Optimal and Parallel Online Memory Checking

Charalampos Papamanthou* — UC Berkeley
Roberto Tamassia† — Brown University

Abstract

Memory checking studies the problem of cryptographically verifying the correctness of untrusted indexed storage. After a series of results yielding checkers with $O(\log n)$ query complexity, Dwork, Naor, Ruthblum and Vaikuntanathan [9] derived an $\Omega(\log n/\log\log n)$ lower bound on the query complexity of any checker operating on memory words of polylogarithmic size, where n is the number of memory indices. In view of this lower bound, we make the following two contributions:

1. We construct an *optimal online memory checker* of $\Theta(\log n/\log\log n)$ query complexity, closing in this way the relevant complexity gap. Our construction employs pseudorandom functions and a simple data grouping technique inspired by I/O algorithms.
2. In our second and main result, we put forth the notion of *parallel online memory checking* and provide parallel checker constructions with $O(1)$ query complexity and $O(\log n)$ processors. We initially show that checkers that use *secret* small memory, including our optimal checker, are easily parallelizable; However, checkers that use only *reliable* small memory cannot be naturally parallelized. We overcome this barrier by employing an algebraic hash function based on lattices assumptions and construct such parallel checkers with only reliable memory. To achieve our result, we establish and exploit a property that we call *repeated linearity* of lattice-based hash functions, that might be of independent interest.

Applications of our checkers include *update-optimal* external memory authenticated data structures. We construct an authenticated B-tree data structure which can be updated with *two* I/Os, outperforming the *logarithmic* update complexity of hash-based external memory Merkle trees.

*Email: cpap@cs.berkeley.edu.
†Email: rt@cs.brown.edu.

1 Introduction

Following the notion of program checking and verification (e.g., [4]), *memory checking* has received a lot of attention (e.g., [7, 9, 20]) since it was first introduced by the seminal work of Blum et al. [5]. Unlike the seemingly close in nature problem of oblivious RAM [21], memory checking is concerned with verifying the *integrity* of dynamic data stored at untrusted repositories. It has also become increasingly relevant nowadays: In the cloud computing paradigm, checking online data storage is critical for businesses and individuals.

In its simplest form, the memory checking problem can be stated as follows: Given an *unreliable* (malicious) memory of n cells, a user desires to read/write a cell i in the memory. The user's request is intercepted by a checker that reads or writes some additional unreliable memory cells and at the same time accesses some *small* reliable (and possibly secret) memory of sublinear size. The checker's task is to either output the correct answer or *reject*, if cell i is corrupted. The probability of returning a corrupted cell as correct should be small, namely an adversary should not be able to tamper with the memory and evade detection.

Two classes of checkers have appeared in the literature so far, namely *secret memory checkers* and *reliable memory checkers*: Secret memory checkers require secret small memory that the adversary cannot access, e.g., see the PRF (pseudorandom function) construction in [9]. Reliable memory checkers employ only reliable small memory that can be accessed but not tampered with by the adversary, e.g., see the UOWHF (universal one-way hash function) construction in [5] or the implicit constructions of Naor and Nissim [19] and Goodrich and Tamassia [12] that use CRHFs (collision-resistant hash functions). Secret memory checkers are more restrictive than reliable memory checkers since all their operations are dependent on a secret key—therefore their operation is not *publicly verifiable*. This restriction is also witnessed through a complexity perspective: More efficient checkers are known to exist in the secret key setting. For example, although Dwork et al. [9] show how to trade-off read and writes for a secret memory checker, they state that it is "intriguingly" difficult to achieve the same result for checkers using UOWHFs.

The quality of a checker construction depends mainly on its *query complexity*, defined as the sum of additional reads and writes that the checker issues to the unreliable memory per user request [20]. Many checkers with $O(\log n)$ query complexity have been constructed so far (see Table 5.1 in the Appendix). While lots of efforts focused on reducing this logarithmic bound, in TCC 2009, Dwork et al. [9] derived an $\Omega(\log n/\log\log n)$ lower bound on the query complexity of any checker operating on words of polylogarithmic size.[1] The existence of the above lower bound, which applies to both secret and reliable memory checkers, gives rise to the results in this paper, outlined in the following paragraph.

Our results. Our first result (Theorem 2) is an *optimal online memory checker*. Our construction can be viewed as an extension of two independently and previously presented constructions (Papamanthou et al. [22] and Dwork et al. [9]). It achieves $\Theta(\log n/\log\log n)$ query complexity while operating on words of $O(\log^2 n)$ size,[2] and is the first construction to match the lower bound by Dwork et al. [9].

Our second and main result (Theorem 4) puts forth the notion of *parallel online memory checking*, defining and constructing parallel online memory checkers with $O(1)$ query complexity and $O(\log n)$ processors. We first observe that parallel algorithms are staightforward to achieve for *secret* memory checkers; however, checkers using only *reliable* memory—mostly applicable in practical scenarios—cannot be naturally parallelized. We overcome this barrier by employing lattice-based cryptography to construct such parallel reliable memory checkers. The security of our solution is based on the difficulty of solving the small integer solution (SIS) problem, which is at least as hard as approximating several hard problems in lattices (e.g., GAPSVP)

[1] Although the lower bound by Dwork et al. [9] does not apply to checkers operating on polynomial-sized words, no such checker that violates this lower bound is known to exist yet. Specifically, constructing such checkers is a long-standing open problem.

[2] Note that the query complexity depends directly on the bit size m of the memory word: Reading b bits from untrusted memory requires $O(b/m)$ number of queries. It essentially simulates the notion of a "page" in I/O complexity.

within polynomial factors [18]. This is the first construction for parallel reliable memory checkers, giving rise to applications such as update-optimal external memory authenticated data structures [26].

Our techniques. An optimal checker can be derived by using a tree of logarithmic degree for organizing the untrusted memory. Since the size of the word is allowed to be polylogarithmic, we observe that one can "compress" sibling data into one word of $O(\log^2 n)$ size, allowing the checker to access a logarithmic number of indexed data with a single query. The detailed construction is in the Appendix, see Section 5.2.

Concerning the parallelism results, for secret memory checkers, a simple observation on the construction of Blum et al. [5] (independence of PRF tags at separate levels of a binary tree), yields parallel algorithms. In the reliable memory setting, which is the main contribution of this part, our solution is a Merkle tree construction [17] with different cryptography: It employs lattice-based hash functions instead of *generic collision-resistant* hash functions (e.g., MD-5 or SHA-256) [17]. Specifically, a property of lattice-based hash functions that we call *repeated linearity* allows us to express the digest of a tree node v as the sum of well-defined functions (called *partial digests*) applied to data stored at the leaves of v's subtree (Theorem 3), enabling Merkle tree updates to be parallelized. This property may also be of general interest and have other applications—more details about this property are given in Section 5.6 of the Appendix.

Applications in authenticated data structures. Our lattice-based memory checker can be applied for verifying queries executed on very large, on-disk, dynamic database indices. In Section 4, we show how to construct a lattice-based authenticated B-tree that has the following desirable and unique features: **(a)** Updating the state (digest) of the index requires *two* I/Os, outperforming the *logarithmic* complexity of hash-based Merkle trees (as opposed to [15]) and at the same time maintaining the logarithmic proof size; **(b)** By employing a *logarithmic* number of machines, all the operations of the authenticated index can be performed with *two* I/Os, due to parallelism. We note that the gap between $O(1)$ I/Os and $O(\log n)$ I/Os is especially relevant in practice since an I/O is orders of magnitude slower than an access to internal memory.

Related work. Memory checking has received a lot of attention in the last few years. The problem has its roots in the seminal paper of Merkle [17], where the first solution for structured data integrity checking is presented. However, the first formal definition of memory checking appeared in the work of Blum et al. [5], where two different constructions of logarithmic query complexity were developed. Lower bounds on space and query complexity of memory checking were subsequently presented by Naor and Rothblum [20] (information theoretic model) and by Dwork et al. [9] (computational model). Apart from the lower bounds in [9], new constructions achieving read/write trade-offs, but still of $\Omega(\log n)$ query complexity, are also presented. Checking the correctness of more complicated data structures and computation has also been studied in the field of *authenticated data structures* [26] and *verifiable computation* [3, 23] respectively, where more efficiency is achieved by using more advanced cryptography and slightly stronger assumptions. All above solutions are not parallelizable, except for the construction of Hall and Jutla [13] (but in the secret key setting). A comparison of existing literature and our work is provided in Table 5.1 in the Appendix.

As we mentioned above, we achieve our parallelism results by employing *lattice-based cryptography*. Lattice-based cryptography began with Ajtai's construction of an one-way hash function based on hard lattices problems [1]. Various generalizations and improvements have appeared since then [10, 11, 16, 18, 24]. Finally, and related to the context of this work, Banerjee et al. [2] recently observed that lattice-based constructions can be used to construct *highly-parallelizable* pseudorandom functions.

2 Preliminaries

We start with some basic definitions on online memory checking and *lattices*. In the following, k denotes the security parameter and PPT stands for *probabilistic polynomial-time*. Upper case bold letters denote ma-

trices, e.g., $\mathbf{B}$, lower case bold letters denote vectors, e.g., $\mathbf{b}$, and lower case italic letters denote scalars. For vector $\mathbf{x}$, $\|\mathbf{x}\|$ denotes the Euclidean norm of $\mathbf{x}$. We finally use the notation $[\delta]$ to denote the set $\{0, 1, \ldots, \delta\}$.

2.1 Online memory checking

The online memory checking model [5], in the sequential model, can be (informally) described as follows: Suppose $\mathcal{M}$ is an *unreliable* (malicious) memory that stores indexed values $\mathbf{T}[0]\ \mathbf{T}[1]\ \ldots\ \mathbf{T}[n-1]$. Memory $\mathcal{M}$ consists of memory words that can take values from an alphabet Σ. A user $\mathcal{U}$ wants to read or write a cell $i \in \{0, 1, \ldots, n-1\}$. All requests go through a checker $\mathcal{C}$. The checker executes the requests by accessing the unreliable memory $\mathcal{M}$ and also some small reliable (and possibly secret) information μ of *sublinear* size. The requests are either executed correctly or otherwise the checker outputs **`reject`**, with high probability. For example, and after a read request made by the user, the probability of the checker returning the corrupted (or outdated) content of a cell as correct should be small, even after several write requests have occurred. 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 only considering such checkers.

In the following we give the definition (Definition 1) of an online memory checker. For ease of presentation, we do not employ the Turing machine definition that was originally used in [5] but we provide an *equivalent* definition: Both definitions (Turing machine one and the definition in this paper) capture standard notions of *correctness* and *security* in the data outsourcing framework: Namely, (i) as long as the adversary does not tamper with the unreliable memory and the checker algorithms (therefore correct answers are returned), verification algorithms always *accept* (correctness—see Definition 2); (ii) a computationally-bounded adversary cannot persuade a checker (except with probability ϵ) for the validity of an incorrect state of $\mathcal{M}$, even if he can adaptively update the memory to a state of his liking (security—see Definition 3).

