Algorithmica
DOI 10.1007/s00453-014-9968-3

Authenticated Hash Tables Based on Cryptographic Accumulators

Charalampos Papamanthou · Roberto Tamassia · Nikos Triandopoulos

Received: 9 November 2010 / Accepted: 27 December 2014

Abstract Suppose a client stores n elements in a hash table that is outsourced to an untrusted server. We address the problem of authenticating the hash table operations, where the goal is to design protocols capable of verifying the correctness of queries and updates performed by the server, thus ensuring the integrity of the remotely stored data across its entire update history. Solutions to this authentication problem allow the client to gain trust in the operations performed by a faulty or even malicious server that lies outside the administrative control of the client. We present two novel schemes that implement an *authenticated hash table*. An authenticated hash table exports the basic hash-table functionality for maintaining a dynamic set of elements,

A preliminary version of this paper [44] was presented at the 15th ACM Conference on Computer and Communications Security (CCS). This research was supported by the U.S. National Science Foundation under grants CNS-1012060, CNS-1012798, CNS-1012910, CNS-1228485, IIS-0713403 and OCI-0724806, by the Center for Geometric Computing and the Kanellakis Fellowship at Brown University, by a gift from NetApp, Inc., and by the Center for Algorithmic Game Theory at the University of Aarhus under an award from the Carlsberg Foundation. The views in this paper do not necessarily reflect the views of the sponsors. We thank C. Christopher Erway, Michael Goodrich, Anna Lysyanskaya, Dimitrios Papadopoulos, John Savage, Edward Tremel and Ioannis Vergados for many useful discussions. We also thank Ivan Damgård for giving us important feedback on the first version of this work.

C. Papamanthou (✉)
Department of Electrical and Computer Engineering and Institute of Advanced Computer Studies (UMIACS), University of Maryland, College Park, MD, USA
e-mail: cpap@umd.edu

R. Tamassia
Department of Computer Science, Brown University, Providence, RI, USA
e-mail: rt@cs.brown.edu

N. Triandopoulos
RSA Laboratories and Department of Computer Science, Boston University, Boston, MA, USA
e-mail: nikolaos.triandopoulos@rsa.com

Published online: 09 January 2015

coupled with the ability to provide short cryptographic proofs that a given element is a member or not of the current set. By employing efficient algorithmic constructs and cryptographic accumulators as the core security primitive, our schemes provide constant proof size, constant verification time and sublinear query or update time, strictly improving upon previous approaches. Specifically, in our first scheme which is based on the RSA accumulator, the server is able to construct a (non-)membership proof in constant time and perform updates in $O(n^{\epsilon} \log n)$ time for any fixed constant $0 < \epsilon < 1$. A variation of this scheme achieves a different trade-off, offering constant update time and $O(n^{\epsilon})$ query time. Our second scheme uses an accumulator based on bilinear pairings to achieve $O(n^{\epsilon})$ update time at the server while keeping all other complexities constant. A variation of this scheme achieves $O(n^{\epsilon} \log n)$ time for queries and constant update time. An experimental evaluation of both solutions shows their practicality.

Keywords Authenticated data structures · Cryptographic accumulators · Cloud computing security

1 Introduction

Storing data in the "cloud" (e.g., Amazon's S3 storage service) is a fast growing computing trend for both corporations and consumers. Clients create virtual drives consisting of online storage units that are operated by remote and geographically dispersed servers. In an adversarial data-outsourcing setting, however, we would like to be able to detect both data corruption caused by a faulty server (e.g., because of hardware issues or software errors) and data tampering performed by an attacker that compromises the server (e.g., deliberate deletion or modification of files). Specifically, answers to client queries should be efficiently verified and either validated to be correct or rejected because they do not reflect the true state of the client's outsourced data.

In this paper, we study the fundamental problem of verifying membership and non-membership queries over a dynamic set of n data elements which is stored in a hash table that is maintained by an untrusted server. Used by numerous applications, hash tables are simple and efficient data structures for answering set-membership queries optimally, in expected constant time—it is therefore important in practice to authenticate their functionality and, therefore, broaden their applicability to data-outsourcing settings in a secure and verifiable manner.

We reach this goal by designing two schemes for an *authenticated hash table*. An authenticated hash table, in addition to the basic functionality of maintaining a dynamic set of elements, exports cryptographic proofs about whether a given element belongs or not in the current set that the hash table stores. Following a standard approach, we augment the hash table with an authentication structure that uses a cryptographic primitive to define a succinct (e.g., a few bytes long) and secure *digest*, a "fingerprint" of the entire stored set. Computed on the correct data, this digest will serve as a secure set description subject to which the answer to any (non-)membership query can be verified by the client by means of a corresponding proof that is provided by the server. Of course, what makes the problem hard is the need to efficiently support *updates*:

Otherwise, verifying (non-)membership in static sets can be optimally performed through per-element digital signatures, where in this case, any update in the set would require refreshing all stored such signatures.

Then our main design goal is to implement the above methodology both *securely*, to protect against attacks performed by a computationally-bounded server, and *efficiently*, with respect to the communication and computation overheads that are incurred to achieve the added security guarantee. In particular, we wish to minimize the size of the proof that the server sends to the client in order to demonstrate the correctness of the answer—ideally, we would like to keep this cost constant. Analogously, since client-side applications may connect to the server from thin-client devices with limited computing power and slow connectivity (e.g., smart-phones), we would like to make the verification computation performed by the client as efficient as possible, ideally with complexity that is independent of the size of set. More importantly, we wish to preserve the optimal query complexity of the hash table, while keeping the costs due to set updates sublinear in the set size: ideally, the server should authenticate (non-)membership queries in constant time, or otherwise we lose the optimal property that hash tables offer!

Developing secure protocols for hash tables that authenticate (non-)membership queries in constant time has been a long-standing open problem [36]. Using collision-resistant hashing and Merkle's tree construction [33] to produce the set digest, (non-)membership queries in sets can be authenticated with *logarithmic costs* (e.g., [6,23,36, 42,43,53]). One can achieve complexities better than logarithmic by using alternative cryptographic primitives. One-way accumulators and their dynamic extensions [2,5, 10,11,37] are constructions for accumulating a set of n elements into a short value, subject to which each accumulated element has a short witness (proof) that can be used to verify in *constant time* its membership in the set. Although this property, along with *precomputed element witnesses*, clearly allows for set-membership verification in $O(1)$ time, it has not been known how this can lead to practical schemes: indeed, straightforward techniques for recomputing the correct witnesses after element updates require at least linear work ($O(n)$ or $O(n \log n)$ depending on the accumulator), thus resulting in high update costs at the server.

In our main result we show how to use two different cryptographic accumulators [11,37] in a hierarchical way over the set and the underlying hash table, to securely authenticate *both membership and non-membership queries* and simultaneously fully achieve our complexity goals. That is, in our authentication schemes the communication and verification costs are both *constant*, the query cost is *constant* and the update cost is *sublinear*, overall realizing the first authenticated hash tables with such performance. (Here, we measure complexity only with respect to the size n of set stored in the hash table; in general, the asymptotic complexities in this paper measure only primitive operations, thus referring only to n and not including any dependences of costs on the security parameter k.) In particular, our schemes strictly improve upon previously proposed set-membership authentication schemes that are based on accumulators. We base the security of our protocols on two widely-accepted assumptions, the strong RSA assumption [2] and the bilinear q-strong Diffie-Hellman assumption [7].

Moreover, to meet different performance needs that are possibly imposed by different data-access patterns over outsourced hash tables, we extend our two schemes

to alternatively achieve an opposite performance trade-offs, namely *sublinear* query cost with *constant* update cost. (As we discuss next, the exact performance adjustment differs in each of our extended schemes.) Finally, aiming at practical solutions, we perform a detailed evaluation and performance analysis of our authentication schemes, discussing many implementation details and showing that, under concrete scenarios and certain standard assumptions related to cryptographic hashing, our protocols achieve very good levels of performance.

1.1 Our Contributions

1. We construct *authenticated hash tables*, a new cryptographic primitive for set-membership verification that is based on combining accumulators in a nested way over a tree of constant depth. We instantiate our solution with two different accumulators, namely the RSA accumulator [2] and the bilinear-map accumulator [37] and formally prove the security of our new schemes based on widely-accepted cryptographic assumptions, namely the strong RSA assumption [2] and the bilinear q-strong Diffie-Hellman assumption [7].
2. We improve the state of art on known techniques and corresponding complexity bounds for authenticating (non)-membership queries on a set of size n, as follows. (See also Table 1). Let $0 < \epsilon < 1$ be a fixed constant. For the scheme based on the RSA accumulator, we reduce the previously best known query cost (using RSA accumulators) from $O(n^{\epsilon})$, as it appears in [22], to $O(1)$, while keeping the update cost sublinear, i.e., $O(n^{\epsilon} \log n)$—overall, this answers an open problem posed in [36]. Also, we extend this scheme to get a different trade-off between query and update costs, namely constant update time with $O(n^{\epsilon})$ query time (as shown in Table 1). For the scheme based on the bilinear-map accumulator, we improve the update cost from $O(n)$, as it appears in [37], to $O(n^{\epsilon})$, while keeping all other costs constant. Analogously, an extension of this scheme achieves $O(n^{\epsilon} \log n)$ query cost and constant update cost.
3. We provide a practical evaluation of our schemes using state-of-the-art software [35,41] for primitive operations (namely, modular exponentiations, multiplications, inverse computations and bilinear maps).

1.2 Related Work

There has been a lot of work on authenticating membership queries using different algorithmic and cryptographic approaches. A summary and qualitative comparison can be found in Table 1.

Several data structures based on cryptographic hashing have been developed for authenticating membership queries on a set of n elements (e.g., [6,23,32,36,43]). These data structures achieve $O(\log n)$ proof size, query time, update time and verification time. As shown in [52], these bounds are optimal for hash-based methods. Variations of this approach and extensions to other types of queries have also been investigated (e.g., [9,13,21,25,28,42,53]).

Table 1 Comparison of our authentication schemes with previous schemes for authenticating element (non-)membership queries on a set of size n, with respect to the underlying cryptographic assumptions used and a variety of complexity measures

References	Assumption	Proof size	Query time	Update time	Verification time
[6,23,32,36,43]	collision resistance of SHA-2	$\log n$	$\log n$	$\log n$	$\log n$
[11,49]	strong RSA	1	1	$n \log n$	1
[37]	bilinear q-strong DH	1	1	n	1
[22]	strong RSA	1	n^ϵ	n^ϵ	1
Theorem 2	strong RSA	1	1	$n^\epsilon \log n$	1
Theorem 4	bilinear q-strong DH	1	1	n^ϵ	1
Theorem 2	strong RSA	1	n^ϵ	1	1
Theorem 4	bilinear q-strong DH	1	$n^\epsilon \log n$	1	1

Here, $0 < \epsilon < 1$ is a fixed constant. All complexity measures refer to n (not the security parameter) and are asymptotic expected values. Allowing sublinear updates and extensions for different update/query cost trade-offs, our schemes perform better than previous approaches. Update costs in our schemes are asymptotic expected amortized values. In all schemes, the required storage for the authenticated data structure is $O(n)$

Solutions for authenticated membership queries using *one-way accumulators* were introduced by Benaloh and de Mare [5]. Based on the RSA exponentiation function, this scheme implements a secure one-way function that is *quasi-commutative*, a useful property that common hash functions lack. This RSA accumulator is used to securely summarize a set so that set membership can be verified with $O(1)$ overhead. Refinements of the RSA accumulator are also given in [2], where except for one-wayness, collision resistance is achieved, and in [20,49]. Dynamic accumulators (along with protocols for zero-knowledge proofs) were introduced in [11], where their security is based on the strong RSA assumption.

The first application of accumulators to the design of authenticated data structures [32,51] was made in [22]; in this work, $O(n^\epsilon)$ bounds are derived for various complexity measures such as query and update time. An authenticated data structure that combines hierarchical hashing with the accumulation-based scheme of [22] is presented in [24], and a similar hybrid authentication scheme appears in [38].

Accumulators using other cryptographic primitives (general groups with bilinear pairings), the security of which is based on other assumptions (hardness of strong Diffie-Hellman problem), are presented in [10,37]. However, updates in [37] are inefficient when the trapdoor information is not known: individual precomputed witnesses can each be updated in constant time, thus incurring a linear total cost for updating all the witnesses after an update in the set. Also in [10], the space needed is proportional to the number of elements *ever* accumulated in the set (book-keeping information of considerable size is needed), or otherwise constraints on the range of the accumulated values are required.

Efficient dynamic RSA accumulators for non-membership proofs are presented in [29] and, subsequently, non-membership proofs for bilinear-map accumulators are presented in [16]. Accumulators for batch updates are presented in [55] and

accumulator-like expressions to authenticate static sets for *provable data possession* are presented in [1,19]. The work in [48] studies efficient algorithms for accumulators with unknown trapdoor information. In [18], logarithmic lower bounds as well as constructions achieving query-update cost trade-offs that are similar to the trade-offs achieved in our work, have been studied in the memory-checking model.

Since set membership can be viewed as an NP statement, the problem that we study in this work can be in principle addressed by recent works for verifiable computation (e.g., [4,8,46]), which can handle the verification of any statement in NP. However, due to their generality, these methods are not quite practical yet and exhibit running times that do not scale well with the size of the statement (in our case, the set size). Therefore, for such fundamental authentication primitives such as set-membership authentication, it is always desirable to settle for customized approaches.

1.3 Organization of the Paper

In Sect. 2, we introduce some necessary algorithmic and cryptographic primitives needed for the development of our constructions and also present our authentication model along with the security definition for our schemes. In Sect. 3, we develop our first scheme that is based on the RSA accumulator. In Sect. 4, we present our second scheme that is based on the bilinear-map accumulator. In Sect. 5, we provide an evaluation and analysis of our authentication schemes showing their practicality, and finally in Sect. 6 we conclude with future work and interesting open problems.

2 Authentication Model and Authentication Primitives

In this section we describe the authentication model as well as some algorithmic and cryptographic primitives and other useful concepts that are used in our work.

2.1 Authenticated Data Structures

We first provide definitions for a *data structure scheme*, its respective *authenticated data structure scheme* as well as the corresponding properties of correctness and security. Similar definitions have initially appeared in the work of Tamassia and Triandopoulos [54] and subsequently in the work of Papamanthou et al. [45]. 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 i inputs $I_1, I_2, \ldots, I_i$ and o outputs $O_1, O_2, \ldots, O_o$.

Definition 1 (*Data structure scheme*) Let D be *any* data structure supporting a set of updates $\mathbf{U}$ and a set of queries $\mathbf{Q}$. Denote with D_h the state of the data structure D at time h, where $h \geq 0$ is an integer and D_0 is the initial state of D. A *data structure scheme* $\mathcal{D}(\mathbf{U}, \mathbf{Q})$ for D is a collection of the following three polynomial-time algorithms {**`update`**, **`query`**, **`check`**}:

1. $D_{h+1} \leftarrow \mathbf{update}(u, D_h)$: On input an update $u \in \mathbf{U}$ and the data structure D_h, this algorithm outputs the updated data structure D_{h+1};
2. $\alpha(q) \leftarrow \mathbf{query}(q, D_h)$: On input a query $q \in \mathbf{Q}$ and the data structure D_h, this algorithm outputs the answer $\alpha(q)$ to query q;
3. accept or reject $\leftarrow \mathbf{check}(q, \alpha, D_h)$: On input a query $q \in \mathbf{Q}$, an answer α and the data structure D_h, this algorithm outputs accept if α is a correct answer for query q on data structure D_h; otherwise it outputs reject.

For example, consider such a scheme for the *dictionary* data structure, implemented with a red-black tree [15]. Algorithm **query**() performs a binary search while algorithm **update**() performs the relevant rotations needed for re-balancing the structure. We note that in Definition 1, script letter $\mathcal{D}$ in the notation $\mathcal{D}(\mathbf{U}, \mathbf{Q})$ is conveniently used to imply that $\mathcal{D}(\mathbf{U}, \mathbf{Q})$ is a data structure scheme for data structure D (non-script letter) where D supports the set of updates $\mathbf{U}$ and the set of queries $\mathbf{Q}$.

Definition 2 (*Authenticated data structure scheme*) Let $\mathcal{D}(\mathbf{U}, \mathbf{Q}) = \{\mathbf{update}, \mathbf{query}, \mathbf{check}\}$ be a data structure scheme. An *authenticated data structure scheme* or ADS scheme $\mathcal{A}$ for the data structure scheme $\mathcal{D}(\mathbf{U}, \mathbf{Q})$ is a collection of the following six polynomial-time algorithms {genkey, setup, update, refresh, query, verify}:

1. $\{\mathsf{sk}, \mathsf{pk}\} \leftarrow \mathsf{genkey}(1^k)$: On input the security parameter k, this algorithm outputs the secret key sk and the public key pk;
2. $\{\mathsf{auth}(D_0), d_0\} \leftarrow \mathsf{setup}(D_0, \mathsf{sk}, \mathsf{pk})$: On input a data structure D_0, the secret key sk and the public key pk, this algorithm computes the authenticated data structure $\mathsf{auth}(D_0)$ and the respective digest d_0 of $\mathsf{auth}(D_0)$;
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 $u \in \mathbf{U}$, a data structure D_h, an authenticated data structure $\mathsf{auth}(D_h)$, the digest d_h of $\mathsf{auth}(D_h)$ and both the secret and the public keys sk, pk, this algorithm outputs the data structure $D_{h+1} \leftarrow \mathbf{update}(u, D_h)$, the authenticated data structure $\mathsf{auth}(D_{h+1})$, the digest d_{h+1} of $\mathsf{auth}(D_{h+1})$ and some update 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 $u \in \mathbf{U}$, a data structure D_h, an authenticated data structure $\mathsf{auth}(D_h)$, the digest d_h of $\mathsf{auth}(D_h)$, the information upd computed by algorithm update() and *only* the public key pk, this algorithm outputs the data structure $D_{h+1} \leftarrow \mathbf{update}(u, D_h)$, the authenticated data structure $\mathsf{auth}(D_{h+1})$ and the digest d_{h+1} of $\mathsf{auth}(D_{h+1})$;
5. $\{\Pi(q), \alpha(q)\} \leftarrow \mathsf{query}(q, D_h, \mathsf{auth}(D_h), \mathsf{pk})$: On input a query $q \in \mathbf{Q}$, a data structure D_h, an authenticated data structure $\mathsf{auth}(D_h)$ and the public key pk, this algorithm returns the answer $\alpha(q) \leftarrow \mathbf{query}(q, D_h)$ to the query q, along with a respective proof $\Pi(q)$;
6. accept or reject $\leftarrow \mathsf{verify}(q, \alpha, \Pi, d_h, \mathsf{pk})$: On input a query $q \in \mathbf{Q}$, an answer α, a proof Π, the digest d_h of $\mathsf{auth}(D_h)$ and the public key pk, this algorithm outputs either accept or reject.

We note that in the definition above the digest d_h of the authenticated data structure $\mathsf{auth}(D_h)$ is used to denote a *succinct and secure* representation of D_h, typically of

constant size and satisfying a collision-resistant property, e.g., the root hash of a Merkle tree [33]. As we will see later, the exact computation of this digest depends on some underlying cryptographic primitive and drastically affects the efficiency and security of the corresponding authenticated data structure. We also note that the difference between algorithms update() and refresh() is only that the latter *makes no use of the secret key* sk; that is, running on *correct* corresponding inputs, both algorithms produce identical output pairs $(\mathsf{auth}(D_{h+1}), d_{h+1})$.

