathbf{Fz} = \mathbf{0} \mod p$ and $\|\mathbf{z}\| \leq \beta$.*

Note that at least one solution to the above problem exists when $\beta \geq \sqrt{m}p^{k/m}$ and $m > k$ [26]. Moreover, if $p \geq 4\sqrt{m}k^{1.5}\beta$, we will see that such a solution is difficult to find. We continue with the definition of SIS', where the solution vector is required to have at least one odd coordinate:

Definition 3 (Problem $\mathsf{SIS}'_{p,m,\beta}$) *Let k be the security parameter. Given an integer p, a matrix $\mathbf{F} \in \mathbb{Z}_p^{k\times m}$ and a real β, find an integer vector $\mathbf{z} \in \mathbb{Z}^m \backslash 2\mathbb{Z}^m$ such that $\mathbf{Fz} = \mathbf{0} \mod p$ and $\|\mathbf{z}\| \leq \beta$.*

Micciancio [26] showed that if p is odd, there is a polynomial time reduction from $\mathsf{SIS}'_{p,m,\beta}$ to $\mathsf{SIS}_{p,m,\beta}$:

Lemma 1 (Reduction from $\mathsf{SIS}'_{p,m,\beta}$ to $\mathsf{SIS}_{p,m,\beta}$ [26]) *For any odd integer $p \in 2\mathbb{Z}+1$, and SIS' instance $I = (p, \mathbf{F}, \beta)$, if I has a solution as an instance of SIS, then it also has a solution as an instance of SIS'. Moreover, there is a polynomial time algorithm that on input a solution to a SIS instance I, outputs a solution to the same SIS' instance I.*

As proved by Micciancio [26], under a certain choice of parameters, GAPSVP_γ can be reduced to SIS' (this can be derived as a combination of Lemma 5.22 and Theorem 5.23 of [26]):

Lemma 2 (Reduction from GAPSVP_γ to $\mathsf{SIS}'_{p,m,\beta}$ [26]) *For any polynomially bounded $\beta, m, p = k^{O(1)}$, with $p \geq 4\sqrt{m}k^{1.5}\beta$ and $\gamma = 14\pi\sqrt{k}\beta$, there is a probabilistic polynomial time reduction from solving GAPSVP_γ in the worst case to solving $\mathsf{SIS}'_{p,m,\beta}$ on the average with non-negligible probability.*

A direct application of Lemma 1 and Lemma 2 gives the following result.

Theorem 1 *Let $p = k^{O(1)}$ be an odd positive integer. For any polynomially bounded $\beta, m = k^{O(1)}$, with $p \geq 4\sqrt{m}k^{1.5}\beta$ and $\gamma = 14\pi\sqrt{k}\beta$, there is a probabilistic polynomial time reduction from solving GAPSVP_γ in the worst case to solving $\mathsf{SIS}_{p,m,\beta}$ on the average with non-negligible probability.*

In other words, Theorem 1 states that if there is an algorithm that solves an average instance of $\mathsf{SIS}_{p,m,\beta}$ (an average instance refers to the fact that the matrix $\mathbf{M} \in \mathbb{Z}_p^{k\times m}$ is chosen uniformly at random), for an odd p, $p \geq 4\sqrt{m}k^{1.5}\beta$ and $\gamma = 14\pi\sqrt{k}\beta$, then, this algorithm can be used to produce a solution to any (the worst) instance of GAPSVP_γ.

Lattice-based hash function. Let $m = 2k^2$ and $\beta = \delta\sqrt{m}$ where δ is $\mathsf{poly}(k)$ and p be a polynomially bounded odd integer such that $p \geq 4\sqrt{m}k^{1.5}\beta$. It is easy to see that given k and δ there is always a $p = O(k^{3.5}\delta)$ to satisfy the above constraints. The collision resistant hash function that we are using is a generalization of the function presented in [26], where $\delta = O(1)$ (in the security parameter) is used instead. In our construction we use bigger values for δ. Namely the value that we use to bound the norm of the vector can be up to $\mathsf{poly}(k)$. This was observed in the original definition of Ajtai's one-way function [1], i.e., that the input vector can contain larger values (but not so large), and was also noted in its extension that achieves collision resistance [17]. This remark is very useful in our context and implies that, the larger value one picks for β, the larger the modulus p should be so that security is guaranteed.

[2]In specific, as outlined in [32], the current state of knowledge indicates that for factors beyond $\sqrt{k/\log k}$, it is unlikely that this problem is NP-hard and no efficient algorithm is known to date.

Our hash function construction, however, uses a different modulus q (not p) that has k bits instead (note that that p has $O(\log k \log \delta)$ bits): Let q be a k-bit modulus that is divided by p, i.e., $q = \Theta(2^k)$ and $p|q$. Let also λ be a value satisfying

$$\lambda = \frac{q}{p} = \Theta\left(\frac{2^k}{k^{3.5}\delta}\right) . \tag{1}$$

We sample a matrix $\mathbf{F} \in \mathbb{Z}_p^{k\times m}$ uniformly at random. After that we compute the matrix $\mathbf{M} = \lambda\mathbf{F}$. Note that the elements of matrix $\mathbf{M}$ have entries in $\mathbb{Z}_q$. Also note that λ defines an *injective homomorphism* from $\mathbb{Z}_p$ to $\mathbb{Z}_q$. We can now define the function $h_{\mathbf{M}} : \mathbb{Z}^m \rightarrow \mathbb{Z}_q^k$ as $h_{\mathbf{M}}(\mathbf{x}) = \mathbf{M}\mathbf{x} \mod q$, where $\|\mathbf{x}\| \leq \beta$ and the modulo operation is taken component-wise. The above function can be proved to be collision resistant (with some constraint on the input's coordinates) based on the difficulty of the problem $\mathsf{GAPSVP}_{14\pi\sqrt{k}\beta}$:

Theorem 2 (Strong collision resistance) *Let $m = 2k^2$, $\beta = \delta\sqrt{m}$ and $p \geq 4\sqrt{m}k^{1.5}\beta$ be an odd positive integer. Let also $\mathbf{F} \in \mathbb{Z}_p^{k\times m}$ be a $k \times m$ matrix that is chosen uniformly at random and $\mathbf{M} = \lambda\mathbf{F} \in \mathbb{Z}_q^{k\times m}$ where q and λ are defined in Equation 1. If there is an algorithm that finds two vectors $\mathbf{x}, \mathbf{y} \in \{0, 1, \ldots, \delta\}^m$ and $\mathbf{x} \neq \mathbf{y}$ such that $\mathbf{M}\mathbf{x} = \mathbf{M}\mathbf{y} \mod q$, then there is an algorithm to solve any instance of the $\mathsf{GAPSVP}_{14\pi\delta\sqrt{km}}$ problem.*

Proof: Suppose there is an algorithm that finds $\mathbf{x}, \mathbf{y} \in \{0, 1, \ldots, \delta\}^m$ with $\mathbf{x} \neq \mathbf{y}$ such that $\mathbf{M}\mathbf{x} = \mathbf{M}\mathbf{y} \mod q \Rightarrow \mathbf{M}(\mathbf{x} - \mathbf{y}) = \mathbf{0} \mod q$. This, dy the definition of q and $\mathbf{M}$ can be written as

$$\lambda\mathbf{F}(\mathbf{x} - \mathbf{y}) = \mathbf{0} \mod \lambda p \Rightarrow \exists \mathbf{r} \in \mathbb{Z}^k : \lambda\mathbf{F}(\mathbf{x} - \mathbf{y}) = \mathbf{r}\lambda p \Rightarrow \mathbf{F}(\mathbf{x} - \mathbf{y}) = \mathbf{r}p \Rightarrow \mathbf{F}(\mathbf{x} - \mathbf{y}) = \mathbf{0} \mod p .$$

Therefore the non-zero vector $\mathbf{z} = \mathbf{x} - \mathbf{y}$, which also has norm $\|\mathbf{z}\| \leq \beta$, since its coordinates are between $-\delta$ and $+\delta$, comprises a solution to the problem $\mathsf{SIS}_{p,m,\beta}$ (note that matrix $\mathbf{F}$ by construction is chosen uniformly at random). By Theorem 1, this can be used to solve GAPSVP_γ for $\gamma = 14\pi\sqrt{k}\beta$. Setting $\beta = \delta\sqrt{m}$ we get the desired result. $\square$

Therefore we have just presented a collision-resistant hash function the security of which is based on the difficulty of the problem GAPSVP_γ. Note that as long as $\delta = \mathsf{poly}(k)$, γ is also $\mathsf{poly}(k)$ and therefore the hash function is secure since for polynomial γ (even for γ slightly smaller than exponential), no efficient algorithm to solve GAPSVP_γ is known to date [32].