Definition 1 (Online memory checker) *Let $\mathbf{T} = \mathbf{T}[0]\ \mathbf{T}[1]\ \ldots\ \mathbf{T}[n-1]$ be n indexed values, $\mathcal{M}$ be some public memory over alphabet Σ and μ be some reliable (and secret) memory. An* online memory checker $\mathcal{C}(\Sigma, \mathbf{T}, n) = \{\mathsf{genkey}, \mathsf{setup}, \mathsf{secureRead}, \mathsf{secureWrite}\}$ *is a tuple of the following four PPT algorithms:*

1. $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{genkey}(1^k)$. *On input the security parameter k, this algorithm outputs secret and public keys* sk *and* pk *respectively;*
2. $(\mathcal{M}, \mu) \leftarrow \mathsf{setup}(\mathbf{T}, \mathsf{pk}, \mathsf{sk})$. *On input n values $\mathbf{T}$ and the secret and public keys* sk *and* pk*, this algorithm outputs the unreliable public memory $\mathcal{M}$*[3] *and the reliable (and secret) memory μ;*
3. $\{\alpha, \texttt{reject}\} \leftarrow \mathsf{secureRead}(\mathsf{index}, \mathcal{M}, \mu, \mathsf{sk}, \mathsf{pk})$. *On input index* $\mathsf{index} \in \{0, \ldots, n-1\}$*, the unreliable public memory $\mathcal{M}$, the reliable (and secret) memory μ, and the secret and public keys* sk *and* pk*, this algorithm outputs either a value α or* **`reject`***;*
4. $\{(\mathcal{M}', \mu'), \texttt{reject}\} \leftarrow \mathsf{secureWrite}(\mathsf{index}, \beta, \mathcal{M}, \mu, \mathsf{sk}, \mathsf{pk})$. *On input index* $\mathsf{index} \in \{0, \ldots, n-1\}$*, a value β to be written at index* index*, the unreliable public memory $\mathcal{M}$, the reliable (and secret) memory μ, and the secret and public keys* sk *and* pk*, this algorithm outputs either new values $\mathcal{M}'$ and μ' or* **`reject`**.

We continue with the definitions of correctness and ϵ-security of an online memory checker.

Definition 2 (Correctness) *Let* $\mathcal{C}(\Sigma, \mathbf{T}, n) = \{\mathsf{genkey}, \mathsf{setup}, \mathsf{secureRead}, \mathsf{secureWrite}\}$ *be an online memory checker. We say that the checker $\mathcal{C}(\Sigma, \mathbf{T}, n)$ is* correct *if, for all $k \in \mathbb{N}$, for all* $\{\mathsf{sk}, \mathsf{pk}\}$ *output by algorithm* genkey*, for all $\mathcal{M}, \mu$ output by one invocation of algorithm* setup *followed by polynomially-many invocations of algorithm* $\mathsf{secureWrite}$ *and for all indices* $0 \leq \mathsf{index} \leq n-1$*, if α is the correct value stored at location* $\mathbf{T}[\mathsf{index}]$*, then* $\alpha \leftarrow \mathsf{secureRead}(\mathsf{index}, \mathcal{M}, \mu, \mathsf{sk}, \mathsf{pk})$.

[3] The unreliable public memory $\mathcal{M}$ output by this algorithm stores, among other quantities, values $\mathbf{T}$ themselves.

Definition 3 (ϵ-security) *Let* $\mathcal{C}(\Sigma, \mathbf{T}, n) = \{\mathsf{genkey}, \mathsf{setup}, \mathsf{secureRead}, \mathsf{secureWrite}\}$ *be an online memory checker, k be the security parameter and* $\{\mathsf{sk}, \mathsf{pk}\} \leftarrow \mathsf{genkey}(1^k)$*. Denote with* Adv *a PPT adversary that has access only to* pk *and with* Chal *a challenger that has also access to* sk*.*

1. ***Initialization and updates.*** Adv *picks an initial state of the index* $\mathbf{T}$ *and sends* $\mathbf{T}$ *to* Chal*.* Chal *computes* $\mathcal{M}, \mu$ *through algorithm* setup*, sends the public memory* $\mathcal{M}$ *to* Adv *and keeps* μ *secret. Then, for* $i = 1, \ldots, h = \mathsf{poly}(k)$*,* Adv *picks an index* $0 \leq \mathsf{ind}_i \leq n-1$ *and a value* β_i *and sends* ind_i *and* β_i *to* Chal*.* Chal *executes* $\mathsf{secureWrite}(\mathsf{ind}_i, \beta_i, \mathcal{M}, \mu, \mathsf{sk}, \mathsf{pk})$*. If* secureWrite *does not reject, then it outputs the new values of* $\mathcal{M}'$ *and* μ' *and sets* $\mathcal{M} = \mathcal{M}'$ *and* $\mu = \mu'$ *(if it rejects the values* $\mathcal{M}$ *and* μ *do not change). Finally* Chal *sends to the adversary the new public memory* $\mathcal{M}$*, keeping the new memory* μ *secret.*
2. ***Forge.*** Adv *picks an index* ind *of* $\mathbf{T}$ *and a final state of the public memory* $\bar{\mathcal{M}}$*.*

We say that the online memory checker $\mathcal{C}(\Sigma, \mathbf{T}, n)$ *is* ϵ-secure *if for all* $k \in \mathbb{N}$*, for all* $\{\mathsf{sk}, \mathsf{pk}\}$ *output by algorithm* genkey*, and for all PPT adversaries* Adv *it is*

$$\Pr\left[\{\mathsf{ind}, \bar{\mathcal{M}}\} \leftarrow \mathsf{Adv}(1^k, \mathsf{pk}, \mathcal{M}); \alpha \leftarrow \mathsf{secureRead}(\mathsf{ind}, \bar{\mathcal{M}}, \mu, \mathsf{sk}, \mathsf{pk}); \mathbf{T}[\mathsf{ind}] \neq \alpha\right] \leq \epsilon\,.$$

On the parameter ϵ. By default, the above parameter ϵ is less than or equal to $1/2$ in memory checking literature [9]. Specifically, the lower bound result [9] applies to all ϵ-secure checkers for $\epsilon \leq 1/3$. As such, our tight upper bound result uses a value of $\epsilon = 1/n^c$ for some $c > 1$. However, our parallel lattice-based checker uses a value of $\epsilon = \mathsf{neg}(k)$.[4] Note that, as observed by Dwork et al. [9], smaller ϵ values result into more practical checkers, e.g., checkers that provide guarantees even after a detected malfunction.

Query complexity. The quality of a checker depends on its *query complexity* [9], defined as follows:

Definition 4 (Query complexity) *Let* $\mathcal{C}(\Sigma, \mathbf{T}, n) = \{\mathsf{genkey}, \mathsf{setup}, \mathsf{secureRead}, \mathsf{secureWrite}\}$ *be a correct and ϵ-secure online memory checker. The* query complexity *of* $\mathcal{C}(\Sigma, \mathbf{T}, n)$ *is defined as the sum of the number of queries that algorithms* secureRead *and* secureWrite *perform on the unreliable public memory.*

2.2 Lattices

Our second result relies on lattice-based cryptography. A full-rank k-dimensional lattice is defined as the infinite-sized set of all vectors produced as the integer combinations of a basis $\mathbf{V} = \{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k\}$, where $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k$ are linearly independent, all belonging to $\mathbb{R}^k$. We denote the lattice produced by $\mathbf{V}$ with $L(\mathbf{V})$. Let $\lambda(\mathbf{V})$ denote the length of the *shortest* vector (in an Euclidean sense) in $L(\mathbf{V})$. Finding such a vector is a difficult problem. The security of our constructions is based on the difficulty of solving a similar well-known problem in lattices, namely the "gap" version of the shortest vector problem (GAPSVP_γ):

Definition 5 (Problem GAPSVP_γ) *An input to the problem* GAPSVP_γ *is a k-dimensional lattice basis* $\mathbf{V}$ *and a number d. In* YES *inputs* $\lambda(\mathbf{V}) \leq d$ *and in* NO *inputs* $\lambda(\mathbf{V}) > \gamma \times d$*, where* $\gamma \geq 1$*.*

For exponential values of γ, i.e., $\gamma = 2^{O(k)}$ (k is the security parameter), one can use the LLL algorithm [14] and decide the above problem in polynomial time. However, for polynomial γ, no efficient algorithm is known to date, even for factors slightly smaller than exponential [25], i.e., very big polynomials. Moreover, for polynomial factors, there is no proof that this problem is NP-hard.[5] A well-accepted assumption we will need to argue about the security of our scheme is the following:

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 5 and k be the security parameter. There is no PPT algorithm for solving* GAPSVP_γ *for* $\gamma = \mathsf{poly}(k)$*, except with negligible probability* $\mathsf{neg}(k)$*.*

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

[5] For $\gamma > \sqrt{k/\log k}$, it is unlikely that this problem is NP-hard and no efficient algorithm is known to date [25].

2.3 Small integer solution problem

After Ajtai introduced the first one-way function based on hard lattices problems [1], Goldreich et al. [11] presented a variation of the function with collision resistance, which was finally generalized by Micciancio and Regev [18]. In our construction we are using a modification (allowing two inputs) of this generalized function. The security of our function is based on the difficulty of the *small integer solution* problem (SIS):

Definition 6 (Problem $\mathsf{SIS}_{q,m,\beta}$) *Given an integer q, a matrix $\mathbf{M} \in \mathbb{Z}_q^{k\times m}$ picked uniformly at random 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 $\|\mathbf{z}\| \leq \beta$.*

For certain parameters, Gentry, Peikert and Vaikuntanathan [10] proved that $\mathsf{SIS}_{q,m,\beta}$ can be reduced to GAPSVP_γ for polynomial γ, and therefore, due to Assumption 1, $\mathsf{SIS}_{q,m,\beta}$ is believed to be intractable:

Lemma 1 (Hardness of $\mathsf{SIS}_{q,m,\beta}$) *Let $\mathsf{SIS}_{q,m,\beta}$ be an instance of the small integer solution problem, as defined in Definition 6. Let also β, m, q be polynomially-bounded, where q is a prime and $q \geq \beta \cdot \sqrt{m} \cdot \omega(\sqrt{k \log k})$. Then the problem $\mathsf{SIS}_{q,m,\beta}$ is as hard as approximating the problem GAPSVP_γ in the worst case to within certain $\gamma = O(\beta \cdot \sqrt{k} \cdot \mathsf{poly}(\log k))$.*

We now give the algorithm that chooses the exact values q, m, β, for which the problem $\mathsf{SIS}_{q,m,\beta}$ is hard:

Algorithm $\{q, m, \beta\} \leftarrow \mathsf{parameters}(k, \delta)$: On input the security parameter k and $\delta = \mathsf{poly}(k)$, let q be the smallest positive prime satisfying $q/\log q \geq \delta k \cdot \omega(\sqrt{k \log k})$. Set $m = \lceil k \log q \rceil$ and $\beta = \delta\sqrt{m}$.

The parameters chosen by the above algorithm comply with Lemma 1: First, as $\delta = \mathsf{poly}(k)$, all q, m, β are $\mathsf{poly}(k)$. Second, $q/\log q \geq \delta k \cdot \omega(\sqrt{k \log k}) \Leftrightarrow q \geq \beta \cdot \sqrt{m} \cdot \omega(\sqrt{k \log k})$, since $\beta = \delta\sqrt{m}$ and $m = k \log q$. Note that there is always a prime $q = \Theta(\delta \cdot k \log k \sqrt{k \log k})$ satisfying the inequality above.

2.4 Lattice-based hash function

We now describe our lattice-based hash function which is the main building block of our approach. The security of our function is based on the difficulty of solving $\mathsf{SIS}_{q,m,\beta}$, as described in the previous paragraph.

