Cryptography for Efficiency: Authenticated Data Structures Based on Lattices and Parallel Online Memory Checking

Charalampos Papamanthou and Roberto Tamassia

Department of Computer Science, Brown University, Providence RI, USA

Abstract. In this work, we initially design a new authenticated data structure for a *dynamic table* with n entries. We present the first dynamic authenticated table that is *update-optimal*, using a *lattice-based* construction. In particular, the update complexity is $O(1)$, improving in this way the "a priori" $O(\log n)$ update bounds of previous constructions, such as the Merkle tree. Moreover, the space complexity of our authenticated data structure is $O(n)$ and logarithmic bounds hold for other performance measures, such as proof complexity (number of group elements contained in the proof). To achieve this result, we establish and exploit a property that we call *repeated linearity* of lattice-based hash functions and show how the security of lattice-based digests can be guaranteed under updates. An one-time preprocessing stage of $O(n \log n)$ complexity is also required at setup. This is the first construction achieving a constant update bound without causing other complexities to increase beyond logarithmic. All previous solutions enjoying such a complexity bound for updates enforce $\Omega(n^{\epsilon})$ proof or query complexity. As an application, we provide the first construction of an authenticated Bloom filter, an update-intensive data structure that falls into our model.

We secondly observe that the repeated linearity of the used lattice-based cryptographic primitive lends itself to a natural notion of parallelism: As such, we describe *parallel* versions of our authenticated data structure algorithms, yielding the first parallel *online memory checker* with $O(1)$ query complexity using $O(\log n)$ checkers in the CREW model and without using a secret key setting, i.e., there is only need for small *reliable* but not *secret* memory. We base the security of our constructions on the difficulty of approximating the gap version of the shortest vector problem in lattices (GAPSVP) within polynomial factors.

1 Introduction

Increasing interest in online data storage and processing has recently led to the establishment of the field of *cloud computing* [17]. In such settings, verifying the validity of *remotely stored* dynamic data structures and of *queries* executed on them is an important security property, or otherwise data can be tampered with by malicious entities. In order to solve such problems efficiently, the model of *authenticated data structures* (see, e.g., [20, 33]) has been developed, which is related to memory checking [6].

A typical setting where an authenticated data structure can be employed involves three participating entities, usually referred to as *three-party* model [33]: A trusted party called *source*, owns a data structure, that is replicated along with some cryptographic information to one or more untrusted parties, called *servers*. *Clients* issue data structure queries to the servers and wish to *publicly* verify the answers received by the servers, based only on the trust they have in the source. This trust is conveyed through a time-stamped signature on a *digest* of the data structure (a collision resistant succinct representation of the data structure, e.g., the roothash of a Merkle tree), that is made available by the source. During an update, the source needs just to compute the new digest, whereas the server needs to update the authenticated data structure as a whole. Variations of this model include the *two-party* model [26], where the source keeps only a small state (i.e., the digest) and performs both the updates and the queries/verifications, as well as the *memory checking* model [6], where a memory of n cells being accessed through read/write operations is to be verified. However, the absence of the notion of *proof computation* in memory checking as well as the requirement for *public verifiability*[1] in authenticated data structures make these two models fundamentally different.

An authenticated data structure should be firstly *secure*: A computationally-bounded adversary should not be able to produce a valid proof for a false answer, under a well-accepted assumption. Secondly, it should be *efficient*: Its algorithms should have low complexity. Successfully combining both these goals comprises a challenging task, substantially depending on the underlying cryptography [8, 27, 28]. Under this premise, we develop the first *efficient* authenticated data structure based on *lattices*, a mathematical tool that has had many applications in cryptography after Ajtai's seminal result [1].

This work combines the simplicity of a Merkle tree [21] with a special property of lattice-based hash functions, which we establish and call *repeated linearity*. Roughly speaking, this property allows using the output of one invocation of the hash function, as an input to another invocation of the function, without losing "structure". This observation, in the authenticated data structures setting, turns out to be crucial in achieving *constant* update complexity, while keeping all the remaining complexity bounds *logarithmic*. This is a trade-off that, to the best of our knowledge, has not been achieved so far in the literature—and is feasible due to the use of lattices: E.g., for a table data structure of n entries, [3, 10, 27] have $O(1)$ update but $\Omega(n^{\epsilon})$ proof (or query) complexity whereas [6, 15] impose $O(\log n)$ bounds on *all* the complexity measures (see lower bound in [34]). Moreover, the repeated linearity of our lattice-based hash function lends itself to a natural notion of parallelism, allowing us to give *parallel* versions of our algorithms, yielding the first parallel online memory checker with $O(1)$ query complexity using $O(\log n)$ checkers and without a secret key setting (as opposed to [16]).

[1] Memory checking might require *secret* memory, e.g., see the PRF construction in [6].

The data structure we are considering in this paper is a *dynamic table* of size n, read and written through indices $1, \ldots, n$. We base the security of our construction on the hardness of the GAPSVP problem in lattices [22], which has its own significance given recent attacks on collision-resistant functions such as MD-5 [32]. We note that our construction requires an one-time $O(n \log n)$ preprocessing, which is is however amortized—in comparison with other works (see Table 1)— after $\Omega(n \log n)$ updates.

Authenticated data structure scheme. To formally describe our solutions and prove their properties we use an *authenticated data structure scheme*, which is a collection of algorithms $\{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$ such that (a) $\mathsf{genkey}()$ produces the secret and public key; (b) $\mathsf{setup}()$ takes as input a plain data structure D and outputs the authenticated data structure $\mathsf{auth}(D)$; (c) On input an update u, $\mathsf{update}()$ updates the authenticated data structure digest (e.g., the roothash of a Merkle tree), so that it could be used later for *query verification*. This algorithm *has* access to the secret key; (d) On input an update u, $\mathsf{refresh}()$ updates the authenticated data structure as a whole—so that it could be used later for *query execution*—, *without* having access to the secret key; (e) $\mathsf{query}()$ computes cryptographic proofs $\Pi(q)$ for answers $\alpha(q)$ to certain data structure queries q; (f) $\mathsf{verify}()$ processes the proof $\Pi(q)$ and the answer $\alpha(q)$ and either *accepts* or *rejects* the answer. Note that both $\mathsf{query}()$ and $\mathsf{verify}()$ are required to have no access to the secret key. The formal definition of all the algorithms above and the properties they should satisfy (correctness/security) are given in Definition 5. We note that both a three-party protocol [33] and a two-party protocol [26] can be realized via an authenticated data structure scheme. See Corollaries 4 and 5 respectively.

Overview of our solution. Our solution can be seen as a generalization of the Merkle tree and related hierarchical hashing constructions [6, 15, 23]. By exploiting a property of lattice-based hash functions, which we call repeated linearity, over a typical Merkle tree, we depart from black-box use of *generic collision-resistant* hash functions (e.g., MD-5 or SHA-256) in the authenticated data structures setting. As a consequence, and in the Merkle tree paradigm, the digest of a tree node v can be expressed as the "sum" of well-defined functions (called *partial digests*) applied to data stored at the leaves of v's subtree (Theorem 3). Exploiting this property enables constant update complexity as well as deriving parallel algorithms. It may also be of general interest and have other applications. A comparison of our solution with existing work is given in Table 1.

Complexity model. In this work, and as it has already been adopted in the authenticated data structures literature, we use the same complexity model as in memory checking [6, 10]: The *access* complexity[2] of an algorithm is defined as the *number of queries* that this algorithm makes to the authenticated data structure (or parts thereof) stored in an indexed memory of n cells, in order for the algorithm to complete its execution. Each memory cell can store up to $O(\mathsf{poly}(\log n))$ bits, a word size also used in memory checking literature [6, 10]. For example, a Merkle tree [21] has $O(\log n)$ update access complexity since the update algorithm needs to read and write $O(\log n)$ memory locations of the authenticated data structure, in order to execute. Similarly, the *group* complexity of an object (e.g., proof group complexity) is defined as the number of "group" elements (hash values, elements in $\mathbb{Z}_p$) contained in that object.

[2] The term "access complexity" is used here instead of "query complexity", so that to avoid ambiguity when referring to the $\mathsf{query}()$ algorithm of the authenticated data structure.

Table 1. Access and group complexities of various authenticated data structure schemes defined by algorithms $\{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$, for a dynamic table of n entries. Parameter $0 < \epsilon < 1$ is a constant, "D. Log" stands for "Discrete Logarithm", "Generic CR" stands for "Generic Collision Resistance" and GAPSVP is the gap shortest vector problem in lattices (Definition 1). In all schemes, the authenticated structure has group complexity (i.e., size) $O(n)$ and $\mathsf{genkey}()$ has $O(1)$ complexity. $\Pi(q)$ denotes the proof for a query q.

	[6, 15, 20, 23]	[3]	[25]	[8, 31]	[14]	[27]	this work
setup()	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n \log n)$
update()	$O(\log n)$	$O(1)$	$O(1)$	$O(1)$	$O(n^\epsilon)$	$O(1)$	$O(1)$
refresh()	$O(\log n)$	$O(1)$	$O(n)$	$O(n \log n)$	$O(n^\epsilon)$	$O(1)$	$O(\log n)$
query()	$O(\log n)$	$O(n)$	$O(1)$	$O(1)$	$O(n^\epsilon)$	$O(n^\epsilon)$	$O(\log n)$
verify()	$O(\log n)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(\log n)$
proof $\Pi(q)$	$O(\log n)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(\log n)$
assumption	Generic CR	D. Log	Strong DH	Strong RSA			GAPSVP

Contributions. There are two main contributions in this paper: (1) We provide the first authenticated table construction with $O(1)$ update complexity and $O(\log n)$ complexity for the remaining performance measures (Theorem 4 and Theorem 6 for the authenticated Bloom filter). To achieve that, we establish and exploit the *repeated linearity* of lattice-based hash functions, which other primitives such as *generic collision-resistant functions* (used in [6, 15, 23]) and *exponentiation functions* (used in [3, 25, 27]) lack; (2) We describe parallel versions of our authenticated data structures algorithms, yielding the first parallel online memory checker (Theorem 5) with $O(1)$ query complexity, using $O(\log n)$ checkers and *only reliable* (not secret) small memory (as opposed to [16]).

