Verifiable Set Operations over Outsourced Databases*

Ran Canetti[1,2], Omer Paneth[1], Dimitrios Papadopoulos[1], and Nikos Triandopoulos[3,1]

[1] Dept. of Computer Science, Boston University, USA
[2] Dept. of Computer Science, Tel Aviv University, Israel
[3] RSA Laboratories, Cambridge MA, USA

Abstract. We study the problem of verifiable delegation of computation over outsourced data, whereby a powerful worker maintains a large data structure for a weak client in a verifiable way. Compared to the well-studied problem of verifiable computation, this setting imposes additional difficulties since the verifier also needs to check the consistency of updates succinctly and without maintaining large state. We present a scheme for verifiable evaluation of *hierarchical set operations* (unions, intersections and set-differences) applied to a collection of *dynamically changing sets of elements* from a given domain. The verification cost incurred is proportional only to the size of the final outcome set and to the size of the query, and is *independent of the cardinalities of the involved sets*. The cost of updates is *optimal* (involving $O(1)$ modular operations per update). Our construction extends that of [Papamanthou et al., CRYPTO 2011] and relies on a modified version of the *extractable collision-resistant hash function* (ECRH) construction, introduced in [Bitansky et al., ITCS 2012] that can be used to succinctly hash univariate polynomials.

1 Introduction

Outsourcing of storage and computation to the cloud has become a common practice for both enterprises and individuals. In this setting, typically, a client with bounded computational and storage capabilities wishes to outsource its database to a cloud provider and, over time, issue queries over the database that are answered by powerful servers.

We consider a client that outsources a dataset D to a server. The client can then issue to the server *informational* queries that are answered according to D, or it can issue *update* queries that change D, for example by inserting or removing elements. This model captures a variety of real-world applications such as outsourced relational databases, streaming datasets and outsourced file systems. We also consider the more general setting where multiple other clients can issue informational queries to D, while only one designated source client can issue update queries. For example, consider a company that outsources its data to a cloud service provider that will also be responsible for accommodating queries from the company's multiple customers.

* Research supported in part by the Check Point Institute for Information Security, an NSF EAGER grant, an NSF Algorithmic foundations grant 1218461, the Simons award for graduate students in theoretical computer science, and NSF grants CNS-1012798 and CNS-1012910.

H. Krawczyk (Ed.): PKC 2014, LNCS 8383, pp. 113–130, 2014.

In such outsourcing scenarios, clients may want to verify the integrity of the outsourced operations over the dataset D to protect themselves against servers that provide wrong results because they are themselves malicious or have been compromised by an external attacker, or simply provide false results (e.g., inaccurate or inconsistent data) due to bugs. Specifically, when answering a client's query, the server will also compute a *proof* of the integrity of the data used to compute the answer as well as the integrity of the computation, i.e., that the correct function was computed. For this purpose, we allow the source client to perform some preprocessing on D before outsourcing it to the server, and to save a small *verification state* that allows it to verify the server's proofs. Analogously, when issuing an update query, the source client will also update its verification state. If the verification state can be made public we say that the server's proofs are *publicly verifiable*, which is particularly important in the multi-client setting.

Several different measures of efficiency can be considered in this setting. First, the time it takes for the client to verify a proof should be short, ideally some fixed polynomial in the security parameter that is independent of the size of server's computation cost and the size of D. Second, the server's computational overhead in computing proofs should be kept minimal. Additional efficiency properties include small proof sizes, efficient update queries as well as non-interactive solutions where the client sends a query and receives back an answer and a proof in one round of interaction.

Set Operations over Outsourced Databases. This work focuses on the problem of verifying *general set operations* in the above outsourcing setting. That is, we consider a dataset D that consists of m sets $S_1, ..., S_m$, where the clients' queries are arbitrary set operations over D represented as formulas of union, intersection, and set-difference gates over some of the inputs $S_1, ..., S_m$. A particularly interesting case is when the sets appearing at intermediate steps of the computation are much larger than the final answer (e.g., consider a number of unions, followed by an intersection resulting in the empty set). The motivation for set operations comes from their great expressiveness and the range of computations that can be mapped by them. Real-world applications of general set operations include a wide class of SQL database queries, keyword search with elaborate queries, access control management and similarity measurement, hence a practical protocol for verifiable general set operations would be of great importance.

Verifiable Computation - The Generic Approach. The settings considered here are closely related to the setting of verifiable computation that has been extensively studied in recent years. In verifiable computation the client outsources a computation to the server and receives an answer that can be quickly verified. The main difference is that in verifiable computation it is usually assumed that the input to the computation is short and known to both parties, while in our settings the server's answers are computed over the outsourced dataset that must also be authenticated. This problem was addressed in the work of [15] on *memory delegation* with a construction based on Micali's CS proofs. One possible approach for designing a *practical* protocol is based on the memory delegation solution where Micali's CS proofs are replaced by a *succinct non-interactive argument-of-knowledge* (SNARK). Good candidates for more practical constructions of such a SNARK are provided in the recent works of [4, 6, 27].

However, one major obstacle for implementing the generic approach described above (discussed already in [27]) is that it only considers computations that are represented as

boolean or arithmetic circuits. For example, in the context of set operations the transformation from formulas of set operations to circuits can be extremely wasteful as the number of sets participating in every query and the set sizes may vary dramatically between queries. Here, another source of inefficiency is that the generic approach considers a universal circuit that gets the query, in the form of the set-operation formula, as input which introduces additional overheads. Overall, while asymptotically the computational overhead of the server can be made poly-logarithmic, in practice the large constants involved can be an obstacle for using the generic solution for set operations.

Our Result. In this work we propose a new practical scheme for publicly verifiable secure delegation of general set operations. The verification state is of constant size and the proof verification time is $O(t + \delta)$ where t is the size of the query formula and δ is the answer set size. That is, a main advantage of our scheme is that the verification time and the proof length do not grow with the sizes of all other sets involved in the computation. For instance, the intersection of two unions, each defined over a constant number of sets each having a large cardinality, may result in intermediate results of size $O(|D|)$ but only produce the empty set as output; in this extreme case, our scheme provides optimal, constant-time verification. The dependence on the answer size is inherent since the client must receive the answer set from the server. Another advantage of our scheme over the generic approach is that is does not involve translating the problem to an arithmetic or boolean circuit. In particular, the server will need to perform only $4N$ exponentiations in a group with a symmetric bilinear pairing, where N is the sum of the sizes of all the intermediate sets in the evaluation of the set formula.

For updates, the source client maintains an update state of length $O(m)$, where m is the number of sets in the dataset, and it can add or remove a single element for every set in constant time. The source then updates the server and all other clients with a new verification state. We note that our definitions and construction can be extended to support also *server-assisted updates*, where the source client updates a given set in D to a new set defined as the output of a set operation performed by the server, thus updating a large number of elements at once—details are deferred to the full version [13].

Overview of Techniques. The starting point for our construction is the scheme of Papamanthou, Tamassia and Triandopoulos [26] that supports verification of a single set operation, one union or intersection, over t sets in time $O(t + \delta)$, where δ is the answer size. The "naive" way to extend that scheme to support general set-operation formulas is to have the server provide a separate proof for each intermediate set produced in the evaluation of the formula. However, proving the security of such an extended scheme is problematic. The problem is that in the scheme of [26] the proofs do not necessarily compose. In particular, it might be easy for a malicious server to come up with a false proof corresponding to an incorrect answer set without "knowing" what this incorrect answer is (if the malicious server would be able to also find the answer set, the scheme of [26] would not have been secure). Therefore, to make the security proof of the naive scheme go though, the server would also have to prove to the client that it "knows" all the intermediate sets produced in the evaluation of the query formula. One way for the server to prove knowledge of these sets is to send them to the client, however, this will result in a proof that is as long as the entire server computation.

