Optimal Verification of Operations on Dynamic Sets

Charalampos Papamanthou[1], Roberto Tamassia[1], and Nikos Triandopoulos[2,3]

[1] Brown University, Providence RI, USA
[2] RSA Laboratories, Cambridge MA, USA
[3] Boston University, Boston MA, USA

Abstract. We study the design of protocols for *set-operation verification*, namely the problem of cryptographically checking the correctness of outsourced set operations performed by an untrusted server over a dynamic collection of sets that are owned (and updated) by a trusted source. We present new authenticated data structures that allow any entity to *publicly* verify a proof attesting the correctness of primitive set operations such as *intersection*, *union*, *subset* and *set difference*. Based on a novel extension of the security properties of *bilinear-map accumulators* as well as on a primitive called *accumulation tree*, our protocols achieve *optimal* verification and proof complexity (i.e., only proportional to the size of the query parameters and the answer), as well as *optimal* update complexity (i.e., constant), while incurring no extra asymptotic space overhead. The proof construction is also efficient, adding a *logarithmic* overhead to the computation of the answer of a set-operation query. In contrast, existing schemes entail high communication and verification costs or high storage costs. Applications of interest include efficient verification of keyword search and database queries. The security of our protocols is based on the *bilinear q-strong Diffie-Hellman* assumption.

1 Introduction

Providing integrity guarantees in third-party data management settings is an active area of research, especially in view of the growth in usage of cloud computing. In such settings, verifying the correctness of outsourced computations performed over remotely stored data becomes a crucial property for the trustworthiness of cloud services. Such a verification process should incur minimal overheads to the clients or otherwise the benefits of computation outsourcing are dismissed; ideally, computations should be verified without having to locally rerun them or to utilize too much extra cloud storage.

In this paper, we study the verification of outsourced operations on general sets and consider the following problem. Assuming that a *dynamic collection* of m sets $S_1, S_2, \ldots, S_m$ is remotely stored at an untrusted server, we wish to *publicly* verify basic operations on these sets, such as *intersection*, *union* and *set difference*. For example, for an intersection query of t sets specified by indices $1 \leq i_1, i_2, \ldots, i_t \leq m$, we aim at designing techniques that allow any client to cryptographically check the correctness of the returned answer $\mathsf{I} = S_{i_1} \cap S_{i_2} \cap \ldots \cap S_{i_t}$. Moreover, we wish the verification of any set operation to be *operation-sensitive*, meaning that the resulting complexity depends *only on the (description and outcome of the) operation, and not on the sizes of the involved sets*. That is, if $\delta = |\mathsf{I}|$ is the answer size then we would like the verification cost

P. Rogaway (Ed.): CRYPTO 2011, LNCS 6841, pp. 91–110, 2011.

to be proportional to $t+\delta$, and independent of m or $\sum_i |S_i|$; note that work at least proportional to $t + \delta$ is needed to verify any such query's answer. Applications of interest include *keyword search* and *database* queries, which boil down to set operations.

Relation to verifiable computing. Recent works on *verifiable computing* [1,12,16] achieve operation-sensitive verification of general functionalities, thus covering set operations as a special case. Although such approaches clearly meet our goal with respect to optimal verifiability, they are inherently inadequate to meet our other goals with respect to *public verifiability* and *dynamic updates*, both important properties in the context of outsourced data querying. Indeed, to outsource the computation as an encrypted circuit, the works in [1,12,16] make use of some secret information which is also used by the verification algorithm, thus effectively supporting only one verifier; instead, we seek for schemes that allow *any client* (knowing only a public key) to query the set collection and verify the returned results. Also, the description of the circuit in these works is fixed at the initialization of the scheme, thus effectively supporting no updates in the outsourced data; instead, we seek for schemes that are dynamic. In other scenarios, but still in the secret-key setting, protocols for general functionalities and polynomial evaluation have recently been proposed in [11] and [6] respectively.

Aiming at both publicly verifiable and dynamic solutions, we study set-operation verification in the model of *authenticated data structures* (ADSs). A typical setting in this model, usually referred to as the *three-party* model [36], involves protocols executed by three participating entities. A trusted party, called *source*, owns a data structure (here, a collection of sets) that is replicated along with some cryptographic information to one or more untrusted parties, called *servers*. Accordingly, *clients* issue data-structure queries to the servers and are able to verify the correctness of the returned answers, based only on knowledge of public information which includes a *public key* and a *digest* produced by the source (e.g., the root hash of a Merkle tree).[1] Updates on the data structure are performed by the source and appropriately propagated by the servers. Variations of this model include: (i) a *two-party* variant (e.g., [30]), where the source keeps only a small state (i.e., only a digest) and performs both the updates/queries and the verifications—this model is directly comparable to the model of verifiable computing; (ii) the *memory checking* model [7], where read/write operations on an array of memory cells is verified—however, the absence of the notion of proof computation in memory checking (the server is just a storage device) as well as the feature of public verifiability in authenticated data structures make the two models fundamentally different.[2]

Achieving operation-sensitive verification. In this work, we design authenticated data structures for the verification of set operations in an *operation-sensitive* manner, where the proof and verification complexity depends only on the description and outcome of the operation and not on the size of the involved sets. Conceptually, this property is similar to the property of *super-efficient verification* that has been studied in certifying algorithms [21] and certification data structures [19,37], which is achieved as well as in the context of verifiable computing [1,12,16], where an answer can be verified with complexity asymptotically less than the complexity required to produce it. Whether the

[1] Conveying the trust clients have in the source, the authentic digest is assumed to be publicly available; in practice, a time-stamped and digitally signed digest is outsourced to the server.

[2] Indeed, memory checking might require *secret* memory, e.g., as in the PRF construction in [7].

above optimality property is achievable for set operations (while keeping storage linear) was posed as an open problem in [23]. We close this problem in the affirmative.

All existing schemes for set-operation verification fall into the following two rather straightforward and highly inefficient solutions. Either short proofs for the answer of every possible set-operation query are precomputed allowing for optimal verification at the client at the cost of exponential storage and update overheads at the source and the server—an undesirable trade-off, as it is against storage outsourcing. Or integrity proofs for all the elements of the sets involved in the query are given to the client who locally verifies the query result: in this case the verification complexity can be linear in the problem size—an undesirable feature, as it is against computation outsourcing.

We achieve optimal verification by departing from the above approaches as follows. We first reduce the problem of verifying set operations to the problem of *verifying the validity of some more primitive relations on sets*, namely *subset containment* and *set disjointness*. Then for each such primitive relation we employ a corresponding cryptographic primitive to optimally verify its validity. In particular, we extend the *bilinear-map accumulator* to optimally verify subset containment (Lemmas 1 and 4), inspired by [32]. We then employ the extended Euclidean algorithm over polynomials (Lemma 5) in combination with subset containment proofs to provide a novel optimal verification test for set disjointness. The intuition behind our technique is that disjoint sets can be represented by polynomials mutually indivisible, therefore there exist other polynomials so that the sum of their pairwise products equals to one—this is the test to be used in the proof. Still, transmitting (and processing) these polynomials is bandwidth (and time) prohibitive and does not lead to operation-sensitive verification. Bilinearity properties, however, allow us to compress their coefficients in the exponent and, yet, use them meaningfully, i.e., compute an internal product. This is why although a conceptually simpler RSA accumulator [5] would yield a mathematically sound solution, a bilinear-map accumulator [28] is essential for achieving the desired complexity goal.

We formally describe our protocols using an *authenticated data structure scheme* or *ADS scheme* (Definition 1). An ADS scheme consists of algorithms {genkey, setup, update, refresh, query, verify} such that: (i) genkey produces the secret and public key of the system; (ii) on input a plain data structure D, setup initializes the authenticated data structure auth(D); (iii) *having* access to the secret key, update computes the updated digest of auth(D); (iv) *without having* access to the secret key, refresh updates auth(D); (v) query computes cryptographic proofs $\Pi(q)$ for answers $\alpha(q)$ to data structure queries q; (vi) verify processes a proof Π and an answer α and either *accepts* or *rejects*. Note that neither query nor verify have access to the secret key, thus modeling computation outsourcing and public verifiability. An ADS scheme must satisfy certain correctness and security properties (Definitions 2 and 3). We note that protocols in both the three-party and the two-party models can be realized via an ADS scheme.

Our main result, Theorem 1, presents the first ADS scheme to achieve *optimal verification* of the set operations *intersection*, *union*, *subset* and *set difference*, as well as *optimal updates* on the underlying collection of sets. Our scheme is proved secure under the bilinear extension of the q-strong Diffie-Hellman assumption (see, e.g., [8]).