Related work. Lattice-based cryptography began with Ajtai's construction of a one-way hash function based on hard lattices problems [1]. Various generalizations and improvements have appeared since then [12, 13, 19, 22, 29]. Also, several authenticated data structures based on cryptographic hashing have been developed [6, 15, 21, 23]. Lower bounds for hash-based authenticated data structures and memory checking are given in [34] and [10, 24] respectively. Authenticated data structures using other cryptographic primitives [2, 4, 8], achieving $O(n^\epsilon)$ bounds, are presented in [14, 27]. We observe that all of the above constructions belong to one of the following two categories: either (1) they have logarithmic update complexity, with all the other complexity measures being also logarithmic, e.g., [6, 15, 23]; or (2) they have sublogarithmic update complexity (e.g., constant) but at least one of the other complexities is $\Omega(n^\epsilon)$, e.g., [3, 25, 27]. This work achieves the best of both worlds. Finally, a parallelizable authentication tree, but in the symmetric key setting, has been developed in [16]. A comparison of our construction with existing literature work (in the sequential model) can be found in Table 1.

2 Lattices and authenticated data structures

We start with some basic definitions. In the following, k denotes the security parameter. 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 $\mathbf{x} = [\mathbf{x}_1\ \mathbf{x}_2\ \ldots\ \mathbf{x}_k]^T$, $\|\mathbf{x}\|$ denotes the Euclidean norm of $\mathbf{x}$.

Lattices. Given the security parameter k, a full-rank k-dimensional lattice is defined as 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 $\mathbf{B} = \{\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_k\}$ is the *basis* of the lattice and $\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_k$ 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 useful in our context:

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

We note that, for exponential values of γ, i.e, $\gamma = 2^{O(k)}$, one can use the LLL algorithm [18] and decide the above problem in polynomial time. 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 [30], i.e., very big polynomials. Moreover, for polynomial factors, there is no proof that this problem is NP-hard[3], which makes the polynomial approximation cryptographically interesting as well. Therefore, a well-accepted assumption on which the security of our scheme is based is as follows:

Assumption 1 (Hardness of GAPSVP_γ) *Let GAPSVP_γ be an instance of the gap version of the shortest vector problem in lattices, as defined in Definition 1 and k be the security parameter. There is no polynomial-time algorithm for solving GAPSVP_γ for $\gamma = \mathsf{poly}(k)$, except with negligible probability*[4].

Reductions. After Ajtai's seminal work [1] where an one-way function based on hard lattices problem is presented, Goldreich et al. [13] presented a variation of the function, providing at the same time collision resistance. Based on this collision resistant hash function, Micciancio [22] 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 $\mathsf{SIS}_{q,m,\beta}$) *Given an integer q, a matrix $\mathbf{M} \in \mathbb{Z}_q^{k \times m}$ and a real β, find a non-zero integer vector $\mathbf{z} \in \mathbb{Z}^m \backslash \{\mathbf{0}\}$ such that $\mathbf{Mz} = \mathbf{0} \mod q$ and $\|\mathbf{z}\| \leq \beta$.*

Note that at least one solution to the above problem exists when $\beta \geq \sqrt{m} q^{k/m}$ and $m > k$ [22]. Moreover, if $q \geq 4\sqrt{m} k^{1.5} \beta$, 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 $\mathsf{SIS}'_{q,m,\beta}$) *Given an integer q, a matrix $\mathbf{M} \in \mathbb{Z}_q^{k \times m}$ and a real β, find an integer vector $\mathbf{z} \in \mathbb{Z}^m \backslash 2\mathbb{Z}^m$ such that $\mathbf{Mz} = \mathbf{0} \mod q$ and $\|\mathbf{z}\| \leq \beta$.*

For odd q, there is a polynomial-time reduction from $\mathsf{SIS}'_{q,m,\beta}$ to $\mathsf{SIS}_{q,m,\beta}$ [22]:

Lemma 1 (Reduction from $\mathsf{SIS}'_{q,m,\beta}$ to $\mathsf{SIS}_{q,m,\beta}$ [22]) *For any odd integer $q \in 2\mathbb{Z} + 1$ and SIS' instance $I = (q, \mathbf{M}, \beta)$, if I has a solution as an instance of SIS, then it has*

[3] In specific, as outlined in [30], the current state of knowledge indicates that for $\gamma > \sqrt{k/\log k}$, it is unlikely that this problem is NP-hard and no efficient algorithm is known to date.

[4] Let $f : \mathbb{N} \to \mathbb{R}$. We say that $f(k)$ is $\mathsf{neg}(k)$ iff for any nonzero polynomial $p(k)$ there exits N such that for all $k > N$ it is $f(k) < 1/p(k)$.

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 [22], by choosing certain parameters, GAPSVP_γ can be reduced to SIS' (derived by combining Lemma 5.22 and Theorem 5.23 of [22]):

Lemma 2 (Reduction from GAPSVP_γ to $\mathsf{SIS}'_{q,m,\beta}$ [22]) *Let the quantities* $\beta, m, q = k^{O(1)}$ *be polynomially-bounded, with* $q \geq 4\sqrt{m}k^{1.5}\beta$ *and* $\gamma = 14\pi\sqrt{k}\beta$. *Then there is a probabilistic polynomial-time reduction from solving* GAPSVP_γ *in the worst case to solving* $\mathsf{SIS}'_{q,m,\beta}$ *on the average with non-negligible probability.*

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

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

Theorem 1 states that if there is an algorithm that solves an average (i.e., $\mathbf{M} \in \mathbb{Z}_q^{k\times m}$ is chosen uniformly at random) instance of $\mathsf{SIS}_{q,m,\beta}$, for an odd q, $q \geq 4\sqrt{m}k^{1.5}\beta$ and $\gamma = 14\pi\sqrt{k}\beta$, then, this algorithm can be used to solve any instance of GAPSVP_γ.

Lattice-based hash function. Let $m = 2k\log q$ and $\beta = \delta\sqrt{m}$, where δ is $\mathsf{poly}(k)$. Note that $\log\delta = O(\log k)$. We also require $q \geq 4\sqrt{m}k^{1.5}\beta = 8k^{2.5}\delta\log q$. It is easy to see that given k and δ there is always a $q = O(k^{2.5}\delta\log k)$ to satisfy the above constraints—since δ is $\mathsf{poly}(k)$, the bit-size of q is $O(\log k)$. The collision resistant hash function that we are using is a generalization of the function presented in [22], 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 solution 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 [13]. This remark is very useful in our context and implies that, the larger value one picks for β, the larger the modulus q should be so that security is guaranteed (still q's bit size is $O(\log k)$).

Let now $\mathbf{M} \in \mathbb{Z}_q^{k\times m}$ be a $k \times m$ matrix that is chosen uniformly at random. We can define the function $h_{\mathbf{M}} : \mathbb{Z}^m \to \mathbb{Z}_q^k$ 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 is collision resistant based on the difficulty of $\mathsf{GAPSVP}_{14\pi\sqrt{k}\beta}$ (see proof in the Appendix):

Theorem 2 (Strong collision resistance) *Let* $m = 2k\log q$, $\beta = \delta\sqrt{m}$ *and* q *be an odd positive integer such that* $q \geq 4\sqrt{m}k^{1.5}\beta$. *Let also* $\mathbf{M} \in \mathbb{Z}_q^{k\times m}$ *be a* $k \times m$ *matrix that is chosen uniformly at random. If there is a polynomial-time 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 a polynomial-time algorithm to solve any instance of* $\mathsf{GAPSVP}_{14\pi\delta\sqrt{km}}$.

Since $\delta = \mathsf{poly}(k)$, γ is also $\mathsf{poly}(k)$ and therefore the presented hash function is secure, by Assumption 1. We can now extend the function h to accept two inputs as follows: Denote with $\mathbb{T}^{\delta,+}$ the set of all $m\times 1$ ($m = 2k\log q$) vectors such that their last $k\log q$ 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 $k\log q$ entries are zero and the remaining entries are in $\{0, 1, \ldots, \delta\}$:

Definition 4 (Lattice-based hash function with two inputs) *We define the function* $h : \mathbb{T}^{\delta,+} \times \mathbb{T}^{\delta,-} \rightarrow \mathbb{Z}_q^k$ *as* $h_{\mathbf{M},\delta}(\mathbf{x},\mathbf{y}) = \mathbf{M}(\mathbf{x}+\mathbf{y}) \mod q$*, where* $\mathbf{x},\mathbf{y} \in \{0,1,\ldots,\delta\}^m$.

Note that we use both $\mathbf{M}$ and δ as subscripts for the function. Similarly as in Theorem 2, this function is strong collision resistant, i.e., if there is a polynomial-time algorithm that finds $(\mathbf{x}_1,\mathbf{y}_1) \in (\mathbb{T}^{\delta,+} \times \mathbb{T}^{\delta,-})$ 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 there is a polynomial-time algorithm that solves 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. We now continue with a formal definition of an *authenticated data structure scheme*. Similar definitions have already appeared [28, 35]. We use the notation $\{O_1,O_2,\ldots,O_o\} \leftarrow \mathsf{alg}(I_1,I_2,\ldots,I_i)$ to denote that algorithm alg has inputs $I_1,I_2,\ldots,I_i$ and outputs $O_1,O_2,\ldots,O_o$. If an input I or an output O appears as $(I)^*$ or $(O)^*$ (e.g., algorithm update()), this means that I or O are not *required* as inputs or outputs but might appear depending on the realized scheme.

