Cryptographic Accumulators for Authenticated Hash Tables*

Charalampos Papamanthou[†] Roberto Tamassia[‡] Nikos Triandopoulos[§]

December 18, 2009

Abstract

Hash tables are fundamental data structures that optimally answer membership queries. Suppose a client stores n elements in a hash table that is outsourced at a remote server. Authenticating the hash table functionality, i.e., verifying the correctness of queries answered by the server and ensuring the integrity of the stored data, is crucial because the server, lying outside the administrative control of the client, can be malicious.

We design efficient and secure protocols for optimally authenticating (non-)membership queries on hash tables, using cryptographic accumulators as our basic security primitive and applying them in a novel hierarchical way over the stored data. We provide the first construction for authenticating a hash table with *constant query* cost and *sublinear update* cost, strictly improving upon previous methods.

Our first solution, based on the RSA accumulator, allows the server to provide a proof of integrity of the answer to a membership query in *constant* time and supports updates in $O(n^\epsilon \log n)$ time for any fixed constant $0 < \epsilon < 1$, yet keeping the communication and verification costs constant. It also lends itself to a scheme that achieves different trade-offs—namely, constant update time and $O(n^\epsilon)$ query time.

Our second solution uses an accumulator that is based on bilinear pairings to achieve $O(n^\epsilon)$ update time at the server while keeping all other complexities constant. Both schemes apply to two concrete data authentication models and an experimental evaluation shows good scalability.

*A preliminary version of this work was presented at the 15th ACM Conference on Computer and Communications Security (CCS), Alexandria, VA, 2008.

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

[‡]Department of Computer Science, Brown University. Email: `rt@cs.brown.edu`.

[§]Department of Computer Science, Boston University and Department of Computer Science, Brown University. Email: `nikos@cs.bu.edu`. Research performed primarily while the author was with the Department of Computer Science at Aarhus University, Denmark.

Contents

1 Introduction

Online storage of data (e.g., Amazon's S3 storage service) is becoming increasingly popular for both corporations and consumers. Clients create virtual drives consisting of online storage units that are operated by remote and geographically dispersed servers. In addition to being a convenient solution for data archiving or backups, remote storage allows for load-balanced distributed data management (e.g., database outsourcing). Indeed, large data volumes can become available to end applications through high-bandwidth connections to the data-hosting servers, which can rapidly answer queries arriving at high rates. Hence, data sources need to be online only when they wish to update their published data.

In several settings, the ability to check the integrity of remotely stored data is an important security property. Namely, we would like 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). Without some kind of verification mechanism, errors and attacks cannot be detected, no matter what trust relations the client and the server may a priori share. Thus, it is desirable that operations on remote data be *authenticated*. That is, answers to client queries should be 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 a fundamental data authentication problem, where we wish to authenticate membership queries over a dynamic set of n data elements stored in a hash table 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, and also theoretically interesting, to authenticate their functionality.

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 a (non-)membership query will be verified at the client by means of a corresponding proof that is provided by the server. Our main goal is to design protocols that implement this methodology both *securely*, against a computationally bounded server, and *efficiently*, with respect to any communication and computation overheads incurred due to the hash-table authentication.

In particular, we wish to minimize the amount of authentication information sent by the data source to the server to perform an update and also the size of the proof sent by the server to the client to demonstrate the correctness of the answer—ideally, we would like to keep both complexities constant. Analogously, since client-side applications may connect to the server from mobile devices with limited computing power and slow connectivity (e.g., cell-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's 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 [32]. Using cryptographic (collision resistant) hashing and Merkle's tree construction [30] to produce the set digest, (non-)membership queries in sets can be authenticated with *logarithmic costs* (e.g., [7, 21, 32, 37, 42]), which is optimal for any hash-based approach, as it has been shown in [41]. Breaking this logarithmic barrier, therefore, requires employing an alternative cryptographic primitive. One-way accumulators and their dynamic extensions [4, 6, 10, 11, 34] 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 accumulator schemes (e.g., [11, 34]) in a hierarchical way over the set and the underlying hash table, to securely authenticate *both membership and non-membership queries* and fully achieve our complexity goals. That is, in our authentication schemes communication and verification costs are constant, the query cost is constant and the update cost is *sublinear*, realizing the first authenticated hash table with this performance. Our scheme strictly improves upon previous schemes based on accumulators. We base the security of our protocols on two widely accepted assumptions, the strong RSA assumption [4] and the q-strong Diffie-Hellman assumption [8].

Moreover, aiming at authentication solutions that cover a wide application area, we instantiate our core authentication scheme—without sacrificing its performance guarantees—to two concrete, widely-used data authentication models, which we call the *three-party* and *two-party* authentication models, both closely related to our remote-storage setting.

The three-party model has been used to define the concept of *authenticated data structures* [32, 40] and involves a trusted *source* that *replicates* a data set to one or more untrusted *servers*, as well as one or more *clients* that access this data set by querying one of the servers. Along with the answer to a query, a server provides the client with a proof that when combined with the (authentic) data-set digest can verify the correctness of the answer. This digest is, periodically or after any update, produced, time-stamped (to defeat replay attacks) and signed by the source (a PKI is assumed), and is forwarded to the server(s) to be included in any answer sent to a client. This model offers load-balancing and computation outsourcing for data publication applications, therefore the source typically keeps the same data set and authentication structure as the server; this can potentially allow the source to facilitate the server's task by communicating appropriate *update authentication information* after set updates.

The two-party model, instead, involves a *client* that, being simultaneously the data source and the data consumer, fully *outsources* the data set to an untrusted *server*, keeping locally only the data-set digest (of constant size), subject to which any operation (update or query, executed by the server) on the remotely stored data is verified, again using a corresponding proof provided by the server. This model offers both storage and computation outsourcing, but only the data owner has access to the stored set. Here, the main challenge is to maintain at all times a state (digest) that is consistent with the history of updates, typically requiring more involved authentication structures. This model is related to the memory-checking model [7, 33]. For a detailed description of the models we refer to [19, 21, 37]. Note that in both models (three-party and two-party) we assume the existence of a public key pk (see Section 2.4) that contains public information available to both the client and the untrusted server (and also to the source for the case of the three-party model). Moreover, in the three-party model, where PKI is used, pk also contains the public key of the signature scheme used by the source, which is used by the client for verification. Finally we note that this public key does not come along with a respective "private" key, as it happens with signature schemes. It just refers to information that is publicly available (to which the adversary has access), as opposed to information that is crucial for the security of the system and is therefore kept secret from the adversary, such as the *trapdoor* information.

Finally, to meet the needs of different data-access patterns, in our solution based on RSA accumulators, we extend both our three-party and our two-party authentication schemes to achieve

Table 1: Comparison of existing schemes for membership authentication in a set of size n w.r.t. used techniques and various complexity measures. Here, $0 < \epsilon < 1$ is a fixed constant, **NA** stands for "not applicable", DH for "Diffie-Hellman", exp for "exponentiation" and BM for "bilinear map". All complexity measures refer to n (not the security parameter) and are asymptotic expected values. Allowing sublinear updates and extensions for different update/query trade-offs, our schemes perform better than existing ones. Update costs in our schemes are expected amortized values. In all schemes, the server uses $O(n)$ space and the client uses $O(1)$ space. In the 3-party model an additional signature cost is incurred (signed digest).

reference	model	assumption	proof size	update info.	query time	update time	verify time	crypto oper.
[7, 21, 29, 32, 37]	both	collision resistance	$\log n$	1	$\log n$	$\log n$	$\log n$	hashing
[3]	2-party	strong RSA	1	**NA**	1	**NA**	1	exp
[11, 39]	both	strong RSA	1	1	1	$n \log n$	1	exp
[34]	both	strong DH	1	1	1	n	1	exp, BM
[20]	3-party	strong RSA	1	n^ϵ	n^ϵ	n^ϵ	1	exp
main scheme (a)	both	strong RSA	1	1	1	$n^\epsilon \log n$	1	exp
main scheme (b)	both	strong DH	1	1	1	n^ϵ	1	exp, BM
extension	both	strong RSA	1	1	n^ϵ	1	1	exp

a reverse performance, i.e., sublinear query cost, but constant update cost. Also, 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 performance, scalability and a high degree of practicality.

1.1 Our contributions

1. We propose a new cryptographic construction 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 [4] and the bilinear-map accumulator [34] and formally prove the security of our new schemes based only on widely accepted and used cryptographic assumptions, namely the strong RSA assumption [4] and the q-strong Diffie-Hellman assumption [8];

2. We introduce *authenticated hash tables* and we show how to exploit the efficiency of hash tables to develop an authenticated data structure supporting both membership and non-membership queries on sets drawn from general (ordered) universes. We give solutions for authenticating a hash table both in the *two-party* and *three-party* authentication models;

3. We improve the complexity bounds of previous work while still being provably secure. Let $0 < \epsilon < 1$ be a fixed constant. For the RSA accumulator solution, we reduce the query time and the size of the update authentication information from $O(n^\epsilon)$, as it appears in [20], (previously, the best known upper bound for authenticating set-membership queries using RSA accumulators) to $O(1)$, keeping the update time sublinear (i.e., $O(n^\epsilon \log n)$). This answers an open problem posed in [32]. Also, we extend our scheme to get a different trade-off between query and update costs, namely constant update time with $O(n^\epsilon)$ query time (see Table 1). For the bilinear-map accumulator solution we improve the update time from $O(n)$, as it appears in [34], to $O(n^\epsilon)$, while keeping all the other complexity measures constant;

4. We give a practical evaluation of our schemes using state-of-the-art software [1, 2] for primitive operations (namely, modular exponentiations, multiplications, inverse computations and bilinear maps);

5. We propose studying *lower bounds* for authenticated set-membership queries using cryptographic accumulators.

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 authenticated data structures based on cryptographic hashing have been developed for membership queries (e.g., [7, 21, 29, 32, 37]), both in the two-party and three-party authentication models. These data structures achieve $O(\log n)$ proof size, query time, update time and verification time. As shown in [41], 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, 18, 23, 42]).

Solutions for authenticated membership queries in various settings using another cryptographic primitive, namely *one-way accumulators*, were introduced by Benaloh and de Mare [6]. Based on the RSA exponentiation function, this scheme implements a secure one-way function that satisfies *quasi-commutativity*, 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 [4], where except for one-wayness, collision resistance is achieved, and also in [17, 39]. Dynamic accumulators (along with protocols for zero-knowledge proofs) were introduced in [11], where, using the trapdoor information (these protocols are secure, assuming an honest prover), the time to update the accumulated value or a witness is independent on the number of the accumulated elements.

A first step towards a different direction, where we assume that we cannot trust the prover and therefore the trapdoor information (e.g., the group order $\phi(N)$) is kept secret, but where the resulting schemes are applicable only to the three-party model, was made in [20]; in this work, general $O(n^\epsilon)$ 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 [20] is presented in [22], and a similar hybrid authentication scheme appears in [35].

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, 34]. However, updates in [34] 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 important constraints on the range of the accumulated values are required. Efficient dynamic accumulators for non-membership proofs are presented in [26]. Accumulators for batch updates are presented in [44] and accumulator-like expressions to authenticate static sets for *provable data possession* are presented in [3, 16]. The work in [38] studies efficient algorithms for accumulators with unknown trapdoor information. Finally in [15] and simultaneously with our work, logarithmic lower bounds as well as constructions achieving query-update cost trade-offs that are similar to our work, have been studied in the memory-checking model.

1.3 Organization of the paper

In Section 2, we introduce some necessary cryptographic and algorithmic ideas needed for the development of our construction. We also give the security definition of our schemes. In Section 3, we develop our first solution based on the RSA accumulator and present the main proof of security. In Section 4, we present our second solution that is based on bilinear maps. In Section 5, we provide an evaluation and analysis of our authentication methods showing their practicality, and finally in Section 6, we conclude with future work and interesting open problems.

2 Preliminaries

In this section we describe some algorithmic and cryptographic primitives and other useful concepts that are used in our approach.

2.1 Hash tables

The main functionality of the hash table data structure is to support look-ups of elements that belong to a general set (i.e., not necessarily ordered). Different ways of implementing hash tables have been extensively studied (e.g., [14, 24, 25, 27, 31]). Suppose we wish to store n elements from a universe $\mathcal{U}$ in a data structure so that we can have expected constant look-up time. For totally ordered universes and by searching based on comparisons, it is well known that we need $\Omega(\log n)$ time. Hash tables, however, achieve better efficiency as follows.

- Set up an one-dimensional table $T[1 \ldots m]$ where $m = O(n)$;
- Pick a function $h : \mathcal{U} \rightarrow \{1, \ldots, m\}$, called *hash function*, randomly selected from a family of two-universal hash functions (also used in Lemma 1). Thus, for any two elements $e_1, e_2 \in \mathcal{U}$, we have $\Pr[h(e_1) = h(e_2)] \leq \frac{1}{m}$;
- Store element e in slot $T[h(e)]$ 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 takes expected constant time.

But the above property of hash tables comes at some cost. The expected constant-time 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 cannot assume anymore that is expected to be constant. 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 may 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 takes $O(n)$ rather than $O(1)$ time. Thus, hash tables for dynamic sets typically have expected $O(1)$ query time $O(1)$ expected *amortized* time. Methods that vary the size of the hash table for the sake of maintaining $O(1)$ query time, fall into the general category of *dynamic hashing*.

The above discussion is summarized in the following theorem.

Theorem 1 (Dynamic hashing [13]) *For a set of size n, dynamic hashing can be implemented to use $O(n)$ space and have $O(1)$ expected query time for (non-)membership queries and $O(1)$ expected amortized cost for elements insertions or deletions.*

Before we define some cryptographic primitives, it is useful to give the definition of a *negligible* function, where k denotes the security parameter.

Definition 1 (Negligible function) *We say that a real-valued function $\nu(k)$ over natural numbers is negligible if for any positive polynomial p, there exists integer m such that $\forall n > m$, $|\nu(n)| < \frac{1}{p(n)}$. We refer to a negligible function $\nu(k)$ also by saying that $\nu(k)$ is $\mathsf{neg}(k)$.*

2.2 The RSA accumulator

We now give an overview of the RSA accumulator, which will be used for the construction of our first solution.

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 in [4], prime representatives provide a solution whenever it is necessary to map general elements to prime numbers. In particular, one can map a k-bit element e_i to a $3k$-bit prime x_i using *two-universal hash functions* [12].

We say that a family of functions $H = \{h : A \rightarrow B\}$ is *two-universal* if, for all $w_1 \neq w_2$ and for a randomly chosen function h from H, we have

$$\Pr[h(w_1) = h(w_2)] \leq \frac{1}{|B|}\,.$$

In our context, set A is the set of of $3k$-bit boolean vectors, B is the set of k-bit boolean vectors, and we use the two universal function

$$h(x) = Fx\,,$$

where F is a $k \times 3k$ boolean matrix. Since the linear system $h(x) = Fx$ has more than one solutions, one k-bit element is mapped 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 [17, 20]) *Let H be a two-universal family of functions mapping $\{0,1\}^{3k}$ to $\{0,1\}^k$ and let $h \in H$. For any element $e_i \in \{0,1\}^k$, we can compute a prime $x_i \in \{0,1\}^{3k}$ such that $h(x_i) = e_i$ by sampling $O(k^2)$ times with high probability from the set of inverses $h^{-1}(e_i)$.*

Lemma 1 implies that we can compute prime representatives in expected constant time, since the dimension of our problem is the number n of the elements in the hash table. Also, solving the $k \times 3k$ linear system in order to compute the set of inverses can be performed in polynomial time 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 above.