Table 1. Asymptotic access and group complexities of various ADS schemes for intersection queries on $t = O(1)$ sets in a collection of m sets with answer size δ. Here, M is the sum of sizes of all the sets and $0 < \epsilon < 1$ is a constant. Also, all sizes of the intersected or updated sets are $\Theta(n)$, $|\Pi|$ denotes the size of the proof, and CR stands from "collision resistance".

	setup	update, refresh	query	verify, $\lvert\Pi\rvert$	**assumption**
[23,38]	$m + M$	$\log n + \log m$	$n + \log m$	$n + \log m$	Generic CR
[26]	$m + M$	$m + M$	n	n	Strong RSA
[29]	$m^t + M$	m^t	1	δ	Discrete Log
this work	$m + M$	1	$n \log^3 n + m^\epsilon \log m$	δ	Bilinear q-Strong DH

Complexity model. To explicitly measure complexity of various algorithms with respect to number of primitive cryptographic operations, without considering the dependency on the security parameter, we adopt the complexity model used in memory checking [7,14], which has been only implicitly used in ADS literature. The *access complexity* of an algorithm is defined as the number of memory accesses performed during its execution on the authenticated data structure that is stored in an indexed memory of n cells.[3] E.g., a Merkle tree [24] has $O(\log n)$ update access complexity since the update algorithm needs to read and write $O(\log n)$ memory cells of the authenticated data structure, each cell storing exactly one hash value. The *group complexity* of a data collection (e.g., proof or ADS group complexity) is defined as the number of elementary data objects (e.g., hash values or elements in $\mathbb{Z}_p$) contained in this collection. Note that although the access and group complexities are respectively related to the time and space complexities, the former are in principle smaller than the latter. This is because time and space complexities are counting number of bits and are always functions of the security parameter which, in turn, is always $\Omega(\log n)$. Therefore time and space complexities are always $\Omega(\log n)$, whereas access and group complexities can be $O(1)$. Finally, whenever it is clear from the context, we omit the terms "access" and "group".

Related work. The great majority of authenticated data structures involve the use of cryptographic hashing [2,7,18,20,23,27,39] or other primitives [17,31,32] to hierarchically compute over the outsourced data one or more digests. Most of these schemes incur verification costs that are proportional to the time spent to produce the query answer, thus they are not operation sensitive. Some bandwidth-optimal and operation-sensitive solutions for verification of various (e.g., range search) queries appear in [2,19].

Despite the fact that privacy-related problems for set operations have been extensively studied in the cryptographic literature (e.g., [9,15]), existing work on the integrity dimension of set operations appears mostly in the database literature. In [23], the importance of coming up with an operation-sensitive scheme is identified. In [26], possibly the closest in context work to ours, set intersection, union and difference are authenticated with linear costs. Similar bounds appear in [38]. In [29], a different approach is taken: In order to achieve operation-sensitivity, expensive pre-processing and

[3] We use the term "access complexity" instead of the "query complexity" used in memory checking [7,14] to avoid ambiguity when referring to algorithm query of the ADS scheme. We also require that each memory cell can store up to $O(\mathsf{poly}(\log n))$ bits, a word size used in [7,14].

exponential space are required (answers to all possible queries are signed). Finally, related to our work are non-membership proofs, both for the RSA [22] and the bilinear-map [3,13] accumulators. A comparison of our work with existing schemes appears in Table 1.

2 Preliminaries

We denote with k the security parameter and with $\mathsf{neg}(k)$ a negligible function.[4]

The bilinear-map accumulator. Let $\mathbb{G}$ be a cyclic multiplicative group of prime order p, generated by element $g \in \mathbb{G}$. Let also $\mathcal{G}$ be a cyclic multiplicative group of the same order p, such that there exists a pairing $e : \mathbb{G} \times \mathbb{G} \rightarrow \mathcal{G}$ with the following properties: (i) Bilinearity: $e(P^a, Q^b) = e(P, Q)^{ab}$ for all $P, Q \in \mathbb{G}$ and $a, b \in \mathbb{Z}_p$; (ii) Non-degeneracy: $e(g, g) \neq 1$; (iii) Computability: For all $P, Q \in \mathbb{G}$, $e(P, Q)$ is efficiently computable. We call $(p, \mathbb{G}, \mathcal{G}, e, g)$ a tuple of bilinear pairing parameters, produced as the output of a probabilistic polynomial-time algorithm that runs on input 1^k.

In this setting, the bilinear-map accumulator [28] is an efficient way to provide short *proofs of membership* for elements that belong to a set. Let $s \in \mathbb{Z}_p^*$ be a randomly chosen value that constitutes the trapdoor in the scheme. The accumulator primitive accumulates elements in $\mathbb{Z}_p - \{s\}$, outputting a value that is an element in $\mathbb{G}$. For a set of elements $\mathcal{X}$ in $\mathbb{Z}_p - \{s\}$ the accumulation value $\mathsf{acc}(\mathcal{X})$ of $\mathcal{X}$ is defined as

$$\mathsf{acc}(\mathcal{X}) = g^{\prod_{x \in \mathcal{X}} (x+s)}.^{5}$$

Value $\mathsf{acc}(\mathcal{X})$ can be constructed using $\mathcal{X}$ and $g, g^s, g^{s^2}, \ldots, g^{s^q}$ (through polynomial interpolation), where $q \geq |\mathcal{X}|$. Subject to $\mathsf{acc}(\mathcal{X})$ each element in $\mathcal{X}$ has a succinct membership proof. More generally, the *proof of subset containment* of a set $\mathcal{S} \subseteq \mathcal{X}$—for $|\mathcal{S}| = 1$, this becomes a membership proof—is the *witness* $(\mathcal{S}, \mathsf{W}_{\mathcal{S},\mathcal{X}})$ where

$$\mathsf{W}_{\mathcal{S},\mathcal{X}} = g^{\prod_{x \in \mathcal{X} - \mathcal{S}} (x+s)}. \tag{1}$$

Subset containment of $\mathcal{S}$ in $\mathcal{X}$ can be checked through relation $e(\mathsf{W}_{\mathcal{S},\mathcal{X}}, g^{\prod_{x \in \mathcal{S}} (x+s)}) \stackrel{?}{=} e\left(\mathsf{acc}(\mathcal{X}), g\right)$ by any verifier with access only to public information. The security property of the bilinear-map accumulator, namely that computing fake but verifiable subset containment proofs is hard, can be proved using the *bilinear* q-strong Diffie-Hellman assumption, which is *slightly* stronger than the q-strong Diffie-Hellman assumption [8].[6]

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

[4] Function $f : \mathbb{N} \rightarrow \mathbb{R}$ is $\mathsf{neg}(k)$ if and only if for any nonzero polynomial $p(k)$ there exits N such that for all $k > N$ it is $f(k) < 1/p(k)$.

[5] $\prod_{x \in S_i} (x + s)$ is called *characteristic polynomial* of set S_i in the literature (e.g., see [25]).

[6] However, the plain q-strong Diffie-Hellman assumption [28] suffices to prove just the *collision resistance* of the bilinear-map accumulator.

We next prove the security of subset witnesses by generalizing the proof in [28]. Subset witnesses also appeared (independent of our work but without a proof) in [10].

Lemma 1 (Subset containment). *Let k be the security parameter and $(p, \mathbb{G}, \mathcal{G}, e, g)$ be a tuple of bilinear pairing parameters. Given the elements $g, g^s, \ldots, g^{s^q} \in \mathbb{G}$ for some s chosen at random from $\mathbb{Z}_p^*$ and a set of elements $\mathcal{X}$ in $\mathbb{Z}_p - \{s\}$ with $q \geq |\mathcal{X}|$, suppose there is a probabilistic polynomial-time algorithm that finds $\mathcal{S}$ and W such that $\mathcal{S} \nsubseteq \mathcal{X}$ and $e(\mathsf{W}, g^{\prod_{x \in \mathcal{S}}(x+s)}) = e(\mathsf{acc}(\mathcal{X}), g)$. Then there is a probabilistic polynomial-time algorithm that breaks the bilinear q-strong Diffie-Hellman assumption.*

Proof. Suppose there is a probabilistic polynomial-time algorithm that computes such a set $\mathcal{S} = \{y_1, y_2, \ldots, y_\ell\}$ and a fake witness W. Let $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ and $y_j \notin \mathcal{X}$ for some $1 \leq j \leq \ell$. This means that

$$e(\mathsf{W}, g)^{\prod_{y \in \mathcal{S}}(y+s)} = e(g, g)^{(x_1+s)(x_2+s)\ldots(x_n+s)}.$$

Note that $(y_j + s)$ does not divide $(x_1 + s)(x_2 + s) \ldots (x_n + s)$. Therefore there exist polynomial $Q(s)$ (computable in polynomial time) of degree $n - 1$ and constant $\lambda \neq 0$, such that $(x_1 + s)(x_2 + s) \ldots (x_n + s) = Q(s)(y_j + s) + \lambda$. Thus we have

$$e(\mathsf{W}, g)^{(y_j+s)\prod_{1 \leq i \neq j \leq \ell}(y_i+s)} = e(g, g)^{Q(s)(y_j+s)+\lambda} \Rightarrow$$
$$e(g, g)^{\frac{1}{y_j+s}} = \left[e(\mathsf{W}, g)^{\prod_{1 \leq i \neq j \leq \ell}(y_i+s)} e(g, g)^{-Q(s)}\right]^{\lambda^{-1}}.$$

Thus, this algorithm can break the bilinear q-strong Diffie-Hellman assumption. □

Tools for polynomial arithmetic. Our solutions use (modulo p) polynomial arithmetic. We next present two results that are extensively used in our techniques, contributing to achieve the desired complexity goals. The first result on polynomial interpolation is derived using an FFT algorithm (see Preparata and Sarwate [34]) that computes the DFT in a finite field (e.g., $\mathbb{Z}_p$) for arbitrary n and performing $O(n \log n)$ field operations. We note that an n-th root of unity is not required to exist in $\mathbb{Z}_p$ for this algorithm to work.