Definition 5 (Authenticated data structure scheme) *Let* D *be* any *data structure supporting queries* q *and updates* u*. We denote with* $\mathsf{auth}(D)$ *the authenticated data structure and with* d *the digest of the authenticated data structure, i.e., a constant-size description of* D*. An* authenticated data structure scheme $\mathcal{A}$ *is a collection of the following six polynomial-time algorithms* $\{\mathsf{genkey},\mathsf{setup},\mathsf{update},\mathsf{refresh},\mathsf{query},\mathsf{verify}\}$*:* ***(1)*** $\{\mathsf{sk},\mathsf{pk}\} \leftarrow \mathsf{genkey}(1^k)$*: This algorithm outputs the secret key* sk *and the public key* pk*, given the security parameter* k*;* ***(2)*** $\{\mathsf{auth}(D_0),d_0\} \leftarrow \mathsf{setup}(D_0,\mathsf{sk},\mathsf{pk})$*: This algorithm computes the authenticated data structure* $\mathsf{auth}(D_0)$ *and the respective digest of it,* d_0*, given a plain data structure* D_0*, the secret key* sk *and the public key* pk*;* ***(3)*** $\{D_{h+1},(\mathsf{auth}(D_{h+1}))^*,d_{h+1}\} \leftarrow \mathsf{update}(u,D_h,(\mathsf{auth}(D_h))^*,d_h,\mathsf{sk},\mathsf{pk})$*: This algorithm takes as input an update* u*, a data structure* D_h*,* possibly *an authenticated data structure* $\mathsf{auth}(D_h)$*, the digest* d_h*, and both the secret and the public keys. It outputs the data structure* D_{h+1}*,* possibly *the authenticated data sturcture* $\mathsf{auth}(D_{h+1})$ *and the digest* d_{h+1}*;*[5] ***(4)*** $\{D_{h+1},\mathsf{auth}(D_{h+1}),d_{h+1}\} \leftarrow \mathsf{refresh}(u,D_h,\mathsf{auth}(D_h),d_h,\mathsf{pk})$*: This algorithm takes as input an update* u*, a data structure* D_h*, an authenticated data structure* $\mathsf{auth}(D_h)$*, the digest* d_h *and only the public key. It outputs* D_{h+1}*, the authenticated data structure* $\mathsf{auth}(D_{h+1})$ *and the digest* d_{h+1}*;*[6] ***(5)*** $\{\Pi(q),\alpha(q)\} \leftarrow \mathsf{query}(q,D_h,\mathsf{auth}(D_h),\mathsf{pk})$*: On input a query* q*, a data structure* D_h*, an authenticated data structure* $\mathsf{auth}(D_h)$ *and the public key* pk*, this algorithm returns the answer to the query* $\alpha(q)$*, along with a respective proof* $\Pi(q)$*;* ***(6)*** $\{\mathsf{accept},\mathsf{reject}\} \leftarrow \mathsf{verify}(q,\alpha(q),\Pi(q),d_h,\mathsf{pk})$*: This algorithm takes as input a query* q*, an answer* $\alpha(q)$*, a proof* $\Pi(q)$*, a digest* d_h *and the public key* pk *and outputs either "*accept*" or "*reject*".*

There are two properties that an authenticated data structure scheme should satisfy, i.e., *correctness* and *security* (intuition follows from signature schemes definitions—see Definitions 1, 2 in the Appendix). Roughly speaking, the correctness property requires that if a proof $\Pi(q)$ for an answer $\alpha(q)$ is computed by algorithm query() (i.e., faith-

[5] Note that this algorithm is only required to output d_{h+1} and the D_{h+1}. Outputting the new authenticated data structure $\mathsf{auth}(D_{h+1})$ is not a requirement—this will be important in improving the complexity of this algorithm. Also, the secret key is required for execution.

[6] Note there that the secret key is not required for execution.

fully), then verify(), on input $\Pi(q)$, will only accept a *correct* answer $\alpha(q)$ corresponding to queries q on an authenticated data structure $\mathsf{auth}(D)$ that is updated through algorithm refresh(); the security property requires that a computationally-bounded adversary should not be able (except with negligible probability) to produce verifying proofs Π for *incorrect* answers α corresponding to queries q on an authenticated data structure $\mathsf{auth}(D)$ whose digest (i.e., signature) is updated through adversarially chosen *oracle* calls to algorithm update()—this is why update() has access to the secret key.

3 Main construction

In this section we present our update-optimal authenticated data structure scheme for a dynamic table, i.e., the scheme $\mathcal{LBT} = \{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$.

Data structure. We recall that the data structure for which we describe an authenticated data structure scheme is a table A that consists of n indices $1, 2, \ldots, n$. In each index i we can store a value from $\mathbb{Z}_q^k$. 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. Our table A supports two operations, naturally defined for the dynamic table: (a) Queries q: Given an index i, return the value $A[i]$; (b) Updates u: Given an index i and value y, set $A[i] = y$.

A direct solution for this problem would be to use a Merkle tree with some collision-resistant hash function (e.g., SHA-2)—see first column of Table 1—, which would bear logarithmic complexities in all the complexity measures—which also inherently enforces *sequential computations*. Here we build an authenticated structure for this data structure that uses the lattice-based hash function introduced in Section 2 and also supports constant complexity updates, allowing at the same time a great deal of *parallelism*.

Algebraic tools. We now discuss some algebraic tools to be used in our construction. Without loss of generality, assume that q, the modulus is a power of two:

Definition 6 (Binary representation) *Define* $f(x) = [\mathbf{f}_0\, \mathbf{f}_1\, \ldots\, \mathbf{f}_{\log q-1}]^T \in \{0,1\}^{\log q}$ *to be the binary representation of* $x \in \mathbb{Z}_q$. *Namely,* $x = \sum_{i=0}^{\log q-1} \mathbf{f}_i 2^i \mod q$.

Definition 7 (Radix-2 representation) *Define* $g(x) = [\mathbf{f}_0\, \mathbf{f}_1\, \ldots\, \mathbf{f}_{\log q-1}]^T \in \mathbb{Z}_q^{\log q}$ *to be* some *radix-2 representation of* $x \in \mathbb{Z}_q$. *Namely,* $x = \sum_{i=0}^{\log q-1} \mathbf{f}_i 2^i \mod q$.

By "some" radix-2 representation we mean that the function $g : \mathbb{Z}_q \to \mathbb{Z}_q^{\log q}$ is "one-to-many". For example, for $q = 16$, $x = 7$, possible values for $g(x)$ can be $[0\ 1\ 1\ 1]^T$ (the usual binary representation), $[0\ -2\ 0\ -1]^T$ or $[-2\ 2\ 0\ -1]^T$ (and many more). We now give an important result for our construction:

Lemma 3 *For any* $x_1, x_2, \ldots, x_t \in \mathbb{Z}_q$ *there exist a radix-2 representation* $g(.)$ *such that* $g(x_1 + x_2 + \ldots + x_t \mod q) = f(x_1) + f(x_2) + \ldots + f(x_t) \mod q$. *Moreover it is* $g(x_1 + x_2 + \ldots + x_t \mod q) \in \{0, \ldots, t\}^{\log q}$.

Lemma 3 is useful in the following sense: Given two binary representations of x_1 and x_2, namely f_1 and f_2, a radix-2 representation of $x_1 + x_2$ is $f_1 + f_2$. Definitions 6 and 7 and also Lemma 3 (see Corollary 1) can be naturally extended for vectors $\mathbf{x} \in \mathbb{Z}_q^k$: For $i = 1, \ldots, k$, $\mathbf{x}_i$ is mapped to the respective $\log q$ entries $f(\mathbf{x}_i)$ (or $g(\mathbf{x}_i)$) in the resulting vector $f(\mathbf{x})$ (or $g(\mathbf{x})$). Therefore we have the following:

Corollary 1 *For any* $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_t \in \mathbb{Z}_q^k$ *there exist a radix-2 representation* $g(.)$ *such that* $g(\mathbf{x}_1 + \mathbf{x}_2 + \ldots + \mathbf{x}_t \mod q) = f(\mathbf{x}_1) + f(\mathbf{x}_2) + \ldots + f(\mathbf{x}_t) \mod q$. *Moreover it is* $g(\mathbf{x}_1 + \mathbf{x}_2 + \ldots + \mathbf{x}_t \mod q) \in \{0, \ldots, t\}^{k \log q}$.

To constrain the inputs to our hash function, we need the following definition:

Definition 8 *Let $x \in \mathbb{Z}_q^k$. We say that the radix-2 representation $g(x) \in \mathbb{Z}_q^{k \log q}$ is* δ-admissible *if and only if $g(x) \in \{0, 1, \ldots, \delta\}^{k \log q}$.*

Algorithms of the scheme. We now describe the algorithms of the scheme $\mathcal{LBT}$ (see Definition 5). All expressions below are reduced modulo q, i.e., we work in $\mathbb{Z}_q$:

Algorithm $\{\mathsf{sk}, \mathsf{pk}\} \leftarrow \mathsf{genkey}(1^k)$: On input the security parameter k, this algorithm computes an odd number $q = O(k^{2.5}\delta \log k)$, for some $\delta = n = \mathsf{poly}(k)$. Namely we set δ to be equal to the size of the table, n. Then it samples $\mathbf{M} \in \mathbb{Z}_q^{k \times m}$ uniformly at random, where $m = 2k \log q$. It sets $\mathsf{sk} = \emptyset$ and $\mathsf{pk} = \{\mathbf{M}, q\}$, i.e., there is no secret (trapdoor information) in our scheme. The access complexity of this algorithm is $O(1)$.

Lattice-based digests. Before we describe algorithm $\mathsf{setup}()$, we describe how we define the lattice-based digests on the table A, by using the hash function of Definition 4. Let D_0 be the initial state of our table, storing values $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n \in \mathbb{Z}_q^k$. Let T be the binary tree of ℓ levels on top of the values $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$—recall we have assumed that $n = 2^\ell$, and r be the root of tree T. By convention, the root of the tree lies at level 0 and the leaves of the tree lie at level ℓ. For every leaf node v_i of the tree, $i = 1, \ldots, n$, the digest $d(v_i)$ is defined as $d(v_i) = \mathbf{x}_i$. Then, for any internal node u, with left child v and right child w, by using the hash function $h_{\mathbf{M},n}(\mathbf{x}, \mathbf{y})$ given in Definition 4 in a recursive way, the digest $d(u)$ of node u can be defined as

$$d(u) = h_{\mathbf{M},n}(\mathbf{U}g(d(v)), \mathbf{D}g(d(w))) = \mathbf{M}\left[\mathbf{U}g(d(v)) + \mathbf{D}g(d(w))\right], \tag{1}$$

where $g(d(v))$ and $g(d(w))$ are *some* n-admissible radix-2 representations of $d(v)$ and $d(w)$, i.e., by Definition 4, it must be $g(d(v)), g(d(w)) \in \{0, 1, \ldots, n\}^{k \log q}$.

In the above relations, matrices $\mathbf{U}$ and $\mathbf{D}$ are special matrices such that multiplying matrices $\mathbf{U}$ and $\mathbf{D}$ with a vector in $\{0, 1, \ldots, n\}^{k \log q}$ doubles the dimension of the vector by shifting its entries accordingly and by filling the vacant entries with zeros. This operation is used to prepare the vectors in the appropriate input format for the hash function. More formally, $\mathbf{U} = [\mathbf{I}_{k \log q}\ \mathbf{O}_{k \log q}]^T$ and $\mathbf{D} = [\mathbf{O}_{k \log q}\ \mathbf{I}_{k \log q}]^T$, where $\mathbf{I}_l$ denotes the square identity matrix of dimension l and $\mathbf{O}_l$ denotes the square zero matrix of dimension l. Indeed, it easy to see that for all $\mathbf{x} \in \{0, 1, \ldots, n\}^{k \log q}$ it is $\mathbf{Ux} \in \mathbb{T}^{n,+}$ and $\mathbf{Dx} \in \mathbb{T}^{n,-}$, where $\mathbb{T}^{n,+}$ and $\mathbb{T}^{n,+}$ are defined in Section 2.