The RSA accumulator. We now give an overview of the *RSA accumulator* [4, 6, 11], which provides an efficient technique to produce a short (computational) proof that a certain element is a member of a set. The RSA accumulator works as follows. Suppose we have the set of k-bit elements $\mathcal{E} = \{e_1, e_2, \ldots, e_n\}$. Let N be a k'-bit RSA modulus ($k' > 3k$), namely $N = pq$, where p, q are strong primes [11]. We can represent $\mathcal{E}$ compactly and securely with an *accumulation* value, which is a k'-bit integer, as follows

$$f(\mathcal{E}) = g^{r(e_1)r(e_2)\ldots r(e_n)} \mod N\,,$$

where $g \in \mathbb{QR}_N$ and $r(e_i)$ is a $3k$-bit prime representative. Note that the RSA modulus N, the exponentiation base g and the two-universal hash functions comprise the public key pk, i.e., information that is available to the adversary. Subject to the accumulation $f(\mathcal{E})$, every element e_i in set $\mathcal{E}$ has a *membership witness* (or proof), namely the value

$$A_{e_i} = g^{\prod_{e_j \in \mathcal{E}: e_j \neq e_i} r(e_j)} \mod N\,.$$

Given the accumulation value $f(\mathcal{E})$ and a witness A_{e_i}, membership of e_i in $\mathcal{E}$ can be verified by computing $A_{e_i}^{r(e_i)} \mod N$ and checking that this equals $f(\mathcal{E})$.

The above representation has also the property that any computationally bounded adversary $\mathcal{A}$ who does not know $\phi(N)$ cannot find another set of elements $\mathcal{E}' \neq \mathcal{E}$ such that $f(\mathcal{E}') = f(\mathcal{E})$, unless $\mathcal{A}$ breaks the *strong RSA assumption* [4], which is stated as follows:

Definition 2 (Strong RSA assumption) *Given an RSA modulus N and a random element $x \in \mathbb{Z}_N$, it is hard (i.e., it can be done with probability that is $\mathsf{neg}(k)$, which is negligible in the security parameter k) for a computationally bounded adversary $\mathcal{A}$ to find $y > 1$ and a such that $a^y = x \mod N$.*

The security of our RSA-accumulator solution is based on the following result. To assist the reader, we also recall the proof of that Lemma, originally given in [4].

Lemma 2 (Security of the RSA accumulator [4]) *Let k be the security parameter, h be a two-universal hash function that maps $3w$-bit integers to w-bit integers and N be a $(3w+1)$-bit RSA modulus. Given a set of elements $\mathcal{E}$, the probability that a computationally bounded adversary $\mathcal{A}$, knowing only N and g, can find a set $\mathcal{E}' \neq \mathcal{E}$ with the same accumulation as $\mathcal{E}$ (i.e, $f(\mathcal{E}') = f(\mathcal{E})$) is $\mathsf{neg}(k)$.*

Proof: Suppose $\mathcal{A}$ finds such a set $\mathcal{E}'$. That means that $\mathcal{A}$ finds another set $\{e'_1, e'_2, \ldots, e'_{n'}\} \neq \{e_1, e_2, \ldots, e_n\}$ such that

$$g^{r(e_1)r(e_2)\ldots r(e_n)} = g^{r(e'_1)r(e'_2)\ldots r(e'_{n'})} \mod N\,.$$

By the way we construct the prime representatives, it is not possible that a prime representative can be associated with two different elements. Therefore, it also holds $\{r(e_1), r(e_2), \ldots, r(e_n)\} \neq \{r(e'_1), r(e'_2), \ldots, r(e'_{n'})\}$ which implies that the adversary can find a value A and an index j such that

$$A^{r(e_j)} = g^{r(e'_1)r(e'_2)\ldots r(e'_{n'})} \mod N\,,$$

where

$$A = g^{\prod_{i \neq j} r(e_i)} \mod N\,.$$

Let now $e = r(e_j)$ and $r = r(e'_1)r(e'_2)\ldots r(e'_{n'})$. The adversary can now compute the e-th root of g as follows: $\mathcal{A}$ computes $a, b \in Z$ such that $ar + br(e_j) = 1$ by using the extended Euclidean algorithm, since $r(e_j)$ is a prime. Let now $y = A^a g^b$. It is

$$y^e = A^{ar(e_j)} g^{br(e_j)} = g^{ar + br(e_j)} = g \mod N$$

and, therefore, $\mathcal{A}$ can break the strong RSA assumption which occurs with probability $\nu(k)$, where $\nu(k)$ is $\mathsf{neg}(k)$. $\Box$

Using the same proof arguments, the following corollary holds.

Corollary 1 *Let k be the security parameter, h be a two-universal hash function mapping $3w$-bit integers to w-bit integers and N be a $(3w+1)$-bit RSA modulus. Given a set of elements $\mathcal{E}$ and h, the probability that a computationally bounded adversary $\mathcal{A}$, knowing only N and g, can find A and $x \notin \mathcal{E}$ such that $A^{r(x)} = f(\mathcal{E})$ is $\mathsf{neg}(k)$.*

2.3 The bilinear-map accumulator

We next give an overview of the bilinear-map accumulator which will be used for the construction of our second solution.

Bilinear pairings. Before presenting the bilinear-map accumulator we describe some basic terminology and definitions about *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 $\psi : \mathbb{G}_2 \rightarrow \mathbb{G}_1$ such that $\psi(g_2) = g_1$. Here, $\mathbb{G}_M$ is a cyclic multiplicative group with the same order p and $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_M$ is 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 1^k and outputs a uniformly random tuple $\mathsf{t} = (p, \mathbb{G}, \mathbb{G}_M, e, g)$ of bilinear pairings parameters. Later we are going to see that the security of the bilinear-map accumulator is based on an assumption that is related to the bilinear pairings.

The bilinear-map accumulator. Similarly with the RSA accumulator, the bilinear-map accumulator [34] is an efficient way to provide short proofs of membership for elements that belong to a set. The bilinear-map accumulator works as follows. It accumulates elements in $\mathbb{Z}_p^*$ (where p is a prime) and the accumulated value is an element in $\mathbb{G}$. Given a set of n elements $\mathcal{E} = \{e_1, e_2, \ldots, e_n\}$ the accumulation value $f'(\mathcal{E})$ is defined as

$$f'(\mathcal{E}) = g^{(e_1+s)(e_2+s)\ldots(e_n+s)},$$

where g is a generator of group $\mathbb{G}$ of prime order p and $s \in \mathbb{Z}_p^*$ is a randomly chosen value that constitutes the trapdoor in the scheme (in the same way that $\phi(N)$ was the trapdoor in the RSA accumulator). The proof of membership for an element e_i that belongs to set $\mathcal{E}$ will be the witness

$$A_{e_i} = g^{\prod_{e_j \in \mathcal{E} : e_j \neq e_i} (e_j + s)}.$$

Accordingly, a verifier can test set membership for e_i by computing $A_{e_i}^{(e_i+s)}$ and checking that this equals $f'(\mathcal{E})$.

Here we have to make an important observation: The group $\mathbb{G}$ is generic. That means that its elements are not simple integers and doing operations between two elements of $\mathbb{G}$ can be complicated. We are going to refer to the implementation of $\mathbb{G}$ later in the paper. Also the operations in the exponent of elements of $\mathbb{G}$ are performed modulo p, since this is the order of the group $\mathbb{G}$. The security of the bilinear pairings accumulator is based on the q-strong Diffie-Hellman assumption which can be stated as follows:

Definition 3 (q-strong Diffie-Hellman assumption) *Given a uniformly randomly generated tuple $\mathsf{t} = (p, \mathbb{G}, \mathbb{G}_M, e, g)$ of bilinear pairings parameters and the elements of $\mathbb{G}_M$ $g, g^s, g^{s^2}, \ldots, g^{s^q}$ for some s chosen at random from $\mathbb{Z}_p^*$, it is hard (i.e., it can be done with probability that is $\mathsf{neg}(k)$, which is negligible in the security parameter k) for a computationally bounded adversary $\mathcal{A}$ to find $c \in \mathbb{Z}_p$ and output $(c, g^{1/(s+c)})$.*

We now recall the main security claim for the bilinear pairings accumulator, i.e., that it provides collision resistance:

Lemma 3 (Security of the bilinear-map accumulator [34]) *Let k be the security parameter and $\mathsf{t} = (p, \mathbb{G}, \mathbb{G}_M, e, g)$ be a uniformly randomly generated tuple of bilinear pairings parameters. Given a set of elements $\mathcal{E}$, the probability that a computationally bounded adversary $\mathcal{A}$, knowing only $g, g^s, g^{s^2}, \ldots, g^{s^q}$ $(q \geq |\mathcal{E}|)$ for some s chosen at random from $\mathbb{Z}_p^*$ and t, can find a set $\mathcal{E}' \neq \mathcal{E}$ $(q \geq |\mathcal{E}'|)$ such that $f'(\mathcal{E}') = f'(\mathcal{E})$ is $\mathsf{neg}(k)$.*

Proof: Suppose $\mathcal{A}$ finds such a set $\mathcal{E}'$. That means that $\mathcal{A}$ finds another set $\{e'_1, e'_2, \ldots, e'_{n'}\} \neq \{e_1, e_2, \ldots, e_n\}$ such that

$$g^{(e_1+s)(e_2+s)\ldots(e_n+s)} = g^{(e'_1+s)(e'_2+s)\ldots(e'_{n'}+s)}$$

which implies that

$$A^{(e'_j+s)} = g^{(e_1+s)(e_2+s)\ldots(e_n+s)},$$

where

$$A = g^{\prod_{i \neq j}(e'_i+s)}$$

for some e'_j that does not belong to the original set. Note now that the quantity

$$\Pi_n = (e_1 + s)(e_2 + s) \ldots (e_n + s)$$

can be viewed as a polynomial in s of degree n. Since $e'_j \notin \mathcal{E}$, we have that $(e'_j + s)$ does not divide Π_n and therefore $\mathcal{A}$ can find c and P such that $\Pi_n = c + P(e'_j + s)$. Therefore

$$A = g^P g^{c/(e'_j+s)}$$

which gives $g^{1/(x+s)} = [A[g^P]^{-1}]^{c^{-1}}$ and the adversary can break the q-strong Diffie-Hellmann assumption which occurs with probability that is $\mathsf{neg}(k)$. □

Here we note that the public key pk in the case of the bilinear-map accumulator is comprised by the exponentiation base g and the elements $g, g^s, g^{s^2}, \ldots, g^{s^q}$ (see Lemma 3). As before, we can now state the following corollary:

Corollary 2 *Let k be the security parameter and $\mathsf{t} = (p, \mathbb{G}, \mathbb{G}_M, e, g)$ be a uniformly randomly generated tuple of bilinear pairings parameters. Given a set of elements $\mathcal{E}$, the probability that a computationally bounded adversary $\mathcal{A}$, knowing only $g, g^s, g^{s^2}, \ldots, g^{s^q}$ $(q \geq |\mathcal{E}|)$ for some s chosen at random from $\mathbb{Z}_p^*$ and t, can find can find A and $x \notin \mathcal{E}$ such that $A^{x+s} = f'(\mathcal{E})$ is $\mathsf{neg}(k)$.*

2.4 Set-membership authentication schemes

We now continue with the definition of set-membership authentication schemes and their main security property, which captures the security requirements of authenticated hash tables. Suppose S is a set for which we wish to authenticate membership of elements (i.e., queries of type "does x belong to S?") and let pk be the public key. A *set-membership authentication scheme* consists of three algorithms update, query, verify, and associated data structures, for respectively updating S, querying S to produce a corresponding set-membership proof, and verifying an answer to a query. In general, these algorithms are:

1. $\{S', d'\} \leftarrow \mathsf{update}(\mathsf{upd}, S)$, where d' is the new digest of S after the update (we recall that the digest of S is a short description of S, e.g., the root hash of a Merkle tree), upd is an update supported by the data structure and S, S' are the old and new (updated) sets respectively;
2. $\Pi(x) \leftarrow \mathsf{query}(x, S)$, where $\Pi(x)$ is the proof returned to a query for membership of element x in S;
3. $\{\mathsf{accept}, \mathsf{reject}\} \leftarrow \mathsf{verify}(x, \Pi(x), d)$, where d is the current digest of S and $\Pi(x)$ is the proof, both used for verifying membership of x in S.

We require that a set-membership authentication scheme $(\mathsf{update}, \mathsf{query}, \mathsf{verify})$ is *correct*, i.e., for any $x \in S$ it holds that $\mathsf{accept} \leftarrow \mathsf{verify}(x, \mathsf{query}(x, S), d)$.

With respect to the security of the scheme, we assume that the adversary is given oracle access to all these algorithms for updating and querying S and also for verifying answers. The formal security definition, which is an adaptation to our setting of the security definition for dynamic accumulators presented in [11], for a membership authentication scheme of a set S is as follows:

Definition 4 (Security) *Suppose k is the security parameter and $\mathcal{A}$ is a computationally bounded adversary that is given the public key* pk. *Set S is initially empty and $S = S_0$. First, in the* update stage*, the adversary $\mathcal{A}$ chooses and issues a series of $t+1$ updates*

$$\mathsf{upd}_i \in \{\mathsf{ins}(x_i), \mathsf{del}(x_i)\} \quad \text{for } i = 0, \ldots, t\,,$$

which yields a series of sets derived from S and corresponding digests

$$\{S_{i+1}, d_{i+1}\} \leftarrow \mathsf{update}(\mathsf{upd}_i, S_i)\,,$$

where d_0 is the digest of an empty set and t is polynomially dependent on the security parameter k. After the update stage, $\mathcal{A}$ possesses the new set S_{t+1} and the corresponding digest d_{t+1}. Next, adversary $\mathcal{A}$ enters the attack stage where he chooses an element $y \notin S_{t+1}$ and computes a proof $\Pi(y)$ for y. We say that the set-membership authentication scheme $(\mathsf{update}, \mathsf{query}, \mathsf{verify})$ *is* secure *if the probability that* $\mathsf{accept} \leftarrow \mathsf{verify}(y, \Pi(y), d_{t+1})$ *is* $\mathsf{neg}(k)$.

Remarks on Definition 4. We make the following observations about the security requirements given in Definition 4.

1. The security definition captures the setting where an adversary tries to forge proofs for elements that do not belong to the existing set, which is the main attack we wish to guard against. Additionally, as in [11], we allow the adversary to choose his own updates and choose his own elements to forge, which provides a stronger notion of security, as the authentication scheme defeats attacks independently of the history of updates in the set;

2. This security definition is applicable to both our concrete authentication models of interest (two-party and three-party) in the sense that in both models security is defined subject to the correct digest defined over the current set, that is, set-membership authentication is guaranteed assuming that the verification algorithm takes as input this (correct and fresh) digest. As we will see, this assumption will however be achieved with different methods: In the three-party model a time-stamped signature on the digest (produced by the source) is used by the client to verify the validity of the digest, whereas in the two-party model the client engages in a protocol with the server that allows him to locally update the digest correctly with non-negligible probability;

3. Modeling the information given to the adversary through oracle access is not restrictive; in our concrete schemes and at all times, the information that the server stores and maintains is completely characterized by invocation of algorithms update, query, verify (or equivalently by polynomial-time functions on the current set and the public key pk). Finally, we note that in essence the above security definition captures the case where the adversary performs an attack in some older state S'_t, $t' < t$, of the data set than the current one S_t, that is, when the attack phase includes some set updates. This is the case exactly because all the new information that the adversary gets between t' and t is a polynomial-time function of pk.

3 Scheme based on the RSA accumulator

In this section, we describe how we can use the RSA accumulator in order to implement authenticated hash tables, that is, set-membership authentication schemes that authenticate the functionality of hash tables.

