Authenticated Hash Tables

Charalampos Papamanthou
Dept. of Computer Science
Brown University
Providence RI, USA

Roberto Tamassia
Dept. of Computer Science
Brown University
Providence RI, USA

Nikos Triandopoulos
Dept. of Computer Science
University of Aarhus
Aarhus, Denmark

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 so that the client can save space or achieve load balancing. 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 membership queries on hash tables: for any fixed constants $0 < \epsilon < 1$ and $\kappa > 1/\epsilon$, the server can provide a proof of integrity of the answer to a (non-)membership query in *constant* time, requiring $O\left(n^{\epsilon}/\log^{\kappa\epsilon-1} n\right)$ time to treat updates, yet keeping the communication and verification costs constant. This is the first construction for authenticating a hash table with *constant query* cost and *sublinear update* cost. Our solution employs the RSA accumulator in a nested way over the stored data, strictly improving upon previous accumulator-based solutions. Our construction applies to two concrete data authentication models and lends itself to a scheme that achieves different trade-offs—namely, constant update time and $O(n^{\epsilon}/\log^{\kappa\epsilon} n)$ query time for fixed $\epsilon > 0$ and $\kappa > 0$. An experimental evaluation of our solution shows very good scalability.

Categories and Subject Descriptors

C.2.4 [**Communication Networks**]: Distributed Systems; E.1 [**Data Structures**]; H.3.4 [**Information Storage and Retrieval**]: Systems and Software

General Terms

Algorithms, Security, Theory, Verification

Keywords

Hash tables, Authentication, Verification, RSA accumulator

CCS'08 October 27–31, 2008, Alexandria, Virginia, USA
Copyright 2008 ACM 978-1-59593-810-7/08/10 ...$5.00.

1. INTRODUCTION

Recently, there has been an increasing interest in *remote storage* of information and data, where people outsource their personal files at service providers that offer huge storage space and fast network connections (e.g., Amazon S3). In this way, clients create *virtual* hard drives consisting of online storage units that are operated by remote and geographically dispersed servers. In addition to a convenient solution to space-shortage, data-archiving or back-up issues, 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 such settings, the ability to check the integrity of remotely stored data is an important security property, or otherwise a malicious server can easily tamper with a client's data (e.g., delete or modify a file). Also, without some kind of verification mechanisms, this attack can never be detected, no matter what trust relations the client and the server may a priori share. Thus, it is always desirable that operations on remote data be *authenticated*. In particular, 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 the simplest, but very important, form of this data authentication problem, where we wish to authenticate *(non-)membership queries* over a *dynamic set* of n data elements that is stored at an untrusted server with the use of a *hash table*. Used widely by numerous applications, hash tables are fundamental 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., few bytes long) 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 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 information communicated (after an update—insertion in/deletion from the set) from the data source to the server, and also the size of the proof sent (after a client's query) from the server to the client, *ideally keeping both complexities constant.* Analogously, since client-side applications usually connect to the servers from mobile devices with limited computing power and slow connectivity (e.g., cell-phones), we wish that all computations (due to verifications) performed at the clients are as efficient as possible, *ideally constant.* 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 [25]. Using cryptographic (collision resistant) hashing and Merkle's tree construction [23] to produce the set's digest, (non-)membership queries in sets can be authenticated with *logarithmic costs* (e.g., [4, 15, 25, 28, 34]), which is optimal for any hash-based approach, as it has been shown in [33]. Breaking this logarithmic barrier, therefore, requires employing an alternative cryptographic primitive. One-way accumulators and their dynamic extensions [2, 3, 6, 26] 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 costs ($O(n)$ or $O(n \log n)$ depending on the accumulator), thus incurring high update cost at the server.

In our main result we show how to use the RSA accumulator (e.g., [6]) 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 scheme 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 the security of the RSA accumulator, which holds under the, currently standard, *strong RSA assumption.*

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* [25, 32] 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 the data set by querying one of the servers. Along with the answer to a query, the 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 [4]. For a detailed description of the models we refer to [13, 15, 28].

Finally, to meet the needs of different data-access patterns, we extend our three-party authentication scheme to achieve 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 assumptions, 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 the RSA accumulator in a nested way over a tree of constant depth. We formally prove the security of the new scheme based only on the widely accepted and used strong RSA assumption.

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 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. Namely, we reduce the query time and the size of the update authentication information from $O(n^\epsilon)$ [14] (at present, the best known upper bound for authenticating set-membership queries using RSA accumulators) to $O(1)$, and also the update time from $O(n^\epsilon)$ to $O\left(n^\epsilon / \log^{\kappa\epsilon - 1} n\right)$ for any fixed constants $\kappa > 0$, $0 < \epsilon < 1$ such that $\kappa\epsilon > 1$. This answers an open problem posed in [25]. Also, we extend our scheme to get a different trade-off between query and update costs, namely constant update time with $O(n^\epsilon / \log^{\kappa\epsilon} n)$ query time for constants $0 < \epsilon < 1$ and $\kappa > 0$ (see Table 1).

Table 1: Comparison of existing schemes for authenticating set-membership queries in a set of size n w.r.t. used techniques and various complexity measures. Here, $0 < \epsilon < 1$ and $\kappa > 1/\epsilon$ are fixed constants, NA stands for "not applicable", DH for "Diffie-Hellman", exp for "exponentiation" and BM for "bilinear mapping". All complexity measures refer to n (not to the security parameter) and are asymptotic expected values. Our main scheme maintains $O(1)$ communication, verification and query costs, still achieving sublinear update cost ($O\left(n^{\epsilon}/\log^{\kappa\epsilon-1} n\right)$), thus outperforming existing accumulation-based schemes; it also uses a standard cryptographic assumption. Confined to the 3-party model, our extended scheme achieves a different trade-off between update and query costs. Update costs in our schemes are amortized expected 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**	**update**	**query**	**update**	**verify**	**crypto**
			size	**info.**	**time**	**time**	**time**	**oper.**
[4, 15, 22, 25, 28]	both	collision resistance	$\log n$	1	$\log n$	$\log n$	$\log n$	hashing
[1]	2-party	strong RSA	1	**NA**	1	**NA**	1	exp
[6, 30]	both	strong RSA	1	1	1	$n \log n$	1	exp
[26]	both	strong DH	1	1	1	n	1	exp, BM
[14]	3-party	strong RSA	1	n^{ϵ}	n^{ϵ}	n^{ϵ}	1	exp
main scheme	both	strong RSA	1	1	1	$n^{\epsilon}/\log^{\kappa\epsilon-1} n$	1	exp
extension	3-party	strong RSA	1	1	$n^{\epsilon}/\log^{\kappa\epsilon} n$	1	1	exp

4. We give a practical evaluation of our scheme using state of the art software [31] for primitive operations (namely, modular exponentiations, multiplications, inverse computations). We show that for $\kappa = 1$ and $\epsilon = 0.1$ our scheme scales very well.

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., [4, 15, 22, 25, 28]), 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. These bounds are optimal for hash-based methods, as was shown in [33]. Variations of this approach and extensions to other types of queries have also been investigated (e.g., [5, 12, 17, 34]).