The computation in Relation 1 is as follows (see Appendix, Figure 1): Suppose a node $u \in T$ has children v and w of digests $d(v), d(w) \in \mathbb{Z}_q^k$. Applying $g(.)$ *transforms* $d(v), d(w)$ into vectors of $k \log q$ small entries (*admissible* radix-2 representations). Multiplying with $\mathbf{U}$ and $\mathbf{D}$ *prepares* $g(d(v)), g(d(w))$ to be input to the hash function. [7]

Partial digests. Here we show how to express the digest $d(u)$ (computed in Relation 1) for every node $u \in T$ somehow differently, which is crucial for deriving our final results. To simplify some notation, we set $\mathbf{MU} = \mathbf{L}$ and $\mathbf{MD} = \mathbf{R}$ (stand for *left*/*right*)—note that $\mathbf{L}, \mathbf{R} \in \mathbb{Z}_q^{k \times k \log q}$. Let also $\mathsf{range}(u)$ be the range of successive indices corresponding to the leaves of the subtree of T rooted on u. E.g., in Figure 1 in the Appendix, it is $\mathsf{range}(r_{11}) = \{1, 2, 3, 4\}$. For every node $u \in T$ and for every $i \in \mathsf{range}(u)$ we define the *partial digest* of u with reference to $\mathbf{x}_i$:

[7] The procedure so far is the same with a Merkle tree construction that uses a collision-resistant function such as SHA-2, i.e., recursive computation over the nodes of a tree.

Definition 9 (Partial digest of a node u) *For a leaf node $u \in T$ storing value $\mathbf{x}_i$, the* partial digest *of u with reference to $\mathbf{x}_i$ is defined as $d(u, \mathbf{x}_i) = \mathbf{x}_i$. Else, for every other node u of T, with* left *child v and* right *child w, and for every $i \in \mathsf{range}(u)$, the* partial digest *$d(u, \mathbf{x}_i)$ of u with reference to $\mathbf{x}_i$ is recursively defined as $d(u, \mathbf{x}_i) = \mathbf{L}f(d(v, \mathbf{x}_i))$, if $\mathbf{x}_i$ belongs to the left subtree of u; Else, $d(u, \mathbf{x}_i) = \mathbf{R}f(d(w, \mathbf{x}_i))$.*

E.g., in Figure 1 of the Appendix, the partial digests of root r with reference to $\mathbf{x}_2$ and $\mathbf{x}_3$ are $d(r, \mathbf{x}_2) = \mathbf{R}f(\mathbf{R}f(\mathbf{L}f(\mathbf{x}_2)))$ and $d(r, \mathbf{x}_3) = \mathbf{R}f(\mathbf{L}f(\mathbf{R}f(\mathbf{x}_3)))$ respectively ($f(z)$ is z's binary representation). We now give the main result of this section.

Theorem 3 *The digest $d(u)$ of node $u \in T$ in Relation 1 can be expressed as $d(u) = \sum_{i \in \mathsf{range}(u)} d(u, \mathbf{x}_i)$, where $d(u, \mathbf{x}_i)$ is the partial digest of node u with reference to $\mathbf{x}_i$.*

Proof. (Sketch) Apply Corollary 1 repeatedly. By induction, the digest can be expressed as in Relation 1 (full proof in the Appendix). □

Algorithm $\{\mathsf{auth}(D_0), d_0\} \leftarrow \mathsf{setup}(D_0, \mathsf{sk}, \mathsf{pk})$: Let D_0 be the initial table, storing values $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n \in \mathbb{Z}_q^k$. The algorithm computes the digests of the nodes: It sets $d(u) = \mathbf{x}_i$ for all leaf nodes u storing value $\mathbf{x}_i$ and $d(u) = \mathbf{M}\left[\mathbf{U}g(d(v)) + \mathbf{D}g(d(w))\right]$ (application of the hash function in Definition 4) for all internal nodes u with left child v and right child w, where $g(d(v))$ and $g(d(w))$, i.e., the radix-2 representations of the children digests, are computed according to the following definition[8]:

Definition 10 *The radix-2 representation of $d(u)$ of node $u \in T$ is computed as the sum of $|\mathsf{range}(u)|$ binary representations, i.e., $g(d(u)) = \sum_{i \in \mathsf{range}(u)} f(d(u, \mathbf{x}_i))$, where $d(u, \mathbf{x}_i)$ is the partial digest of node u with reference to $\mathbf{x}_i$.*

By combining Theorem 3 and Definition 10, by Corollary 1, we have:

Corollary 2 *Let u be an internal node of tree T. The $g(.)$ representation of $d(u)$ defined in Definition 10 is an n-admissible radix-2 representation of $d(u)$.*

This concludes the description of $\mathsf{setup}()$. The algorithm outputs $d_0 = d(r)$, where r is the root of T (i.e., the digest of the data structure is the digest of the root of the tree) and also it outputs $\mathsf{auth}(D_0)$ to be a structure that contains: (a) Tree T; (b) $g(d(u))$ for all nodes u of T as computed in Definition 10. The complexity of the algorithm is $O(n \log n)$, since the computation of $g(d(u))$ involves a linear number of operations per tree level, and there are $O(\log n)$ levels in total (full proof in the Appendix).

We continue by noting that Theorem 3 allows us to express $d(r)$ as a sum of well-defined functions of the leaves, namely the *partial digests* of the root r with reference to values in the table. This allows us to achieve our desired complexity bounds:

Corollary 3 *Let $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$ be the values stored in our table. Then the digest $d(r)$ of the root r of the tree T can be expressed as $d(r) = \sum_{i=1}^{n} d(r, \mathbf{x}_i)$, where $d(r, \mathbf{x}_i)$ is the partial digest of the root r with reference to $\mathbf{x}_i$.*

We observe that computing the partial digest $d(r, \mathbf{x}_i)$ requires *one* query to the authenticated data structure, i.e., a query for value $\mathbf{x}_i$, therefore yielding $O(1)$ access complexity. Matrices $\mathbf{L}$ and $\mathbf{R}$, both used for its computation (Definition 9) are *not* part of the authenticated data structure (they are fixed by $\mathsf{setup}()$ as public information) and accessing them any number of times does not add to the access complexity. We continue with describing the remaining algorithms of our authenticated data structure scheme:

[8] Note here that the *binary* representations $f(d(v)), f(d(w))$ could be used instead; However, in lieu of achieving our efficiency goals, the algorithm uses Definition 10.

Algorithm $\{d_{h+1}, D_{h+1}\} \leftarrow \mathsf{update}(u, D_h, d_h, \mathsf{sk}, \mathsf{pk})$: Let the update u be "set $A[i] = \mathbf{x}_i'$" and let the value of $A[i]$ before the update be $\mathbf{x}_i$. Then the algorithm sets $d_{h+1} = d_h - d(r, \mathbf{x}_i) + d(r, \mathbf{x}_i')$, where $d(r, \mathbf{x}_i)$ and $d(r, \mathbf{x}_i')$ are the *partial digests* of r with reference to $\mathbf{x}_i$ and $\mathbf{x}_i'$, defined in Definition 9. Due to Corollary 3, d_{h+1} is the correct updated digest. Since the computation of partial digests has constant access complexity, algorithm $\mathsf{update}()$ has $O(1)$ access complexity, since it involves two operations in $\mathbb{Z}_q^k$. The algorithm outputs d_{h+1} as well as the updated table D_{h+1}:

Lemma 4 *Algorithm* $\mathsf{update}()$ *has* $O(1)$ *access complexity.*

Algorithm $\{D_{h+1}, \mathsf{auth}(D_{h+1}), d_{h+1}\} \leftarrow \mathsf{refresh}(u, D_h, \mathsf{auth}(D_h), d_h, \mathsf{pk})$: This algorithm updates the authenticated $\mathsf{auth}(D_h)$. Let the update u be "set $A[i] = \mathbf{x}_i'$" and let the value of $A[i]$ before the update be $\mathbf{x}_i$. Suppose $v_\ell, v_{\ell-1}, \ldots, v_1$ is the path from the node of index i to the child v_1 of the root of the tree. The algorithm should update the values $g(d(v_j))$ for $j = \ell, \ell-1, \ldots, 1$. This is achieved via Definition 10, by setting $g(d'(v_j)) = g(d(v_j)) - f(d(v_j, \mathbf{x}_i)) + f(d(v_j, \mathbf{x}_i'))$, (the invariant of Definition 10 must be maintained) for $j = \ell, \ell-1, \ldots, 1$ and where $d(v_j, \mathbf{x}_i)$, $d(v_j, \mathbf{x}_i')$ are the partial digests of node v_j with reference to $\mathbf{x}_i$ and $\mathbf{x}_i'$. The algorithm outputs D_{h+1}, $\mathsf{auth}(D_{h+1})$ (i.e., the $g(d'(.))$ representations) and d_{h+1} as in $\mathsf{update}()$.

Lemma 5 *Algorithm* $\mathsf{refresh}()$ *has* $O(\log n)$ *access complexity. Moreover, it is parallelizable with* $O(1)$ *access complexity using* $O(\log n)$ *processors in the CREW model.*

Proof. (Sketch) Note that the binary representations $f(d(v_j, \mathbf{x}_i))$, $f(d(v_j, \mathbf{x}_i'))$ used in the update relations can be computed in $O(1)$ access complexity, since they are functions of only $\mathbf{x}_i$ and $\mathbf{x}_i'$ respectively. Since $\ell = O(\log n)$ the result follows. For the parallel implementation, note that for each $j = \ell, \ell-1, \ldots, 1$, the updates at each node v_j are *independent* from one another[9]. Therefore, by allowing concurrent read (values $\mathbf{x}_i$ and $\mathbf{x}_i'$ must be read concurrently by $O(\log n)$ processors) and exclusive write (each processor writes the outputs on a separate tree node), the result follows. □

Algorithm $\{\Pi(q), \alpha(q)\} \leftarrow \mathsf{query}(q, D_h, \mathsf{auth}(D_h), \mathsf{pk})$: Let the query q be "return the value stored at index i". Suppose $v_\ell, v_{\ell-1}, \ldots, v_1$ is the path from the node of index i to the child v_1 of the root of the tree. The algorithm sets $\alpha(q) = A[i]$ and sets the proof $\Pi(q)$ to be the array π of $g(.)$ representations such that $\pi_i = (g(d(v_i)), g(d(\mathsf{sib}(v_i))))$, for $i = \ell, \ell-1, \ldots, 1$, where $\mathsf{sib}(u)$ denotes the sibling of a node u in tree T. Since $\ell = O(\log n)$, the access complexity of the algorithm is $O(\log n)$. Also it is parallelizable in the EREW model with $O(1)$ access complexity and $O(\log n)$ processors.