We make our solution applicable to two concrete data authentication models, the two-party model and the three-party model that were briefly described in Section 1. We recall that the two-party model refers to the data outsourcing scenario, where a client relocates all of its data to an untrusted server, the client being the party that issues both queries and updates to the outsourced data. The three-party model refers to a slightly different scenario, where a trusted source makes its data available to an untrusted server that answers queries submitted by a client, the source being the party that issues updates and the client being the party that issues queries. In both settings, the goal is to design secure and efficient protocols for verifying that the untrusted server correctly manages the outsourced data.

In Section 3.1 we describe *accumulation trees*, the main data structure behind our solution, which is also used in Section 4. In Section 3.2, we describe the setup that is needed for our solution in both models. In Section 3.3 we describe the main authenticated data structure used in our authentication schemes. Focusing on the three-party model, in Section 3.4 we show how our construction applies to the special case of static data, providing some intuition for the general dynamic solution (authenticated hash tables) that follows in Section 3.5. In Section 3.6 we apply our results to the two-party model and in Section 3.7 we show how to achieve a more practical scheme by using random oracles.

3.1 The accumulation tree

In this section we describe the main construction for authenticating set-membership in a hash table. Initially we present a general scheme which can be extended in order to achieve better complexity bounds for the hash table.

Let $S = \{e_1, e_2, \ldots, e_n\}$ be the set of elements we would like to authenticate. Given a constant $\epsilon < 1$ such that $0 < \epsilon < 1$, the *accumulation tree* of S, denoted $T(\epsilon)$, is a rooted tree with n leaves defined as follows:

1. The leaves of $T(\epsilon)$ store the elements $e_1, e_2, \ldots, e_n$;
2. $T(\epsilon)$ consists of exactly $l = \lceil \frac{1}{\epsilon} \rceil$ levels;
3. All the leaves are at the same level;
4. Every node of $T(\epsilon)$ has $O(n^\epsilon)$ children;
5. Level i in the tree contains $O(n^{1-i\epsilon})$ nodes, where the leaves are at level 0 and the root is at level l.

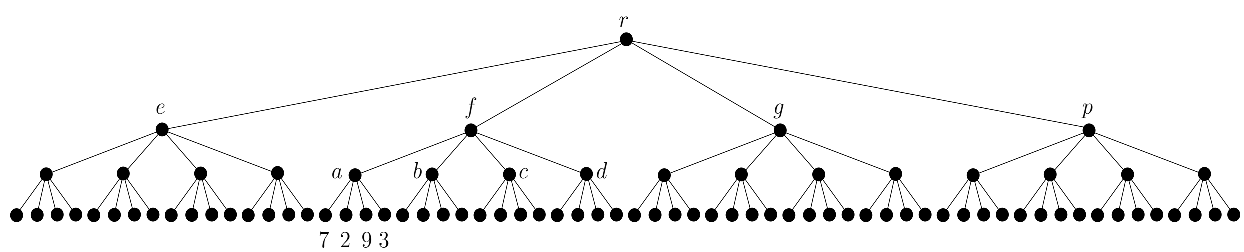

Figure 1: The accumulation tree of a set of 64 elements for $\epsilon = \frac{1}{3}$: every internal node has $4 = 64^{\frac{1}{\epsilon}}$ children, there are $3 = \frac{1}{\epsilon}$ levels in total, and there are $64^{1-i/3}$ nodes at level $i = 0, 1, 2, 3$.

We note that the levels of the 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 Figure 1, resembles that of normal "flat" search trees, in particular, the structure of a B-tree. 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 simple to construct the accumulation tree when n^ϵ is an integer (see Figure 1). Else, we define the accumulation tree to be the unique 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 S that matches the *optimal querying performance* of hash tables. The idea is as follows: we wish to hierarchically employ the RSA accumulator over the subsets (of accumulation values) defined by each internal node in the accumulation tree, so that 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).

3.2 System setup

Towards employing the RSA accumulator hierarchically over the accumulation tree, we now describe the initial setup of our authentication schemes.

Let k be the security parameter. In the two-party model, the client initially picks constant $0 < \epsilon < 1$ and $l = \lceil 1/\epsilon \rceil$ RSA moduli $N_i = p_i q_i$ ($i = 1, \dots, l$), where p_i, q_i are strong primes [11]. The length of the RSA modului is defined by the recursive relation

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

where $|N_1| = 3k + 1$ and $i = 1, \dots, l-1$. Note that since l is constant all the RSA moduli have asymptotically the same dependence on the security parameter k. For $i = 1, \dots, l$, the client reveals N_i to the untrusted server but keeps $\phi(N_i) = (p_i - 1)(q_i - 1)$ secret. The client also picks l public bases $g_i \in \mathbb{QR}_{N_i}$ to be used for exponentiation. Finally, given l families of two-universal hash functions $H_1, H_2, \dots, H_l$, the client randomly picks one function $h_i \in H_i$ and reveals h_i to the server (to be used for computing multiple prime representatives). The function h_i is such that it maps $(|N_i|-1)$-bit primes to $((|N_i| - 1)/3)$-bit integers. Also, the choice of the domains and ranges of functions h_i and of the lengths of moduli N_i is due to the requirement that prime representatives should be smaller numbers than the respective moduli (see [39]). As we will see in Section 5, using ideas from [4] it is possible to avoid the increasing size of the RSA moduli and instead use only one size for all N_i's. By doing so, however, we are forced to prove security in the random oracle model (using cryptographic hash functions), which is fine for practical applications. It is crucial that $\phi(N_i)$ is not revealed to the untrusted server, since otherwise the security of the whole system collapses, as the server would be able to compute inverses and, as it will become clear, forge proofs. Note that since $1/\epsilon$ is constant, the client needs constant space.

In the three-party model, the setup is exactly as above, but now all the public information (RSA moduli, two-universal hash functions) and the secret information (factorization ϕ) is generated by the source; also, the client now gets only the public information.

3.3 Main authenticated data structure

We next present the main component of our authentication schemes, an authenticated data structure that is based on the accumulation tree. This structure is stored at all times at the server; in the three-party model the structure is also stored by the source. Let S be the set we would like to authenticate. Our authenticated data structure is defined with respect to the accumulation tree as follows. By hierarchically employing the RSA accumulator over set S, we 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, as a function of the accumulation value of its children (in a similar way as in a Merkle tree). In particular, let $h_1, h_2, \dots, h_l$, $l = \lceil \frac{1}{\epsilon} \rceil$, be two-universal hash functions, where h_i maps w_i-bit elements to $3w_i$-bit primes, $i = 1, \dots, l$. For every leaf node v in tree $T(\epsilon)$ that lies at level 0 and stores element e, we set $\chi(v) = e$, while for every non-leaf node v in $T(\epsilon)$ that lies in level $1 \le i \le l$, we set:

$$\chi(v) = g_i^{\prod_{u \in N(v)} r_i(\chi(u))} \mod N_i\,, \tag{1}$$

where $r_i(\chi(u))$ is a prime representative of $\chi(u)$ computed using function h_i, $N(v)$ is the set of children of node v and $g_i \in \mathbb{QR}_{N_i}$. Additionally, we store at each node v at level i of the accumulation tree the prime representative $r_{i+1}(\chi(v))$ of its accumulation value $\chi(v)$ (except for the root of the tree that lies at level l where we do not need a prime representative since its $\chi()$ value is the digest of the set).

We call the above authenticated data structure an *augmented accumulation tree* built on top of S, but often, for simplicity and when it is clear from the context, we refer to it as the accumulation tree $T(\epsilon)$. Given these accumulation values, the augmented accumulation tree can be seen as a systematic way to define a digest over an underlying set.

Definition 5 *Given a set $S = \{e_1, e_2, \ldots, e_n\}$ of n elements, l RSA moduli $N_1, N_2, \ldots, N_l$, l two-universal functions $h_1, h_2, \ldots, h_l$ and the accumulation tree $T(\epsilon)$ built on top of S, we define the* RSA digest *of node v of the accumulation tree to be equal to $\chi(v)$, also denoted with $\chi(S_v)$, where $S_v \subseteq S$ is the set of elements associated with the subtree rooted at v. The* RSA digest *of the set S is equal to $\chi(r)$, where r is the root of tree $T(\epsilon)$.*

Note that, given a set S, the RSA digest $\chi(S)$ depends on the elements in S, the used RSA moduli and two-universal functions, but not on the structure of the tree, because the structure of $T(\epsilon)$, for a given ϵ, is deterministic and the RSA exponentiation function is quasi-commutative. We next show the main security property of our new authenticated data structure.

Theorem 2 (Collision resistance) *Let k be the security parameter and $U = \{u_1, u_2, \ldots, u_n\}$ a set of n elements. Given the associated accumulation tree $T(\epsilon)$ built on top of U, under the strong RSA assumption, the probability that a computationally bounded adversary $\mathcal{A}$, knowing only the RSA moduli N_i and g_i, $1 \leq i \leq l$ ($l = \lceil 1/\epsilon \rceil$), can find another set $V \neq U$ such that $\chi(V) = \chi(U)$ is $\mathsf{neg}(k)$.*

Proof: We are going to prove the following claim by induction: Given an accumulation tree of l levels, it is difficult for a computationally bounded adversary to find two different sets U, V such that $\chi_l(U) = \chi_l(V)$, where $\chi_l(S)$ is the RSA digest that is computed using an l-level accumulation tree on set S. For the base case $l = 1$ the claim trivially holds by Lemma 2; in particular, we have that for any sets $U \neq V$ it holds:

$$\Pr[\chi_1(U) = \chi_1(V) \wedge U \neq V] \leq \nu(k),$$

where $\nu(k)$ is the appropriate negligible function that we get from Lemma 2. Suppose the claim holds for $l = i$, i.e., for any sets $U \neq V$ for the inductive case we have

$$\Pr[\chi_i(U) = \chi_i(V) \wedge U \neq V] \leq \nu(k).$$

Let now

$$\chi_{i+1}(U) = g_{i+1}^{r_{i+1}(\chi_i(U_1)) r_{i+1}(\chi_i(U_2)) \ldots r_{i+1}(\chi_i(U_t))} \mod N_{i+1}$$

for $U_1, U_2, \ldots, U_t \subseteq U$ and

$$\chi_{i+1}(V) = g_{i+1}^{r_{i+1}(\chi_i(V_1)) r_{i+1}(\chi_i(V_2)) \ldots r_{i+1}(\chi_i(V_{t'}))} \mod N_{i+1}$$

for $V_1, V_2, \ldots, V_{t'} \subseteq V$. Consider now the set of prime numbers

$$P(U) = \{r_{i+1}(\chi_i(U_1)), r_{i+1}(\chi_i(U_2)), \ldots, r_{i+1}(\chi_i(U_t))\}$$

and

$$P(V) = \{r_{i+1}(\chi_i(V_1)), r_{i+1}(\chi_i(V_2)), \ldots, r_{i+1}(\chi_i(V_{t'}))\}.$$

We want to compute the probability $\Pr[\chi_{i+1}(U) = \chi_{i+1}(V) \wedge U \neq V]$. The event $\chi_{i+1}(U) = \chi_{i+1}(V) \wedge U \neq V$ can be written as

$$[\chi_{i+1}(U) = \chi_{i+1}(V) \wedge P(U) = P(V) \wedge U \neq V] \vee [\chi_{i+1}(U) = \chi_{i+1}(V) \wedge P(U) \neq P(V) \wedge U \neq V]$$

and therefore by the union bound and by the fact that $\Pr(A \cap B) \leq \Pr(A)$ we can derive the following inequalities:

$$\begin{aligned}
& \Pr[\chi_{i+1}(U) = \chi_{i+1}(V) \wedge U \neq V] \\
\leq\; & \Pr\left[\chi_{i+1}(U) = \chi_{i+1}(V) \wedge P(U) = P(V) \wedge U \neq V\right] \\
& \quad + \Pr\left[\chi_{i+1}(U) = \chi_{i+1}(V) \wedge P(U) \neq P(V) \wedge U \neq V\right] \\
\leq\; & \Pr\left[\chi_{i+1}(U) = \chi_{i+1}(V) \wedge P(U) = P(V) \wedge U \neq V\right] + \Pr\left[\chi_{i+1}(U) = \chi_{i+1}(V) \wedge P(U) \neq P(V)\right] \\
\leq\; & \Pr\left[\chi_{i+1}(U) = \chi_{i+1}(V) \wedge P(U) = P(V) \wedge U \neq V\right] + \nu(k) \\
\leq\; & \Pr\left[P(U) = P(V) \wedge U \neq V\right] + \nu(k)\,.
\end{aligned}$$

Note that $\Pr\left[\chi_{i+1}(U) = \chi_{i+1}(V) \wedge P(U) \neq P(V)\right] \leq \nu(k)$ trivially holds from Lemma 2. Beginning now from the event $P(U) = P(V) \wedge U \neq V$ and for some permutation f of the elements in set $P(V)$, we can derive the following implications:

$$P(U) = P(V) \wedge U \neq V \Rightarrow \bigwedge_{j=1}^{t} \chi_i(U_j) = \chi_i(V_{f(j)}) \wedge U \neq V \Rightarrow \chi_i(U_a) = \chi_i(V_{f(a)}) \wedge U_a \neq V_{f(a)}\,. \quad (2)$$

This is because for two prime representatives $r_1(x_1), r_2(x_2)$ of x_1, x_2 we have that $r_1(x_1) = r_2(x_2) \Rightarrow x_1 = x_2$ and because there has to be some a such that $U_a \neq V_{f(a)}$ since for all i it is $U_i \subseteq U$ and $V_i \subseteq V$ and also $U \neq V$. Since for all events A, B such that $A \Rightarrow B$ it is $\Pr(A) \leq \Pr(B)$, we have that

$$\Pr\left[P(U) = P(V) \wedge U \neq V\right] + \nu(k) \leq \Pr[\chi_i(U_a) = \chi_i(V_{f(a)}) \wedge U_a \neq V_{f(a)}] + \nu(k)\,,$$

for some index a. By the inductive step we have $\Pr[\chi_i(U_a) = \chi_i(V_{f(a)}) \wedge U_a \neq V_{f(a)}] \leq \nu(k)$ and therefore

$$\Pr[\chi_{i+1}(U) = \chi_{i+1}(V) \wedge U \neq V] \leq 2\nu(k)\,,$$

which completes the proof. $\square$

3.4 Authenticating static sets

We now describe how we can use our accumulation-tree based structure to optimally verify membership in a static set in constant time. The following methods will also form the basis for our main authentication schemes for the three-party model in Section 3.5.

Let $S = \{e_1, e_2, \ldots, e_n\}$ be the static set that is outsourced to an untrusted server. As we saw in Section 3.2, the RSA moduli N_i and bases g_i, $1 \leq i \leq l$, are public. The server stores set S and builds the (augmented) accumulation tree $T(\epsilon)$ on top of S. We recall that for every node v of $T(\epsilon)$ that lies at level i ($0 \leq i \leq l-1$), the server stores the prime representative $r_{i+1}(\chi(S_v))$ along with the RSA digest $\chi(S_v)$. Having access only to the set digest $d = \chi(S)$, the client should be able to verify membership in S. We next describe how this is possible.