Definition 7 (Hash function) *Let k be the security parameter, $\delta = \mathsf{poly}(k)$ and q, m, β be the parameters output by algorithm $\mathsf{parameters}(k, \delta)$. Let also $\mathbf{L}, \mathbf{R} \in \mathbb{Z}_q^{k\times m}$ be two $k \times m$ matrices picked uniformly at random. We define the function $h_\delta : [\delta]^m \times [\delta]^m \rightarrow \mathbb{Z}_q^k$ as $h_\delta(\mathbf{x}, \mathbf{y}) = \mathbf{L} \cdot \mathbf{x} + \mathbf{R} \cdot \mathbf{y} \mod q$.*

Function h_δ is collision-resistant based on the difficulty of GAPSVP_γ:

Theorem 1 (Strong collision resistance) *Let k be the security parameter, $\delta = \mathsf{poly}(k)$ and $\{q, m, \beta\} \leftarrow \mathsf{parameters}(k, \delta)$. Let also $\mathbf{L}, \mathbf{R} \in \mathbb{Z}_q^{k\times m}$ be two $k \times m$ matrices picked uniformly at random. Assuming hardness of GAPSVP_γ for $\gamma = \mathsf{poly}(k)$ (see Assumption 1), there is no PPT algorithm that outputs two pairs of vectors $(\mathbf{x}_1, \mathbf{y}_1) \in [\delta]^m \times [\delta]^m$ and $(\mathbf{x}_2, \mathbf{y}_2) \in [\delta]^m \times [\delta]^m$ with $(\mathbf{x}_1, \mathbf{y}_1) \neq (\mathbf{x}_2, \mathbf{y}_2)$ such that*

$$\mathbf{L} \cdot \mathbf{x}_1 + \mathbf{R} \cdot \mathbf{y}_1 = \mathbf{L} \cdot \mathbf{x}_2 + \mathbf{R} \cdot \mathbf{y}_2 \mod q\,, \tag{1}$$

except with negligible probability $\mathsf{neg}(k)$.

Proof: Let $\{q, m, \beta\} \leftarrow \mathsf{parameters}(k, \delta)$. We reduce the problem $\mathsf{SIS}_{q,2m,\beta\sqrt{2}}$ to the problem of finding collisions to the above function. Consider the matrix

$$\mathbf{M} = [\mathbf{L}\ \mathbf{R}] \in \mathbb{Z}_q^{k\times 2m}\,.$$

Matrix $\mathbf{M}$ is distributed uniformly at random, since $\mathbf{L}$ and $\mathbf{R}$ are picked uniformly at random. It is easy to write $\mathbf{L}$ and $\mathbf{R}$ as a function of $\mathbf{M}$, i.e., $\mathbf{L} = \mathbf{MU}$ and $\mathbf{R} = \mathbf{MD}$, where $\mathbf{U} = [\mathbf{I}_m\ \mathbf{O}_m]^\mathsf{T}, \mathbf{D} = [\mathbf{O}_m\ \mathbf{I}_m]$, $\mathbf{I}_m$

denotes the square identity matrix of dimension m and $\mathbf{O}_m$ denotes the square zero matrix of dimension m. Therefore Equation 1 can be written as

$$\mathbf{MU}\cdot\mathbf{x}_1+\mathbf{MD}\cdot\mathbf{y}_1=\mathbf{MU}\cdot\mathbf{x}_2+\mathbf{MD}\cdot\mathbf{y}_2\Leftrightarrow\mathbf{M}\cdot[\mathbf{U}(\mathbf{x}_1-\mathbf{x}_2)+\mathbf{D}(\mathbf{y}_1-\mathbf{y}_2)]=\mathbf{0}\,.$$

Consider now the $k\times 2m$ vector $\mathbf{z}=\mathbf{U}(\mathbf{x}_1-\mathbf{x}_2)+\mathbf{D}(\mathbf{y}_1-\mathbf{y}_2)$. Since **(a)** $(\mathbf{x}_1,\mathbf{y}_1)\neq(\mathbf{x}_2,\mathbf{y}_2)$; **(b)** the entries of $\mathbf{U}(\mathbf{x}_1-\mathbf{x}_2)$ *do not overlap* with the entries of $\mathbf{D}(\mathbf{y}_1-\mathbf{y}_2)$ (due to matrices $\mathbf{U}$ and $\mathbf{D}$), it is $\mathbf{z}\neq\mathbf{0}$. Moreover, it is $\|\mathbf{z}\|\leq\beta\sqrt{2}=\delta\sqrt{2m}$. This is because its coordinates are between $-\delta$ and $+\delta$ (by the fact that $\mathbf{x}_1,\mathbf{x}_2,\mathbf{y}_1,\mathbf{y}_2$ have coordinates in $[\delta]$) and because $\mathbf{z}$ has dimension $2m$. Therefore $\mathbf{z}$ is a solution to the problem $\mathsf{SIS}_{q,2m,\beta\sqrt{2}}$. By Lemma 1 this gives a solution to GAPSVP_γ for polynomial γ, which, by Assumption 1, happens with probability $\mathsf{neg}(k)$. □

3 Main constructions

In this section we present our constructions. In Section 3.1, we derive an optimal (and parallel) secret memory checker. In Section 3.2, we construct a parallel reliable memory checker based on lattice assumptions.

3.1 Optimal online memory checking (secret memory)

Our first theorem can be stated as follows. Its detailed proof can be found in the Appendix, see Section 5.2.

Theorem 2 *There exists an online secret memory checker $\mathcal{C}(\Sigma,\mathbf{T},n)$ using memory words of $O(\log^2 n)$ size such that: **(1)** It is correct according to Definition 2 and ϵ-secure according to Definition 3 and assuming the existence of pseudorandom functions, where $\epsilon=\frac{1}{n^c}$ and $c>1$; **(2)** Its sequential query complexity is $O(\log n/\log\log n)$; **(3)** Its parallel query complexity is $O(1)$ using $O(\log n/\log\log n)$ processors in the EREW (exclusive read/exclusive write) model.*

Proof: (sketch) Our construction uses PRF tags [5] and a tree of $O(\log n/\log\log n)$ levels and $O(\log n)$ degree. In order to verify an operation (read or write), our checker's algorithms traverse $O(\log n/\log\log n)$ levels from the leaves up to the root of the tree, accessing necessary information at each level (e.g., PRF tags, counters). However, only a constant number of queries are issued per level (instead of $O(\log n)$ queries). To achieve that, we organize sibling nodes in a *single* word of $O(\log^2 n)$ size. We also make sure that the PRF we instantiate has range of polynonial size (and thus $\epsilon=\frac{1}{n^c}$), so that PRF tags can fit into a single memory word. Put together, such checker maintains small memory words and provides all its functionality (read/write) by issuing one query per level, thus achieving the desired complexity bound. □

Observation. It is worth noticing that the construction presented in Theorem 2 outperforms other constructions in the *memory checking model* with memory words of *polylogarithmic* size, where *query complexity* (see Definition 4) accounts for the number of queries on words of such size. This is the exact model for which the lower bound by Dwork et al. [9] was given. If one counts the number of bits that need to be accessed (i.e., bandwidth), then our construction has $O(\log n)$ complexity, just like previous constructions.

3.2 Parallel online memory checking (reliable memory)

We continue with our main contribution, i.e., a parallel checker that uses *only* reliable (but not secret) small memory. We first present some definitions and algebraic tools needed for proving important properties of our lattice-based function. Crucial for our constructions are *binary* and *radix-2* representations, used to represent a vector of big norm with a vector of shorter norm (but of slightly higher dimension). We note that these were independently used by Brakerski and Vaikuntanathan [6] in the context of fully-homomorphic encryption.

Definition 8 (Binary representation) *We denote with $\mathbf{b}(x)=[\mathbf{b}_0\,\mathbf{b}_1\,\ldots\,\mathbf{b}_{\log q-1}]^\mathsf{T}\in\{0,1\}^{\log q}$ the binary representation of $x\in\mathbb{Z}_q$, namely $x=\sum_{i=0}^{\log q-1}\mathbf{b}_i 2^i \mod q$.*

Definition 9 (Radix-2 representation) *Define* $\mathbf{r}(x) = [\mathbf{r}_0\ \mathbf{r}_1\ \ldots\ \mathbf{r}_{\log q-1}]^\top \in \mathbb{Z}_q^{\log q}$ *to be a radix-2 representation of* $x \in \mathbb{Z}_q$ *if and only if* $x = \sum_{i=0}^{\log q-1} \mathbf{r}_i 2^i \mod q$.

We note here, that while a binary representation is unique, a radix-2 representation is not. For example, for $q = 16$, $x = 7$, $\mathbf{r}(x)$ can be $[0\ 1\ 1\ 1]^\top$, $[0\ \ -2\ 0\ \ -1]^\top$ or $[-2\ 2\ 0\ \ -1]^\top$ (and other). However $\mathbf{b}(x)$ is always $[0\ 1\ 1\ 1]^\top$. We now give an important result:

Lemma 2 *For any* $x_1, x_2, \ldots, x_t \in \mathbb{Z}_q$ *there exists a radix-2 representation* $\mathbf{r}(.)$ *such that* $\mathbf{r}(x_1+x_2+\ldots+x_t \mod q) = \mathbf{b}(x_1) + \mathbf{b}(x_2) + \ldots + \mathbf{b}(x_t) \mod q$. *Also, it is* $\mathbf{r}(x_1 + x_2 + \ldots + x_t \mod q) \in \{0, \ldots, t\}^{\log q}$.

Lemma 2 is useful in the following sense: Given the binary representations of x_1 and x_2, namely b_1 and b_2, a radix-2 representation of $x_1 + x_2$ is $b_1 + b_2$. Note now that Definitions 8 and 9 and also Lemma 2 (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 $\mathbf{b}(\mathbf{x}_i)$ (or $\mathbf{r}(\mathbf{x}_i)$) in the resulting vector $\mathbf{b}(\mathbf{x})$ (or $\mathbf{r}(\mathbf{x})$). 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 exists a radix-2 representation* $\mathbf{r}(.)$ *such that* $\mathbf{r}(\mathbf{x}_1+\mathbf{x}_2+\ldots+\mathbf{x}_t \mod q) = \mathbf{b}(\mathbf{x}_1)+\mathbf{b}(\mathbf{x}_2)+\ldots+\mathbf{b}(\mathbf{x}_t) \mod q$. *Also, it is* $\mathbf{r}(\mathbf{x}_1+\mathbf{x}_2+\ldots+\mathbf{x}_t \mod q) \in \{0, \ldots, t\}^{k \log q}$.

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

Definition 10 (δ-admissible radix-2 representation) *Let* $\mathbf{x}$ *be a vector in* $\mathbb{Z}_q^k$. *We say that the radix-2 representation* $\mathbf{r}(\mathbf{x}) \in \mathbb{Z}_q^{k \log q}$ *is* δ-admissible *if and only if* $\mathbf{r}(\mathbf{x}) \in \{0, 1, \ldots, \delta\}^{k \log q}$.

Lattice-based Merkle tree. As we mentioned in the introduction, our core construction is a Merkle tree with different cryptography. Instead of employing a typical collision-resistant function such as SHA-2, it uses our lattice-based hash function $h_\delta(\mathbf{x}, \mathbf{y})$ from Definition 7. We refer to such a Merkle tree as *lattice-based Merkle tree* and to its digests as *lattice-based digests*.

Recall that our lattice-based hash function has domain $[\delta]^m \times [\delta]^m$ and range $\mathbb{Z}_q^k$. For our construction we set $\delta = n$, i.e., we allow the entries of the input vectors to be bounded by $n = \mathsf{poly}(k)$, the number of the indices of our memory. One of the main technical challenges in combining the idea of a Merkle tree with our lattice-based hash function is to assure that the output of the function $y = h_n(\mathbf{x}, \mathbf{y}) \in \mathbb{Z}_q^k$ can be appropriately prepared to become an input to the function again, namely a vector in $[n]^m$. We achieve that by transforming $y \in \mathbb{Z}_q^k$ to $\mathbf{r}(y) \in \mathbb{Z}_q^{k \log q}$, namely to an n-admissible radix-2 representation of y. This operation provides a representation with small entries, as required by the function definition.