Solutions for authenticated membership queries in various settings using another cryptographic primitive, namely *one-way accumulators*, were introduced by Benaloh and de Mare [3]. Based on the RSA exponentiation function, this scheme implements a secure one-way function that satisfies *quasi-commutativity*, a useful property that usual hash functions lack. This RSA accumulator is used to summarize a set so that set-membership can be verified with $O(1)$ overhead. Refinements of the RSA accumulator are also given in [2], where except for one-wayness, collision resistance is achieved, and also in [11, 30]. Dynamic accumulators (along with protocols for zero-knowledge proofs) were introduced in [6], 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 applicable only to the three-party model, was made in [14]; 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 [14] is presented in [16], and a similar hybrid authentication scheme appears in [27].

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 [26]. However, updates in [26] 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. Efficient dynamic accumulators for non-membership proofs are presented in [20]. Accumulators for batch updates are presented in [35] and accumulator-like expressions to authenticate static sets for *provable data possession* are presented in [1]. The work in [29] studies efficient algorithms for accumulators with unknown trapdoor information. Finally in [10] and simultaneously with our work, logarithmic lower bounds as well as constructions achieving query-update cost tradeoffs that are similar to our work, have been studied in the memory-checking model [4].

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 scheme. In Section 3, we develop our main authentication construction and prove its security, providing a solution for static sets. In Section 4, we apply our construction to hash tables and derive our main results. In Section 5, we provide an evaluation and analysis of our method showing its 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 methods and other concepts used in our approach.

Hash Tables. The main functionality of the hash table data structure is the support of super efficient look-ups in $O(1)$ time of elements that belong to a general set (not necessarily ordered). Different ways to implement the hash table data structure have been extensively studied (e.g., [9, 18, 19, 21, 24]), where the basic idea behind them is the following. 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. However, we can do better: Set up a one-dimensional table $T[1 \ldots m]$ where $m = O(n)$, fix a special function $h : \mathcal{U} \rightarrow \{1, \ldots, m\}$, such that for any two elements $e_1, e_2 \in \mathcal{U}$ it is $\Pr[h(e_1) = h(e_2)] \leq \frac{1}{m}$, and store element e in slot $T(h(e))$. The probabilistic property that holds for h, combined with the fact that h can be computed in $O(1)$ time, leads to the conclusion that there is always a constant expected number of elements that map to the same slot i $(1 \leq i \leq m)$ and, therefore, look-up takes constant expected time.

But this nice property of hash tables comes at some cost. The expected constant-time look-up holds only when the number of elements stored in the hash table does not change, i.e., when the hash table is static. When there are insertions and deletions, it is obvious that we might end up in a situation where there are more than $O(1)$ elements stored in some slot of the hash table, which increases the time needed for queries. To maintain slots that few elements map to, we might have to increase the size of the hash table by a constant factor (e.g., double the size), which is expensive since we have to rehash all the elements of the hash table by using another hash function. Therefore, there might be one update (over a course of $O(n)$ updates) that takes time $O(n)$ and not $O(1)$. That leads to the fact that hash tables support expected $O(1)$ time queries, but updates in $O(1)$ 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*, satisfying:

Theorem 1 (Dynamic Hashing [8]). *For a set of size n, dynamic hashing can be implemented to use $O(n)$ space and have $O(1)$ expected query cost for membership queries and $O(1)$ expected amortized cost for insertions or deletions.*

Note that by choosing a suitable function that distributes n elements into *buckets* (i.e., array slots), we can have $f_1(n)$ buckets and maintain the expected *bucket size* (i.e., elements within the bucket) to be $f_2(n)$, for general $f_1(\cdot), f_2(\cdot)$ such that $f_1(n) \cdot f_2(n) = O(n)$. Therefore we have the following corollary:

Corollary 1. *Let $f_1(\cdot), f_2(\cdot)$ be functions such that for all n it is $f_1(n) \cdot f_2(n) = O(n)$. Dynamic hashing can be implemented to use $O(n)$ space and have $O(f_2(n))$ expected query cost for membership queries, $O(f_2(n))$ expected amortized cost for insertions and deletions, and $O(f_1(n))$ buckets.*

We use the above result in our schemes, where $f_1(n) = O(n/\log^{\kappa} n)$ and $f_2(n) = O(\log^{\kappa} n)$ for some fixed $\kappa > 0$.

Prime Representatives. In our construction we extensively use the notion of *prime representatives*, which were initially introduced in [2] and provide a solution whenever it is necessary to map general elements to prime numbers—in our setting, for security reasons. In particular, a method for mapping a k-bit element e_i to a $3k$-bit prime x_i is to use *two-universal hash functions*, as introduced by Carter and Wegman [7]. A family $H = \{h : A \rightarrow B\}$ of functions is two-universal if, for all $w_1 \neq w_2$ and for a randomly chosen function h from H, it is $\Pr[h(w_1) = h(w_2)] \leq 1/|B|$. In our context, set A consists of $3k$-bit boolean vectors and set B consists 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 solution, 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 as follows:

Lemma 1 (Prime Representatives [11, 14]). *Let H be a two-universal family of functions from $\{0,1\}^{3k}$ to $\{0,1\}^k$ and $h \in H$. For any element $e_i \in \{0,1\}^k$, with high probability, we can compute a prime $x_i \in \{0,1\}^{3k}$ so that $h(x_i) = e_i$, by sampling $O(k^2)$ times from the set of inverses $h^{-1}(e_i)$.*

This means that we can compute prime representatives in expected constant time, since the dimension of our problem is the number of the elements in the hash table, n. 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, note that prime representatives are computed (and stored) only once, since computing multiple times a prime representative of the same element does not output the same prime, 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.

Cryptographic Accumulators. If k denotes the security parameter, we first give the definition of *negligible* functions.

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

One basic cryptographic primitive that we use in our approach is the *cryptographic accumulator* [2, 3, 6, 26], which provides an efficient technique to produce a short (computational) proof that a certain element is a member of a certain set. We use the RSA accumulator which 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 [6]. We can efficiently represent $\mathcal{E}$ with a k'-bit integer, namely the integer $f(\mathcal{E}) = g^{r(e_1)r(e_2)\ldots r(e_n)} \mod N$, where $g \in QR_N$ [6] and $r(e_i)$ is a $3k$-bit prime representative. This representation has the property that any computational bounded adversary $\mathcal{A}$, that 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* [2], which is stated as follows:

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

We now present a useful lemma (we defer the proof to the full version of the paper).

Lemma 2. *Let k be the security parameter, h be a two-universal hash function that maps $3w$-bit primes to w-bit integers and N be a $(w+1)$-bit RSA modulus with $w = \Theta(k)$. 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 set $\mathcal{E}' \neq \mathcal{E}$ such that $f(\mathcal{E}') = f(\mathcal{E})$ is $\mathsf{neg}(k)$.*

The following result is a consequence of Lemma 2:

Corollary 2. *Let k be the security parameter, h be a two-universal hash function mapping $3w$-bit primes to w-bit integers and N be a $(w+1)$-bit RSA modulus with $w = \Theta(k)$. 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 $\mathsf{neg}(k)$.*

Note that we need to use prime representatives in order to avoid collisions, as observed in [2]. We now continue with the security definition of a membership authentication scheme, which captures the security requirements of the hash table authenticated data structure.