To solve this problem we need to further understand the structure of the proofs in [26] which is based on the notion of a bilinear accumulator [24]. We can think of a bilinear accumulator as a succinct hash of a large set that makes use of a representation of a set by its *characteristic polynomial* (i.e., a polynomial that has as roots the set elements). The main idea in our work is to use a different type of accumulator, a *knowledge accumulator*, that has "knowledge" properties, i.e., the only way for an algorithm to produce a valid accumulation value is to "know" the set that corresponds to this value. This knowledge property of our accumulator together with the soundness of the proof for every single operation allows us to prove the soundness of the composed scheme. Our construction of knowledge accumulators is very similar to the previous constructions of knowledge commitments in [6, 20], which are based on the q-PKE assumption, a variant of the knowledge-of-exponent assumption [16]. We capture the knowledge properties of our accumulator by using the notion of an *extractable collision-resistant hash function* (ECRH), originally introduced in [6]. However, we follow the weaker definition of ECRH with respect to auxiliary input, for which the recent negative evidence presented in [7] does not apply and the auxiliary-input distributions we consider here are not captured by the negative result of [11] either.

We also need to change the way a single set operation is proven. Specifically, in [26], a proof for a single union of sets requires one accumulation value for every element in the union. This will again result in a proof that is as long as the entire server computation. Instead our scheme involves proofs that are independent of the set sizes.

Moreover, in order to verify a proof in our scheme, the client only needs to know the accumulation values for the sets that participate in a computation. Instead of storing the accumulation values of all sets in the dataset, the client only stores a constant-size verification state that contains a special hash of these accumulation values. We compute this special hash using an *accumulation tree*, introduced in [25]. This primitive can be thought of as a special "tree hash" that makes use of the algebraic structure of accumulators to provide authentication paths of constant length.

Finally we note that our definition of security follows the popular framework of *authenticated data structures* introduced in [29].

Related Work. The very recent work of [3] also considers a practical secure database delegation scheme supporting a restricted class of queries, namely functions expressed by arithmetic circuits of degree up to 2. This scheme is based on homomorphic MACs and appears practical while also having a security proof that is based on standard hardness assumptions. However, their solution is only privately verifiable and it does not support deletions from the dataset. In a sense, the work of [3] is complementary to ours, as arithmetic and set operations are two desirable classes of computations for a database outsourcing scheme.

With respect to set operations, previous works focused mostly on the aspect of privacy and less on the aspect of integrity [2, 10, 18, 21]. There exists a number of works from the database community that address this problem [22, 30], but to the best of our knowledge, this is the first work that directly addresses the case of nested operations.

Characteristic polynomials for set representation have been used before in the cryptography literature (see for example [24, 26]) and this directly relates our work with a line of publications coming from the *cryptographic accumulators* literature [12, 24].

Indeed our ECRH construction, viewed as a mathematical object, is identical to a pair of bilinear accumulators (introduced in [24]) with related secret key values. Our ECRH can be viewed as an extractable extension to the bilinear accumulator that allows an adversarial party to prove knowledge of a subset to an accumulated set (without explicitly providing said subset). It also allows us to use the notion of *accumulation trees* which was originally defined for bilinear accumulators.

The *authenticated data structure* (ADS) paradigm, originally introduced in [29], appears extensively both in the cryptography and databases literature (see for example [1, 19, 22, 23, 26, 31, 32]). A wide range of functionalities has been authenticated in this context including range queries and basic SQL joins.

2 Tools and Definitions

We denote with l the security parameter and with $\nu(l)$ a negligible function. We say that an event can occur with negligible probability if its occurrence probability is upper bounded by a negligible function. Respectively, an event takes place with overwhelming probability if its complement takes place with negligible probability. In our technical exposition we adopt the *access complexity* model: Used mainly in the memory checking literature [8, 17], this model allows us to measure complexity expressed in the number of primitive cryptographic operations made by an algorithm without considering the related security parameter.

Bilinear Pairings. Let $\mathbb{G}$ be a cyclic multiplicative group of prime order p, generated by g. Let also $\mathbb{G}_T$ be a cyclic multiplicative group with the same order p and $e : \mathbb{G} \times \mathbb{G} \rightarrow \mathbb{G}_T$ be a bilinear pairing with the following properties: (1) Bilinearity: $e(P^a, Q^b) = e(P, Q)^{ab}$ for all $P, Q \in \mathbb{G}$ and $a, b \in \mathbb{Z}_p$; (2) Non-degeneracy: $e(g, g) \neq 1$; (3) Computability: There is an efficient algorithm to compute $e(P, Q)$ for all $P, Q \in \mathbb{G}$. We denote with $pub := (p, \mathbb{G}, \mathbb{G}_T, e, g)$ the bilinear pairings parameters, output by a randomized polynomial-time algorithm $\mathsf{GenBilinear}$ on input 1^l.

For cleaner presentation, in what follows we assume a symmetric (Type 1) pairing e. In [13] we discuss the modifications needed to implement our construction in the (more efficient) asymmetric pairing case (see [14] for a general discussion of pairings).

Our security analysis makes use of the following two assumptions :

Assumption 1 (q-Strong Bilinear Diffie-Hellman [9]). *For any poly-size adversary $\mathcal{A}$ and for q being a parameter of size $poly(l)$, the following holds:*

$$\Pr\left[\begin{array}{c} pub \leftarrow \mathsf{GenBilinear}(1^l); s \leftarrow_R \mathbb{Z}_p^*; \\ (z, \gamma) \in \mathbb{Z}_p^* \times \mathbb{G}_T \leftarrow \mathcal{A}(pub, (g, g^s, ..., g^{s^q})) \text{ s.t. } \gamma = e(g, g)^{1/(z+s)} \end{array}\right] \leq \nu(l)] \ .$$

Assumption 2 (q-Power Knowledge of Exponent [20]). *For any poly-size adversary $\mathcal{A}$, there exists a poly-size extractor $\mathcal{E}$ such that:*

$$\Pr\left[\begin{array}{c} pub \leftarrow \mathsf{GenBilinear}(1^l); a, s \leftarrow_R \mathbb{Z}_p^*; \sigma = (g, g^s, ..., g^{s^q}, g^a, g^{as}, ..., g^{as^q}) \\ (c, \tilde{c}) \leftarrow \mathcal{A}(pub, \sigma); (a_0, ..., a_n) \leftarrow \mathcal{E}(pub, \sigma) \\ \text{s.t. } e(\tilde{c}, g) = e(c, g^a) \quad \wedge \quad c \neq \prod_{i=0}^{n} g^{a_i s^i} \text{ for } n \leq q \end{array}\right] \leq \nu(l) \ .$$

Extractable Collision-Resistant Hash Functions. These functions (or ECRH for short) were introduced in [6] as a strengthening of the notion of collision-resistant hash functions. The key property implied by an ECRH is the hardness of oblivious sampling from the image space. Informally, for a function f, sampled from an ECRH function ensemble, any adversary producing a hash value h must have knowledge of a value $x \in Dom(f)$ s.t. $f(x) = h$. Formally, an ECRH function is defined as follows:

Definition 1 (ECRH [6]). *A function ensemble $\mathcal{H} = \{\mathcal{H}_l\}_l$ from $\{0,1\}^{t(l)}$ to $\{0,1\}^l$ is an ECRH if:*

Collision-Resistance. *For any poly-size adversary $\mathcal{A}$:*

$$\Pr_{h \leftarrow \mathcal{H}_l} \left[x, x' \leftarrow \mathcal{A}(1^l, h) \text{ s.t. } h(x) = h(x') \wedge x \neq x' \right] \leq \nu(l) \ .$$

Extractability. *For any poly-size adversary $\mathcal{A}$, there exists poly-size extractor $\mathcal{E}$ s.t.:*

$$\Pr_{h \leftarrow \mathcal{H}_l} \left[\begin{array}{c} y \leftarrow \mathcal{A}(1^l, h); x' \leftarrow \mathcal{E}(1^l, h) \\ \text{s.t. } \exists x : h(x) = y \wedge h(x') \neq y \end{array} \right] \leq \nu(l) \ .$$

An ECRH Construction from q-PKE. We next provide an ECRH construction from the q-PKE assumption defined above. In [6] the authors suggest that an ECRH can be constructed directly from q-PKE (without explicitly providing the construction). Here we present the detailed construction and a proof of the required properties with respect to q-PKE for extractability and q-SBDH for collision-resistance.