We can now extend the function h to accept two inputs as follows: Denote with $\mathbb{T}^{\delta,+}$ the set of all $m \times 1$ ($m = 2k^2$) vectors such that their last k^2 entries are zero and the remaining entries are in $\{0, 1, \ldots, \delta\}$ and analogously with $\mathbb{T}^{\delta,-}$ the set of all $m \times 1$ vectors such that their first k^2 entries are zero and the remaining entries are in $\{0, 1, \ldots, \delta\}$. For a $k \times m$ matrix $\mathbf{M}$ sampled uniformly at random we can define the function $h : \mathbb{T}^{\delta,+} \times \mathbb{T}^{\delta,-} \rightarrow \mathbb{Z}_q^k$

$$h_{\mathbf{M}}(\mathbf{x}, \mathbf{y}) = \mathbf{M}(\mathbf{x} + \mathbf{y}) \mod q , \tag{2}$$

where $\mathbf{x}, \mathbf{y} \in \{0, 1, \ldots, \delta\}^m$. Similarly as in Theorem 2, this function is strong collision resistant, i.e., if someone can find $(\mathbf{x}_1, \mathbf{y}_1) \in (\mathbb{T}^{\delta,+} \times \mathbb{T}^{\delta,-})$ and $(\mathbf{x}_2, \mathbf{y}_2) \in (\mathbb{T}^{\delta,+} \times \mathbb{T}^{\delta,-})$ with $(\mathbf{x}_1, \mathbf{y}_1) \neq (\mathbf{x}_2, \mathbf{y}_2)$ such that $\mathbf{M}(\mathbf{x}_1 + \mathbf{y}_1) = \mathbf{M}(\mathbf{x}_2 + \mathbf{y}_2) \mod q$ then one can solve the problem GAPSVP_γ for polynomial γ. To see that, note that the vector $\mathbf{x}_1 - \mathbf{x}_2 + \mathbf{y}_1 - \mathbf{y}_2$ has coordinates in $\{0, 1, \ldots, \delta\}$, since, by the definition of $\mathbb{T}^{\delta,+}$ and $\mathbb{T}^{\delta,-}$, the entries of $\mathbf{x}_1 - \mathbf{x}_2$ and $\mathbf{y}_1 - \mathbf{y}_2$ do not overlap.

Authenticated data structures. As we mentioned in the introduction, there are three entities participating in the authenticated data structures computational model [24, 36]. A trusted *source* that owns, updates and outsources his data structure D_i, along with a signed, timestamped, collision resistant digest of it, d_i, to the untrusted *servers* that respond to queries sent by the *clients*. The *servers* should be able to provide with proofs to the queries and the *clients* should be able to verify these proofs based on their trust to the source, by basically using the correct and signed digest d_i. Complexities relevant to the source are the *source update time* (time taken for the source to compute the updated digest), *source space* and *update information* (size of information sent to the servers per update, i.e., the signed digest). Relevant to the servers are *server update time* (time taken by the server per update), *server space*, *query time* (time taken by the server to compute a proof for a query) and *proof size*. Finally, relevant to the client are *verification time* and *client space* with obvious meaning. The client verification is performed using an algorithm $\{\mathsf{accept}, \mathsf{reject}\} \leftarrow \mathsf{verify}(q, \Pi(q), d_i)$, where q is a query on data structure D_i and $\Pi(q)$ is a proof provided by the server. Note that the digest d_i, the digest of D_i, is an input as well. All these complexity measures are listed in Table 1.

Let now $\{\mathsf{reject}, \mathsf{accept}\} \leftarrow \mathsf{check}(q, \alpha(q), D_i)$ be an algorithm that, given a query q on data structure D_i and an answer $\alpha(q)$ checks to see if this is the correct answer to query q. We can now present the formal security definition, which states that it should be difficult (except with negligible probability) for a computationally bounded adversary to produce verifying proofs for incorrect answers, even after he brings the data structure to a state of his liking:

Definition 4 (Security) *Suppose k is the security parameter and $\mathcal{A}$ is a computationally bounded adversary that is given the public key of the source* pk. *Our data structure D_0 is in the initial state with digest d_0. The adversary $\mathcal{A}$ issues an update* upd_i *in the data structure D_i for $i = 0, \ldots, h = \mathsf{poly}(k)$ and therefore computes D_{i+1} and d_{i+1}. Suppose now the source is given d_h and issues a number of updates (say u updates), bringing the data structure to the state $D_{h'}$, where $h' = h + u + 1$. Then the adversary enters the attack stage where he chooses a query q and computes an answer $\alpha(q)$ and a verification proof $\Pi(q)$. We say that the authenticated data structure is secure if*

$$\Pr\left[\begin{array}{ll} \{q, \Pi(q), \alpha(q)\} \leftarrow \mathcal{A}(1^k, \mathsf{pk}); & \mathsf{accept} \leftarrow \mathsf{verify}(q, \Pi(q), d_{h'}); \\ & \mathsf{reject} \leftarrow \mathsf{check}(q, \alpha(q), D_{h'}); \end{array} \middle| \; \mathsf{digest}(D_{h'}) = d_{h'} \right] \leq \nu(k)\,,$$

where $\nu(k)$ is negligible in the security parameter k.

3 Main construction

Suppose we are given a table that consists of n indices $1, 2, \ldots, n$. In each index i we can store a value $\mathbf{x}_i$ from the set $S = \{0, 1, \ldots, C\}$ where $|S| = O(1)$. In this section we describe the application of the lattice-based hash function on top of a structure of n indices, in the model described in the introduction.

Without loss of generality assume that n is a power of two so that we can build a complete binary tree on top of the table. Let T be that tree and let $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$ be the original values of the table. Assume each of the elements in $\{0, 1, \ldots, C\}$ can be represented with a vector of size k that has entries in $\mathbb{Z}_q$. Namely $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n \in \mathbb{Z}_q^k$. We are going to use the hash function $h_{\mathbf{M}}(\mathbf{x}, \mathbf{y})$ defined in Equation 2 in a recursive way to define the digest of the structure. We recall that k is the security parameter, $\mathbf{M}$ is a $k \times m$ matrix with elements sampled uniformly at random from $\mathbb{Z}_p$ and then multiplied with λ, $m = 2k^2$, $\beta = \delta\sqrt{m}$, $p \geq 4\sqrt{m}k^{1.5}\beta$ and $q = \lambda p$. We now prove some useful properties:

Definition 5 (Radix-2 representation) *Let $x \in \mathbb{Z}_q$. We define $f(x) \in \mathbb{Z}_q^k$ to be* some *radix-2 representation of x. Namely if $f(x) = [\mathbf{f}_0\ \mathbf{f}_1\ \ldots\ \mathbf{f}_{k-1}]^T$ then it holds $x = \sum_{i=0}^{k-1} \mathbf{f}_i 2^i \mod q$.*

By "some" radix-2 representation we mean that the function $f : \mathbb{Z}_q \rightarrow \mathbb{Z}_q^k$ is "one-to-many". For example, for $q = 16$, $x = 7$, possible values for $f(x)$ can be $[1\ 1\ 1\ 0]^T$ (the usual binary representation), $[-1\ 0\ 2\ 0]^T$ or $[-5\ \ -2\ 0\ 4]^T$ (and many more). When the representation $f(.)$ is the binary representation, we will explicitly denote it with $f_{\mathsf{bin}}(.)$. We can now prove the following:

Lemma 3 *For any $x, y \in \mathbb{Z}_q$ there exist radix-2 representations $f(.)$ such that $f(x + y \mod q) = f(x) + f(y) \mod q$.*

Proof: Let $\mathbf{x} = f(x)$ be a radix-2 representation of x and $\mathbf{y} = f(y)$ be a radix-2 representation of y. Then $\mathbf{x} + \mathbf{y} = [\mathbf{x}_0 + \mathbf{y}_0\ \mathbf{x}_1 + \mathbf{y}_1 \ldots \mathbf{x}_{k-1} + \mathbf{y}_{k-1}]^T \mod q$. The resulting vector is a radix-2 representation of $(\mathbf{x}_0 + \mathbf{y}_0) \times 2^0 + (\mathbf{x}_1 + \mathbf{y}_1) \times 2^1 + \ldots + (\mathbf{x}_{k-1} + \mathbf{y}_{k-1}) \times 2^{k-1} \mod q$ which can be written as

$\mathbf{x}_0 \times 2^0 + \mathbf{x}_1 \times 2^1 + \ldots + \mathbf{x}_{k-1} \times 2^{k-1} + \mathbf{y}_0 \times 2^0 + \mathbf{y}_1 \times 2^1 + \ldots + \mathbf{y}_{k-1} \times 2^{k-1} \mod q = x + y \mod q$. Therefore $f(x) + f(y) = \mathbf{x} + \mathbf{y} = f(x + y \mod q) \mod q$. $\square$

Note that Lemma 3 is useful in the following sense. It tells us that if one has an $f(.)$ representation of $x + y \mod q$ (i.e., f_1) and picks an $f(.)$ representation of $x \mod q$ (i.e., f_2), one can always find an $f(.)$ representation for $y \mod q$, namely $f_1 - f_2$. Definition 5 can be naturally extended for vectors:

Definition 6 *Let $\mathbf{x} \in \mathbb{Z}_q^k$. We define $f(\mathbf{x}) \in \mathbb{Z}_q^{k^2}$ to be* some *radix-2 representation of* $\mathbf{x}$. *Namely every $\mathbf{x}_i$, for $i = 1, \ldots, k$, is mapped to the respective k entries $f(\mathbf{x}_i)$ in the resulting vector $f(\mathbf{x})$.*

We can analogously generalize Lemma 3 as follows:

Corollary 1 *For any $\mathbf{x}, \mathbf{y} \in \mathbb{Z}_q^k$ there exist radix-2 representations $f(.)$ such that $f(\mathbf{x} + \mathbf{y} \mod q) = f(\mathbf{x}) + f(\mathbf{y}) \mod q$.*

Finally, let $\mathbf{U} = [\mathbb{I}_{k^2}\ \mathbb{O}_{k^2}]^T$ (**U** stands for "up") and $\mathbf{D} = [\mathbb{O}_{k^2}\ \mathbb{I}_{k^2}]^T$ (**D** stands for "down") be $m \times k^2$ matrices, where $\mathbb{I}_t$ denotes the square unit matrix of dimension t and $\mathbb{O}_t$ denotes the square zero matrix of dimension t. It easy to see that for all $\mathbf{x} \in \{0, 1, \ldots, \delta\}^{k^2}$ it is $\mathbf{U}\mathbf{x} \in \mathbb{T}^{\delta,+}$ and $\mathbf{D}\mathbf{x} \in \mathbb{T}^{\delta,-}$. Namely multiplying matrices **U** and **D** with a vector in $\{0, 1, \ldots, \delta\}^{k^2}$ doubles the dimension of the vector by shifting its entries accordingly and by filling the vacant entries with zeros. This operation will be used to prepare the vectors in the appropriate input format for the hash function.

Digest definition. As we mentioned in the beginning of Section 3, we build a binary tree of ℓ levels on top of our n-index table. For each node v of the tree, we are going to define a collision resistant digest $d(v)$, based on the lattice-based hash function we introduce in Section 2. The digest of the root will serve as the digest of the whole structure.

For every leaf node v_i of the tree, $i = 1, \ldots, n$ (note that at node v_i we store the value $\mathbf{x}_i$) we define the *leaf digest* $d(v_i)$ simply as $d(v_i) = \mathbf{x}_i \mod q$. For an internal node u, with left child $\mathsf{left}(u)$ and right child $\mathsf{right}(u)$, we define the *internal digest* as

$$d(u) = \mathbf{M}\left[\mathbf{U}f(d(\mathsf{left}(u))) + \mathbf{D}f(d(\mathsf{right}(u)))\right] \mod q\,, \tag{3}$$

where, by the constraint of the inputs in the definition of the hash function in Equation 2, it must be

$$f(d(\mathsf{left}(u))), f(d(\mathsf{right}(u))) \in \{0, 1, \ldots, \delta\}^{m/2}\,. \tag{4}$$

We note here that when the table is built for the first time (initial state), the radix-2 representation $f(\mathbf{z})$ used is the k^2-bit *binary* representation of $\mathbf{z}$, i.e., the entries of $f(\mathbf{z})$ are in $\{0, 1\}$. Note that the binary representation satisfies the constraints of Equation 4 since $\{0, 1\} \subset \{0, 1, \ldots, \delta\}$. Part of the challenging job would be to ensure that whenever we perform an update the constraints of Equation 4 are in place.

The flow of the computation in Equation 3 is as follows (see Figure 1): Given an internal node u, with children $\mathsf{left}(u)$ and $\mathsf{right}(u)$, digests $d(\mathsf{left}(u)), d(\mathsf{right}(u)) \in \mathbb{Z}_q^k$. By applying $f(.)$ we transform them into vectors of k^2 "small" entries. By multiplying with **U** and **D** we "prepare" them to be input to the hash function, as defined in Equation 2. The *lattice digest* of a table $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n \in \mathbb{Z}_q^k$ is defined as follows:

Definition 7 *Let $n = 2^\ell$, $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n \in \mathbb{Z}_q^k$ be the values of the table that is to be authenticated and T be the complete binary tree of root r and height ℓ that is built on top of the table. Suppose we compute the digests $d(u)$ of the nodes u of the tree as above (Equation 3). We define the* lattice digest *of a node u to be the value $d(u)$ and the* lattice digest *of the table to be the value $d(r)$.*

We now present the main result of this section, namely the fact that the lattice digest can be expressed as a sum of n terms, which will eventually allow for more efficient updates. Let $\mathsf{bin}(x)$ denote the binary representation of $x - 1$ and $\mathsf{bin}(x)_i$ the i-th bit of $\mathsf{bin}(x)$:

Theorem 3 *Let $n = 2^\ell$, $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n \in \mathbb{Z}_q^k$ be the values of the table that is to be authenticated and T be the complete binary tree of ℓ levels that is built on top of the table. Let z be an internal node of T at level $0 \le t < \ell$, $T(z)$ be the subtree rooted at z and* $\mathsf{range}(z)$ *be the range of successive indices contained in the leaves of $T(z)$. Then the* lattice digest *$d(z)$ of z can be expressed as*

$$d(z) = \sum_{i \in \mathsf{range}(z)} \mathbf{MA}_{i(t+1)} f_{i(t+1)}(\mathbf{MA}_{i(t+2)} f_{i(t+2)}(\ldots f_{i(\ell-1)}(\mathbf{MA}_{i\ell} f_{i\ell}(\mathbf{x}_i))\ldots)) \mod q\,,$$

where $\mathbf{A}_{ij} = \mathbf{U}$ if $\mathsf{bin}(i)_j = 0$, $\mathbf{A}_{ij} = \mathbf{D}$ if $\mathsf{bin}(i)_j = 1$ and f_{ij} are some $f(.)$ representations.

Proof: (sketch) By induction on the levels of the tree: We use the definition of the digest (Equation 3) recursively on all the nodes of the tree and start applying Corollary 1. Then we can express the digest as a sum of terms that are functions of the specific values stored in the table (full proof in the Appendix). □

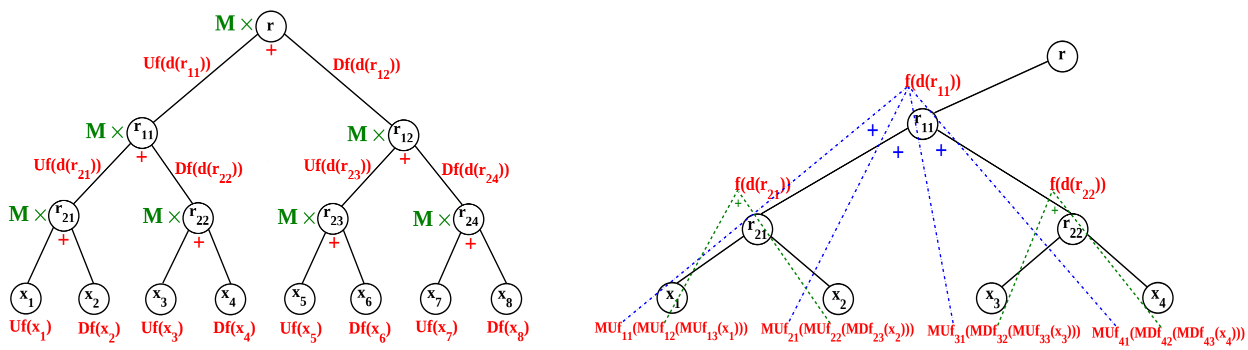

Figure 1: (Left) Tree T built on top of a table with 8 values $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_8$. After producing an $f(.)$ representation (with entries in $\{0, 1, \ldots, \delta\}$) of the child digests, we multiply with either $\mathbf{U}$ or $\mathbf{D}$, then we add the two resulting digests and we compute the hash function on them by multiplying with $\mathbf{M}$. (Right) Left part of the tree and relation (indicated with dash lines) between specific $f_{ij}(.)$ representations of the additive terms computed by Theorem 3 and the $f(.)$ representation of the internal nodes. Note that the $f(.)$ representations of the internal nodes are the sum of those specific $f_{ij}(.)$ representations of the leaves, for example, $f(d(r_{11})) = f_{11} + f_{21} + f_{31} + f_{41} \mod q$.