Security. Suppose S is a set we wish to authenticate membership in (essentially to authenticate the correctness of queries of type "does x belong to S?") and let pk be the public key. The adversary is given oracle access to all the algorithms for updating and querying S and also for verifying answers. 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. $\Pi(x) \leftarrow \mathsf{query}(x)$, where $\Pi(x)$ is the proof returned to a query for an element x;

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.

The formal security definition for a membership authentication scheme of a set S is as follows:

Definition 3 (Security). *Suppose k is the security parameter and $\mathcal{A}$ is a computationally bounded adversary that is given the public key pk. Our set S initially is empty and $S = S_0$. The adversary $\mathcal{A}$ chooses and issues an update $\mathsf{upd}_i \in \{\mathsf{ins}(x_i), \mathsf{del}(x_i)\}$ in the set S_i for $i = 0, \ldots, t$ and therefore computes $\{S_{i+1}, d_{i+1}\} \leftarrow \mathsf{update}(\mathsf{upd}_i, S_i)$ where d_0 is defined to be the state (digest) of an empty set and t is polynomially dependent on the security parameter k. After that stage (update stage) the adversary has produced S_{t+1} and d_{t+1}. Then he enters the attack stage where he chooses an element $y \notin S_{t+1}$ and computes a proof $\Pi(y)$. We say that the authentication scheme for set membership is secure if the probability that $\mathsf{accept} \leftarrow \mathsf{verify}(y, \Pi(y), d_{t+1})$ is $\mathsf{neg}(k)$.*

Note that the above security definition captures the notion of an adversary that tries to forge proofs for elements that do not belong to the existing set. Also we provide the adversary with the flexibility to do his own updates and choose his own elements to forge. Note that this security definition is applicable to both models (two-party/three-party).

Finally, we describe the setup phase of our schemes, where we will see why $w = \Theta(k)$ (in Lemma 2 and Corollary 2).

System Setup. Let k be the security parameter. In the two-party model, the client initially picks constants $0 < \epsilon < 1$ and $\kappa > 1/\epsilon$, and also picks $l = \lceil 1/\epsilon \rceil$ RSA moduli $N_i = p_i q_i$ $(i = 1, \ldots, l)$, where p_i, q_i are strong primes [6]. The length of the RSA modulus N_i, $|N_i|$, is defined by the recursive relation $|N_{i+1}| = 3|N_i| + 1$, where $|N_1| = 3k + 1$ and $i = 1, \ldots, l-1$. Note that since l is constant all the RSA moduli have asymptotically the same dependence on the security parameter k. The client reveals N_i $(i = 1, \ldots, l)$ to the untrusted server but keeps $\phi(N_i) = (p_i - 1)(q_i - 1)$ secret. The client also picks l public bases $g_i \in QR_{N_i}$ to be used for exponentiation. Finally, given l families of two-universal hash functions $H_1, H_2, \ldots, H_l$, the client randomly picks one function $h_i \in H_i$ and reveals h_i to the server who will be using that function to compute multiple prime representatives. The function h_i is such that it maps $(|N_i| - 1)$-bit primes to $((|N_i| - 1)/3)$-bit integers. For the three-party model the setup is exactly the same with the difference that the source now knows everything (and picks the RSA moduli and two-universal hash functions himself) that the untrusted servers know and also $\phi(N_i)$. Note that it is very important not to reveal $\phi(N_i)$ to the untrusted entities, since if we do, the security of the whole system collapses, as they would be able in this way to compute inverses and forge proofs. Note that since $1/\epsilon$ is constant, the client needs constant space.

The choice of the domains and ranges of functions h_i and of the lengths of moduli N_i is due to the fact that the prime representatives that we use should be less than the respective moduli (see [30]). This will be clear in the description of our main construction in Section 3. Finally note that, using ideas from [2], it is possible to avoid the increasing size of the 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 (see Section 5).

3. BASE CONSTRUCTION

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.

3.1 The RSA Tree

Let $0 < \epsilon < 1$ be a constant and $S = \{e_1, e_2, \ldots, e_n\}$ be the set of elements we would like to authenticate. We build a tree $T(\epsilon)$ (the tree is a function of ϵ) on top of S, which we call the *RSA tree* of S, such that:

1. The leaves of the tree $T(\epsilon)$ store the actual elements $e_1, e_2, \ldots, e_n$;

2. $T(\epsilon)$ consists of exactly $\lceil \frac{1}{\epsilon} \rceil + 1$ levels;

3. Every node of $T(\epsilon)$ has $O(n^\epsilon)$ children;

4. Level i in the tree contains $O(n^{1-i\epsilon})$ nodes (the root node of the tree lies in the maximum level).

The RSA tree, whose structure for a set of 64 elements is shown in Figure 1, can be viewed as a normal search tree that resembles the B-tree data structure. However there are some differences: first, every internal node of the RSA 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 ($O\left(\frac{1}{\epsilon}\right)$).

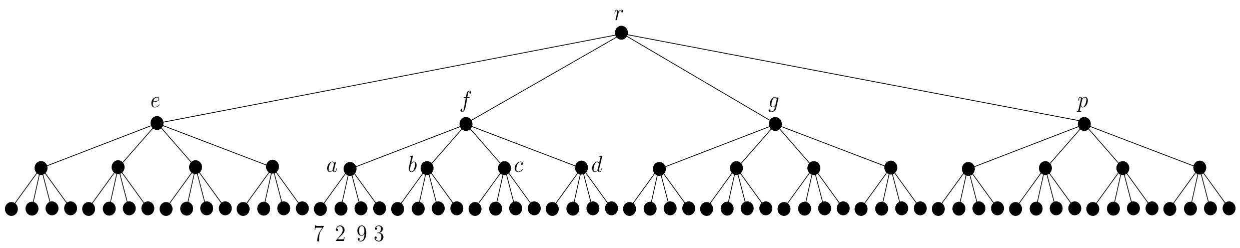

Figure 1: The RSA tree of a set consisting of 64 elements. Here, $\epsilon = \frac{1}{3}$. Therefore, every internal node has 4 children, there are $4 = \frac{1}{\epsilon} + 1$ levels and at level $i = 0, 1, 2, 3$ there are $64^{1-i/3}$ nodes respectively.

It easy to see that, with such a tree, we can support queries in $O(n^\epsilon)$ time. Also, updates take $O(n^\epsilon)$ *amortized* time since, as we are not allowed to vary the tree's depth, we periodically (e.g., when the number of elements of the tree doubles) rebuild the tree spending $O(n)$ time. Note that, by using binary trees to index the elements within each internal node of the RSA tree, queries can be answered in $O(\log n)$ time and updates can 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.