- To sample from $\mathcal{H}_l$, choose $q \in O(poly(l))$, run algorithm $\mathsf{GenBilinear}(1^l)$ to generate bilinear pairing parameters $pub = (p, \mathbb{G}, \mathbb{G}_T, e, g)$ and sample $a, s \leftarrow_R \mathbb{Z}_p^* \times \mathbb{Z}_p^*$ s.t. $a \neq s$. Output public key $pk = (pub, g^s, ..., g^{s^q}, g^a, g^{as}, ..., g^{as^q})$ and trapdoor information $sk = (s, a)$. It should be noted that the pk fully describes the chosen function h. Trapdoor sk can be used for a more efficient computation of hash values, by the party initializing the ECRH .
- To compute a hash value on $\mathbf{x} = (x_1, ..., x_q)$, output $h(\mathbf{x}) = \left(\prod_{i \in [q]} g^{x_i s^i}, \prod_{i \in [q]} g^{a x_i s^i} \right)$.

Lemma 1. *If the q-SBDH and q-PKE assumptions hold, the above is a $(q \cdot l, 4l)$-compressing ECRH.*

Proof. Extractability follows directly from the q-PKE assumption. To argue about collision-resistance, assume there exists adversary $\mathcal{A}$ outputting with probability ϵ, $(\mathbf{x}, \mathbf{y})$ such that there exists $i \in [q]$ with $x_i \neq y_i$ and $h(\mathbf{x}) = h(\mathbf{y})$. We denote with $P(r)$ the q-degree polynomial from $\mathbb{Z}_p[r]$, $\sum_{i \in [q]} (x_i - y_i) r^i$. From the above, it follows that $\sum_{i \in [q]} x_i s^i = \sum_{i \in [q]} y_i s^i$. Hence, while $P(r)$ is not the 0-polynomial, the evaluation of $P(r)$ at point s is $P(s) = 0$ and s is a root of $P(r)$. By applying a randomized polynomial factorization algorithm as in [5], one can extract the (up to q) roots of $P(r)$ with overwhelming probability, thus computing s. By choosing a to compute the second part of the public key to run $\mathcal{A}$ and then randomly selecting $c \in \mathbb{Z}_p^*$ and computing $\beta = g^{1/(c+s)}$ one can output $(c, e(g, \beta))$, breaking the q-SBDH with

probability $\epsilon(1-\epsilon')$ where ϵ' is the negligible probability of error in the polynomial factoring algorithm. Therefore any poly-size $\mathcal{A}$ can find a collision only with negligible probability. The $4l$ factor follows from the representation cost of elliptic curve points as a pair of p-bit coefficients. □

One natural application for the above ECRH construction would be the compact computational representation of polynomials from $\mathbb{Z}_p[r]$ of degree $\leq q$. A polynomial $P(r)$ with coefficients $p_1, ..., p_q$ can be succinctly represented by the hash value $h(P) = (f, f') = \left(\prod_{i\in[q]} g^{p_i s^i}, \prod_{i\in[q]} g^{a p_i s^i}\right)$.

Authenticated Data Structure Scheme. Such schemes, originally defined in [26], model verifiable computations over outsourced data structures. Let D be any data structure supporting queries and updates. We denote with $auth(D)$ some authenticated information on D and with d the digest of D, i.e., a succinct secure computational description of D. An authenticated data structure scheme ADS is a collection of six algorithms shown in Figure 1. Let $\{\text{accept, reject}\} = \textit{check}(q, a(q), D_h)$ be a method that decides whether $a(q)$ is a correct answer for query q on data structure D_h (this method is not part of the scheme but only introduced for ease of notation.) Then an authenticated data structure scheme ADS should satisfy the following:

1. $\{sk, pk\} \leftarrow$ **genkey**(1^k). Outputs secret and public keys, given the security parameter l.
2. $\{auth(D_0), d_0\} \leftarrow$ **setup**(D_0, sk, pk): Computes the authenticated data structure $auth(D_0)$ and its respective digest, d_0, given data structure D_0, the secret key sk and the public key pk.
3. $\{auth(D_{h+1}), d_{h+1}, upd\} \leftarrow$ **update**$(u, auth(D_h), d_h, sk, pk)$: On input update u on data structure D_h, the authenticated data structure $auth(D_h)$ and the digest d_h, it outputs the updated data structure D_{h+1} along with $auth(D_{h+1})$, the updated digest d_{h+1} and some relative information upd. It requires the secret key for execution.
4. $\{D_{h+1}, auth(D_{h+1}), d_{h+1}\} \leftarrow$ **refresh**$(u, D_h, auth(D_h), d_h, upd, pk)$: On input update u on data structure D_h, the authenticated data structure $auth(D_h)$, the digest d_h and relative information upd output by **update**, it outputs the updated data structure D_{h+1} along with $auth(D_{h+1})$ and the updated digest d_{h+1}, without access to sk.
5. $\{a(q), \Pi(q)\} \leftarrow$ **query**$(q, D_h, auth(D_h), pk)$: On input query q on data structure D_h and $auth(D_h)$ it returns the answer to the query $a(q)$, along with a proof $\Pi(q)$.
6. $\{\text{accept, reject}\} \leftarrow$ **verify**$(q, a(q), \Pi(q), d_h, pk)$: On input query q, an answer $a(q)$, a proof $\Pi(q)$, a digest d_h and pk, it outputs either "accept" or "reject".

Fig. 1. Authenticated data structure

Correctness. We say that ADS is *correct* if, for all $l \in \mathbb{N}$, for all (sk, pk) output by algorithm **genkey**, for all $(D_h, auth(D_h), d_h)$ output by one invocation of **setup** followed by polynomially-many invocations of **refresh**, where $h \geq 0$, for all queries q and for all $a(q), \Pi(q)$ output by $query(q, D_h, auth(D_h), pk)$, with all but negligible probability, whenever $\textit{check}(q, a(q), D_h)$ accepts, so does **verify**$(q, a(q), \Pi(q), d_h, pk)$.

Security. Let $l \in \mathbb{N}$ be a security parameter and $(sk, pk) \leftarrow genkey(1^l)$ and $\mathcal{A}$ be a poly-size adversary that is only given pk and has oracle access to all algorithms of the ADS. The adversary picks an initial state of the data structure D_0 and computes $D_0, auth(D_0), d_0$ through oracle access to algorithm **setup**. Then, for $i = 0, ..., h = poly(l)$, $\mathcal{A}$ issues an update u_i for the data structure D_i and outputs $D_{i+1}, auth(D_{i+1})$ and d_{i+1} through oracle access to algorithm **update**. At any point during these update queries, he can make polynomially many oracle calls to algorithms **prove** and **verify**. Finally the adversary picks an index $0 \leq t \leq h+1$, a query q, an answer $a(q)$ and a proof $\Pi(q)$. We say that an ADS is *secure* if for all large enough $k \in \mathbb{N}$, for all poly-size adversaries $\mathcal{A}$ it holds that:

$$\Pr\left[\begin{array}{c}(q, a(q), \Pi(q), t) \leftarrow \mathcal{A} \text{ s.t} \\ \text{accept } \leftarrow \mathbf{verify}(q, a(q), \Pi(q), d_t, pk) \wedge \text{reject } \leftarrow check(q, a(q), D_t)]\end{array}\right] \leq \nu(l)$$

where the probability is taken over the randomness of **genkey** and the coins of $\mathcal{A}$. The above security definition maps the mode of operation of an outsourced computation protocol where the database used is originally "finger-printed" by a trusted party that is also solely responsible for dynamically changing it. Clients can trust that the answers they get are "as-good-as" computed by the trusted party.