Lemma 2 (Polynomial interpolation with FFT [34]). *Let $\prod_{i=1}^{n}(x_i + s) = \sum_{i=0}^{n} b_i s^i$ be a degree-n polynomial. The coefficients $b_n \neq 0, b_{n-1}, \ldots, b_0$ of the polynomial can be computed with $O(n \log n)$ complexity, given $x_1, x_2, \ldots, x_n$.*

Lemma 2 refers to an efficient process for computing the coefficients of a polynomial, given its roots $x_1, x_2, \ldots, x_n$. In our construction, we make use of this process a numbers of times, in particular, when, given some values $x_1, x_2, \ldots, x_n$ to be accumulated, an untrusted party needs to compute $g^{(x_1+s)(x_2+s)\ldots(x_n+s)}$ without having access to s. However, access to $g, g^s, \ldots, g^{s^n}$ (part of the public key) is allowed, and therefore computing the accumulation value boils down to a polynomial interpolation.

We next present a second result that will be used in our verification algorithms. Related to *certifying algorithms* [21], this result states that if the vector of coefficients $\mathbf{b} = [b_n, b_{n-1}, \ldots, b_0]$ is claimed to be correct, then, given the vector of roots $\mathbf{x} = [x_1, x_2, \ldots, x_n]$, with high probability, vector $\mathbf{b}$ can be certified to be correct with complexity asymptotically less than $O(n \log n)$, i.e., without an FFT computation from scratch. This is achieved with the following algorithm:

Algorithm $\{\texttt{accept}, \texttt{reject}\} \leftarrow \mathsf{certify}(\mathbf{b}, \mathbf{x}, \mathsf{pk})$**:** The algorithm picks a random $\kappa \in \mathbb{Z}_p^*$. If $\sum_{i=0}^{n} b_i \kappa^i = \prod_{i=1}^{n}(x_i + \kappa)$, then the algorithm accepts, else it rejects.

Lemma 3 (Polynomial coefficients verification). *Let* $\mathbf{b} = [b_n, b_{n-1}, \ldots, b_0]$ *and* $\mathbf{x} = [x_1, x_2, \ldots, x_n]$. *Algorithm* $\mathsf{certify}(\mathbf{b}, \mathbf{x}, \mathsf{pk})$ *has* $O(n)$ *complexity. Also, if* $\texttt{accept} \leftarrow \mathsf{certify}(\mathbf{b}, \mathbf{x}, \mathsf{pk})$*, then* $b_n, b_{n-1}, \ldots, b_0$ *are the coefficients of the polynomial* $\prod_{i=1}^{n}(x_i + s)$ *with probability* $\Omega(1 - \mathsf{neg}(k))$.

Authenticated data structure scheme. We now define our *authenticated data structure scheme* (ADS scheme), as well as the correctness and security properties it must satisfy.

Definition 1 (ADS scheme). *Let* D *be* any *data structure that supports queries* q *and updates* u. *Let* $\mathsf{auth}(D)$ *denote the resulting authenticated data structure and* d *the digest of the authenticated data structure, i.e., a constant-size description of* D. *An ADS scheme* $\mathcal{A}$ *is a collection of the following six probabilistic polynomial-time algorithms:*

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

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

Definition 2 (Correctness). *Let* $\mathcal{ASC}$ *be an ADS scheme* $\{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$. *We say that the ADS scheme* $\mathcal{ASC}$ *is* correct *if, for all* $k \in \mathbb{N}$*, for all* $\{\mathsf{sk}, \mathsf{pk}\}$ *output by algorithm* genkey*, for all* $D_h, \mathsf{auth}(D_h), d_h$ *output by one invocation of* setup *followed by polynomially-many invocations of* $\mathsf{refresh}$*, where* $h \geq 0$*, for all queries* q *and for all* $\Pi(q), \alpha(q)$ *output by* $\mathsf{query}(q, D_h, \mathsf{auth}(D_h), \mathsf{pk})$*, with all but negligible probability, whenever algorithm* $\mathsf{check}(q, \alpha(q), D_h)$ *outputs* $\texttt{accept}$*, so does algorithm* $\mathsf{verify}(q, \Pi(q), \alpha(q), d_h, \mathsf{pk})$.

Definition 3 (Security). *Let $\mathcal{ASC}$ be an ADS scheme* $\{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$*, k be the security parameter, $\nu(k)$ be a negligible function and* $\{\mathsf{sk}, \mathsf{pk}\} \leftarrow \mathsf{genkey}(1^k)$*. Let also* Adv *be a probabilistic polynomial-time adversary that is only given* pk*. The adversary has unlimited access to all algorithms of $\mathcal{ASC}$, except for algorithms* setup *and* update *to which he has only oracle access. The adversary picks an initial state of the data structure D_0 and computes $D_0, \mathsf{auth}(D_0), d_0$ through oracle access to algorithm* setup*. Then, for $i = 0, \ldots, h = \mathsf{poly}(k)$,* Adv *issues an update u_i in the data structure D_i and computes D_{i+1}, $\mathsf{auth}(D_{i+1})$ and d_{i+1} through oracle access to algorithm* update*. Finally the adversary picks an index $0 \leq t \leq h+1$, and computes a query q, an answer α and a proof Π. We say that the ADS scheme $\mathcal{ASC}$ is* secure *if for all $k \in \mathbb{N}$, for all* $\{\mathsf{sk}, \mathsf{pk}\}$ *output by algorithm* genkey*, and for any probabilistic polynomial-time adversary* Adv *it holds that*

$$\Pr\left[\begin{array}{r}\{q, \Pi, \alpha, t\} \leftarrow \mathsf{Adv}(1^k, \mathsf{pk}); \mathtt{accept} \leftarrow \mathsf{verify}(q, \alpha, \Pi, d_t, \mathsf{pk}); \\ \mathtt{reject} \leftarrow \mathsf{check}(q, \alpha, D_t).\end{array}\right] \leq \nu(k)\,. \quad (2)$$

3 Construction and Algorithms

In this section we present an ADS scheme for set-operation verification. The underlying data structure for which we design our ADS scheme is called *sets collection*, and can be viewed as a generalization of the *inverted index* [4] data structure.

Sets collection. The *sets collection* data structure consists of m sets, denoted with $S_1, S_2, \ldots, S_m$, each containing elements from a universe $\mathcal{U}$. Without loss of generality we assume that the universe $\mathcal{U}$ is the set of nonnegative integers in the interval $[m+1, p-1] - \{s\}$,[7] where p is k-bit prime, m is the number of the sets in our collection that has bit size $O(\log k)$, k is the security parameter and s is the trapdoor of the scheme (see algorithm genkey). A set S_i does not contain duplicate elements, however an element $x \in \mathcal{U}$ can appear in more than one set. Each set is sorted and the total space needed is $O(m + M)$, where M is the sum of the sizes of the sets.

In order to get some intuition, we can view the sets collection as an *inverted index*. In this view, the elements are pointers to documents and each set S_i corresponds to a term w_i in the dictionary, containing the pointers to documents where term w_i appears. In this case, m is the number of terms being indexed, which is typically in the hundreds of thousands, while M, bounded from below by the number of documents being indexed, is typically in the billions. Thus, the more general terms "elements" and "sets" in a sets collection can be instantiated to the more specific "documents" and "terms".

The operations supported by the sets collection data structure consist of *updates* and *queries*. An update is either an *insertion* of an element into a set or a *deletion* of an element from a set. An update on a set of size n takes $O(\log n)$ time. For simplicity, we assume that the number m of sets does not change after updates. A query is one of the following standard set operations: (i) *Intersection:* Given indices $i_1, i_2, \ldots, i_t$, return set $\mathsf{I} = S_{i_1} \cap S_{i_2} \cap \ldots \cap S_{i_t}$; (ii) *Union:* Given indices $i_1, i_2, \ldots, i_t$, return set $\mathsf{U} = S_{i_1} \cup S_{i_2} \cup \ldots \cup S_{i_t}$; (iii) *Subset query:* Given indices i and j, return `true` if

[7] This choice simplifies the exposition; however, by using some collision-resistant hash function, universe $\mathcal{U}$ can be set to $\mathbb{Z}_p - \{s\}$.

$S_i \subseteq S_j$ and `false` otherwise; (iv) *Set difference:* Given indices i and j, return set $\mathsf{D} = S_i - S_j$. For the rest of the paper, we denote with δ the size of the answer to a query operation, i.e., δ is equal to the size of I, U, or D. For a subset query, δ is $O(1)$.

We next detail the design of an ADS scheme $\mathcal{ASC}$ for the *sets collection* data structure. This scheme provides protocols for verifying the integrity of the answers to set operations in a dynamic setting where sets evolve over time through updates. The goal is to achieve optimality in the communication and verification complexity: a query with t parameters and answer size δ should be verified with $O(t + \delta)$ complexity, and at the same time query and update algorithms should be efficient as well.

3.1 Setup and Updates

We describe an ADS scheme $\mathcal{ASC} = \{\mathsf{genkey}, \mathsf{setup}, \mathsf{update}, \mathsf{refresh}, \mathsf{query}, \mathsf{verify}\}$ for the sets collection data structure and we prove that its algorithms satisfy the complexities of Table 1. We begin with the algorithms that are related to the setup and the updates of the authenticated data structure.