Therefore, by hierarchically employing the RSA accumulator over set S, we augment the RSA tree with a collection of accumulation values. In particular, for any tree node v we define an *RSA digest* $\chi(v)$ of node v, recursively along the tree structure, in the same way as in a Merkle tree; namely, as a function of the RSA digests of its children. Let $h_1, h_2, \ldots, h_l$, $l = \lceil \frac{1}{\epsilon} \rceil$, be two-universal hash functions, where h_i maps w_i-bit elements to $3w_i$-bit primes, $i = 1, \ldots, l$. For every leaf node v in tree $T(\epsilon)$ (recall: leaf nodes lie at level 0) that stores element e, we set $\chi(v) = e$, while for every non-leaf node v in $T(\epsilon)$ that lies in level $1 \leq i \leq l$, 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 (i.e., the prime representative of a quantity modulo N_{i-1} for $i > 1$), $N(v)$ is the set of children of node v and $g_i \in QR_{N_i}$.

Definition 4. *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 RSA tree $T(\epsilon)$ built on top of them, we define the RSA digest $\chi(S)$ of the set S to be equal to $\chi(r)$, where r is the root of the tree $T(\epsilon)$.*

Note that, given a set S, the RSA digest $\chi(S)$ depends on the elements in S and 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. For simplicity, we use both $\chi(S_v)$ and $\chi(v)$ to denote the RSA digest of node v, S_v being the set of elements contained in the subtree rooted at node v. We next show the main security property of our new authentication structure.

Theorem 2 (Collision Resistance). *Let k be the security parameter and $S_1 = \{e_1, e_2, \ldots, e_n\}$ a set of n elements. Given the associated RSA tree $T_1(\epsilon)$ built on top of S_1, 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 $S_2 \neq S_1$ such that $\chi(S_1) = \chi(S_2)$ is $\mathsf{neg}(k)$.*

Proof: (Sketch.) Suppose the adversary $\mathcal{A}$ has found another set $S_2 \neq S_1$ on which $\mathcal{A}$ has built a tree $T_2(\epsilon)$ (ϵ is fixed) such that $\chi(S_2) = \chi(S_1)$. We apply Lemma 2 for all levels and prove that the probability of finding a collision at the last level of the tree is bounded by a negligible function. We defer the complete proof to the full version of the paper. □

3.2 Authenticating Static Sets

We now describe how we can use the RSA tree authentication structure to optimally verify membership in a static set in constant time. The following methods form the basis for our main authentication schemes in the next section.

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 2 (system setup), the RSA moduli N_i and bases g_i, $1 \leq i \leq l$, are public. The server stores set S and builds the RSA tree $T(\epsilon)$ on top of S. For every node v of $T(\epsilon)$ that lies at level i ($0 \leq i \leq l-1$), the server also stores the prime representative $r_{i+1}(S_v)$ that has been computed for this subtree using function h_{i+1}. The client stores only the set digest $d = \chi(S)$.

Queries. We describe the algorithm that the server runs for constructing a proof needed 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 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:

$$\pi_i = \left(r_i(\chi(v_i)), g_i^{\prod_{u \in B(v_i)} r_i(\chi(u))} \mod N_i \right). \tag{2}$$

For simplicity, we set $\alpha_i = r_i(\chi(v_i))$ and

$$\beta_i = g_i^{\prod_{u \in B(v_i)} r_i(\chi(u))} \mod N_i. \tag{3}$$

Clearly, the size of the proof is $O(1)$, since $l = \lceil \frac{1}{\epsilon} \rceil$. For example in Figure 1, the proof for element 2 consists of 3 tuples:

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

Note that, as we are considering the static case, it is more time-efficient to use precomputed witnesses.

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

$$h_i(\alpha_i) = \beta_{i-1}^{\alpha_{i-1}} \mod N_{i-1}. \tag{4}$$

Also, the client verifies the *global* RSA digest against the locally stored digest, namely that the following relation holds:

$$d = \beta_l^{\alpha_l} \mod N_l. \tag{5}$$

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 3) consists of $l = \lceil \frac{1}{\epsilon} \rceil$, the RSA moduli $N_1, N_2, \dots, N_l$ (not $\phi(N_i)$), the exponentiation bases $g_1, g_2, \dots, g_l$ and the two-universal functions $h_1, h_2, \dots, h_l$. Also the adversary is given oracle access to all the algorithms that update and query the RSA tree and also verify queries. The digest d that appears in Definition 3 is the *root* digest of the RSA 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 scheme that uses the RSA tree for authenticating a static set of n elements is secure under the strong RSA assumption and according to Definition 3.*

Proof: (Sketch.) Suppose the adversary $\mathcal{A}$ has found a membership proof for an element $y \notin S$, i.e., $\mathcal{A}$ has found l tuples $(\alpha_1, \beta_1), (\alpha_2, \beta_2), \dots, (\alpha_l, \beta_l)$ such that $h_1(a_1) = y$, $h_i(\alpha_i) = \beta_{i-1}^{\alpha_{i-1}} \mod N_{i-1}$ for $i = 2, \dots, l$ and $\beta_l^{\alpha_l} = \chi(S) \mod N_l$. We use Corollary 2 and Equation 4 inductively on the (constant) number of levels of the RSA tree to find an upper bound on the probability that the adversary can forge proofs of membership. Using the property that the number of levels of the RSA tree is $O(1)$, we show that the probability of a successful attack is negligible. We defer the complete proof to the full version of the paper. □

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 4. *Let $0 < \epsilon < 1$ be a fixed constant. Under the strong RSA assumption, we can use the RSA tree 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 3; (2) The 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 $O(1)$; (6) The update authentication information has size $O(1)$.*

Note that this result applies also in the two-party model, with the difference that there is no update authentication information. Note also that all the constant time complexities have direct relation to $\frac{1}{\epsilon}$, namely they are $O\left(\frac{1}{\epsilon}\right)$ but since ϵ is constant they are considered $O(1)$.

4. AUTHENTICATING HASH TABLES

In this section we describe how to use our authentication structure that is based on the RSA 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 for the two-party model.

4.1 Three-Party Model

Let $\epsilon > 1$ and $\kappa > \frac{1}{\epsilon}$ be fixed constants. The general idea behind our approach for using the RSA tree to authenticate hash tables is the following. Let $S = \{e_1, e_2, \dots, e_n\}$ be the set of elements we would like to authenticate. Instead of building the RSA 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 $O(n/\log^\kappa n)$ buckets and each bucket contains $O(\log^\kappa n)$ elements. Note that in this case the internal nodes of the RSA tree have $O(n^\epsilon/\log^{\kappa\epsilon} n)$ children. Overall, we build the same RSA tree, as before, except that the leaves now hold prime representatives of the *accumulated bucket values* instead of the elements.