Set Representation with Polynomials. Sets can be represented with polynomials, using the notion of characteristic polynomial, e.g., as introduced in [18, 24, 26]. Given a set $X = \{x_1, .., x_m\}$, the polynomial $\mathcal{C}_X(r) = \prod_{i=1}^{m}(x_i + r)$ from $\mathbb{Z}_p[r]$, where r is a formal variable, is called the *characteristic polynomial* of X (when possible we will denote this polynomial simply by $\mathcal{C}_X$). Characteristic polynomials constitute representations of sets by polynomials that have the additive inverses of their set elements as roots. What is of particular importance to us is that characteristic polynomials enjoy a number of homomorphic properties w.r.t. set operations. For example, given sets A, B with $A \subseteq B$, it must hold that $\mathcal{C}_B | \mathcal{C}_A$ and given sets X, Y with $I = X \cap Y$, $\mathcal{C}_I = gcd(\mathcal{C}_X, \mathcal{C}_Y)$.

The following lemma characterizes the efficiency of computing the characteristic polynomial of a set.

Lemma 2 ([28]). *Given set $X = x_1, ..., x_n$ with elements from $\mathbb{Z}_p$, characteristic polynomial $\mathcal{C}_X(r) := \sum_{i=0}^{n} c_i r^i \in \mathbb{Z}_p[r]$ can be computed with $O(n \log n)$ operations with FFT interpolation.*

Note that, while the notion of a unique characteristic polynomial for a given set is well-defined, from elementary algebra it is known that there exist many distinct polynomials having as roots the additive inverses of the elements in this set. For instance, recall that multiplication of a polynomial in $\mathbb{Z}_p[r]$ with an invertible unit in $\mathbb{Z}_p^*$ leaves the roots of the resulting polynomial unaltered. We define the following:

Definition 2. *Given polynomials $P(r), Q(r) \in \mathbb{Z}_p[r]$ with degree n, we say that they are* associate *(denoted as $P(r) \approx_a Q(r)$) iff $P(r) | Q(r)$ and $Q(r) | P(r)$.*

Thus, associativity can be equivalently expressed by requesting that $P(r) = \lambda Q(r)$ for some $\lambda \in \mathbb{Z}_p^*$.

Note that although polynomial-based set representation provides a way to verify the correctness of set operations by employing corresponding properties of the characteristic polynomials, it does not provide any computational speedup for this verification process. Intuitively, verifying operations over sets of cardinality n, involves dealing with polynomials of degree n with associated cost that is proportional to performing operations directly over the sets themselves. We overcome this obstacle, by applying our ECRH construction (which can be naturally defined over univariate polynomials with coefficients in $\mathbb{Z}_p$, as already discussed) to the characteristic polynomial $\mathcal{C}_X$: Set X will be succinctly represented by hash value $h(\mathcal{C}_X) = \left(g^{\mathcal{C}_X(s)}, g^{a\mathcal{C}_X(s)}\right)$ (parameter q is an upper bound on the cardinality of sets that can hashed), and a operation of sets X and Y will be optimally verified by computing only on hash values $h(\mathcal{C}_X)$ and $h(\mathcal{C}_Y)$.

Observe that, while every set has a uniquely defined characteristic polynomial, not every polynomial is a characteristic polynomial of some set. Hence extractability of sets from hash values is not guaranteed. For our ADS construction, we will combine the use of the ECRH construction for sets, with an authentication mechanism deployed by the source in a pre-processing phase over the hash values of the original m sets.

3 Setup and Update Algorithms

An *authenticated data structure* (ADS) is a protocol for secure data outsourcing involving the owner of a dataset (referred to as *source*), an untrusted server and multiple clients that issue computational queries over the dataset. The protocol consists of a preprocessing phase where the source uses a secret key to compute some authentication information over the dataset D, outsources D along with this information to the server and publishes some public digest d related to the current state of D. Subsequently, the source can issue update queries for D (which depend on the data type of D), in which case, the source updates the digest and both the source and the server update the authentication information to correspond consistently with the updated dataset state. Moreover, multiple clients (including the source itself), issue computational queries q addressed to the server, which responds with appropriate answer α and proof of correctness Π. Responses can be verified both for *integrity of computation* of q and *integrity of data* used (i.e., that the correct query was run on the correct dataset D) with access only to public key and digest d.

Here we present an ADS supporting hierarchical set operations. We assume a data structure D consisting of m sorted sets $S_1, ..., S_m$, consisting of elements from $\mathbb{Z}_p$,[1] where sets can change under element insertions and deletions; here, p is a l-bit prime number and l is a security parameter. If $M = \sum_{i=1}^{m} |S_i|$, then the total space complexity needed to store D is $O(m + M)$. The supported class of queries is any set-operation formula over a subset of the sets S_i, consisting of unions and intersections.

In this section we present the scheme algorithms for original setup and updates. The basic idea is to use the ECRH construction from Section 2 to represent sets S_i by the hash values $h(C_{S_i})$ of their characteristic polynomials. For the rest of the paper, we will refer to value $h(C_{S_i})$ as h_i, implying the hash value of the characteristic polynomial of

[1] Actually elements must come from $\mathbb{Z} \setminus \{s, 1, ..., m\}$, because s is the secret key in our construction and the m smallest integers modulo p will be used for numbering the sets.

the i-th set of D or the i-th set involved in a query, when obvious from the context. Recall that a hash value h consists of two group elements, $h = (f, f')$. We will refer to the first element of h_i as f_i, i.e., for a set $S_i = (x_1, ..., x_n)$, $f_i = g^{\prod_{j=1}^{n}(x_j+s)}$ and likewise for f'_i.

During the setup phase, the source computes the m hash values $h(C_{S_i})$ of sets S_i and then deploys an authentication mechanism over them, that will provide proofs of integrity for these values under some public digest that corresponds to the current state of D. This mechanism should be able to provide proofs for statements of the form "h_i is hash of the i-th set of the current version of D."

While there exist multiple such mechanisms in the literature (e.g., digital signatures, Merkle trees), here we will be using *accumulation trees*, introduced in [25] (and specifically in the bilinear group setting in [26]) as an alternative to Merkle trees that yields constant time updates and constant size proofs. In our construction, we use the accumulation tree to verify the correctness of hash values for the sets involved in a particular query. On a high level, the public tree digest guarantees the integrity of the hash values and in turn the hash values validate the elements of the sets.

An accumulation tree AT is a tree with $\lceil 1/\epsilon \rceil$ levels, where $0 < \epsilon < 1$ is a parameter chosen upon setup, and m leaves. Each internal node of T has degree $O(m^\epsilon)$ and T has constant height for a fixed ϵ. Intuitively, it can be seen as a "flat" version of Merkle trees. Each leaf node contains the (first half of the) hash value of a set S_i and each internal node contains the (first half of the) hash of the values of its children. Since, under our ECRH construction, hash values are elements in $\mathbb{G}$ we will need to map these bilinear group elements to values in $\mathbb{Z}_p^*$ at each level of the tree before they can be used as inputs for the computation of hash values of higher level nodes. This can be achieved by a function ϕ that outputs a bit level description of hash values under some canonical representation of $\mathbb{G}$ (see below). The setup and update algorithms of our ADS construction can be seen in Figure 2:

The runtime of setup is $O(m+M)$ as computation of the hash values using the secret key takes $O(M)$ and the tree construction has access complexity $O(m)$ for post-order traversal of the tree as it has constant height and it has m leaves. Similarly, update and refresh have access complexity of $O(1)$.