There are two properties that an ADS scheme should satisfy, namely *correctness* and *security*. The intuition behind these properties follows naturally by considering the corresponding correctness and security properties of signature schemes (see, e.g., the definitions in the signature scheme by Camenisch and Lysyanskaya [12]). Roughly speaking, the correctness property requires that, for every query $q \in \mathbf{Q}$, if a proof $\Pi(q)$ is computed by algorithm query() *faithfully*, then verify(), on input $\Pi(q)$ and a *correct* answer $\alpha(q)$, should always accept, as long as the digest d is *consistently* updated via algorithm refresh() for any update $u \in \mathbf{U}$. The security property requires that any computationally-bounded adversary, i.e., an adversary that has access to polynomially-bounded (time and space) resources in the security parameter k, should not be able, except with negligible probability in k, to produce verifying proofs Π for *incorrect* answers α corresponding to queries $q \in \mathbf{Q}$ on an authenticated data structure $\mathsf{auth}(D)$ whose digest d is *consistently* updated even through *oracle* calls to algorithm update() on *adversarially chosen* updates $u \in \mathbf{U}$. More formally:

Definition 3 (*Correctness of ADS scheme*) Let $\mathcal{D}(\mathbf{U}, \mathbf{Q}) = \{\mathbf{update}, \mathbf{query}, \mathbf{check}\}$ be a data structure scheme and let $\mathcal{A} = \{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$ be an ADS scheme for $\mathcal{D}(\mathbf{U}, \mathbf{Q})$. We say that $\mathcal{A}$ is correct if, for all sufficiently large $k \in \mathbb{N}$, 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 algorithm setup() followed by polynomially-many invocations of algorithm refresh(), where $h \geq 0$, for all queries $q \in \mathbf{Q}$ and for all $\Pi(q), \alpha(q)$ output by algorithm $\mathsf{query}(q, D_h, \mathsf{auth}(D_h), \mathsf{pk})$, with all but negligible probability in k, whenever algorithm $\mathbf{check}(q, \alpha(q), D_h)$ accepts, so does algorithm $\mathsf{verify}(q, \Pi(q), \alpha(q), d_h, (\mathsf{sk})^*, \mathsf{pk})$.

Definition 4 (*Security of ADS scheme*) Let $\mathcal{D}(\mathbf{U}, \mathbf{Q}) = \{\mathbf{update}, \mathbf{query}, \mathbf{check}\}$ be a data structure scheme, let $\mathcal{A} = \{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$ be an ADS scheme for $\mathcal{D}(\mathbf{U}, \mathbf{Q})$, k be a security parameter and $\{\mathsf{sk}, \mathsf{pk}\} \leftarrow \mathsf{genkey}(1^k)$. Denote with Adv a polynomially-bounded adversary that is only given the public key pk, has unlimited access to all algorithms of $\mathcal{A}$, except for algorithms setup() and update(), to which he has only oracle access, and which operates as follows:

- Adv picks an initial state of the data structure D_0 and computes $\mathsf{auth}(D_0), d_0$ through oracle access to algorithm setup();
- Then, for $i = 0, \ldots, h = \mathsf{poly}(k)$, Adv issues an update $u_i \in \mathbf{U}$ in the data structure D_i and outputs D_{i+1}, the authenticated data structure $\mathsf{auth}(D_{i+1})$ and d_{i+1} through oracle access to algorithm update();
- Finally the adversary enters the attack stage where he picks an index $0 \leq t \leq h+1$, a query $q \in \mathbf{Q}$, an answer α and a proof Π.

We say that $\mathcal{A}$ is secure if for all sufficiently large $k \in \mathbb{N}$, for all $\{\mathsf{sk}, \mathsf{pk}\}$ output by algorithm $\mathsf{genkey}()$, and for all polynomially-bounded adversaries Adv it is

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

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

Above and in what follows, $\mathsf{neg}(k)$ denotes a negligible function in k.

2.2 Two-Party and Three-Party Authentication Protocols

We next describe how the algorithms of an ADS scheme (given in Definition 2) can be securely used in two concrete, widely-used data authentication protocols, which we call the *three-party* and *two-party* authentication protocols and which are both closely related to our remote-storage setting.

Three-party authentication protocol In the three-party authentication protocol [51], 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 $\mathsf{genkey}()$ and $\mathsf{setup}()$ and outputs the authenticated data structure $\mathsf{auth}(D_h)$ along with the digest d_h. The source subsequently signs the digest d_h, and it outsources the data structure D_h, the authenticated data structure $\mathsf{auth}(D_h)$, the digest d_h and the signature on this digest to one (or more) untrusted entities, called *servers*.

On input a data structure query q (e.g., a set-membership query) sent by a client, any such server uses $\mathsf{auth}(D_h)$ and D_h to compute proof $\Pi(q)$, by running algorithm $\mathsf{query}()$, and then returns to the client, along with the answer $a(q)$ to q, the proof $\Pi(q)$, the digest d_h and the source's signature on d_h. The client can then verify this proof $\Pi(q)$ by running algorithm $\mathsf{verify}()$ (since the client has access to the signature of d_h, the client can verify that d_h is authentic).

When the source issues an update in the data structure, it uses algorithm $\mathsf{update}()$ to produce (and sign) the new digest d'_h to be used for future answer verifications, and provides the servers with the corresponding update information upd, with which the servers update the authenticated data structure by running algorithm $\mathsf{refresh}()$. Note that to defend against possible replay attacks by the server, the source should issue updates following a schedule that is known to the client.

Two-party authentication protocol Moreover, an ADS scheme can also be used in a two-party authentication protocol [43]. In this case, the source and the client functionalities (of the three-party authentication protocol above) coincide. That is, the client (is also the source and) issues both the updates and the queries of the data structure maintained by a server. Importantly, for this two-party data-outsourcing model to be meaningful, the client does not only outsource the computation (of query answering as above) to the server but also the storage of the data structure, or otherwise there is no need to interact with the server. That is, the client is required to keep only some authentication state of size that is sublinear in the data structure size—in fact, the

client is able to only keep constant state, typically (including) the digest d_h of the authenticated data structure. The protocol maintains the invariant property that at all times this authentication state (i.e., digest) is *correctly locally updated* by the client over the course of updates to the data structure. Then, querying over the data structure is performed almost identically as in the three-party case: proof computation and verification is performed through algorithms query() and verify() executed by the server and the client respectively, only now there is no need for the server to provide the client with the digest d_h (neither a signature on it), as d_h is authentically maintained locally by the client.

Updating the data structure, though, becomes more involved in the two-party authentication case. Clearly, with possession of d_h, should the client locally store the data structure D_h and the authenticated data structure $\mathsf{auth}(D_h)$, updates would readily be performed identically to the three-party authentication case: the client would first run update() to locally update d_h and provide the server with update information upd, the latter then allowing the server to run refresh() to update D_h and $\mathsf{auth}(D_h)$. To allow the client run update() without knowledge of D_h and $\mathsf{auth}(D_h)$, the following generic solution is adopted: the client interacts with the server to *partially* and *verifiably* retrieve those (well-defined) portions of D_h and $\mathsf{auth}(D_h)$, say denoted by $D_h(u)$ and $\mathsf{auth}(D_h, u)$, that are relevant (or necessary) for correctly running update() on the specific new update u of its choice. That is, the correct execution $\{D_{h+1}, \mathsf{auth}(D_{h+1}), d_{h+1}, \mathsf{upd}\} \leftarrow \mathsf{update}(u, D_h, \mathsf{auth}(D_h), d_h, \mathsf{sk}, \mathsf{pk})$ on authentic, locally stored D_h and $\mathsf{auth}(D_h)$ should be identical to the execution $\{D_{h+1}, \mathsf{auth}(D_{h+1}), d_{h+1}, \mathsf{upd}\} \leftarrow \mathsf{update}(u, D_h(u), \mathsf{auth}(D_h, u), d_h, \mathsf{sk}, \mathsf{pk})$ on retrieved $D_h(u)$ and $\mathsf{auth}(D_h, u)$ from the server.

Clearly, the challenge here is to receive $D_h(u)$ and $\mathsf{auth}(D_h, u)$ from the server in a way that allows efficient verification of their integrity, namely, that they are correct with respect to the current digest d_h. Interestingly, for a large class of authenticated data structure schemes, the above requirement is simply satisfied by having the client simply retrieve $D_h(u)$ and $\mathsf{auth}(D_h, u)$ through the execution of algorithm query() on a query $q(u)$ that is related to the update of interest u (for a concrete example see [43]). Then, very conveniently in the two-party authentication protocol, the execution $\{\Pi(q(u)), \alpha(q(u))\} \leftarrow \mathsf{query}(q(u), D_h, \mathsf{auth}(D_h), \mathsf{pk})$ by the server produces an proof-answer pair $\{\Pi(q), \alpha(q)\}$ which, when coupled with a successful verification through execution $\mathsf{accept} \leftarrow \mathsf{verify}(q(u), \alpha(q(u)), \Pi(q(u)), d_h, \mathsf{pk})$ by the client, allows to also (explicitly or implicitly) verify the partial information $D_h(u)$ and $\mathsf{auth}(D_h, u)$. In short, whenever there is an update by the client, the client retrieves a verifiable portion of the authenticated data structure that is used to locally perform the update and, in turn, compute the new local state, i.e., the new digest, thus to maintain the invariant.

A more detailed description of how to construct the respective two-party and three-party authentication protocols given an ADS scheme as well as how the corresponding protocol security properties are reduced to the security of the underlying ADS scheme is provided by Papamanthou [40, Chapter 2.2].

Accordingly, using the above design framework, our new ADS schemes for realizing an authenticated hash table, lend themselves to corresponding secure authentication protocols in both the three-party and the two-party authentication models.

2.3 Hash Tables

The main functionality of a *hash table* data structure $\mathbf{T}(\mathcal{X})$ is to support look-ups of optimal complexity for elements that belong to a general dynamic set $\mathcal{X}$. That is, elements drawn from a universe $\mathcal{U}$ can be inserted in or deleted from $\mathcal{X}$ and are not necessarily ordered in $\mathcal{X}$.

The data structure scheme Following Definition 1, a data structure scheme {**query**(), **update**(), **check**()} for a hash table $\mathbf{T}(\mathcal{X})$ is as follows:

1. true or false $\leftarrow$ **query**$(x, \mathbf{T}(\mathcal{X}))$: Given an element $x \in \mathcal{U}$, return true if $x \in \mathcal{X}$ or false otherwise;
2. $\mathbf{T}(\mathcal{X}') \leftarrow$ **update**$(x, \mathbf{T}(\mathcal{X}))$: Given an element $x \in \mathcal{U}$ such that $x \notin \mathcal{X}$, *insert* element x into $\mathcal{X}$ and output $\mathbf{T}(\mathcal{X}')$; Given an element $x \in \mathcal{U}$ such that $x \in \mathcal{X}$, *delete* element x from $\mathcal{X}$ and output $\mathbf{T}(\mathcal{X}')$;
3. accept or reject $\leftarrow$ **check**$(x, b \in \{\text{true}, \text{false}\}, \mathbf{T}(\mathcal{X}))$: If $x \in \mathcal{X}$ and $b =$ false or $x \notin \mathcal{X}$ and $b =$ true, then return reject; otherwise, return accept.

Implementation Different ways of implementing hash tables have been extensively studied (e.g., [17,26,27,30,34]). Here we consider a simple approach for optimally implementing a hash table: Suppose we wish to store n elements from $\mathcal{U}$ in a data structure so that we can have expected constant look-up complexity. When $\mathcal{U}$ is a totally ordered universe, it is well known that look-ups through searching based on comparisons induce an $\Omega(\log n)$ lower bound. Essential for any construction of a hash table that achieves $o(\log n)$ complexity is a *two-universal hash function*.

Definition 5 (*Two-universal hash function* [14]) A *two-universal hash function* $H : \mathcal{U} \rightarrow \{1, \ldots, m\}$, randomly selected from a family of two-universal hash functions $\mathcal{H}$, is a function such that for any two elements $e_1, e_2 \in \mathcal{U}$, it is

$$\Pr[H(e_1) = H(e_2)] \leq \frac{1}{m}. \tag{1}$$

By using a two-universal hash function, hash tables can be constructed as follows:

- Set up a one-dimensional table $\mathbf{T}[1 \ldots m]$ where $m = O(n)$;
- Pick a two-universal hash function $H : \mathcal{U} \rightarrow \{1, \ldots, m\}$ according to Definition 5;
- Store element e in slot $\mathbf{T}[H(e)]$ of the table.

The probabilistic property that holds for hash function H implies that for any slot of the table, the expected number of elements mapped to it is $O(1)$. Also, if H can be computed in $O(1)$ time, looking-up an element has expected constant complexity.

But the above property of hash tables comes at some cost. The expected constant-complexity look-up holds when the number of elements stored in the hash table does not change, i.e., when the hash table is static. In particular, because of insertions, the number of elements stored in a slot may grow and we can no longer assume that this is constant in expectation. A different problem arises in the presence of deletions as the number n of elements may become much smaller than the size m of the hash table. Thus, we can no longer assume that the hash table uses $O(n)$ space.

In order to deal with updates, we periodically update the size of the hash table by a constant factor (e.g., doubling or halving its size). This is an expensive operation since we have to rehash all the elements. Therefore, there might be one update (over a course of $O(n)$ updates) that has $O(n)$ rather than $O(1)$ complexity. Thus, hash tables for dynamic sets typically have expected $O(1)$ query complexity and $O(1)$ expected *amortized* update complexity. Methods that vary the size of the hash table in order to maintain $O(1)$ expected query complexity, fall into the general category of *dynamic hashing*. The above discussion is summarized in the following theorem.

Theorem 1 (Dynamic hashing [15]) *For a set of size n, dynamic hashing can be implemented to use $O(n)$ space and have $O(1)$ expected query complexity for (non-)membership queries and $O(1)$ expected amortized complexity for elements insertions or deletions.*

2.4 The RSA Accumulator

We next overview the *RSA accumulator*, a cryptographic primitive which will be used for our first solution, i.e., the construction of the ADS scheme $\mathcal{RHT}$.

Prime representatives For security and correctness reasons that will soon become clear, in our construction we extensively use the notion of *prime representatives* of elements. Initially introduced by Baric and Pfitzmann [2], prime representatives provide a solution whenever it is necessary to map general elements to prime numbers. In particular, for any integer k, one can map a k-bit element e_i to a $3k$-bit prime x_i using a two-universal hash function (Definition 5). In our context, we are using a two-universal hash function $h : A \rightarrow B$, which is different than the function $H(\cdot)$ we use to map elements to buckets, and where set A is the set of $3k$-bit boolean vectors and B is the set of k-bit boolean vectors. Specifically, we use the two-universal hash function

$$h(x) = \mathbf{F}x,$$

where $\mathbf{F}$ is a $k \times 3k$ boolean matrix. Since the linear system $h(x) = \mathbf{F}x$ has more than one solutions, one k-bit element is mapped (through matrix $\mathbf{F}$) to more than one $3k$-bit elements. We are interested in finding only one such solution which is prime; this can be computed efficiently according to the following result.

Lemma 1 (Prime representatives [20,22]) *Let $\mathcal{H}$ be a two-universal family of functions mapping $\{0,1\}^{3k}$ to $\{0,1\}^k$ and let $h \in \mathcal{H}$. For any element $e_i \in \{0,1\}^k$, we can compute with high probability a prime $x_i \in \{0,1\}^{3k}$ such that $h(x_i) = e_i$ by sampling $O(k^2)$ times from the set of inverses $h^{-1}(e_i)$.*

By Lemma 1, we have that computing prime representatives has expected constant complexity, i.e., independent of n. Also, solving the $k \times 3k$ linear system in order to compute the set of inverses has polynomial complexity in k by using standard methods (e.g., Gaussian elimination). Finally, we note that, in our context, prime representatives

are computed and stored only once. Indeed, using the above method multiple times for computing the prime representative of the same element will not yield the same prime as output, for Lemma 1 describes a randomized process. From now on, given a k-bit element x, we denote with $r(x)$ the $3k$-bit prime representative that is computed as described by Lemma 1.

Description of the RSA accumulator We now give an overview of the *RSA accumulator* [2,5,11,29], which provides an efficient technique to produce a short computational proof that a certain element is (or is not) a member of a set. The RSA accumulator works as follows.

Suppose we have the set of k-bit elements $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$. Let N be a k'-bit RSA modulus with $k' > 3k$, namely $N = pq$, where p, q are strong primes [11]. Using the RSA accumulator, we can represent $\mathcal{X}$ compactly and securely with an *accumulation* value $\mathsf{acc}(\mathcal{X})$, which is a k'-bit integer defined as

$$\mathsf{acc}(\mathcal{X}) = g^{r(x_1)r(x_2)\ldots r(x_n)} \mod N,$$

where $g \in \mathbb{QR}_N$ and $r(x_i)$ is a $3k$-bit prime representative, computed using a two-universal hash function h.

Here, the RSA modulus N, the exponentiation base g and the two-universal hash function h comprise the public key of the RSA accumulator, that is, publicly available information that allows to compute the accumulation value of any set—of course, the factorization of N, which is the trapdoor of the RSA accumulator, is kept secret.

Subject to the accumulation $\mathsf{acc}(\mathcal{X})$, every element x in set $\mathcal{X}$ has a *membership witness* $(\mathsf{W}_x, r(x), x)$, where

$$\mathsf{W}_x = g^{\prod_{x_j \in \mathcal{X}: x_j \neq x} r(x_j)} \mod N. \tag{2}$$

Given accumulation value $\mathsf{acc}(\mathcal{X})$ and (membership) witness $(\mathsf{W}_x, r(x), x)$, membership of x in $\mathcal{X}$ is verified by means of the following tests:

1. Checking that $r(x)$ is a prime number;
2. Checking that $h(r(x)) = x$;
3. Checking that $\mathsf{W}_x^{r(x)} \mod N$ equals $\mathsf{acc}(\mathcal{X})$.

Moreover, as shown in [29], subject to the accumulation $\mathsf{acc}(\mathcal{X})$, every element $x \notin \mathcal{X}$ has a *non-membership witness*, namely the integer values $(\mathsf{A}_x, \mathsf{B}_x, r(x), x)$ such that

$$\left(\prod_{i=1}^{n} r(x_i)\right)\mathsf{A}_x + r(x)\mathsf{B}_x = 1. \tag{3}$$

Note that A_x and B_x can be computed by running the extended Euclidean algorithm [50] on $r(x_1)r(x_2)\ldots r(x_n)$ and $r(x)$. Given accumulation value $\mathsf{acc}(\mathcal{X})$ and (non-membership) witness $(\mathsf{A}_x, \mathsf{B}_x, r(x), x)$, non-membership of x in $\mathcal{X}$ can be verified by means of the following tests:

1. Checking that $r(x)$ is a prime number;

2. Checking that $h(r(x)) = x$;
3. Checking that $\mathsf{acc}(\mathcal{X})^{\mathsf{A}_x} g^{x\mathsf{B}_x} \mod N$ equals g.

Finally, the representation $\mathsf{acc}(\mathcal{X})$ has the crucial property that any computationally bounded adversary Adv who does not know the trapdoor $\phi(N)$ cannot find another set of elements $\mathcal{X}' \neq \mathcal{X}$ such that $\mathsf{acc}(\mathcal{X}') = \mathsf{acc}(\mathcal{X})$, unless Adv breaks the *the factoring assumption*. However, in order to achieve some more advanced security goals, we are going to use a stronger assumption [2]:

Assumption 1 (Strong RSA assumption) *Let k be the security parameter. Given a k-bit RSA modulus N and a random element $x \in \mathbb{Z}_N^*$, there is no polynomial-time algorithm that outputs (y, a) such that $y > 1$ and $a^y = x \mod N$, except with negligible probability* $\mathsf{neg}(k)$.

The security of our RSA-based solution relies on the following result, for which for completeness we provide the proofs of the security properties of both membership [2] and non-membership [29] witnesses.

Lemma 2 (Security of the RSA accumulator [2,29]) *Let k be the security parameter, h be a two-universal hash function mapping $3k$-bit integers to k-bit integers, N be a $(3k + 1)$-bit RSA modulus and $g \in \mathbb{QR}_N$. Given N, g, an arbitrary set of k-bit elements $\mathcal{X}$ and h, suppose there is a polynomial-time algorithm for any of the two tasks below:*

- *The algorithm outputs $x \notin \mathcal{X}$, W and prime r such that $h(r) = x$ and $\mathsf{W}^r = \mathsf{acc}(\mathcal{X}) \mod N$;*
- *The algorithm outputs $x \in \mathcal{X}$, A, B and prime r such that $h(r) = x$ and $\mathsf{acc}(\mathcal{X})^{\mathsf{A}} g^{r\mathsf{B}} = g \mod N$.*

Then there is a polynomial-time algorithm for breaking the strong RSA assumption on an RSA modulus N of $3k + 1$ bits.

