# Verifiable Set Operations over Outsourced Databases

Ran Canetti* Omer Paneth† Dimitrios Papadopoulos‡ Nikos Triandopoulos§

**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 needs to verify consistency of updates succinctly and without maintaining large state. In particular, existing general solutions are far from practical in this setting.

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. That is, we consider two types of queries issued by the client: updates (insertions and deletions) and data queries, which consist of "circuits" of unions, intersections, and set-differences on the current collection of sets.

This type of queries comes up in database queries, keyword search and numerous other applications, and indeed our scheme can be effectively used in such scenarios. The computational 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 ($O(1)$ 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.

**Keywords:** *secure data outsourcing; verifiable computation; knowledge accumulators; extractable collision-resistant hash functions; secure set operations; authenticated query processing; authenticated data structures*

*Boston University and Tel Aviv University. Email: canetti@bu.edu. Supported by the Check Point Institute for Information Security, an NSF EAGER grant, and an NSF Algorithmic foundations grant 1218461.

†Boston University. Email: omer@bu.edu. Supported by the Simons award for graduate students in theoretical computer science and an NSF Algorithmic foundations grant 1218461.

‡Boston University. Email: dipapado@bu.edu. Supported by NSF grants CNS-1012798 and CNS-1012910.

§RSA Laboratories and Boston University. Email: nikolaos.triandopoulos@rsa.com.

# 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 and, over time, issue queries that are answered by powerful servers.

We consider a *client* that outsources a dataset $D$ to a *server*. The client can then issue 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 SQL queries, streaming datasets and outsourced file systems. We also consider the more general setting where multiple clients issue informational queries to $D$, but only one *source* client can issue update queries. For an example, consider a small company that wishes to store its dataset on a cloud server while accommodating queries from the company's multiple clients.

In the above settings clients may want to verify the integrity of the server's answers to protect themselves against servers that behave maliciously, are compromised by an external attacker, or simply provide false data due to bugs. Specifically, when answering a client 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 itself. For this purpose, we allow the client to perform some polynomial time 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. When issuing an update query, the 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*. Public verifiability is particularly important in the multi-client setting.

In this work we study efficient mechanisms for securely outsourcing databases. Several different measures of efficiency have been considered. First, we would like that the time it takes for the client to verify a proof is 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, we would like the server's computational overhead in computing proofs to be minimal. Additional efficiency considerations include the proof size, and the efficiency of update queries. The number of communication rounds should also be minimized. In this work we concentrate on 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.** The focus of this work is on the problem of *general set operations* in an outsourced setting. That is, we consider a dataset $D$ that consists of $m$ sets $S_1, ..., S_m$, where the clients' queries are arbitrary set-operation over $D$ represented as formulas of union, intersection, and set-difference gates over some of the inputs $S_1, ..., S_m$. 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, authenticated keyword search with elaborate queries, access control management and similarity measurement, hence a practical protocol would be of great importance.

## 1.1 Verifiable Computation - The Generic Approach

Our settings are closely related to the setting of verifiable computation that was 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 we usually assume 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*. They propose a non-interactive solution that is publicly verifiable for every polynomial time computable query over an arbitrary database. Their solution is based on the assumption that Micali's non-interactive CS proofs exist even without ran-

dom oracles. One possible approach for designing a *practical* protocol is based on the memory delegation solution where Micali's CS proofs are replaced with a *succinct non-interactive argument-of-knowledge* (SNARK) construction. Good candidates for such a SNARK include recent works of [4, 6, 29] that provide more practical constructions.[1]

One major obstacle for implementing the generic approach described above (discussed already in [29]) is that it only considers computations 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 (including the size of the answer) may vary dramatically between queries. 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. Such universal circuit introduces additional overhead. Overall, while asymptotically, the computational overhead of the server can be made poly-logarithmic, identifying the constants involved is the main obstacle for using the generic solution to implement secure outsourcing of set operations.