Remark 1. Observe that the only algorithms that make explicit use of the trapdoor s are **update** and **setup** when updating hash value efficiently. Both algorithm can be executed without s (given only the public key) in time $\tilde{O}(D)$.

4 Query Responding and Verification

As mentioned before, we wish to achieve two verification properties: *integrity-of-data* and *integrity-of-computation*. We begin with our algorithms for achieving the first property, and then present two protocols for achieving the second one, i.e., for validating the correctness of a single set operation (union or intersection). These algorithms will be used as subroutines by our final query responding and verification processes.

Algorithm $\{sk, pk\} \leftarrow$**genkey**(1^l). The owner of D runs the sampling algorithm for our ECRH, chooses an injective[2] function $\phi : \mathbb{G} \setminus \{1_{\mathbb{G}}\} \rightarrow \mathbb{Z}_p^*$, and outputs $\{\phi, pk, sk\}$.

Algorithm $\{auth(D_0), d_0\} \leftarrow$ **setup**(D_0, sk, pk). The owner of D computes values $f_i = g^{\prod_{x \in S_i}(x_i+s)}$ for sets S_i. Following that, he constructs an accumulation tree AT over values f_i. A parameter $0 < \epsilon < 1$ is chosen. For each node v of the tree, its value $d(v)$ is computed as follows. If v is a leaf corresponding to f_i then $d(v) = f_i^{(i+s)}$ where the number i is used to denote that this is the i-th set in D (recall that, by definition, sets S_i contain elements in $[m+1, ..., p-1]$). Otherwise, if $N(v)$ is the set of children of v, then $d(v) = g^{\prod_{u \in N(v)}(\phi(d(u))+s)}$ (note that the exponent is the characteristic polynomial of the set containing the elements $\phi(d(u))$ for all $u \in N(v)$). Finally, the owner outputs $\{auth(D_0) = f_1, ..., f_t, d(v)\ \forall v \in AT\ , d_0 = d(r)\}$ where r is the root of AT.

Algorithm$\{auth(D_{h+1}), d_{h+1}, upd\} \leftarrow$ **update**$(u, auth(D_h), d_h, sk, pk)$. For the case of insertion of element x in the i-th set, the owner computes $x + s$ and $\eta = f_i^{x+s}$. For deletion of element x from S_i, the owner computes $(x + s)^{-1}$ and $\eta = f_i^{(x+s)^{-1}}$. Let v_0 be the leaf of AT that corresponds to the i -th set and $v_1, ..., v_{\lceil 1/\epsilon \rceil}$ the node path from v_0 to r. Then, the owner sets $d'(v_0) = \eta$ and for $j = 1, ..., \lceil 1/\epsilon \rceil$ he sets $d'(v_j) = d(v_j)^{(\phi(d'(v_{j-1}))+s)(\phi(d(v_{j-1}))+s)^{-1}}$. He replaces node values in $auth(D_h)$ with the corresponding computed ones to produce $auth(D_{h+1})$. He then sets $upd = d(v_0), ..., d(r), x, i, b$ where b is a bit denoting the type of operation and sends upd to server. Finally, he publishes updated digest $d_{h+1} = d'(r)$.

Algorithm $\{D_{h+1}, auth(D_{h+1}), d_{h+1}\} \leftarrow$ **refresh**$(u, D_h, auth(D_h), d_h, upd, pk)$. The server replaces values in $auth(D_h)$ with the corresponding ones in upd, d_h with d_{h+1} and updates set S_i accordingly.

Fig. 2. Setup and update operations

Authenticity of Hash Values. We present two algorithms that make use of the accumulation tree deployed over the hash values of S_i in order to prove and verify that the sets used for answering are the ones specified by the query description.

Algorithm $\pi \leftarrow$ *QueryTree*$(pk, d, i, auth(D))$ The algorithm computes proof of membership for value x_i validating that it is the i-th leaf of the accumulation tree. Let v_0 be the i-th node of the tree an $v_1, ..., v_{\lceil 1/\epsilon \rceil}$ be the node path from v_0 to the root r. For $j = 1, ..., \lceil 1/\epsilon \rceil$ let $\gamma_j = g^{\prod_{u \in N(v_j) \setminus \{v_{j-1}\}}(\phi(d(u))+s)}$ (note that the exponent is the characteristic polynomial of the set containing the elements $\phi(d(u))$ for all $u \in N(v)$ except for node v_{j-1}). The algorithm outputs $\pi := (d(v_0), \gamma_1), ..., (d(v_{\lceil 1/\epsilon \rceil - 1}), \gamma_{\lceil 1/\epsilon \rceil})$.

Algorithm $\{0, 1\} \leftarrow$ *VerifyTree*(pk, d, i, x, π). The algorithm verifies membership of x as the i-th leaf of the tree by checking the equalities: (i) $e(d(v_1), g) = e(x, g^i g^s)$; (ii) for $j = 1, ..., \lceil 1/\epsilon \rceil - 1$, $e(d(v_j), g) = e(\gamma_j, g^{\phi(d(v_{j-1}))} g^s)$; (iii) $e(d, g) = e(\gamma_{\lceil 1/\epsilon \rceil}, g^{\phi(d(v_{\lceil 1/\epsilon \rceil - 1}))} g^s)$. If none of them fails, it outputs accept.

The above algorithms make use of the property that for any two polynomials $A(r), B(r)$ with $C(r) := A(r) \cdot B(r)$, for our ECRH construction it must be that $e(f(C), g) = e(f(A), f(B))$. In particular for sets, this allows the construction of a

[2] The restriction that ϕ is injective is in fact too strong; it suffices that it is collision-resistant. A good candidate for ϕ is a CRHF that hash the bit-level description of an element of $\mathbb{G}$ to $\mathbb{Z}_p^*$.

single-element proof for set membership (or subset more generally). For example, for element $x_1 \in X = \{x_1, ..., x_n\}$ this witness is the value $g^{\prod_{i=2}^{n}(x_i+s)}$. Intuitively, for the integrity of a hash value, the proof consists of such set membership proofs starting from the desired hash value all the way to the root of the tree, using the sets of children of each node. The following lemma (stated in [26], for an accumulation tree based on bilinear accumulators; it extends naturally to our ECRH) holds for these algorithms:

Lemma 3 ([26]). *Under the q-SBDH assumption, for any adversarially chosen proof π with (j, x^*, π) s.t. VerifyTree$(pk, d, j, x^*, \pi) \rightarrow 1$, it must be that x^* is the j-th element of the tree except for negligible probability. Algorithm QueryTree has access complexity $O(m^{\epsilon} \log m)$ and outputs a proof of $O(1)$ group elements and algorithm VerifyTree has access complexity $O(1)$.*

Algorithms for the Single Operation Case. The algorithms presented here are used to verify that a set operation was performed correctly, by checking a number of relations between the hash values of the input and output hash values, that are related to the type of set operation. The authenticity of these hash values is not necessarily established. Since these algorithms will be called as sub-routines by the general proof construction and verification algorithms, this property should be handled at that level.

Intersection. Let $I = S_1 \cap ... \cap S_t$ be the wanted operation. Set I is uniquely identified by the following two properties: **(Subset)** $I \subseteq S_i$ for all S_i and **(Complement Disjointness)** $\cap_{i=1}^{t}(S_i \setminus I) = \emptyset$. The first captures that all elements of I appear in all of S_i and the second that no elements are left out.

Regarding the subset property, we argue as follows. Let X, S be sets s.t. $S \subseteq X$ and $|X| = n$. Observe that $\mathcal{C}_S | \mathcal{C}_X$, i.e. $\mathcal{C}_X$ can be written as $\mathcal{C}_X = \mathcal{C}_S(r)Q(r)$ where $Q(r) \in \mathbb{Z}_p[r]$ is $\mathcal{C}_{X \setminus S}$. The above can be verified by checking the equality: $e(f_S, W) = e(f_X, g)$ where $W = g^{Q(s)}$. If we denote with W_i the values $g^{\mathcal{C}_{S_i \setminus I}(s)}$, the subset property can be verified by checking the above relation for I w.r.t each of S_i.