Proof Let $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ and let $x \notin \mathcal{X}$. For the membership proof, suppose there is an algorithm that finds W, r and x such that r is a prime number, $h(r) = x$ and

$$\mathsf{W}^r = g^{r(x_1)r(x_2)\ldots r(x_n)} \mod N.$$

Since $x \notin \mathcal{X}$, by construction of the prime representatives, it is $r \notin \{r(x_1), r(x_2), \ldots, r(x_n)\}$ (recall that $h(r) = x$). Let now $e = r$ and $R = r(x_1)r(x_2)\ldots r(x_n)$. The algorithm can now compute the e-th root of g as follows: It computes $a, b \in \mathbb{Z}$ such that $aR + br = 1$ by using the extended Euclidean algorithm, since r is a prime and $r \notin \{r(x_1), r(x_2), \ldots, r(x_n)\}$. Let now $y = \mathsf{W}^a g^b \mod N$. It is

$$y^e = \mathsf{W}^{ar} g^{br} = g^{aR+br} = g \mod N.$$

Therefore the algorithm can be used for breaking the strong RSA assumption. For the non-membership proof case, since $x \in \mathcal{X}$ the algorithm can output the e-th root of g

as $y = \mathsf{W}_x^{\mathsf{A}} g^{\mathsf{B}}$, where W_x is the membership witness defined in Relation 2. Then

$$y^e = \mathsf{W}_x^{e\mathsf{A}} g^{e\mathsf{B}} = \mathsf{acc}(\mathcal{X})^{\mathsf{A}} g^{r\mathsf{B}} = g \mod N.$$

This completes the proof. □

2.5 The Bilinear-Map Accumulator

We next overview of the *bilinear-map accumulator* [37] which will be used for our second solution, i.e., the construction of the ADS scheme $\mathcal{BHT}$.

Bilinear pairings Let $\mathbb{G}_1, \mathbb{G}_2$ be two cyclic multiplicative groups of prime order p, generated by g_1 and g_2 and for which there exists an isomorphism $\psi : \mathbb{G}_2 \to \mathbb{G}_1$ such that $\psi(g_2) = g_1$. Let also $\mathcal{G}$ be a cyclic multiplicative group with the same order p and $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathcal{G}$ be a *bilinear pairing* with the following properties:

1. Bilinearity: $e(P^a, Q^b) = e(P, Q)^{ab}$ for all $P \in \mathbb{G}_1$, $Q \in \mathbb{G}_2$ and $a, b \in \mathbb{Z}_p$;
2. Non-degeneracy: $e(g_1, g_2) \neq 1$;
3. Computability: There is an efficient algorithm to compute $e(P, Q)$ for all $P \in \mathbb{G}_1$ and $Q \in \mathbb{G}_2$.

In our setting we have $\mathbb{G}_1 = \mathbb{G}_2 = \mathbb{G}$ and $g_1 = g_2 = g$. A *bilinear pairing instance generator* is a probabilistic polynomial-time algorithm that takes as input the security parameter k and outputs a uniformly random tuple $\mathsf{t} = (p, \mathbb{G}, \mathcal{G}, e, g)$ of bilinear pairings parameters, where p is a k-bit prime number.

Description of the bilinear-map accumulator Similar to the RSA accumulator, the *bilinear-map accumulator* [16,37] comprises an efficient technique to provide short computational proofs of (non-)membership for elements that (do not) belong to a set. The bilinear-map accumulator works as follows.

Suppose we have a set of n elements $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ in $\mathbb{Z}_p^* - \{-s\}$, where p be a k-bit prime. Let $s \in \mathbb{Z}_p^*$ be a randomly chosen value that constitutes the trapdoor of the bilinear-map accumulator (in the same way that $\phi(N)$ was the trapdoor of the RSA accumulator). Using the bilinear-map accumulator, we can represent $\mathcal{X}$ compactly and securely with an accumulation value $\mathsf{acc}(\mathcal{X})$, which is an element in $\mathbb{G}$ defined as

$$\mathsf{acc}(\mathcal{X}) = g^{(x_1+s)(x_2+s)\ldots(x_n+s)},$$

where g is a generator of group $\mathbb{G}$ of (prime) order p.

Here, group elements $g, g^s, g^{s^2}, \ldots, g^{s^q}$, for some integer q, and the associated bilinear pairings parameters, comprise the public key of the bilinear-map accumulator that allows to compute the accumulation value of any set $\mathcal{X}$ of size $n \leq q$, using polynomial interpolation (see Lemma 14).

Subject to the accumulation value $\mathsf{acc}(\mathcal{X})$, every element x in $\mathcal{X}$ has a membership witness (W_x, x) where

$$\mathsf{W}_x = g^{\prod_{x_j \in \mathcal{X} : x_j \neq x} (x_j + s)}. \tag{4}$$

Accordingly, given accumulation value $\mathsf{acc}(\mathcal{X})$ and witness (W_x, x), membership of x in $\mathcal{X}$ is verified by checking that $e(\mathsf{W}_x, g^s g^x)$ equals $e(\mathsf{acc}(\mathcal{X}), g)$. Moreover, as shown in [16], subject to accumulation value $\mathsf{acc}(\mathcal{X})$, every element $x \notin \mathcal{X}$ has a non-membership witness, namely the elements $(\mathsf{A}_x = g^{\alpha(s)}, \mathsf{B}_x = g^{\beta(s)}, x)$ such that

$$\left(\prod_{i=1}^{n}(x_i + s)\right)\alpha(s) + (x + s)\beta(s) = 1. \tag{5}$$

Note that $\alpha(s)$ and $\beta(s)$ are polynomials that can be computed by running the extended Euclidean algorithm on polynomials $(x_1 + s)(x_2 + s)\ldots(x_n + s)$ and $(x + s)$. Given accumulation value $\mathsf{acc}(\mathcal{X})$ and witness elements A_x and B_x, non-membership of x in $\mathcal{X}$ can be verified by checking that $e(\mathsf{acc}(\mathcal{X}), \mathsf{A}_x)e(g^s g^x, \mathsf{B}_x)$ equals $e(g, g)$.

Again, the representation $\mathsf{acc}(\mathcal{X})$ has the crucial property that any computationally bounded adversary Adv who does not know the trapdoor s cannot find another set of elements $\mathcal{X}' \neq \mathcal{X}$ such that $\mathsf{acc}(\mathcal{X}') = \mathsf{acc}(\mathcal{X})$, unless Adv breaks the so-called *q-strong Diffie-Hellman assumption*. In our work, we need the slightly stronger *bilinear q-strong Diffie-Hellman assumption* [7] that can be stated as follows:

Assumption 2 (Bilinear q-strong Diffie-Hellman assumption) *Let k be the security parameter. Let $(p, \mathbb{G}, \mathcal{G}, e, g)$ be a uniformly randomly generated tuple of bilinear pairings parameters where p is a k-bit prime. Given the elements $g, g^s, \ldots, g^{s^q} \in \mathbb{G}$ for some s chosen at random from $\mathbb{Z}_p^*$, where $q = \mathsf{poly}(k)$, there is no polynomial-time algorithm that can output a pair $(a, e(g, g)^{1/(s+a)}) \in \mathbb{Z}_p \times \mathcal{G}$ except with negligible probability $\mathsf{neg}(k)$.*

The security of our bilinear-map based solution relies on the following result, for which for completeness we provide the proofs of the security properties of both membership [37] and non-membership [16] witnesses.

Lemma 3 (Security of the bilinear-map accumulator [16,37]) *Let k be the security parameter. Let $(p, \mathbb{G}, \mathcal{G}, e, g)$ be a uniformly randomly generated tuple of bilinear pairings parameters where p is a k-bit prime. Given the elements $g, g^s, \ldots, g^{s^q} \in \mathbb{G}$ for some s chosen at random from $\mathbb{Z}_p^*$ and an arbitrary set of k-bit elements $\mathcal{X}$ $(q \geq |\mathcal{X}|)$, suppose there is a polynomial-time algorithm for any of the tasks below:*

- *The algorithm outputs $x \notin \mathcal{X}$ and W such that $e(\mathsf{W}, g^s g^x) = e(\mathsf{acc}(\mathcal{X}), g)$;*
- *The algorithm outputs $x \in \mathcal{X}$, A and B such that $e(\mathsf{acc}(\mathcal{X}), \mathsf{A})e(g^s g^x, \mathsf{B}) = e(g, g)$.*

Then there is a polynomial-time algorithm for breaking the bilinear q-strong Diffie-Hellman assumption.

Proof Let $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ and let $x \notin \mathcal{X}$. Suppose there is an algorithm that finds W such that $e(\mathsf{W}, g^s g^x) = e(\mathsf{acc}(\mathcal{X}), g)$. This implies

$$e(\mathsf{W}, g)^{s+x} = e(g, g)^{(s+x_1)(s+x_2)\ldots(s+x_n)}.$$

Note now that the quantity

$$\Pi_n = (s + x_1)(s + x_2) \dots (s + x_n)$$

can be viewed as a polynomial in s of degree n. Since $x \notin \mathcal{X}$, we have that $(s + x)$ does not divide Π_n and therefore values c and P can be computed such that $\Pi_n = c + P(s + x)$. Therefore the algorithm can output $(x, e(g, g)^{1/(s+x)})$ as

$$\left(x, \left[e(\mathsf{W}, g)e(g, g)^{-P}\right]^{c^{-1}}\right).$$

For a non-membership proof, note that we can output $e(g, g)^{1/(s+x)}$ as

$$e(\mathsf{W}_x, \mathsf{A})e(g, \mathsf{B}),$$

since $x \in \mathcal{X}$ and $e(\mathsf{acc}(\mathcal{X}), \mathsf{A})e(g^s g^x, \mathsf{B}) = e(g, g)$, where W_x is a membership witness given in Relation 4. Therefore the bilinear q-strong Diffie-Hellmann assumption can be broken in both cases. □

Size of non-membership witnesses We note here that although non-membership witnesses are constructed in the same fashion in both instantiations of the accumulator schemes, their sizes differ considerably. In the case of RSA accumulators, integers A_x and B_x in Relation 3 can have size proportional to the number of the elements in $\mathcal{X}$. In the case of bilinear-map accumulators, witness elements A_x and B_x in Relation 5 are always *two* group elements in $\mathbb{G}$ (this takes advantage of the bilinear map $e(\cdot, \cdot)$), therefore their size never depends on set $\mathcal{X}$.

2.6 The Accumulation Tree

We conclude the section by presenting an algorithmic construction that we call an *accumulation tree* that will be at the core of both of our constructions.

Let $\mathcal{X} = \{x_1, x_2, \dots, x_n\}$ be a set of elements. Given a constant $\epsilon < 1$ such that $0 < \epsilon < 1$, the *accumulation tree* of $\mathcal{X}$, denoted with $T(\epsilon)$, is a rooted tree with n leaves defined as follows:

1. The leaves of $T(\epsilon)$ store the elements $x_1, x_2, \dots, x_n$;
2. $T(\epsilon)$ consists of exactly $l = \lceil \frac{1}{\epsilon} \rceil$ levels;
3. Every node of $T(\epsilon)$ has $O(n^\epsilon)$ children;
4. All the leaves are located at the same level;
5. Level i in the tree contains $O(n^{1-i\epsilon})$ nodes (where leaves are at level 0 and the root is at level l).

Note that levels in our accumulation tree are numbered from the leaves to the root of the tree, i.e., the leaves have level 0, their parents level 1 and finally the root has level l. The structure of the accumulation tree, which for a set of 64 elements is shown in Fig. 1, resembles that of perfect "flat" search trees, in particular, the structure of

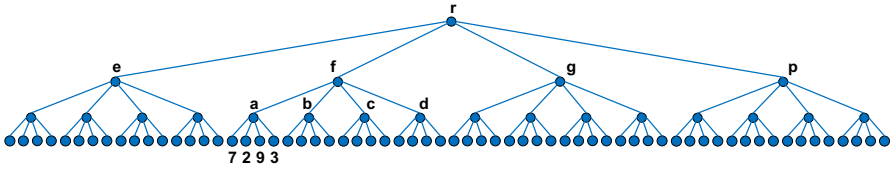

Fig. 1 The accumulation tree of a set of 64 elements for $\epsilon = \frac{1}{3}$: every internal node has $4 = 64^{\epsilon}$ children, there are $3 = \frac{1}{\epsilon}$ levels in total, and there are $64^{1-i/3}$ nodes at level $i = 0, 1, 2, 3$

a B-tree [15]. However there are some differences: First, every internal node of the accumulation tree, instead of having a constant upper bound on its degree, it has a bound that is a function of the number of its leaves, n; also, its depth is always maintained to be constant, namely $O\left(\frac{1}{\epsilon}\right)$. Note that it is straightforward to construct the accumulation tree when n^{ϵ} is an integer (see Fig. 1); whenever this is not the case, we define the accumulation tree to be the (unique) complete tree of degree $\lceil n^{\epsilon} \rceil$ (by assuming a certain ordering of the leaves). This maintains the degree of internal nodes to be $O(n^{\epsilon})$.

Using the accumulation tree and search keys stored at the internal nodes, one can search for an element in $O(n^{\epsilon})$ time and perform updates in $O(n^{\epsilon})$ *amortized* time. Indeed, as the depth of the tree is not allowed to vary, one should periodically (e.g., when the number of elements of the tree doubles) rebuild the tree spending $O(n)$ time. Actually, by using individual binary trees to index the search keys within each internal node, queries could be answered in $O(\log n)$ time and updates could be processed in $O(\log n)$ amortized time. Yet, the reason we build this flat tree is not to use it as a search structure, but rather to design an authentication structure for defining the digest of $\mathcal{X}$ that matches the *optimal querying performance* of hash tables. The idea is as follows: we wish to hierarchically employ an RSA or bilinear-map accumulator over the subsets (of accumulation values) defined by each internal node in the accumulation tree, so that (non)-membership proofs of size proportional to the depth of the tree (hence of constant size) are defined with respect the root digest (accumulation value of the entire set).

Of course, applying this intuitive idea to the design of ADS schemes entails many challenges related to the nested computation of accumulation values as well as the complexity and security analysis of the final schemes. The next two sections provide all the details on how to overcome these challenges.

3 ADS Scheme Based on the RSA Accumulator

In this section we present a secure ADS scheme for the hash table data structure that employs the RSA accumulator as a main authentication primitive. We refer to this scheme as $\mathcal{RHT} = \{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$. We next elaborate on the descriptions of these algorithms along with their complexity and security analysis. Our scheme $\mathcal{RHT}$ comes in two flavors, one plain that computes witnesses on demand, and one that makes use of precomputed witnesses; accordingly, for each algorithm two constructions are often described.

3.1 Main Authenticated Data Structure

We start with the description of the key-generation and setup algorithms.

Algorithm $\{\mathsf{sk}, \mathsf{pk}\} \leftarrow \mathsf{genkey}(1^k)$: The algorithm picks a constant $0 < \epsilon < 1$ and $\lceil 1/\epsilon \rceil + 1$ RSA moduli $N_i = p_i q_i$ $(i = 0, \ldots, l)$, where p_i, q_i are strong primes [11] and $l = \lceil 1/\epsilon \rceil$. The length of the RSA moduli is defined by the recursive relation

$$|N_{i+1}| = 3|N_i| + 1,$$

where $|N_0| = 3k + 1$ and $i = 0, \ldots, l - 1$. The algorithm also picks $l + 1$ public bases $g_i \in \mathbb{QR}_{N_i}$ to be used for exponentiation. Finally, given $l + 1$ families of two-universal hash functions $\mathcal{H}_0, \mathcal{H}_2, \ldots, \mathcal{H}_l$, the algorithm randomly picks one function $h_i \in \mathcal{H}_i$, for $i = 0, \ldots, l$ (h_i will be used for computing prime representatives). The function h_i is such that it maps $(|N_i| - 1)$-bit primes to $((|N_i| - 1)/3)$-bit integers. The algorithm sets $\mathsf{sk} = \{\phi(N_i) = (p_i - 1)(q_i - 1) : i = 0, \ldots, l\}$ and $\mathsf{pk} = \{N_i, g_i, h_i : i = 0, \ldots, l; \epsilon\}$. Since $1/\epsilon$ is constant, all RSA moduli have size that only depends on the security parameter k, thus the algorithm runs in $O(1)$ time to produce output of $O(1)$ size.

We note that in the above key-generation algorithm, the restrictions on the domains and ranges of functions h_i and on the lengths of moduli N_i are due to the requirement that prime representatives are smaller numbers than the respective moduli (see [49]). (In Sect. 3.5, we discuss how to use ideas from [2] to avoid increasing size of the RSA moduli but instead allow all N_i's be of the same size; this is achieved using cryptographic hashing.)

Let now $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ be a set of n elements. $\mathcal{X}$ is stored in a dynamic hash table D_0 by using a two-universal hash function $H(\cdot)$ that maps each element to a certain bucket (see Theorem 1). Specifically, the hash table D_0 has $m = O(n)$ buckets $L_1, L_2, \ldots, L_m$, where each bucket contains $O(1)$ elements *in expectation* (by the property of the two-universal hash function—see Relation 1). Let now $0 < \epsilon < 1$ be the fixed constant chosen by algorithm $\mathsf{setup}()$. We build the accumulation tree $T(\epsilon)$ on top of the buckets, i.e., every leaf of the tree corresponds to a specific bucket and *not* to an element within the bucket. Since the number of buckets is $m = O(n)$, the internal nodes of the accumulation tree have $O(n^\epsilon)$ children.

Our authenticated data structure $\mathsf{auth}(D_0)$ is constructed with respect to the accumulation tree $T(\epsilon)$ by hierarchically employing the RSA accumulator over the sets defined by the buckets of the hash table D_0 and the parent-child relations in $T(\epsilon)$, to finally augment the accumulation tree with a collection of corresponding *accumulation values*. That is, assuming the setup parameters are in place, for any node v in the accumulation tree we define its accumulation value $\chi(v)$ recursively along the tree structure induced by $T(\epsilon)$, as a function of the accumulation values of its children (in a similar way as in a Merkle tree). We detail algorithm $\mathsf{setup}()$ below.

Algorithm $\{\mathsf{auth}(D_0), d_0\} \leftarrow \mathsf{setup}(D_0, \mathsf{sk}, \mathsf{pk})$: The algorithm first builds the accumulation tree $T(\epsilon)$ on top of the m buckets $L_1, L_2, \ldots, L_m$ of hash table D_0. For every leaf node v in tree $T(\epsilon)$ that lies at level 0 and corresponds to a bucket L_j, the algorithm

computes (and stores at v)

$$\chi(v) = g_0^{\prod_{x \in L_j} r_0(x)} \mod N_0 \in \mathbb{Z}^*_{N_0}. \tag{6}$$

Then, for every non-leaf node v in $T(\epsilon)$ that lies at level $1 \leq i \leq l$, the algorithm computes (and stores at v)

$$\chi(v) = g_i^{\prod_{u \in \mathcal{N}(v)} r_i(\chi(u))} \mod N_i \in \mathbb{Z}^*_{N_i}, \tag{7}$$

where $r_i(a)$ is a prime representative of a computed using function h_i, $g_i \in \mathbb{QR}_{N_i}$ and $\mathcal{N}(v)$ denotes the set of children of node v. The algorithm then outputs the authenticated data structure $\mathsf{auth}(D_0)$ and its digest d_0 computed as follows. If r denotes the root of $T(\epsilon)$, then $d_0 = \chi(r)$, i.e., the digest is the $\chi(\cdot)$ value of the root of the accumulation tree. Finally, $\mathsf{auth}(D_0)$ consists of the following components:

1. The accumulation tree $T(\epsilon)$;
2. For all levels i and for all nodes $v \in T(\epsilon)$ at level i, the prime representatives $r_i(\chi(v))$ that correspond to the values $\chi(v)$, such that $h_i(r_i(\chi(v))) = \chi(v)$ (as used in Relations 6 and 7).