Algorithm $\{\mathsf{sk}, \mathsf{pk}\} \leftarrow \mathsf{genkey}(1^k)$**:** Bilinear pairing parameters $(p, \mathbb{G}, \mathcal{G}, e, g)$ are picked and an element $s \in \mathbb{Z}_p^*$ is chosen at random. Subsequently, an one-to-one function $h(\cdot) : \mathbb{G} \rightarrow \mathbb{Z}_p^*$ is used. This function simply outputs the bit description of the elements of $\mathbb{G}$ according to some canonical representation of $\mathbb{G}$. Finally the algorithm outputs $\mathsf{sk} = s$ and $\mathsf{pk} = \{h(\cdot), p, \mathbb{G}, \mathcal{G}, e, g, \mathbf{g}\}$, where vector $\mathbf{g}$ contains values

$$\mathbf{g} = \left[g^s, g^{s^2}, \ldots, g^{s^q}\right] ,$$

where $q \geq \max\{m, \max_{i=1,\ldots,m}\{|S_i|\}\}$. The algorithm has $O(1)$ access complexity.

Algorithm $\{D_0, \mathsf{auth}(D_0), d_0\} \leftarrow \mathsf{setup}(D_0, \mathsf{sk}, \mathsf{pk})$**:** Let D_0 be our initial data structure, i.e., the one representing sets $S_1, S_2, \ldots, S_m$. The authenticated data structure $\mathsf{auth}(D_0)$ is built as follows. First, for each set S_i its accumulation value $\mathsf{acc}(S_i) = g^{\prod_{x \in S_i}(x+s)}$ is computed (see Section 2). Subsequently, the algorithm picks a constant $0 < \epsilon < 1$. Let T be a tree that has $l = \lceil 1/\epsilon \rceil$ levels and m leaves, numbered $1, 2, \ldots, m$, where m is the number of the sets of our sets collection data structure. Since T is a *constant-height* tree, the degree of any internal node of it is $O(m^\epsilon)$. We call such a tree an *accumulation tree*, which was originally introduced (combined with different cryptography) in [32]. For each node of the tree v, the algorithm recursively computes the *digest* $d(v)$ of v as follows. If v is a leaf corresponding to set S_i, where $1 \leq i \leq m$, the algorithm sets $d(v) = \mathsf{acc}(S_i)^{(i+s)}$; here, raising value $\mathsf{acc}(S_i)$ to exponent $i + s$, under the constraint that $i \leq m$, is done to also accumulate the index i of set S_i (and thus prove that $\mathsf{acc}(S_i)$ refers to S_i). If node v is not a leaf, then

$$d(v) = g^{\prod_{w \in \mathcal{N}(v)}(h(d(w)+s))} , \tag{3}$$

where $\mathcal{N}(v)$ denotes the set of children of node v. The algorithm outputs all the sets S_i as the data structure D_0, and all the accumulation values $\mathsf{acc}(S_i)$ for $1 \leq i \leq m$, the tree T and all the digests $d(v)$ for all $v \in T$ as the authenticated data structure $\mathsf{auth}(D_0)$. Finally, the algorithm sets $d_0 = d(r)$ where r is the root of T, i.e., d_0 is

the digest of the authenticated data structure (defined similarly as in a Merkle tree).[8] The access complexity of the algorithm is $O(m + M)$ (for postorder traversal of T and computation of $\mathsf{acc}(S_i)$), where $M = \sum_{i=1}^{m} |S_i|$. The group complexity of $\mathsf{auth}(D_0)$ is also $O(m + M)$ since the algorithm stores one digest per node of T, T has $O(m)$ nodes and there are M elements contained in the sets, as part of $\mathsf{auth}(D_0)$.

Algorithm $\{D_{h+1}, \mathsf{auth}(D_{h+1}), d_{h+1}, \mathsf{upd}\} \leftarrow \mathsf{update}(u, D_h, \mathsf{auth}(D_h), d_h, \mathsf{sk}, \mathsf{pk})$**:** We consider the update "insert element $x \in \mathcal{U}$ into set S_i" (note that the same algorithm could be used for element deletions). Let v_0 be the leaf node of T corresponding to set S_i. Let $v_0, v_1, \ldots, v_l$ be the path in T from node v_0 to the root of the tree, where $l = \lceil 1/\epsilon \rceil$. The algorithm initially sets $d'(v_0) = \mathsf{acc}(S_i)^{(x+s)}$, i.e., it updates the accumulation value that corresponds to the updated set (note that in the case where x is deleted from S_i, the algorithm sets $d'(v_0) = \mathsf{acc}(S_i)^{(x+s)^{-1}}$). Then the algorithm sets

$$d'(v_j) = d(v_j)^{(h(d'(v_{j-1}))+s)(h(d(v_{j-1}))+s)^{-1}} \text{ for } j = 1, \ldots, l\,, \tag{4}$$

where $d(v_{j-1})$ is the current digest of v_{j-1} and $d'(v_{j-1})$ is the updated digest of v_{j-1}.[9] All these newly computed values (i.e., the new digests) are stored by the algorithm. The algorithm then outputs the new digests $d'(v_{j-1})$, $j = 1, \ldots, l$, along the path from the updated set to the root of the tree, as part of information upd. Information upd also includes x and $d'(v_l)$. The algorithm also sets $d_{h+1} = d'(v_l)$, i.e., the updated digest is the newly computed digest of the root of T. Finally the new authenticated data structure $\mathsf{auth}(D_{h+1})$ is computed as follows: in the current authenticated data structure $\mathsf{auth}(D_h)$ that is input of the algorithm, the values $d(v_{j-1})$ are overwritten with the new values $d'(v_{j-1})$ ($j = 1, \ldots, l$), and the resulting structure is included in the output of the algorithm. The number of operations performed is proportional to $1/\epsilon$, therefore the complexity of the algorithm is $O(1)$.

Algorithm $\{D_{h+1}, \mathsf{auth}(D_{h+1}), d_{h+1}\} \leftarrow \mathsf{refresh}(u, D_h, \mathsf{auth}(D_h), d_h, \mathsf{upd}, \mathsf{pk})$**:** We consider the update "insert element $x \in \mathcal{U}$ into set S_i". Let v_0 be the node of T corresponding to set S_i. Let $v_0, v_1, \ldots, v_l$ be the path in T from node v_0 to the root of the tree. Using the information upd, the algorithm sets $d(v_j) = d'(v_j)$ for $j = 0, \ldots, l$, i.e., it updates the digests that correspond to the updated path. Finally, it outputs the updated sets collection as D_{h+1}, the updated digests $d(v_j)$ (along with the ones that belong to the nodes that are not updated) as $\mathsf{auth}(D_{h+1})$ and $d'(v_l)$ (contained in upd) as d_{h+1}.[10] The algorithm has $O(1)$ complexity as the number of performed operations is $O(1/\epsilon)$.

3.2 Authenticity of Accumulation Values

So far we have described the authenticated data structure $\mathsf{auth}(D_h)$ that our ADS scheme $\mathcal{ASC}$ will use for set-operation verifications. Overall, $\mathsf{auth}(D_h)$ comprises a set

[8] Digest $d(r)$ is a "secure" succinct description of the set collection data structure. Namely, the accumulation tree protects the integrity of values $\mathsf{acc}(S_i)$, $1 \leq i \leq m$, and each accumulation value $\mathsf{acc}(S_i)$ protects the integrity of the elements contained in set S_i.

[9] Note that these update computations are efficient because update has access to secret key s.

[10] Note that information upd is not *required* for the execution of $\mathsf{refresh}$, but is rather used for efficiency. Without access to upd, algorithm $\mathsf{refresh}$ could compute the updated values $d(v_j)$ using polynomial interpolation, which would have $O(m^{\epsilon} \log m)$ complexity (see Lemma 2).

of m accumulation values $\mathsf{acc}(S_i)$, one for each set S_i, $i = 1, \ldots, m$, and a set of $O(m)$ digests $d(v)$, one for each internal node v of the accumulation tree T. Our proof construction and verification protocols for set operations (described in Section 3.3) make use of the accumulation values $\mathsf{acc}(S_i)$ (subject to which subset-containment witnesses can be defined), and therefore it is required that the authenticity of each such value can be verified. Tree T serves this exact role by providing short correctness proofs for each value $\mathsf{acc}(S_i)$ stored at leaf i of T, this time subject to the (global) digest d_h stored at the root of T. We next provide the details related to proving the authenticity of $\mathsf{acc}(S_i)$.