### 1.2 Our Result

In this work we propose a scheme for publicly verifiable secure delegation of set-operations. The main 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 (when the pairing is asymmetric, $8N$ exponentiations are required).

In our scheme, the verification state is of constant size, and proof verification time is $O(\delta + t)$ where $t$ is the size of the query formula and $\delta$ is the answer set size. The dependence on the answer size is inherent since the client must receive the answer set from the server. We stress that the verification time (and proof length) do not grow with the sizes of all other sets involved in the computation.

Our scheme also supports two types of updates: source updates and server-assisted updates. In a source update, the source client maintains an update state of length $O(m)$ ($m$ 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. A source update does not require any party to compute any proofs. Server-assisted updates are used to perform updates that change a large number of elements in the dataset. The idea is for the client to delegate the update to the server (as in [15]). The client can set the value of every set by applying a set-operation formula to the current state of the dataset. The answer to a server-assisted update query includes a new verification state for the client and a proof that the update was performed correctly. Verifying this proof with the old verification state requires the same time as verifying informational queries and no update state. In the current version of the paper, the definition of the model and the construction only consider source updates. In the full version of this paper we plan to extend the definitions and the construction to support also server-assisted updates.

### 1.3 Overview of Techniques

The starting point for our construction is the scheme of Papamanthou, Tamassia and Triandopoulos [28] that supports a single set operation (one union or one intersection). For a query consisting of a single union or intersection over $t$ sets, where the answer set is of size $\delta$, the proof verification time in the scheme of [28] is $O(t + \delta)$. The "naive" way to extend the scheme of [28] 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 this construction is problematic. The problem is that in the scheme of [28] 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

[1] We do not make the distinction between a SNARK with or without a long pre-processing phase since in our setting the client anyhow performs some preprocessing over the entire dataset before outsourcing it.

(if the malicious server would be able to also find the answer set, the scheme of [28] 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.

**Knowledge accumulators.** To solve this problem we need to farther understand the structure of the proofs in [28]. The construction of [28] is based on the notion of a bilinear accumulator [26]. 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 accumulators have homomorphic properties that allow verifying relations between sets via arithmetic relations between the accumulators of the sets. The main idea in this work is to use a different type of accumulator that has "knowledge" properties. That is, the only way for an algorithm to produce a valid accumulation value is to "know" the set that corresponds to that value. The knowledge property of our accumulator together with the soundness of the proof for every single operation (one union or one intersection) allows us to prove the soundness of the composed scheme. Our construction of knowledge accumulators is very similar to previous construction of knowledge commitments in [6, 21]. The construction is based on the $q$-**PKE** assumption which is a variant of 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] (we follow the weaker definition of ECRH with respect to auxiliary input, for which the recent negative evidence presented in [8] do not apply and the distributions we consider here are not captured by the negative result of [12] either).

We also need to change the way a single set operation is proven. Specifically, in [28], 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 we change the proof for union so it only involves a constant number of accumulation values. Additionally, we change the way single operations are proven to support efficient generation of proofs for formulas that have gates with large fan-in.

**The verification state and accumulation trees.** In order to verify a proof in our scheme, the client only needs to know the accumulation values for the sets that participate in the computation. Instead of saving the accumulation values of all sets in the dataset, the client will only save a constant size verification state that contains a special hash of these accumulation values. We hash the accumulation values of the sets in the dataset using an *accumulation tree*, introduced in [27]. This primitive can be thought of as a special "tree hash" that makes use of the algebraic structure of the accumulators to gain in efficiency (authentication paths are of constant length).

Finally we note that the formal definition of our model, and consequently also the security analysis, use the popular framework of *authenticated data structures* introduced in [32].