Algorithm $\{\mathsf{accept}, \mathsf{reject}\} \leftarrow \mathsf{verify}(q, \alpha(q), \Pi(q), d_h, \mathsf{pk})$: Let the query q be "return the value at index i", $y = \alpha(q)$, and $\Pi(q) = \pi$ such that $\pi_j = (\alpha_j, \beta_j)$ ($j = \ell, \ell - 1, \ldots, 1$). For $j = \ell, \ell-1, \ldots, 1$ the algorithm performs the following:

1. If $g(y) \neq \alpha_j$ or α_j, β_j are not n-admissible $g(.)$ representations, output "reject";
2. Set $y = \mathbf{M}(\mathbf{U}\alpha_j + \mathbf{D}\beta_j)$ if v_j is v_{j-1}'s left child, or $y = \mathbf{M}(\mathbf{D}\alpha_j + \mathbf{U}\beta_j)$ otherwise.

After the loop terminates, if $y \neq d_h$, "reject" is output, else, "accept" is output. Our final result is as follows (note also Theorem 6 for the Bloom filter in the Appendix):

Theorem 4 *Let* k *be the security parameter. Then there exists an authenticated data structure scheme* $\mathcal{LBT} = \{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$ *for a dynamic*

[9] This is not the case for a typical Merkle tree update: In order to compute the hash at node v_j, the hash of node v_{j+1} is required, making such a procedure inherently sequential.

table D of n entries such that: (1) It is correct and secure according to Definitions 1, 2 respectively and assuming the hardness of GAPSVP_γ *for* $\gamma = O(nk\sqrt{\log n + \log k})$*; (2) The access complexity of (i)* $\mathsf{setup}()$ *is* $O(n \log n)$ *or* $O(n)$ *using* $O(\log n)$ *processors in the CREW model; (ii)* $\mathsf{update}()$ *is* $O(1)$*; (iii)* $\mathsf{refresh}()$ *is* $O(\log n)$ *or* $O(1)$ *using* $O(\log n)$ *processors in the CREW model; (iv)* $\mathsf{query}()$ *is* $O(\log n)$ *or* $O(1)$ *using* $O(\log n)$ *processors in the EREW model; (v)* $\mathsf{verify}()$ *is* $O(\log n)$ *or* $O(1)$ *using* $O(\log n)$ *processors in the ERCW model; (3) The group complexity of (i) the proof* $\Pi(q)$ *for a query* q *is* $O(\log n)$*; (ii) the authenticated data structure* $\mathsf{auth}(D)$ *is* $O(n)$.

A note on repeated linearity. We note here that the fact that the used hash function is additive, i.e., it is $\mathbf{Mx} + \mathbf{My} = \mathbf{M}(\mathbf{x} + \mathbf{y})$, is not enough for deriving our results. This is the reason that other *homomorphic* collision-resistant hash functions (e.g., exponentiation with secret factorization) could not be employed instead. The crucial property we can exhibit here, which is what we call *repeated linearity*, is a means of "feeding" the output of the function again as an input, so that certain homomorphic properties are still satisfied—and in specific the properties of Corollary 1. Therefore, it might be the case that other functions could be also used instead, would they satisfy such a property.

4 Parallel online memory checking

In this section, we establish our results concerning *parallel* online memory checking[10]. The online memory checking model [6] can be (informally) described as follows: Suppose $\mathcal{M}$ is an *unreliable* (malicious) memory of $O(n)$ cells. A user $\mathcal{U}$ wants to read (through operation $\mathsf{read}(i)$) or write (through operation $\mathsf{write}(i, x)$, where x is the new content) a cell $i \in \{1, 2, \ldots, n\}$. However, his requests go through a checker $\mathcal{C}$. The checker is supposed to read cells from the unreliable memory $\mathcal{C}$ and also some reliable (and possibly secret) information s of *sublinear* size and output either the correct answer (i.e., the latest content of cell i) or **BUGGY**, if the content of cell i is corrupted. The probability of returning the corrupted content of a cell as correct should be negligible. The checker is called *non-adaptive*, if, given an index i, the set and the order of the cells accessed in order to output the answer is deterministic. In this paper we are considering such checkers. For the formal definition, see Definition 11 in the Appendix.

In online memory checking settings, the complexity measure we are interested in minimizing is the *query* complexity, which is defined as the sum of the *number of requests* that the checker makes to the *unreliable* memory $\mathcal{M}$ during a $\mathsf{read}(i)$ operation plus the *number of requests* that the checker makes to the *unreliable* memory $\mathcal{M}$ during a $\mathsf{write}(i, x)$ operation [24]. So far in the literature, and in the computational model, checkers with $O(\log n)$ [6] or $O(\log_d n)$ [10] query complexity have appeared. Specifically for these checkers, we can distinguish two cases: **(a)** In the secret key setting, i.e., when there is requirement for *both* reliable and secret small memory s, these checkers have been shown to be parallelizable, e.g., [16], as well as the construction based on PRFs [6]—although this has not been reported in the literature[11]; **(b)** In the non-secret key setting, i.e., when there is requirement for *only* reliable memory (e.g., the

[10] Our $\mathcal{LBT}$ scheme also yields a sequential memory checker of $O(\log n)$ query complexity.

[11] The construction based on PRFs appearing in [6] is easily parallelizable since the PRF tag computed on each node of the tree is not a function of the PRF tags of its children.

construction using UOWHFs from [6] and Merkle tree constructions), these checkers have appeared to be *inherently* sequential. However, here we establish the first parallel online memory checker in the non-secret key setting (full proof in the Appendix):

Theorem 5 *In the non-secret key setting and in the CREW model of parallel computation, there is a non-adaptive online memory checker for an unreliable memory of n cells with $O(1)$ query complexity, using $O(\log n)$ checkers and $O(1)$ reliable memory.*

Proof: (Sketch) Use the authenticated data structure scheme $\mathcal{LBT}$ of Theorem 4. The checker will use algorithm query() to read a location i and algorithm refresh() to write a location i. The reliable memory is the lattice digest $d(r)$ of the root r of the tree. Since both query() and refresh() are parallelizable, the result follows. □

Protocols. As we mentioned in the introduction, an authenticated data structure scheme $\mathcal{A}$ may be used by a three-party protocol [33]: A trusted entity, called *source*, owns a data structure D_h, but desires to outsource query answering, in a trustworthy (verifiable) way. The source runs genkey() and setup(), outputs the authenticated data structure auth(D_h) along with the digest d_h. The source subsequently signs the digest d_h, and it outsources auth(D_h), D_h, the digest d_h and its signature (which is forwarded to the *clients* during verification) to some untrusted entities, called *servers*. On input a data structure query q sent by the clients, the servers use auth(D_h) and D_h to compute proofs $\Pi(q)$, by running algorithm query(). Clients can verify these proofs $\Pi(q)$ by running algorithm verify(), and since they have access to the signature of d_h. When there is an update in the data structure—issued by the source—, the source uses algorithm update() to produce the new digest d'_h to be used for the next verification, while the servers update the authenticated data structure through refresh(). Therefore:

Corollary 4 (Three-party model) *Let k be the security parameter. There exists a three-party authenticated data structures protocol involving a trusted source, an untrusted server and a client for verifying queries on a table of n entries such that: (a) The setup at the source has $O(n \log n)$ access complexity or $O(n)$ using $O(\log n)$ processors in the CREW model; (b) The update at the source has $O(1)$ access complexity; (c) The space needed at the source has $O(n)$ group complexity; (d) The communication between the source and the server has $O(1)$ group complexity; (e) The update at the server has $O(\log n)$ access complexity or $O(1)$ using $O(\log n)$ processors in the CREW model; (f) The query at the server has $O(\log n)$ access complexity or $O(1)$ using $O(\log n)$ processors in the EREW model; (g) The space needed at the server has $O(n)$ group complexity; (h) The proof has $O(\log n)$ group complexity; (i) The verification at the client has $O(\log n)$ access complexity or $O(1)$ using $O(\log n)$ processors in the CRCW model; (j) For a query q sent by the client to the server at any time (even after updates), let α be an answer and let π be a proof returned by the server. With probability $\Omega(1 - \mathsf{neg}(k))$, the client accepts the answer α if and only if α is correct.*

We note here that an authenticated data structure scheme $\mathcal{A}$ can also be used by a two-party protocol [26] (Corollary 5 in the Appendix), yielding similar complexity measures: In this case, the source issues both the updates and the queries but is required to keep only constant state, i.e., the digest. Also the source, before updating, engages in a protocol with the server (that may involve calls to algorithms query() and verify()) that will allow him to verify the portion of the data structure he is updating, and use it in algorithm update() as input, which will output the new digest (e.g., see [26]).

Bibliography

[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 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.

[4] 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.

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

[6] M. Blum, W. S. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. *Algorithmica*, 12(2/3):225–244, 1994.

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

[8] J. Camenisch and A. Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In *Proc. CRYPTO*, pages 61–76, 2002.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 2nd edition, 2001.