The *correctness proof* Π_i of accumulation value $\mathsf{acc}(S_i)$, $1 \leq i \leq m$, is a collection of $O(1)$ bilinear-map accumulator witnesses (as defined in Section 2). In particular, Π_i is set to be the ordered sequence $\Pi = (\pi_1, \pi_2, \ldots, \pi_l)$, where π_j is the pair of the digest of node v_{j-1} and a witness that authenticates v_{j-1}, subject to node v_j, in the path $v_0, v_1, \ldots, v_l$ defined by leaf v_0 storing accumulation value $\mathsf{acc}(S_i)$ and the root v_l of T. Conveniently, π_j is defined as $\pi_j = (\beta_j, \gamma_j)$, where

$$\beta_j = d(v_{j-1}) \text{ and } \gamma_j = \mathsf{W}_{v_{j-1}(v_j)} = g^{\prod_{w \in \mathcal{N}(v_j) - \{v_{j-1}\}} (h(d(w))+s)}. \tag{5}$$

Note that π_j is the witness for a subset of *one* element, namely $h(d(v_{j-1}))$ (recall, $d(v_0) = \mathsf{acc}(S_i)^{(i+s)}$). Clearly, pair π_j has group complexity $O(1)$ and can be constructed using polynomial interpolation with $O(m^\epsilon \log m)$ complexity, by Lemma 2 and since v_j has degree $O(m^\epsilon)$. Since Π_i consists of $O(1)$ such pairs, we conclude that the proof Π_i for an accumulation value $\mathsf{acc}(S_i)$ can be constructed with $O(m^\epsilon \log m)$ complexity and has $O(1)$ group complexity. The following algorithms queryTree and verifyTree are used to formally describe the construction and respectively the verification of such correctness proofs. Similar methods have been described in [32].

Algorithm $\{\Pi_i, \alpha_i\} \leftarrow \mathsf{queryTree}(i, D_h, \mathsf{auth}(D_h), \mathsf{pk})$**:** Let $v_0, v_1, \ldots, v_l$ be the path of T from the node storing $\mathsf{acc}(S_i)$ to the root of T. The algorithm computes Π_i by setting $\Pi_i = (\pi_1, \pi_2, \ldots, \pi_l)$, where $\pi_j = (d(v_{j-1}), \mathsf{W}_{v_{j-1}(v_j)})$ and $\mathsf{W}_{v_{j-1}(v_j)}$ is given in Equation 5 and computed by Lemma 2. Finally, the algorithm sets $\alpha_i = \mathsf{acc}(S_i)$.

Algorithm $\{\mathtt{accept}, \mathtt{reject}\} \leftarrow \mathsf{verifyTree}(i, \alpha_i, \Pi_i, d_h, \mathsf{pk})$**:** Let the proof be $\Pi_i = (\pi_1, \pi_2, \ldots, \pi_l)$, where $\pi_j = (\beta_j, \gamma_j)$. The algorithm outputs `reject` if one of the following is true: (i) $e(\beta_1, g) \neq e(\alpha_i, g^i g^s)$; or (ii) $e(\beta_j, g) \neq e\left(\gamma_{j-1}, g^{h(\beta_{j-1})} g^s\right)$ for some $2 \leq j \leq l$; or (iii) $e(d_h, g) \neq e\left(\gamma_l, g^{h(\beta_l)} g^s\right)$. Otherwise, it outputs `accept`.

We finally provide some complexity and security properties that hold for the correctness proofs of the accumulated values. The following result is used as a building block to derive the complexity of our scheme and prove its security (Theorem 1).

Lemma 4. *Algorithm* queryTree *runs with* $O(m^\epsilon \log m)$ *access complexity and outputs a proof of* $O(1)$ *group complexity. Moreover algorithm* verifyTree *has* $O(1)$ *access complexity. Finally, for any adversarially chosen proof* Π_i *(*$1 \leq i \leq m$*), if* $\mathtt{accept} \leftarrow \mathsf{verifyTree}(i, \alpha_i, \Pi_i, d_h, \mathsf{pk})$*, then* $\alpha_i = \mathsf{acc}(S_i)$ *with probability* $\Omega(1 - \mathsf{neg}(k))$.

3.3 Queries and Verification

With the correctness proofs of accumulation values at hand, we complete the description of our scheme $\mathcal{ASC}$ by presenting the algorithms that are related to the construction

and verification of proofs attesting the correctness of set operations. These proofs are efficiently constructed using the authenticated data structure presented earlier, and they have optimal size $O(t+\delta)$, where t and δ are the sizes of the query parameters and the answer. In the rest of the section, we focus on the detailed description of the algorithms for an *intersection* and a *union* query, but due to space limitations, we omit the details of the *subset* and the *set difference* query. We note, however, that the treatment of the subset and set difference queries is analogous to that of the intersection and union queries.

The parameters of an intersection or a union query are t indices $i_1, i_2, \ldots, i_t$, with $1 \leq t \leq m$. To simplify the notation, we assume without loss of generality that these indices are $1, 2, \ldots, t$. Let n_i denote the size of set S_i ($1 \leq i \leq t$) and let $N = \sum_{i=1}^{t} n_i$. Note that the size δ of the intersection or union is always $O(N)$ and that operations can be performed with $O(N)$ complexity, by using a generalized merge.

Intersection query. Let $\mathsf{I} = S_1 \cap S_2 \cap \ldots \cap S_t = \{y_1, y_2, \ldots, y_\delta\}$. We express the correctness of the set intersection operation by means of the following two conditions:

$$\textbf{Subset condition: } \mathsf{I} \subseteq S_1 \wedge \mathsf{I} \subseteq S_2 \wedge \ldots \wedge \mathsf{I} \subseteq S_t\,; \tag{6}$$

$$\textbf{Completeness condition: } (S_1 - \mathsf{I}) \cap (S_2 - \mathsf{I}) \cap \ldots \cap (S_t - \mathsf{I}) = \emptyset\,. \tag{7}$$

The completeness condition in Equation 7 is necessary since set I must contain *all* the common elements. Given an intersection I, and for every set S_j, $1 \leq i \leq t$, we define the degree-n_j polynomial

$$P_j(s) = \prod_{x \in S_j - \mathsf{I}} (x + s)\,. \tag{8}$$

The following result is based on the extended Euclidean algorithm over polynomials and provides our core verification test for checking the correctness of set intersection.

Lemma 5. *Set I is the intersection of sets $S_1, S_2, \ldots, S_t$ if and only if there exist polynomials $q_1(s), q_2(s), \ldots, q_t(s)$ such that $q_1(s)P_1(s)+q_2(s)P_2(s)+\ldots+q_t(s)P_t(s) = 1$, where $P_j(s)$, $j = 1, \ldots, t$, are defined in Equation 8. Moreover, the polynomials $q_1(s), q_2(s), \ldots, q_t(s)$ can be computed with $O(N \log^2 N \log\log N)$ complexity.*

Using Lemmas 2 and 5 we next construct efficient proofs for both conditions in Equations 6 and 7. In turn, the proofs are directly used to define the algorithms query and verify of our ADS scheme $\mathcal{ASC}$ for *intersection* queries.

Proof of subset condition. For each set S_j, $1 \leq j \leq t$, the *subset witnesses* $\mathsf{W}_{\mathsf{I},j} = g^{P_j(s)} = g^{\prod_{x \in S_j - \mathsf{I}}(x+s)}$ are computed, each with $O(n_j \log n_j)$ complexity, by Lemma 2. (Recall, $\mathsf{W}_{\mathsf{I},j}$ serves as a proof that I is a subset of set S_j.) Thus, the total complexity for computing all t required subset witnesses is $O(N \log N)$, where $N = \sum_{i=1}^{t} n_i$.[11]

Proof of completeness condition. For each $q_j(s)$, $1 \leq j \leq t$, as in Lemma 5 satisfying $q_1(s)P_1(s) + q_2(s)P_2(s) + \ldots + q_t(s)P_t(s) = 1$, the *completeness witnesses* $\mathsf{F}_{\mathsf{I},j} = g^{q_j(s)}$ are computed, by Lemma 5 with $O(N \log^2 N \log\log N)$ complexity.

[11] This is because $\sum n_j \log n_j \leq \log N \sum n_j = N \log N$.

Algorithm $\{\Pi(q), \alpha(q)\} \leftarrow \mathsf{query}(q, D_h, \mathsf{auth}(D_h), \mathsf{pk})$ **(Intersection):** Query q consists of t indices $\{1, 2, \ldots, t\}$, asking for the intersection I of $S_1, S_2, \ldots, S_t$. Let $\mathsf{I} = \{y_1, y_2, \ldots, y_\delta\}$. Then $\alpha(q) = \mathsf{I}$, and the proof $\Pi(q)$ consists of the following parts.

1. *Coefficients* $b_\delta, b_{\delta-1}, \ldots, b_0$ of polynomial $(y_1 + s)(y_2 + s)\ldots(y_\delta + s)$ that is associated with the intersection $\mathsf{I} = \{y_1, y_2, \ldots, y_\delta\}$. These are computed with $O(\delta \log \delta)$ complexity (Lemma 2) and they have $O(\delta)$ group complexity.
2. *Accumulation values* $\mathsf{acc}(S_j)$, $j = 1, \ldots, t$, which are associated with sets S_j, along with their respective *correctness proofs* Π_j. These are computed by calling algorithm $\mathsf{queryTree}(j, D_h, \mathsf{auth}(D_h), \mathsf{pk})$, for $j = 1, \ldots, t$, with $O(tm^\epsilon \log m)$ total complexity and they have $O(t)$ total group complexity (Lemma 4).
3. *Subset witnesses* $\mathsf{W}_{\mathsf{I},j}$, $j = 1, \ldots, t$, which are associated with sets S_j and intersection I (see proof of subset condition). These are computed with $O(N \log N)$ complexity and have $O(t)$ total group complexity (Lemma 2).
4. *Completeness witnesses* $\mathsf{F}_{\mathsf{I},j}$, $j = 1, \ldots, t$, which are associated with polynomials $q_j(s)$ of Lemma 5 (see proof of completeness condition). These are computed with $O(N \log^2 N \log\log N)$ complexity and have $O(t)$ group complexity (Lemma 5).