### 1.4 Combining our Approach with the Generic Approach

One disadvantage of our solution compared to the generic solution of [15] is that our proofs are not constant in size and depend on the size of the query formula. One way to fix this is to compose our proofs with generic SNARK's. That is, we can consider a hybrid scheme where the server proves using a SNARK that it knows a proof (following our scheme) for the validity of its answer. The security of the hybrid scheme can be shown by a reduction to the security of our scheme based on the knowledge property of the SNARK (via a standard composition argument [7]).

The advantage of using a hybrid scheme is that on one hand the proof size is constant and independent of the size of the query (this follows from the succinctness of the SNARK). On the other hand, the hybrid scheme might be much more practical than simply using a generic solution in terms of server computation since the overhead of the circuit reductions might not be significant. This saving is due to the server using

the SNARK only to certify a small computation (involving only the small verification circuit of our scheme). Moreover, most of the server's work, when answering the query, is certified using our scheme that is tailored for set operations, resulting in less overhead for the server compared to the generic solution alone.

### 1.5 Related Work

The very recent work of [3] also considers a practical secure database delegation scheme supporting a restricted class of queries. They consider functions expressed by arithmetic circuits of degree up to 2. Their construction is based on homomorphic MAC's and their protocol appears practical, however their solution is only privately verifiable and it does not support deletions from the dataset. Additionally we note that the security proof in [3] is not based on non-falsifiable assumptions. In a sense, that work 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, 11, 19, 22]. There exists a number of works from the database community that address this problem [23, 34], 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 [26, 28]) and this directly relates this work with a line of publications coming from the *cryptographic accumulators* literature [13, 26]. Indeed our ECRH construction, viewed as a mathematical object, is identical to a pair of bilinear accumulators (introduced in [26]) 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). Indeed, this idea is central to all of our proofs for validity of set-operation computations. 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 [32], appears extensively both in the cryptography and databases literature (see for example [1, 20, 23, 25, 28, 35, 36]). A wide range of functionalities has been addressed in this context including range queries and basic SQL joins.

## 2 Tools and Definitions

In the following, we denote with $l$ the security parameter and with $\nu(l)$ a negligible function.[2] 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 [9, 18], 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. For example, an algorithm making $k$ modular multiplications over a group of size $O(n)$ where $n$ is $O(exp(l))$ for a security parameter $l$, runs in time $O(k \log n)$. In the access complexity model, this is $O(k)$ ignoring the "representation" cost for each group element (that is specified by $l$).

**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 $1^l$.

[2] A function $f(l)$ is negligible if for each polynomial function $poly(l)$ and all large enough values of $l$, $f(l) < 1/(poly(l))$.

For cleaner presentation, in what follows we assume a symmetric (Type 1) pairing $e$. In Section 4 we discuss the modifications needed to implement our construction in the (more efficient) asymmetric pairing case (see [14] for a general discussion of types of pairings).

Our security analysis makes use of the following two assumptions over groups with bilinear pairings:

**Assumption 1 ($q$-Strong Bilinear Diffie-Hellman [10])** *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 [21])** *For any poly-size adversary $\mathcal{A}$, there exists a poly-size extractor $\mathcal{E}$ such that:*

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

**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 $\mathcal{E}$ such that:*

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

**An ECRH construction from $q$-PKE.** We next provide an ECRH construction from the $q$-**PKE** assumption defined above. In [6] the authors provide an ECRH construction from an assumption that is conceptually similar and can be viewed as a simplified version of $q$-**PKE** and acknowledge 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.[3]

- 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 .

[3] It should be noted that while the construction from [6] is conceptually similar, its collision resistance cannot be proven by a reduction to $q$-**SBDH**; it is instead provable with a direct reduction to the computation of discrete logarithms in $\mathbb{G}$.

- 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 $(q \cdot l, 2l)$-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 $\epsilon$, $(\mathbf{x}, \mathbf{y})$ 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 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 $\epsilon'$ 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. □

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 [28], 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 the following six polynomial-time algorithms:

1. $\{sk, pk\} \leftarrow$ **genkey**$(1^k)$. Outputs secret and public keys $sk$ and $pk$, 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 the secret key.
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. $\{$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".