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

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] E. Hall and C. S. Julta. Parallelizable authentication trees. In *Proc. Seclected Areas in Cryptography (SAC)*, pages 95–109, 2005.
[17] B. Hayes. Cloud computing. *Commun. ACM*, 51(7):9–11, 2008.
[18] A. K. Lenstra, H. W. L. Jr, and L. Lovasz. Factoring polynomials with rational coefficients. *Math.Ann.*, (261):515–534, 1982.
[19] V. Lyubashevsky and D. Micciancio. Generalized compact knapsacks are collision resistant. In *ICALP (2)*, pages 144–155, 2006.
[20] 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.
[21] R. C. Merkle. A certified digital signature. In G. Brassard, editor, *Proc. CRYPTO '89*, volume 435 of *LNCS*, pages 218–238. Springer-Verlag, 1989.
[22] D. Micciancio and O. Regev. Worst-case to average-case reductions based on gaussian measures. *SIAM J. Comput.*, 37(1):267–302, 2007.
[23] M. Naor and K. Nissim. Certificate revocation and certificate update. In *Proc. 7th USENIX Security Symposium*, pages 217–228, Berkeley, 1998.
[24] M. Naor and G. N. Rothblum. The complexity of online memory checking. *J. ACM*, 56(1), 2009.
[25] L. Nguyen. Accumulators from bilinear pairings and applications. In *Proc. RSA Conference, Cryptographers' track (CT-RSA) , LNCS 3376, pp. 275-292, Springer.*, 2005.
[26] C. Papamanthou and R. Tamassia. Time and space efficient algorithms for two-party authenticated data structures. In *Proc. Int. Conference on Information and Communications Security (ICICS)*, volume 4861 of *LNCS*, pages 1–15. Springer, 2007.
[27] 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.
[28] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Optimal authenticated data structures with multilinear forms. In *Proc. Int. Conference on Pairing-Based Cryptography (PAIRING)*, pages 246–264, 2010.
[29] C. Peikert and A. Rosen. Efficient collision-resistant hashing from worst-case assumptions on cyclic lattices. *Electronic Colloquium on Computational Complexity (ECCC)*, (158), 2005.
[30] O. Regev. On the complexity of lattice problems with polynomial approximation factors. *The LLL algorithm*, pages 475–496, 2010.
[31] 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.
[32] M. Stevens, A. Sotirov, J. Appelbaum, A. Lenstra, D. Molnar, D. A. Osvik, and B. Weger. Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate. In *CRYPTO '09: Proceedings of the 29th Annual International Cryptology Conference on Advances in Cryptology*, pages 55–69, Berlin, Heidelberg, 2009. Springer-Verlag.
[33] R. Tamassia. Authenticated data structures. In *Proc. European Symp. on Algorithms*, volume 2832 of *LNCS*, pages 2–5. Springer-Verlag, 2003.

[34] 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.

[35] R. Tamassia and N. Triandopoulos. Certification and authentication of data structures. In *Proc. Alberto Mendelzon Workshop on Foundations of Data Management*, 2010.

5 Appendix

5.1 Proof of Theorem 2

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 $\mathbf{Mx} = \mathbf{My} \mod q$. 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 $\mathsf{SIS}_{q,m,\beta}$ (note that matrix $\mathbf{M}$ 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. □

5.2 Correctness and security definitions

Definition 1 (Correctness of authenticated data structure scheme). *Let $\mathcal{A}$ be an authenticated data structure scheme defined by the collection of algorithms* $\{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$. *We say that the authenticated data structure scheme $\mathcal{A}$ is correct if, for all $k \in \mathbb{N}$, for all* $\{\mathsf{sk}, \mathsf{pk}\}$ *output by algorithm* $\mathsf{genkey}()$, *for all $D_h, \mathsf{auth}(D_h), d_h$ output by one invocation of* $\mathsf{setup}()$ *followed by polynomially-many invocations of* $\mathsf{refresh}()$, *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, whenever* $\mathsf{check}(q, \alpha(q), D_h)$ *accepts, so does* $\mathsf{verify}(q, \Pi(q), \alpha(q), d_h, \mathsf{pk})$.

Definition 2 (Security of authenticated data structure scheme). *Let $\mathcal{A}$ be an authenticated data structure scheme defined by the collection of algorithms* $\{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$, *$k$ be the security parameter and* $\{\mathsf{sk}, \mathsf{pk}\} \leftarrow \mathsf{genkey}(1^k)$. *Let also* Adv *be a polynomially-bounded adversary that is only given* pk. *The adversary has unlimited access to all algorithms of $\mathcal{A}$, except for algorithms* $\mathsf{setup}()$, $\mathsf{update}()$ *and* possibly *algorithm* $\mathsf{verify}()$, *to which he has only oracle access. The adversary picks an initial state of the data structure D_0 and computes $D_0, \mathsf{auth}(D_0), d_0$ through oracle access to algorithm* $\mathsf{setup}()$. *Then, for $i = 0, \ldots, h = \mathsf{poly}(k)$,* Adv *issues an update u_i in the data structure D_i and outputs D_{i+1} and d_{i+1} through oracle access to algorithm* $\mathsf{update}()$. *Finally the adversary enters the attack stage where he picks an index $0 \leq t \leq h+1$, a query Q, an answer $\alpha(Q)$ and a proof $\Pi(Q)$. We say that the authenticated data structure scheme $\mathcal{A}$ is secure if for all $k \in \mathbb{N}$, for all* $\{\mathsf{sk}, \mathsf{pk}\}$ *output by algorithm* $\mathsf{genkey}()$, *and for all polynomially-bounded adversaries* Adv *the probability*

$$\Pr\left[\begin{array}{l}\{Q, \Pi(Q), \alpha(Q), t\} \leftarrow \mathsf{Adv}(1^k, \mathsf{pk});\ \mathsf{accept} \leftarrow \mathsf{verify}(Q, \alpha(Q), \Pi(Q), d_t, \mathsf{pk});\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{reject} = \mathsf{check}(Q, \alpha(Q), D_t).\end{array}\right]$$

is $\mathsf{neg}(k)$.

5.3 Proof of Lemma 3

Let $\mathbf{x}_i = f(x_i)$ be the binary representation of x_i for $i = 1, \ldots, t$. Then

$$\sum_{i=1}^{t} \mathbf{x}_i = \left[\sum_{i=1}^{t} \mathbf{x}_{i0} \ \sum_{i=1}^{t} \mathbf{x}_{i1} \ldots \sum_{i=1}^{t} \mathbf{x}_{i(k-1)}\right]^T \mod q\,.$$

The resulting vector is a radix-2 representation of

$$\left(\sum_{i=1}^{t} \mathbf{x}_{i0}\right) \times 2^0 + \left(\sum_{i=1}^{t} \mathbf{x}_{i1}\right) \times 2^1 + \ldots + \left(\sum_{i=1}^{t} \mathbf{x}_{i(k-1)}\right) \times 2^{k-1} \mod q\,,$$

which can be written as

$$\sum_{j=0}^{k-1} \mathbf{x}_{1j} \times 2^j + \sum_{j=0}^{k-1} \mathbf{x}_{2j} \times 2^j + \ldots + \sum_{j=0}^{k-1} \mathbf{x}_{tj} \times 2^j = x_1 + x_2 + \ldots + x_t \mod q.$$

Therefore there exists a radix-2 representation g such that $g(x_1+x_2+\ldots+x_t \mod q) = f(x_1)+f(x_2)+\ldots+f(x_t) \mod q$. Finally note that since $g(.)$ is the sum of t binary representations, it cannot contain a entry that is greater than t. □

5.4 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 $\mathbf{x}_j$ (right child) and belong to $\mathsf{range}(u)$) in the subtree rooted on u. Therefore

$$\begin{aligned} d(u, \mathbf{x}_i) + d(u, \mathbf{x}_j) &= \mathbf{L}f(\mathbf{x}_i) + \mathbf{R}f(\mathbf{x}_j) = \mathbf{MU}f(\mathbf{x}_i) + \mathbf{MD}f(\mathbf{x}_j) \\ &= \mathbf{M}\left[\mathbf{U}g(\mathbf{x}_i) + \mathbf{D}g(\mathbf{x}_j)\right] = d(u)\,. \end{aligned}$$

This is due to Relation 1 and also due to the fact that $g(.)$ can be picked to be $f(.)$, which is an n-admissible radix-2 representation, therefore satisfying the constraint of the inputs of Definition 4. Hence the base case holds. Assume the theorem holds for any internal node z that lies at level $0 < t + 1 \leq \ell$. Therefore

$$d(z) = \sum_{i \in \mathsf{range}(z)} d(z, \mathbf{x}_i)\,.$$

Let u be an internal node that lies at level t and let $i_1, i_2, \ldots, i_u$ be the indices in $\mathsf{range}(u)$ in *sorted* order. Let v be the left child of u and w be the right child of u. Then, by the definition of the partial digest of the node u (Definition 9) we

$$\begin{aligned} d(u) = \sum_{i \in \mathsf{range}(u)} d(u, \mathbf{x}_i) &= \sum_{j=1}^{u/2} \mathbf{L}f(d(v, \mathbf{x}_j)) + \sum_{j=u/2+1}^{u} \mathbf{R}f(d(w, \mathbf{x}_j)) \\ = \mathbf{MU} \sum_{j=1}^{u/2} f(d(v, \mathbf{x}_j)) &+ \mathbf{MD} \sum_{j=u/2+1}^{u} f(d(w, \mathbf{x}_j))\,. \end{aligned}$$

By Corollary 1 there exist $g(.)$ representations whose entries are at most $u/2 \leq n$ such that

$$d(u) = \mathbf{MU}g\left(\sum_{j=1}^{u/2} d(v, \mathbf{x}_j)\right) + \mathbf{MD}g\left(\sum_{j=u/2+1}^{u} d(w, \mathbf{x}_j)\right)\,.$$

By the inductive step this can be written as

$$d(u) = \mathbf{M}[\mathbf{U}g(d(v)) + \mathbf{D}g(d(w))]\,,$$

where $g(.)$ are radix-2 representations that are n-admissible, since they are the sum of at most $u/2 = n/2$ binary representations. Therefore this satisfies Definition 1 and $d(u)$ is indeed the correct digest of any internal node u, as computed by Relation 1. This completes the proof. □

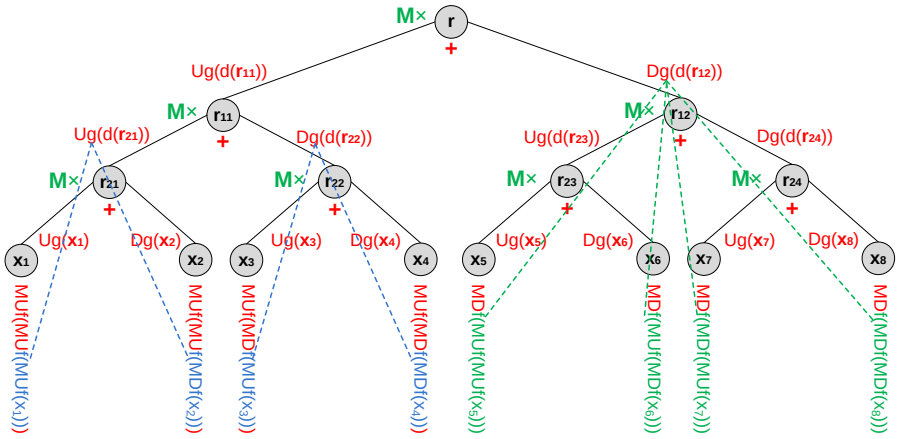

Fig. 1. Tree T built on top of a table with 8 values $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_8$. After producing an *$n$-admissible* radix-2 $g(.)$ representation of the children digests, we multiply with either **U** or **D**, then we add the two resulting digests and we compute the hash function on them by multiplying with **M**. At the leaves of the tree we show the terms that correspond to each index, as computed by Theorem 3 (i.e., the *partial digests* of the root r with reference to every value at the table). The $g(.)$ representation of the internal nodes are indicated with dashed lines (see Definition 10). Note that the $g(.)$ representations of the internal nodes are the sum of specific $f(.)$ representations of the leaves, for example, $g(d(r_{12})) = f(\mathbf{L}f(\mathbf{L}f(\mathbf{x}_5))) + f(\mathbf{L}f(\mathbf{R}f(\mathbf{x}_6))) + f(\mathbf{R}f(\mathbf{L}f(\mathbf{x}_7))) + f(\mathbf{R}f(\mathbf{R}f(\mathbf{x}_8)))$, where $\mathbf{MU} = \mathbf{L}$ and $\mathbf{MD} = \mathbf{R}$.