Algorithm $\{\mathtt{accept}, \mathtt{reject}\} \leftarrow \mathsf{verify}(q, \alpha, \Pi, d_h, \mathsf{pk})$ **(Intersection):** Verifying the result of an intersection query includes the following steps.

1. First, the algorithm uses the coefficients $\mathbf{b} = [b_\delta, b_{\delta-1}, \ldots, b_0]$ and the answer $\alpha(q) = \{y_1, y_2, \ldots, y_\delta\}$ as an input to algorithm $\mathsf{certify}(\mathbf{b}, \alpha(q), \mathsf{pk})$, in order to certify the validity of $b_\delta, b_{\delta-1}, \ldots, b_0$. If certify outputs `reject`, the algorithm also outputs `reject`.[12] This step has $O(\delta)$ complexity (Lemma 3).
2. Subsequently, the algorithm uses the proof Π_j to verify the correctness of $\mathsf{acc}(S_j)$, by running algorithm $\mathsf{verifyTree}(j, \mathsf{acc}(S_j), \Pi_j, d_h, \mathsf{pk})$ for $j = 1, \ldots, t$. If, for some j, verifyTree running on $\mathsf{acc}(S_j)$ outputs `reject`, the algorithm also outputs `reject`. This step has $O(t)$ complexity (Lemma 4).
3. Next, the algorithm checks the subset condition:[13]

$$e\left(\prod_{i=0}^{\delta}\left(g^{s^i}\right)^{b_i}, \mathsf{W}_{\mathsf{I},j}\right) \stackrel{?}{=} e\left(\mathsf{acc}(S_j), g\right), \text{ for } j = 1, \ldots, t. \tag{9}$$

   If, for some j, the above check on subset witness $\mathsf{W}_{\mathsf{I},j}$ fails, the algorithm outputs `reject`. This step has $O(t + \delta)$ complexity.
4. Finally, the algorithm checks the completeness condition:

$$\prod_{j=1}^{t} e\left(\mathsf{W}_{\mathsf{I},j}, \mathsf{F}_{\mathsf{I},j}\right) \stackrel{?}{=} e(g, g). \tag{10}$$

   If the above check on the completeness witnesses $\mathsf{F}_{\mathsf{I},j}$, $1 \le j \le t$, fails, the algorithm outputs `reject`. Or, if this relation holds, the algorithm outputs `accept`, i.e., it accepts $\alpha(q)$ as the *correct* intersection. This step has $O(t)$ complexity.

[12] Algorithm certify is used to achieve optimal verification and avoid an $O(\delta \log \delta)$ FFT computation from scratch.

[13] Group element $\prod_{i=0}^{\delta} g^{s^i b_i} = g^{(y_1+s)(y_2+s)\ldots(y_\delta+s)}$ is computed once with $O(\delta)$ complexity.

Note that for Equation 10, it holds $\prod_{j=1}^{t} e\left(\mathsf{W}_{\mathsf{I},j}, \mathsf{F}_{\mathsf{I},j}\right) = e(g,g)^{\sum_{j=1}^{t} q_j(s)P_j(s)} = e(g,g)$ when all the subset witnesses $\mathsf{W}_{\mathsf{I},j}$, all the completeness witnesses $\mathsf{F}_{\mathsf{I},j}$ and all the sets accumulation values $\mathsf{acc}(S_j)$ have been computed *honestly*, since $q_1(s)P_1(s) + q_2(s)P_2(s) + \ldots + q_t(s)P_t(s) = 1$. This is a required condition for proving the correctness of our ADS scheme, as defined in Definition 2. We continue with the description of algorithms query and verify for the union query.

Union query. Let $\mathsf{U} = S_1 \cup S_2 \cup \ldots \cup S_t = \{y_1, y_2, \ldots, y_\delta\}$. We express the correctness of the set union operation by means of the following two conditions:

$$\textbf{Membership condition:} \quad \forall y_i \in \mathsf{U}\ \exists j \in \{1,2,\ldots,t\} : y_i \in S_j\,; \tag{11}$$

$$\textbf{Superset condition:} \quad (\mathsf{U} \supseteq S_1) \wedge (\mathsf{U} \supseteq S_2) \wedge \ldots \wedge (\mathsf{U} \supseteq S_t)\,. \tag{12}$$

The superset condition in Equation 12 is necessary since set U must exclude none of the elements in sets $S_1, S_2, \ldots, S_t$. We formally describe algorithms query and verify of our ADS scheme $\mathcal{ASC}$ for *union* queries.

Algorithm $\{\Pi(q), \alpha(q)\} \leftarrow \mathsf{query}(q, D_h, \mathsf{auth}(D_h), \mathsf{pk})$ **(Union):** Query q asks for the union U of t sets $S_1, S_2, \ldots, S_t$. Let $\mathsf{U} = \{y_1, y_2, \ldots, y_\delta\}$. Then $\alpha(q) = \mathsf{U}$ and the proof $\Pi(q)$ consists of the following parts. **(1)** *Coefficients* $b_\delta, b_{\delta-1}, \ldots, b_0$ of polynomial $(y_1 + s)(y_2 + s)\ldots(y_\delta + s)$ that is associated with the union $\mathsf{U} = \{y_1, y_2, \ldots, y_\delta\}$. **(2)** *Accumulation values* $\mathsf{acc}(S_j)$, $j = 1, \ldots, t$, which are associated with sets S_j, along with their respective *correctness proofs* Π_j, both output of algorithm $\mathsf{queryTree}(j, D_h, \mathsf{auth}(D_h), \mathsf{pk})$. **(3)** *Membership witnesses* W_{y_i,S_k} of y_i, $i = 1, \ldots, \delta$ (see Equation 1), which prove that y_i belongs to *some* set S_k, $1 \le k \le t$, and which are computed with $O(N \log N)$ total complexity and have $O(\delta)$ total group complexity (Lemma 2). **(4)** *Subset witnesses* $\mathsf{W}_{S_j,\mathsf{U}}$, $j = 1, \ldots, t$, which are associated with sets S_i and union U and prove that U is a superset of S_j, $1 \le k \le t$, and which are computed with $O(N \log N)$ total complexity and have $O(t)$ total group complexity (Lemma 2).

Algorithm $\{\mathtt{accept}, \mathtt{reject}\} \leftarrow \mathsf{verify}(q, \alpha, \Pi, d_h, \mathsf{pk})$**: (Union):** Verifying the result of a union query includes the following steps. **(1)** First, the algorithm uses $\mathbf{b} = [b_\delta, b_{\delta-1}, \ldots, b_0]$ and the answer $\mathsf{U} = \alpha(q) = \{y_1, y_2, \ldots, y_\delta\}$ as an input to algorithm $\mathsf{certify}(\mathbf{b}, \alpha(q), \mathsf{pk})$, in order to certify the validity of $b_\delta, b_{\delta-1}, \ldots, b_0$. **(2)** Subsequently, the algorithm uses the proofs Π_j to verify the correctness of $\mathsf{acc}(S_j)$, by using algorithm $\mathsf{verifyTree}(j, \mathsf{acc}(S_j), \Pi_j, d_h, \mathsf{pk})$ for $j = 1, \ldots, t$. If the verification fails for at least one of $\mathsf{acc}(S_j)$, the algorithm outputs `reject`. **(3)** Next, the algorithm verifies that each element y_i, $i = 1, \ldots, \delta$, of the reported union belongs to some set S_k, for some $1 \le k \le t$ ($O(\delta)$ complexity). This is done by checking that relation $e(\mathsf{W}_{y_i,S_k}, g^{y_i} g^{s}) = e(\mathsf{acc}(S_k), g)$ holds for all $i = 1, \ldots, \delta$; otherwise the algorithm outputs `reject`. **(4)** Finally, the algorithm verifies that all sets specified by the query are *subsets* of the union, by checking the following conditions:

$$e\left(\mathsf{W}_{S_j,\mathsf{U}}, \mathsf{acc}(S_j)\right) \stackrel{?}{=} e\left(\prod_{i=0}^{\delta} \left(g^{s^i}\right)^{b_i}, g\right)\,, \text{ for } j = 1, \ldots, t.$$

If any of the above checks fails, the algorithm outputs `reject`, otherwise, it outputs `accept`, i.e., U is accepted as the *correct* union.

Subset and set difference query. For a subset query (positive or negative), we use the property $S_i \subseteq S_j \Leftrightarrow \forall y \in S_i, y \in S_j$. For a set difference query we use the property

$$\mathsf{D} = S_i - S_j \Leftrightarrow \exists F : F \cup \mathsf{D} = S_i \wedge F = S_i \cap S_j \,.$$

The above conditions can both be checked in an operation-sensitive manner using the techniques we have presented before. We now give the main result in our work.