Let $\{$accept, reject$\} = check(q, a(q), D_h)$ 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:

**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 $check(q, a(q), D_h)$ 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:

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

where the probability is taken over the randomness of **genkey** and the coins of $\mathcal{A}$.

### 2.1 Set Representation with Polynomials

Sets can be represented with polynomials, using the notion of characteristic polynomial, e.g., as introduced in [19, 26, 28]. 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 ([30])** *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 $h(\mathcal{C}_Y)$.

**A note on extractability.** In the above, we are essentially using a pre-processing step representing sets as polynomials, before applying the extractable hash function on the polynomial representations. We cannot define the ECRH directly for sets since, while every set has a uniquely defined characteristic polynomial, not every polynomial is a characteristic polynomial of some set. Hence extractability of sets (using only public key information) is not guaranteed. For example, an adversary can compute an irreducible polynomial $Y \in \mathbb{Z}_p[r]$, of degree $> 1$, and output $h(Y)$. Since $Y$ has no roots, no extractor (without access to the secret key) can output a set for which $Y$ is the characteristic polynomial (it can however extract polynomial $Y$ with overwhelming probability). In fact, defined directly over sets with elements from $\mathbb{Z}_p$, the

function ensemble $\{\mathcal{H}_l\}_l$ with an internal computation of the characteristic polynomial, can be shown to be extractable collision-resistant under the $\mathcal{ECRH}_2$ definition recently introduced in [17]. In the context of a cryptographic protocol for sets, additional mechanisms need to be deployed in order to guarantee that a given hash value corresponds to the characteristic polynomial of some set. 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. This will allow any client to verify the authenticity and freshness of the hash values corresponding to sets that are input to its query.

# 3 An $\mathcal{ADS}$ scheme for hierarchical set operations

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 pre-processing 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 $\alpha$ and proof of correctness $\Pi$. 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 information and digest $d$. The service offered to clients in this setting is that the received answers are "as-good-as" being directly computed by the trusted source. A restricted version of this setting, is a two-party model where the owner of $D$ outsources it to a server and issues updates and computational queries, benefiting in both storage and computation cost.

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 $\mathbb{Z}_p$,[4] 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.

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 the $i$-th set of $D$ or the $i$-th set involved in a query, when it is obvious in 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 $f'_i$. For the authenticity of these values, an authentication mechanism similar to Merkle trees (but allowing more efficient updates) will be deployed by the source.

Each answer provided by 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 by the answer computation process (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 $\alpha$. The key technique is that by using our ECRH construction we can map relations between input and output sets in a set operation, to similar relations in their hash values. This allows the verification process to run in time independent of the cardinality of involved sets and only linear to the length of $q$ and $\alpha$ making it asymptotically as fast as simply reading the input and output. In the following sections, we present the algorithms of our construction.

[4] 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.

### 3.1 Setup and Updates

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 $D$."

There is a wide variety of such mechanisms that can be deployed by the owner of $D$ and the choice must be made with optimization of a number of parameters in mind, including digest size, proof size and verification time, setup and update cost and storage size. For example, using a standard collision resistant hash function, the owner can compute the hash of the string $h_1||...||h_m$ as a single hash value. However, a single update in $D$ will require $O(m)$ work in order to compute the updated digest from scratch. On the other hand, the owner can use a digital signature scheme to sign a hash representation of each set. This yields an update cost of $O(1)$ (a single signature computation) but the digest consists of $m$ signatures.