Precomputed witnesses In order to achieve optimal querying complexity, $\mathsf{setup}()$ can also compute *precomputed* witnesses. Namely, for every node v of the accumulation tree that lies at level $0 \leq i \leq l$, let $\mathcal{N}(v)$ be the set of its children, if v is a non-leaf node, or otherwise the set of the elements in the corresponding bucket. For every $j \in \mathcal{N}(v)$ the algorithm computes (and stores at v)

$$\mathsf{W}_{j(v)} = \chi(v)^{r_i(\chi(j))^{-1}} = g_i^{\prod_{u \in \mathcal{N}(v) - \{j\}} r_i(\chi(u))} \mod N_i. \tag{8}$$

With precomputed witnesses, $\mathsf{auth}(D_0)$ is augmented to also include $\mathsf{W}_{j(v)}$, for all $v \in T(\epsilon)$ and all $j \in \mathcal{N}(v)$.

Lemma 4 *Algorithm* $\mathsf{setup}()$ *of ADS scheme* $\mathcal{RHT}$ *has* $O(n)$ *complexity both with and without precomputed witnesses. Moreover, the authenticated data structure* $\mathsf{auth}(D_0)$ *output by* $\mathsf{setup}()$ *has* $O(n)$ *size.*

Proof For a node v that has degree d, computing $\chi(v)$ from Relation 7 has $O(d)$ complexity. At level $i \geq 1$, there are $O(m^{1-i\epsilon})$ such nodes, of degree $O(m^{\epsilon})$, where m is the number of the buckets (at level 0 there are m nodes of constant degree). Since $m = O(n)$ and $T(\epsilon)$ has $O(1)$ levels, the complexity without precomputed witnesses is $O(n)$. For a node v at level i that has degree d, computing $\mathsf{W}_{j(v)}$ for all $j \in \mathcal{N}(v)$ from Relation 8 has $O(d)$ complexity: Compute $\chi(v)$ first, and then set

$$\mathsf{W}_{j(v)} = \chi(v)^{r_i(\chi(j))^{-1}} \mod N_i.$$

Note that the computation of the inverse in the exponent is feasible because $\mathsf{setup}()$ has access to the secret key which contains the factorization (trapdoor) $\phi(N_0)$ (we will see that this computational task requires more work when the factorization is

not available). Therefore the complexity of setup() with precomputed witnesses is also $O(n)$, since computing one such witness requires $O(1)$ work and there are $O(n)$ such witnesses. Finally, every node of $T(\epsilon)$ stores one group element (and two group elements in the precomputed witnesses case). Since the tree $T(\epsilon)$ has $O(n)$ nodes, the size of $\mathsf{auth}(D_0)$ is $O(n)$. □

3.2 Updates

We now describe how updates can be efficiently supported in our scheme $\mathcal{RHT}$. Our scheme stores n set elements by employing a hash table with $m = O(n)$ buckets (at the lowest level of the accumulation tree), thus, as explained in Sect. 2, the hash table is expected to be completely rebuilt at some point by the update algorithms update() and refresh() (i.e., by rehashing all elements and reinserting them in a bigger or smaller hash table), which in turn would require also rebuilding the corresponding authenticated data structure. We thus consider the following *update rule*, by adopting and appropriately adjusting in our setting the rebuilding technique by Cormen et al. [15]:

Definition 6 (*Update rule*) Let m be the number of buckets of authenticated hash table $\mathsf{auth}(D_h)$ storing set $\mathcal{X}$ and let n be the number of elements that will be contained in $\mathcal{X}$ after a given update will be performed in $\mathsf{auth}(D_h)$. Define $\alpha = \frac{n}{m}$ to be the *load factor* of the authenticated hash table $\mathsf{auth}(D_{h+1})$. If $\alpha = 1$ (full table) the capacity of hash table D_{h+1} is doubled. If $\alpha = \frac{1}{4}$ (near empty table) the capacity of hash table D_{h+1} is halved.

Our update algorithms use the above rule in their rebuilding strategy. This is essential to achieve the necessary amortized analysis results described by Lemmas 5 and 7, which in turn constitute the main complexity results in our work. We detail algorithms update() and refresh() below.

Algorithm $\{D_{h+1}, \mathsf{auth}(D_{h+1}), d_{h+1}, \mathsf{upd}\} \leftarrow \mathsf{update}(u, D_h, \mathsf{auth}(D_h), d_h, \mathsf{sk}, \mathsf{pk})$: Let $\alpha = n/m$ be the load factor of $\mathsf{auth}(D_{h+1})$ as specified in Definition 6. We distinguish two cases:
Case 1. $\frac{m}{4} < n < m$: In this case there is no need to rebuild the table and the update is performed as follows. If the update is insertion of element e, the algorithm computes the bucket $j = H(e)$ (see Relation 1) and inserts e in bucket j; let v_0 be the node of $T(\epsilon)$ referring to bucket j and $r_0(e)$ be a new prime representative for element e computed using function h_0, i.e., $h_0(r_0(e)) = e$. If the update is deletion of element e, let v_0 be the node of $T(\epsilon)$ referring to the bucket from which e is deleted. Let $v_0, v_1, \ldots, v_l$ be the path in $T(\epsilon)$ from node v_0 to the root of the tree. The algorithm initially computes (and stores at v_0) for an insertion

$$\chi'(v_0) = \chi(v_0)^{r_0(e)} \mod N_0,$$

or computes (and stores at v_0) for a deletion

$$\chi'(v_0) = \chi(v_0)^{r_0(e)^{-1}} \mod N_0. \tag{9}$$

to replace the accumulation value that corresponds to the updated bucket. Subsequently, for $j = 1, \ldots, l$ the algorithm computes (and stores at node v_j)

$$\chi'(v_j) = \chi(v_j)^{r_j(\chi'(v_{j-1}))r_j(\chi(v_{j-1}))^{-1}} \mod N_j, \tag{10}$$

where $r_j(\chi(v_{j-1}))$ is the prime representative of the old accumulation value $\chi(v_{j-1})$ and $r_j(\chi'(v_{j-1}))$ is a new prime representative for the updated accumulation value $\chi'(v_{j-1})$ such that

$$h_j(r_j(\chi'(v_{j-1}))) = \chi'(v_{j-1}),$$

both of which such accumulation values corresponding to (the lower) tree level $j-1$ (most recently updated). The algorithm then outputs the updated hash table D_{h+1}, the updated authenticated hash table $\mathsf{auth}(D_{h+1})$, its updated digest d_{h+1} and some information upd, computed as follows. First, D_{h+1} is the hash table resulted from D_h by updating an element in one of its buckets as described above. Then, $\mathsf{auth}(D_{h+1})$ is $\mathsf{auth}(D_h)$ with prime $r_j(\chi(v_{j-1}))$ replaced by new prime $r_j(\chi'(v_{j-1}))$ for all $j = 1, \ldots, l$. Also, its new digest is the updated $\chi(\cdot)$ value of the root of $T(\epsilon)$, i.e., $d_{h+1} = \chi'(v_l)$. Finally, information upd consists of the new prime representatives $r_0(e)$ and $r_j(\chi'(v_{j-1}))$ $(j = 1, \ldots, l)$ (lying along the path from the updated bucket to the root of the tree) as well as of accumulation value $\chi'(v_l)$. The behavior of the algorithm in the case where precomputed witnesses are used is the same with the difference that upd is empty.

Case 2. $n = \frac{m}{4}$ or $n = m$: In this case the hash table is rebuilt according to the update rule of Definition 6. If $n = \frac{m}{4}$, then the algorithm builds a data structure D_{h+1} with $m/2$ buckets. Otherwise, i.e., when $n = m$, the algorithm builds a data structure D_{h+1} with $2m$ buckets. Subsequently, it outputs $\mathsf{auth}(D_{h+1})$, d_{h+1} and upd by calling algorithm $\mathsf{setup}(D_{h+1}, \mathsf{sk}, \mathsf{pk})$ and setting upd to empty.

Lemma 5 *By using the update rule of Definition 6, algorithm* $\mathsf{update}()$ *of ADS scheme* $\mathcal{RHT}$ *has* $O(1)$ *expected amortized complexity. Moreover, the update information* upd *output by* $\mathsf{update}()$ *has* $O(1)$ *size.*

Proof The $O(1)$ expected complexity bound comes from the fact that the number of operations that $\mathsf{update}()$ performs (as well as the number of the group elements included in upd) is always a function of $l = 1/\epsilon = O(1)$, and that the actual hash table update has expected $O(1)$ complexity. Note that performing the operations in Relations 9 and 10 has also constant complexity, since sk contains the factorizations $\phi(N_i)$ and, therefore, inverses can be computed with the extended Euclidean algorithm. When the hash table has to be rebuilt, algorithm $\mathsf{setup}()$ is called, which has $O(n)$ complexity by Lemma 4. Finally, this constant total complexity is expected amortized: Using the rebuilding strategy of Definition 6, the amortized analysis in [15] applies directly to the analysis of algorithm $\mathsf{update}()$, especially because of the linear complexity of algorithm $\mathsf{setup}()$. □

Algorithm $\{D_{h+1}, \mathsf{auth}(D_{h+1}), d_{h+1}\} \leftarrow \mathsf{refresh}(u, D_h, \mathsf{auth}(D_h), d_h, \mathsf{upd}, \mathsf{pk})$: Let $\alpha = n/m$ be the load factor of $\mathsf{auth}(D_{h+1})$ as specified in Definition 6. We distinguish two cases:
Case 1. $\frac{m}{4} < n < m$: If the update is insertion of element e, the algorithm computes the bucket $j = H(e)$ (see Relation 1) and inserts e in bucket j; let v_0 be the node of $T(\epsilon)$ referring to bucket j. If the update is deletion of element e, let v_0 be the node of $T(\epsilon)$ referring to the bucket from which e is deleted. Let $v_0, v_1, \ldots, v_l$ be the path in $T(\epsilon)$ from node v_0 to the root of the tree. The algorithm, for $j = 0, \ldots, l$, sets

$$r_j(\chi(v_j)) = r_j(\chi'(v_j)),$$

i.e., it updates the prime representatives that correspond to the updated path by the prime representatives contained in upd. (Note that upd is *not* required for $\mathsf{refresh}()$ to perform this task but only used for efficiency: Alternatively, updated values $r_j(\chi(v_j))$ could be computed by performing explicit exponentiations with $O(n^\epsilon)$ complexity.) Finally the algorithm outputs the updated hash table as D_{h+1}, the updated prime representatives $r_j(\chi(v_j))$ (along with the ones that belong to the nodes that are not updated) as $\mathsf{auth}(D_{h+1})$ and $\chi'(v_l)$ (contained in upd) as d_{h+1}.

Precomputed witnesses When precomputed witnesses are used, the algorithm should update $\mathsf{W}_{j(v)}$ for $v = v_0, v_1, \ldots, v_l$ and for all $j \in \mathcal{N}(v)$ (see Relation 8). To achieve that efficiently, the following result from Sander *et al.* [49] for maintaining updated precomputed witnesses is used:

Lemma 6 (Computing witnesses [49]) *Given set of elements* $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$, *an RSA modulus* N *and* $g \in \mathbb{QR}_N$, *and without knowledge of* $\phi(N)$, *witnesses* $\mathsf{W}_i = g^{\prod_{j \neq i} x_j} \mod N$ *for* $i = 1, \ldots, n$ *can be computed with* $O(n \log n)$ *complexity.*

Then, in order to compute the updated witnesses, the algorithm uses the above result for all nodes $v_i, 0 \leq i \leq l$, and for all $j \in \mathcal{N}(v_i)$, as follows. For each v_i, it uses the result from Lemma 6 with inputs the *updated* elements $\{r_i(\chi(j)) : j \in \mathcal{N}(v_i)\}$, the RSA modulus N_i and the exponentiation base g_i. In this computation the updated prime representative $r_i(\chi(v_{i-1}))$, computed with $O(n^\epsilon)$ exponentiations, is used (note that $O(n^\epsilon)$ exponentiations are required since sk is not available). This computation outputs the witnesses $\mathsf{W}_{j(v_i)}$ for $j \in \mathcal{N}(v_i)$ (note that the witness $\mathsf{W}_{v_{i-1}(v_i)}$, for $i > 0$, remains the same). Also, since the algorithm for updating witnesses [49] is run on $O(1/\epsilon)$ nodes v with $|\mathcal{N}(v)| = O(n^\epsilon)$, we have, by Lemma 6, that the witnesses update complexity is $O(n^\epsilon \log n)$ (for the complete result see Lemma 7).
Case 2. $m = \frac{m}{4}$ or $n = m$: In this case the hash table is rebuilt according to the update rule of Definition 6. If $n = \frac{m}{4}$, then the algorithm builds a data structure D_{h+1} with $m/2$ buckets. Otherwise, i.e., when $n = m$, the algorithm builds a data structure D_{h+1} with $2m$ buckets. Subsequently, it outputs $\mathsf{auth}(D_{h+1})$ and d_{h+1} by using Relations 6 and 7. In the case of precomputed witnesses, the algorithm computes the new witnesses to be included in $\mathsf{auth}(D_{h+1})$ by using Lemma 6 (note that $\mathsf{refresh}()$ cannot call $\mathsf{setup}()$ directly since it does not have access to the secret key sk).

Lemma 7 *By using the update rule of Definition 6, algorithm* refresh() *of ADS scheme* $\mathcal{RHT}$ *has* $O(1)$ *or* $O(n^{\epsilon}\log n)$ *expected amortized complexity, without or, respectively, with precomputed witnesses.*

Proof For the case when no precomputed witnesses are used, the argument is the same as in Lemma 5. For the case of precomputed witnesses, suppose there are currently n elements in the hash table and that the capacity of the table (i.e., number of buckets) is m. Note that, by the rebuilding policy of Definition 6, it is $m/4 < n < m$. As we know, each one of the m buckets stores $O(1)$ elements in expectation. When an update takes place and no rebuilding of the table is triggered, all the witnesses along the path of the update of the accumulation tree have to be updated. By using the algorithm described in Lemma 6, the witnesses within the bucket can be updated in expected complexity $O(1)$, since the size of the bucket is an expected value. The witnesses of the internal nodes can be updated in $O(m^{\epsilon}\log m)$ complexity and therefore the overall complexity is $O(m^{\epsilon}\log m)$ in expectation. When a rebuilding of the table is triggered then the total complexity is $O(m\log m)$, since there is a constant number of levels in the accumulation tree, processing each node has complexity $O(m^{\epsilon}\log m)$ (since the degree of any internal node is $O(m^{\epsilon})$) and the maximum number of nodes that lie in any level is $O(m^{1-\epsilon})$. Therefore, the *actual complexity* of an update is *expected* $O(m^{\epsilon}\log m)$, when no rebuilding is triggered and $O(m\log m)$ otherwise. We are interested in the expected value of the amortized complexity (expected amortized complexity) of an update. Let n_i be the number of elements contained in the hash table after update i and m_i be the number of buckets after update i. We do the analysis by defining the following potential function:

$$F_i = \begin{cases} c(2n_i - m_i)\log m_i, & \alpha_i \geq \frac{1}{2} \\ c\left(\frac{m_i}{2} - n_i\right)\log m_i, & \alpha_i < \frac{1}{2} \end{cases},$$

where $\alpha_i = \frac{n_i}{m_i}$. The amortized complexity for an update i will be equal to $\hat{\gamma}_i = \gamma_i + F_i - F_{i-1}$. Therefore $\mathbb{E}[\hat{\gamma}_i] = \mathbb{E}[\gamma_i] + F_i - F_{i-1}$, since F_i is a deterministic function. To perform the analysis more precisely we define some constants. Let c_1 be that constant such that if the update complexity C is $O(m_i^{\epsilon}\log m_i)$, it is

$$C \leq c_1 m_i^{\epsilon}\log m_i. \tag{11}$$

Also, let r_1 be that constant such that if the rebuilding complexity R is $O(n_i\log n_i)$, it is

$$R \leq r_1 n_i \log n_i. \tag{12}$$

Also we note that in all cases it holds

$$\frac{m_i}{4} \leq n_i \leq m_i. \tag{13}$$

We perform the analysis by distinguishing the following cases:

1. $\alpha_{i-1} \geq \frac{1}{2}$ (insertion). For this case, we examine the cases where the hash table is rebuilt or not. In case the hash table is not rebuilt, we have $m_{i-1} = m_i$ and $n_i = n_{i-1} + 1$. Therefore the amortized complexity will be:

$$\begin{aligned}\mathbb{E}[\hat{\gamma}_i] &= \mathbb{E}[\gamma_i] + F_i - F_{i-1} \\ &\leq c_1 m_i^{\epsilon} \log m_i + c(2n_i - m_i - 2n_{i-1} + m_{i-1}) \log m_i \\ &= c_1 m_i^{\epsilon} \log m_i + 2c \log m_i .\end{aligned}$$

In case the hash table is rebuilt (which takes $O(n \log n)$ complexity in total) we have $m_i = 2m_{i-1}, n_i = n_{i-1}+1$ and $n_{i-1} = m_{i-1}$ (which give $n_i = m_i/2+1 \leq m_i/2$) and the amortized complexity will be:

$$\begin{aligned}\mathbb{E}[\hat{\gamma}_i] &= \mathbb{E}[\gamma_i] + F_i - F_{i-1} \\ &\leq r_1 n_i \log n_i + c(2n_i - m_i) \log m_i - c(2n_{i-1} - m_{i-1}) \log m_{i-1} \\ &= r_1 n_i \log n_i + c(2n_i - m_i) \log m_i - c\frac{m_i}{2} \log m_i/2 \\ &\leq r_1 \frac{m_i}{2} \log m_i/2 + 2c \log m_i - c\frac{m_i}{2} \log m_i/2 \\ &\leq 2c \log m_i\end{aligned}$$

for a constant c of the potential function such that $c > r_1$.

2. $\alpha_{i-1} < \frac{1}{2}$ (insertion). Note that that there is no way that the hash table is rebuilt in this case. Therefore $m_{i-1} = m_i$ and $n_i = n_{i-1} + 1$. If now $\alpha_i < \frac{1}{2}$ the amortized complexity will be:

$$\begin{aligned}\mathbb{E}[\hat{\gamma}_i] &= \mathbb{E}[\gamma_i] + F_i - F_{i-1} \\ &\leq c_1 m_i^{\epsilon} \log m_i + c(m_i/2 - n_i) \log m_i - c(m_{i-1}/2 - n_{i-1}) \log m_{i-1} \\ &= c_1 m_i^{\epsilon} \log m_i + c(m_i/2 - n_i - m_i/2 + n_{i-1}) \log m_i \\ &= c_1 m_i^{\epsilon} \log m_i - c \log m_i .\end{aligned}$$

In case now $\alpha_i \geq \frac{1}{2}$ the amortized complexity will be:

$$\begin{aligned}\mathbb{E}[\hat{\gamma}_i] &= \mathbb{E}[\gamma_i] + F_i - F_{i-1} \\ &\leq c_1 m_i^{\epsilon} \log m_i + c(2n_i - m_i) \log m_i - c(m_{i-1}/2 - n_{i-1}) \log m_{i-1} \\ &= c_1 m_i^{\epsilon} \log m_i + c(2(n_{i-1} + 1) - m_{i-1} - m_{i-1}/2 + n_{i-1}) \log m_i \\ &= c_1 m_i^{\epsilon} \log m_i + c(3n_{i-1} - 3m_{i-1}/2 + 2) \log m_i \\ &= c_1 m_i^{\epsilon} \log m_i + c(3\alpha m_{i-1} - 3m_{i-1}/2 + 2) \log m_i \\ &< c_1 m_i^{\epsilon} \log m_i + c(3m_{i-1}/2 - 3m_{i-1}/2 + 2) \log m_i \\ &= c_1 m_i^{\epsilon} \log m_i + 2c \log m_i .\end{aligned}$$

3. $\alpha_{i-1} < \frac{1}{2}$ (deletion). Here we have $n_i = n_{i-1} - 1$. In case the hash table does not have to be rebuilt (i.e., $\frac{1}{4} < \alpha_i < \frac{1}{2}$ and $m_i = m_{i-1}$), we have that the amortized

complexity of the deletion is going to be:

$$\begin{aligned}\mathbb{E}[\hat{\gamma}_i] &= \mathbb{E}[\gamma_i] + F_i - F_{i-1}\\ &\le c_1 m_i^{\epsilon}\log m_i + c(m_i/2 - n_i)\log m_i - c(m_{i-1}/2 - n_{i-1})\log m_{i-1}\\ &= c_1 m_i^{\epsilon}\log m_i + c(m_i/2 - n_i - m_i/2 + n_{i-1})\log m_i\\ &= c_1 m_i^{\epsilon}\log m_i + c\log m_i.\end{aligned}$$

In case now the hash table has to be rebuilt (which has $O(n_i \log n_i)$ complexity), we have that $m_i = m_{i-1}/2$, $m_i = 4n_i$ and therefore the amortized complexity is:

$$\begin{aligned}\mathbb{E}[\hat{\gamma}_i] &= \mathbb{E}[\gamma_i] + F_i - F_{i-1}\\ &\le r_1 n_i \log n_i + c(m_i/2 - n_i)\log m_i - c(m_{i-1}/2 - n_{i-1})\log m_{i-1}\\ &\le r_1 n_i \log n_i + c(m_i/2 - n_i)\log m_i - c(m_i - (n_i + 1))\log 2m_i\\ &\le r_1 n_i \log n_i - c(m_i/2 - 1)\log m_i - c(3n_i - 1)\\ &\le r_1 n_i \log n_i - cm_i/2\log m_i + c\log m_i\\ &\le r_1 m_i \log m_i - (c/2)m_i \log m_i + c\log m_i\\ &\le c\log m_i,\end{aligned}$$

where c must also be chosen to satisfy $c > 2r_1$.