For the second property, we make use of the fact that $\mathcal{C}_{S_i \setminus I}(r)$ are disjoint for $i = 1, ..., t$ if and only if there exist polynomials $q_i(r)$ s.t. $\sum_{i=1}^{t} \mathcal{C}_{S_i \setminus I}(r) q_i(r) = 1$, i.e. the gcd of the characteristic polynomials of the the complements of I w.r.t S_i should be 1. Based on the above, we propose the algorithms in Figure 3 for the case of a single intersection:

Algorithm$\{\Pi, f_I\} \leftarrow$ *proveIntersection*$(S_1, ..., S_t, I, h_1, ..., h_t, h_I, pk)$.
1. Compute values $W_i = g^{\mathcal{C}_{S_i \setminus I}(s)}$.
2. Compute polynomials $q_i(r)$ s.t. $\sum_{i=1}^{t} \mathcal{C}_{S_i \setminus I}(r) q_i(r) = 1$ and values $F_i = g^{q_i(s)}$.
3. Let $\Pi = \{(W_1, F_1), ..., (W_t, F_t)\}$ and output $\{\Pi, f_I\}$.

Algorithm{accept,reject} $\leftarrow$ *verifyIntersection*$(f_1, ..., f_t, \Pi, f_I, pk)$.
1. Check the following equalities. If any of them fails output reject, otherwise accept:
   - $e(f_I, W_i) = e(f_i, g) \qquad \forall i = 1, ..., t$
   - $\prod_{i=1}^{t} e(W_i, F_i) = e(g, g)$.

Fig. 3. Intersection proof construction and verification

Union. Now we want to provide a similar method for proving the validity of a union operation of some sets. Again we denote set $U = S_1 \cup ... \cup S_t$ and let h_i be the corresponding hash values as above. The union set U is uniquely characterized by the following two properties: **(Superset)** $S_i \subseteq U$ for all S_i and **(Membership)** For each element $x_i \in U$, $\exists j \in [t]$ s.t. $x_i \in S_j$. These properties can be verified, with values W_i, w_j for $i = 1, ...t$ and $j = 1, ..., |U|$ defined as above checking the following equalities (assuming h_U is the hash value of U):

$$\begin{aligned} e(f_i, W_i) &= e(f_U, g) \qquad \forall i = 1, ..., t \\ e(g^{x_j} g^s, w_j) &= e(f_U, g) \qquad \forall j = 1, ..., |U| \ . \end{aligned}$$

The problem with this approach is that the number of equalities to be checked for the union case is linear to the number of elements in the output set. Such an approach would lead to an inefficient scheme for general operations (each intermediate union operation the verification procedure would be at least as costly as computing that intermediate result). Therefore, we are interested in restricting the number of necessary checks. In the following we provide a union argument that achieves this.

Our approach stems from the fundamental inclusion-exclusion principle of set theory. Namely for set $U = A \cup B$ it holds that $U = (A + B) \setminus (A \cap B)$ where $A + B$ is a simple concatenation of elements from sets A, B (allowing for multisets), or equivalently, $A + B = U \cup (A \cap B)$. Given the hash values h_A, h_B the above can be checked by the bilinear equality $e(f_A, f_B) = e(f_U, f_{A \cap B})$. Thus one can verify the correctness of h_U by checking a number of equalities independent of the size of U by checking that the above equality holds. In practice, our protocol for the union of two sets, consists of a proof for their intersection, followed by a check for this relation. Due to the extractability property of our ECRH, the fact that h_I is included in the proof acts as a proof-of-knowledge by the prover for the set I, hence we can remove the necessity to explicitly include I in the answer.

There is another issue to be dealt with. Namely that this approach does not scale well with the number of input sets for the union operation. To this end we will recursively apply our construction for two sets in pairs of sets until finally we have a single union output. Let us describe the semantics of a set union operation over t sets. For the rest of the section, without loss of generality, we assume $\exists k \in \mathbb{N}$ s.t. $2^k = t$, i.e., t is a power of 2. Let us define as $U_1^{(1)}, ..., U_{t/2}^{(1)}$ the sets $(S_1 \cup S_2), ..., (S_{t-1} \cup S_t)$. For set U it holds that $U = U_1 \cup ... \cup U_{t/2}$ due to the commutativity of the union operation.

All intermediate results $U_i^{(j)}$ will be represented by their hash values $h_{U_i^{(j)}}$ yielding a proof that is of size independent of their cardinality. One can use the intuition explained above, based on the inclusion-exclusion principle, in order to prove the correctness of (candidate) hash values $h_{U_i^{(1)}}$ corresponding to sets U_i and, following that, apply repeatedly pairwise union operations and provide corresponding arguments, until set U is reached. Semantically this corresponds to a binary tree $\mathcal{T}$ of height k with the original sets S_i at the t leafs (level 0), sets $U_i^{(1)}$ as defined above at level 1, and so on, with set U at the root at level k. Each internal node of the tree corresponds to a set resulting from the union operation over the sets of its two children nodes. In general we denote

by $U_1^{(j)}, ..., U_{t/2^j}^{(j)}$ the sets appearing at level j. We propose the algorithms in Figure 4 for proof construction and verification for a single union.

We denote by A, B the two sets corresponding to the children nodes of each non-leaf node of $\mathcal{T}$, by U, I their union and intersection respectively and by F the final union output.

Algorithm$\{\Pi, f_F\} \leftarrow$*proveUnion*$(S_1, ..., S_t, U, h_1, ..., h_t, h_U, pk)$.

1. Initialize $\Pi = \emptyset$.
2. For each $U_i^{(j)}$ of level $j = 1, ..., k$, corresponding to sets U, I as defined above, compute U, I and values h_U, h_I. Append values h_U, h_I to Π.
3. For each $U_i^{(j)}$ of level $j = 1, ..., k$, run algorithm *proveIntersection*(A, B, h_A, h_B, pk) to receive (Π_I, f_I) and append Π_I to Π. Observe that sets A, B and their hash values have been computed in the previous step.
4. Output $\{\Pi, f_F\}$. (h_F has already been computed at step (2) but is provided explicitly for ease of notation).

Algorithm{accept,reject} $\leftarrow$ *verifyUnion*$(f_1, ..., f_t, \Pi, f_F, pk)$.

1. For each intersection argument $\{\Pi_I, f_I\} \in \Pi$ run *verifyIntersection*$(f_A, f_B, \Pi_I, f_I, pk)$. If for any of them it outputs reject, output reject.
2. For each node of $\mathcal{T}$ check the equality $e(f_A, f_B) = e(f_U, f_I)$. If any check fails, reject.
3. For each hash value $h_U \in \Pi$ check $e(f_U, g^a) = e(f'_U, g)$ and likewise for values h_I. If any check fails output reject, otherwise accept.

Fig. 4. Union proof construction and verification

Analysis of the Algorithms. Let $N = \sum_{i=1}^{t} |S_i|$ and $\delta = |I|$ or $|F|$ respectively, depending on the type of operations. For both cases, the runtimes of the algorithms are $O(N \log^2 N \log\log N \log t)$ for proof construction and $O(t + \delta)$ for verification and the proofs contain $O(t)$ bilinear group elements. A proof of the complexity analysis for these algorithms can be found in the full version of our paper [13].

It can be shown that these algorithms, along with appropriately selected proofs-of-validity for their input hash values can be used to form a complete ADS scheme for the case of a single set operation. Here however, these algorithms will be executed as subroutines of the general proof construction and verification process for our ADS construction for more general queries, presented in the next section. In [13], we present similar algorithms for the set difference operation.