Another popular authentication mechanism for proofs of membership are Merkle hash trees [24] that provide logarithmic size proofs, updates and a single value digest. Such a mechanism, allows the server to provide proofs that a value $h_i$ belongs in the set of hash values of the sets in $D$. An alternative to Merkle trees, introduced in [27] (and specifically in the bilinear group setting in [28]) are *accumulation trees*. The difference between them is that their security is based on different cryptographic assumptions (secure hashing versus bilinear group assumptions) and, arguably more importantly, accumulation trees yield constant size proofs (independently of the number of elements in the tree) and constant time updates. Another useful property of the accumulation tree is that it can be computed using the same ECRH construction we will be using for the rest of the algorithms of our scheme. Thus, we can avoid the cost for additional public/secret key generation and maintenance. 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 $\epsilon$. 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 $\phi$ that outputs a bit level description of hash values under some canonical representation of $\mathbb{G}$ (see below). The accumulation tree primitive we are using here was introduced in [28] where the corresponding "hashing" function used was the bilinear accumulator construction from [26]. We are implicitly making use of the fact that the outputs of our ECRH construction can be interpreted as pairs of accumulation values of sets.

Now we present the setup and update algorithms or our ADS construction:

**Algorithm** $\{sk, pk\} \leftarrow$**genkey**$(1^l)$. The owner of $D$ runs the sampling algorithm for our ECRH construction, chooses an injective[5] 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

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

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.

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 that is proportional the size of $D$.*

## 3.2 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.

### 3.2.1 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*$(pk, d, i, x, \pi)$. 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 output 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 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 [28], for an accumulation tree based on bilinear accumulators; it extends naturally to our ECRH construction) holds for these algorithms:

**Lemma 3 ([28])** *Under the q-**SBDH** assumption, for any adversarially chosen proof $\pi$ s.t. $\{j, x^*, \pi)$ 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 $O(1)$.*

### 3.2.2 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.

**Algorithm for 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 property 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 1 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:
   - $e(f_I, W_i) = e(f_i, g) \qquad \forall i = 1, ..., t$
   - $\prod_{i=1}^{t} e(W_i, F_i) = e(g, g)$

*Figure 1: Intersection*

**Algorithm for 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 $U$ is 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 2 for proof construction and verification for a single union:

For ease of notation 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 $\Pi$.
3. For each $U_i^{(j)}$ of level $j = 1, ..., k$, run algorithm *proveIntersection*$(A, B, h_A, h_B, pk)$ to receive $(\Pi_I, f_I)$ and append $\Pi_I$ to $\Pi$. 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, output 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.

*Figure 2: Union*

**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 Section 3.4.

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 the full version of the paper, we present similar algorithms for the set difference operation

### 3.2.3 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 $q$.[6]

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 3.

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. Observe that, each prover's algorithm "binds" some bilinear group elements (the first parts of the input hash values) to a single bilinear group element (the first part of the hash value of the output set). We made explicit use of that, in order to create a proof of union for more than two sets in the previous section. Here we generalize it, to be able to obtain similar proofs for hierarchical queries containing intersections and unions. The proof for $q$ is constructed by putting together smaller proofs for all the internal nodes in $\mathcal{T}$. Let $\Pi$ 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 $\Pi$. 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 a 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 $\Pi$ is $O(t+\delta)$. This follows from the fact that the $t$ proofs $\pi_i$ consist of a constant number of group elements and $\Pi$ is of size $O(t)$ since each of the $O(|T|) = O(t)$ nodes participates in a single operation. Also, there are $\delta$ coefficients $b_i$ therefore the total size of $\Pi$ is $O(t+\delta)$.

[6] 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. This is not a limitation of our model but to simplify the execution of the algorithms, every set encountered is treated uniquely. This can incur a redundant overhead in the analysis, that is avoidable in practice (e.g., by not including duplicate values in proofs).

$D$ is the most recent version of the data structure and $auth(D), d$ 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 $\pi_i$ for value $f_i$ by running *QueryTree*$(pk, d, i, auth(D))$ for $i \in [t]$ and append $\pi_i, f_i$ to $\Pi$.
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 $\Pi_v$ 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 $\Pi$.
   - If $v$ corresponds to a set union, obtain $\Pi_v$ by running *proveUnion*$(R_1, ..., R_t, h_1, ..., h_t, O, h_O, pk)$. Append $\Pi_v, h_O$ to $\Pi$.
4. Append to $\Pi$ 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 $\eta_i$ are the values $f_i$).