Theorem 1. *Consider a collection of m sets $S_1, \ldots, S_m$ and let $M = \sum_{i=1}^{m} |S_i|$ and $0 < \epsilon < 1$. For a query operation involving t sets, let N be the sum of the sizes of the involved sets, and δ be the answer size. Then there exists an ADS scheme $\mathcal{ASC} =$ $\{$*genkey, setup, update, refresh, query, verify$\}$ *for a sets collection data structure D with the following properties: (1) $\mathcal{ASC}$ is correct and secure according to Definitions 2 and 3 and based on the bilinear q-strong Diffie-Hellman assumption; (2) The access complexity of algorithm (i)* genkey *is $O(1)$; (ii)* setup *is $O(m + M)$; (iii)* update *is $O(1)$ outputting information* upd *of $O(1)$ group complexity; (iv)* refresh *is $O(1)$; (3) For all queries q (intersection/union/subset/difference), constructing the proof with algorithm* query *has $O(N \log^2 N \log\log N + tm^{\epsilon} \log m)$ access complexity, algorithm* verify *has $O(t+\delta)$ access complexity and the proof $\Pi(q)$ has $O(t+\delta)$ group complexity; (4) The group complexity of the authenticated data structure* auth(D) *is $O(m + M)$.*

4 Security, Protocols and Applications

In this section we give an overview of the security analysis of our ADS scheme, describe how it can be employed to provide verification protocols in the three-party [36] and two-party [30] authentication models, and finally discuss some concrete applications.

Security proof sketch. We provide some key elements of the security of our verification protocols focusing on set intersection queries. The security proofs of the other set operations share similar ideas. Let D_0 be a sets collection data structure consisting of m sets $S_1, S_2, \ldots, S_m$,[14] and consider our ADS scheme $\mathcal{ASC} = \{$genkey, setup, update, refresh, query, verify$\}$. Let k be the security parameter and let $\{\mathsf{sk}, \mathsf{pk}\} \leftarrow \mathsf{genkey}(1^k)$. The adversary is given the public key pk, namely $\{h(\cdot), p, \mathbb{G}, \mathcal{G}, e, g, g^s, \ldots, g^{s^q}\}$, and unlimited access to all the algorithms of $\mathcal{ASC}$, except for setup and update to which he only has oracle access. The adversary initially outputs the authenticated data structure auth(D_0) and the digest d_0, through an oracle call to algorithm setup. Then the adversary picks a polynomial number of updates u_i (e.g., insertion of an element x into a set S_r) and outputs the data structure D_i, the authenticated data structure auth(D_i) and the digest d_i through oracle access to update. Then he picks a set of indices $q = \{1, 2, \ldots, t\}$ (wlog), all between 1 and m and outputs a proof $\Pi(q)$ and an answer $\mathcal{I} \neq \mathsf{I} = S_1 \cap S_2 \cap \ldots \cap S_t$ which is rejected by check as incorrect. Suppose the answer $\alpha(q)$ contains d elements. The proof $\Pi(q)$ contains (i) *Some* coefficients

[14] Note here that since the sets are picked by the adversary, we have to make sure that no element in any set is equal to s, the trapdoor of the scheme (see definition of the bilinear-map accumulator domain). However, this event occurs with negligible probability since the sizes of the sets are polynomially-bounded and s is chosen at random from a domain of exponential size.

$b_0, b_1, \ldots, b_d$; (ii) *Some* accumulation values acc_j with *some* respective correctness proofs Π_j, for $j = 1, \ldots, t$; (iii) *Some* subset witnesses W_j with *some* completeness witnesses F_j, for $j = 1, \ldots, t$ (this is, what algorithm verify expects for input).

Suppose verify accepts. Then: (i) By Lemma 3, $b_0, b_1, \ldots, b_d$ are *indeed* the coefficients of the polynomial $\prod_{x \in \mathcal{I}}(x + s)$, except with negligible probability; (ii) By Lemma 4, values acc_j are *indeed* the accumulation values of sets S_j, except with negligible probability; (iii) By Lemma 1, values W_j are *indeed* the subset witnesses for set $\mathcal{I}$ (with reference to S_j), i.e., $\mathsf{W}_j = g^{P_j(s)}$, except with negligible probability; (iv) However, $P_1(s), P_2(s), \ldots, P_t(s)$ are not coprime since $\mathcal{I}$ is incorrect and therefore $\mathcal{I}$ cannot contain *all* the elements of the intersection. Thus the polynomials $P_1(s), P_2(s), \ldots, P_t(s)$ (Equation 8) have at least one common factor, say $(r + s)$ and it holds that $P_j(s) = (r + s)Q_j(s)$ for some polynomials $Q_j(s)$ (computable in polynomial time), for all $j = 1, \ldots, t$. By the verification of Equation 10 (completeness condition), we have

$$e(g,g) = \prod_{j=1}^{t} e\left(\mathsf{W}_j, \mathsf{F}_j\right) = \prod_{j=1}^{t} e\left(g^{P_j(s)}, \mathsf{F}_j\right) = \prod_{j=1}^{t} e\left(g^{(r+s)Q_j(s)}, \mathsf{F}_j\right)$$

$$= \prod_{j=1}^{t} e\left(g^{Q_j(s)}, \mathsf{F}_j\right)^{(r+s)} = \left(\prod_{j=1}^{t} e\left(g^{Q_j(s)}, \mathsf{F}_j\right)\right)^{(r+s)}.$$

Therefore we can derive an $(r + s)$-th root of $e(g, g)$ as

$$e(g,g)^{\frac{1}{r+s}} = \prod_{j=1}^{t} e\left(g^{Q_j(s)}, \mathsf{F}_j\right).$$

This means that if the intersection $\mathcal{I}$ is incorrect and all the verification tests are satisfied, we can derive a polynomial-time algorithm that outputs a bilinear q-strong Diffie-Hellman challenge $(r, e(g, g)^{1/(r+s)})$ for an element r that is a common factor of the polynomials $P_1(s), P_2(s), \ldots, P_t(s)$, which by Assumption 1 happens with probability $\mathsf{neg}(k)$. This concludes an ouline of the proof strategy for the case of intersection.

Protocols. As mentioned in the introduction, our ADS scheme $\mathcal{ASC}$ can be used by a verification protocol in the three-party model [36]. Here, a trusted entity, called source, owns a sets collection data structure D_h, but desires to outsource query answering, in a trustworthy (verifiable) way. The source runs genkey and setup and outputs the authenticated data structure $\mathsf{auth}(D_h)$ along with the digest d_h. The source subsequently signs the digest d_h, and it outsources $\mathsf{auth}(D_h)$, D_h, the digest d_h and its signature to some untrusted entities, called servers. On input a data structure query q (e.g., an intersection query) sent by clients, the servers use $\mathsf{auth}(D_h)$ and D_h to compute proofs $\Pi(q)$, by running algorithm query, and they return to the clients $\Pi(q)$ and the signature on d_h along with the answer $a(q)$ to q. Clients can verify these proofs $\Pi(q)$ by running algorithm verify (since they have access to the signature of d_h, they can verify that d_h is authentic). When there is an update in the data structure (issued by the source), the source uses algorithm update to produce the new digest d'_h to be used in next verifications, while the servers update the authenticated data structure through refresh.

Additionally, our ADS scheme $\mathcal{ASC}$ can also be used by a *non-interactive* verification protocol in the two-party model [30]. In this case, the source and the client coincide, i.e., the client issues both the updates and the queries, and it is required to keep only constant state, i.e., the digest of the authenticated data structure. Whenever there is an update by the client, the client retrieves a verifiable, constant-size portion of the authenticated data structure that is used for locally performing the update and for computing the new local state, i.e., the new digest. A non-interactive two-party protocol that uses an ADS scheme for a data structure D is directly comparable with the recent protocols for verifiable computing [1,12,16] for the functionalities offered by the data structure D, e.g., computation of intersection, union, etc. Due to space limitations, we defer the detailed description of these protocols to the full version of the paper.

Applications. First of all, our scheme can be used to verify *keyword-search queries* implemented by the *inverted index* data structure [4]: Each term in the dictionary corresponds to a set in our sets collection data structure which contains all the documents that include this term. A usual text query for terms m_1 and m_2 returns those documents that are included in both the sets that are represented by m_1 and m_2, i.e., their intersection. Moreover, the derived authenticated inverted index can be efficiently updated as well. However, sometimes in keyword searches (e.g., keyword searches in the email inbox) it is desirable to introduce a "second" dimension: For example, a query could be "return emails that contain terms m_1 and m_2 and which were received between time t_1 and t_2", where $t_1 < t_2$. We call this variant a *timestamped keyword-search*. One solution for verifying such queries could be to embed a timestamp in the documents (e.g., each email message) and have the client do the filtering locally, after he has verified—using our scheme—the intersection of the sets that correspond to terms m_1 and m_2. However, this approach is not operation-sensitive: The intersection can be bigger than the set output after the local filtering, making this solution inefficient. To overcome this inefficiency, we can use a *segment-tree* data structure [35], verifying in this way timestamped keyword-search queries efficiently with $O(t \log r + \delta)$ complexity, where r is the total number of timestamps we are supporting. This involves building a binary tree T on top of sets of messages sent at certain timestamps and requiring each internal node of T be the union of messages stored in its children. Finally, our method can be used for verifying equi-join queries over relational tables, which boil down to set intersections.