Queries. We show how the server constructs a proof that is used to validate an element $x \in S$. Let $v_0, v_1, \ldots, v_l$ be the path from x to the root of $T(\epsilon)$, $r = v_l$. Let $B(v)$ denote the set of siblings of node v in $T(\epsilon)$. The proof $\Pi(x)$ is the ordered sequence $\pi_1, \pi_2, \ldots, \pi_l$, where π_i is a tuple of a prime representative and a "branch" witness, i.e., a witness that authenticates every node of the path from the queried node to the root of the tree, v_l. Thus, item π_i of proof $\Pi(x)$ $(i = 1, \ldots, l)$ is defined as:

$$\pi_i = \left(r_i(\chi(v_{i-1})), g_i^{\prod_{u \in B(v_{i-1})} r_i(\chi(u))} \mod N_i \right). \tag{3}$$

For simplicity, we set $\alpha_i = r_i(\chi(v_{i-1}))$ and

$$\beta_i = g_i^{\prod_{u \in B(v_{i-1})} r_i(\chi(u))} \mod N_i \,. \tag{4}$$

For example in Figure 1, the proof for element 2 consists of 3 tuples:

$$\begin{aligned}
\pi_1 &= \left(r_1(2), g_1^{r_1(7) r_1(9) r_1(3)} \mod N_1 \right), \\
\pi_2 &= \left(r_2(\chi(a)), g_2^{r_2(\chi(b)) r_2(\chi(c)) r_2(\chi(d))} \mod N_2 \right), \\
\pi_3 &= \left(r_3(\chi(f)), g_3^{r_3(\chi(e)) r_3(\chi(g)) r_3(\chi(p))} \mod N_3 \right).
\end{aligned}$$

Using the prime representatives, the above proofs can be computed from scratch in $O(\frac{1}{\epsilon} n^{\epsilon})$ time every time there is a new query. However, as we are considering the static case, the server does not have to compute witnesses again and again; it is more time-efficient to use *precomputed witnesses*, i.e., to have the server compute the witnesses once and store them for future use.

Verification. Given the proof $\Pi(x) = \pi_1, \pi_2, \ldots, \pi_l$ for an element x, the client verifies the membership of x in S as follows. First the client checks if $h_1(\alpha_1) = x$, i.e., that α_1 is the prime representative used for the queried element x; then, for $i = 2, \ldots, l$, the client also checks that the following relations hold

$$h_i(\alpha_i) = \beta_{i-1}^{\alpha_{i-1}} \mod N_{i-1} \,, \tag{5}$$

thus verifying that the proof contains correct prime representatives that are correctly accumulated (along the path corresponding to x) in the accumulation tree. Finally, the client verifies the RSA digest (i.e., the RSA accumulation value of the root of the tree) against the locally stored digest, namely that the following relation holds:

$$d = \beta_l^{\alpha_l} \mod N_l \,. \tag{6}$$

The client accepts only if all the relations above hold. As we prove later, the server can forge a proof for an element $y \notin S$ with negligible probability in the security parameter k.

Security. The public key pk in our scheme (see Definition 4) consists of $l = \lceil \frac{1}{\epsilon} \rceil$, the RSA moduli $N_1, N_2, \ldots, N_l$ (not $\phi(N_i)$), the exponentiation bases $g_1, g_2, \ldots, g_l$ and the two-universal functions $h_1, h_2, \ldots, h_l$. Also the adversary is given oracle access to all the algorithms that update and query the accumulation tree and also verify queries. The digest d that appears in Definition 4 is the *root* digest of the accumulation tree. Also, for an element x, $\Pi(x)$ is the set of branch witnesses as defined in Equation 3. The following theorem describes the security of our new construction. The security of our scheme is based on the strong RSA assumption.

Theorem 3 *Our set-membership authentication scheme that combines the accumulation tree and the RSA accumulator is secure according to Definition 4 under the strong RSA assumption.*

Proof: Let $\mathcal{A}$ be a computationally bounded adversary. Let also S be the original set of elements that has been accumulated with the accumulation tree. We define the events:

1. $E_1 =$"$\mathcal{A}$ finds $y \notin S$ and α_1 such that $h_1(a_1) = y$";

2. $E_i =$"$\mathcal{A}$ finds α_{i-1}, α_i and β_{i-1} such that $h_i(\alpha_i) = \beta_{i-1}^{\alpha_{i-1}} \mod N_{i-1}$" for $i = 2, \ldots, l$;

3. $E_{l+1} =$"$\mathcal{A}$ finds α_l and β_l such that $\beta_l^{\alpha_l} = \chi(S) \mod N_l$".

We want to bound the probability $\Pr[E_1 \cap E_2 \cap \ldots \cap E_{l+1}]$. We are using induction. For $l = 1$, we consider the event $E_1 \cap E_2 =$"$\mathcal{A}$ finds $y \notin S$ and α_1, β_1 such that $h_1(a_1) = y$ and $\beta_1^{\alpha_1} = \chi(S) \mod N_1$". By Corollary 1, it is $\Pr[E_1 \cap E_2] \leq \nu(k)$, where $\nu(k)$ is the appropriate negligible function. For $l = i$, suppose now $\Pr[E_1 \cap E_2 \cap \ldots \cap E_i] \leq \nu(k)$. For $l = i + 1$ we write the event $E_{i+1} = E_{i+1}^{(1)} \cup E_{i+1}^{(0)}$ depending on whether $h_i(\alpha_i)$ is the RSA digest of some subset of S or not. Therefore we have

$$\begin{aligned} \Pr[E_1 \cap E_2 \cap \ldots \cap E_{i+1}] &\leq \Pr\left[E_1 \cap E_2 \cap \ldots \cap E_{i+1}^{(1)}\right] + \Pr\left[E_1 \cap E_2 \cap \ldots \cap E_{i+1}^{(0)}\right] \\ &\leq \Pr\left[E_1 \cap E_2 \cap \ldots \cap E_{i+1}^{(1)}\right] + \Pr\left[E_{i+1}^{(0)}\right] . \end{aligned}$$

Consider now the set of RSA digests at level l, i.e., the level of the children of the root of the accumulation tree: It is $\chi(S) = g_i^{a_1 a_2 \ldots a_k} \mod N_i$, where $h_i(a_i)$ is the RSA digest of some subset of S. This observation combined with Corollary 1 and the definition of the event $E_{i+1}^{(0)}$ gives $\Pr\left[E_{i+1}^{(0)}\right] \leq \nu(k)$. Therefore $\Pr[E_1 \cap E_2 \cap \ldots \cap E_{i+1}] \leq \Pr\left[E_1 \cap E_2 \cap \ldots \cap E_{i+1}^{(1)}\right] + \nu(k)$. Note now that the event $E_1 \cap E_2 \cap \ldots \cap E_{i+1}^{(1)}$ implies the event $E_1 \cap E_2 \cap \ldots \cap E_i$ since $h_i(a_i)$ is an RSA digest of some subset of S (i.e., S') and if $y \notin S$ then $y \notin S'$ for all subsets S' of S. Therefore

$$\Pr[E_1 \cap E_2 \cap \ldots \cap E_{i+1}] \leq \Pr\left[E_1 \cap E_2 \cap \ldots \cap E_{i+1}^{(1)}\right] + \nu(k) \leq \Pr[E_1 \cap E_2 \cap \ldots \cap E_i] + \nu(k) \leq 2\nu(k) ,$$

which is $\mathsf{neg}(k)$. □

Complexity. We can now present the main result of this section.

Theorem 4 *Let $0 < \epsilon < 1$ be a fixed constant. Under the strong RSA assumption, we can use the accumulation tree and the RSA accumulator with precomputed witnesses to authenticate a static set S of n elements in the three-party model by storing a data structure of size $O(n)$ at both the source and the server such that:*

1. *Our scheme is secure according to Definition 4;*
2. *The expected query time is $O(1)$;*
3. *The size of the proof is $O(1)$;*
4. *The verification time is $O(1)$;*
5. *The client keeps space $O(1)$.*

Proof: The security of our scheme is derived by Theorem 3. In the static case, we do not have to compute the witnesses each time we query for an element. Namely, we can store all the witnesses in the corresponding nodes of the tree. Therefore the server takes $O(1)$ time in expectation (by using a hash table structure) to pick the correct witness for each level and there are $= O(1/\epsilon) = O(1)$ levels in total, which gives $O(1)$ expected query time. The proof for an element is given by the l

pairs in Equation 3. Since $l = O(1)$ and both the prime representatives and the branch witnesses are quantities that are reduced modulo some N_i (which is independent of n), we have that the size of the proof is $O(1)$. Using now Equations 5 and 6, which are $O(1)$ total, we can verify in $O(1)$ time, since an exponentiation is considered to be a constant time operation. In order to do the verification, the client needs to keep the root RSA digest which has constant size. Finally the update authentication information consists of a signature of the root RSA digest, which has constant size. $\square$

Note that this result applies also in the two-party model. Also, we point out that the same complexity result for a static set can also be achieved without using an accumulation tree and by using a straight-forward application of the RSA accumulator. However, we present this construction to give some intuition about the following section that refers to dynamic sets.

3.5 Authenticating dynamic hash tables

In this section we describe how to use our authentication structure that is based on the accumulation tree to authenticate a dynamic hash table. We first describe our general algorithms and protocols for the three-party model, and then extend our results to the two-party model.

Let $0 < \epsilon < 1$ be a fixed constant. The general idea behind our approach for using the accumulation tree to authenticate hash tables is the following. Let $S = \{e_1, e_2, \ldots, e_n\}$ be the set of elements we would like to authenticate. Instead of building the accumulation tree $T(\epsilon)$ on the elements themselves, as we did in the case of static sets, we consider the elements to be in a hash table that has $O(n)$ buckets, where each bucket contains $O(1)$ elements, and we build the accumulation tree over the buckets. As in the static case, since the size of each bucket is constant, the internal nodes of the accumulation tree have $O(n^\epsilon)$ children. Therefore, we overall end up building a similarly-structured accumulation tree as before, except that now each leaf of the accumulation tree holds the prime representative of the accumulation value of the elements in the bucket corresponding to this leaf (instead of one corresponding element stored at this leaf before).

In particular, consider a bucket L that contains the elements $x_1, x_2, \ldots, x_h$, where $h = O(1)$ (i.e., these elements are mapped to the same bucket through the function used by the hash table to uniformly place the elements in the buckets). The *accumulated bucket value* of bucket L, denoted A_L, is defined as follows:

$$A_L = g_1^{r_1(x_1)r_1(x_2)\ldots r_1(x_h)} \mod N_1 .$$

The accumulated bucket value is computed for each bucket and an accumulation tree is built over the resulting $O(n)$ accumulated bucket values. Note that by doing so, in essence, we add one additional level of accumulations in the accumulation tree, that is, instead of using $l = \lceil \frac{1}{\epsilon} \rceil$ levels of accumulations, we are now using $l' = l + 1$ levels. At the additional (lowest) level, the number of elements that are accumulated is $O(1)$.

Queries and verification. Suppose we want to construct the membership proof for an element $x \in S$. Let $v_0, v_2, \ldots, v_{l'}$ be the path from x to the root r of the tree, $r = v_{l'}$. As before, the proof $\Pi(x)$ is the ordered sequence $\pi_1, \pi_2, \ldots, \pi_{l'}$, where π_i is defined in Equation 3. In order to achieve constant-time queries we must avoid computing π_i repeatedly for every separate query, and therefore we store *precomputed* witnesses. Namely, for every non-leaf node v of the accumulation tree (we consider as leaves the elements within the buckets) that lies in level $1 \le i \le l'$, let $N(v)$ be the set of its children. For every $j \in N(v)$ we store at node v the witness

$$A_j^{(v)} = g_i^{\prod_{u \in N(v) - \{j\}} r_i(\chi(u))} \mod N_i .$$

Therefore, when we query for x, the server follows the path $v_0, v_1, \ldots, v_{l'}$ and collects the corresponding precomputed witnesses $\beta_1 = A_{j_1}^{(v_1)}, \beta_2 = A_{j_2}^{(v_2)}, \ldots, \beta_{l'} = A_{j_{l'}}^{(v_{l'})}$ for some $j_1, j_2, \ldots, j_{l'}$ and as defined in Equation 4. Since the depth of the tree is constant $\left(\lceil \frac{1}{\epsilon} \rceil + 1\right)$, the time needed to construct the proof and thus answer a query is $O(1)$. (We define query time to be the time needed to construct the proof and not the time to search for the specific element, which can however be achieved with another hash table data structure in expected constant time.) Finally, proof verification is performed exactly according to Equations 5 and 6 and, thus, this takes $O(1)$ time.

Updates. We now describe how we can efficiently support updates in the authenticated hash table. Suppose our hash table currently holds n elements and the source wants to insert an element x in the hash table. That element is inserted into a certain bucket L. Let $v_0, v_1, \ldots, v_{l'}$ be the path from the newly inserted element to the root of the tree. The goal of the update algorithm is twofold:

1. All the RSA digests $\chi(v_i)$, $0 \leq i \leq l'$ (note that $\chi(v_0) = x$) and respective prime representatives $r_i(\chi(v_{i-1}))$ $(i = 1, \ldots, l')$ along the path from bucket L to the root of the tree, need to be updated;

2. For all nodes v_i, $1 \leq i \leq l'$, we have to update the witnesses $A_j^{(v_i)}$ where $j \in N(v_i)$. This is required to maintain the query complexity constant.

In order for the first requirement to be satisfied, whenever an update is performed, the RSA digests can be updated by the source and sent to the server as part of the update authentication information. The source also sends the updated (new) prime representatives too, i.e., the values $r_i(\chi(v_{i-1}))$ for all $i = 1, \ldots, l'$. In this way the untrusted server does not compute them from scratch but just replaces the old ones with the ones received. To satisfy the second requirement, we use the following result from [39] for efficiently maintaining updated precomputed witnesses and overall achieving constant query time.

Lemma 4 (Updating precomputed witnesses [39]) *Let N be an RSA modulus. Given the elements $x_1, x_2, \ldots, x_n$, N and g, without the knowledge of $\phi(N)$, we can compute $A_i = g^{\prod_{j \neq i} x_j} \bmod N$ for $i = 1, \ldots, n$ in $O(n \log n)$ time.*

In order to compute the updated witnesses, the server uses the algorithm in [39] that provides the above result for all nodes v_i, $1 \leq i \leq l'$ as follows. For each v_i (we recall that v_i is the node on the path of the update), it uses the algorithm with inputs the elements $r_i(\chi(j))$ for $j \in N(v_i)$, the RSA modulus N_i and the exponentiation base g_i. In this computation the updated prime representative $r_i(\chi(v_{i-1}))$, where $v_{i-1} \in N(v_i)$, that was received by the source, is used. This computation outputs the witnesses $A_j^{(v_i)}$ where $j \in N(v_i)$ (note that the witness $A_{v_{i-1}}^{(v_i)}$ remains the same). Also, since it is run on $O(1/\epsilon)$ nodes v with $|N(v)| = O(n^\epsilon)$, we have, by Lemma 4, that the witnesses can be updated in $O(n^\epsilon \log n)$ time (for the complete result see Lemma 5).

However, since we are using a hash table (with $O(n)$ buckets) we should expect that at some point we will need to rebuild the table (i.e., rehash all the elements and reinsert them in a bigger or smaller hash table). This is done as follows. During the evolution of the data structure, we maintain a hash table that always can store more elements than the currently stored elements. We call the number of the elements that can be stored in the hash table we are using "capacity" (the capacity can also be viewed as the number of buckets of the hash table). Let M_i be the capacity of the hash table after update i and m_i be the number of the elements actually stored in the hash table after update i. Note that whenever we have an update in the hash table, it is $m_i = m_{i-1} \pm 1$ and whenever update i causes a rebuilt of the hash table it is $M_i \neq M_{i-1}$. In order to ensure that the complexity results are maintained we have to make sure that $\frac{M_i}{4} \leq m_i \leq M_i$. If m_i violates the above bounds, we have to rebuild the hash table from scratch. The general policy we follow is given in the following definition:

Definition 6 (Rebuilding the hash table) *Define $\alpha_i = \frac{m_i}{M_i}$ to be the load factor of the authenticated hash table after update i. If $\alpha_i = 1$ (full table) we double the capacity of the hash table. If $\alpha_i = \frac{1}{4}$ (near empty table) we halve the capacity of the hash table.*

The rebuilding method described in Definition 6, adjusted to our authenticated hash table construction is essential to get the necessary amortized results of Lemma 5 which constitutes the main complexity result of our work (for similar methods see [13]).