We are now ready to describe our lattice-based Merkle tree in detail. Let $\mathbf{T}$ be the initial state of our table (memory), storing values $\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{n-1} \in \mathbb{Z}_q^k$. Let T be the binary tree of $\ell = \log n$ levels on top of the values $\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{n-1}$ 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, $i = 0, \ldots, n-1$, the lattice-based digest $d(v_i)$ is defined as $d(v_i) = \mathbf{x}_i$. For any internal node w, with left child u and right child v, we define its lattice-based digest by using the hash function $h_n(\mathbf{x}, \mathbf{y})$ given in Definition 7 and the transformation of the function range outlined above. Namely, the lattice-based digest $d(w)$ of node w is recursively defined as

$$d(w) = h_n(\mathbf{r}(d(u)), \mathbf{r}(d(v))) = \mathbf{L} \cdot \mathbf{r}(d(u)) + \mathbf{R} \cdot \mathbf{r}(d(v))\,, \tag{2}$$

where $\mathbf{r}(d(u))$ and $\mathbf{r}(d(v))$ are n-admissible radix-2 representations of $d(u)$ and $d(v)$. Note that our construction follows the logic of a plain Merkle tree that uses a collision-resistant function such as SHA-2, i.e., computation over the nodes of a binary tree. The difference is, however, that we choose "structure preserving" transformations (which are not feasible in other primitives such as SHA-2) to go from one level to another, a general property that we call *repeated linearity*. We present an abstraction of this propery in Section 5.6 of the Appendix.

Partial digests. In Relation 2 of the previous section, we showed that in order to compute the lattice-based digest $d(w)$ of some internal node w of tree T, we need to access the lattice-based digests of its *children*. However, due to the algebraic nature of the used hash function, we can express the digest $d(w)$, as well as the n-admissible radix-2 representation $\mathbf{r}(d(w))$ of it, somehow differently, namely as a sum of *explicit functions* (called here *partial digests*) of data at the leaves of the tree. We establish these expressions in Theorem 3 and Corollary 2. These are crucial for deriving our final results.

Let now $\mathsf{range}(w)$ be the range of successive indices corresponding to the leaves of the subtree of T rooted on w. E.g., in Figure 1 in the Appendix, it is $\mathsf{range}(v_{11}) = \{0, 1, 2, 3\}$. For every node $w \in T$ and for every $i \in \mathsf{range}(w)$ we define the *partial digest* $\mathcal{D}(w, \mathbf{x}_i)$ of w with reference to $\mathbf{x}_i$:

Definition 11 (Partial digests of a node w) *For a leaf node w of tree T storing value $\mathbf{x}_i$, the* partial digest *of w (wrt $\mathbf{x}_i$) is defined as $\mathcal{D}(w, \mathbf{x}_i) = \mathbf{x}_i$. For every other internal node w of tree T, with* left *child u and* right *child v, and for every $i \in \mathsf{range}(w)$, the* partial digest *$\mathcal{D}(w, \mathbf{x}_i)$ of w wrt $\mathbf{x}_i$ is recursively defined as*

$$\mathcal{D}(w, \mathbf{x}_i) = \begin{cases} \mathbf{L} \cdot \mathbf{b}(\mathcal{D}(u, \mathbf{x}_i)), & \textit{if } \mathbf{x}_i \textit{ belongs to the left subtree of } w\,, \\ \mathbf{R} \cdot \mathbf{b}(\mathcal{D}(v, \mathbf{x}_i)), & \textit{if } \mathbf{x}_i \textit{ belongs to the right subtree of } w\,. \end{cases}$$

E.g., in Figure 1 in the Appendix, the partial digests of root r with reference to $\mathbf{x}_1$ and $\mathbf{x}_2$ are $\mathcal{D}(r, \mathbf{x}_1) = \mathbf{R} \cdot \mathbf{b}(\mathbf{R} \cdot \mathbf{b}(\mathbf{L} \cdot \mathbf{b}(\mathbf{x}_1)))$ and $\mathcal{D}(r, \mathbf{x}_2) = \mathbf{R} \cdot \mathbf{b}(\mathbf{L} \cdot \mathbf{b}(\mathbf{R} \cdot \mathbf{b}(\mathbf{x}_2)))$ respectively, where $\mathbf{b}(z)$ is z's *binary representation*. We note that the left-to-right order of matrices $\mathbf{R}$ and $\mathbf{L}$ appearing in $\mathcal{D}(r, \mathbf{x}_i)$ coincides with the binary representation of i (0 is represented with $\mathbf{R}$ and 1 is represented with $\mathbf{L}$).

Theorem 3 *The lattice-based digest $d(w)$ of a tree node w (given in Relation 2) can be expressed as a sum of node w's partial digests wrt node w's range, i.e., $d(w) = \sum_{i \in \mathsf{range}(w)} \mathcal{D}(w, \mathbf{x}_i)$.*

Proof. (sketch) Apply Corollary 1. By induction, the sum of the partial digests of node w with reference to its range can be expressed as in Relation 2. □

By Corollary 1 and Theorem 3, we can derive a similar result for the radix-2 representation of node w:

Corollary 2 *Let $\lambda(w) = \sum_{i \in \mathsf{range}(w)} \mathbf{b}(\mathcal{D}(w, \mathbf{x}_i))$. Then $\lambda(w)$ is an n-admissible radix-2 representation of the lattice-based digest $d(w)$ of tree node w (given in Relation 2).*

3.3 Algorithms of the parallel checker

Setup. We now describe the algorithms of our parallel online memory checker $\mathcal{C}(\Sigma, \mathbf{T}, n)$. Our alphabet Σ contains words that are vectors in $\mathbb{Z}_q^k$, i.e., of *polynomial* bit-size ($O(k \log q)$). The algorithms that we are going to describe write (read) data to (from) either the unreliable or the reliable memory. We recall that for the query complexity, we are only interested in counting the *number of queries to the unreliable memory*.

Algorithm $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{genkey}(1^k)$**:** Call $\{q, m, \beta\} \leftarrow \mathsf{parameters}(k, n)$, on input the security parameter k and the size n of table $\mathbf{T}$. Then sample two matrices $\mathbf{L}, \mathbf{R} \in \mathbb{Z}_q^{k \times m}$ uniformly at random. Output $\mathsf{sk} = \emptyset$ and $\mathsf{pk} = \{\mathbf{L}, \mathbf{R}, q\}$.

Algorithm $(\mathcal{M}, \mu) \leftarrow \mathsf{setup}(\mathbf{T}, \mathsf{pk}, \mathsf{sk})$**:** Let $\mathbf{T}$ be the initial table, storing values $\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{n-1} \in \mathbb{Z}_q^k$ and T be the binary tree built on top on the values of table $\mathbf{T}$. The leaves of T store $\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{n-1} \in \mathbb{Z}_q^k$.

1. Store in unreliable memory $\mathcal{M}$ the labels $\lambda(w)$ for every node w of T, as computed in Corollary 2;
2. Store in reliable memory μ the lattice-based digest $d(r)$ of the root r of T, as computed in Theorem 3.

We continue with algorithms $\mathsf{secureRead}$ and $\mathsf{secureWrite}$, which are both parellelizable.

Reading in parallel. Algorithm secureRead is *exactly the same* as the verification algorithm used in a Merkle tree [17], performing fully-parallelizable computations along a certain tree path to verify whether the leaf value is correct or not. We give the algorithm in the following.

Algorithm $\{\alpha, \mathtt{reject}\} \leftarrow \mathsf{secureRead}(\mathsf{index}, \mathcal{M}, \mu, \mathsf{sk}, \mathsf{pk})$**:** Let $v_\ell, \ldots, v_1$ be the path in the lattice-based Merkle tree T from node v_ℓ (v_ℓ stores the value $\mathbf{T}[\mathsf{index}]$) to the child v_1 of the root r of T. Let also $w_\ell, \ldots, w_1$ be the *sibling nodes* of $v_\ell, \ldots, v_1$. Perform the following steps:

1. (computation from untrusted memory) Compute a value $y_i \in \mathbb{Z}_q^k$ as follows:

$$y_i = \begin{cases} \mathbf{T}[\mathsf{index}], & \text{if } i = \ell\,, \\ \mathbf{L} \cdot \lambda(v_{i+1}) + \mathbf{R} \cdot \lambda(w_{i+1}), & \text{if } 0 \le i < \ell \text{ and } v_{i+1} \text{ is } v_i\text{'s left child}\,.^6 \end{cases}$$

2. (verification) For $i = \ell, \ldots, 1$, if $\lambda(v_i)$ is not a radix-2 representation of y_i **or** $\lambda(v_i), \lambda(w_i)$ have entries greater than n output $\mathtt{reject}$. If $y_0 \neq \mu$, output $\mathtt{reject}$. Output $\alpha = \mathbf{T}[\mathsf{index}]$.

Writing in parallel. Algorithm secureWrite is *similar* with an update algorithm in a Merkle tree. The crucial difference is that *updating a label of a node w in a lattice-based Merkle tree does not require updating the labels of w's children first*—these operations can happen in parallel (Relations 3 and 4). This is due to the properties of the lattice-based function. The detailed description of the algorithms follows:

Algorithm $\{(\mathcal{M}', \mu'), \mathtt{reject}\} \leftarrow \mathsf{secureWrite}(\mathsf{index}, \beta, \mathcal{M}, \mu, \mathsf{sk}, \mathsf{pk})$**:** Let $v_\ell, \ldots, v_1$ be the path in the lattice-based Merkle tree T from node v_ℓ (v_ℓ stores the value $\mathbf{T}[\mathsf{index}]$) to the child v_1 of the root r of T. Output $\alpha \leftarrow \mathsf{secureRead}(\mathsf{index}, \mathcal{M}, \mu, \mathsf{sk}, \mathsf{pk})$. If $\alpha = \mathtt{reject}$, then output $\mathtt{reject}$. Else, let $\lambda(v_i)$ ($i = 1, \ldots, \ell$) be the values accessed by secureRead. Set $\mathbf{T}[\mathsf{index}] = \beta$ and

$$\lambda'(v_i) = \lambda(v_i) - \mathbf{b}(\mathcal{D}(v_i, \alpha)) + \mathbf{b}(\mathcal{D}(v_i, \beta)) \text{ for } i = \ell, \ell - 1, \ldots, 1\,, \quad (3)$$

$$\mu' = \mu - \mathcal{D}(r, \alpha) + \mathcal{D}(r, \beta)\,. \quad (4)$$

Update $\mathcal{M}$ to $\mathcal{M}'$ with the new labels $\lambda'(.)$ computed above. Finally, update μ to μ'.

Note that Relations 3 and 4 assure that the properties of Theorem 3 and Corollary 2 are maintained and therefore the new values $\lambda'(v_j)$ and μ' are correct. We now state our main result for this section:

Theorem 4 *Let k be the security parameter. There exists an online reliable memory checker $\mathcal{C}(\Sigma, \mathbf{T}, n)$ using memory words of polynomial size such that:* ***(1)*** *It is correct according to Definition 2 and ϵ-secure according to Definition 3 and assuming the hardness of GAPSVP_γ, where $\gamma = n \cdot k \cdot \mathsf{poly}(\log k)$ and $\epsilon = \mathsf{neg}(k)$;* ***(2)*** *Its sequential query complexity is $O(\log n)$;* ***(3)*** *Its parallel query complexity is $O(1)$ using $O(\log n)$ processors in the CREW (concurrent read/exclusive write) model.*

Proof: (sketch) Both algorithms secureRead and secureWrite are parallelizable, since the execution of loop i is completely independent from the execution of the loop for $i + 1$. Also, correctness follows from the code of algorithms secureRead and secureWrite: Algorithm secureWrite always updates the lattice-based digests in a way that these remain consistent with Corollary 3 and algorithm secureRead performs a typical Merkle tree verification. Finally, security is due to the security of the Merkle tree construction [17], and is based on the collision resistance of the used lattice-based hash function, which is due to the hardness of approximating GAPSVP_γ for polynomial γ (see Theorem 1). □

[6] If v_{i+1} is v_i's right child, we set $y_i = \mathbf{R} \cdot \lambda(v_{i+1}) + \mathbf{L} \cdot \lambda(w_{i+1})$.