For example, suppose we have a table of eight values $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_8$. Root r lies at level 0 and the leaves lie at level ℓ, as in Figure 1. Let r_{ij} be the j-th node at level i for $i = 1, \ldots, \ell$, with the numbering going also from the left to the right. The *lattice digest* of the table can be expressed as follows (see Figure 1), according to Theorem 3 and by setting the internal node in Theorem 3 to be the root of the tree:

$$\begin{aligned} d(r) \;=\; & \mathbf{MU}f_{11}(\mathbf{MU}f_{12}(\mathbf{MU}f_{13}(\mathbf{x}_1))) + \mathbf{MU}f_{21}(\mathbf{MU}f_{22}(\mathbf{MD}f_{23}(\mathbf{x}_1))) + \mathbf{MU}f_{31}(\mathbf{MD}f_{32}(\mathbf{MU}f_{33}(\mathbf{x}_3))) + \\ & \mathbf{MU}f_{41}(\mathbf{MD}f_{42}(\mathbf{MD}f_{43}(\mathbf{x}_4))) + \mathbf{MD}f_{51}(\mathbf{MU}f_{52}(\mathbf{MU}f_{53}(\mathbf{x}_5))) + \mathbf{MD}f_{61}(\mathbf{MU}f_{62}(\mathbf{MD}f_{63}(\mathbf{x}_6))) + \\ & \mathbf{MD}f_{71}(\mathbf{MD}f_{72}(\mathbf{MU}f_{73}(\mathbf{x}_7))) + \mathbf{MD}f_{81}(\mathbf{MD}f_{82}(\mathbf{MD}f_{83}(\mathbf{x}_8))) \mod q\,. \end{aligned}$$

Here it is very important to say that the above $f_{ij}(.)$ expressions are some $f(.)$ representations that are **not** the *binary representations* (although it can happen sometimes, in the absence of carries). We recall that the binary representation is only used in the **initial** computation of the digest of the table. The flow of the computation of the digests is depicted in Figure 1.

Digest security. We now give the main security claim for the strong collision resistance of the *lattice digest*, given the results from Merkle [25] and Naor and Nissim [27]. In fact, Naor and Nissim [27] and

Merkle [25] used exactly the same algorithmic construction (i.e., a binary tree) to provide a solution for an authenticated dictionary, generalizing their result for every strong collision resistant hash function h:

Remark 1 (Naor and Nissim [27]) *Possible choices for h include the more efficient MD4 [33], MD5 [34] or SHA [38] (collisions for MD4 and for the compress function of MD5 were found by Dobbertin [12, 13]) and functions based on a computational hardness assumption such as the hardness of discrete log [3, 8, 11] and subset-sum [17, 21] (these are much less efficient).*

The importance of the above remark is that essentially, one can use any strong collision resistant hash function $h(x, y)$ for a Merkle tree construction, given the hash function $h(x, y)$ is secure according to a widely acceptable computational assumption. Namely, it should be difficult (i.e., it should happen with negligible probability $\nu(k)$) for a computationally bounded adversary to find $(x, y) \neq (x', y')$ such that $h(x, y) = h(x', y')$. We therefore have the following result:

Theorem 4 (Strong collision resistance of the lattice digest) *Let k be the security parameter, $m = 2k^2$, $\beta = \delta\sqrt{m}$ and $p \geq 4\sqrt{m}k^{1.5}\beta$ be an odd positive integer. Let also $\mathbf{F} \in \mathbb{Z}_p^{k\times m}$ be a $k \times m$ matrix that is chosen uniformly at random and $\mathbf{M} = \lambda\mathbf{F} \in \mathbb{Z}_q^{k\times m}$ where q and λ are defined in Equation 1. Let also $n = 2^\ell$, $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n \in \mathbb{Z}_q^k$ be the values of the table that is to be authenticated, having a* lattice digest *equal to d. It is computationally infeasible, i.e., it happens with negligible probability $\nu(k)$, for a computationally bounded adversary to find a different table $\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_n \in \mathbb{Z}_q^k$ of lattice digest equal to d, unless there is a polynomial-time algorithm for any instance of the problem GAPSVP_γ for $\gamma = 14\pi\delta\sqrt{km}$.*

Proof: By Remark 1 we can use any strong collision resistant hash function to recursively define a digest of a Merkle tree. Here we are using the function of Equation 2 which is strong collision resistant according to Theorem 2, unless there is a polynomial-time algorithm for any instance of the problem GAPSVP_γ for $\gamma = 14\pi\delta\sqrt{km}$. $\square$

Digest update. Suppose now that $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n \in \mathbb{Z}_q^k$ are the values of the table and that the *lattice digests* have been computed. Let d be the initial *lattice digest* of the table. The objective of the update is to compute the new *lattice digest* of the table, in constant time, whenever the content of some index changes. We show how an update at index $1 \leq w \leq n$ can be performed, which applies for all indices. Note that for index w, where the value $\mathbf{x}_w$ is stored, the additive term from Theorem 3 is

$$\mathsf{term}(\mathbf{x}_w) = \mathbf{M}\mathbf{A}_{w1} f_{w1}(\mathbf{M}\mathbf{A}_{w2} f_{w2}(\ldots f_{w(\ell-1)}(\mathbf{M}\mathbf{A}_{w\ell} f_{w\ell}(\mathbf{x}_w))\ldots)) \mod q\,, \tag{5}$$

where $f_{wi}(.)$ $(i = 1, \ldots, \ell)$ are the suitable $f(.)$ representations (Definition 5) after the *lattice digests* have been computed and Corollary 1 has been applied and $\mathbf{A}_{wj}$ are either $\mathbf{U}$ or $\mathbf{D}$ according to the binary representation of w (see Theorem 3). Let now $q_{w\ell} = \mathbf{x}_w$ be the content of the $f_{w\ell}(.)$ representation in Equation 5 and

$$q_{wi} = \mathbf{M}\mathbf{A}_{w(i+1)} f_{w(i+1)}(\mathbf{M}\mathbf{A}_{w(i+2)} f_{w(i+2)}(\ldots f_{w(\ell-1)}(\mathbf{M}\mathbf{A}_{w\ell} f_{w\ell}(\mathbf{x}_w))\ldots)) \mod q\,, \tag{6}$$

for $i = 1, \ldots, \ell - 1$ be the content of the $f_{wi}(.)$ representation in Equation 5. Note that $\mathbf{x}_w$ does not appear in any other additive term $\mathsf{term}(\mathbf{x}_j)$ for all $j \neq w$ (see Theorem 3). Suppose now we update index w and we replace $\mathbf{x}_w$ with $\mathbf{y}_w$. The new digest, by Theorem 3, can be computed as

$$d' = d - \mathsf{term}(\mathbf{x}_w) + \mathsf{term}(\mathbf{y}_w) \mod q\,. \tag{7}$$

where

$$\mathsf{term}(\mathbf{y}_w) = \mathbf{M}\mathbf{A}_{w1} f'_{w1}(\mathbf{M}\mathbf{A}_{w2} f'_{w2}(\ldots f'_{w(\ell-1)}(\mathbf{M}\mathbf{A}_{w\ell} f'_{w\ell}(\mathbf{y}_w))\ldots))) \mod q\,, \tag{8}$$

and where now the updated q'_{wi} values are defined as in Equation 6. Note however that one cannot use any radix-2 representation $f'_{wi}(.)$ $(i = 1, \ldots, \ell)$ during the update (Equation 8). One has to be careful to use such a representation that does not violate the "small input" requirement (i.e., vector entries in $\{0, 1 \ldots, \delta\}$) for the hash function, after the update takes place. The new $f'_{wi}(.)$ representations, in order to satisfy that, crucial for the security (see Definition 2), requirement, are computed according to the following definition:

Definition 8 (Updated radix-2 representations) *Suppose the value of index w is $\mathbf{x}_w$. An update is issued and the value of index w changes to $\mathbf{y}_w$. Let $q'_{w\ell} = \mathbf{y}_w$. Then, for $i = \ell, \ldots, 1$ the updated q'_{wi} and $f'_{wi}(.)$ values are computed as follows, (a) $f'_{wi}(q'_{wi}) = f_{wi}(q_{wi}) + f_{\mathsf{bin}}(q'_{wi} - q_{wi}) \mod q$, and (b) $q'_{w(i-1)} = \mathbf{MA}_{wi} f'_{wi}(q'_{wi}) \mod q$, where $q'_{w0} = \mathsf{term}(\mathbf{y}_w)$ and $\mathbf{A}_{wj}$ are either $\mathbf{U}$ or $\mathbf{D}$ according to the binary representation of w.*

The representation $f'_{wi}(q'_{wi})$ computed in Definition 8 is a correct $f(.)$ representation of q'_{wi}, since, as $f_{\mathsf{bin}}(q'_i - q_i)$ is an $f(.)$ representation, by Corollary 1 we have $f'_{wi}(q'_{wi}) = f(q_{wi} + q'_{wi} - q_{wi}) = f(q'_{wi}) \mod q$, which is a correct $f(.)$ representation of q'_{wi}. We now present the main theorem of this section:

Theorem 5 *Let $n = 2^\ell$, $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n \in \mathbb{Z}_q^k$ be the values of the table that is to be authenticated and T be the complete binary tree of ℓ levels that is built on top of the table. For $i = \ell, \ldots, 1$, let $\{v_i\}$ be the logarithmic-sized path from some index w to the root's child v_1, $d(v_i)$ be the respective* lattice digests *and $f(d(v_i)) \in \{0, 1, \ldots, \delta\}^{m/2}$ be the $f(.)$ representations of them. An update is issued and the value of index w changes to $\mathbf{y}_w$. If $f(d'(v_i))$, $i = \ell, \ldots, 1$ are the updated $f(.)$ representations of the path nodes, then for every $i = \ell, \ldots, 1$, after the update, it holds $f(d'(v_i)) \leq f(d(v_i)) + [1\ 1\ \ldots\ 1]^T$, where inequality is defined component-wise and $[1\ 1\ \ldots\ 1]^T$ has size $m/2$.*

Proof: (sketch) The relation between an $f(.)$ representation of a node v_t lying at level t and the f_{ij} representations of the leaf nodes is $f(v_t) = \sum_{i \in \mathsf{range}(v)} f_{it}(q_{it})$, where q_{it} are defined in Equation 6 (see Figure 1). If we update the $f_{ij}(.)$ representations according to Definition 8, the entries of the $f(.)$ representations of the internal nodes can be increased by at most 1 (a binary vector is added) (full proof in Appendix). □

Note that the above theorem is very important for proving the desired update complexity (see Theorem 6) since it ensures, that even after updates, the security of the hash function (small inputs) is maintained.

4 Authenticated data structure

In this section we describe how exactly the lattice-based construction is used in a three-party authenticated data structure model, which consists of three entities, the *trusted* source, the *untrusted* servers and the clients. Let $1, \ldots, n$ be the indices of the table and $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n \in \mathbb{Z}_q^k$ are the *initial* values of the table. Due to space limitations, all the proofs in this section appear in the Appendix.

System setup. We fix the parameters that we are using in our construction as follows: We recall that k is the security parameter, $\mathbf{M}$ is a $k \times m$ matrix with elements sampled uniformly at random from $\mathbb{Z}_q$, $m = 2k^2$, $\beta = \delta\sqrt{m}$, $p \geq 4\sqrt{m}k^{1.5}\beta$, q is a k-bit modulus and $\lambda = q/p$. It is easy to see that given k and δ there is always a $p = O(k^{3.5}\delta)$ to satisfy the above constraints. Let's set $p = \lceil c_1 k^{3.5}\delta \rceil + 1$ or $p = \lceil c_1 k^{3.5}\delta \rceil$ such that p is an odd positive integer, as required by Theorem 1, for some suitable constant c_1. Finally we set $\delta = n$, where n is the size of our structure, which is a polynomially bounded value (we are in the computational model). This setup, by Theorem 2, will give a construction that is secure based on the difficulty of GAPSVP_γ for $\gamma = 14\pi\delta\sqrt{km}$. In specific, since $m = 2k^2$ and $\delta = n$ we have that $\gamma = O(nk\sqrt{k}) = O(k^c)$ for some $c = O(1)$.

Source. We recall that in each index in $\{1, \ldots, n\}$ the source can store one of the values of the set $S = \{0, 1, \ldots, C\}$. Each element of the set S is represented with a distinct element of $\mathbb{Z}_q^k$ and $|S| = O(1)$. Note that the possible states of the table is therefore $|S|^n$, exponentially large. Suppose now that $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n \in \mathbb{Z}_q^k$ are the *initial* values of the table and that the *lattice digests* have been computed using the *binary* representations in Equation 3. The source, for each index $w \in \{1, \ldots, n\}$ does the following precomputations: For each value $y_{wj} \in S - \{x_w\}$ ($j = 0, 1, \ldots, C$) it computes and *stores* $\mathsf{term}(y_{wj})$ as defined in Equation 5 and by using Definition 8 to compute the updated $f'_{wi}(.)$ representations. The initial

q_{wi} values that appear in Definition 8 are the ones derived by $\mathsf{term}(x_w)$, the initial value of the index. These initial q_{wi} values are used for the computation of all the the updated $f'_{wi}(.)$ representations that correspond to terms $\mathsf{term}(y_{wj})$. Note that the source does not have to store the tree and the digest of the internal nodes, since the source is only interested in correctly updating the lattice digest.

Theorem 6 *The source update time is $O(1)$ per update, the source performs $O(1)$ group operations per update and keeps $O(n)$ space. Moreover, the update authentication information has size $O(1)$ and consists of $O(1)$ group elements.*

Servers. The servers, whenever an update at index w is issued by the source, have to update the *lattice digest* in the same way that the source did. Therefore they could achieve this task again in $O(1)$ time. However, since they have to provide proofs to the clients for future queries, they have to update the digests of the internal nodes (the nodes belonging to the logarithmic-sized path from index w to the root of the tree) that are influenced by the update and as a result the *servers* update time cannot be $O(1)$:

Theorem 7 *The server update time is $O(\log^2 n)$ per update, the server performs $O(\log^2 n)$ operations in $\mathbb{Z}_q^k$ per update and keeps $O(n)$ space. Also, the server query time is $O(\log n)$, the proof for a query has size $O(\log n)$ and consists of $O(\log n)$ group elements.*

Clients. Suppose a client sends a query to the server for the value of index w. After the client verifies the freshness of the lattice digest sent by the source (which takes time $O(1)$), it verifies the logarithmic sized proof sent by the server by performing multiplications with matrix **M**, until the client computes the authentic digest sent by the source. This verification is very similar (only the cryptographic primitive changes) with the one performed when using a Merkle tree [25]. If there is a match with the signed digest, the client accepts the answer, else it rejects. Each multiplication at every node of the path takes time $O(k^4)$ and since it is performed $O(\log n)$ times, the verification time is $O(k^4 \log n) = O(\log n)$. We give the following result for the client:

Theorem 8 *The client verification time is $O(\log n)$ per query, the client performs $O(\log n)$ operations in $\mathbb{Z}_q^k$ per query and the client keeps $O(1)$ local space.*

Putting everything together we can state our main theorem for the three-party model:

Theorem 9 *Let k be the security parameter. Then there exists a three-party authenticated data structure for authenticating a dynamic table of n indices such that: (1) It is secure according to Definition 4 and assumming the hardness of GAPSVP_γ for $\gamma = O(nk\sqrt{k})$; (2) The source update time is $O(1)$ and involves $O(1)$ group operations; (3) The server update time is $O(\log^2 n)$ and involves $O(\log^2 n)$ group operations; (4) The source space is $O(n)$; (5) The server space is $O(n)$; (6) The client space is $O(1)$; (7) The server query time is $O(\log n)$; (8) The client verification time is $O(\log n)$ and involves $O(\log n)$ group operations; (9) The proof has size $O(\log n)$ and consists of $O(\log n)$ group elements; (10) The update authentication information has size $O(1)$ and consists of $O(1)$ group elements.*

Proof: The security is proved from Theorem 4, i.e., we are using a provably secure collision resistant hash function and we maintain its security under updates (by using Theorem 5). All the other points are due to Theorems 6, 7 and 8. Also note that $\gamma = O(nk\sqrt{k})$, since by Theorem 4 we need $\gamma = 14\pi\delta\sqrt{km}$ and, $m = 2k^2$ and $\delta = n$. □

5 Authenticated Bloom filters and discussion

In this section we show how we can use the lattice-based hash function to authenticate the Bloom filter functionality, a space efficient dictionary data structure, originally introduced in [6]. The Bloom filter consists of an array (table) $A[0 \ldots n-1]$ storing n bits. All the bits are initially set to 0. Suppose one needs to store a set S of r elements. Then K hash functions $h_i(.)$ with range $\{0, \ldots, n-1\}$ are used and for each element $s \in S$ we set the bits $A[h_i(s)]$ to 1, for $i = 1, \ldots, K$. In this way, false positives can occur, i.e., even if an element does not belong to the S, it might be represented in A. The probability of a false positive can be proved to be $(1-p)^K$, where $p = e^{-Kr/n}$, which is minimized for $K = \ln 2(n/r)$ [6].

The Bloom filter above supports only insertions though. A deletion (i.e., setting some bits to 0) can cause the undesired deletion of many elements. To deal with this problem, *counting Bloom filters* were introduced by Fan et al. [15]. In this solution, by keeping a counter for each index of A (instead of just 0 or 1), we can tolerate deletions by incrementing the counter during insertions and decrementing the counter during deletions. However, the problem of *overflow* exists. As observed in [9], the overflow (at least one counter goes over some value C) occurs with probability $n(e \ln 2/C)^C$, for a certain set of r elements. Setting $C = O(1)$ (e.g., $C = 16$) is suitable for most of the applications [9].