In particular, consider a bucket L that contains the elements $x_1, x_2, \dots, x_h$ (i.e., these elements lie in the same bucket, since they have the same value according to the function used by the hash table to uniformly put elements in the buckets). The *accumulated bucket value* of L is defined as $A_L = g_1^{r_1(x_1) r_1(x_2) \dots r_1(x_h)} \mod N_1$. Therefore, by computing these accumulated values of all the buckets, we add one additional level of accumulations in the RSA tree, that is, instead of using $l = \lceil \frac{1}{\epsilon} \rceil$ levels of accumulations, we are now using $l' = l + 1$ levels. Note that in this first (additional) level of accumulation, the number of elements that are accumulated is $O(\log^\kappa n)$, and not $O(n^\epsilon/\log^{\kappa\epsilon} n)$ as before.

Queries and Verification. Suppose we want to construct the proof for an element $x \in S$. Let $v_0, v_2, \dots, v_{l'}$ 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 $\pi_1, \pi_2, \dots, \pi_{l'}$, where π_i is defined in Equation 2. 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 RSA tree (we consider as leaves the elements within the buckets) that lies in level $1 \leq i \leq l'$, 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 $v_0, v_1, \dots, v_{l'}$ and collects the corresponding precomputed witnesses $\beta_1 = A_{j_1}^{(v_1)}, \beta_2 = A_{j_2}^{(v_2)}, \dots, \beta_{l'} = A_{j_{l'}}^{(v_{l'})}$ for some $j_1, j_2, \dots, j_{l'}$ (β_i is defined in Equation 3). Since the depth of the tree is constant ($\lceil \frac{1}{\epsilon} \rceil + 2$), the time needed for querying is $O(1)$ (we assume that query time is the time 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, verification can be performed exactly as indicated by Equations 4 and 5 and, thus, takes also $O(1)$ 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 belongs to a certain bucket L. Let $v_0, v_2, \ldots, v_{l'}$ be the path from the newly inserted element to the root of the tree. Note that the goal of the update algorithm is twofold: **(1)** All the RSA digests $\chi(v_i)$, $1 \leq i \leq l'$ (note that $\chi(v_0) = x$), 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 *local* witnesses $A_j^{(v_i)}$ where $j \in N(v_i)$. This is required to maintain the query complexity to be constant. Finally, note that, whenever an update is performed, the source sends, together with the signed RSA digest, the prime representatives of the updated RSA digests along the path of the update. We use the following result from [30] for efficiently maintaining updated precomputed witnesses and overall achieving constant query time.

Lemma 3 ($O(n \log n)$ Witness Updates [30]). *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 $A_i = g^{\prod_{j \neq i} x_j} \mod N$ for $i = 1, \ldots, n$ in $O(n \log n)$ time.*

By using the above lemma, we can prove the following:

Lemma 4. *Let $0 < \epsilon < 1$ and $\kappa > 1/\epsilon$ be fixed constants. Given a hash table for n elements with $O(n/\log^\kappa n)$ buckets of expected size $O(\log^\kappa n)$ and the RSA tree $T(\epsilon)$ built on top of it, without the knowledge of $\phi(N_i)$ $(i = 1, \ldots, l')$, we can support updates in $O(n^\epsilon/\log^{\kappa\epsilon-1} n)$ amortized expected time.*

Proof: (Sketch.) We use Lemma 3 for the internal nodes and the buckets. Then, we note that over a course of $O(n)$ updates, there will be an expensive update. Therefore, we can amortize and since we are using a two-universal hash function to distribute elements in the bucket, all the results hold in expectation. We defer the complete proof to the full version of the paper. □

Note that if we know $\phi(N_i)$ $(i = 1, \ldots, l')$, we can support updates in $O(1)$ amortized expected time (see details in Section 5). This is because we can compute inverses and update witnesses in constant time. However, rebuilding the hash table is needed and the amortized expected time will be in this case $\frac{\sum_{i=1}^t O(1)+O(n)}{t+1} = O(1)$. This is very important in the three-party model, since the source can do the updates even more efficiently. We are now ready to present our main result in the three-party model:

Theorem 5. *Let $0 < \epsilon < 1$ and $\kappa > \frac{1}{\epsilon}$ be fixed constants. Under the strong RSA assumption, we can use the RSA tree 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 3; (2) The amortized expected update time at the server is $O\left(n^\epsilon/\log^{\kappa\epsilon-1} n\right)$; (3) The amortized expected update time at the source is $O(1)$; (4) The 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 $O(1)$.*

Finally, note that if we restrict ourselves to the three-party model, we can achieve constant amortized update time at the untrusted server too, by keeping the update authentication information constant and increasing the query time to expected $O\left(n^\epsilon/\log^{\kappa\epsilon} n\right)$. Namely, the source sends the updated digests along the path of the update; therefore the untrusted server does not have to perform the update itself. However, this method does not use precomputed witnesses and thus the query time is not constant. Namely, the query time is *expected* $O\left(n^\epsilon/\log^{\kappa\epsilon} n\right)$, since it depends on the randomness of the two-universal function used to distribute the elements in the $O(n/\log^\kappa n)$ buckets. Therefore, we have the following extended result for the three-party model (note that in this case we do not need $\kappa > \frac{1}{\epsilon}$):

Theorem 6. *Let $0 < \epsilon < 1$ and $\kappa > 0$ be fixed constants. Under the strong RSA assumption, we can use the RSA tree 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 3; (2) The amortized expected update time at the server is $O(1)$; (3) The amortized expected update time at the source is $O(1)$; (4) The expected query time is $O\left(n^\epsilon/\log^{\kappa\epsilon} n\right)$; (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 $O(1)$.*

However, as we will see next, this solution does not apply to the two-party model, since in the three-party model we have substantial help from the source.

We finally note that we can choose that scheme being best suited for the application of interest: in particular, we can use the scheme of Theorem 5 for applications where updates do not happen very often and queries are intensive and frequent, whereas we can use the scheme of Theorem 6 for applications where updates are much more frequent than queries (e.g., auditing). Finally, we point out that in the query cost of Theorem 6, there is a low-order term that is absorbed in the $O(\cdot)$ notation. This term is $O(\log^\kappa n)$ and is the time needed to compute the witness within the bucket. Therefore, parameter κ affects (though not asymptotically) the time to compute the witness within the bucket.

4.2 Two-Party Model

We now describe how we can implement an authenticated hash table using the RSA 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 third trusted party and no PKI is used.

The query and the verification for a certain element are done exactly in the same way as in the three-party model, namely both the query and verification take time $O(1)$. Moreover, the updates at the untrusted server can also be performed in the same way, namely in $O\left(n^\epsilon/\log^{\kappa\epsilon-1} n\right)$ amortized time. Suppose now the client wants to do an update, e.g., insert an element x in the existing set of elements S. The client should somehow be able to locally compute the new RSA digest $\chi(S \cup x)$, because the client cannot get the new digest from the untrusted server as the untrusted server can return a false digest. We now describe how the client can update the digest. Let x belong to some bucket L. The server picks another element $y \in S$ that belongs to the *same* bucket and sends the proof for y. Subsequently, the server performs the update for x and, together with the previous proof, it sends the new prime representatives along the update path. This extended proof (which not only contains the ordinary proof for y, but also the new prime representatives along the path) is called *consistency proof*, for it is used for updates.