4 Authenticated data structures with optimal updates

Apart from parallelism, our lattice-based memory checker possesses another desirable feature that emerges in an application domain: Updating the lattice-based hash of the root of the binary tree T (this hash is used for verification purposes) involves *two* operations (see Relation 4). Such an update complexity is essentially optimal, as opposed to the *logarithmic* complexity of conventional hash trees, e.g., [5]. Applications that index and process a large amount of data in external memory (disk)—therefore issuing multiple disk I/Os—can benefit significantly from this feature. In such scenarios, the difference between $O(1)$ I/Os and $O(\log n)$ I/Os is especially relevant since an I/O is orders of magnitude slower than an access to internal memory.[7]

Consider for example a typical authenticated data structures setting [23, 26]: A trusted authority (e.g., government) owns a very large, on-disk, database $\mathbf{T}$ to be queried by the citizens. However the government does not have the computational resources to support queries from such a big population and resorts to distributed untrusted cloud machines. Aiming to offer security guarantees to the citizens, the government signs and publishes the Merkle tree digest of the database $\mathbf{T}$ (this is the reliable memory μ of the checker) so that clients can use the digest (along with some proof returned by the cloud) for *verifying* answers to database queries executed in the cloud. The database is highly-dynamic, therefore the government should refresh this digest whenever there is an *update* to the database. Using our version of Merkle trees, the government needs to perform only *two* I/Os per update.

We have therefore designed a lattice-based authenticated data structure [26] for a database table indexed in external memory, implemented with a B-tree [8]. We recall that a B-tree is more or less similar to a binary tree with the difference that it is used to index data on disk: Its internal nodes, instead of two children, have B children, where B is a parameter representing the number of entries/pages that fit into one disk block. As such, the main overhead in the operation of a B-tree is accessing a block of B entries, which is equivalent to one I/O. We note here that another construction for verifiable B-trees using generic hash functions was also proposed by Li et al. [15]—however updates in their schemes require $\log_B N$ I/Os, where N is the number of indexed records, and all other operations are inherently sequential.

Our new external memory authenticated data structure is implemented using the lattice-based memory checker as a building block. However, in order to go from a binary tree to a B-tree, we need to slightly change our construction (see Appendix). Now, being a cryptographic construction, an authenticated data structure is described by a collection of algorithms $\{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$ (see [23]). In our application, algorithms genkey, setup are initially executed by the government, algorithm update refreshes the digest signature, algorithm verify is executed by the citizens and the remaining algorithms are executed by the untrusted cloud. Formal definitions of the above algorithms are given in the Appendix (also appeared in [23]). We now give our final result. Its proof can be found in the Appendix.

Theorem 5 *Let k be the security parameter. There exists an authenticated data structure scheme* $\{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$ *for an external memory dynamic table D of N entries, using blocks of $B = O(1)$ entries each, such that:* ***(1)*** *The scheme is correct according to Definition 13 and secure according to Definition 14 and assuming the hardness of* GAPSVP_γ*, where $\gamma = N \cdot k \cdot \mathsf{poly}(\log k)$;* ***(2)*** *The I/O complexity of* setup *is $O(\frac{N}{B}\log_B N)$ or $O(\frac{N}{B})$ using $O(\log_B N)$ processors in the CREW model, outputting an authenticated data structure* $\mathsf{auth}(D)$ *occupying $O(\frac{N}{B})$ blocks;* ***(3)*** *The I/O complexity of* update *is $O(1)$;* ***(4)*** *The I/O complexity of* refresh *is $O(\log_B N)$ or $O(1)$ using $O(\log_B N)$ processors in the CREW model;* ***(5)*** *The I/O complexity of* query *is $O(\log_B N)$ or $O(1)$ using $O(\log_B N)$ processors in the EREW model, outputting a proof $\Pi(q)$ for a query q occupying $O(\log_B N)$ blocks;* ***(6)*** *The I/O complexity of* verify *is $O(\log_B N)$ or $O(1)$ using $O(\log_B N)$ processors in the CRCW model.*

[7] From en.wikipedia.org/wiki/Hard_disk, for a typical $7,200$rpm desktop hard drive, average seek time is 9ms and average latency time is 4.17ms, making the total time for an I/O equal to 13.4ms.

References

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

[2] A. Banerjee, C. Peikert, and A. Rosen. Pseudorandom functions and lattices. Cryptology ePrint Archive, Report 2011/401, 2011. `http://eprint.iacr.org/`.

[3] S. Benabbas, R. Gennaro, and Y. Vahlis. Verifiable delegation of computation over large datasets. In *CRYPTO*, pages 111–131, 2011.

[4] M. Blum. Program result checking: A new approach to making programs more reliable. In *ICALP*, pages 1–14, 1993.

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

[6] Z. Brakerski and V. Vaikuntanathan. Efficient fully homomorphic encryption from (standard) lwe. *Electronic Colloquium on Computational Complexity (ECCC)*, 18:109, 2011.

[7] M. Chu, S. Kannan, and A. McGregor. Checking and spot-checking the correctness of priority queues. In *ICALP*, pages 728–739, 2007.

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

[9] 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.

[10] 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.

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

[12] 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.

[13] E. Hall and C. S. Julta. Parallelizable authentication trees. In *Proc. Seclected Areas in Cryptography (SAC)*, pages 95–109, 2005.

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

[15] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Dynamic authenticated index structures for outsourced databases. In *Proc. of ACM SIGMOD International Conference on Management of Data*, pages 121–132, 2006.

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

[17] R. C. Merkle. A certified digital signature. In *Advances in Cryptology—CRYPTO*, volume 435 of *Lecture Notes in Computer Science*, pages 218–238. Springer, 1989.

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

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

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

[21] R. Ostrovsky. Efficient computation on oblivious rams. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, STOC '90, pages 514–523, 1990.

[22] 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.

[23] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Optimal verification of operations on dynamic sets. In *CRYPTO*, pages 91–110, 2011.

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

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

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

5 Appendix

5.1 Detailed comparison with related work

In the table below, we compare sequential and parallel query complexities of memory checkers that have appeared in the literature, with our solutions. We denote with n the number of user-accessible indices and with d any function that is $O(n)$. Below, NA stands for *not applicable*, PRF stands for *pseudorandom function*, UOWHF stands for *universal one-way hash function* and CRHF stands for *collision-resistant hash function*. All solutions with $O(1)$ query complexity use $O(\log n)$ processors. Also, although the works [12, 13, 19] do not explicitly refer to memory checking, they could be straightforwardly applied therein.

work	**sequential query complexity**	**parallel query complexity**	**secret memory**	**cryptographic assumption**
Blum et al. [5]	$O(\log n)$ $O(\log n)$	NA NA	yes no	PRF UOWHF
Dwork et al. [9]	$O(d \log_d n)$	NA	yes	PRF
Naor and Nissim [19] Goodrich and Tamassia [12]	$O(\log n)$	NA	no	CRHF (generic)
Hall and Jutla [13]	$O(\log n)$	$O(1)$	yes	PRF
this	$\Theta(\log n / \log\log n)$ $O(\log n)$	$O(1)$ $O(1)$	yes no	PRF CRHF (lattice-based)

5.2 Optimal secret memory checker based on PRFs (proof of Theorem 2)

Our construction is a modification of the PRF construction from the paper of Blum et al. [5]. Let k be the security parameter, Σ be the memory alphabet and let U be a polynomially-bounded estimate of how many writes are going to be performed on the memory. We take the bit size of the word in Σ to be $O(\log^2 n)$. Let also S be a k-bit seed to be used for the pseudorandom function. The seed S is kept in secret memory. Finally, consider a tree T that has $O(\log n / \log\log n)$ levels and $O(\log n)$ degree (note that we can get exactly the similar results—with increased, still polylogarithmic, memory word size— even when the degree is $O(\mathsf{poly}(\log n))$) and is built on top of our table of values $\mathbf{T}$ and denote with ID the sets of all unique identifiers $\mathsf{id}(v)$ of the nodes v of the tree T and with DATA the set of values that can be written in the database index (we consider binary checkers but it can be extended to checkers of $\mathsf{poly}(\log n)$ bits). We use a pseudorandom function

$$f_S : \mathsf{DATA} \cup \{\mathbf{0}\} \times \mathsf{ID} \times \{0, \ldots, U\} \rightarrow \{0,1\}^L ,$$

where $L = O(\log n)$. For each leaf node v_i, set $\mathsf{data}(v_i) = \mathbf{T}[i] \in \mathsf{DATA}$, otherwise for all other nodes of the tree set $\mathsf{data}(v_i) = \mathbf{0}$. Define the pseudorandom function tag of a node v of the tree as

$$t(v) = f_S(\mathsf{data}(v), \mathsf{id}(v), \mathsf{cnt}(v)) ,$$

where $\mathsf{cnt}(v)$ is the number of times that values at the leaves of the subtree rooted on node v have been written (therefore, initially $\mathsf{cnt}(v) = 0$ for all nodes v of tree T). In local reliable memory, we also store $\mathsf{cnt}(r)$, where r is the root of T (note that $\mathsf{cnt}(r)$ need not be stored in secret memory).

Memory word organization and grouping. Our result is based on a clever organization of the memory words, inspired by I/O algorithms (e.g., B-trees). As we said before, the bit size of the memory word of our checker is $O(\log^2 n)$, meaning that our checker can read and write $O(\log^2 n)$-sized words with one query.

Let v be a node of our tree and let $u_1, u_2, \ldots, u_\ell$ be its children, where $\ell = O(\log n)$. At every node v we store in one word the following information

$$\mathsf{memory}(v) = [t(v), \mathsf{label}(v), t(u_1), \mathsf{label}(u_1), \ldots, t(u_\ell), \mathsf{label}(u_\ell)] \,,$$

where $\mathsf{label}(v) = (\mathsf{data}(v), \mathsf{id}(v), \mathsf{cnt}(v))$ (for a leaf node, there are no values for $t(u_i)$ and $\mathsf{label}(u_i)$). Note that $\mathsf{data}(v)$ can be represented with an $O(\log n)$ number of bits, $\mathsf{id}(v) \leq 2n$ (one identifier per tree node) and $\mathsf{cnt}(v) \leq U = \mathsf{poly}(n)$. Therefore $\mathsf{label}(v)$ can also be represented with an $O(\log n)$ number of bits. Since $\mathsf{memory}(v)$ consists of $O(\log n)$ pairs of labels and tags (note that tags also require an $O(\log n)$ number of bits, since the range of f_S is $\{0,1\}^L$ and $L = O(\log n)$), it follows that the bit size of $\mathsf{memory}(v)$ is $O(\log^2 n)$.

Reading and writing. In order to read the contents of location i, all the nodes v on the path from the root to location i along with the respective information $\mathsf{memory}(v)$ are accessed. For each internal node v on the path, verify that $\mathsf{cnt}(v)$ equals the sum of $\sum_{i=1}^{\ell} \mathsf{cnt}(u_i)$, where $u_1, u_2, \ldots, u_\ell$ are v's children. Moreover, the tags $t(v), t(u_1), t(u_2), \ldots, t(u_\ell)$ are verified as well by having the checker recomputing them (since it has access to the seed S). Since the levels of the tree T is $O(\log n / \log\log n)$ and the checker does one query per level (i.e., accessing $\mathsf{memory}(v)$), it follows that the read complexity of the described solution is $O(\log n / \log\log n)$. Note that reading can be parallelized easily as well, since all the checks performed at some node v require only local information (i.e., information $\mathsf{memory}(v)$). So we can read securely by employing $O(\log n / \log\log n)$ checkers, each one performing $O(1)$ queries.