4. $\alpha_{i-1} \ge \frac{1}{2}$ (deletion). In this case we have $m_{i-1} = m_i$. If $\alpha_i \ge \frac{1}{2}$, the amortized complexity will be:

$$\begin{aligned}\mathbb{E}[\hat{\gamma}_i] &= \mathbb{E}[\gamma_i] + F_i - F_{i-1}\\ &\le c_1 m_i^{\epsilon}\log m_i + c(2n_i - m_i - 2n_{i-1} + m_{i-1})\log m_i\\ &\le c_1 m_i^{\epsilon}\log m_i - 2c\log m_i.\end{aligned}$$

Finally for the case that $\alpha_i < \frac{1}{2}$ we have

$$\begin{aligned}\mathbb{E}[\hat{\gamma}_i] &= \mathbb{E}[\gamma_i] + F_i - F_{i-1}\\ &\le c_1 m_i^{\epsilon}\log m_i + c(m_{i-1}/2 - n_i - 2n_{i-1} + m_{i-1})\log m_i\\ &= c_1 m_i^{\epsilon}\log m_i + c(3m_{i-1}/2 - (n_{i-1} - 1) - 2n_{i-1})\log m_i\\ &= c_1 m_i^{\epsilon}\log m_i + c(3m_{i-1}/2 - 3n_{i-1} + 1)\log m_i\\ &= c_1 m_i^{\epsilon}\log m_i + c(3(1/\alpha_{i-1})n_{i-1}/2 - 3n_{i-1} + 1)\log m_i\\ &\le c_1 m_i^{\epsilon}\log m_i + c\log m_i.\end{aligned}$$

Therefore we conclude that for all constants $c > 2r_1$ of the potential function, the expected value of the amortized complexity of any operation is bounded by

$$\mathbb{E}[\hat{\gamma}_i] \le c_1 m_i^{\epsilon}\log m_i + 2c\log m_i.$$

By using now Relation 13, there is a constant r such that $\mathbb{E}[\hat{\gamma}_i] \le r n_i^{\epsilon}\log n_i$ which implies that the expected value of the amortized complexity of any update (inser-

tion/deletion) in an authenticated hash table containing n elements is $O(n^{\epsilon} \log n)$ for $0 < \epsilon < 1$. □

3.3 Queries and Verification

We now present how a proof for an element $e \in \mathcal{X}$ (or an element $e \notin \mathcal{X}$) can be constructed, by using the authenticated data structure described in the previous subsection. Let $H(e) = j$, i.e., the bucket that corresponds to element e is j. Let $v_0, v_1, \ldots, v_l$ be the path from the node that corresponds to bucket j to the root of $T(\epsilon)$. We add a fictitious node v_{-1} that stores element e within bucket j such that $v_{-1}, v_0, v_1, \ldots, v_l$ is the path in $T(\epsilon)$ from the node to corresponds to element e to the root of $T(\epsilon)$. We consider two cases, one for *membership* and one for *non-membership* proofs:

– *Element e is contained in the hash table* The proof is the ordered sequence $\pi_0, \pi_1, \ldots, \pi_l$, where π_i is a tuple of a prime representative and a witness that authenticates every node of the path $v_{-1}, v_0, \ldots, v_l$ from the element in question e to the root of the tree v_l. Thus, item π_i of proof $\Pi(e)$ $(i = 0, \ldots, l)$ is defined as:

$$\pi_i = \left(r_i(\chi(v_{i-1})), \mathsf{W}_{v_{i-1}(v_i)}\right), \tag{14}$$

where $\mathsf{W}_{v_{i-1}(v_i)}$ is defined in Relation 8 and $\chi(v_{-1}) = e$. For simplicity, we set $\alpha_i = r_i(\chi(v_{i-1}))$ and

$$\beta_i = \mathsf{W}_{v_{i-1}(v_i)}. \tag{15}$$

For example in Fig. 1, the proof for an element that belongs to the bucket of node a (e.g., element 2) consists of the following tuples:

$$\begin{aligned}
\pi_0 &= \left(r_0(2), g^{r_0(3)r_0(7)r_0(9)} \mod N_0\right), \\
\pi_1 &= \left(r_1(\chi(a)), g_1^{r_1(\chi(b))r_1(\chi(c))r_1(\chi(d))} \mod N_1\right), \\
\pi_2 &= \left(r_2(\chi(f)), g_2^{r_2(\chi(e))r_2(\chi(g))r_2(\chi(p))} \mod N_2\right).
\end{aligned}$$

– *Element e is not contained in the hash table* Let $y_1, y_2, \ldots, y_u$ be the elements contained in bucket j (all different than e). First, output a *membership proof* (as above) for an element y_i in bucket j (note that $H(y_i) = H(e)$). Then, and by running the extended Euclidean algorithm, output a non-membership witness

$$\pi_v = (\mathsf{A}_e, \mathsf{B}_e, r_0(e), e), \tag{16}$$

where $\mathsf{A}_e, \mathsf{B}_e$ and $r_0(e)$ are defined in Relation 3. Note that $\mathsf{A}_e, \mathsf{B}_e$ are integer values proving non-membership of e in the set $\{y_1, y_2, \ldots, y_u\}$.

We now describe the algorithm formally:

Algorithm $\{\Pi(q), \alpha(q)\} \leftarrow \mathsf{query}(q, D_h, \mathsf{auth}(D_h), \mathsf{pk})$: Let $e = q$ be the queried element. If e is *contained* in D_h, the algorithm computes $\Pi(q) = (\pi_0, \pi_1, \ldots, \pi_l)$, as in Relation 14 and outputs $\alpha(q) = \mathsf{true}$. If e is *not contained* in D_h, it computes a membership proof $\Pi(y_i)$ for some other element y_i in bucket j, such that $H(e) = H(y_i)$, computes a non-membership proof π_v for e in bucket j, as defined in Relation 16, and outputs $\Pi(q) = (\Pi(y_i), \pi_v)$ and $\alpha(q) = \mathsf{false}$.

Lemma 8 *Algorithm* query() *of ADS scheme* $\mathcal{RHT}$ *has* $O(n^\epsilon)$ *or* $O(1)$ *expected complexity, without or, respectively, with precomputed witnesses. Moreover, the proof* $\Pi(q)$ *output by* query() *has* $O(1)$ *expected size.*

Proof (a) *Membership proof*: Without precomputed witnesses, the construction of π_0 has always $O(1)$ expected complexity since each bucket contains $O(1)$ elements in expectation. Also, the construction of each element π_i $(i = 1, \ldots, l)$ has $O(n^\epsilon)$ complexity due to the degree bound of the nodes in $T(\epsilon)$. Therefore the total complexity is expected $O(n^\epsilon)$. With precomputed witnesses, each π_i can be "read" directly from memory with $O(1)$ complexity (not expected). Finally, the size of $\Pi(q)$ for a membership proof is $O(1)$ (not expected), since one witness for each level of $T(\epsilon)$ must be provided. (b) *Non-membership proof*: A non-membership proof consists of a membership proof (thus the above arguments apply here as well) plus the proof π_v (see Relation 16). Therefore in both cases (without precomputed witnesses and with precomputed witnesses), π_v turns the complexities into *expected*, since the complexity of A_e and B_e depends on the number of the elements in the bucket in question (see observation before Sect. 2.6), which is expected $O(1)$. This completes the proof. □

We finally complete the presentation of our scheme $\mathcal{RHT}$ by formally describing the verification algorithm which takes as input a proof and an answer and either accepts or rejects the answer.

Algorithm accept or reject $\leftarrow \mathsf{verify}(q, \alpha, \Pi, d_h, \mathsf{pk})$: Let the query q refer to element e, i.e., $q = e$. We distinguish two cases:

1. *Membership proof*: In this case it is $\alpha = \mathsf{true}$. The proof Π contains a membership proof for e, denoted with $\Pi(e) = \pi_0, \pi_1, \ldots, \pi_l$, where $\pi_i = (\alpha_i, \beta_i)$ for $i = 0, \ldots, l$, and where α_i are all primes. The algorithm outputs reject if any of the following conditions are true:
   (a) $h_0(\alpha_0) \neq e$ (the prime representative of element e is not correct);
   (b) $h_i(\alpha_i) \neq \beta_{i-1}^{\alpha_{i-1}} \mod N_{i-1}$ for some $1 \leq i \leq l$ (false witness);
   (c) $d_h \neq \beta_l^{\alpha_l} \mod N_l$ (final digest mismatch).
2. *Non-membership proof*: In this case it is $\alpha = \mathsf{false}$. The proof Π in this case contains (a) a membership proof $\Pi(y) = \pi_0, \pi_1, \ldots, \pi_l$ for element $y \neq e$ such that $H(y) = H(e)$, where $\pi_i = (\alpha_i, \beta_i)$ for $i = 0, \ldots, l$ (α_i are all primes); (b) a non-membership proof for e, denoted with $\pi_v = (\mathsf{A}, \mathsf{B}, r, e)$, where r is a prime. The algorithm outputs reject if any of the following conditions are true:
   (a) $H(e) \neq H(y)$ (e and y do not belong in the same bucket);
   (b) reject $\leftarrow \mathsf{verify}(y, \mathsf{true}, \Pi(y), d_h, \mathsf{pk})$ (the membership proof for y is rejected);

(c) $h_0(r) \neq e$ (the prime representative r contained in π_v for element e is not correct);
(d) $\alpha_1^{\mathsf{A}} g_0^{r\mathsf{B}} \neq g_0 \mod N_0$ (verification test for non-membership proof of e in the corresponding bucket does not succeed, see Lemma 2).

If all the above tests are successful, the algorithm outputs accept.

Lemma 9 *Algorithm* verify() *of ADS scheme* $\mathcal{RHT}$ *has* $O(1)$ *expected complexity.*

Proof Processing the membership proof has $O(1)$ complexity, since it requires processing $l = O(1)$ pairs of witnesses and prime representatives. Moreover, processing the non-membership proof has $O(1)$ expected complexity, due to the size of the non-membership witness, that depends on the number of the elements in the bucket in question. Therefore, the expected complexity of verify() is $O(1)$.

3.4 Correctness and Security

The following lemmas describe the correctness and security properties of our scheme $\mathcal{RHT}$, according to Definitions 3 and 4. The security of our scheme is based on the strong RSA assumption.

Lemma 10 *The ADS scheme* $\mathcal{RHT}$ = {genkey, setup, update, refresh, query, verify} *is correct according to Definition 3.*

Proof Let D_0 be any hash table containing n elements and having $m = O(n)$ buckets. Fix the security parameter k and output $\mathsf{pk} = \{N_i, g_i, h_i : i = 0, \ldots, l; \epsilon\}$ and $\mathsf{sk} = \{\phi(N_i) : i = 0, \ldots, l\}$ by calling algorithm genkey(). Then output an authenticated data structure auth(D_0) and the respective digest d_0, by calling algorithm setup(). Pick a polynomial number of updates—namely, pick a polynomial number of elements for insertion or deletion—and update auth(D_0) and d_0 by calling algorithm refresh(). Let D_h be the final hash table, auth(D_h) be the produced authenticated data structure and d_h be the final digest. Let e be an element that belongs (or, it should belong) to bucket j (i.e., $H(e) = j$). Output a proof $\Pi(e)$ and an answer by calling query(). We distinguish two cases:

1. *Element e is contained in the hash table* Then $\Pi(e)$ is a membership proof as defined in Relation 14. Note that π_0 contains the prime representative of e with the respective witness, therefore verify() does not reject at condition 1a. By the definitions of the accumulation values that are output by setup() and maintained under updates by refresh() (see Relations 6 and 7), and by the definition of proof element π_i in Relation 14 for $i = 1, \ldots, l$, verify() does not reject at conditions 1b and 1c;
2. *Element e is not contained in the hash table* Let $y_1, y_2, \ldots, y_u$ be the elements in bucket j, where $H(e) = j$. In this case, the non-membership proof consists of (a) a membership proof $\pi_i = (\alpha_i, \beta_i)$ for an element y contained in bucket j (which verifies due to condition 1) such that $H(e) = H(y) = j$, therefore verify() does not reject at either condition 2a or condition 2b and, (b) a non-membership proof $(\mathsf{A}_e, \mathsf{B}_e, r_0(e), e)$ for element e that *should* belong to bucket j. Therefore verify()

does not reject at condition 2c, since $h_0(r_0(e)) = e$. Also it does not reject at condition 2d as

$$\alpha_1^{\mathsf{A}_e} g_0^{r_0(e)\mathsf{B}_e} = g_0^{\left(\prod_{j=1}^{u} r_0(y_j)\right)\mathsf{A}_e + r_0(e)\mathsf{B}_e} = g_0 \mod N_0,$$

since, by construction it is

$$\alpha_1 = g_0^{\prod_{j=1}^{u} r_0(y_j)} \mod N_0,$$

and by Relation 3, A_e and B_e are computed to satisfy

$$\left(\prod_{j=1}^{u} r_0(y_j)\right)\mathsf{A}_e + r_0(e)\mathsf{B}_e = 1.$$

This completes the proof. □

Lemma 11 *The ADS scheme* $\mathcal{RHT}$ = {genkey, setup, update, refresh, query, verify} *is secure according to Definition 4 and under the strong RSA assumption.*

Proof Let k be the security parameter. Output $\mathsf{pk} = \{N_i, g_i, h_i : i = 0, \ldots, l; \epsilon\}$ and $\mathsf{sk} = \{\phi(N_i) : i = 0, \ldots, l\}$ by calling algorithm genkey(). Let Adv be a polynomially-bounded adversary. First, Adv picks an initial set $\mathcal{X}$ of n elements, stored in hash table D_0. Next, Adv outputs an authenticated data structure $\mathsf{auth}(D_0)$, by calling algorithm setup() through oracle access. Then Adv picks a polynomial number of updates—namely, he picks a polynomial number of elements for insertion or deletion. Let D_h be the final hash table, let the updated final set of element be $\mathcal{X}$, and let d_h be the final digest, all as produced by the adversary through oracle access to algorithm update(). We will compute the probability that **check**() rejects, while verify() accepts, as required by Definition 4. We distinguish two cases:

1. *Membership proof*: The adversary outputs a membership proof $\Pi(e) = (\pi_0, \pi_1, \ldots, \pi_l)$ $(l = \lceil \frac{1}{\epsilon} \rceil)$ where $\pi_i = (\alpha_i, \beta_i)$ (see algorithm query()) for an element $e \notin \mathcal{X}$ (thus, a proof for an incorrect answer). Let $v_0, v_1, \ldots, v_l$ be a path of nodes in $T(\epsilon)$ from the bucket referring to e to the root of the tree. We next define some events that are related to the choice made by the adversary in constructing proof $\Pi(e)$ above. Our goal will be to the express the probability that verify(e, true, $\Pi(e)$, d_h, pk) accepts and $e \notin \mathcal{X}$ as a function of these events. Note that d_h is the correct digest of the authenticated data structure:
   (a) $\mathcal{E}_{0,0}$: The value e and α_0 picked by Adv are such that $e \notin \mathcal{X}$, α_0 is prime and $h_0(\alpha_0) = e$;
   (b) $\mathcal{E}_j$: For $j = 1, \ldots, l$, the values α_j, α_{j-1} and β_{j-1} picked by Adv are such that both α_j and α_{j-1} are primes and

$$h_j(\alpha_j) = \beta_{j-1}^{\alpha_{j-1}} \mod N_{j-1} \text{ for all } 1 \le j \le l.$$

This event can be partitioned into two mutually exclusive events, i.e., $\mathcal{E}_j = \mathcal{E}_{j,0} \cup \mathcal{E}_{j,1}$ such that
- $\mathcal{E}_{j,0}$: Value $h_j(\alpha_j)$ is *not* the correctly formed digest (i.e., an accumulation of the digests of its children) of some node $v_{j-1} \in \mathcal{N}(v_j)$, as defined in Relation 7;
- $\mathcal{E}_{j,1}$: Value $h_j(\alpha_j)$ is the correctly formed digest of a node $v_{j-1} \in \mathcal{N}(v_j)$, as defined in Relation 7.

(c) $\mathcal{E}_{l+1,1}$: The values α_l and β_l picked by Adv are such that

$$\beta_l^{\alpha_l} = d_h \mod N_l.$$

The probability that $\mathsf{verify}()$ accepts, while $e \notin \mathcal{X}$ is the probability

$$\begin{aligned}
&\Pr[\mathcal{E}_{0,0} \cap \mathcal{E}_1 \cap \mathcal{E}_2 \cap \ldots \cap \mathcal{E}_{l+1,1}] \\
&= \Pr[\mathcal{E}_{0,0} \cap (\mathcal{E}_{1,0} \cup \mathcal{E}_{1,1}) \cap (\mathcal{E}_{2,0} \cup \mathcal{E}_{2,1}) \cap \ldots \cap \mathcal{E}_{l+1,1}] \\
&\leq \Pr[\mathcal{E}_{1,1}|\mathcal{E}_{0,0}] + \Pr[\mathcal{E}_{2,1}|\mathcal{E}_{1,0}] + \Pr[\mathcal{E}_{3,1}|\mathcal{E}_{2,0}] + \ldots + \Pr[\mathcal{E}_{l+1,1}|\mathcal{E}_{l,0}] \\
&= \Pr[\mathcal{E}_{1,1}|\mathcal{E}_{0,0}] + \sum_{j=2}^{l+1} \Pr[\mathcal{E}_{j,1}|\mathcal{E}_{j-1,0}]. \qquad (17)
\end{aligned}$$

First we examine the event $\mathcal{E}_{1,1}|\mathcal{E}_{0,0}$. This event implies that the adversary has found a value $e \notin \mathcal{X}$, a prime α_0 such that $h_0(\alpha_0) = e$ and a value β_0 such that

$$\beta_0^{\alpha_0} = g_0^{\prod_{t=1,\ldots,l'} r_0(x_t)} \mod N_0,$$

where $x_1, x_2, \ldots, x_{l'}$ is a subset of the set $\mathcal{X}$. Since $e \notin \mathcal{X}$, it is $e \notin \{x_1, x_2, \ldots, x_{l'}\}$. Also, since every prime representative is mapped to a *unique* element through function h_0, we conclude that it should be $\alpha_0 \notin \{r_0(x_1), r_0(x_2), \ldots, r_0(x_{l'})\}$. By Lemma 2 and Assumption 1, this probability is $\mathsf{neg}(k)$. Therefore $\Pr[\mathcal{E}_{1,1}|\mathcal{E}_{0,0}] \leq \mathsf{neg}(k)$.