The client now verifies y (by using the previous RSA digest) and has all the information needed to compute the new RSA digest that corresponds to the set $S \cup \{x\}$ (or $S - \{x\}$ in the case of a deletion). The most important step is to update the bucket digest (where the update takes place) and then propagate the update along the path, by executing $O(1)$ exponentiations and also checking that the new updated digests are consistent with the new prime representatives sent by the untrusted server (as part of the consistency proof). The client is able to update the RSA digest in $O(1)$ time, since the client knows $\phi(N_i)$ and therefore can compute inverses (see details in Section 5). In the case of a query, the proof provided by the server has similar structure as in the update case and is simply called *verification proof.*

Note that when the server rebuilds the hash table, the client has to receive all the elements and locally rebuild the hash table. Therefore, the amortized space the client needs is $O(1)$ and the consistency proof has $O(1)$ amortized size. We can now state our main result for the two-party model:

THEOREM 7. *Let $0 < \epsilon < 1$ and $\kappa > \frac{1}{\epsilon}$ be fixed constants. Under the strong RSA assumption, we can use the RSA tree 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 3; (2) The amortized expected update time at the server is $O\left(n^{\epsilon}/\log^{\kappa\epsilon-1} n\right)$; (3) The amortized expected 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 $O(1)$.*

4.3 Authenticating Non-Membership

So far our results have been presented for authenticated membership queries. We describe now how non-membership queries can also be supported. To do that, 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 $O(\log^{\kappa} n)$, Theorems 5 and 7 hold for non-membership proofs as well. Finally, Theorem 6 holds as well, with the only difference that the amortized expected update time at the source and at the server is now $O(\log^{\kappa} n)$, since at every update the source will have to recompute the digest of the $O(\log^{\kappa} n)$-sized bucket, for the sorted pairs must be updated.

5. EVALUATION AND ANALYSIS

In this section we provide an evaluation of our authenticated hash table structure. First, we count the number of primitive operations (mainly exponentiations) that every complexity measure (update, query, proof size, size of update authentication information) uses for general values of ϵ and κ. Note that this is feasible since there is no significant overhead of (hidden) constant factors in our scheme. The only constant factors included in our complexities are well-understood, namely, functions of ϵ and κ. We are going to evaluate the three-party version of the hash table described in Theorem 6, where everything is constant, apart from the query time, which is $O(n^{\epsilon}/\log^{\kappa\epsilon} n)$. As we saw in Section 2, we have to use multiple RSA moduli $N_1, N_2, \ldots, N_l$.

However, since the size of each modulus is increasing with $1/\epsilon$, for the experiments, as observed in Section 2, we are going to restrict the input of each level of accumulation to be the output of a cryptographic hash function, e.g., SHA-256 plus a constant number of extra bits that, when appended to the output of the hash function, give a prime number. Therefore, we do not use prime representatives, but use a 1024-bit modulus N for all tree levels. Basically, instead of having prime representatives r_i to represent an element x_i through the two-universal function h_i, where $h_i(r_i) = x_i$, we use directly the prime value $g(x_i) + 2^t$ (and not $r_i = h_i^{-1}(x_i)$), where $t = O(1)$ is the minimum number of bits we need to append to $g(x_i)$ such that $g(x_i) + 2^t$ is prime and $g(x_i)$ is the output of SHA-256 for example. This makes our scheme more practical and still secure, since, as correctly observed by Baric and Pfitzmann [2], the security of using the output of a cryptographic hash function plus some extra bits as the input of the RSA accumulator is based on random oracles, which is sufficiently good for practical reasons. Therefore, for each level, the input values to the accumulator are $(256+t)$-bit values (output of SHA-256 plus the bits we need to append to get the prime) and the same RSA modulus N is used, which is 1024 bits. We defer the proof of security of our scheme in the random oracle model to the full version of the paper.

In our scheme we set $t = 14$. By the prime distribution theorem, which states that the number of primes less than x is approximately $\frac{x}{\ln x}$, we have that the number of primes between x and y ($x \leq y$) is $\simeq \frac{y}{\ln y} - \frac{x}{\ln x}$. Since 2^b is a very large number, we can approximate the number of primes in the interval $[2^{b+t}, 2^{b+t} + 2^t]$ with $\frac{2^t}{\ln(2^{b+t})}$, where b is the bit-length of the output of the cryptographic hash function and t is the number of extra bits we add. We can now prove that we hit a prime with probability p, if we sample (there are around 87 primes for $b = 256$ and $t = 14$)

$$\frac{\ln(1-p)}{\ln\left(1 - \frac{2}{(b+t)\ln(2)}\right)}$$

times in the interval $[2^{b+t}, 2^{b+t} + 2^t]$ (we only sample from the odd numbers in $[2^{b+t}, 2^{b+t} + 2^t]$). For our scheme, for $p = 0.99$, if we use SHA-256, i.e., $b = 256$, and add an extra $t = 14$ bits, we need to sample around 428 times. That means that we can efficiently extend our SHA-256 digest to a 270-bit prime exponent by using an extra 14 bits. Finally, we note that if we use the algorithm presented in [2], which for $i = 0, \ldots, t$ checks to see if $2^{t+b} + 2^i$ is a prime, we need only $O(t)$ time. However, this method does not guarantee finding a prime.

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 of 1024-bit integers.

We benchmarked the time needed for these operations on 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 we used NTL [31], a standard optimized library for number theory, interfaced with C++. For 200 runs the average time for computing the power of a 1024-bit number to a 270-bit exponent and then reducing modulo N was found to be $t_1 = 1.5$ms, and the average time for computing the inverse of a 270-bit number modulo $\phi(N)$ was $t_2 = 0.00009$ms. Also multiplication of 270-bit numbers modulo $\phi(N)$ was found to be $t_3 = 0.00016$ms. Finally, 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. As expected, exponentiation dominates most of the time.

Updates. Let f be a function that takes as input a 1024-bit integer x, computes its SHA-256-bit digest and extends it to a 270-bit prime, by adding the appropriate 14 bits. We are going to assume that the time for applying $f(\cdot)$ to x is dominated by the SHA-256 computation and is equal to $t_4 = 0.01$ms. As we saw in 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 certain 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 is

$$t_{\mathsf{update}} = t_1 + t_2 + t_4 + \epsilon^{-1}(t_1 + t_2 + t_3 + t_4), \qquad (6)$$

which is not dependent on n.

Verification. The verification is performed by doing $\epsilon^{-1}+1$ exponentiations and $f(\cdot)$ computations. Namely, by using $f(\cdot)$ instead of prime representatives, Equation 4 becomes $\alpha_i = f(\beta_{i-1}^{\alpha_{i-1}} \mod N)$ (this is actually performed by cutting the last 14 bits of a_i and comparing the result with the SHA-256 digest of $\beta_{i-1}^{\alpha_{i-1}} \mod N$). Therefore,

$$t_{\mathsf{verify}} = (\epsilon^{-1} + 1)(t_1 + t_4), \qquad (7)$$

which is also not dependent on n.