In order to write into location i a new value $\mathbf{T}'[i]$, all the nodes v on the path from the root to location i and the respective information $\mathsf{memory}(v)$ are accessed. First, a verification of $\mathbf{T}[i]$ as above is performed. Then the PRF tags along the path need to be recomputed. For leaf v the new tag is $f_S(\mathbf{T}'[i], \mathsf{id}(v), \mathsf{cnt}(v)+1)$ whereas for the other nodes on the path v, the new tag is $f_S(\mathbf{0}, \mathsf{id}(v), \mathsf{cnt}(v)+1)$. This is also easily parallelizable: The computation of the tag $t(v)$ does not depend of the tags of any of v's children (unlike the constructions using UOWHFs). So we can write securely by employing $O(\log n / \log\log n)$ checkers, each one performing $O(1)$ queries. Note that there is no need for concurrent read, since no value needs to be read at the same time (unlike the construction of Theorem 4).

Finally, the correctness and security of the scheme is derived by the same arguments in [5], since we are only changing the organization of data into words and the algorithms to access them. Moreover, our checker is ϵ-secure, where $\epsilon = 1/n^c$, since the range of the PRF that we are using has bit size $O(\log n)$. We note that essentially, this construction resembles the optimized construction given by Dwork et al. [9] that has $O(d \log_d n)$ complexity, for $d = O(\log n)$. However, instead of forcing the checker to perform d queries per level, we group all the maximum possible amount of information into one memory word of $O(\log^2 n)$ size, achieving one query per level.

5.3 Adding binary representations (proof of Lemma 2)

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

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

The resulting vector is a radix-2 representation of

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

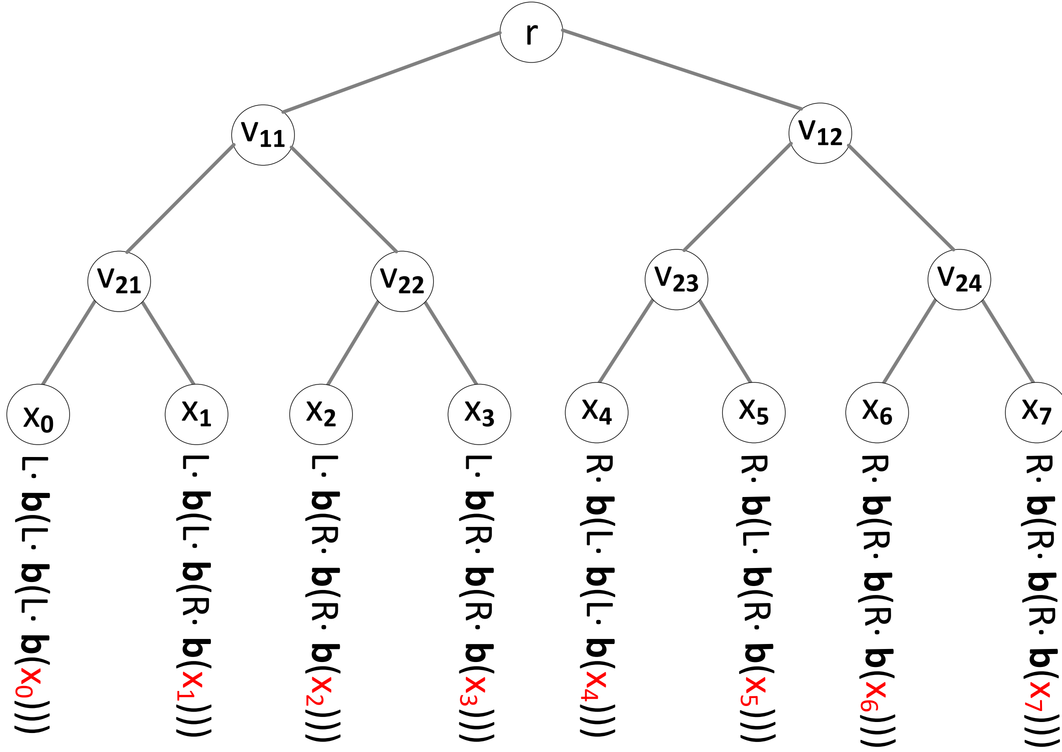

Figure 1: The lattice-based Merkle tree T built on top of a table with 8 values $\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_7$. At the leaves of the tree we show the partial digests of the root r with respect to each one of the leaves $x_0, x_1, \ldots, x_{n-1}$. The digest of the lattice-based tree is the sum of these terms (see Theorem 3).

which can be written as

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

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

5.4 Expressing lattice-based digests as a sum of partial digests (proof of Theorem 3)

We prove the claim by induction on the levels of the tree T. For any internal node w 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 $\mathsf{range}(w)$) in the subtree rooted on w. Therefore

$$\mathcal{D}(w, \mathbf{x}_i) + \mathcal{D}(w, \mathbf{x}_j) \quad = \quad \mathbf{L} \cdot \mathbf{b}(\mathbf{x}_i) + \mathbf{R} \cdot \mathbf{b}(\mathbf{x}_j) = d(w) \, .$$

This is due to Relation 2 and also due to the fact that $\mathbf{r}(.)$ can be picked to be $\mathbf{b}(.)$, which is an n-admissible radix-2 representation, therefore satisfying the constraint of the inputs of Definition 7. Hence the base case

holds. Assume the theorem holds for any internal node z that lies at level $0 < t+1 \leq \ell$. Therefore

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

Let w be an internal node that lies at level t and let $i_1, i_2, \ldots, i_q$ be the indices in $\mathsf{range}(w)$ in *sorted* order. Let u be the left child of w and v be the right child of w. Then, by the definition of the partial digest of the node w (Definition 11) we can write

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

By Corollary 1 there exist $\mathbf{r}(.)$ representations whose entries are at most $q/2 \leq n$ such that

$$d(w) = \mathbf{L} \cdot \mathbf{r}\left(\sum_{j=1}^{q/2} \mathcal{D}(u, \mathbf{x}_j)\right) + \mathbf{R} \cdot \mathbf{r}\left(\sum_{j=q/2+1}^{u} \mathcal{D}(v, \mathbf{x}_j)\right)\,.$$

By the inductive step this can be written as

$$d(w) = \mathbf{L} \cdot \mathbf{r}(d(u)) + \mathbf{R} \cdot \mathbf{r}(d(v))\,,$$

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

5.5 Parallel reliable memory checker based on lattices (proof of Theorem 4)

Complexity. For both algorithms $\mathsf{secureRead}$ and $\mathsf{secureWrite}$ the following are true: (a) since $\ell = O(\log n)$, the number of the queries performed to the unreliable public memory is $O(\log n)$; (b) the execution of loop i is completely independent from the execution of the loop for $i+1$; (c) for each $i \leq \ell$, exactly $O(1)$ queries to the unreliable memory are performed. Specifically, for the case of $\mathsf{secureWrite}$, this is the case since the binary representation of the partial digest $\mathbf{b}(\mathcal{D}(v_i, \alpha))$ (see Relations 3 and 4) can be computed with a *single* query to the unreliable memory, namely a query for α (all the other information for its computation is fixed at reliable memory, e.g., matrices $\mathbf{L}$ and $\mathbf{R}$). Therefore, both algorithms are parallelizable using $O(\log n)$ processors, each one executing $O(1)$ queries to the unreliable memory. Finally, the algorithms require *concurrent* write only to the reliable memory, since all the processors need to write to a location storing the `reject` bit concurrently. However, the queries to the unreliable memory can be implemented in parallel only with exclusive write (EW), since Relation 3 writes on different memory locations.

Correctness. Let $\mathbf{T}$ be any table of n entries. Fix the security parameter k and output sk and $\mathsf{pk} = (\mathbf{L}, \mathbf{R}, q)$ by calling algorithm genkey. Then output the unreliable memory $\mathcal{M}$ and the respective reliable memory μ, by calling algorithm setup. We recall that the unreliable memory $\mathcal{M}$ stores the lattice-based Merkle tree T of ℓ levels. Pick a polynomial number of updates—namely, pick a polynomial number of pairs of indices and values—and update $\mathcal{M}$ and μ by calling algorithm $\mathsf{secureWrite}$. Let $\mathbf{T}$ be the final table, $\mathcal{M}$ be the produced

unreliable memory and μ be the final reliable memory. Let index be an index and let $\alpha = \mathbf{T}[\mathsf{index}]$ be the *correct* value of index index. To prove correctness, we must show that, with all but negligible probability[8], $\alpha \leftarrow \mathsf{secureRead}(\mathsf{index}, \mathcal{M}, \mu, \mathsf{sk}, \mathsf{pk})$. The only way this could fail is for algorithm secureRead to reject at Item 2. We prove that this cannot be the case if algorithms genkey, setup, secureWrite are executed honestly. First, note that algorithm secureRead accesses pairs $\lambda(v_i), \lambda(w_i)$ $(i = \ell, \ell-1, \ldots, 1)$ of n-admissible representations, where $v_\ell, v_{\ell-1}, \ldots, v_1$ are the nodes on the path from index index (i.e., node v_ℓ) to the first child v_1 of the root of the tree T and $w_\ell, w_{\ell-1}, \ldots, w_1$ are their respective siblings. By the way $\lambda(v_i), \lambda(w_i)$ $(i = \ell, \ell-1, \ldots, 1)$ have been computed by the initial call to setup and the subsequent calls to secureWrite, the following are true:

1. $\lambda(v_\ell) = \mathbf{b}(\alpha)$ (definition of a leaf digest);
2. $d(v_i) = \mathbf{L}\lambda(v_{i+1}) + \mathbf{R}\lambda(w_{i+1})$ or $d(v_i) = \mathbf{R}\lambda(v_{i+1}) + \mathbf{L}\lambda(w_{i+1})$—according to left child or right child relation—, for $i = \ell-1, \ldots, 0$ and where v_0 is the root of the tree. This follows by Relation 2 and Corollary 2;
3. For $i = \ell, \ldots, 0$, the representations in $\lambda(v_i)$ accessed by secureRead are always n-admissible radix-2 representations of $d(v_i)$, since secureWrite always updates $\lambda(v_i)$ through Relation 3 so that the invariant of Corollary 2 is maintained.

Based on the above observations, and the code of secureRead, we conclude that if α is the correct value stored at index index, even after updates, then α is always output by $\mathsf{secureRead}(\mathsf{index}, \mathcal{M}, \mu, \mathsf{sk}, \mathsf{pk})$, namely Item 2 of secureRead never rejects.

Security.