By the above description, it is clear that we can use our lattice-based construction to authenticate the Bloom filter functionality: Each index of our table can take values from the set $\{0, \ldots, C\}$, where $C = O(1)$. Note that constant update complexity in this application is very important given that a Bloom filter is an *update-intensive* data structure (i.e., an insertion or deletion of an element involves K operations). Therefore we have the following result:

Theorem 10 *Let k be the security parameter. Then there exists a three-party authenticated data structure for authenticating a Bloom filter of size n, storing r elements and using K hash functions such that: (1) It is secure according to Definition 4 and assumming the hardness of* GAPSVP_γ *for $\gamma = O(nk\sqrt{k})$; (2) The source update time is $O(K)$ and involves $O(K)$ group operations; (3) The server update time is $O(K \log^2 n)$ and involves $O(K \log^2 n)$ group operations; (4) The source space is $O(n)$; (5) The server space is $O(n)$; (6) The client space is $O(1)$; (7) The server query time is $O(K \log n)$; (8) The client verification time is $O(K \log n)$ and involves $O(K \log n)$ group operations; (9) The proof has size $O(K \log n)$ and consists of $O(K \log n)$ group elements; (10) The update authentication information has size $O(1)$ and consists of $O(1)$ group elements.*

It is safe to assume the hardness of GAPSVP_γ, for $\gamma = O(nk\sqrt{k})$. This is because we are in the computational model, therefore n has to be polynomial in the security parameter and GAPSVP_γ is assumed to be hard for any γ polynomial in k [32]. For future work we envision reducing the complexities of our construction (e.g., server update) and, more importantly, applying lattices to more authenticated data structures problems, e.g., deriving a lattice-based cryptographic accumulator.

References

[1] M. Ajtai. Generating hard instances of lattice problems (extended abstract). In *STOC*, pages 99–108, 1996.

[2] N. Baric and B. Pfitzmann. Collision-free accumulators and fail-stop signature schemes without trees. In *Advances in Cryptology: Proc. EUROCRYPT*, volume 1233 of *LNCS*, pages 480–494. Springer-Verlag, 1997.

[3] M. Bellare, O. Goldreich, and S. Goldwasser. Incremental cryptography: The case of hashing and signing. In *Advances in Cryptology—CRYPTO '94*, volume 839 of *LNCS*, pages 216–233. Springer-Verlag, 1994.

[4] M. Bellare and D. Micciancio. A new paradigm for collision-free hashing: Incrementality at reduced cost. In *EUROCRYPT: Advances in Cryptology: Proceedings of EUROCRYPT, LNCS 1233*, pages 163–192. Springer-Verlag, 1997.

[5] J. Benaloh and M. de Mare. One-way accumulators: A decentralized alternative to digital signatures. In *Advances in Cryptology—EUROCRYPT 93*, volume 765 of *LNCS*, pages 274–285. Springer-Verlag, 1993.

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

[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] S. Brands. An efficient off-line electronic cash system based on the representation problem. CWI Technical Report, CS-R9323, 1993.

[9] A. Broder and M. Mitzenmacher. Network applications of Bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2005.

[10] J. Camenisch and A. Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In *Proc. CRYPTO*, 2002.

[11] D. Chaum, E. van Heijst, and B. Pfitzmann. Cryptographically strong undeniable signatures, unconditionally secure for the signer. In *CRYPTO*, pages 470–484, 1991.

[12] H. Dobbertin. Cryptanalysis of MD4. In *Fast Software Encryption — FSE 1996*, volume 1039 of *Lecture Notes in Computer Science*, pages 53–69. Springer Verlag, 1996.

[13] H. Dobbertin. Cryptanalysis of MD5 compress. Presented at the Rump-session of EUROCRYPT '96, may 1996.

[14] C. Dwork, M. Naor, G. N. Rothblum, and V. Vaikuntanathan. How efficient can memory checking be? In *TCC*, pages 503–520, 2009.

[15] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Networking*, 8(3):281–293, 2000.

[16] C. Gentry, C. Peikert, and V. Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In *STOC '08: Proceedings of the 40th annual ACM symposium on Theory of computing*, pages 197–206, New York, NY, USA, 2008. ACM.

[17] O. Goldreich, S. Goldwasser, and S. Halevi. Collision-free hashing from lattice problems, 1996.

[18] M. T. Goodrich, R. Tamassia, and J. Hasic. An efficient dynamic and distributed cryptographic accumulator. In *Proc. of Information Security Conference (ISC)*, volume 2433 of *LNCS*, pages 372–388. Springer-Verlag, 2002.

[19] M. T. Goodrich, R. Tamassia, and A. Schwerin. Implementation of an authenticated dictionary with skip lists and commutative hashing. In *Proc. DARPA Information Survivability Conference and Exposition II (DISCEX II)*, pages 68–82, 2001.

[20] B. Hayes. Cloud computing. *Commun. ACM*, 51(7):9–11, 2008.

[21] R. Impagliazzo and M. Naor. Efficient cryptographic schemes provably as secure as subset sum. *J. Cryptology*, 9(4):199–216, 1996.

[22] A. K. Lenstra, H. W. L. Jr, and L. Lovasz. Factoring polynomials with rational coefficients. *Math.Ann.*, (261):515–534, 1982.

[23] V. Lyubashevsky and D. Micciancio. Generalized compact knapsacks are collision resistant. In *ICALP (2)*, pages 144–155, 2006.

[24] C. Martel, G. Nuckolls, P. Devanbu, M. Gertz, A. Kwong, and S. G. Stubblebine. A general model for authenticated data structures. *Algorithmica*, 39(1):21–41, 2004.

[25] R. C. Merkle. A certified digital signature. In G. Brassard, editor, *Proc. CRYPTO '89*, volume 435 of *LNCS*, pages 218–238. Springer-Verlag, 1989.

[26] D. Micciancio and O. Regev. Worst-case to average-case reductions based on gaussian measures. *SIAM J. Comput.*, 37(1):267–302, 2007.

[27] M. Naor and K. Nissim. Certificate revocation and certificate update. In *Proc. 7th USENIX Security Symposium*, pages 217–228, Berkeley, 1998.

[28] M. Naor and G. N. Rothblum. The complexity of online memory checking. *J. ACM*, 56(1), 2009.

[29] L. Nguyen. Accumulators from bilinear pairings and applications. In *Proc. CT-RSA 2005, LNCS 3376, pp. 275-292, Springer-Verlag, 2005.*, 2005.

[30] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Authenticated hash tables. In *Proc. ACM Conference on Computer and Communications Security (CCS)*, pages 437–448. ACM, October 2008.

[31] C. Peikert and A. Rosen. Efficient collision-resistant hashing from worst-case assumptions on cyclic lattices. *Electronic Colloquium on Computational Complexity (ECCC)*, (158), 2005.

[32] O. Regev. On the complexity of lattice problems with polynomial approximation factors. *The LLL algorithm*, pages 475–496, 2010.

[33] R. Rivest. The MD4 message-digest algorithm. RFC 1320, Apr. 1992.

[34] R. Rivest. The MD5 message-digest algorithm. RFC 1321, Apr. 1992.

[35] T. Sander, A. Ta-Shma, and M. Yung. Blind, auditable membership proofs. In *Proc. Financial Cryptography (FC 2000)*, volume 1962 of *LNCS*. Springer-Verlag, 2001.

[36] R. Tamassia. Authenticated data structures. In *Proc. European Symp. on Algorithms*, volume 2832 of *LNCS*, pages 2–5. Springer-Verlag, 2003.

[37] R. Tamassia and N. Triandopoulos. Computational bounds on hierarchical data processing with applications to information security. In *Proc. Int. Colloquium on Automata, Languages and Programming (ICALP)*, volume 3580 of *LNCS*, pages 153–165. Springer-Verlag, 2005.

[38] U.S. National Institute of Standards and Technology. Secure hash standard. Federal Information Processing Standards Publication 180, 1992.