For the remaining events $\mathcal{E}_{j,1}|\mathcal{E}_{j-1,0}$ $(2 \leq j \leq l+1)$, we have:
- By the one-to-one property of the function $h_{j-1}(\cdot)$, $\mathcal{E}_{j-1,0}$ implies that value α_{j-1} is *not* the prime representative of the correctly formed digest of some node $v_{j-2} \in \mathcal{N}(v_{j-1})$, as defined in Relation 7, namely that

$$\alpha_{j-1} \notin \{r_{j-1}(\chi(v_t)) : v_t \in \mathcal{N}(v_{j-1})\}.$$

- However, the event $\mathcal{E}_{j,1}$ implies that (1) digest $h_j(\alpha_j)$ (for $j = l+1$ this is just d_h) is the correctly formed digest of node v_{j-1}; and (2)

$$\beta_{j-1}^{\alpha_{j-1}} = g_{j-1}^{\prod_{v_t \in \mathcal{N}(v_{j-1})} r_{j-1}(\chi(v_t))} \mod N_{j-1},$$

where $r_{j-1}(\chi(v_t))$ are the prime representatives of correctly formed digests of the set of neighbors of v_{j-1}.

Since $\alpha_{j-1} \notin \{r_{j-1}(\chi(v_t)) : v_t \in \mathcal{N}(v_{j-1})\}$, by Lemma 2 and Assumption 1, this probability is $\mathsf{neg}(k)$. Therefore for all $j = 1, \ldots, l+1$, $\Pr[\mathcal{E}_{j,1}|\mathcal{E}_{j-1,0}]$ is $\mathsf{neg}(k)$. Since $l = O(1)$, the total probability is $\mathsf{neg}(k)$.

2. *Non-membership proof*: For this case, we define the events:
   (a) $\mathcal{B}_0$: Adv finds $e \in \mathcal{X}$ and y such that $H(e) = H(y) = j$;
   (b) $\mathcal{B}_1$: Adv finds a membership proof for y, namely the proof $\pi = \pi_0, \pi_1, \ldots, \pi_l$ where $\pi_i = (\alpha_i, \beta_i)$ (where α_i are prime numbers) and $\mathsf{accept} \leftarrow \mathsf{verify}(y, \mathsf{true}, \pi, d_h, \mathsf{pk})$;
   (c) $\mathcal{B}_2$: Adv finds α_1, a non-membership proof $(\mathsf{A}, \mathsf{B}, r, e)$ for e such that r is a prime number, $h_0(r) = e$ and $\alpha_1^{\mathsf{A}} g_0^{r\mathsf{B}} = g_0 \mod N_0$. This event is partitioned into two events:
      i. $\mathcal{B}_{20} : \alpha_1 \neq \mathsf{acc}(L_j) = g_0^{\prod_{x \in L_j} r_0(x)} \mod N_0$;
      ii. $\mathcal{B}_{21} : \alpha_1 = \mathsf{acc}(L_j) = g_0^{\prod_{x \in L_j} r_0(x)} \mod N_0$.

   We need to compute the probability $\Pr[\mathcal{B}_0 \cap \mathcal{B}_1 \cap \mathcal{B}_2] = \Pr[\mathcal{B}_0 \cap \mathcal{B}_1 \cap (\mathcal{B}_{20} \cup \mathcal{B}_{21})] \leq \Pr[\mathcal{B}_{20}|\mathcal{B}_1|\mathcal{B}_0] + \Pr[\mathcal{B}_{21}|\mathcal{B}_0]$. By Lemma 2 and Assumption 1, we have that $\Pr[\mathcal{B}_{21}|\mathcal{B}_0]$ is $\mathsf{neg}(k)$. Note also that we can express $\mathcal{B}_{20}|\mathcal{B}_1|\mathcal{B}_0$ as a function of the events $\mathcal{E}$ in the membership proof case. Specifically, the event $\mathcal{B}_{20}|\mathcal{B}_1|\mathcal{B}_0$ implies the event $\mathcal{E}_{1,0} \cap \mathcal{E}_2 \cap \mathcal{E}_3 \cap \ldots \cap \mathcal{E}_{l+1,1}$, the probability of which, by following the same logic as in Relations 17 is (bounded by) $\mathsf{neg}(k)$.

This concludes the proof for both the membership and the non-membership cases. □

We can now present the main result of this section.

Theorem 2 *Let k be the security parameter and $0 < \epsilon < 1$. Then there exists an ADS scheme $\mathcal{RHT} = \{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$ for a data structure scheme defined for dynamic hash table D storing n elements such that:*

1. *It is correct according to Definition 3 and secure according to Definition 4 under the strong RSA assumption;*
2. *The complexity of* $\mathsf{setup}()$ *is $O(n)$, outputting an authenticated data structure* $\mathsf{auth}(D)$ *of $O(n)$ size;*
3. *The expected amortized complexity of* $\mathsf{update}()$ *is $O(1)$, outputting update information* upd *of $O(1)$ size;*
4. *The expected amortized complexity of* $\mathsf{refresh}()$ *is $O(1)$ (or $O(n^\epsilon \log n)$);*
5. *The expected complexity of* $\mathsf{query}()$ *is $O(n^\epsilon)$ (or $O(1)$), outputting proof $\Pi(q)$ for query q of $O(1)$ expected size;*
6. *The expected complexity of* $\mathsf{verify}()$ *is $O(1)$.*

Proof This result follows directly from Lemmas 4, 5, 7, 8, 9, 10 and 11. The complexities that appear in parentheses, namely $O(n^\epsilon \log n)$ for $\mathsf{refresh}()$ and $O(1)$ for $\mathsf{query}()$, refer to the case when precomputed witnesses are used. The presented scheme $\mathcal{RHT}$ is publicly verifiable since $\mathsf{verify}()$ does not take the secret key as an input. □

3.5 A More Practical Scheme

The scheme that we have presented uses different RSA moduli for each level of the tree and each new RSA modulus has a bit-length that is three times longer than the bit-

length of the previous-level RSA modulus. Therefore, computations corresponding to higher levels in the accumulation tree are more expensive, since they involve modular arithmetic operations over longer elements.

To overcome this complexity overhead, we wish to use the same RSA modulus for each level of the tree, and to achieve this, we present a heuristic inspired by a similar method originally used in the work of Baric and Pfitzmann [2]. Instead of using two-universal hash functions to map (general) integers to primes of increased size, the idea is to employ random oracles [3] for consistently computing primes of relatively small size. In particular, given a k-bit integer x, instead of mapping it to a $3k$-bit prime, we can map it to a prime in the interval $[a, 2^t a]$, where $a = g(x)$ (g is a random oracle like SHA-256) and t is some appropriately chosen integer. We detail the exact choice of this interval with the following theorem.

Theorem 3 *Let x be a k-bit integer and let $a = g(x) \in \{2, 3, \ldots, 2^b\}$ be the output of a b-bit random oracle. If $2^t \geq 2b + \frac{2^{b+1} b \ln 2}{2^b - 1}$, then the interval $[a, 2^t a]$ contains a prime with probability at least $1 - 2^{-b}$.*

Proof By the Prime Distribution Theorem we have that the number of primes less than n is approximately $\frac{n}{\ln n}$. Therefore, if $\mathsf{PRIMES}(a, t)$ is the number of primes in the interval $[a, 2^t a]$, we have

$$\mathsf{PRIMES}(a, t) = \frac{2^t a}{\ln(2^t a)} - \frac{a}{\ln a} = \frac{2^t a}{\ln a + t \ln 2} - \frac{a}{\ln a}. \tag{18}$$

Since now a is the output of the described b-bit random oracle, it is $2 \leq a \leq 2^b$. Therefore

$$\ln 2 \leq \ln a \leq b \ln 2 \Leftrightarrow \frac{1}{\ln 2} \geq \frac{1}{\ln a} \geq \frac{1}{b \ln 2} \Leftrightarrow -\frac{1}{\ln 2} \leq -\frac{1}{\ln a} \leq -\frac{1}{b \ln 2}. \tag{19}$$

By plugging Relation 19 into Relation 18, and since $t < b$, we have

$$\mathsf{PRIMES}(a, t) \geq \frac{2^t a}{(b + t) \ln 2} - \frac{a}{\ln 2} > \frac{2^t a}{2b \ln 2} - \frac{a}{\ln 2}.$$

Therefore, we compute the probability

$$\begin{aligned}
\Pr[\mathsf{PRIMES}(a, t) \geq 1] &\geq \Pr\left[\frac{2^t a}{2b \ln 2} - \frac{a}{\ln 2} \geq 1\right] \\
&= \Pr\left[a \geq \frac{2b \ln 2}{2^t - 2b}\right] \text{ (by hypothesis we can divide by } \\
&\quad 2^t - 2b > 0) \\
&\geq 1 - \Pr\left[a \leq \frac{2b \ln 2}{2^t - 2b}\right] \text{ (simple probability rule)}
\end{aligned}$$

$$= 1 - \frac{1}{2^b - 1}\left(\frac{2b \ln 2}{2^t - 2b}\right) \text{ (by the property of the random oracle)}$$

$$\geq 1 - \frac{1}{2^b} \text{ (by hypothesis, since } 2^t - 2b \geq \frac{2^{b+1} b \ln 2}{2^b - 1}\text{)}.$$

This completes the proof. □

Using Theorem 3, we can pick the length of the output of the random oracle to ensure hitting a prime with high probability. For instance, it is enough to set $t = 10$ for $b = 256$.

4 ADS Scheme Based on the Bilinear-Map Accumulator

In this section we present a second secure ADS scheme for the hash table data structure that this time employs the bilinear-map accumulator as a main authentication primitive. We refer to this scheme as $\mathcal{BHT} = \{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$. Similar to scheme $\mathcal{RHT}$, the new scheme $\mathcal{BHT}$ uses an identical methodology with the one described in Sect. 3.

We note, however, that the use of different accumulators imposes a few significant differences both in the complexity and in the cryptographic design of new scheme. For example, we now operate over a fundamentally different underlying algebraic group defined over elliptic curves (and not over $\mathbb{Z}_N$ as before), which in turn imposes certain differences in the complexity of some algorithms that will be discussed later.

4.1 Main Authenticated Data Structure

We begin by presenting the algorithms $\mathsf{genkey}()$ and $\mathsf{setup}()$. As before, the underlying (plain) data structure is a dynamic hash table $\mathbf{T}(\mathcal{X})$ storing n elements $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ which are distributed into $m = O(n)$ buckets $L_1, L_2, \ldots, L_m$ (using a two-universal hash function H).

Algorithm $\{\mathsf{sk}, \mathsf{pk}\} \leftarrow \mathsf{genkey}(1^k)$: The algorithm first runs the bilinear pairing instance generator on input k to generate a uniformly random tuple $\mathsf{t} = (p, \mathbb{G}, \mathcal{G}, e, g)$ of bilinear pairings parameters (as described in Sect. 2.5, where p is a k-bit prime. Then it randomly picks a number $s \in \mathbb{Z}_p^*$ as the trapdoor, selects an upper bound q of the total number of elements that will be accumulated, and also computes elements

$$g^s, g^{s^2}, \ldots, g^{s^q}$$

in $\mathbb{G}$. Finally, the algorithm defines a function $h : \mathbb{G} \rightarrow \mathbb{Z}_p$ that outputs the bit-description of elements in $\mathbb{G}$. (This function allows to accumulate (in a higher level) accumulation values in $\mathbb{G}$ as elements in $\mathbb{Z}_p$. Since $\mathbb{G}$ has exactly p elements, $h(\cdot)$ maps each element in $\mathbb{G}$ to an integer in $\mathbb{Z}_p$, and in order to simplify notation, we

assume that $h(x) = x$ whenever $x \in \mathbb{Z}_p$). The algorithm outputs $s \in \mathbb{Z}_p^*$ as sk and everything else as pk.

Algorithm $\{\mathsf{auth}(D_0), d_0\} \leftarrow \mathsf{setup}(D_0, \mathsf{sk}, \mathsf{pk})$: The algorithm builds the accumulation tree $T(\epsilon)$ on top of the m buckets $L_1, L_2, \ldots, L_m$. For every leaf node v in tree $T(\epsilon)$ that lies at level 0 and corresponds to a bucket L_j, the algorithm computes

$$\chi(v) = g^{\prod_{x \in L_j}(x+s)} \in \mathbb{G}, \tag{20}$$

while for every non-leaf node v in $T(\epsilon)$ that lies at level $1 \leq i \leq l$, the algorithm computes

$$\chi(v) = g^{\prod_{u \in \mathcal{N}(v)}(h(\chi(u))+s)} \in \mathbb{G}, \tag{21}$$

where $h(\chi(u))$ is an element in $\mathbb{Z}_p$, computed using function $h(\cdot)$. The algorithm then outputs the authenticated data structure $\mathsf{auth}(D_0)$ and its digest d_0 computed as follows. If r denotes the root of $T(\epsilon)$, then $d_0 = \chi(r)$, i.e., the digest is the $\chi(\cdot)$ value of the root of the accumulation tree. Finally, $\mathsf{auth}(D_0)$ consists of the following components:

1. The accumulation tree $T(\epsilon)$;
2. For all levels i and for all nodes $v \in T(\epsilon)$ at level i, the accumulation value $\chi(v)$.

Precomputed witnesses When precomputed witnesses are used, for every $j \in \mathcal{N}(v)$, the algorithm additionally computes (and stores at v) the witness

$$\mathsf{W}_{j(v)} = g^{\prod_{u \in \mathcal{N}(v)-\{j\}}(h(\chi(u))+s)}. \tag{22}$$

Accordingly, $\mathsf{auth}(D_0)$ is augmented to also include $\mathsf{W}_{j(v)}$, for all $v \in T(\epsilon)$ and all $j \in \mathcal{N}(v)$.

Lemma 12 *Algorithm* $\mathsf{setup}()$ *of ADS scheme* $\mathcal{BHT}$ *has* $O(n)$ *complexity both with and without precomputed witnesses. Moreover, the authenticated data structure* $\mathsf{auth}(D_0)$ *output by* $\mathsf{setup}()$ *has* $O(n)$ *size.*

Proof It follows from Lemma 4 with the difference that the efficient computation of the exponent expressions is now feasible since sk contains the trapdoor s. □

4.2 Updates

We continue by presenting the algorithms used for updates where we use the update rule of Definition 6.

Algorithm $\{D_{h+1}, \mathsf{auth}(D_{h+1}), d_{h+1}, \mathsf{upd}\} \leftarrow \mathsf{update}(u, D_h, \mathsf{auth}(D_h), d_h, \mathsf{sk}, \mathsf{pk})$: The algorithm is very similar to the equivalent algorithm in scheme $\mathcal{RHT}$. Let $\alpha = n/m$ be the load factor of $\mathsf{auth}(D_{h+1})$ as specified in Definition 6. We distinguish two cases:

Case 1. $\frac{m}{4} < n < m$: The algorithm computes the bucket $j = H(e)$ that is related to the update u on element $e \in \mathbb{Z}_p$; if v_0 is the node in $T(\epsilon)$ referring to bucket j, let

$v_0, v_1, \ldots, v_l$ be the path in $T(\epsilon)$ from node v_0 to the root of the tree. The algorithm initially computes (and stores at v_0) for an insertion

$$\chi'(v_0) = \chi(v_0)^{e+s},$$

or computes (and stores at v_0) for a deletion

$$\chi'(v_0) = \chi(v_0)^{(e+s)^{-1}}, \tag{23}$$

to replace the accumulation value that corresponds to the updated bucket. Subsequently, for $j = 1, \ldots, l$ the algorithm computes (and stores at v_j)

$$\chi'(v_j) = \chi(v_j)^{(h(\chi'(v_{j-1}))+s)(h(\chi(v_{j-1}))+s)^{-1}}, \tag{24}$$

where $\chi(v_{j-1})$, $\chi'(v_{j-1})$ are the previous and, respectively, the updated accumulation value. The algorithm then outputs the updated D_{h+1}, resulted by updating e in bucket j, the updated $\mathsf{auth}(D_{h+1})$, resulted by replacing in $\mathsf{auth}(D_h)$ accumulation value $\chi(v_{j-1})$ with $\chi'(v_{j-1})$ for all $j = 1, \ldots, l$, the updated $d_{h+1} = \chi'(v_l)$, as well as some information upd that consists of element e and values $\chi'(v_j)$ $(j = 0, \ldots, l)$. The behavior of the algorithm in the *precomputed witnesses* case is the same, with the difference that upd is empty.

Case 2. $m = \frac{m}{4}$ or $n = m$: In this case the new hash table D_{h+1} is rebuilt according to the update rule of Definition 6. Then, the algorithm outputs $\mathsf{auth}(D_{h+1}), d_{h+1}$ and upd by calling algorithm $\mathsf{setup}(D_{h+1}, \mathsf{sk}, \mathsf{pk})$ and setting $\mathsf{upd} = \{\mathsf{auth}(D_{h+1}), d_{h+1}\}$.

Lemma 13 *By using the update rule of Definition 6, algorithm* $\mathsf{update}()$ *of ADS scheme* $\mathcal{BHT}$ *has* $O(1)$ *expected amortized complexity. Moreover, the update information* upd *output by* $\mathsf{update}()$ *has* $O(1)$ *amortized size.*

Proof The proof is the same as in Lemma 5. However, the complexity of upd is *amortized*, because when the hash table is rebuilt, it contains the new authenticated data structure (of complexity $O(n)$).

We now state a complexity result that is needed to analyse the remaining algorithms in scheme $\mathcal{BHT}$. It is derived by an FFT algorithm (e.g., see Preparata and Sarwate [47]) that computes the DFT of any n-size input using $O(n \log n)$ field operations in a finite field (e.g., in $\mathbb{Z}_p$, actually, without requiring existence of an n-th root of unity in $\mathbb{Z}_p$).

Lemma 14 (Polynomial interpolation with FFT [47]) *Let* $\prod_{i=1}^{n}(s + x_i) = \sum_{i=0}^{n} a_i s^i$ *be a degree-n polynomial. The coefficients* $a_n, a_{n-1}, \ldots, a_0$ *can be computed with* $O(n \log n)$ *complexity, given* $x_1, x_2, \ldots, x_n$.

Algorithm $\{D_{h+1}, \mathsf{auth}(D_{h+1}), d_{h+1}\} \leftarrow \mathsf{refresh}(u, D_h, \mathsf{auth}(D_h), d_h, \mathsf{upd}, \mathsf{pk})$: Let $\alpha = n/m$ be the load factor of $\mathsf{auth}(D_{h+1})$ as specified in Definition 6. We distinguish two cases:

Case 1. $\frac{m}{4} < n < m$: The algorithm computes the bucket $j = H(e)$ that is related to the update u on element $e \in \mathbb{Z}_p$; if v_0 is the node in $T(\epsilon)$ referring to bucket j, let $v_0, v_1, \ldots, v_l$ be the path in $T(\epsilon)$ from node v_0 to the root of the tree. Then it updates the accumulation values on this path using information upd, setting for $j = 0, \ldots, l$

$$\chi(v_j) = \chi'(v_j).$$

(Not required to perform this task, upd is again only used for efficiency: Alternatively, updated values $\chi(v_j)$ could be computed through polynomial interpolation with $O(n^\epsilon \log n)$ complexity.) Finally it outputs the updated D_{h+1}, the updated $\mathsf{auth}(D_{h+1})$ and the updated d_{h+1} (simply by using the new accumulation values contained in upd).

Precomputed witnesses When precomputed witnesses are used, the algorithm should update $\mathsf{W}_{j(v)}$ for $v = v_0, v_1, \ldots, v_l$ and for all $j \in \mathcal{N}(v)$ (see Relation 8). To achieve that efficiently, the following result is used (by appropriately extending the witness update techniques in [37]):

Lemma 15 (Updating witnesses) *For a set of elements* $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$, *if* $\mathsf{W}_i = g^{\prod_{j \neq i}(x_j + s)}$ *is the witness of* x_i, $i = 1, 2, \ldots, n$, *then the following hold:*

1. *(Element addition) If* $\mathcal{X}' = \mathcal{X} \cup \{x_{n+1}\}$, *then for all* $i = 1, \ldots, n+1$ *it is*

$$\mathsf{W}'_i = \mathsf{acc}(\mathcal{X})\mathsf{W}_i^{x_{n+1} - x_i}. \tag{25}$$

2. *(Element deletion) If* $\mathcal{X}' = \mathcal{X} - \{x_j\}$, *then for all* $i \neq j$ *it is*

$$\mathsf{W}'_i = \left(\frac{\mathsf{W}_i}{\mathsf{W}_j}\right)^{\frac{1}{x_j - x_i}}. \tag{26}$$

3. *(Element modification) If* $\mathcal{X}' = \mathcal{X} - \{x_j\} \cup \{x'_j\}$, *then for all* $i \neq j$ *it is*

$$\mathsf{W}'_i = \mathsf{W}_j \left(\frac{\mathsf{W}_i}{\mathsf{W}_j}\right)^{\frac{x'_j - x_i}{x_j - x_i}}. \tag{27}$$

For $i = j$, *it is* $\mathsf{W}'_i = \mathsf{W}_i$.