Lemma 5 *Let $0 < \epsilon < 1$ be a fixed constant. Given a hash table for n elements with $O(n)$ buckets of expected size $O(1)$ and the accumulation tree $T(\epsilon)$ built on top of it, we can support updates in $O(n^\epsilon \log n)$ expected amortized time without the knowledge of $\phi(N_i)$ $(i = 1, \dots, l')$ by using the rebuilding policy of Definition 6.*

Proof: Suppose there are currently n elements in the hash table and that the capacity of the table is M. Note than $M/4 \leq n \leq M$. As we know, there are M buckets and each bucket stores $O(1)$ elements in expectation. When an update takes place and no rebuilding of the table is triggered, we have to update all the witnesses along the path of the update of the accumulation tree. By using the algorithm described in Lemma 4, we can update the witnesses within the bucket in expected time $O(1)$, since the size of the bucket is an expected value. The witnesses of the internal nodes can be updated in time $O(M^\epsilon \log M)$ and therefore the overall time is $O(M^\epsilon \log M)$ in expectation. When a rebuilding of the table is triggered then the total time needed is $O(M \log M)$ since there is a constant number of levels in the accumulation tree, the time we need to spend at each node is $O(n^\epsilon \log n)$ (since the degree of any internal node is $O(n^\epsilon)$) and the maximum number of nodes that lie in any level is $O(n^{1-\epsilon})$. Therefore the *actual cost* of an update is *expected* $O(M^\epsilon \log M)$, when no rebuilding is trigerred and $O(M \log M)$ otherwise. We are interested in the expected value of the amortized cost (expected amortized cost) of an update. We do the analysis by defining the following potential function:

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

The amortized cost 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 do the analysis more precise we define some constants. Let c_1 be that constant such that if the update cost C is $O(M_i^\epsilon \log M_i)$, it is

$$C \leq c_1 M_i^\epsilon \log M_i\,. \tag{7}$$

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

$$R \leq r_1 m_i \log m_i\,. \tag{8}$$

Also we note that in all cases it holds

$$\frac{M_i}{4} \leq m_i \leq M_i\,. \tag{9}$$

We do 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 $m_i = m_{i-1} + 1$. Therefore the amortized cost will be:

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

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

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

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 $m_i = m_{i-1} + 1$. If now $\alpha_i < \frac{1}{2}$ the amortized cost will be:

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

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

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

3. $\alpha_{i-1} < \frac{1}{2}$ (deletion). Here we have $m_i = m_{i-1} - 1$. 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 cost of the deletion is going to be:

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

In case now the hash table has to be rebuilt (which takes time $O(m_i \log m_i)$), we have that

$M_i = M_{i-1}/2$, $M_i = 4m_i$ and therefore the amortized cost is:

$$\begin{aligned}
\mathbb{E}[\hat{\gamma_i}] = \mathbb{E}[\gamma_i] + F_i - F_{i-1} &\leq r_1 m_i \log m_i + c(M_i/2 - m_i)\log M_i - c(M_{i-1}/2 - m_{i-1})\log M_{i-1} \\
&\leq r_1 m_i \log m_i + c(M_i/2 - m_i)\log M_i - c(M_i - (m_i+1))\log 2M_i \\
&\leq r_1 m_i \log m_i - c(M_i/2 - 1)\log M_i - c(3m_i - 1) \\
&\leq r_1 m_i \log m_i - cM_i/2 \log M_i + c\log M_i \\
&\leq r_1 M_i \log M_i - (c/2)M_i \log M_i + c\log M_i \\
&\leq c\log M_i\,,
\end{aligned}$$

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

4. $\alpha_{i-1} \geq \frac{1}{2}$ (deletion). In this case we have $M_{i-1} = M_i$. If $\alpha_i \geq \frac{1}{2}$, the amortized cost will be:

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

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

$$\begin{aligned}
\mathbb{E}[\hat{\gamma_i}] = \mathbb{E}[\gamma_i] + F_i - F_{i-1} &\leq c_1 M_i^\epsilon \log M_i + c(M_{i-1}/2 - m_i - 2m_{i-1} + M_{i-1})\log M_i \\
&= c_1 M_i^\epsilon \log M_i + c(3M_{i-1}/2 - (m_{i-1} - 1) - 2m_{i-1})\log M_i \\
&= c_1 M_i^\epsilon \log M_i + c(3M_{i-1}/2 - 3m_{i-1} + 1)\log M_i \\
&= c_1 M_i^\epsilon \log M_i + c(3(1/\alpha_{i-1})m_{i-1}/2 - 3m_{i-1} + 1)\log M_i \\
&\leq c_1 M_i^\epsilon \log M_i + c\log M_i\,.
\end{aligned}$$

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

$$\mathbb{E}[\hat{\gamma_i}] \leq c_1 M_i^\epsilon \log M_i + 2c\log M_i\,.$$

By using now Equation 9 there is a constant r such that $\mathbb{E}[\hat{\gamma_i}] \leq r m_i^\epsilon \log m_i$ which implies that the expected value of the amortized cost of any update (insertion/deletion) in an authenticated hash table containing n elements is $O(n^\epsilon \log n)$ for $0 < \epsilon < 1$. □

Note that so far the results we have presented refer to *positive* hash table queries (i.e., hash table queries made for elements that exist in our set). We describe now how we can support non-membership queries as well. We are going to do that by using positive membership queries: In each bucket L, we maintain all elements $y_i \in L$ sorted—in case elements are drawn from an unordered universe, we first apply a cryptographic hash function to impose some order on the elements. Let $y_1, y_2, \ldots, y_{|L|}$ be the elements stored in a bucket L in increasing order. Instead of computing prime representatives of y_i we compute prime representatives of the $|L|+1$ intervals (y_i, y_{i+1}) for $i = 0, \ldots, |L|$, where y_0 and $y_{|L|+1}$ denote $-\infty$ and $+\infty$, respectively. The proof of non-membership for an element $x \in (y_i, y_{i+1})$ is equivalent to the proof of membership for interval (y_i, y_{i+1}). As the bucket size is maintained to be $O(1)$ the query complexity is maintained expected $O(1)$ for non-membership queries as well (note that in general, this construstion adds an $O(\log k)$ overhead in the update time for k-sized buckets but in our case it does not matter since the buckets are of constant size). Note that we do not impose an ordering across all the elements stored in the hash table. A total ordering would increase the complexity and would not take advantage of the fact that we are using a hash table.

The main result of this section (for the three-party model) is as follows:

Theorem 5 *Let $0 < \epsilon < 1$ be a fixed constant. Under the strong RSA assumption, we can use the accumulation tree and the RSA accumulator with precomputed witnesses to authenticate a dynamic hash table of n elements in the three-party model by storing a data structure of size $O(n)$ at both the source and the server such that:*

1. *Our scheme is secure according to Definition 4;*
2. *The expected amortized update time at the server is $O(n^\epsilon \log n)$;*
3. *The expected amortized update time at the source is $O(1)$;*
4. *The expected query time is $O(1)$;*
5. *The size of the proof is $O(1)$;*
6. *The verification time is $O(1)$;*
7. *The client keeps space $O(1)$;*
8. *The update authentication information has size $O(1)$.*

Proof: The security of our scheme is derived by Theorem 3. The complexity of the update time at the server is derived by the amortized analysis in Lemma 5. For the update time at the source (which involves computation of the new RSA digests and prime representatives along the path of the update), we are using the fact that the source knows $\phi(N_i)$ in order to derive the constant amortized time: Suppose the source wants to insert/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 certain bucket and d_l is the root RSA digest). The source first computes the new value of the bucket d'_1 by exponentiating d_1 to $r_1(x)$ (insertion) or to $r_1(x)^{-1}$ (deletion) and then reducing modulo N_1. Note that this is always feasible to compute, since the source knows $\phi(N_1)$. Next, for each $i = 2, \ldots, l$, the source computes d'_i by exponentiating d_i to the product $r_i(d_{i-1})^{-1} r_i(d'_{i-1})$ (where $r_i(d'_{i-1})$ is the new prime representative computed and which will be sent to the server) and then reducing modulo N_i. Therefore, the total update time is $O(1)$ since l is $O(1)$. However, rebuilding the hash table is needed and the expected amortized time will be in this case $O(1)$ (we can prove that by using a potential function that does not contain the logarithmic factor of the potential function used in Lemma 5). The query time at the server is expected $O(1)$ since the server can use another hash table (note that the update time of this hash table does not influence the amortized analysis of Lemma 5) to pick the correct witness at every node of the accumulation tree. (5), (6) and (7) are derived as in Theorem 4. The update authentication information contains all the RSA digests along the path of the update, the respective new prime representatives and a constant size signature of the root RSA digest: The size of the path is constant and each RSA digest/prime representative is a number reduced modulo N_i, which occupies $O(1)$ space. Therefore the update authentication information is $O(1)$. Finally, in order to implement a complete authenticated hash table, we have to authenticate non-membership queries as well. We do that by keeping elements sorted in the buckets which as we saw before, does not increase the complexity due to the constant-sized buckets. □

Finally, note that if we restrict ourselves to the three-party model, we can achieve constant expected amortized update time at the untrusted server too, by keeping the update authentication information constant and increasing the query time to expected $O(n^\epsilon)$:

Theorem 6 *Let $0 < \epsilon < 1$ be a fixed constant. Under the strong RSA assumption, we can use the accumulation tree and the RSA accumulator without precomputed witnesses to authenticate a dynamic hash table of n elements in the three-party model by storing a data structure of size $O(n)$ at both the source and the server such that:*

1. *Our scheme is secure according to Definition 4;*
2. *The expected amortized update time at the server is $O(1)$;*
3. *The expected amortized update time at the source is $O(1)$;*
4. *The expected query time is $O(n^\epsilon)$;*
5. *The size of the proof is $O(1)$;*
6. *The verification time is $O(1)$;*
7. *The client keeps space $O(1)$;*
8. *The update authentication information has size $O(1)$.*

Proof: (1), (3), (5), (6), (7) and (8) are derived as in Theorem 5. The server, whenever there is an update, does not have to do anything: It just receives the updated digests and prime representatives from the source and uses them to overwrite the previous ones. Therefore the update time for the server is the same with the update time for the source. As for the query time, the server computes the witnesses online, i.e., it performs $O(n^\epsilon)$ exponentiations for the internal nodes of the accumulation tree and $O(1)$ exponentiations (in expectation) for the buckets. Therefore the query time is expected $O(n^\epsilon)$. $\square$

As we will see next, Theorem 6 also applies to the two-party model provided we add and extra round of communication between the client and the server. Finally, we note that we can choose the scheme being best suited for the application of interest: in particular, we can use the scheme of Theorem 5 for applications where updates are rare and queries are frequent, whereas we can use the scheme of Theorem 6 for applications where updates are much more frequent than queries (e.g., auditing).

3.6 Two-party model

We now describe how we can implement an authenticated hash table using the accumulation tree with precomputed witnesses in the two-party model. We recall that the two-party model has the following main differences from the three-party model:

1. The client locally stores (and updates) the RSA digest and does not receive a signed RSA digest from the trusted source, as it happens in the three party model;
2. The client is not issuing only queries to the untrusted server but is also issuing updates;
3. There is no trusted party and no PKI is used.

In the two-party model the untrusted server also computes the new prime representatives whenever there is an update. This is possible since the information used to compute prime representatives is included in the public key pk. We also recall that, as we discussed at the end of Section 2.4, it is very important, after the client issues an update transforming the set of elements from S to

S', that the client correctly updates the local digest to a new one that is consistent with the new set S'. This is crucial for the security of a set-membership authentication scheme; and although this can be trivially achieved in the three-party model where the source computed the new digest over the new locally stored set S', it is more challenging to achieve in the two-party model where the client does not locally store the data set.

Consider now the case where the untrusted server stores the current set S and the augmented accumulation tree built on it which in particular includes the prime representatives of all the RSA digests. Assuming that the client stores the correct RSA digest $d(S)$ of the set S and that the server uses precomputed witnesses at every node of the accumulation tree, it is easy to see that set-membership queries for any given element can be answered and verified exactly in the same way as in the three-party model; namely, a query takes $O(1)$ expected time and a verification takes $O(1)$ time. Thus we only need to describe how updates are handled and, in particular, how the client can correctly update the set digest, thus maintaining the invariant that at all times the client locally stores the correct set digest, i.e., a digest that corresponds to the exact history of updates in the set. In what follows, we give the details on updates in the two-party model.

Updates. Suppose the client issues the update $\mathsf{delete}(x)$ to set S resulting in set S', and let $v_0, v_1, \ldots, v_{l'}$ be the nodes of the accumulation tree from the node v_0 that stores x to the root of the accumulation tree. The update protocol needs to achieve two goals: the server needs to perform the update and the client needs to compute the new digest and verify the update performed by the server. While the server performs the update, it also constructs a *consistency proof* that is given to the client in order for the client to perform its digest update and verification.

In particular, the server initially treats the update as a membership query for element x and constructs the membership proof for x as defined in Equations 3 and 4, i.e., the pairs of prime representatives and branch witnesses (α_i, β_i) for $i = 1, \ldots, l'$. Then the server performs the update as in the three-party model (i.e., computing new witnesses, RSA digests and prime representatives along the update path). Let α'_i for $i = 2, \ldots, l'$ be the new prime representatives computed by the server (note that since x has been removed there is no new prime representative for x). The consistency proof that the server sends to the client, which corresponds to the performed update, consists of the following two components:

1. The set of pairs (α_i, β_i) for $i = 1, \ldots, l'$, which is a verification proof for x in S (i.e., the set *before* the update);
2. The set of new prime representatives α'_i for $i = 2, \ldots, l'$.

We distinguish between the two kinds of proofs returned by the server: After a query, as we have seen also in the three-party model the server returns a *verification* proof; after an update, the server returns a *consistency* proof which in fact includes a verification proof for the updated element (plus the new prime representatives along the update path).

After receiving the consistency proof, the client performs the following. First, it verifies the verification proof as in Equations 5 and 6. If the verification accepts, the client, apart from knowing that $x \in S$, can also compute the correct RSA digests of nodes $v_1, \ldots, v_{l'}$ by setting

$$\chi(v_i) = \beta_i^{\alpha_i} \mod N_i \,,$$

for all $i = 1, \ldots, l'$. Note that these are the correct RSA digests corresponding to S, i.e., the set before the removal of x. These digests must be updated to reflect the removal of x and thus locally replace the old RSA digest $\chi(v'_l)$ with the new RSA digest $\chi'(v'_l)$ of the root of the accumulation tree corresponding to S'. In order to compute and verify this new digest the client uses the prime representatives α'_i, for $i = 2, \ldots, l'$ and the old *already verified* RSA digests $\chi(v_i)$, for $i = 1, \ldots, l'$, as follows. One-by-one, and since the client knows the trapdoor information $\phi(N_i)$ for $i = 1, \ldots, l'$, the client can efficiently update $\chi(v_i)$ to $\chi'(v_i)$ first by setting

$$\chi'(v_1) = \chi(v_1)^{\alpha_1^{-1}} \mod N_1 \,, \tag{10}$$

therefore computing the correct *updated* RSA digest of node v_1. Then the client sets

$$\chi'(v_i) = \chi(v_i)^{\alpha_i^{-1}\alpha'_i} \mod N_i \,, \tag{11}$$

for $i = 2, \ldots, l'$. In Equation 11, since $\chi(v_i)$ and α_i have been verified to be the correct corresponding values of the accumulation tree before the update, it suffices for the client to verify that α'_i is the correct *new* prime representative after the update. To achieve that, the client firstly verifies that α'_i is a $3|N_i| - 1$-bit prime number (this is feasible with an efficient primality check) and also that

$$h_i(\alpha'_i) = \chi'(v_{i-1}) \,, \tag{12}$$

for $i = 2, \ldots, l'$. By the fact that $\chi'(v_1)$ is provably correct by Equation 10, the client can verify the correctness of α'_2 by Equation 12 for $i = 2$, and therefore verify the correctness of $\chi'(v_2)$ by Equation 11. By following this chain of computations the client ends up computing the provably correct updated digest $\chi'(v'_l)$ with non-negligible probability (due to the security of the verification test), as required by the security definition, i.e., the client, after any update provably possesses the correct updated digest with non-negligible probability. The insertion of an element can be performed similarly.