Hierarchical Set-Operation Queries. We now use the algorithms we presented in the previous subsection to define appropriate algorithms **query**, **verify** for our ADS scheme. A hierarchical set-operations computation can be abstracted as a tree, the nodes of which contain sets of elements. For a query q over t sets $S_1, ..., S_t$, corresponding to such a computation, each leaf of the tree $\mathcal{T}$ contains an input set for q and each internal node is related to a set operation (union or intersection) and contains the set that results to applying this set operation on its children nodes. Finally the root of the tree contains the output set of q. In order to maintain the semantics of a tree, we assume that each input is treated as a distinct set, i.e., t is not the number of different sets that appear in q,

but the total number of involved sets counting multiples. Another way to see the above, would be to interpret t as the length of the set-operations formula corresponding to q.[3]

Without loss of generality, assume q is defined over the t first sets of D. For reasons of simplicity we describe the mode of operation of our algorithms for the case where all sets S_i are at the same level of the computation, i.e., all leafs of $\mathcal{T}$ are at the same level. The necessary modifications in order to explicitly cover the case where original sets occur higher in $\mathcal{T}$, are implied in a straight-forward manner from the following analysis, since any set S_i encountered at an advanced stage of the process can be treated in the exact same manner as the sets residing at the tree leafs. The algorithms for query processing and verification of our ADS scheme are described in Figure 5.

Each answer from the server is accompanied by a proof that includes a number of hash values for all sets computed during answer computation, the exact structure of which depends on the type of operations. The verification process is essentially split in two parts. First, the client verifies the validity of the hash values of the sets used as input (i.e., the validity of sets specified in q) and subsequently, that the hash values included in the proof respect the relations corresponding to the operations in q, all the way from the input hash values to the hash value of the returned answer α.

Intuitively, with the algorithms from the previous section a verifier can, by checking a small number of bilinear equations, gain trust on the hash value of a set computed by a single set operation. The proof for q is constructed by putting together smaller proofs for all the internal nodes in $\mathcal{T}$. Let Π be a concatenation of single union and single intersection proofs that respect q, i.e., each node in $\mathcal{T}$ corresponds to an appropriate type of proof in Π. The hash value of each intermediate result will also be included in the proof and these values at level i will serve as inputs for the verification process at level $i+1$. The reason the above strategy will yield a secure scheme is that the presence of the hash values serves as proof by a cheating adversary that he has "knowledge" of the sets corresponding to these partial results. If one of these sets is not honestly computed, the extractability property allows an adversary to either attack the collision-resistance of the ECRH or break the q-SBDH assumption directly, depending on the format of the polynomial used to cheat.

Observe that the size of the proof Π is $O(t+\delta)$. This follows from the fact that the t proofs π_i consist of a constant number of group elements and Π is of size $O(t)$ since each of the $O(|T|) = O(t)$ nodes participates in a single operation. Also, there are δ coefficients b_i therefore the total size of Π is $O(t+\delta)$. The runtime of the verification algorithm is $O(t+\delta)$ as steps 2,3 takes $O(t)$ operations and steps 4,5 take $O(\delta)$. A proof of the complexity analysis for these algorithms can be found in the full version of our paper. We can now state the following theorem that is our main result (full proof in [13]).

Theorem 1. *The scheme* $\mathcal{AHSO} = \{$**genkey**, **setup**, **query**, **verify**, **update**, **refresh**$\}$ *is a dynamic* $\mathcal{ADS}$ *scheme for queries* q *from the class of hierarchical set-operations formulas involving unions, intersections and set difference operations. Assuming a data structure* D *consisting of* m *sets* $S_1, ..., S_m$*, and a hierarchical set-operations query* q *involving* t *of them, computable with asymptotic complexity* $O(N)$ *with answer size*

[3] More generally q can be seen as a DAG. Here, for simplicity of presentation we assume that all sets S_i participate only once in q hence it corresponds to a tree.

D is the most recent version of the data structure and let $auth(D), d$ be the corresponding authenticated values and public digest. Let q be a set-operation formula with nested unions and intersections and $\mathcal{T}$ be the corresponding semantics tree. For each internal node $v \in \mathcal{T}$ let $R_1, ..., R_{t_v}$ denote the sets corresponding to its children nodes and O be the set that is produced by executing the operation in v (union or intersection) over R_i. Finally, denote by $\alpha = x_1, ..., x_\delta$ the output set of the root of $\mathcal{T}$.

Algorithm $\{\alpha, \Pi\} \leftarrow \mathbf{query}(q, D, auth(D), pk)$.

1. Initialize $\Pi = \emptyset$.
2. Compute proof-of-membership π_i for value f_i by running *Query-Tree*$(pk, d, i, auth(D))$ for $i \in [t]$ and append π_i, f_i to Π.
3. For each internal node $v \in \mathcal{T}$ (as parsed with a DFS traversal):
   - Compute set O and its hash value $h_O = h(\mathcal{C}_O)$.
   - If v corresponds to a set intersection, obtain Π_v by running *proveIntersection*$(R_1, ..., R_t, h_1, ..., h_t, O, h_O, pk)$. For each subset witness $W_i \in \Pi$ corresponding to polynomial $\mathcal{C}_{R_i \setminus O}$, compute values $\tilde{W}_i = g^{a\mathcal{C}_{R_i \setminus O}(s)}$. Let $\mathcal{W}_v = \{W_1, ..., W_{t_v}\}$. Append $\Pi_v, \mathcal{W}_v, h_O$ to Π.
   - If v corresponds to a set union, obtain Π_v by running *proveUnion*$(R_1, ..., R_t, h_1, ..., h_t, O, h_O, pk)$. Append Π_v, h_O to Π.
4. Append to Π the coefficients $(c_0, ..., c_\delta)$ of the polynomial $\mathcal{C}_\alpha$ (already computed at step 3) and output $\{\alpha, \Pi\}$.

Algorithm $\{$accept,reject$\} \leftarrow \mathbf{verify}(q, \alpha, \Pi, d, pk)$. For internal node $v \in \mathcal{T}$, let $\eta_1, ..., \eta_{t_v}$ denote the hash values of its children node sets $\in \Pi$ (for internal nodes at level 1, the values η_i are the values f_i).

1. Parse each hash value $h \in \Pi$ as $h = (f, f')$.
2. Verify the validity of values f_i. For each value $f_i \in \Pi$ run *VerifyTree*(pk, d, i, f_i, π_i). If it outputs reject for any of them, output reject and halt.
3. For each internal node $v \in \mathcal{T}$ (as parsed with a DFS traversal):
   - Check the equality $e(f_O, g^a) = e(g, f'_O)$. If it does not hold, reject and halt.
   - If v corresponds to a set intersection:
     (a) Run *verifyIntersection*$(\eta_1, ..., \eta_{t_v}, \Pi_v, f_O, pk)$, If it rejects, reject and halt.
     (b) For each pair $W_i, \tilde{W}_i \in \Pi_v$, check the equality $e(W_i, g^a) = e(\tilde{W}_i, g)$. If any of the checks fails, output reject and halt.
   - If v corresponds to a set union, run *verifyUnion*$(\eta_1, ..., \eta_{t_v}, \Pi_v, f_O, pk)$. If it outputs reject, output reject and halt.
4. Validate the correctness of coefficients $\mathbf{c}$. Choose $z \leftarrow_R \mathbb{Z}_p^*$ and compare the values $\sum_{i=0}^{\delta} c_i z^i$ and $\prod_{i=1}^{\delta}(x_i + z)$. If they are not equivalent, output reject and halt.
5. Check the equality $e(\prod_{i=0}^{\delta} g^{c_i s^i}, g) = e(f_\alpha, g)$. If it holds output accept, otherwise reject.