6 Appendix

6.1 Proof of Theorem 3

We prove the claim by induction on the levels of the tree T. For any internal node u that lies at level $\ell - 1$, there are only two nodes (that store for example values $\mathbf{x}_i$ (left child and odd index i) and $\mathbf{x}_j$ (right child and even index j) and belong to $\mathsf{range}(u)$) in the subtree rooted on u, the root is u and therefore, by Equation 3 we indeed have

$$d(u) = \mathbf{M}\left[\mathbf{U}f(\mathbf{x}_i) + \mathbf{D}f(\mathbf{x}_j)\right] = \mathbf{MU}f_{\mathsf{bin}}(\mathbf{x}_i) + \mathbf{MD}f_{\mathsf{bin}}(\mathbf{x}_j)\,,$$

where $\mathbf{A}_{i1} = \mathbf{U}$ and $\mathbf{A}_{j1} = \mathbf{D}$, since i is odd and j is even. Assume the theorem holds for any internal node v that lies at level $0 < t + 1 \leq \ell$. Therefore

$$d(v) = \sum_{i \in \mathsf{range}(v)} \mathbf{MA}_{i(t+2)} f_{i(t+2)}(\mathbf{MA}_{i(t+3)} f_{i(t+3)}(\ldots f_{i(\ell-1)}(\mathbf{MA}_{i\ell} f_{i\ell}(\mathbf{x}_i))\ldots)) \mod q\,,$$

where $\mathbf{A}_{ij} = \mathbf{U}$ if $\mathsf{bin}(i)_j = 0$ and $\mathbf{A}_{ij} = \mathbf{D}$ if $\mathsf{bin}(i)_j = 1$ and $f_{ij}()$ are some $f(.)$ representations. For any internal node z that lies at level t, the new digest is produced by combining (see Equation 3) two digests that correspond to two different trees rooted at level $t+1$ and of roots $\mathsf{left}(z)$ and $\mathsf{right}(z)$ (the left and the right child of z respectively), i.e.,

$$\begin{aligned} d(z) &= \mathbf{M}[\mathbf{U}f(d(\mathsf{left}(z))) + \mathbf{D}f(d(\mathsf{right}(z)))] \mod q \\ &= \mathbf{MU}f\left(\sum_{i \in \mathsf{range}(\mathsf{left}(z))} \mathbf{MA}_{i(t+2)} f_{i(t+2)}(\mathbf{MA}_{i(t+3)} f_{i(t+3)}(\ldots f_{i(\ell-1)}(\mathbf{MA}_{i\ell} f_{i\ell}(\mathbf{x}_i))\ldots))\right) \\ &+ \mathbf{MD}f\left(\sum_{i \in \mathsf{range}(\mathsf{right}(z))} \mathbf{MA}_{i(t+2)} f_{i(t+2)}(\mathbf{MA}_{i(t+3)} f_{i(t+3)}(\ldots f_{i(\ell-1)}(\mathbf{MA}_{i\ell} f_{i\ell}(\mathbf{x}_i))\ldots))\right) \mod q\,. \end{aligned}$$

By applying Corollary 1, we can break each $f(.)$ expression into multiple $f(.)$ expressions and therefore we have

$$d(z) = \sum_{i \in \mathsf{range}(z)} \mathbf{MA}_{i(t+1)} f_{i(t+1)}(\mathbf{MA}_{i(t+2)} f_{i(t+2)}(\ldots f_{i(\ell-1)}(\mathbf{MA}_{i\ell} f(\mathbf{x}_i)) \ldots)) \mod q\,,$$

where $\mathbf{A}_{ij} = \mathbf{U}$ if $\mathsf{bin}(i)_j = 0$ and $\mathbf{A}_{ij} = \mathbf{D}$ if $\mathsf{bin}(i)_j = 1$. This completes the proof. $\square$

6.2 Proof of Theorem 5

Let v_t be an internal node of T at level $1 \leq t \leq \ell$, $T(v_t)$ be the subtree rooted on v_t and $\mathsf{range}(v_t)$ be its range, where $w \in \mathsf{range}(v_t)$. By Theorem 3 we have that, before the update it is

$$d(v_t) = \sum_{i \in \mathsf{range}(v_t)} \mathsf{term}(\mathbf{x}_i) = \mathsf{term}(\mathbf{x}_w) + \sum_{i \in \mathsf{range}(v_t) - \{w\}} \mathsf{term}(\mathbf{x}_i) \mod q\,.$$

After we apply Corollary 1 and by using the notation of Definition 8 we have

$$f(d(v_t)) = f(\mathsf{term}(\mathbf{x}_w)) + f\left(\sum_{i \in \mathsf{range}(v_t) - \{w\}} \mathsf{term}(\mathbf{x}_i)\right) = f_{wt}(q_{wt}) + f\left(\sum_{i \in \mathsf{range}(v_t) - \{w\}} \mathsf{term}(\mathbf{x}_i)\right) \mod q\,.$$

By the way updates are performed (see Definition 8) the quantity $f(d(v_t)) + f_{\mathsf{bin}}(q'_t - q_t)$ can be written as

$$\begin{aligned}
f(d(v_t)) + f_{\mathsf{bin}}(q'_{wt} - q_{wt}) &= f_{\mathsf{bin}}(q'_{wt} - q_{wt}) + f_{wt}(q_{wt}) + f\left(\sum_{i \in \mathsf{range}(v_t) - \{w\}} \mathsf{term}(\mathbf{x}_i)\right) \mod q \\
&= f'_{wt}(q'_{wt}) + f\left(\sum_{i \in \mathsf{range}(v_t) - \{w\}} \mathsf{term}(\mathbf{x}_i)\right) \mod q \\
&= f(\mathsf{term}(\mathbf{y}_w)) + f\left(\sum_{i \in \mathsf{range}(v_t) - \{w\}} \mathsf{term}(\mathbf{x}_i)\right) \mod q \\
&= f\left(\mathsf{term}(\mathbf{y}_w) + \sum_{i \in \mathsf{range}(v_t) - \{w\}} \mathsf{term}(\mathbf{x}_i)\right) \mod q \\
&= f(d'(v_t)) \mod q\,.
\end{aligned}$$

From the above argument and by the facts that $f(d(v_t)) \in \{0, 1, \ldots, \delta\}^{m/2}$ and $\mathbf{0} \leq f_{\mathsf{bin}}(q'_{wt} - q_{wt}) \leq [1\ 1\ \ldots\ 1]^T$, we have that

$$f(d'(v_t)) = f(d(v_t)) + f_{\mathsf{bin}}(q'_{wt} - q_{wt}) \Rightarrow f(d'(v_t)) \leq f(d(v_t)) + [1\ 1\ \ldots\ 1]^T\,.$$

$\square$

6.3 Proof of Theorem 6

Assume the setup of Section 4. Suppose the initial state of the table is $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n \in \mathbb{Z}_q^k$ and that the initial digest of the table is d. As we showed before, for each index $w \in \{1, \ldots, n\}$ the source does the following precomputations: For each value $y_{wj} \in S - \{x_w\}$ ($j = 0, 1, \ldots, C$) it computes and stores $\mathsf{term}(y_{wj})$ as defined in Equation 5, where $S = \{0, \ldots, C\}$. Each term $\mathsf{term}(y_{wj})$ is an element in $\mathbb{Z}_q^k$ and therefore the source needs $O(k^2) \times O(|S|)$ bits for each index w. Therefore the space needed is $O(n)$.

The source issues an update that changes the value of index w from $\mathbf{x}_w$ to $\mathbf{y}_w$. Then the updated digest d' is computed by Equation 7 by setting

$$d' = d - \mathsf{term}(\mathbf{x}_w) + \mathsf{term}(\mathbf{y}_w) \mod q\,,$$

which requires two additions (i.e., $O(1)$ operations) in $\mathbb{Z}_q^k$, which take time $O(k^2) = O(1)$ (k is a constant). By using the precomputed table with the $g(.)$ representations, this can be done is $O(1)$ time, increasing the space to $O(n\delta \log \delta)$, as shown in Section 4. We now prove that there is no internal node of the tree, whose lattice digest has an $f(.)$ that has coordinates not in $\{0, 1, \ldots, \delta = n\}$, and therefore during all the updates, secure digests are being produced. Suppose in the worst case, there is an internal node v such that all the logarithmic-sized paths of the updates cross through it. Let $d_0(v)$ be the digest of v in the initial state. Then, since the binary $f(.)$ representation is used in the beginning, we have that

$$f(d_0(v)) \leq [1\ 1\ \ldots\ 1]^T\,,$$

where $[1\ 1\ \ldots\ 1]^T$ has size $m/2$. Suppose an update is issued which increases the Hamming distance by one and which changes the digest to $d_1(v)$ (note that updates that do not increase the Hamming distance, i.e., they update already updated indices do not increase the bound of Theorem 5). Then by Theorem 5 we have

$$f(d_1(v)) \leq f(d_0(v)) + [1\ 1\ \ldots\ 1]^T \leq 2[1\ 1\ \ldots\ 1]^T\,.$$

Similarly, for the i-th update that increases the Hamming distance by one we have that

$$f(d_i(v)) \leq f(d_{i-1}(v)) + [1\ 1\ \ldots\ 1]^T \leq (i+1)[1\ 1\ \ldots\ 1]^T\,.$$

This implies that while the table is kept in Hamming distance δ, there cannot be any internal node, whose $f(.)$ representation has a coordinate greater than δ, as required by the "small input" constraint (Equation 4). Since we have set $\delta = n$ and the maximum Hamming distance is n, we can have unlimited updates. This means that at any state of the table, there cannot be an internal node whose $f(.)$ rerpesentation violates Equation 4. As for the update authentication information, this is a signature of the lattice digest, which is $O(1)$ bits long and therefore the signature is also $O(1)$ bits. □