Proof Relations 25 and 26 are given in the initial work of Nguyen [37]. Relation 27 is derived as a corollary of Relations 25 and 26. Indeed, for all $i \neq j$:

$$\begin{aligned}
\mathsf{W}_j \left(\frac{\mathsf{W}_i}{\mathsf{W}_j}\right)^{\frac{x'_j - x_i}{x_j - x_i}} &= g^{\prod_{x \in \mathcal{X} - \{x_j\}}(x+s)} \left(\frac{g^{\prod_{x \in \mathcal{X} - \{x_i\}}(x+s)}}{g^{\prod_{x \in \mathcal{X} - \{x_j\}}(x+s)}}\right)^{\frac{x'_j - x_i}{x_j - x_i}} \\
&= g^{\prod_{x \in \mathcal{X} - \{x_j\}}(x+s)} \left(g^{(x_j - x_i)\prod_{x \in \mathcal{X} - \{x_j, x_i\}}(x+s)}\right)^{\frac{x'_j - x_i}{x_j - x_i}}
\end{aligned}$$

$$
\begin{aligned}
&= g^{\prod_{x\in\mathcal{X}-\{x_j\}}(x+s)}\, g^{(x'_j-x_i)\prod_{x\in\mathcal{X}-\{x_j,x_i\}}(x+s)} \\
&= g^{\prod_{x\in\mathcal{X}-\{x_j\}}(x+s)}\, g^{-x_i\prod_{x\in\mathcal{X}-\{x_j,x_i\}}(x+s)}\, g^{x'_j\prod_{x\in\mathcal{X}-\{x_j,x_i\}}(x+s)} \\
&= g^{s\prod_{x\in\mathcal{X}-\{x_j,x_i\}}(x+s)}\, g^{x'_j\prod_{x\in\mathcal{X}-\{x_j,x_i\}}(x+s)} \\
&= g^{\prod_{x\in\mathcal{X}'-\{x_i\}}(x+s)} \\
&= \mathsf{W}'_i.
\end{aligned}
$$

For $i = j$, the witness W_j does not change since, by definition, W_j is not a function of the value of j (x_j or x'_j). This completes the proof. □

Corollary 1 (Updating precomputed witnesses) *Given set of elements* $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ *and witnesses* $\mathsf{W}_i = g^{\prod_{j\neq i}(x_j+s)}$, $i = 1, \ldots, n$, *and without the knowledge of the trapdoor* s, *updated witnesses* W'_i *of updated set* $\mathcal{X} \cup \{x_{n+1}\}$, $\mathcal{X} - \{x_j\}$ *or* $\mathcal{X} - \{x_j\} \cup \{x'_j\}$ *can be computed with* $O(n)$ *complexity.*

Then, algorithm refresh() computes the updated witnesses by appropriately applying one of the transformations of Lemma 15 to each previous witnesses W_i, which is stored at each node v_i, for $i = 1, \ldots, l$. In particular, Relations 25 or 26 is used to update the witnesses within the bucket of the update (depending on whether there is an element addition or deletion). Also, Relation 27 is used to update the witnesses corresponding internal nodes of the tree: For an internal node v with children $v_1, v_2, \ldots, v_t$, if the accumulation value $\chi(v_j)$ of v_j is modified, then the sets $\mathcal{X}$ and $\mathcal{X}'$ in Relation 27 are as follows:

$$\mathcal{X} = \{h(\chi(v_1)), h(\chi(v_2)), \ldots, h(\chi(v_t))\},$$

and

$$\mathcal{X}' = \{h(\chi(v_1)), h(\chi(v_2)), \ldots, h(\chi(v_{j-1})), h(\chi'(v_j)), h(\chi(v_{j+1})), \ldots, h(\chi(v_t))\}.$$

Case 2. $m = \frac{m}{4}$ or $n = m$: In this case the new hash table D_{h+1} is rebuilt according to the update rule of Definition 6. The algorithm outputs auth(D_{h+1}) and d_{h+1} using information upd (which includes all new witnesses).

Lemma 16 *By using the update rule of Definition 6, algorithm* refresh() *of ADS scheme* $\mathcal{BHT}$ *has* $O(1)$ *or* $O(n^{\epsilon})$ *expected amortized complexity, without or, respectively, with precomputed witnesses.*

Proof To prove this result, we can use the same amortized analysis as in Lemma 7 with two differences: The work that refresh() does when rebuilding the hash table is $O(m)$ and not $O(m \log m)$, since it copies information from upd; and the analysis makes use of the complexity bound of Corollary 1 instead of Lemma 6. □

4.3 Queries and Verification

We finish with presenting the algorithms query() and verify(). We first show now how a proof for an element $e \in \mathcal{X}$ (or an element $e \notin \mathcal{X}$) can be constructed. As before, let

$H(e) = j$ (bucket assignment for e) and let $v_0, v_1, \ldots, v_l$ be the path from the node that corresponds to bucket j to the root of $T(\epsilon)$. We recall v_{-1} is a fictitious node that stores element e within bucket j such that $v_{-1}, v_0, v_1, \ldots, v_l$ is the path in $T(\epsilon)$ from the node that corresponds to element e to the root of $T(\epsilon)$. We consider two cases, one for *membership* and one for *non-membership* proofs:

– *Element e is contained in the hash table* The proof is the ordered sequence $\pi_0, \pi_1, \ldots, \pi_l$, where π_i is a tuple of *an accumulation value* $\chi()$ and a witness that authenticates every node of the path $v_{-1}, v_0, \ldots, v_l$ from the element in question e to the root of the tree v_l. Thus, item π_i of proof $\Pi(e)$ $(i = 0, \ldots, l)$ is defined as:

$$\pi_i = \left(\chi(v_{i-1})), \mathsf{W}_{v_{i-1}(v_i)}\right), \tag{28}$$

where $\mathsf{W}_{v_{i-1}(v_i)}$ is defined in Relation 22. For simplicity, we set $\alpha_i = \chi(v_{i-1})$ (note that $\chi(v_{-1}) = e$) and

$$\beta_i = \mathsf{W}_{v_{i-1}(v_i)}. \tag{29}$$

For example in Fig. 1, the proof for an element that belongs to bucket of node a (e.g., element 2) consists of the following tuples:

$$\pi_0 = \left(2, g^{(s+3)(s+7)(s+9)}\right),$$
$$\pi_1 = \left(\chi(a), g^{(h(\chi(b))+s)(h(\chi(c))+s)(h(\chi(d))+s)}\right),$$
$$\pi_2 = \left(\chi(f), g^{(h(\chi(e))+s)(h(\chi(g))+s)(h(\chi(p))+s)}\right).$$

– *Element e is not contained in the hash table* Let $y_1, y_2, \ldots, y_u$ be the elements contained in bucket j (all different than e). First, output a *membership proof* (as above) for an element y_i in bucket j (note that $H(y_i) = H(e)$). Then, and by running the extended Euclidean algorithm for polynomials, output a non-membership witness

$$\pi_v = (\mathsf{A}_e, \mathsf{B}_e, e), \tag{30}$$

where $\mathsf{A}_e, \mathsf{B}_e$ are elements in $\mathbb{G}$ defined in Relation 5. Note that $\mathsf{A}_e, \mathsf{B}_e$ have group complexity $O(1)$ (and not *expected* $O(1)$ as in the RSA accumulator case—see Relation 3—since they consist of just one group element each) and they are used to prove non-membership of e in the set $\{y_1, y_2, \ldots, y_u\}$.

We now describe the algorithm formally:

Algorithm $\{\Pi(q), \alpha(q)\} \leftarrow \mathsf{query}(q, D_h, \mathsf{auth}(D_h), \mathsf{pk})$: Let $e = q$ be the queried element. If e is *contained* in D_h, the algorithm sets $\Pi(q) = (\pi_0, \pi_1, \ldots, \pi_l)$, as in Relation 28 and outputs $\alpha(q) = \mathsf{true}$. If e is *not contained* in D_h, it outputs a membership proof for some other element y_i in bucket j, such that $H(e) = H(y_i)$, computes a non-membership proof π_v for e in bucket j, as defined in Relation 30, and outputs $\Pi(q) = (\Pi(y_i), \pi_v)$ and $\alpha(q) = \mathsf{false}$.

Lemma 17 *Algorithm* query() *of ADS scheme* $\mathcal{BHT}$ *has* $O(n^{\epsilon} \log n)$ *or* $O(1)$ *expected complexity, without or, respectively, with precomputed witnesses. Moreover, the proof* $\Pi(q)$ *output by* query() *has* $O(1)$ *size.*

Proof The proof is the same with Lemma 8, but with the following differences:

1. Without precomputed witnesses, a witness cannot be constructed with direct exponentiation, since the trapdoor s is not known. It is constructed with polynomial interpolation as follows: Suppose the witness is $g^{(y_1+s)(y_2+s)\ldots(y_t+s)}$ (where $t = O(n^\epsilon)$). Compute $a_0, a_1, \ldots, a_t$ by using Lemma 14 ($O(n^\epsilon \log n)$ complexity). Then output the witness as

$$g^{a_0} \times \left(g^s\right)^{a_1} \times \left(g^{s^2}\right)^{a_2} \times \ldots \times \left(g^{s^t}\right)^{a_t},$$

   where $g, g^s, \ldots, g^{s^t}$ are contained in the public key. The entire computation has $O(n^\epsilon)$ complexity.
2. The proof has $O(1)$ size and not *expected* $O(1)$ size, due the compactness of the non-membership proof in the bilinear-map accumulator construction (see Relation 5).

This completes the proof. □

Algorithm accept or reject $\leftarrow$ verify$(q, \alpha, \Pi, d_h, \mathsf{pk})$: Let the query q refer to element e, i.e., $q = e$. We distinguish two cases:

1. *Membership proof*: In this case it is $\alpha = \mathsf{true}$. The proof Π should contain $\Pi(e) = \pi_0, \pi_1, \ldots, \pi_l$, i.e., the membership proof for element e, where $\pi_i = (\alpha_i, \beta_i)$. The algorithm outputs reject if any of the following conditions are true (note that the verification algorithm is using the bilinear map function $e(\cdot, \cdot)$):
   (a) $\alpha_0 \neq e$ (element α_0 is not correct);
   (b) $e(\alpha_i, g) \neq e\left(\beta_{i-1}, g^s g^{h(\alpha_{i-1})}\right)$ for some $1 \leq i \leq l$ (false witness);
   (c) $e(d_h, g) \neq e\left(\beta_l, g^s g^{h(\alpha_l)}\right)$ (final digest mismatch).
2. *Non-membership proof*: In this case it is $\alpha = \mathsf{false}$. The proof Π in this case contains $\Pi(y) = \pi_0, \pi_1, \ldots, \pi_l$, i.e., the membership proof for an element $y \neq e$, where $\pi_i = (\alpha_i, \beta_i)$ for $i = 0, \ldots, l$. It also contains $\pi_v = (\mathsf{A}, \mathsf{B}, r)$, the non-membership proof for e. The algorithm outputs reject if any of the following are true:
   (a) $H(e) \neq H(y)$; (e and y do not belong in the same bucket);
   (b) reject $\leftarrow$ verify$(y, \mathsf{true}, \Pi(y), d_h, \mathsf{pk})$ (the membership proof for y is rejected);
   (c) $r \neq e$ (the data element contained in π_v for element e is not correct);
   (d) $e(\alpha_1, \mathsf{A})\, e(g^s g^r, \mathsf{B}) \neq e(g, g)$ (verification test for non-membership proof of e does not succeed, see Lemma 3).

If all the above tests are successful, the algorithm outputs accept.

Lemma 18 *Algorithm* verify() *of ADS scheme* $\mathcal{BHT}$ *has* $O(1)$ *complexity.*

Proof We use the same proof as in Lemma 9, with the difference that the complexity is not *expected* any more, due to the compactness of the non-membership proof. □

4.4 Correctness and Security

The following lemmas describe the correctness and security properties of our scheme $\mathcal{BHT}$. The security of our scheme is based on the bilinear q-strong Diffie-Hellman assumption.

Lemma 19 *The ADS scheme* $\mathcal{BHT} = \{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$ *is correct according to Definition 3.*

Proof The proof follows the same logic with the proof of Lemma 10. As such, we only show the correctness for the non-membership proof case. Let $y_1, y_2, \ldots, y_u$ be the elements contained in the bucket where e *should belong*. The non-membership proof, as computed by $\mathsf{query}()$, that is needed for verification, is $(\mathsf{A}_e, \mathsf{B}_e, e)$. Therefore $\mathsf{verify}()$ does not reject at condition 2c, since $r = e$. Also it does not reject at condition 2d since

$$e\left(\alpha_1, \mathsf{A}_e\right) e\left(g^s g^e, \mathsf{B}_e\right) = e\left(g^{\prod_{j=1}^{u}(y_j+s)}, \mathsf{A}_e\right) e\left(g^s g^e, \mathsf{B}_e\right) = e(g, g),$$

since, by Relation 5, $\mathsf{A}_e = g^{\alpha(s)}$ and $\mathsf{B}_e = g^{\beta(s)}$ such that $\left[\prod_{j=1}^{u}(y_j + s)\right]\alpha(s) + (e + s)\beta(s) = 1$. □

Lemma 20 *The ADS scheme* $\mathcal{BHT} = \{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$ *is secure according to Definition 4 and under the bilinear q-strong Diffie-Hellman assumption.*

Proof The proof follows exactly the same logic with the proof of Lemma 11 and therefore it is omitted. □

Theorem 4 *Let k be the security parameter and* $0 < \epsilon < 1$. *Then there exists an ADS scheme* $\mathcal{BHT} = \{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$ *for a data structure scheme defined for a dynamic hash table D storing n elements such that:*

1. *It is correct according to Definition 3 and secure according to Definition 4 and under the bilinear q-strong Diffie-Hellman assumption;*
2. *The complexity of* $\mathsf{setup}()$ *is* $O(n)$*, outputting an authenticated data structure* $\mathsf{auth}(D)$ *of* $O(n)$ *complexity;*
3. *The expected amortized complexity of* $\mathsf{update}()$ *is* $O(1)$*, outputting update information* upd *of* $O(1)$ *amortized complexity;*
4. *The expected amortized complexity of* $\mathsf{refresh}()$ *is* $O(1)$ *(or* $O(n^\epsilon)$*);*
5. *The expected complexity of* $\mathsf{query}()$ *is* $O(n^\epsilon \log n)$ *(or* $O(1)$*), outputting a proof* $\Pi(q)$ *for query q of* $O(1)$ *size;*
6. *The complexity of* $\mathsf{verify}()$ *is* $O(1)$.

Proof It follows directly from Lemmas 12, 13, 16, 17, 18, 19 and 20. The complexities that appear in parentheses ($O(n^\epsilon)$ for $\mathsf{refresh}()$ and $O(1)$ for $\mathsf{query}()$) refer to the case when precomputed witnesses are used. □

5 Analysis and Evaluation

In this section we provide an evaluation of our two authenticated hash table constructions. We analyze the computational efficiency of our schemes by counting the number of modular exponentiations (in the appropriate group) involved in each of the complexity measures (namely, update, query and verification cost) and for general values of ϵ, the basic parameter in our schemes that controls the flatness of the accumulation tree. The number of exponentiations turns out to be a very good estimate of the computational complexity that our schemes have, mainly for two reasons. First, because modular exponentiations are the primitive operations performed in our authentication schemes, and, second, because there is no significant overheads due to hidden constant factors in the asymptotic complexities of our schemes—the only constant factors included in our complexities are well-understood functions of ϵ. We also analyze the communication complexity of our schemes by computing the exact sizes of the verification proofs and the update authentication information. Finally, we experimentally validate our computational and communication analysis.

We evaluate the authenticated hash table schemes described in Theorems 2 and 4, where every complexity measure is constant, except from the update time that is $O(n^{\epsilon} \log n)$ (RSA accumulator) and $O(n^{\epsilon})$ (bilinear-map accumulator) respectively. For the experiments we used a 64-bit, 2.8 GHz Intel based, dual-core, dual-processor machine with 2GB main memory and 2MB cache, running Debian Linux. For modular exponentiation, inverse computation and multiplication in the RSA accumulator scheme we used NTL [35], a standard, optimized library for number theory, interfaced with C++. For bilinear maps and generic-group operations in the bilinear-accumulator scheme, we used the PBC library [41], a library for pairing-based cryptography, interfaced with C. Finally, for both schemes, we used the efficient *sparsehash* hash table implementation from Google[1] for on-the-fly computation and efficient updates of the witnesses during a query or an update respectively.

5.1 Authenticated Hash Table Using the RSA Accumulator

As we saw in the system setup of the RSA accumulator authenticated hash table, the standard scheme uses multiple RSA moduli $N_1, N_2, \ldots, N_l$, where the size of each modulus is increasing with $1/\epsilon$. In our experimental analysis, we make use of the more practical version of our scheme that is described in Sect. 3.5. That is, we restrict the input of each level of accumulation to be the output of a cryptographic hash function, e.g., SHA-256, plus a constant number of extra bits (t bits) that, when appended to the output of the hash function, give a prime number. For the experiments we set $t = 10$ and we use a random oracle that outputs a value of length $b = 256$ bits. Therefore, the exponent in the solution that uses the RSA accumulator is $256 + 10 = 266$ bits. Note that $t = 10$ is the smallest value satisfying Theorem 3 for $b = 256$.

[1] See http://code.google.com/p/google-sparsehash/.

Primitive Operations The main (primitive) operations used in our scheme are:

1. Exponentiation modulo N;
2. Computation of inverses modulo $\phi(N)$;
3. Multiplication modulo $\phi(N)$;
4. SHA-256 computation over 1,024-bit integers.

We have benchmarked the time needed for these operations. For 200 runs, the average time for computing the power of a 1,024-bit number to a 266-bit exponent and then reducing the result modulo N was found to be $T_1 = 2.13$ ms, and the average time for computing the inverse of a 266-bit number modulo $\phi(N)$ was $T_2 = 0.000105$ ms. Similarly, multiplication of 266-bit numbers modulo $\phi(N)$ was found to be $T_3 = 0.0011$ ms. For SHA-256, we used the standard C implementation from gcrypt.h and, over 200 runs, the time to compute the 256-bit digest of a 1,024-bit number was found to be $T_4 = 0.01$ ms. Finally the *sparsehash* query and update time was benchmarked and was found to be $t_{\mathsf{table}} = 0.003$ ms. As expected, exponentiations are the most expensive operations.