Finally we note that when the server rebuilds the hash table, the client has to receive all the elements, verify them, rebuild the hash table by computing new prime representatives and send everything over to the server. This will provide the amortized results in Theorem 7. We present now the main result for the two-party model:

Theorem 7 *Let $0 < \epsilon < 1$ be a fixed constant. Under the strong RSA assumption, we can use the accumulation tree and the RSA accumulator with precomputed witnesses to authenticate a dynamic hash table of n elements in the two-party model by storing a data structure of size $O(n)$ such that:*

1. *Our scheme is secure according to Definition 4;*
2. *The expected amortized update time at the server is $O(n^\epsilon \log n)$;*
3. *The amortized update time at the client is $O(1)$;*
4. *The expected query time is $O(1)$;*
5. *The size of the verification proof is $O(1)$;*
6. *The amortized size of the consistency proof is $O(1)$;*
7. *The verification time is $O(1)$;*
8. *The client keeps space $O(1)$.*

Proof: The security in the two-party model is derived by Theorem 3 and by the fact that we provide a way for the client to update the digest whenever he issues an update, with non-negligible probability (see the protocol in the above description). The update time at the server is derived by

Lemma 5 (use of precomputed witnesses). Verification time, verification proof size and query time are derived as in Theorem 5. The consistency proof has $O(1)$ amortized size, since there will be an update by the client that will trigger a rebuilding of the hash table, thus making the server send all the elements over to the client. This also makes the update time at the client to be amortized $O(1)$, by following a similar amortized analysis as in Lemma 5. □

We now present the result that uses on-line witness computation in the two party model:

Theorem 8 *Let $0 < \epsilon < 1$ be a fixed constant. Under the strong RSA assumption, we can use the accumulation tree and the RSA accumulator without precomputed witnesses to authenticate a dynamic hash table of n elements in the two-party model by storing a data structure of size $O(n)$ such that:*

1. *Our scheme is secure according to Definition 4;*
2. *The expected amortized update time at the server is $O(1)$;*
3. *The amortized update time at the client is $O(1)$;*
4. *The expected query time is $O(n^\epsilon)$;*
5. *The size of the verification proof is $O(1)$;*
6. *The amortized size of the consistency proof is $O(1)$;*
7. *The verification time is $O(1)$;*
8. *The client keeps amortized space $O(1)$.*

Proof: The security in the two-party model is derived by Theorem 3. In this scenario we assume that the server always keeps the digests updated and therefore, he can compute witnesses on-line in expected $O(n^\epsilon)$ time. Again, the query time is expected due to the expected bound on the capacity of the buckets. We describe now the extension in the communication protocol so that the update time at the server is constant. Whenever the client issues an update, the server sends back a consistency proof as in Theorem 7. However, the server does not do the update as before. After the client verifies the proof, he computes new prime representatives along the path (note that this can be done in constant time) of the update and sends the new digests to the server. The server, just receives the digests and overwrites the old ones in constant time. That keeps the update time constant and introduces another round of communication between the client and the server. □

3.7 A more practical scheme

The solution we have presented so far uses different RSA moduli for each level of the tree, where as we move higher in the tree, each new RSA moduli has a bit-length that is three times longer than the bit-length of the previous-level RSA moduli. Therefore, computations corresponding to higher levels in the accumulation tree are more expensive, since they involve modular arithmetic operations over longer elements. This increase in the lengths of the RSA moduli is due to the need to compute, for the elements stored at every level in the tree, prime representatives of size that is three times as large as the size of the elements (see Lemma 1). Although from a theoretical point of view this is not a concern as the number of the levels of the tree is constant (i.e., $1/\epsilon$), from a practical point of view this can be really prohibitive for efficiently implementing our schemes.

To overcome this complexity overhead, we want 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 [4]. Instead of using two-universal hash functions to map (general) integers to primes of increased size, the idea is to employ random oracles [5] 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 the value $2^t 2^b g(x) + d$, where $g(x)$ is the output of length b of a random oracle (which in practice is the output of a cryptographic hash function) at the end of which we append b zeros so that we make this number large enough, t is a value that equals to the number of bits we are shifting $2^b g(x)$ to the left, and $d = 1, 3, \ldots, 2^t - 1$ is a number we are adding so that $2^t 2^b g(x) + d$ is a prime. Note that we require that t is related to b according to Equation 13 of Theorem 9.

In the following, we denote by $q(x)$ a prime representative of x computed by the above procedure, i.e., the output of a procedure that transforms a k-bit integer into a k'-bit prime, where $k' < k$. Note that the above procedure (i.e., the computation of $q(x)$) cannot map two different integers to the same prime. This can be derived by the random oracle property, namely that for $x_1 \neq x_2$, w.h.p. it is $g(x_1) \neq g(x_2)$. This implies that the intervals $[2^t 2^b g(x_1), 2^t 2^b g(x_1) + 2^t - 1]$ and $[2^t 2^b g(x_2), 2^t 2^b g(x_2) + 2^t - 1]$ are disjoint. Finally we show that we can make sure that with high probability we will always be able to find a prime within the specified interval.

Theorem 9 *Let x be a k-bit integer and let $a = 2^b g(x)$ be the output of a b-bit random oracle with b zeros appended at the end. The interval $[2^t a, 2^t a + 2^t - 1]$ contains a prime with probability at least $1 - 2^{-b}$ provided*

$$b \leq \left\lfloor \log(1 + \sqrt{2^t + 4e^{2^t - 1}}) - 1 \right\rfloor . \tag{13}$$

Proof: By the Prime Distribution Theorem we have that the number of primes less than n is approximately $\frac{n}{\ln n}$. Therefore, we want to compute the probability

$$\Pr\left[\frac{2^t a + 2^t - 1}{\ln(2^t a + 2^t - 1)} - \frac{2^t a}{\ln(2^t a)} \geq 1\right] = \Pr\left[a \leq \frac{e^{2^t - 1}}{2^t}\right],$$

by assuming $\ln(2^t a + 2^t - 1) \simeq \ln(2^t a)$ since $a > 2^b >> 2^t$. By the random oracle property we have that

$$\Pr\left[a \leq \frac{e^{2^t-1}}{2^t}\right] = \Pr\left[2^b g(x) \leq \frac{e^{2^t-1}}{2^t}\right] = \frac{e^{2^t-1}}{2^{b+t}} \frac{1}{2^b}.$$

Note that

$$\frac{e^{2^t-1}}{2^{b+t}} \frac{1}{2^b} \geq 1 - \frac{1}{2^b} \Leftrightarrow \frac{1 - \sqrt{2^t + 4e^{2^t-1}}}{2} \leq 2^b \leq \frac{1 + \sqrt{2^t + 4e^{2^t-1}}}{2},$$

which gives $b \leq \left\lfloor \log(1 + \sqrt{2^t + 4e^{2^t-1}}) - 1 \right\rfloor$ since b is a positive integer. This completes the proof. □

Using Theorem 9, we can pick the length of the output of the random oracle to ensure hitting a prime with high probability. For example, for $t = 9$ we get $b \leq 368$, which is true for most practical hash functions used today (e.g., SHA-256).

Using the above method, we can still accumulate primes in the exponent but this time without having to increase the size of the RSA moduli at any level of the tree. The only conditions we need in order to securely use the RSA accumulator are:

1. the safe accumulation of primes that map to unique integers (i.e., each accumulated prime can only represent one integer), and

2. the bit-length of accumulated primes is smaller than the bit-length of the used RSA modulus.

Thus, we can apply our new procedure for computing prime representatives to all of the constructions in Section 3 with one important efficiency improvement: the same RSA moduli and exponentiation bases are used at all levels of the accumulation tree. With this heuristic, we overall get the same security and complexity results as before, but now we have a more practical accumulator with security that is now based on both the strong RSA and the random oracle assumptions.

4 Scheme based on the bilinear-map accumulator

In this section we use the bilinear-map accumulator to construct authenticated hash tables. We use exactly the same methodology as the one used in Section 3, that is, nested invocations of accumulators in a constant-depth tree, to overall obtain similar complexity and security results with the solution presented before. Accordingly, we use the same structure in presenting and proving our results.

4.1 System setup

The setup for this solution is simpler than the authenticated hash tables based on the RSA accumulator. As we will see, at every level, the digests are elements of the same group and not of different groups as it was the case with the nested applications of the RSA accumulators. To achieve that, we are going to use a collision resistant hash function h that takes as input elements of the multiplicative cyclic group and outputs an element in $\mathbb{Z}_p^*$.

Therefore, in the two-party model, the client picks an exponentiation base g that is a generator of a multiplicative cyclic group $\mathbb{G}$ of prime order p. Then the client randomly picks a number $s \in \mathbb{Z}_p^*$ and keeps that secret (trapdoor information). The generator g is used as the exponentiation base in all the levels as well. All the above are chosen uniformly at random as indicated by Definition 3 (basically the client has to generate the tuple $\mathsf{t} = (p, \mathbb{G}, \mathbb{G}_M, e, g)$). Finally, the client decides on an upper bound q of the total number of elements that will be accumulated and sends the numbers $g, g^s, g^{s^2}, \ldots, g^{s^q}$ to the untrusted server.

In the three-party model, we have the same setup with the difference that the source is responsible for choosing p, $\mathbb{G}$, $\mathbb{G}_M$, e, g and s. Everything is made public (both to the servers and to the client) except for the trapdoor information s, which is crucial for the security of the scheme.

Finally, in both models, we are using a collision resistant hash function $h : \mathbb{G} \to \mathbb{Z}_p^*$. In this way we make sure that the output accumulated value can be used as input to the next level of accumulation, since we can only accumulate elements of $\mathbb{Z}_p^*$ and not elements of $\mathbb{G}$. The collision resistance hash function h serves as the respective "prime representative" function we used in Section 3, with the difference that there is no constraint that the output should be a prime number: it suffices that the output is a number in $\mathbb{Z}_p^*$.

4.2 Main authenticated data structure

Let S be the set we would like to authenticate. The elements of the set are integers belonging to $\mathbb{Z}_p^*$. Similarly with the RSA construction, we can hierarchically employ the bilinear-map accumulator over set S. Therefore, for any tree node v we define an accumulation value $\psi(v)$ of node v, recursively along the tree structure: For every leaf node v in tree $T(\epsilon)$ that stores element e, we set $\psi(v) = e$, while for every non-leaf node v in $T(\epsilon)$ that lies in level $1 \leq i \leq l$, we set:

$$\psi(v) = g^{\prod_{u \in N(v)} (h(\psi(u)) + s)}, \tag{14}$$

where $\psi(u)$ is the bilinear digest of node u (which is an element of the multiplicative cyclic group $\mathbb{G}$), $N(v)$ is the set of children of node v, g is the generator of the multiplicative group $\mathbb{G}$, s is the trapdoor information that is kept secret and $h(\psi(u)) \in \mathbb{Z}_p^*$ is a cryptographic hash of $\psi(u)$, computed with the collision resistant hash function h that we introduced before.

Definition 7 *Given a set $S = \{e_1, e_2, \ldots, e_n\}$ of n elements in $\mathbb{Z}_p^*$, a multiplicative group $\mathbb{G}$ of prime order p and the accumulation tree $T(\epsilon)$ built on top of them, we define the bilinear digest of any node v of the accumulation tree to be equal to $\psi(v)$, also denoted with $\psi(S_v)$, where S_v is the set that is defined by the subtree rooted at v. The* bilinear digest *of the set S is equal to $\psi(r)$, where r is the root of tree $T(\epsilon)$.*

For simplicity, we use both $\psi(S_v)$ and $\psi(v)$ to denote the bilinear digest of node v, S_v being the set of elements contained in the subtree rooted at node v. In the following we prove the main collision-resistance property of the above authentication structure.

Theorem 10 (Collision resistance) *Let k be the security parameter and $U = \{u_1, u_2, \ldots, u_n\}$ a set of n elements. Given the associated accumulation tree $T(\epsilon)$ built on top of U, under the q-strong Diffie-Hellman assumption, the probability that a computationally bounded adversary $\mathcal{A}$, knowing only the bilinear pairings parameters $\mathsf{t} = (p, \mathbb{G}, \mathbb{G}_M, e, g)$ and the elements of $\mathbb{G}_M$ $g, g^s, g^{s^2}, \ldots, g^{s^q}$ ($q \geq n$) for some s chosen at random from $\mathbb{Z}_p^*$, can find another set $V \neq U$ such that $\psi(V) = \psi(U)$ is $\mathsf{neg}(k)$.*

Proof: As in Theorem 2, we prove the following claim by induction: Given an accumulation tree of l levels, it is difficult for a computationally bounded adversary to find two different sets U, V such that $\psi_l(U) = \psi_l(V)$, where $\psi_l(S)$ is the bilinear digest that is computed using an l-level accumulation tree on set S. For the base case $l = 1$ the claim trivially holds by Lemma 3; in particular, we have that for any sets $U \neq V$ it holds:

$$\Pr[\psi_1(U) = \psi_1(V) \wedge U \neq V] \leq \nu(k)\,,$$

where $\nu(k)$ is the appropriate negligible function that we get from Lemma 3. Suppose the claim holds for $l = i$, i.e., for any sets $U \neq V$ for the inductive case we have

$$\Pr[\psi_i(U) = \psi_i(V) \wedge U \neq V] \leq \nu(k)\,.$$

Let now

$$\psi_{i+1}(U) = g^{(h(\psi_i(U_1))+s)(h(\psi_i(U_2))+s)\ldots(h(\psi_i(U_t))+s)}$$

for $U_1, U_2, \ldots, U_t \subseteq U$ and

$$\psi_{i+1}(V) = g^{(h(\psi_i(V_1))+s)(h(\psi_i(V_2))+s)\ldots(h(\psi_i(V_{t'}))+s)}$$

for $V_1, V_2, \ldots, V_{t'} \subseteq V$. Consider now the set of element in $\mathbb{Z}_p^*$

$$P(U) = \{h(\psi_i(U_1)) + s, h(\psi_i(U_2)) + s, \ldots, h(\psi_i(U_t)) + s\}$$

and

$$P(V) = \{h(\psi_i(V_1)) + s, h(\psi_i(V_2)) + s, \ldots, h(\psi_i(V_{t'})) + s\}\,.$$

We want to compute the probability $\Pr[\psi_{i+1}(U) = \psi_{i+1}(V) \wedge U \neq V]$. By following the same logic as in the proof of Theorem 2 we have that

$$\Pr[\psi_{i+1}(U) = \psi_{i+1}(V) \wedge U \neq V] \leq \Pr\left[P(U) = P(V) \wedge U \neq V\right] + \nu(k)\,.$$