1. *Initialization and updates.* Fix the security parameter k and output sk and $\mathsf{pk} = (\mathbf{L}, \mathbf{R}, q)$ by calling algorithm genkey. Let Adv be a polynomially-bounded adversary. Adv picks an initial table $\mathbf{T}_0$ of n entries and sends it to the challenger Chal. Chal outputs the unreliable memory $\mathcal{M}_0$ (storing a lattice-based tree T of ℓ levels) and the respective reliable memory μ_0 by calling algorithm setup and sends the unreliable memory to the adversary. Then, for $t = 1, \ldots, h = \mathsf{poly}(k)$, the adversary Adv picks an index $0 \leq \mathsf{ind}_t \leq n-1$ and a value $\beta_t \in \Sigma$ and sends them to Chal. Chal outputs the final values of $\mathcal{M}_t$ and μ_t by calling algorithm secureWrite (namely the adversary chooses arbitrary write queries to the public memory). Note that the adversary, at each point can tamper with $\mathcal{M}_t$ but not with μ_t.
2. *Forge.* Let ind be a query index and $\alpha \neq \mathbf{T}[\mathsf{ind}]$ be an incorrect value for index ind picked by Adv (as in the security definition, see Definition 3). Let also $v_\ell, v_{\ell-1}, \ldots, v_0$ be the path of T from the node referring to index ind to the root of T. We prove, that for all polynomial $t \geq 0$ and for all $0 \leq \mathsf{ind} \leq n-1$, the probability that $\alpha \leftarrow \mathsf{secureRead}(\mathsf{ind}, \mathcal{M}_t, \mu_t, \mathsf{sk}, \mathsf{pk})$ while $\alpha \neq \mathbf{T}[\mathsf{ind}]$ is negligible. To proceed with the proof, we recall that algorithm secureRead accesses pairs $\lambda(v_i), \lambda(w_i)$ $(i = \ell, \ell-1, \ldots, 1)$, where $v_\ell, v_{\ell-1}, \ldots, v_1$ are the nodes on the path from index ind (i.e., node v_ℓ) to the first child v_1 of the root of the tree T and $w_\ell, w_{\ell-1}, \ldots, w_1$ are their respective siblings. These pairs reside in public unreliable memory $\mathcal{M}_t$ and are completely controlled by the adversary. We prove our claim with induction on t:

   Base case. First, we prove our claim for $t = 0$: To do that, we define the following events, related to the values of $\lambda(v_i), \lambda(w_i)$, as output by the adversary. Our goal will be to express the probability that $\alpha \leftarrow \mathsf{secureRead}(\mathsf{ind}, \mathcal{M}_0, \mu_0, \mathsf{sk}, \mathsf{pk})$ while $\alpha \neq \mathbf{T}[\mathsf{ind}]$ (for all $0 \leq \mathsf{ind} \leq n-1$) as a function of the following events. Initially, μ_0 is the *correct* digest of the initial lattice-based Merkle tree:

   (a) $\mathcal{E}_{\ell,0}$: The value $\lambda(v_\ell)$ picked by Adv is such that $\lambda(v_\ell)$ is an n-admissible radix-2 representation of $\alpha \neq \mathbf{T}[\mathsf{ind}]$;

[8] In our scheme, correctness holds with probability 1.

(b) $\mathcal{E}_i$: For $i = \ell - 1, \ldots, 1$, the values $\lambda(v_i)$ and $\lambda(v_{i+1}), \lambda(w_{i+1}) \in \{0, 1, \ldots, n\}^{k \log q}$ picked by Adv are such that $\lambda(v_i)$ *is* an n-admissible radix-2 representation of

$$\mathbf{L} \cdot \lambda(v_{i+1}) + \mathbf{R} \cdot \lambda(w_{i+1} .$$

Assume, without loss of generality that a convenient index $\mathsf{ind} = 0$ is used so that the order of $\mathbf{L}$ and $\mathbf{R}$ is always the same (the proof is independent of ind). This event can be partitioned into two mutually exclusive events, i.e., $\mathcal{E}_i = \mathcal{E}_{i,0} \cup \mathcal{E}_{i,1}$ such that

- $\mathcal{E}_{i,0}$: Value $\lambda(v_i)$ is *not* an n-admissible radix-2 representation of the lattice-based digest of node v_i, as defined in Relation 2;
- $\mathcal{E}_{i,1}$: Value $\lambda(v_i)$ is an n-admissible radix-2 representation of the lattice-based digest of node v_i, as defined in Relation 2.

(c) $\mathcal{E}_{0,1}$: The values $\lambda(v_1) \in \{0, 1, \ldots, n\}^{k \log q}$ and $\lambda(w_1) \in \{0, 1, \ldots, n\}^{k \log q}$ picked by Adv are such that

$$\mu_0 = \mathbf{L} \cdot \lambda(v_1) + \mathbf{R} \cdot \lambda(w_1).$$

The probability that $\alpha \leftarrow \mathsf{secureRead}(\mathsf{ind}, \mathcal{M}_0, \mu_0, \mathsf{sk}, \mathsf{pk})$, while $\alpha \neq \mathbf{T}[\mathsf{ind}]$ is the probability

$$\begin{aligned} & \Pr[\mathcal{E}_{\ell,0} \cap \mathcal{E}_{\ell-1} \cap \mathcal{E}_{\ell-2} \cap \ldots \cap \mathcal{E}_{0,1}] \\ = \; & \Pr[\mathcal{E}_{\ell,0} \cap (\mathcal{E}_{\ell-1,0} \cup \mathcal{E}_{\ell-1,1}) \cap (\mathcal{E}_{2,0} \cup \mathcal{E}_{2,1}) \cap \ldots \cap \mathcal{E}_{0,1}] \\ \leq \; & \Pr[\mathcal{E}_{\ell,0} | \mathcal{E}_{\ell-1,1}] + \Pr[\mathcal{E}_{\ell-1,0} | \mathcal{E}_{\ell-2,1}] + \Pr[\mathcal{E}_{\ell-2,0} | \mathcal{E}_{\ell-3,1}] + \ldots + \Pr[\mathcal{E}_{1,0} | \mathcal{E}_{0,1}] \\ = \; & \sum_{i=1}^{\ell} \Pr[\mathcal{E}_{i,0} | \mathcal{E}_{i-1,1}] . \end{aligned}$$

Note that the event $\mathcal{E}_{i,0} | \mathcal{E}_{i-1,1}$ implies the following:

(a) $\lambda(v_i)$ is not an n-admissible radix-2 representation of $d(v_i)$;
(b) $\lambda(v_{i-1})$ is an n-admissible radix-2 representation of $d(v_{i-1})$, where $d(v_{i-1}) = \mathbf{L} \cdot \lambda(v_i) + \mathbf{R} \cdot \lambda(w_i)$.

However, from Relation 2, it should be that

$$d(v_{i-1}) = \mathbf{L} \cdot \mathbf{r}(d(v_i)) + \mathbf{R} \cdot \mathbf{r}(d(w_i)) ,$$

where $d(v_i)$ and $d(w_i)$ are the digests of nodes v_i and w_i respectively and $\mathbf{r}(d(v_i))$ and $\mathbf{r}(d(w_i))$ are n-admissible radix-2 representations of them. Therefore $(\lambda(v_i), \lambda(w_i))$ is a collision with the pair $(\mathbf{r}(d(v_i)), \mathbf{r}(d(w_i)))$, since $\lambda(v_i) \neq \mathbf{r}(d(v_i))$. Note now that by Theorem 1 and since all the parameters used have been output by algorithm parameters (called in algorithm genkey), the probability $\Pr[\mathcal{E}_{i,0} | \mathcal{E}_{i-1,1}]$ is $\mathsf{neg}(k)$, for all $i = \ell, \ell - 1, \ldots, 1$. Therefore the sum

$$\sum_{i=1}^{\ell} \Pr[\mathcal{E}_{i,0} | \mathcal{E}_{i-1,1}]$$

is also $\mathsf{neg}(k)$, since $\ell = O(\log n) = O(\log k)$. All the above claims hold assuming the hardness of GAPSVP_γ for polynomial values of γ (see Lemma 1) and specifically for

$$\gamma = \beta\sqrt{2} \cdot \sqrt{k} \cdot \mathsf{poly}(\log k) = \delta\sqrt{2m} \cdot \sqrt{k} \cdot \mathsf{poly}(\log k) = n \cdot k \cdot \mathsf{poly}(\log k) ,$$

since $\beta = n\sqrt{2m}$ and $m = 2k \log q$. This concludes the proof for the base case.

Inductive hypothesis. Let $(\mathsf{ind}_1, \beta_1), (\mathsf{ind}_2, \beta_2), \ldots, (\mathsf{ind}_{r+1}, \beta_{r+1})$ be the pairs of *update* indices and values chosen by the adversary, according to the security definition. Suppose our claim holds for $1 \leq t \leq r$, i.e., the probability that $\alpha \leftarrow \mathsf{secureRead}(\mathsf{ind}, \mathcal{M}_t, \mu_t, \mathsf{sk}, \mathsf{pk})$ while $\alpha \neq \mathbf{T}[\mathsf{ind}]$ (for all ind) is $\mathsf{neg}(k)$ and where $\mathcal{M}_t, \mu_t$ are output by $\mathsf{secureWrite}(\mathsf{ind}_t, \beta_t, \mathcal{M}_{t-1}, \mu_{t-1}, \mathsf{sk}, \mathsf{pk})$.

Inductive step. We prove our claim for $t = r + 1$: By the base case of the induction and the inductive hypothesis, $\mathbf{T}[\mathsf{ind}_t] \leftarrow \mathsf{secureRead}(\mathsf{ind}_t, \mathcal{M}_{t-1}, \mu_{t-1}, \mathsf{sk}, \mathsf{pk})$ with probability $1 - \mathsf{neg}(k)$, for all $t = 1, \ldots, r+1$. Note now that since $\mathsf{secureRead}(\mathsf{ind}_t, \mathcal{M}_{t-1}, \mu_{t-1}, \mathsf{sk}, \mathsf{pk})$ does not reject and outputs the correct value $\mathbf{T}[\mathsf{ind}_t]$, $\mathsf{secureWrite}(\mathsf{ind}_t, \beta_t, \mathcal{M}_{t-1}, \mu_{t-1}, \mathsf{sk}, \mathsf{pk})$ does not reject either[9] and outputs the *correct* lattice-based digest μ_t with probability $1 - \mathsf{neg}(k)$. This is because, by Relation 3, updating μ_{t-1} to μ_t depends only on the current value of the updated index (i.e., $\mathbf{T}[\mathsf{ind}_t]$) and the new value β_t to be written at ind_t, i.e.,

$$\mu_t = \mu_{t-1} - \mathbf{b}(\mathcal{D}(r, \mathbf{T}[\mathsf{ind}_t])) + \mathbf{b}(\mathcal{D}(r, \beta_t)) \,,$$

where r is the root of the tree T. Therefore for all $t = 1, \ldots, r + 1$, as long as

$$\mathsf{secureRead}(\mathsf{ind}_t, \mathcal{M}_{t-1}, \mu_{t-1}, \mathsf{sk}, \mathsf{pk})$$

does not reject, μ_t, output by $\mathsf{secureWrite}(\mathsf{ind}_t, \beta_t, \mathcal{M}_{t-1}, \mu_{t-1}, \mathsf{sk}, \mathsf{pk})$ is the *correct* lattice-based digest of the updated table $\mathbf{T}$. Therefore, we can follow exactly the same procedure with the base case of the induction to prove that the probability that $\alpha \leftarrow \mathsf{secureRead}(\mathsf{ind}, \mathcal{M}_{r+1}, \mu_{r+1}, \mathsf{sk}, \mathsf{pk})$ while $\alpha \neq \mathbf{T}[\mathsf{ind}]$ is negligible.

5.6 Repeated linearity property

One of the main properties of our lattice-based hash function is *additive homomorphism*, namely, for any eligible inputs $\mathbf{x}_1$ and $\mathbf{x}_2$, we have $h_n(\mathbf{x}_1, \mathbf{x}_2) = \mathbf{L} \cdot \mathbf{x}_1 + \mathbf{R} \cdot \mathbf{x}_2$. However, this property by itself does not suffice for deriving our results and this is the reason why we could not employ other additively-homomorphic collision-resistant hash functions. The crucial additional property of our lattice-based hash function is what we call *repeated linearity*: Repeated linearity is the existence of an additively-homomorphic mapping of the range of our function back to elements of its domain. In our case, this mapping is a radix-2 representation. We express this property formally as follows. Let $(\mathbb{G}, +)$ and $(\Delta, +)$ be finite groups under addition.