Table 2: Actual values of the complexity measures of our scheme (computed with the derived equations) for various values of $\epsilon = 0.1, 0.125, 0.3, 0.5$, $\kappa = 1$ and $n = 100,000,000$. Note that n (number of items in the hash table) influences only query time.

ϵ	proof size (KB)	update info. (KB)	query time (ms)	update time (ms)	verify time (ms)
0.1	1.77	1.90	107.95	16.61	16.60
0.125	1.45	1.58	119.38	13.59	13.58
0.3	0.70	0.82	507.99	6.54	6.53
0.5	0.48	0.61	5831.70	4.53	4.52

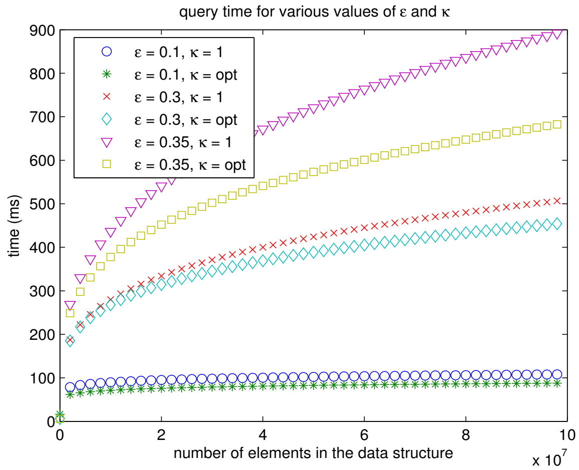

Figure 2: This figure shows how query time scales as the number of the elements in the hash table increases, for different values of $\epsilon = 0.35, 0.3, 0.1$ and $\kappa = 1$ or $\kappa = \text{opt}$, an optimal value for given ϵ. Note that for $\epsilon = 0.1$ querying is very efficient, as we have fewer exponentiations per level.

Queries. For the queries we have to do $\log^{\kappa} n$ exponentiations at the first level and $n^{\epsilon}/\log^{\kappa\epsilon} n$ exponentiations for the remaining ϵ^{-1} levels. Therefore,

$$t_{\mathsf{query}} = (\epsilon^{-1} n^{\epsilon}/\log^{\kappa\epsilon} n + \log^{\kappa} n)t_1. \qquad (8)$$

Note that we do not have to do $f(\cdot)$ computations since all the $f(\cdot)$ values are precomputed. Also we cannot use cheap multiplications (and one exponentiation per level), since the security collapses if we reveal $\phi(N)$ to the server.

Communication Complexity. The proof and the update authentication information are $\epsilon^{-1}+1$ pairs of 1024-bit numbers and 270-bit $f(\cdot)$ values. Thus,

$$s_{\mathsf{proof}} = (\epsilon^{-1} + 1)(1024 + 270). \qquad (9)$$

The update authentication information includes also a signature on the final (root) RSA digest, i.e., 1024 bits more.

In order to perform an actual evaluation of our scheme, we set $\epsilon = 0.1, 0.125, 0.3, 0.5$ (the RSA tree has 10, 8, 3, 2 levels respectively) and $\kappa = 1$ (each bucket has $\log n$ elements). Table 2 shows the evaluation of all the above measures for a hash table that contains 100,000,000 elements, where we vary the value of ϵ, namely the number of levels of the RSA tree. We can now make the following observations: First, as ϵ increases, the verification time and the update time decrease since they are proportional to ϵ^{-1}. However, query time increases since the internal nodes of the tree become larger and more exponentiations have to be performed. Finally, in terms of communication cost, our system is very efficient since only at most 1.90KB have to be communicated.

In Figure 2, we can see how query time scales with increasing number of elements in the hash table. Here we observe that for $\epsilon = 0.1$, the query time scales very efficiently and specifically, for $n = 100,000,000$ elements stored in the hash table, the time to produce a proof for an element is about 100ms. Note that in Figure 2 we use the value $\kappa = 1$ or the optimal value $\kappa = \mathsf{opt}$ (see the last paragraph about how opt is obtained). The query time for $\kappa = \mathsf{opt}$ is about 70 ms.

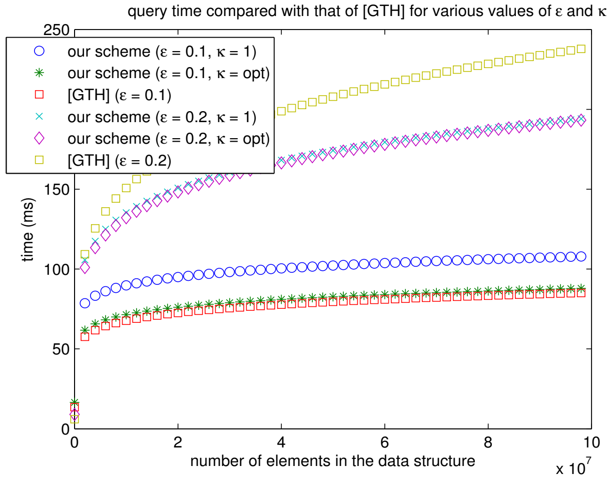

Figure 3: Comparison of query time with [GTH] for $\epsilon = 0.1, 0.2$.

We next compare our scheme with the scheme of [14], denoted [GTH], which uses the RSA accumulator and proves security using the strong RSA assumption. As shown in Table 1, their query time, update authentication information and update time is $O(n^\epsilon)$ for some constant $0 < \epsilon < 1$. However the constant hidden in the $O(\cdot)$ notation is $\frac{1-\epsilon}{\epsilon}$. Therefore, their actual query time is $\frac{1-\epsilon}{\epsilon} n^\epsilon t_1$. In Figure 3, we plot our scheme's query time and the query time from [GTH] for $\epsilon = 0.1, 0.2$ and $\kappa = 1$ or $\kappa = \mathsf{opt}$. We see that our system scales better for $\epsilon = 0.2$. For $\epsilon = 0.1$ our system scales almost the same for $\kappa = \mathsf{opt}$ and scales better for $\kappa = 1$ only when $n \geq 10^{16}$, which justifies the better asymptotic notation. However, the update time and update authentication information of our scheme are maintained constant and do not depend on n, justifying the scalability of our system.