5 Conclusion

In this paper, we presented an authenticated data structure for the optimal verification of set operations. The achieved efficiency is mainly due to new, extended security properties of accumulators based on pairing-based cryptography. Our solution provides two important properties, namely *public verifiability* and *dynamic updates*, as opposed to existing protocols in the verifiable computing model that provide *generality* and *secrecy*, but *verifiability in a static, secret-key setting* only.

A natural question to ask is whether outsourced verifiable computations with *secrecy* and *efficient dynamic updates* are feasible. Analogously, it is interesting to explore whether other specific functionalities (beyond set operations) can be optimally and publicly verified. Finally, according to a recently proposed definition of optimality [33], our

construction is *nearly* optimal: verification and updates are optimal, but not queries. It is interesting to explore whether an *optimal* authenticated sets collection data structure exists, i.e., one that asymptotically matches the bounds of the plain sets collection data structure, reducing the query time from $O(N \log^2 N)$ to $O(N)$.

Acknowledgments. This work was supported in part by the U.S. National Science Foundation under grants CNS–1012060 and CNS–1012798 and by the Kanellakis Fellowship and the Center for Geometric Computing at Brown University, the RISCS Center at Boston University and NetApp. The authors thank Michael T. Goodrich for useful discussions. The views in this paper do not necessarily reflect the views of the sponsors.

References

1. Applebaum, B., Ishai, Y., Kushilevitz, E.: From secrecy to soundness: Efficient verification via secure computation. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010. LNCS, vol. 6198, pp. 152–163. Springer, Heidelberg (2010)
2. Atallah, M.J., Cho, Y., Kundu, A.: Efficient data authentication in an environment of untrusted third-party distributors. In: Int. Conference on Data Engineering (ICDE), pp. 696–704 (2008)
3. Au, M.H., Tsang, P.P., Susilo, W., Mu, Y.: Dynamic universal accumulators for DDH groups and their application to attribute-based anonymous credential systems. In: Fischlin, M. (ed.) CT-RSA 2009. LNCS, vol. 5473, pp. 295–308. Springer, Heidelberg (2009)
4. Baeza-Yates, R., Ribeiro-Neto, B.: Modern Information Retrieval. Addison-Wesley Publishing Company, Reading (1999)
5. Bellare, M., Micciancio, D.: A new paradigm for collision-free hashing: Incrementality at reduced cost. In: Fumy, W. (ed.) EUROCRYPT 1997. LNCS, vol. 1233, pp. 163–192. Springer, Heidelberg (1997)
6. Benabbas, S., Gennaro, R., Vahlis, Y.: Verifiable delegation of computation over large datasets. In: Int. Cryptology Conference, CRYPTO (2011)
7. Blum, M., Evans, W.S., Gemmell, P., Kannan, S., Naor, M.: Checking the correctness of memories. Algorithmica 12(2/3), 225–244 (1994)
8. Boneh, D., Boyen, X.: Short signatures without random oracles and the SDH assumption in bilinear groups. J. Cryptology 21(2), 149–177 (2008)
9. Boneh, D., Waters, B.: Conjunctive, subset, and range queries on encrypted data. In: Vadhan, S.P. (ed.) TCC 2007. LNCS, vol. 4392, pp. 535–554. Springer, Heidelberg (2007)
10. Canard, S., Gouget, A.: Multiple denominations in E-cash with compact transaction data. In: Sion, R. (ed.) FC 2010. LNCS, vol. 6052, pp. 82–97. Springer, Heidelberg (2010)
11. Chung, K.-M., Kalai, Y., Liu, F.-H., Raz, R.: Memory delegation. In: Int. Cryptology Conference, CRYPTO (2011)
12. Chung, K.-M., Kalai, Y., Vadhan, S.: Improved delegation of computation using fully homomorphic encryption. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 483–501. Springer, Heidelberg (2010)
13. Damgård, I., Triandopoulos, N.: Supporting non-membership proofs with bilinear-map accumulators. Cryptology ePrint Archive, Report 2008/538 (2008), `http://eprint.iacr.org/`
14. Dwork, C., Naor, M., Rothblum, G.N., Vaikuntanathan, V.: How efficient can memory checking be? In: Reingold, O. (ed.) TCC 2009. LNCS, vol. 5444, pp. 503–520. Springer, Heidelberg (2009)

15. Freedman, M.J., Nissim, K., Pinkas, B.: Efficient private matching and set intersection. In: Cachin, C., Camenisch, J.L. (eds.) EUROCRYPT 2004. LNCS, vol. 3027, pp. 1–19. Springer, Heidelberg (2004)
16. Gennaro, R., Gentry, C., Parno, B.: Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 465–482. Springer, Heidelberg (2010)
17. Goodrich, M.T., Tamassia, R., Hasic, J.: An efficient dynamic and distributed cryptographic accumulator. In: Chan, A.H., Gligor, V.D. (eds.) ISC 2002. LNCS, vol. 2433, pp. 372–388. Springer, Heidelberg (2002)
18. Goodrich, M.T., Tamassia, R., Schwerin, A.: Implementation of an authenticated dictionary with skip lists and commutative hashing. In: DARPA Information Survivability Conference and Exposition II (DISCEX II), pp. 68–82 (2001)
19. Goodrich, M.T., Tamassia, R., Triandopoulos, N.: Super-efficient verification of dynamic outsourced databases. In: Malkin, T. (ed.) CT-RSA 2008. LNCS, vol. 4964, pp. 407–424. Springer, Heidelberg (2008)
20. Goodrich, M.T., Tamassia, R., Triandopoulos, N.: Efficient authenticated data structures for graph connectivity and geometric search problems. Algorithmica 60(3), 505–552 (2011)
21. Kratsch, D., McConnell, R.M., Mehlhorn, K., Spinrad, J.P.: Certifying algorithms for recognizing interval graphs and permutation graphs. In: Symposium on Discrete Algorithms (SODA), pp. 158–167 (2003)
22. Li, J., Li, N., Xue, R.: Universal accumulators with efficient nonmembership proofs. In: Katz, J., Yung, M. (eds.) ACNS 2007. LNCS, vol. 4521, pp. 253–269. Springer, Heidelberg (2007)
23. Martel, C., Nuckolls, G., Devanbu, P., Gertz, M., Kwong, A., Stubblebine, S.G.: A general model for authenticated data structures. Algorithmica 39(1), 21–41 (2004)
24. Merkle, R.C.: A certified digital signature. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 218–238. Springer, Heidelberg (1990)
25. Minsky, Y., Trachtenberg, A., Zippel, R.: Set reconciliation with nearly optimal communication complexity. IEEE Transactions on Information Theory 49(9), 2213–2218 (2003)
26. Morselli, R., Bhattacharjee, S., Katz, J., Keleher, P.J.: Trust-preserving set operations. In: Int. Conference on Computer Communications, INFOCOM (2004)
27. Naor, M., Nissim, K.: Certificate revocation and certificate update. In: USENIX Security Symposium, pp. 217–228 (1998)
28. Nguyen, L.: Accumulators from bilinear pairings and applications. In: Menezes, A. (ed.) CT-RSA 2005. LNCS, vol. 3376, pp. 275–292. Springer, Heidelberg (2005)
29. Pang, H., Tan, K.-L.: Authenticating query results in edge computing. In: Int. Conference on Data Engineering (ICDE), pp. 560–571 (2004)
30. Papamanthou, C., Tamassia, R.: Time and space efficient algorithms for two-party authenticated data structures. In: Qing, S., Imai, H., Wang, G. (eds.) ICICS 2007. LNCS, vol. 4861, pp. 1–15. Springer, Heidelberg (2007)
31. Papamanthou, C., Tamassia, R.: Cryptography for efficiency: Authenticated data structures based on lattices and parallel online memory checking. In: Cryptology ePrint Archive, Report 2011/102 (2011), http://eprint.iacr.org/
32. Papamanthou, C., Tamassia, R., Triandopoulos, N.: Authenticated hash tables. In: Int. Conference on Computer and Communications Security (CCS), pp. 437–448 (2008)
33. Papamanthou, C., Tamassia, R., Triandopoulos, N.: Optimal authenticated data structures with multilinear forms. In: Joye, M., Miyaji, A., Otsuka, A. (eds.) Pairing 2010. LNCS, vol. 6487, pp. 246–264. Springer, Heidelberg (2010)
34. Preparata, F.P., Sarwate, D.V.: Computational complexity of Fourier transforms over finite fields. Mathematics of Computation 31(139), 740–751 (1977)

35. Preparata, F.P., Shamos, M.I.: Computational Geometry: An Introduction. Springer, New York (1985)
36. Tamassia, R.: Authenticated data structures. In: Di Battista, G., Zwick, U. (eds.) ESA 2003. LNCS, vol. 2832, pp. 2–5. Springer, Heidelberg (2003)
37. Tamassia, R., Triandopoulos, N.: Certification and authentication of data structures. In: Alberto Mendelzon Workshop on Foundations of Data Management (2010)
38. Yang, Y., Papadias, D., Papadopoulos, S., Kalnis, P.: Authenticated join processing in outsourced databases. In: Int. Conf. on Management of Data (SIGMOD), pp. 5–18 (2009)
39. Yiu, M.L., Lin, Y., Mouratidis, K.: Efficient verification of shortest path search via authenticated hints. In: Int. Conference on Data Engineering (ICDE), pp. 237–248 (2010)