Assume the following: **(a)** There exists a collision-resistant hash function $h : \Delta \times \Delta \rightarrow \mathbb{G}$ that is additively-homomorphic under some functions $\lambda_1 : \Delta \rightarrow \mathbb{G}$ and $\lambda_1 : \Delta \rightarrow \mathbb{G}$, i.e., for all $x_1, x_2 \in \Delta$, it is $h(x_1, x_2) = \lambda_1(x_1) + \lambda_2(x_2)$; **(b)** There exists an additively-homomorphic function[10] $g : \mathbb{G} \rightarrow \Delta$ such that for all $y_1, y_2 \in \mathbb{G}$, it is $g(y_1 + y_2) = g(y_1) + g(y_2)$.

Then we say that function h satisfies the repeated linearity property. Any function h satisfying the repeated linearity property could be used to derive our results. In this work, we have: $\mathbb{G} = \mathbb{Z}_q^k$; $\Delta = [n]^{k \log q}$; $h(\mathbf{x}_1, \mathbf{x}_2) = \mathbf{L} \cdot \mathbf{x}_1 + \mathbf{R} \cdot \mathbf{x}_2$; and $g(\mathbf{x}) = \mathbf{r}(\mathbf{x})$ is the radix-2 representation function[11] whose additive property is established in Corollary 1.

5.7 Authenticated data structures definitions

We first give with the authenticated data structures definition, as appeared in [23]:

[9] This is because the first thing $\mathsf{secureWrite}$ does is calling $\mathsf{secureRead}$ in order to verify the current value of the index it is updating.

[10] These functions should be "at least 1-1", namely, one input should map to at least one output.

[11] Function $\mathbf{r}(.)$ outputs radix-2 representations of elements in $\mathbb{Z}_q^k$ with entries less than or equal to n.

Definition 12 (ADS scheme) *Let D be* any *data structure that supports queries q and updates u. Let* $\mathsf{auth}(D)$ *denote the resulting authenticated data structure and d the digest of the authenticated data structure, i.e., a constant-size description of D. An ADS scheme $\mathcal{A}$ is a collection of the following six probabilistic polynomial-time algorithms:*

1. $\{\mathsf{sk}, \mathsf{pk}\} \leftarrow \mathsf{genkey}(1^k)$*: On input the security parameter k, it outputs a secret key* sk *and a public key* pk*;*
2. $\{\mathsf{auth}(D_0), d_0\} \leftarrow \mathsf{setup}(D_0, \mathsf{sk}, \mathsf{pk})$*: On input a (plain) data structure D_0 and the secret and public keys, it computes the authenticated data structure* $\mathsf{auth}(D_0)$ *and the respective digest d_0 of it;*
3. $\{D_{h+1}, \mathsf{auth}(D_{h+1}), d_{h+1}, \mathsf{upd}\} \leftarrow \mathsf{update}(u, D_h, \mathsf{auth}(D_h), d_h, \mathsf{sk}, \mathsf{pk})$*: On input an update u on data structure D_h, the authenticated data structure* $\mathsf{auth}(D_h)$*, the digest d_h, and the secret and public keys, it outputs the updated data structure D_{h+1} along with the updated authenticated data structure* $\mathsf{auth}(D_{h+1})$*, the updated digest d_{h+1} and some relative information* upd*;*
4. $\{D_{h+1}, \mathsf{auth}(D_{h+1}), d_{h+1}\} \leftarrow \mathsf{refresh}(u, D_h, \mathsf{auth}(D_h), d_h, \mathsf{upd}, \mathsf{pk})$*: On input an update u on data structure D_h, the authenticated data structure* $\mathsf{auth}(D_h)$*, the digest d_h, relative information* upd *(output by* update*), and the public key, it outputs the updated data structure D_{h+1} along with the updated authenticated data structure* $\mathsf{auth}(D_{h+1})$ *and the updated digest d_{h+1};*
5. $\{\Pi(q), \alpha(q)\} \leftarrow \mathsf{query}(q, D_h, \mathsf{auth}(D_h), \mathsf{pk})$*: On input a query q on data structure D_h, the authenticated data structure* $\mathsf{auth}(D_h)$ *and the public key, it returns the answer $\alpha(q)$ to the query, along with a proof $\Pi(q)$;*
6. $\{\mathtt{accept}, \mathtt{reject}\} \leftarrow \mathsf{verify}(q, \alpha, \Pi, d_h, \mathsf{pk})$*: On input a query q, an answer α, a proof Π, a digest d_h and the public key, it outputs either* $\mathtt{accept}$ *or* $\mathtt{reject}$.

Let $\{\mathtt{accept}, \mathtt{reject}\} \leftarrow \mathsf{check}(q, \alpha, D_h)$ be an algorithm that decides whether α is a correct answer for query q on data structure D_h (check is not part of the definition of an ADS scheme). There are two properties that an ADS scheme should satisfy, namely *correctness* and *security* (intuition follows from signature schemes definitions).

Definition 13 (Correctness) *Let $\mathcal{ASC}$ be an ADS scheme* $\{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$. *We say that the ADS scheme $\mathcal{ASC}$ is* correct *if, for all $k \in \mathbb{N}$, for all* $\{\mathsf{sk}, \mathsf{pk}\}$ *output by algorithm* genkey*, for all $D_h, \mathsf{auth}(D_h), d_h$ output by one invocation of* 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 algorithm* $\mathsf{check}(q, \alpha(q), D_h)$ *outputs* $\mathtt{accept}$*, so does algorithm* $\mathsf{verify}(q, \Pi(q), \alpha(q), d_h, \mathsf{pk})$.

Definition 14 (Security) *Let $\mathcal{ASC}$ be an ADS scheme* $\{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$*, k be the security parameter, $\nu(k)$ be a negligible function and* $\{\mathsf{sk}, \mathsf{pk}\} \leftarrow \mathsf{genkey}(1^k)$. *Let also* Adv *be a probabilistic polynomial-time adversary that is only given* pk. *The adversary has unlimited access to all algorithms of $\mathcal{ASC}$, except for algorithms* setup *and* update *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* setup. *Then, for $i = 0, \ldots, h = \mathsf{poly}(k)$,* Adv *issues an update u_i in the data structure D_i and computes D_{i+1},* $\mathsf{auth}(D_{i+1})$ *and d_{i+1} through oracle access to algorithm* update. *Finally the adversary picks an index $0 \leq t \leq h+1$, and computes a query q, an answer α and a proof Π. We say that the ADS scheme $\mathcal{ASC}$ is* secure *if for all $k \in \mathbb{N}$, for all* $\{\mathsf{sk}, \mathsf{pk}\}$ *output by algorithm* genkey*, and for any probabilistic polynomial-time adversary* Adv *it holds that*

$$\Pr\left[\begin{array}{ll} \{q, \Pi, \alpha, t\} \leftarrow \mathsf{Adv}(1^k, \mathsf{pk}); & \mathtt{accept} \leftarrow \mathsf{verify}(q, \alpha, \Pi, d_t, \mathsf{pk}); \\ & \mathtt{reject} \leftarrow \mathsf{check}(q, \alpha, D_t). \end{array}\right] \leq \nu(k)\,. \tag{5}$$

5.8 External memory authenticated B-tree with optimal updates (proof of Theorem 5)

Let $\mathcal{C}(\Sigma, \boldsymbol{T}, n) = \{\mathsf{genkey}, \mathsf{setup}, \mathsf{secureRead}, \mathsf{secureWrite}\}$ be the lattice-based memory checker derived in our main contribution. Security and correctness of the authenticated table follows directly from the security and correctness of $\mathcal{C}(\Sigma, \boldsymbol{T}, n)$.

Overview of the construction. We build a B-tree [8] T on top of table $\mathbf{T}$. We recall that the B-tree is a balanced tree that has internal degree B and height $O(\log_B N)$. The internal nodes of the tree contain B entries and are usually stored in one disk block. Now, algorithm $\mathsf{genkey}()$ works as follows: On input the security parameter k, it computes q as before, for some $\delta = N = \mathsf{poly}(k)$. Then it samples $B = O(1)$ matrices $\mathbf{L}_i \in \mathbb{Z}_q^{k\times m}$ uniformly at random ($i = 1, \ldots, B$). It outputs an empty secret key sk and $\mathsf{pk} = \{\mathbf{L}_1, \mathbf{L}_2, \ldots, \mathbf{L}_B, q\}$. Note that instead of two matrices $\mathbf{L}$ and $\mathbf{R}$, we have set B matrices $\mathbf{L}_1, \mathbf{L}_2, \ldots, \mathbf{L}_B$, to compensate for the increased internal degree.

Let now u be an internal node of the B-tree T, that has children $v_1, v_2, \ldots, v_B$. By following the same techniques as in the main section of the paper, but expanded for internal nodes of degree B, we can recursively define the digest $d(u)$ of node u of the B-tree by using a lattice-based hash function that takes B inputs instead of two inputs (see Definition 7), as $d(u) = \sum_{i=1}^{B} \mathbf{L}_i \mathbf{r}(d(v_i))$, where $\mathbf{r}(d(v_i))$ are n-admissible radix-2 representations of $d(v_i)$ ($i = 1, \ldots, B$).

We can now naturally define the partial digest of a node u in the B-tree T, as in Definition 11 (which was for a binary tree):

Definition 15 (Partial digest for a B-tree 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 $\mathcal{D}(u, \mathbf{x}_i) = \mathbf{x}_i$. Else, for every other node u of T, and for every $i \in \mathsf{range}(u)$, the* partial digest *$\mathcal{D}(u, \mathbf{x}_i)$ of u with reference to $\mathbf{x}_i$ is recursively defined as $\mathcal{D}(u, \mathbf{x}_i) = \mathbf{L}_j \mathbf{b}(d(v_j, \mathbf{x}_i))$, if $\mathbf{x}_i$ belongs to the j-th subtree of u and where v_j is the j-th child of u.*

Having a new definition for the partial digest on a B-tree, we can express the digest $d(u)$ that we showed above as a function (sum) of the partial digests, and exactly as in Theorem 3. Moreover, the state (digest) of the external memory authenticated table can be updated again by using Relation 3 and with only two additions of vectors in $\mathbb{Z}_q^k$.

Complexity of algorithm setup **and group complexity of** $\mathsf{auth}(D)$**.** The algorithm needs to compute the N-admissible radix-2 representations $\lambda(u)$ of digests $d(u)$ for every internal node u of the tree T. Note that by Corollary 2, there are $N/B, N/B^2, N/B^3, \ldots, B$ such representations that need to be computed for levels $\log_B N - 1, \log_B N - 2, \log_B N - 3, \ldots, 1$ respectively, each one being the sum of $B, B^2, B^3 \ldots, N/B$ binary representations respectively, i.e.,

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

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

$$\frac{N}{B} + B\frac{N}{B^2} + B^2\frac{N}{B^3} + \ldots + \frac{N}{B} = O\left(\frac{N}{B}\log_B N\right) \,.$$

Note now in the CREW model, we can use $O(\log_B N)$ processors, i.e., one processor for each level of the tree. By reading the values $\mathbf{x}_i$ concurrently and writing the values $\lambda(u)$ at different memory locations, it follows that each processor will have to do $O(N/B)$ accesses 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 $\lambda(u)$. Therefore the authenticated data structure $\mathsf{auth}(D)$ occupies has $O(N/B)$ blocks.

Complexity of algorithm update**.** Follows from the complexity of secureWrite and from the fact that update needs to update only $\lambda(r)$, where r is the root of the tree. In other words, update does not update the whole authenticated data structure, but just the digest.

Complexity of algorithm refresh**.** Follows from the complexity of secureWrite.

Complexity of algorithms query **and** verify **and size of** $\Pi(q)$**.** Follow from the complexity of secureRead.