Beginning now from the event $P(U) = P(V) \wedge U \neq V$ and for some permutation f of the elements in set $P(V)$, we can derive the following implications:

$$\begin{aligned} P(U) = P(V) \wedge U \neq V & \Rightarrow \bigwedge_{j=1}^{t} \left[h(\psi_i(U_j)) + s = h(\psi_i(V_{f(j)})) + s\right] \wedge U \neq V \\ & \Rightarrow \psi_i(U_a) = \psi_i(V_{f(a)}) \wedge U_a \neq V_{f(a)} \,. \end{aligned}$$

This is because we are using a collision resistance hash function h such that $h(x_1) = h(x_2) \Rightarrow x_1 = x_2$ with probability $1 - \mathsf{neg}(k)$ (we can apply a union bound here and be even more formal by adding another $\nu(k)$, i.e., the probability of finding a collision in h, to the final bound) and because there has to be some a such that $U_a \neq V_{f(a)}$ since for all i it is $U_i \subseteq U$ and $V_i \subseteq V$ and also $U \neq V$. Since for all events A, B such that $A \Rightarrow B$ it is $\Pr(A) \leq \Pr(B)$, we have that

$$\Pr\left[P(U) = P(V) \wedge U \neq V\right] + \nu(k) \leq \Pr[\psi_i(U_a) = \psi_i(V_{f(a)}) \wedge U_a \neq V_{f(a)}] + \nu(k) \,,$$

for some index a. By the inductive step we have $\Pr[\psi_i(U_a) = \psi_i(V_{f(a)}) \wedge U_a \neq V_{f(a)}] \leq \nu(k)$ and therefore

$$\Pr[\psi_{i+1}(U) = \psi_{i+1}(V) \wedge U \neq V] \leq 2\nu(k) \,,$$

which completes the proof. $\square$

4.3 Authenticating static sets

Similarly with the RSA accumulator, the construction of a proof for an element x is done as follows. Let $v_0, v_1, \ldots, v_l$ be the path from x to the root of $T(\epsilon)$, $r = v_l$. Let $B(v)$ denote the set of siblings of node v in $T(\epsilon)$. The proof $\Pi(x)$ is the ordered sequence $\pi_1, \pi_2, \ldots, \pi_l$, where π_i is a tuple of an element of $\mathbb{G}$, hash value and a "branch" witness (that is also an element of $\mathbb{G}$), i.e., a witness that authenticates the missing node of the path from the queried node to the root of the tree, v_l. Thus, item π_i of proof $\Pi(x)$ $(i = 1, \ldots, l)$ is defined as:

$$\pi_i = \left(\psi(v_{i-1}), g^{\prod_{u \in B(v_{i-1})}(h(\psi(u))+s)}\right) \,. \tag{15}$$

Now we set $\alpha_i = \psi(v_{i-1})$ and $\beta_i = g^{\prod_{u \in B(v_{i-1})}(h(\psi(u))+s)}$. For the verification, given the proof $\Pi(x) = \pi_1, \pi_2, \ldots, \pi_l$ for an element x, the client verifies the membership of x in S as follows. Since the client does not know the trapdoor information s (unless we are in the two-party model), the client has to use the bilinear map as follows. First, the client checks to see if $\alpha_1 = x$. Then, for $i = 2, \ldots, l$, the client verifies that the following relation holds:

$$e(\alpha_i, g) = e(\beta_{i-1}, g^{s+h(\alpha_{i-1})}) \,. \tag{16}$$

Note that the client can easily compute $g^{s+h(\alpha_{i-1})}$ (since g^s is public) and also that, by the bilinear mapping properties, we have

$$e(\beta_{i-1}, g^{s+h(\alpha_{i-1})}) = e(\beta_{i-1}^{s+h(\alpha_{i-1})}, g) \,,$$

and therefore verifying Equation 16 is equivalent with checking if $\beta_{i-1}^{s+h(\alpha_{i-1})} = a_i$ holds, which is exactly what we want. Also, the client verifies the *global* bilinear digest against the locally stored digest d, namely that the relation $e(d, g) = e(\beta_l, g^{s+h(\alpha_l)})$ holds. The client accepts only if all the relations above hold. As we prove later, the server can forge a proof for an element $y \notin S$ with negligible probability in the security parameter k.

Security. Concerning security, the public key pk in our scheme (see Definition 4) consists of the bilinear pairings parameters $\mathsf{t} = (p, \mathbb{G}, \mathbb{G}_M, e, g)$ and the elements of $\mathbb{G}_M$ $g, g^s, g^{s^2}, \ldots, g^{s^q}$ for some s chosen at random from $\mathbb{Z}_p^*$. The adversary behaves in the same way as in the description of the security of the RSA accumulator.

Theorem 11 *There exists a set-membership authentication scheme that combines the accumulation tree and the bilinear-map accumulator for authenticating a set of n elements that is secure under the q-strong Diffie-Hellman assumption and according to Definition 4.*

Proof: The security of the new scheme that uses the bilinear-map accumulator can be proved in the same way it was proved for the RSA accumulator (Theorem 3) by using, instead of Lemma 1, Lemma 2. $\square$

Complexity. In the static case, we do not have to compute the witnesses each time we query for an element. Namely, we can store the witnesses in the corresponding nodes of the tree and therefore reduce the query complexity from $O(n^\epsilon)$ to $O(1)$ (since the depth of the tree is constant). We can now present the main result of this section.

Theorem 12 *Let $0 < \epsilon < 1$ be a fixed constant. Under the q-strong Diffie-Hellman assumption, we can use the accumulation tree and the bilinear-map accumulator with precomputed witnesses to authenticate a static set S of n elements in the three-party model by storing a data structure of size $O(n)$ at both the source and the server such that:*

1. *Our scheme is secure according to Definition 4;*
2. *The expected query time is $O(1)$;*
3. *The size of the proof is $O(1)$;*
4. *The verification time is $O(1)$;*
5. *The client keeps space $O(1)$.*

Proof: Same as Theorem 4 with the difference that for the proof of security we use Theorem 11. $\square$

Note that this result applies also in the two-party model, with the difference that there is no need to use the bilinear map to do the verification (the client knows the value s). Moreover, in the proof of the two-party model, there is no need to communicate both the hash value and the element in $\mathbb{G}$, since the bilinear map function is not used. Finally, for verification, where we use the $e(\cdot, \cdot)$ function, we assume that the computation of $e(\cdot, \cdot)$ takes constant time (i.e., time that is does not depend on the number of elements in the hash table).

In the following we describe how to use our bilinear-map authentication structure that is based on the accumulation tree to authenticate a dynamic hash table. We first describe our general algorithms and protocols for the three-party model, and then extend our results to the two-party model, where as we will see, more work is required to get the same complexity results as in the three-party model.

4.4 Three-party model

Let $0 < \epsilon < 1$ be a fixed constant. We use the same separation technique as before and split the elements of the hash table into $O(n)$ buckets, each bucket containing $O(1)$ elements. The two differences we have identified between the RSA accumulator and the bilinear-map accumulator that can influence the complexity are as follows:

1. In the bilinear-map accumulator, one cannot compute witnesses on the fly with the straightforward method (i.e., in $O(n)$ time by a series of exponentiations). This is because the "on-the-fly" witnesses computation, which should be done by the untrusted server, requires knowledge of the parameter s, which is kept secret for the sake of security;

2. Witness updates in the bilinear-map accumulator can be done in $O(n)$ time (see Theorem 3), as opposed to Lemma 4, where we use an $O(n \log n)$ algorithm for the witness updates in the RSA accumulator.

We now present a useful lemma from [34], that is important in our solution.

Lemma 6 (Updating precomputed witnesses [34]) *Let $S = \{x_1, x_2, \ldots, x_n\}$ where $x_i \in \mathbb{Z}_p^*$. Let $V = g^{(x_1+s)(x_2+s)\ldots(x_n+s)}$ for some s and W_i be the respective witness of x_i. Then the following hold:*

1. If we add an element x_{n+1} to S, then for all $i = 1, \ldots, n+1$ we have that

$$W_i' = V W_i^{x_{n+1} - x_i} .$$

2. If we delete an element x_j from S, then for all $i \neq j$ we have that

$$W_i' = \left(\frac{W_i}{V'}\right)^{\frac{1}{x_j - x_i}} ,$$

where V' is the bilinear digest of the updated set.

We recall that in the above lemma it is $W_i = g^{\prod_{j \neq i}^{n}(x_j+s)}$. Using this lemma, we can derive the following corollary for the update time of the witnesses in the bilinear-map accumulator (since the computation of a witness takes constant time):

Corollary 3 (Witnesses updates in $O(n)$ time) *Suppose we are given the bilinear pairings parameters $\mathsf{t} = (p, \mathbb{G}, \mathbb{G}_M, e, g)$, the elements of $\mathbb{G}_M$ $g, g^s, g^{s^2}, \ldots, g^{s^q}$ for some s chosen at random from $\mathbb{Z}_p^*$ and our set $S = \{x_1, x_2, \ldots, x_n\}$ where $x_i \in \mathbb{Z}_p^*$, along with the witnesses W_i for all $i = 1, \ldots, n$. Let V be the bilinear digest of S, V' be the bilinear digest of S after an update has taken place (either insertion or deletion). Then, without the knowledge of s (and only by knowing V, V' and all the previous witnesses W_i), after an update, we can compute the new witnesses W_i' in $O(n)$ time.*

We now have the following result:

Theorem 13 *Let $0 < \epsilon < 1$ be a fixed constant. Under the q-strong Diffie-Hellman assumption, we can use the accumulation tree and the bilinear-map accumulator with precomputed witnesses to authenticate a dynamic hash table of n elements in the three-party model by storing a data structure of size $O(n)$ at both the source and the server such that:*

1. *Our scheme is secure according to Definition 4;*
2. *The expected amortized update time at the server is $O(n^\epsilon)$;*
3. *The expected amortized update time at the source is $O(1)$;*
4. *The expected query time is $O(1)$;*
5. *The size of the proof is $O(1)$;*
6. *The verification time is $O(1)$;*
7. *The client keeps space $O(1)$;*
8. *The update authentication information has size $O(1)$.*

Proof: The security of our scheme is derived by Theorem 11. Doing now an analysis that is exactly the same with the analysis in Lemma 5 where instead of the $O(n \log n)$ algorithm of Lemma 4 we use the $O(n)$ algorithm of Theorem 3 [1] and also we use a potential function that does not contain the logarithm factor, we can derive the main complexity result for the update time at the server. For the update time at the source, we take advantage of the fact that the source knows the trapdoor (which in this case is the value s) and therefore can efficiently update the digests. The other results are derived as in Theorem 5. Finally in order to implement non-membership queries, as in Theorem 5 we accumulate the cryptographic hashes $h(.)$ of intervals (y_i, y_{i+1}), i.e., if we have three elements in the bucket, $y_1 < y_2 < y_3$, the bilinear digest of the bucket that also supports non-membership proofs will be

$$g^{(h((-\infty,y_1))+s)(h((y_1,y_2))+s)(h((y_2,y_3))+s)(h((y_3,+\infty))+s)} .$$

This completes the proof. $\square$

4.5 Two-party model

As we saw in the previous section, in order to achieve efficient witnesses updates by using Theorem 3, one needs to know the updated bilinear digest after the update has taken place. This was easy in the three-party model, since the entity that computes the witnesses (i.e., the untrusted server) could receive the updated digests from the trusted source, without increasing the update authentication information. However, in the two party model, the untrusted server has to perform this himself, without knowing the trapdoor information s. In the following we show how one can do that in $O(n)$ time, by using Viète's formulas [43].

Suppose we have a set of elements $S = \{x_1, x_2, \ldots, x_n\}$ and the respective bilinear digest $\psi(S) = g^{\prod_{i=1}^{n}(x_i+s)}$ for some s. We recall that the server has the elements $g, g^s, g^{s^2}, \ldots, g^{s^q}$, where $q \geq n$ is an upper bound to the number of elements that are going to be accumulated. Note now that the exponent of the bilinear digest is a n-degree polynomial in s and therefore can be written

[1] Note that one important thing to achieve the $O(n)$ witness updates without knowing s is that someone **needs** to know the new (updated) bilinear digest in order to use it in the formulas. As we will see later, this can be computed in $O(n)$ time. However, this is not needed for the three-party model, since the source, that knows s can do that in constant time. Therefore, and referring back to the accumulation tree, whenever there is an update, the source sends to the server the updated bilinear digests along the path of the update. Then the server, knowing the previous bilinear digests and the previous witnesses at every node, can use Lemma 6 to compute the new witnesses.

as $b_n s^n + b_{n-1}s^{n-1} + \ldots + b_1 s + b_0$ where $b_n = 1$ and thus the bilinear digest can be expressed as follows:

$$\psi(S) = g^{\prod_{i=1}^{n}(x_i+s)} = g^{b_0} \times (g^s)^{b_1} \times \left(g^{s^2}\right)^{b_2} \times \ldots \times \left(g^{s^{n-1}}\right)^{b_{n-1}} \times g^{s^n}. \tag{17}$$

Therefore, one way to compute the bilinear digest is by using b_i and g^{s^i}. In this way, we can still compute the digest without the knowledge of s. From Viète's formulas [43], we know that

$$b_{n-k} = \sum_{i_1,i_2,\ldots,i_k : i_1<i_2<\ldots<i_k} x_{i_1}x_{i_2}\ldots x_{i_k}.$$

Suppose now we are adding an element x_{n+1} to our set. The new coefficients of the polynomial will be $a_{n+1}, a_n, \ldots, a_0$ and they can be computed in $O(n)$ time (therefore we do not have to do it from scratch which would very expensive) by using the previous coefficients in the iterative relation $a_i = b_{i-1} + x_{n+1}b_i$ for $i = 1, \ldots, n$. Note that $a_0 = \prod_{j=1}^{n+1} x_j$ and $a_{n+1} = 1$. Similarly when you delete an element x_j one can compute the coefficients of the new polynomial $b_0, b_1, \ldots, b_n$ (from the coefficients $a_0, a_1, \ldots, a_{n+1}$) by setting $b_i = x_j^{-1}(a_i - b_{i-1})$ for $i = 1, \ldots, n$, where $b_0 = \prod_{j=1}^{n} x_j$, which again is an $O(n)$ computation. Finally, note that all the computations that refer to the exponent are reduced modulo p, the order of the multiplicative cyclic group.

Therefore in the two-party model, whenever there is an update, the server updates the coefficients of the polynomial ($O(n)$ time), then computes the new digest by using Equation 17 ($O(n)$) and then updates the witnesses by using Theorem 3 ($O(n)$). When using the accumulation tree however, all these operations have to be performed $1/\epsilon$ times on $O(n^\epsilon)$ sets. Also the server has to store the n coefficients of the current polynomial which however does not increase the asymptotic space needed. Therefore we have the following result:

Theorem 14 *Let $0 < \epsilon < 1$ be a fixed constant. Under the q-strong Diffie-Hellman assumption, we can use the accumulation tree and the bilinear-map accumulator with precomputed witnesses to authenticate a dynamic hash table of n elements in the two-party model by storing a data structure of size $O(n)$ such that:*

1. *Our scheme is secure according to Definition 4;*
2. *The expected amortized update time at the server is $O(n^\epsilon)$;*
3. *The amortized update time at the client is $O(1)$;*
4. *The query time is $O(1)$;*
5. *The size of the verification proof is $O(1)$;*
6. *The amortized size of the consistency proof is $O(1)$;*
7. *The verification time is $O(1)$;*
8. *The client keeps amortized space $O(1)$.*

Finally we note that all the algorithms for the update at the client side are the same with those described in Section 3.6. However, one important difference with the three party model is the fact that the verification can be done without the use of $e(.,.)$ function, since the client knows the value s (which also enables him to do very efficient updates, i.e., in $O(1)$ time). Also we note that the server should not be holding all the elements $g, g^s, g^{s^2}, \ldots, g^{s^q}$ as this can be very space inefficient. He can receive the appropriate values from the client whenever there is an insertion (i.e., when the set size increases from q to $q+1$ the client, along with the update query, also sends $g^{s^{q+1}}$) and he can delete $g^{s^{q+1}}$ when the set size drops from $q+1$ to q.