Updates Let f be a function that takes as input a 1,024-bit integer x and outputs 266-bit prime, as in Theorem 3. We make the reasonable assumption that the time for applying $f(\cdot)$ to x is dominated by the SHA-256 computation—practically ignoring the time to perform the appropriate shifting—and is thus equal to $T_4 = 0.01$ ms. As we saw in Sect. 3 the updates with the trapdoor information (i.e., algorithm update()) are performed as follows. Suppose we want to delete element x in bucket L. Let $d_1, d_2, \ldots, d_l$ be the RSA digests along the path from x to the root (d_1 is the RSA digest of the corresponding bucket and d_l is the root RSA digest). We first compute

$$d_1' = d_1^{f(x)^{-1}} \mod N$$

which is the new value of the bucket. Note that this is feasible to compute, since $\phi(N)$ is known. Therefore so far, we have performed one $f(\cdot)$ computation (actually we have to to do this $f(\cdot)$ computation only during insertions, since during deletions the value $f(x)$ of the element x that is deleted has already been computed), one inverse computation and one exponentiation. Next, for each $i = 2, \ldots, l$, we compute d_i' by setting

$$d_i' = d_i^{f(d_{i-1})^{-1} f(d_{i-1}')} \mod N.$$

Since $f(d_{i-1})$ is precomputed, we need to do one $f(\cdot)$ computation, one inverse computation, one multiplication and one exponentiation. Therefore, the total update time is

$$t_{\mathsf{update}} = T_1 + T_2 + T_4 + \epsilon^{-1}(T_1 + T_2 + T_3 + T_4),$$

which is not dependent on n. During an update without the trapdoor information (i.e., algorithm refresh()) we need to compute the witnesses explicitly and, therefore, perform $\epsilon^{-1} n^{\epsilon} \log n^{\epsilon}$ exponentiations and ϵ^{-1} $f(\cdot)$ computations in total. Additionally,

Table 2 Cost expressions in our RSA accumulator scheme for $n = 100{,}000{,}000$ and various values of ϵ

Operation	Cost expression	$\epsilon = 0.1$	$\epsilon = 0.3$	$\epsilon = 0.5$
Update (ms)	$T_1 + T_2 + T_4 + \epsilon^{-1}(T_1 + T_2 + T_3 + T_4)$	24.46	10.22	7.33
Refresh (ms)	$(\epsilon^{-1} + 1)(n^{\epsilon} \log n^{\epsilon} T_1 + T_4) + (\epsilon^{-1} n^{\epsilon} + 1)t_{\mathsf{table}}$	134.75	6,303.60	289,490.00
Verify(ms)	$(\epsilon^{-1} + 1)(T_1 + T_4)$	26.59	12.28	9.46
Query(ms)	$(\epsilon^{-1} + 1)t_{\mathsf{table}}$	0.03	0.01	0.01
Proof size (KB)	$(\epsilon^{-1} + 1)(1{,}024 + 266)$	1.73	0.68	0.47

The size n of the hash table influences algorithm refresh()

after we compute the new witnesses for each internal node of the accumulation tree, these witnesses have to be stored in a hash table. Therefore,

$$t_{\mathsf{refresh}} = (\epsilon^{-1} + 1)(n^{\epsilon} \log n^{\epsilon} T_1 + T_4) + (\epsilon^{-1} n^{\epsilon} + 1)t_{\mathsf{table}}.$$

Verification The verification is performed by doing $\epsilon^{-1} + 1$ exponentiations and $f(\cdot)$ computations. Namely, by using $f(\cdot)$ to compute prime representatives, Equation 1b becomes

$$\alpha_i = f(\beta_{i-1}^{\alpha_{i-1}} \mod N).$$

Therefore,

$$t_{\mathsf{verify}} = (\epsilon^{-1} + 1)(T_1 + T_4), \tag{31}$$

which is not dependent on n.

Queries To answer queries using precomputed witnesses, the we just to pick the right witness at each level. By using an efficient hash table structure with search time t_{table} we have that

$$t_{\mathsf{query}} = (\epsilon^{-1} + 1)t_{\mathsf{table}}. \tag{32}$$

Communication complexity The proof and the update authentication information consist of $\epsilon^{-1} + 1$ pairs of 1,024-bit numbers and 266-bit $f(\cdot)$ values. Thus,

$$s_{\mathsf{prf}} = (\epsilon^{-1} + 1)(1024 + 266).$$

In order to precisely evaluate the practical efficiency of our scheme, we set $\epsilon = 0.1, 0.3, 0.5$ (modeling the cases where the accumulation tree has 10, 3, 2 levels respectively). Table 2 shows the various cost measures expressed as functions of ϵ, and the actual values these measures take on for a hash table that contains 100,000,000 elements and a varying value of ϵ (i.e., varying number of levels of the RSA tree). We can make the following observations: As ϵ increases, the verification time and the communication complexity decrease. However, update time increases since the internal nodes of the tree become larger and more exponentiations have to be performed. In

terms of communication cost, our system is very efficient since only at most 2.25 KB have to be communicated.

5.2 Authenticated Hash Table Using the Bilinear-Map Accumulator

For the analysis of our bilinear-accumulator scheme, we chose to use type A pairings, as described in [31]. These pairings are constructed on the curve $y^2 = x^3 + x$ over the base field $\mathbb{F}_q$, where q is a prime number. The multiplicative cyclic group $\mathbb{G}$ we are using is a subgroup of points in $E(\mathbb{F}_q)$, namely a subset of those points of $\mathbb{F}_q$ that belong to the elliptic curve E. Therefore this pairing is symmetric. The order of $E(\mathbb{F}_q)$ is $q + 1$ and the order of the group $\mathbb{G}$ is some prime factor p of $q + 1$. The group of the output of the bilinear map $\mathbb{G}_M$ is a subgroup of $\mathbb{F}_{q^2}$.

In order to instantiate type A pairings in the PBC library, we have to choose the size of the primes q and p. The main constraint in choosing the bit-sizes of q and p is that we want to make sure that discrete logarithm is difficult in $\mathbb{G}$ (that has order p) and in $\mathbb{F}_{q^2}$. Typical values are 160 bits for p and 512 bits for q for 80 bits of security. Since the accumulated elements in our construction are the output of SHA-256 (plus the trapdoor s), we choose the size of p to be 260 bits. We use the typical value for the size of q, i.e., 512 bits. Note that with this choice of parameters the size of the elements in $\mathbb{G}$ (which have the form (x, y), i.e., points on the elliptic curve) is 1,024 bits. The main operations we benchmarked using PBC are the following:

1. Exponentiation of an element $x \in \mathbb{G}$ to $y \in \mathbb{Z}_p^*$, which takes $t_1 = 13.7$ ms;
2. Computation of inverses modulo p, which takes $t_2 = 0.0001$ ms;
3. Multiplication modulo p, which takes $t_3 = 0.0005$ ms;
4. SHA-256 computation of 1,024-bit integers (elements of $\mathbb{G}$), which takes $t_4 = 0.01$ ms;
5. Multiplication of two elements $x, y \in \mathbb{G}$, which takes $t_5 = 0.04$ ms;
6. Bilinear map computation $e(x, y)$, where $x, y \in \mathbb{G}$, which takes $t_6 = 13.08$ ms.

Note that operations related to bilinear-map accumulators take significantly more time than the respective operations related to the RSA accumulator.

By following a similar method with that followed for the RSA accumulator, we are able to derive formulas for the exact times of the bilinear-map accumulator (see Table 3). The main differences in the cost expressions are in the server's update time, where the witnesses are computed in a different way (in addition to exponentiations, multiplications and inverse computations are also required) and in the client's verification time, where two bilinear-map computations are also performed.

5.3 Comparison

As we can see from the experimental evaluation, the RSA accumulator scheme is more efficient in practice than the bilinear-map accumulator scheme. This is due to the costly operations of applying the bilinear-map function $e(\cdot, \cdot)$ and the performing exponentiations in the group $\mathbb{G}$. However, asymptotically, the bilinear-accumulator scheme outperforms the RSA accumulator scheme by a logarithmic factor. In terms

Table 3 Cost expressions in our scheme bilinear-accumulator scheme for $n = 100{,}000{,}000$ and various values of ϵ

Operation	Cost expression	$\epsilon = 0.1$	$\epsilon = 0.3$	$\epsilon = 0.5$
Update (ms)	$t_1 + t_2 + t_4 + \epsilon^{-1}(t_1 + t_2 + t_3 + t_4)$	150.82	59.41	41.13
Refresh (ms)	$(\epsilon^{-1} + 1)[n^{\epsilon}(t_1 + t_2 + t_3) + t_4] + (\epsilon^{-1}n^{\epsilon} + 1)t_{\mathsf{table}}$	951.20	14,915.00	411,080.00
Verify(ms)	$(\epsilon^{-1} + 1)(t_1 + t_4 + 2t_6)$	438.61	172.81	119.65
Query(ms)	$(\epsilon^{-1} + 1)t_{\mathsf{table}}$	0.03	0.01	0.01
Proof size (KB)	$(\epsilon^{-1} + 1)(1{,}024 + 1{,}024)$	2.83	1.12	0.77

The size n of the hash table influences only the server's update time

of communication efficiency, we see that there is almost no difference since the size of the elements of the field $\mathbb{G}$ is 1,024 bits, equal to the size of the RSA modulus used in the RSA accumulator scheme. We note that for a system implementation of our schemes it would make sense to make constant ϵ as small as possible, since the update cost may become prohibitive for large values of ϵ. Finally, for updates, we observe the RSA scheme is far more efficient than the bilinear-map scheme.

Overall, our results are primarily of theoretical interest. From the evaluation, we can see that the cost for performing an update is much higher that the cost induced by using Merkle trees and other structures such as skip lists (see for example [21]). However, the communication complexity scales very well with the data set size and compares well with the hash-based methods. The most important property of our results is that asymptotically the client can *optimally* authenticate operations on hash tables with *constant* time and communication complexities; this makes our scheme suitable for certain applications where verification for example should not depend on the size of the data we are authenticating.

6 Conclusions and Open Problems

In this paper, we propose a new provably secure cryptographic construction for authenticating the fundamental hash-table functionality. We use nested cryptographic accumulators on a tree of constant depth to achieve *constant* query and verification costs as well as *sublinear* update costs. We use our method to authenticate general set-membership queries and overall improve over previous techniques that use cryptographic accumulators, reducing the main complexity measures to constant, yet keeping sublinear update time.

An important open problem is whether one can achieve logarithmic update cost and still keep the communication complexity constant. There has been no such solution to date. In particular, no method is known that can construct constant-size accumulator proofs (witnesses) in logarithmic time. Note that achieving constant complexity for all the complexity measures is infeasible (under certain assumptions) due to the

$\Omega(\log n / \log \log n)$ lower bound [18] on the query complexity of memory checking (namely, the sum of read and write complexity). Finally, it would be interesting to modify our schemes to obtain exact (non-amortized) complexity bounds for updates using e.g., Overmar's global rebuilding technique [39].

References

1. Ateniese, G., Burns, R.C., Curtmola, R., Herring, J., Khan, O., Kissner, L., Peterson, Z.N.J., Song, D.: Remote data checking using provable data possession. ACM Trans. Inf. Syst. Secur. **14**(1), 12 (2011)
2. Baric, N., Pfitzmann, B.: Collision-free accumulators and fail-stop signature schemes without trees. In: Proceedings of Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT), vol. 1233 of LNCS, pp. 480–494 (1997)
3. Bellare, M., Rogaway, P.: Random oracles are practical: a paradigm for designing efficient protocols. In: Proceedings of Conference on Computer and Communications Security (CCS), ACM, pp. 62–73 (1993)
4. Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E., Virza, M.: SNARKs for C: verifying program executions succinctly and in zero knowledge. In: Proceedings of International Cryptology Conference (CRYPTO), vol. 8043 of LNCS, pp. 90–108 (2013)
5. Benaloh, J., de Mare, M.: One-way accumulators: a decentralized alternative to digital signatures. In: Proceedings of Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT), vol. 765 of LNCS, pp. 274–285 (1993)
6. Blum, M., Evans, W.S., Gemmell, P., Kannan, S., Naor, M.: Checking the correctness of memories. Algorithmica **12**(2/3), 225–244 (1994)
7. Boneh, D., Boyen, X.: Short signatures without random oracles and the SDH assumption in bilinear groups. J. Cryptol. **21**(2), 149–177 (2008)
8. Braun, B., Feldman, A.J., Ren, Z., Setty, S.T.V., Blumberg, A.J., Walfish, M.: Verifying computations with state. In: Proceedings of Symposium on Operating Systems Principles (SOSP), ACM, pp. 341–357 (2013)
9. Buldas, A., Laud, P., Lipmaa, H.: Accountable certificate management using undeniable attestations. In: Proceedings of Conference on Computer and Communications Security (CCS), ACM, pp. 9–18 (2000)
10. Camenisch, J., Kohlweiss, M., Soriente, C.: An accumulator based on bilinear maps and efficient revocation for anonymous credentials. In: Proceedings of Public Key Cryptography (PKC), vol. 5443 of LNCS, pp. 481–500 (2009)
11. Camenisch, J., Kohlweiss, M., Soriente, C.: An accumulator based on bilinear maps and efficient revocation for anonymous credentials. In: Proceedings of Public Key Cryptography (PKC), vol. 5443 of LNCS, pp. 481–500 (2009)
12. Camenisch, J., Lysyanskaya, A.: A signature scheme with efficient protocols. In: Proceedings of Security in Communication Networks (SCN), vol. 2576 of LNCS, pp. 268–289 (2002)
13. Canetti, R., Paneth, O., Papadopoulos, D., Triandopoulos, N.: Verifiable set operations over outsourced databases. In: Proceedings of Public Key Cryptography (PKC), vol. 8383 of LNCS, pp. 113–130 (2014)
14. Carter, I.L., Wegman, M.N.: Universal classes of hash functions. In: Proceedings of Symposium on Theory of Computing (STOC), ACM, pp. 106–112 (1977)
15. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. MIT Press, Cambridge (2009)
16. Damgård, I., Triandopoulos, N.: Supporting non-membership proofs with bilinear-map accumulators. http://eprint.iacr.org/, Cryptology ePrint Archive, Report 2008/538 (2008)
17. Dietzfelbinger, M., Karlin, A., Mehlhorn, K., Meyer auf der Heide, F., Rohnert, H., Tarjan, R.E.: Dynamic perfect hashing: upper and lower bounds. SIAM J. Comput. **23**, 738–761 (1994)
18. Dwork, C., Naor, M., Rothblum, G.N., Vaikuntanathan, V.: How efficient can memory checking be? In: Proceedings of Theoretical Cryptography Conference (TCC), vol. 5444 of LNCS, pp. 503–520 (2009)
19. Erway, C., Küpçü, A., Papamanthou, C., Tamassia, R.: Dynamic provable data possession. In: Proceedings of Conference on Computer and Communications Security (CCS), ACM, pp. 213–222 (2009)

20. Gennaro, R., Halevi, S., Rabin, T.: Secure hash-and-sign signatures without the random oracle. In: Proceedings of Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT), vol. 1592 of LNCS, pp. 123–139 (1999)
21. Goodrich, M.T., Papamanthou, C., Tamassia, R.: On the cost of persistence and authentication in skip lists. In: Proceedings of Workshop on Experimental Algorithms (WEA), vol. 4525 of LNCS, pp. 94–107 (2007)
22. Goodrich, M.T., Tamassia, R., Hasic, J.: An efficient dynamic and distributed cryptographic accumulator. In: Proceedings of Information Security Conference (ISC), vol. 2433 of LNCS, pp. 372–388 (2002)
23. Goodrich, M.T., Tamassia, R., Schwerin, A.: Implementation of an authenticated dictionary with skip lists and commutative hashing. In: Proceedings of DARPA Information Survivability Conference and Exposition II (DISCEX II), pp. 68–82 (2001)
24. Goodrich, M.T., Tamassia, R., Triandopoulos, N.: Super-efficient verification of dynamic outsourced databases. In: Proceedings of RSA Conference, Cryptographers' Track (CT-RSA), vol. 4964 of LNCS, pp. 407–424 (2008)
25. Goodrich, M.T., Tamassia, R., Triandopoulos, N.: Efficient authenticated data structures for graph connectivity and geometric search problems. Algorithmica **60**(3), 505–552 (2011)
26. Hutflesz, A., Six, H.-W., Widmayer, P.: Globally order preserving multidimensional linear hashing. In: Proceedings of International Conference on Data Engineering (ICDE), IEEE, pp. 572–579 (1988)
27. Kenyon, C.M., Vitter, J.S.: Maximum queue size and hashing with lazy deletion. Algorithmica **6**, 597–619 (1991)
28. Kosba, A.E., Papadopoulos, D., Papamanthou, C., Sayed, M.F., Shi, E., Triandopoulos, N.: TRUESET: nearly practical verifiable set computations. In: Usenix Security Symposium (USENIX SECURITY) (2014)
29. Li, J., Li, N., Xue, R.: Universal accumulators with efficient nonmembership proofs. In: Proceedings of Applied Cryptography and Network Security (ACNS), vol. 4521 of LNCS, pp. 253–269 (2007)
30. Linial, N., Sasson, O.: Non-expansive hashing. In: Proceedings of Symposium on Theory of Computing (STOC), ACM, pp. 509–517 (1996)
31. Lynn, B.: On the implementation of pairing-based cryptosystems. Ph.D. thesis, Stanford University, Stanford, California (2008)
32. Martel, C., Nuckolls, G., Devanbu, P., Gertz, M., Kwong, A., Stubblebine, S.G.: A general model for authenticated data structures. Algorithmica **39**(1), 21–41 (2004)
33. Merkle, R.C.: A certified digital signature. In: Proceedings of International Cryptology Conference (CRYPTO), vol. 435 of LNCS, pp. 218–238 (1989)
34. Mullin, J.K.: Spiral storage: efficient dynamic hashing with constant-performance. Comput. J. **28**, 330–334 (1985)
35. NTL: A library for doing number theory. http://www.shoup.net/ntl/
36. Naor, M., Nissim, K.: Certificate revocation and certificate update. In: Usenix Security Symposium (USENIX SECURITY), pp. 217–228 (1998)
37. Nguyen, L.: Accumulators from bilinear pairings and applications. In: Proceedings of RSA Conference, Cryptographers' Track (CT-RSA), vol. 3376 of LNCS, pp. 275–292 (2005)
38. Nuckolls, G.: Verified query results from hybrid authentication trees. In: Proceedings of Working Conference on Data and Applications Security (DBSEC), vol. 3654 of LNCS, pp. 84–98 (2005)
39. Overmars, M.H.: The Design of Dynamic Data Structures, vol. 156 of LNCS, Springer, London (1983)
40. Papamanthou, C.: Cryptography for efficiency: new directions in authenticated data structures. Ph.D. thesis, Brown University, Providence, Rhode Island (2011)
41. PBC: The pairing-based cryptography library. http://crypto.stanford.edu/pbc/
42. Papamanthou, C., Shi, E., Tamassia, R., Yi, K.: Streaming authenticated data structures. In: Proceedings of Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT), vol. 7881 of LNCS, pp. 353–370 (2013)
43. Papamanthou, C., Tamassia, R.: Time and space efficient algorithms for two-party authenticated data structures. In: Proceedings of International Conference on Information and Communications Security (ICICS), vol. 4861 of LNCS, pp. 1–15 (2007)
44. Papamanthou, C., Tamassia, R., Triandopoulos, N.: Authenticated hash tables. In: Proceedings of Conference on Computer and Communications Security (CCS), ACM, pp. 437–448 (2008)

45. Papamanthou, C., Tamassia, R., Triandopoulos, N.: Optimal verification of operations on dynamic sets. In: Proceedings of International Cryptology Conference (CRYPTO), vol. 6841 of LNCS, pp. 91–110 (2011)
46. Parno, B., Howell, J., Gentry, C., Raykova, M.: Pinocchio: Nearly practical verifiable computation. In: Proceedings of Symposium on Security and Privacy (SSP), IEEE, pp. 238–252 (2013)
47. Preparata, F.P., Sarwate, D.V.: Computational complexity of Fourier transforms over finite fields. Math. Comput. **31**(139), 740–751 (1977)
48. Sander, T.: Efficient accumulators without trapdoor (extended abstract). In: Proceedings of International Conference on Information and Communications Security (ICICS), vol. 1726 of LNCS, pp. 252–262 (1999)
49. Sander, T., Ta-Shma, A., Yung, M.: Blind, auditable membership proofs. In: Proceedings of Financial Cryptography (FC), vol. 1962 of LNCS, pp. 53–71 (2001)
50. Shoup, V.: A Computational Introduction to Number Theory and Algebra. Cambridge University Press, New York (2005)
51. Tamassia, R.: Authenticated data structures. In: Proceedings of European Symposium on Algorithms (ESA), vol. 2832 of LNCS, pp. 2–5 (2003)
52. Tamassia, R., Triandopoulos, N.: Computational bounds on hierarchical data processing with applications to information security. In: Proceedings of International Colloquium on Automata, Languages and Programming (ICALP), vol. 3580 of LNCS, pp. 153–165 (2005)
53. Tamassia, R., Triandopoulos, N.: Efficient content authentication in peer-to-peer networks. In: Proceedings of Applied Cryptography and Network Security (ACNS), vol. 4521 of LNCS, pp. 354–372 (2007)
54. Tamassia, R., Triandopoulos, N.: Certification and authentication of data structures. In: Proceedings of Alberto Mendelzon Workshop on Foundations of Data Management (2010)
55. Wang, P., Wang, H., Pieprzyk, J.: A new dynamic accumulator for batch updates. In: Proceedings of International Conference on Information and Communications Security (ICICS), vol. 4861 of LNCS, pp. 98–112 (2007)