5.5 Proof of Theorem 4

Correctness. Let $A = D_0$ be any table of n entries. Fix the security parameter k and output sk and $\mathsf{pk} = (\mathbf{M}, q)$ by calling algorithm $\mathsf{genkey}()$. Then output an authenticated data structure $\mathsf{auth}(D_0)$ and the respective digest d_0, by calling algorithm $\mathsf{setup}()$. Pick a polynomial number of updates—namely, pick a polynomial number of pairs of indices and values to be written on the respective indices—and update $\mathsf{auth}(D_0)$ and d_0 by calling algorithm $\mathsf{refresh}()$. Let D_h be the final table A, $\mathsf{auth}(D_h)$ be the produced authenticated data structure and d_h be the final digest. Let i be an index and let $y = A[i]$. Output a proof $\Pi(q)$ for index i and answer y by calling $\mathsf{query}()$. $\Pi(q)$ contains pairs $(g(d(v_j)), g(d(\mathsf{sib}(v_j))))$ $(j = \ell, \ell-1, \ldots, 1)$ of n-admissible representations, where $v_\ell, v_{\ell-1}, \ldots, v_1$ are the nodes on the path from index i (i.e., node v_ℓ) to the first child v_1 of the root of the root of the tree T. For the elements of the proof, the following are true: (a) $g(d(v_\ell)) = f(y)$ (definition of a leaf digest); (b) $d(v_{j-1}) = \mathbf{M}(\mathbf{U}g(d(v_j)) + \mathbf{D}g(d(\mathsf{sib}(v_j))))$ or $d(v_{j-1}) = \mathbf{M}(\mathbf{D}g(d(v_j)) + \mathbf{U}g(d(\mathsf{sib}(v_j))))$—according to left child or right child relation—, for $j = \ell, \ell-1, \ldots, 1$ and where v_0 is the root of the tree (by Relation 1); (c) The $g(.)$ representations in $\Pi(q)$ are always n-admissible, i.e., they are maintained to be n-admissible during updates, since $\mathsf{refresh}()$ always updates the $g(.)$ representations so that Definition 10 is satisfied, which by Corollary 2 gives n-admissible representations. Based on (a), (b) and (c) and the code

of verify() from Section 3, we conclude that verify() always accepts a proof for index i (of answer $y = A[i]$) computed by query().

Security. Fix the security parameter k and output sk and pk $= (\mathbf{M}, q)$ by calling algorithm genkey(). Let Adv be a polynomially-bounded adversary. Adv picks an initial table $A = D_0$ of n entries and outputs authenticated data structure auth(D_0), the respective digest d_0, tree T of ℓ levels, by calling algorithm setup() through oracle access. Then Adv picks a polynomial number of updates—namely, he picks a polynomial number of pairs of indices and values to be written on the respective indices: Let D_h be the final table A, and d_h be the final digest as produced by the adversary through oracle access to algorithm update(). Let i be an index, $y = A[i]$ be the value stored in this index and $v_l, v_{l-1}, \ldots, v_0$ be the path of T from the node referring to index i to the root of T. The adversary Adv outputs an incorrect answer $\alpha(q) \neq y$ and also a proof $\Pi_i = (\pi_l, \pi_{l-1}, \ldots, \pi_1)$ ($l = O(\log n)$) where $\pi_j = (\alpha_j, \beta_j)$ (see algorithm query()). We define now the following events, related to the choice of the proof above made by the adversary. Our goal will be to the express the probability that verify$(i, \alpha(q), \Pi_i, d_h,$ pk$)$ accepts while $\alpha(q) \neq y$ as a function of the following events. Note that d_h is the correct digest of the authenticated data structure:

1. $E_{l,0}$: The value α_l picked by Adv is such that α_l is *not* an n-admissible $g(.)$ representation of y;
2. E_j: For $j = l-1, \ldots, 1$, the values α_j and $\alpha_{j+1}, \beta_{j+1} \in \{0, 1, \ldots, n\}^{k \log q}$ picked by Adv are such that α_j *is* an n-admissible $g(.)$ representation of
$$\mathbf{M}(\mathbf{U}\alpha_{j+1} + \mathbf{D}\beta_{j+1}) .$$
Assume, without loss of generality that a convenient index $i = 0$ is used so that the order of **U** and **D** is always the same. This event can be partitioned into two mutually exclusive events, i.e., $E_j = E_{j,0} \cup E_{j,1}$ such that
   - $E_{j,0}$: Value α_j is *not* an n-admissible $g(.)$ representation of the digest of node v_j, as defined in Relation 1;
   - $E_{j,1}$: Value α_j is an n-admissible $g(.)$ representation of the digest of node v_j, as defined in Relation 1.
3. $E_{0,1}$: The values $\alpha_1 \in \{0, 1, \ldots, n\}^{k \log q}$ and $\beta_1 \in \{0, 1, \ldots, n\}^{k \log q}$ picked by Adv are such that
$$d_h = \mathbf{M}(\mathbf{U}\alpha_1 + \mathbf{D}\beta_1).$$

The probability that verify() accepts, while α_l is *not* an n-admissible $g(.)$ representation of y is the probability

$$\begin{aligned}
&\Pr[E_{l,0} \cap E_{l-1} \cap E_{l-2} \cap \ldots \cap E_{0,1}] \\
&= \Pr[E_{l,0} \cap (E_{l-1,0} \cup E_{l-1,1}) \cap (E_{2,0} \cup E_{2,1}) \cap \ldots \cap E_{0,1}] \\
&\leq \Pr[E_{l,0}|E_{l-1,1}] + \Pr[E_{l-1,0}|E_{l-2,1}] + \Pr[E_{l-2,0}|E_{l-3,1}] + \ldots + \Pr[E_{1,0}|E_{0,1}] \\
&= \sum_{j=1}^{l} \Pr[E_{j,0}|E_{j-1,1}] .
\end{aligned}$$

Note that the event $E_{j,0}|E_{j-1,1}$ is equivalent with the event value "value α_j is *not* an n-admissible $g(.)$ representation of the digest of node v_j (as defined in Relation 1)

given that: (a) value α_{j-1} *is* an n-admissible $g(.)$ representation of the digest $d(v_{j-1})$ of node v_{j-1}; (b) $d(v_{j-1}) = \mathbf{M}(\mathbf{U}\alpha_j + \mathbf{D}\beta_j)$. However, from Relation 1, it should be that $d(v_{j-1}) = \mathbf{M}(\mathbf{U}g(d(v_j)) + \mathbf{D}g(d(\mathsf{sib}(v_j))))$, where $g(d(v_j))$ and $g(d(\mathsf{sib}(v_j)))$ are the digests of nodes v_j and $\mathsf{sib}(v_j)$ respectively. Therefore (α_j, β_j) is a collision with $(g(d(v_j)), g(d(\mathsf{sib}(v_j))))$. By Theorem 2—which gives $\gamma = O(nk\sqrt{\log n + \log k}) = \mathsf{poly}(k)$ since $q = O(k^{2.5}\delta \log k)$ and $\delta = n$—and Assumption 1, $\Pr[E_{j,0}|E_{j-1,1}]$ is $\mathsf{neg}(k)$, for all $j = l, l-1, \ldots, 1$. Therefore the sum

$$\sum_{j=1}^{l} \Pr[E_{j,0}|E_{j-1,1}]$$

is also $\mathsf{neg}(k)$, since $l = O(\log n) = O(\log k)$. This concludes the proof.

Algorithm $\mathsf{setup}()$ **and group complexity of** $\mathsf{auth}(D)$**.** The algorithm needs to compute the n-admissible radix-2 representations $g(d(u))$ of digests $d(u)$ for every internal node u of the tree T. Note that by Definition 10, there are $n/2, n/4, n/8, \ldots, 2$ such representations that need to be computed for levels $\ell-1, \ell-2, \ell-3, \ldots, 1$ respectively, each one being the sum of $2, 4, 8 \ldots, n/2$ binary representations respectively, i.e.,

$$g(d(u)) = \sum_{i \in \mathsf{range}(u)} f(d(u, \mathbf{x}_i)) \,.$$

Since computing $f(d(u, \boldsymbol{x}_i))$ has access complexity $O(1)$ (they are just functions of specific values), it follows that the computation of the $g(.)$ representations for all the internal nodes of the tree requires access complexity

$$\frac{n}{2} \times 2 + \frac{n}{4} \times 4 + \frac{n}{8} \times 8 + \ldots + 2 \times \frac{n}{2} = O(n \log n) \,.$$

Note now in the CREW model, we can use $O(\log n)$ processors, i.e., one processor for each level of the tree. By reading the values $\mathbf{x}_i$ concurrently and writing the values $g(d(u))$ at different memory locations, it follows that each processor will have to do $O(n)$ work in the CREW model. Finally, we note that the output authenticated data structure stores with each internal node u of the tree T the respective n-admissible radix-2 representations $g(d(u))$. Therefore the group complexity of $\mathsf{auth}(D)$ is $O(n)$. This completes the proof.

Algorithm $\mathsf{update}()$**.** For each update from $\mathbf{x}_i$ to $\mathbf{x}'_i$, the algorithm sets $d_{h+1} = d_h - d(r, \mathbf{x}_i) + d(r, \mathbf{x}'_i)$, where $d(r, \mathbf{x}_i)$ and $d(r, \mathbf{x}'_i)$ are the *partial digests* of the root r, defined in Definition 9. Therefore the access complexity is $O(1)$.