5 Analysis and evaluation

In this section we provide an evaluation of our two authenticated hash table structures. 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 three-party—and most complete and representative—version of our schemes, namely the authenticated hash tables described in Theorems 5 and 13, 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, 2.8GHz 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 [1], 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 [2], a library for pairing-based cryptography, interfaced with C. Finally, for both schemes, we used the efficient *sparsehash* hash table implementation from Google (`http://code.google.com/p/google-sparsehash/`) for on-the-fly computation and efficient updates of the witnesses during a query or an update respectively.

5.1 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 Section 3.7. That is, we restrict the input of each level of accumulation to be two times 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 $t = 9$ 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 $2 \times 256 + 9 = 521$ bits. Note that $t = 9$ is the smallest value satisfying Theorem 9 for $b = 256$.

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 1024-bit integers.

We have benchmarked the time needed for these operations. For 200 runs, the average time for computing the power of a 1024-bit number to a 521-bit exponent and then reducing the result modulo N was found to be $T_1 = 3.04$ms, and the average time for computing the inverse of a 521-bit number modulo $\phi(N)$ was $T_2 = 0.000105$ms. Similarly, multiplication of 521-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 1024-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 1024-bit integer x and outputs 521-bit prime, as in Theorem 9. 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 the proof of Theorem 6 the updates are performed by the source as follows. Suppose the source wants 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). The source first computes $d'_1 = d_1^{f(x)^{-1}} \mod N$ which is the new value of the bucket. Note that this is feasible to compute, since the source knows $\phi(N)$. Therefore so far, the source has performed one $f(\cdot)$ computation (actually the source has 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 $i = 2, \ldots, l$, the source computes 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, the source has to do one $f(\cdot)$ computation, one inverse computation, one multiplication and one exponentiation. Therefore, the total update time at the source is

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

which is not dependent on n. During an update the server has 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, after the server has computed the new witnesses for each internal node of the accumulation tree, these witnesses have to be stored in a hash table. Therefore,

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

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 5 becomes $\alpha_i = f(\beta_{i-1}^{\alpha_{i-1}} \mod N)$. This can be checked by cutting the last 9 bits of α_i and comparing the result (from which we also cut the last 256 0's) with the SHA-256 digest of $\beta_{i-1}^{\alpha_{i-1}} \mod N$. Finally, the client has to perform one signature verification (i.e., to verify the signed digest from the source). Therefore,

$$t_{\mathsf{verify}} = (\epsilon^{-1} + 1)(t_1 + t_4) + t_1\,, \tag{18}$$

which is also not dependent on n.

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

$$t_{\mathsf{query}} = (\epsilon^{-1} + 1) t_{\mathsf{table}}\,. \tag{19}$$

Communication complexity. The proof and the update authentication information consist of $\epsilon^{-1} + 1$ pairs of 1024-bit numbers and 521-bit $f(\cdot)$ values plus the signature of the digest from the source. Thus,

$$s_{\mathsf{proof}} = (\epsilon^{-1} + 1)(1024 + 521) + 1024\,.$$

Finally, the update authentication information consists only of the bilinear digests that lie in the update path plus a signature, and therefore, its size is $(\epsilon^{-1}+1)1024+1024$ bits.

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 2.25KB have to be communicated.

Table 2: Cost expressions in our RSA-accumulator scheme for $n = 100,000,000$ and various values of ϵ. The size n of the hash table influences only the server's update time.

operation	**cost expression**	$\epsilon = 0.1$	$\epsilon = 0.3$	$\epsilon = 0.5$
source update (ms)	$T_1 + T_2 + T_4 + \epsilon^{-1}(T_1 + T_2 + T_3 + T_4)$	33.56	13.22	9.15
server update (ms)	$(\epsilon^{-1}+1)(n^\epsilon \log n^\epsilon T_1 + T_4) + (\epsilon^{-1}n^\epsilon + 1)t_{\mathsf{table}}$	184.75	8680.60	398690.00
verify (ms)	$(\epsilon^{-1}+1)(T_1+T_4)+T_1$	36.59	16.22	12.19
query (ms)	$(\epsilon^{-1}+1)t_{\mathsf{table}}$	0.03	0.01	0.01
proof size (KB)	$(\epsilon^{-1}+1)(1024+521)+1024$	2.25	0.97	0.70
update info (KB)	$(\epsilon^{-1}+1)1024+1024$	1.53	0.68	0.51

5.2 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 [28]. 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. 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 1024 bits. The main operations we benchmarked using PBC are the following:[2]

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;

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

4. SHA-256 computation of 1024-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.

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.

Table 3: Cost expressions in our scheme bilinear-accumulator scheme for $n = 100,000,000$ and various values of ϵ. The size n of the hash table influences only the server's update time.

operation	**cost expression**	$\epsilon = 0.1$	$\epsilon = 0.3$	$\epsilon = 0.5$
source update (ms)	$t_1 + t_2 + t_4 + \epsilon^{-1}(t_1 + t_2 + t_3 + t_4)$	150.82	59.41	41.13
server update (ms)	$(\epsilon^{-1} + 1)[n^\epsilon(t_1 + t_2 + t_3) + t_4] + (\epsilon^{-1} n^\epsilon + 1)t_{\mathsf{table}}$	951.20	14915.00	411080.00
verify (ms)	$(\epsilon^{-1} + 1)(t_1 + t_4 + 2t_6) + T_1$	441.61	175.81	122.65
query (ms)	$(\epsilon^{-1} + 1)t_{\mathsf{table}}$	0.03	0.01	0.01
proof size (KB)	$(\epsilon^{-1} + 1)(1024 + 1024) + 1024$	2.94	1.24	0.89
update info (KB)	$(\epsilon^{-1} + 1)1024 + 1024$	1.53	0.68	0.51

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 field $\mathbb{G}$. However, asymptotically, the bilinear-accumulator scheme outperforms the RSA-accumulator scheme by a logarithmic factor. In terms of communication efficiency, we see that there is almost no difference since the size of the elements of the field $\mathbb{G}$ is 1024 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. In Figure 2, we can see how the update time scales with increasing number of elements in the hash table, for both authentication schemes. Here, we observe that for $\epsilon = 0.1$, the RSA-accumulator scheme is far more efficient than the bilinear-accumulator 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 [18]). 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.

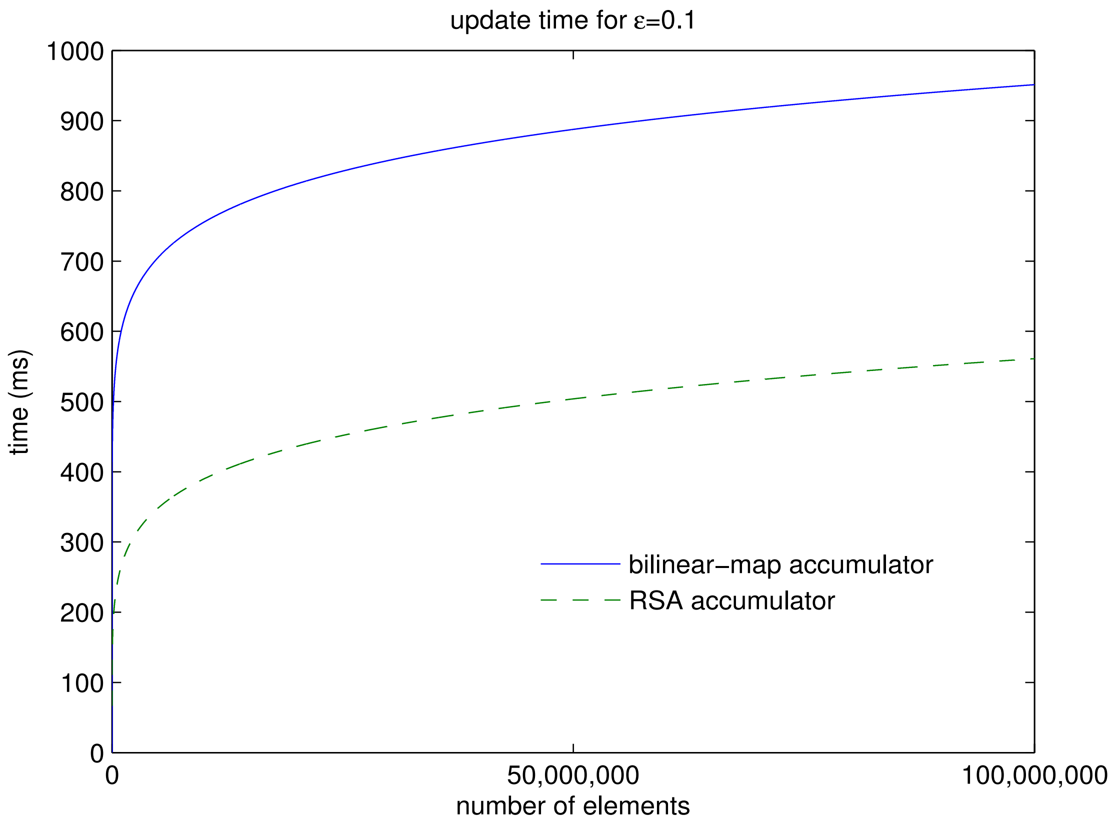

Figure 2: Comparison of update time for $\epsilon = 0.1$.

6 Conclusions and future work

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 and *sublinear* update costs. Our results are applicable to both the two-party and three-party data authentication models. 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 unfeasible for the two-party model due to the $\Omega(\log n/\log\log n)$ memory checking lower bound [15] on query complexity (the sum of read and write complexity). This result, however, motivates seeking better lower bounds for set-membership authentication (as, e.g., in [15, 41]): given a cryptographic primitive or authentication model, what is the best we can do in terms of complexity (and still being provable secure)?

Finally, it would be interesting to modify our schemes to obtain non-amortized bounds for updates using e.g., Overmar's global rebuilding technique [36].

Acknowledgments

This research was supported by the U.S. National Science Foundation under grants IIS–0713403 and OCI–0724806, by the Center for Geometric Computing and the Kanellakis Fellowship at Brown University, 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 Michael Goodrich, Anna Lysyanskaya, John Savage and Ioannis Vergados for many useful discussions, and C. Chris Erway for providing pointers to the NTL library. We also thank Ivan Damgård for giving us important feedback on the first version of this work.

References

[1] NTL: A library for doing number theory. `http://www.shoup.net/ntl/`.

[2] PBC: The pairing-based cryptography library. `http://crypto.stanford.edu/pbc/`.

[3] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *Proc. Computer and Communication Security (CCS)*, 2007.

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

[5] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, pages 62–73. ACM Press, 1993.

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

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

[8] D. Boneh and X. Boyen. Short signatures without random oracles and the SDH assumption in bilinear groups. *J. Cryptology*, 21(2):149–177, 2008.

[9] A. Buldas, P. Laud, and H. Lipmaa. Accountable certificate management using undeniable attestations. In *ACM Conference on Computer and Communications Security*, pages 9–18. ACM Press, 2000.

[10] J. Camenisch, M. Kohlweiss, and C. Soriente. An accumulator based on bilinear maps and efficient revocation for anonymous credentials. In *Public Key Cryptography (PKC)*, pages 481–500, 2009.

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

[12] I. L. Carter and M. N. Wegman. Universal classes of hash functions. In *Proc. ACM Symp. on Theory of Computing*, pages 106–112, 1977.

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

[14] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: upper and lower bounds. *SIAM J. Comput.*, 23:738–761, 1994.

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

[16] C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia. Dynamic provable data possession. In *Proc. ACM Int. Conference on Computer and Communications Security (CCS)*, 2009.

[17] R. Gennaro, S. Halevi, and T. Rabin. Secure hash-and-sign signatures without the random oracle. In *Proc. EUROCRYPT*, volume 1592 of *LNCS*, pages 123–139. Springer-Verlag, 1999.

[18] M. T. Goodrich, C. Papamanthou, and R. Tamassia. On the cost of persistence and authentication in skip lists. In *Proc. Int. Workshop on Experimental Algorithms (WEA)*, pages 94–107, 2007.

[19] M. T. Goodrich, C. Papamanthou, R. Tamassia, and N. Triandopoulos. Athos: Efficient authentication of outsourced file systems. In *Proc. Information Security Conference*, LNCS, pages 80–96. Springer, 2008.

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

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

[22] M. T. Goodrich, R. Tamassia, and N. Triandopoulos. Super-efficient verification of dynamic outsourced databases. In *Proc. RSA Conference, Cryptographers' Track (CT-RSA)*, volume 4964 of *LNCS*, pages 407–424. Springer, 2008.

[23] M. T. Goodrich, R. Tamassia, N. Triandopoulos, and R. Cohen. Authenticated data structures for graph and geometric searching. In *Proc. RSA Conference—Cryptographers'Track*, pages 295–313. Springer, LNCS 2612, 2003.

[24] A. Hutflesz, H.-W. Six, and P. Widmayer. Globally order preserving multidimensional linear hashing. In *Proc. 4th Intl. Conf. on Data Engineering*, pages 572–579, 1988.

[25] C. M. Kenyon and J. S. Vitter. Maximum queue size and hashing with lazy deletion. *Algorithmica*, 6:597–619, 1991.

[26] J. Li, N. Li, and R. Xue. Universal accumulators with efficient nonmembership proofs. In *ACNS*, pages 253–269, 2007.

[27] N. Linial and O. Sasson. Non-expansive hashing. In *Proc. 28th Annu. ACM Sympos. Theory Comput.*, pages 509–517, 1996.

[28] B. Lynn. *On the implementation of pairing-based cryptosystems*. PhD thesis, Stanford University, November 2008.

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

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

[31] J. K. Mullin. Spiral storage: Efficient dynamic hashing with constant-performance. *Comput. J.*, 28:330–334, 1985.

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

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

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

[35] G. Nuckolls. Verified query results from hybrid authentication trees. In *DBSec*, pages 84–98, 2005.

[36] M. H. Overmars. *The Design of Dynamic Data Structures*, volume 156 of *Lecture Notes Comput. Sci.* Springer-Verlag, Heidelberg, West Germany, 1983.

[37] C. Papamanthou and R. Tamassia. Time and space efficient algorithms for two-party authenticated data structures. In *Proc. Int. Conference on Information and Communications Security (ICICS)*, volume 4861 of *LNCS*, pages 1–15. Springer, 2007.

[38] T. Sander. Efficient accumulators without trapdoor (extended abstract). In *ICICS '99: Proc. Int. Conf. on Information and Communication Security*, pages 252–262. Springer-Verlag, 1999.

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

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

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

[42] R. Tamassia and N. Triandopoulos. Efficient content authentication in peer-to-peer networks. In *Proc. Int. Conf. on Applied Cryptography and Network Security (ACNS)*, volume 4521 of *LNCS*, pages 354–372. Springer, 2007.

[43] E. B. Vinberg. *A course in algebra.* American Mathematical Society, Providence RI, 2003.

[44] P. Wang, H. Wang, and J. Pieprzyk. A new dynamic accumulator for batch updates. In *ICICS*, pages 98–112, 2007.