6.4 Proof of Theorem 7

Suppose the initial state of the table is $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n \in \mathbb{Z}_q^k$. The server stores the binary tree on top of the table, and at each internal node v of the binary tree it also stores the $f(.)$ representations of two lattice digests: Firstly, it stores the binary representation $f(d_0(v))$ of the lattice digest $d_0(v) \in \mathbb{Z}_k^q$ of node v that corresponds to the initial state of the table. Secondly it stores the $f(.)$ representation $f(d(v))$ of the *current* lattice digest of the table, denoted with $d(v)$. Since each $f(.)$ representation requires $O(k^3)$ bits (we recall that each $f(.)$ representation has k^2 entries in $\mathbb{Z}_q$ and therefore $O(k^3)$ bits are needed) and the tree has $O(n)$ nodes in total, the server needs space $O(k^3 n) = O(n)$. Suppose now an update is issued, that changes the value of the index w from $\mathbf{x}_w$ to $\mathbf{y}_w$. Let $v_\ell, v_{\ell-1}, \ldots, v_1$ be the path from the node of index w to the child v_1 of the root of the tree. Let

$$\mathsf{term}(\mathbf{x}_w) = \mathbf{MA}_{w1} f_{w1}(\mathbf{MA}_{w2} f_{w2}(\ldots f_{w(\ell-1)}(\mathbf{MA}_{w\ell} f_{w\ell}(\mathbf{x}_w))\ldots)) \mod q\,,$$

and

$$\mathsf{term}(\mathbf{y}_w) = \mathbf{MA}_{w1} f'_{w1}(\mathbf{MA}_{w2} f'_{w2}(\ldots f'_{w(\ell-1)}(\mathbf{MA}_{w\ell} f'_{w\ell}(\mathbf{x}_w))\ldots)) \mod q\,.$$

Let now q_{ij} and q'_{ij} be the contents of the representations $f_{ij}(.)$ and $f'_{ij}()$, as defined in Equation 6. Note that according to the proof of Theorem 5, we have that, for $i = \ell, \ldots, 1$

$$f(d'(v_i)) = f(d(v_i)) + f_{\mathsf{bin}}(q'_{wi} - q_{wi}) \mod q\,.$$

For $i = \ell, \ldots, 1$ all this computation can be performed in $O(k^3 \log n) = O(\log n)$ time, given q'_{wi} and q_{wi} are known. We show now how we compute q_{wi} and q'_{wi}. We first compute q_{wi} and $f_{wi}(q_{wi})$. If we begin from the $f(.)$ representation of $d_0(v_1)$, which is actually a binary representation since it is refers to the initial state of the table, we have that

$$f_{\mathsf{bin}}(d_0(v_1)) = f_{\mathsf{bin}}(\mathbf{M}[\mathbf{U} f_{\mathsf{bin}}(d(\mathsf{left}(v_1))) + \mathbf{D} f_{\mathsf{bin}}(d(\mathsf{right}(v_1)))]) \mod q\,.$$

Assume that $v_i = \mathsf{left}(v_{i-1})$ (the same applies for every other combination) for all $i = 2, \ldots, \ell$. Then the above is written as

$$\begin{aligned} f_{\mathsf{bin}}(d_0(v_1)) &= f_{\mathsf{bin}}(\mathbf{M}[\mathbf{U} f_{\mathsf{bin}}(d_0(v_2)) + \mathbf{D} f_{\mathsf{bin}}(d_0(\mathsf{right}(v_1)))]) \\ &= f_{\mathsf{bin}}(\mathbf{MU} f_{\mathsf{bin}}(d_0(v_2)) + \mathbf{MD} f_{\mathsf{bin}}(d_0(\mathsf{right}(v_1)))) \\ &= f^*(\mathbf{MU} f_{\mathsf{bin}}(d_0(v_2))) + f^*(\mathbf{D} f_{\mathsf{bin}}(d_0(\mathsf{right}(v_1)))) \mod q\,, \end{aligned} \tag{9}$$

where $f^*(.)$ are some $f(.)$ representations that are computed deterministically, and, in the same way by the source: For example, it can be the case that the left $f^*(.)$ expression is always a binary representation. But how long does it take to implement the above "break-up" and compute the $f^*(.)$ representations? First of all, one needs to compute the products $\mathbf{MU} f_{\mathsf{bin}}(d(v_2))$ and $\mathbf{MD} f_{\mathsf{bin}}(d(\mathsf{right}(v_1)))$, which take time $O(k^4)$ and then run the deterministic algorithm that computes the two $f^*(.)$ representations in $O(k^3)$ time, since, in order to compute a binary representation $f^*(.)$ one needs time $O(k^2)$ and the substraction to compute the other $f^*(.)$ representation takes time $O(k^3)$.

Note now that we are interested to continue this computation only for the left term of Equation 9. Therefore

$$\begin{aligned} f^*(\mathbf{MU} f_{\mathsf{bin}}(d_0(v_2))) &= f^*(\mathbf{MU} f_{\mathsf{bin}}(\mathbf{M}[\mathbf{U} f_{\mathsf{bin}}(d_0(v_3)) + \mathbf{D} f_{\mathsf{bin}}(d_0(\mathsf{right}(v_2)))])) \\ &= f^*(\mathbf{MU} f_{\mathsf{bin}}(\mathbf{MU} f_{\mathsf{bin}}(d_0(v_3)) + \mathbf{MD} f_{\mathsf{bin}}(d_0(\mathsf{right}(v_2))))) \\ &= f^*(\mathbf{MU} f^*(\mathbf{MU} f_{\mathsf{bin}}(d_0(v_3))) + \mathbf{MU} f^*(\mathbf{MD} f_{\mathsf{bin}}(d_0(\mathsf{right}(v_2))))) \\ &= f^{**}(\mathbf{MU} f^*(\mathbf{MU} f_{\mathsf{bin}}(d_0(v_3)))) + f^{**}(\mathbf{MU} f^*(\mathbf{MD} f_{\mathsf{bin}}(d_0(\mathsf{right}(v_2))))) \mod q\,. \end{aligned}$$

Namely, at the second level, we have one more "break-up" and therefore, in this way, at the $O(\log n)$-th level we have $O(\log n)$ "break-ups". Therefore the time complexity of computing q_{wi} and $f_{wi}(q_{wi})$ is

$$\sum_{i=1}^{O(\log n)} O(ik^4) = O(k^4 \log^2 n)\,,$$

which makes that time complexity of the update algorithm equal to $O(\log^2 n)$ (we recall, that, in our setting, k is a constant). The query time involves the computation of the proof, basically computing the collection of $f(.)$ representations along the path of the queried index. The proof is going to be the following logarithmic-sized tuple:

$$\{f(d(v_\ell)), f(d(\mathsf{sib}(v_\ell))), f(d(v_{\ell-1})), f(d(\mathsf{sib}(v_{\ell-1}))), \ldots, f(d(v_1)), f(d(\mathsf{sib}(v_1)))\}\,,$$

exactly as is done in the computation of a Merkle tree proof. This takes $O(k^3 \log n) = O(\log n)$ time to compute, since we have to collect $O(\log n)$ vectors of $O(k^3)$ bits each, which makes the proof size also $O(k^3 \log n) = O(\log n)$. □

6.5 Proof of Theorem 8

Suppose the client queries for index w. Let $v_\ell, v_{\ell-1}, \ldots, v_1$ be the path from the node of index w to the child v_1 of the root of the tree. The server computes the following proof

$$\{f(d(v_\ell)), f(d(\mathsf{sib}(v_\ell))), f(d(v_{\ell-1})), f(d(\mathsf{sib}(v_{\ell-1}))), \ldots, f(d(v_1)), f(d(\mathsf{sib}(v_1)))\}$$

and also sends the answer "the value of index w is $\mathbf{r}_w$". The client checks to see if $f(d(v_\ell)) = f(\mathbf{r}_w)$ and accordingly performs the following checks:

$$f(\mathbf{M}[\mathbf{A}_{i1} f(d(v_i)) + \mathbf{A}_{i2} f(d(\mathsf{sib}(v_\ell)))]) = f(d(v_{i-1}))\,?$$

for $i = \ell, \ldots, 2$ and where $\mathbf{A}_{i1}$ and $\mathbf{A}_{i2}$ are either $\mathbf{U}$ or $\mathbf{D}$ depending on the binary representation of w. During these computations the client should also check to see that the coordinates of the $f(.)$ representations are in $\{0, 1, \ldots, n\}$, so that the constraint of Equation 4 is satisfied. Finally, if d is the authentic digest received by the source the client performs the final verification, i.e., he checks to see if $\mathbf{M}[\mathbf{A}_{11} f(d(v_1)) + \mathbf{A}_{12} f(d(\mathsf{sib}(v_1)))] = d$? If all the checks succeed, then the client accepts the answer, otherwise the client rejects. Since the client has to do $O(\log n)$ checks, each one taking time $O(k^4)$, since matrix multiplications are involved, the verification time is $O(k^4 \log n) = O(\log n)$. Finally, the client needs only to locally store the public key of the source, in order to verify the signature on the digest. Therefore the local space needed is $O(1)$. $\square$