Tuning Constants ϵ and κ. We conclude this analysis section by providing a way to optimally tune the parameters ϵ and κ so that we can achieve the minimum query time in practice. Suppose our hash table stores at some point n elements. We want to choose the values of ϵ and κ such that the query time is minimum. If we take the partial derivatives of $t_{\mathsf{query}} = (\epsilon^{-1} n^\epsilon / \log^{\kappa\epsilon} n + \log^\kappa n) t_1$ (we fix n, and our free variables are ϵ and κ), after some algebra, we see that the minimum query time is achieved for $\epsilon = \frac{1}{\ln n - 1}$ and $\kappa = \frac{1}{\ln \log n}$. If we now keep κ fixed, the minimum query time is achieved for $\epsilon = \frac{1}{\ln n - \ln \log^\kappa n}$, whereas if we keep ϵ fixed, the minimum query time is achieved for $\kappa = \frac{\epsilon \ln n}{(1+\epsilon) \ln \log n}$. Recall that the hash table is rebuilt when its size is doubled. Thus, we can use the above formulas to optimally tune ϵ and κ when rebuilding the hash table, thus optimizing query authentication. Finally, since for security reasons ϵ should not be a function of n, it is preferable to fix ϵ and tune κ accordingly (we have used this in Figures 2 and 3 for $\kappa = \mathsf{opt}$). This is actually equivalent to optimally balancing the work within the buckets and the work of an internal node of the RSA tree, and gives a partition of our elements in $O(n^{1/(1+\epsilon)})$ buckets, each bucket having $O(n^{\epsilon/(1+\epsilon)})$ elements. In this way, we can prove using the same techniques that the amortized expected update time in Theorem 5 is $O(n^{\epsilon/(1+\epsilon)} \log n)$, while the expected query time in Theorem 6 is $O(n^{\epsilon/(1+\epsilon)})$. All other complexity measures are still $O(1)$.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we propose a new, provably secure, cryptographic construction for authenticating the hash table functionality. We use nested RSA accumulators on a tree of constant depth in order to provide with authenticated hash table queries with 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 previous works that use cryptographic accumulators, reducing main complexity measures (such as query costs—time to produce the proof) to constant, yet, still improving the update time, which for any fixed constants $0 < \epsilon < 1$ and $\kappa > 1/\epsilon$ is shown to be $O\left(n^\epsilon / \log^{\kappa\epsilon - 1} n\right)$. An alternative scheme we propose keeps update cost constant while having $O\left(n^\epsilon / \log^{\kappa\epsilon} n\right)$ query cost.

An open problem of great importance is reducing these bounds to $\log n$ and still keeping all the other complexity measures constant. Even better, one could ask if there is a solution for authenticating hash tables and still keeping all the complexities constant. This also implies a direction towards studying lower bounds for set-membership authentication (as, e.g., in [10, 33]): given a cryptographic primitive or authentication model, what is the best we can do in terms of complexity (and still being provable secure)? This work suggests that there is a trade-off between security and complexity which might be a starting point in studying lower bounds. Finally, it would be interesting to look into how we can provide with non-amortized results for Theorem 7.

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 Chris Erway for providing pointers to the NTL library.

7. REFERENCES

[1] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *Proc. ACM Conf. on Computer and Communications Security (CCS)*, pp. 598-609, 2007.

[2] N. Baric and B. Pfitzmann. Collision-free accumulators and fail-stop signature schemes without trees. In *Proc. EUROCRYPT*, pp. 480-494, 1997.

[3] J. Benaloh and M. de Mare. One-way accumulators: A decentralized alternative to digital signatures. In *Proc. EUROCRYPT*, pp. 274-285, 1993.

[4] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. In *Proc.IEEE Symp. on Foundations of Computer Science (FOCS)*, pp. 90-99, 1991.

[5] A. Buldas, P. Laud, and H. Lipmaa. Accountable certificate management using undeniable attestations. In *Proc. ACM Conf. on Computer and Communications Security (CCS)*, pp. 9-18, 2000.

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

[7] I. L. Carter and M. N. Wegman. Universal classes of hash functions. In *Proc. ACM Symp. on Theory of Computing (STOC)*, pp. 106-112, 1977.

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

[9] 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.

[10] C. Dwork, M. Naor, G. N. Rothblum, and V. Vaikuntanathan. How efficient can memory checking be? Manuscript, 2008.

[11] R. Gennaro, S. Halevi, and T. Rabin. Secure hash-and-sign signatures without the random oracle. In *Proc. EUROCRYPT*, pp. 123-139, 1999.

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

[13] M. T. Goodrich, C. Papamanthou, R. Tamassia, and N. Triandopoulos. Athos: Efficient authentication of outsourced file systems. In *Proc. Information Security Conf. (ISC)*, pp. 80-96, 2008.

[14] M. T. Goodrich, R. Tamassia, and J. Hasic. An efficient dynamic and distributed cryptographic accumulator. In *Proc. Information Security Conf. (ISC)*, pp. 372-388, 2002.

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

[16] M. T. Goodrich, R. Tamassia, and N. Triandopoulos. Super-efficient verification of dynamic outsourced databases. In *Proc. CT-RSA*, pp. 407-424, 2008.

[17] M. T. Goodrich, R. Tamassia, N. Triandopoulos, and R. Cohen. Authenticated data structures for graph and geometric searching. In *Proc. CT-RSA*, pp. 295-313, 2003.

[18] A. Hutflesz, H.-W. Six, and P. Widmayer. Globally order preserving multidimensional linear hashing. In *Proc. IEEE Int. Conf. on Data Engineering (ICDE)*, pp. 572-579, 1988.

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

[20] J. Li, N. Li, and R. Xue. Universal accumulators with efficient nonmembership proofs. In *Proc. Applied Cryptography and Network Security (ACNS)*, pp. 253-269, 2007.

[21] N. Linial and O. Sasson. Non-expansive hashing. In *Proc. ACM Symp. on Theory of Computing (STOC)*, pp. 509-517, 1996.

[22] 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.

[23] R. C. Merkle. A certified digital signature. In *Proc. CRYPTO*, pp. 218–238, 1989.

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

[25] M. Naor and K. Nissim. Certificate revocation and certificate update. In *Proc. USENIX Security Symposium*, pp. 217-228, 1998.

[26] L. Nguyen. Accumulators from bilinear pairings and applications. In *Proc. CT-RSA*, pp. 275-292, 2005.

[27] G. Nuckolls. Verified query results from hybrid authentication trees. In *Proc. Data and Applications Security (DBSec)*, pages 84–98, 2005.

[28] C. Papamanthou and R. Tamassia. Time and space efficient algorithms for two-party authenticated data structures. In *Proc. Int. Conf. on Information and Communications Security (ICICS)*, pp. 1-15, 2007.

[29] T. Sander. Efficient accumulators without trapdoor (Extended abstract). In *Proc. Int. Conf. on Information and Communications Security (ICICS)*, pp. 252-262, 1999.

[30] T. Sander, A. Ta-Shma, and M. Yung. Blind, auditable membership proofs. In *Proc. Financial Cryptography (FC)*, pp. 53-71, 2000.

[31] V. Shoup. NTL: A library for doing number theory. http://www.shoup.net/ntl/.

[32] R. Tamassia. Authenticated data structures. In *Proc. European Symp. on Algorithms (ESA)*, pp. 2-5, 2003.

[33] 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)*, pp. 153-165, 2005.

[34] R. Tamassia and N. Triandopoulos. Efficient content authentication in peer-to-peer networks. In *Proc. Applied Cryptography and Network Security (ACNS)*, pp. 354-372, 2007.

[35] P. Wang, H. Wang, and J. Pieprzyk. A new dynamic accumulator for batch updates. In *Proc. Int. Conf. on Information and Communications Security (ICICS)*, pp. 98-112, 2007.