Fig. 5. General set-operations proof construction and verification

δ, $\mathcal{AHSO}$ has the following properties: (i) correct *and* secure *under the q-SBDH and the q-PKE assumptions; (ii) the complexity of algorithm* **genkey** *is $O(|D|)$; (iii) that of* **setup** *is $O(m + |D|)$ (iv) that of* **query** *is $O(N \log^2 N \log\log N \log t + tm^\epsilon \log m)$ for $0 < \epsilon \leq 1$ and it yields proofs of $O(t+\delta)$ group elements; (v) that of* **verify** *is $O(t+\delta)$;*

(vi) and those of **update** *and* **refresh** *are* $O(1)$*; (vii) the authenticated data structure consists of* $O(m)$ *group elements; (viii) the public digest* d *is a single group element.*

Corollary 1. *If the server maintains a list of* m *fresh proofs* $\pi_1, ..., \pi_m$ *for the validity of values* f_i*,* **refresh** *has complexity* $O(m^{2\epsilon} \log m)$*, in order to update the* m^ϵ *proofs* π_i *affected by an update, and* **query** *has complexity* $O(N \log^2 N \log \log N \log t + t)$.

Corollary 2. *In a two-party setting, where only the source issues queries, proofs consist of* $O(t)$ *elements.*

Proof Sketch. Due to the interactive nature of the security game, extracting directly from a successful cheating adversary $\mathcal{A}$ is not possible. Recall however, that all algorithms of $\mathcal{AHSO}$ can be efficiently run with access to pk only. Hence the existence of $\mathcal{A}$ implies the existence of (non-interactive) $\mathcal{A}'$ that upon input pk, runs $\mathcal{A}$ internally providing perfect simulation of the security game and finally outputs the cheating tuple of $\mathcal{A}$. The proof accompanying this cheating answer consists of polynomially many hash values of our ECRH, therefore there exists corresponding extractor $\mathcal{E}'$ that upon the same input outputs the correct pre-image polynomials with overwhelming probability. We then proceed to show that each of these polynomials must be an associate of the characteristic polynomial of the correctly computed set at that point of the computation (or the q-SBDH can be broken). From this, it immediately follows that this holds also for set α^* hence, if it is not the correctly computed set corresponding to query q, the characteristic polynomial of the correctly computed set α and the characteristic polynomial of α^* form a collision for the ECRH. □

References

[1] Atallah, M.J., Cho, Y., Kundu, A.: Efficient data authentication in an environment of untrusted third-party distributors. In: ICDE, pp. 696–704 (2008)

[2] Ateniese, G., De Cristofaro, E., Tsudik, G.: (If) size matters: Size-hiding private set intersection. In: Catalano, D., Fazio, N., Gennaro, R., Nicolosi, A. (eds.) PKC 2011. LNCS, vol. 6571, pp. 156–173. Springer, Heidelberg (2011)

[3] Backes, M., Fiore, D., Reischuk, R.M.: Verifiable delegation of computation on outsourced data. Cryptology ePrint Archive. Report 2013/469 (2013)

[4] Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E., Virza, M.: Snarks for c: Verifying program executions succinctly and in zero knowledge. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013, Part II. LNCS, vol. 8043, pp. 90–108. Springer, Heidelberg (2013)

[5] Berlekamp, E.R.: Factoring polynomials over large finite fields*. In: Proceedings of the Second ACM Symposium on Symbolic and Algebraic Manipulation, SYMSAC 1971, p. 223. ACM, New York (1971)

[6] Bitansky, N., Canetti, R., Chiesa, A., Tromer, E.: From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In: ITCS, pp. 326–349 (2012)

[7] Bitansky, N., Canetti, R., Paneth, O., Rosen, A.: Indistinguishability obfuscation vs. auxiliary-input extractable functions: One must fall. Cryptology ePrint Archive, Report 2013/641 (2013)

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

[9] Boneh, D., Boyen, X.: Short signatures without random oracles. In: Cachin, C., Camenisch, J.L. (eds.) EUROCRYPT 2004. LNCS, vol. 3027, pp. 56–73. Springer, Heidelberg (2004)

[10] 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)
[11] Boyle, E., Pass, R.: Limits of extractability assumptions with distributional auxiliary input. Cryptology ePrint Archive. Report 2013/703 (2013)
[12] Camenisch, J.L., Lysyanskaya, A.: Dynamic accumulators and application to efficient revocation of anonymous credentials. In: Yung, M. (ed.) CRYPTO 2002. LNCS, vol. 2442, pp. 61–76. Springer, Heidelberg (2002)
[13] Canetti, R., Paneth, O., Papadopoulos, D., Triandopoulos, N.: Verifiable set operations over outsourced databases. Cryptology ePrint Archive. Report 2013/724 (2013)
[14] Chatterjee, S., Menezes, A.: On cryptographic protocols employing asymmetric pairings - the role of psi revisited. Discrete Applied Mathematics 159(13), 1311–1322 (2011)
[15] Chung, K.-M., Kalai, Y.T., Liu, F.-H., Raz, R.: Memory delegation. In: Rogaway, P. (ed.) CRYPTO 2011. LNCS, vol. 6841, pp. 151–168. Springer, Heidelberg (2011)
[16] Damgård, I.B.: Towards practical public key systems secure against chosen ciphertext attacks. In: Feigenbaum, J. (ed.) CRYPTO 1991. LNCS, vol. 576, pp. 445–456. Springer, Heidelberg (1992)
[17] 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)
[18] 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)
[19] Goodrich, M.T., Tamassia, R., Triandopoulos, N.: Efficient authenticated data structures for graph connectivity and geometric search problems. Algorithmica 60(3), 505–552 (2011)
[20] Groth, J.: Short pairing-based non-interactive zero-knowledge arguments. In: Abe, M. (ed.) ASIACRYPT 2010. LNCS, vol. 6477, pp. 321–340. Springer, Heidelberg (2010)
[21] Kissner, L., Song, D.: Privacy-preserving set operations. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 241–257. Springer, Heidelberg (2005)
[22] Martel, C.U., Nuckolls, G., Devanbu, P.T., Gertz, M., Kwong, A., Stubblebine, S.G.: A general model for authenticated data structures. Algorithmica 39(1), 21–41 (2004)
[23] Naor, M., Nissim, K.: Certificate revocation and certificate update. IEEE Journal on Selected Areas in Communications 18(4), 561–570 (2000)
[24] Nguyen, L.: Accumulators from bilinear pairings and applications. In: Menezes, A. (ed.) CT-RSA 2005. LNCS, vol. 3376, pp. 275–292. Springer, Heidelberg (2005)
[25] Papamanthou, C., Tamassia, R., Triandopoulos, N.: Authenticated hash tables. In: ACM CCS, pp. 437–448 (2008)
[26] Papamanthou, C., Tamassia, R., Triandopoulos, N.: Optimal verification of operations on dynamic sets. In: Rogaway, P. (ed.) CRYPTO 2011. LNCS, vol. 6841, pp. 91–110. Springer, Heidelberg (2011)
[27] Parno, B., Howell, J., Gentry, C., Raykova, M.: Pinocchio: Nearly practical verifiable computation. In: IEEE SP Symposium, pp. 238–252 (2013)
[28] Preparata, F., Sarwate, D., I. U. A. U.-C. C. S. LAB: Computational Complexity of Fourier Transforms Over Finite Fields. DTIC (1976)
[29] Tamassia, R.: Authenticated data structures. In: Di Battista, G., Zwick, U. (eds.) ESA 2003. LNCS, vol. 2832, pp. 2–5. Springer, Heidelberg (2003)
[30] Yang, Y., Papadias, D., Papadopoulos, S., Kalnis, P.: Authenticated join processing in outsourced databases. In: SIGMOD Conference, pp. 5–18 (2009)
[31] Yiu, M.L., Lin, Y., Mouratidis, K.: Efficient verification of shortest path search via authenticated hints. In: ICDE, pp. 237–248 (2010)
[32] Zheng, Q., Xu, S., Ateniese, G.: Efficient query integrity for outsourced dynamic databases. IACR Cryptology ePrint Archive, 2012:493 (2012)