Algorithm $\mathsf{refresh}()$**.** For each update from $\mathbf{x}_i$ to $\mathbf{x}'_i$, the algorithm should update the values $g(d(v_j))$ for $j = \ell, \ell-1, \ldots, 1$. This is achieved via Definition 10, by setting

$$g(d'(v_j)) = g(d(v_j)) - f(d(v_j, \mathbf{x}_i)) + f(d(v_j, \mathbf{x}'_i)) \,, \tag{2}$$

(the invariant of Definition 10 must be maintained) for $j = \ell, \ell-1, \ldots, 1$ and where $d(v_j, \mathbf{x}_i)$, $d(v_j, \mathbf{x}'_i)$ are the partial digests of node v_j with reference to $\mathbf{x}_i$ and $\mathbf{x}'_i$ respectively. Since $\ell = O(\log n)$ the result follows. Note also that the update Relations 2 are

independent from one another. In the CREW model, we can use $O(\log n)$ processors, i.e., one processor for each level of the tree. By reading the values $\mathbf{x}_i$ concurrently and writing the values $g(d(u))$ at different memory locations, it follows that each processor will have to do $O(1)$ work in the CREW model.

Algorithm query(). Since $\ell = O(\log n)$ values have to be collected to construct the proof, the result follows. Moreover, with $O(\log n)$ processors—one processor per node, this algorithm is parallelizable in the EREW model, with $O(1)$ complexity.

Algorithm verify(). Since $\ell = O(\log n)$ values have to be processed to do the verification of the proof, the result follows. However, parallelizing the algorithm requires *concurrent* write, since all the processors need to write on the same location either "accept" or "reject". □

5.6 An authenticated Bloom filter

In this paragraph we show how we can use the lattice-based hash function to authenticate the Bloom filter functionality, a space-efficient dictionary, originally introduced in [5]. The Bloom filter consists of an array (table) $A[0 \dots n-1]$ 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 $\{0, \dots, n-1\}$ are used (these are not lattice-based hash functions) and for each element $s \in S$ we set the bits $A[h_i(s)]$ to 1, for $i = 1, \dots, K$. In this way, false positives can occur, i.e., an element that is not present 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 $K = \ln 2(n/r)$ [5].

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. [11]. 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 [7], 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 [7].

By the above description, it is clear that we can use our lattice-based construction to authenticate the Bloom filter functionality: 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):

Theorem 6 *Let k be the security parameter. Then there exists an authenticated data structure scheme $\mathcal{ABF} = \{$*genkey, setup, update, refresh, query, verify*$\}$ for a Bloom filter D of n entries, storing r elements and using K hash functions such that: (1) It is correct and secure according to Definitions 1, 2 respectively and assuming the hardness of* GAPSVP$_\gamma$ *for $\gamma = O(nk\sqrt{\log n + \log k})$; (2) The access complexity of (i)* setup() *is $O(n \log n)$ or $O(n)$ using $O(\log n)$ processors in the CREW model; (ii)* update() *is $O(K)$; (iii)* refresh() *is $O(K \log n)$ or $O(K)$ using $O(\log n)$ processors in the CREW model; (iv)* query() *is $O(K \log n)$ or $O(K)$ using $O(\log n)$ processors in the EREW model; (v)* verify() *is $O(K \log n)$ or $O(K)$ using $O(\log n)$ processors in the ERCW*

model; (3) The group complexity of (i) the proof $\Pi(q)$ for a query q is $O(K \log n)$; (ii) the authenticated data structure $\mathsf{auth}(D)$ *is $O(n)$.*

Proof. The construction for an authenticated Bloom filter is the same with Theorem 4. The extra K multiplicative factor in the complexities is due to the fact that one operation in the authenticated Bloom filter (insertion/deletion of an element) requires $O(K)$ operations on an authenticated table. This follows by the construction and the definition of the Bloom filter data structure. □

5.7 Online memory checking definition

Definition 11 *Let $\mathcal{M}$ be an n-cell unreliable memory. An online non-adaptive memory checker $\mathcal{C} = (\Sigma, n, q, s)$ over an alphabet Σ having query complexity q and keeping reliable (and possibly secret) memory s is a probabilistic Turing machine with five tapes:*

- *A read-only input tape for receiving read/write requests from the user $\mathcal{U}$ to the unreliable memory $\mathcal{M}$ of n cells, indexed by $1, 2, \ldots, n$;*
- *A write-only output tape for sending responses back to the user;*
- *A read-write work tape, i.e., the (secret) reliable memory s;*
- *A write-only tape for sending read/write requests to the memory $\mathcal{M}$;*
- *A read only input tape for receiving $\mathcal{M}$'s responses.*

A checker is presented with $\mathsf{write}(i, x)$ *and* $\mathsf{read}(i)$ *requests made by $\mathcal{U}$ to $\mathcal{M}$, where $i \in \{1, 2, \ldots, n\}$. After each* read *request $\mathcal{C}$ returns an answer or outputs that $\mathcal{M}$'s operation is* **BUGGY**. *$\mathcal{C}$'s operation should be both correct and secure:*

1. Correctness*: For any polynomially-large sequence of user requests, as long as $\mathcal{M}$ answers all of $\mathcal{C}$'s* read *requests correctly, $\mathcal{C}$ also answers all of the user's* read *requests correctly;*
2. Security*: For any any polynomially-large sequence of user requests, for any (even incorrect or malicious) answers returned by $\mathcal{M}$, the probability that $\mathcal{C}$ answers a user request incorrectly is* $\mathsf{neg}(k)$*, where k is the security parameter. $\mathcal{C}$ may either recover the correct answer independently or answer that $\mathcal{M}$ is* **BUGGY***, but it may not answer a request incorrectly (beyond negligible probability).*

5.8 Proof of Theorem 5

Let $\mathcal{LBT} = \{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$ be the authenticated data structure scheme derived in Theorem 4. We show how to construct a parallel online memory checker by using this scheme, in the non-secret key setting. Let $\mathcal{M}$ be the unreliable memory accessed through indices $1, 2, \ldots, n$. Assume we can use $O(\log n)$ checkers $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_u$ where $u = O(\log n)$. The user $\mathcal{U}$ sends his requests to all the chekers simultaneously and all the checkers have access to the *unreliable* memory $\mathcal{M}$ and to some *reliable* memory s. We work in the CREW model—i.e., all the checkers can read simultaneously the same value but writing at the same location simultaneously is not feasible. Let $\{\mathsf{sk}, \mathsf{pk}\} \leftarrow \mathsf{genkey}()$, where $\mathsf{sk} = \emptyset$. The checkers run the algorithm $\{\mathsf{auth}(\mathcal{M}), d_0\} \leftarrow \mathsf{setup}(\mathcal{M}, \mathsf{pk})$ (since $\mathsf{sk} = \emptyset$ we do not use the secret key as input from now on) in parallel, requiring $O(n)$ access complexity in the CREW model (Theorem 4). The authenticated structure $\mathsf{auth}(\mathcal{M})$ is stored in the unreliable memory

(all its parts can be uniquely referenced) and d_0 is stored in the small reliable memory, i.e., $s = d_0$. We have two cases:

1. User $\mathcal{U}$ sends the request $\mathsf{read}(i)$ to all checkers $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_u$. The checkers run algorithm $\mathsf{query}(i, \mathcal{M}, \mathsf{auth}(\mathcal{M}), \mathsf{pk})$ in parallel and output the answer $\mathcal{M}[i]$ and the proof $\Pi(i)$. This requires $O(1)$ requests to the unreliable memory per checker in the EREW model (Theorem 4). Then the algorithm $\mathsf{verify}(i, \mathcal{M}[i], \Pi(i), s, \mathsf{pk})$ is run by the checkers (note that running $\mathsf{query}()$ and $\mathsf{verify}()$ can be combined in one algorithm). The algorithm writes either $\mathcal{M}[i]$ (in this case $\mathsf{verify}()$ accepts) or **BUGGY** (in this case $\mathsf{verify}()$ rejects) in a location of the reliable memory. User $\mathcal{U}$ reads that location and gets the result. We note here that the fact that $\mathsf{verify}()$ is parallelizable in the ERCW model does not affect our complexity results since the *write* part of the algorithm is done on the *reliable* memory—however, requests to the reliable memory are not taken into account in query complexity (only requests to the unreliable memory). Therefore the query complexity of the parallel checker due to *read* operations is $O(1)$ in EREW model;
2. User $\mathcal{U}$ sends the request $\mathsf{write}(i, x)$ to all checkers $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_u$. First the current content of cell i is verified through a $\mathsf{read}(i)$ operation. If this verification succeeds the checkers run algorithm $\{\mathcal{M}', \mathsf{auth}(\mathcal{M}'), s'\} \leftarrow \mathsf{refresh}(x, \mathcal{M}, \mathsf{auth}(\mathcal{M}), s, \mathsf{pk})$ in parallel. Note that this algorithm has $O(1)$ access complexity using $O(\log n)$ processors in the CREW model, by Theorem 4. We need *concurrent read* because all the checkers should be able to read the same value of the old (verified) content of cell i.

Finally, we note that the correctness and the security of the checker comes as a direct result of the correctness and the security of the authenticated data structure scheme $\mathcal{LBT}$. Also, since our lattice-based construction does not use any secret key, it follows that the construction we have described is in the non-secret key setting. This completes the proof. □

5.9 Two-party model

Corollary 5 (Two-party model) *Let k be the security parameter. There exists a two-party authenticated data structures protocol involving a trusted source and an untrusted server for verifying queries on a table of n entries such that: (a) The setup at the source has $O(n \log n)$ access complexity or $O(n)$ using $O(\log n)$ processors in the CREW model; (b) The update at the source has $O(1)$ access complexity; (c) The space needed at the source has $O(1)$ group complexity; (d) The update at the server has $O(\log n)$ access complexity or $O(1)$ using $O(\log n)$ processors in the CREW model; (e) The query at the server has $O(\log n)$ access complexity or $O(1)$ using $O(\log n)$ processors in the EREW model; (f) The space needed at the server has $O(n)$ group complexity; (g) The proof has $O(\log n)$ group complexity; (h) The verification at the source has $O(\log n)$ access complexity or $O(1)$ using $O(\log n)$ processors in the CRCW model; (i) For a query q sent by the source to the server at any time (even after updates), let α be an answer and let π be a proof returned by the server. With probability $\Omega(1 - \mathsf{neg}(k))$, the client accepts the answer α if and only if α is correct.*