1. Verify the validity of values $f_i$. For each value $f_i \in \Pi$ run *VerifyTree*$(pk, d, i, f_i, \pi_i)$. If it outputs reject for any of them, output reject and halt.
2. 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 outputs reject, output 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.
3. 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.
4. 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.

*Figure 3: General Set-Operations Proof Construction and Verification*

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 Section 3.4.

### 3.3 Main Result

We can now state the following theorem that is our main result.

**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* $\delta$*,* $\mathcal{AHSO}$ *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.*

*Proof:* Let $\mathcal{A}_{ADS}$ be an adversary for and $\mathcal{AHSO}$ scheme. Recall the $\mathcal{A}_{ADS}$ is given a public key generated by **genkey** containing a description of an ECRH $h$. $\mathcal{A}_{ADS}$ then calls **setup** with the parameter $D_0$ and subsequently makes additional oracle calls to the algorithms **update**, **refresh**, **verify**, **proof**, of $\mathcal{AHSO}$. Finally $\mathcal{A}_{ADS}$ outputs $\{\alpha, \Pi, q, D_k, auth(D_k), d_k\}$ where $\Pi$ is a proof that contains images of the hash $h$. We show that there exists an extractor $\mathcal{E}$ that except with negligible probability over the choice of $h$, when $\mathcal{E}$ is given $h$, outputs a pre-image for every valid image of $h$ in $\Pi$. We cannot directly use the extraction property of the ECRH since the adversary $\mathcal{A}_{ADS}$ is getting access to oracles for the algorithms of $\mathcal{AHSO}$ and we do not have access to the code of these oracles. The idea of this proof is to use the fact that all the algorithms of $\mathcal{AHSO}$ (except **genkey**) can be executed over the initial database $D_0$ in polynomial time given only the public key $h$ (see Remark 1), and therefore there exists an adversary $\mathcal{A}'_{ADS}$ that internally emulates $\mathcal{A}'_{ADS}$ together with its oracles and outputs the same as $\mathcal{A}_{ADS}$. Let $\mathcal{A}'_i$ be the adversary that emulates $\mathcal{A}'_{ADS}$ and outputs the $i$'th hash value $h_i$ in proof $\Pi$ contained in the output of $\mathcal{A}'_{ADS}$. It follows from the properties of the ECRH that there exists an extractor $\mathcal{E}_i$ for $\mathcal{A}'_i$ that outputs a pre-image of $h_i$ whenever $h_i$ is indeed in the image of $h$. Therefore there exists a single extractor $\mathcal{E}$ that outputs the pre-images for all valid $h_i$'s with overwhelming probability. Finally, observe that valid hash values $h_i$ are efficiently recognizable as elements of $\mathbb{G}$.

As a building block for our proof, we prove the following lemma:

**Lemma 4** *If the $q$-**SBDH** assumption holds, then for any poly-size adversary $\mathcal{A}$ that upon input $pk$ outputs $(S_1, ..., S_t, O, \Pi, f_O)$ s.t. (i)* verifyIntersection$(f(\mathcal{C}_{S_1}), ..., f(\mathcal{C}_{S_t}), \Pi, f_I, pk)$ *(resp.* verifyUnion*) accepts and (ii) $f(\mathcal{C}_O) = f_O$, $O = \cap_{i=1}^{t} S_i$ (resp. $O = \cup_{i=1}^{t} S_i$) with all but negligible probability.*

*Proof:* We examine the two cases separately.

**Intersection.** Let us assume that there exists $\mathcal{A}$ that outputs $S_1, ..., S_t, O, \Pi, f_O$ s.t. *verifyIntersection* accepts and $O \neq I := \cap_{i=1}^{t} S_i$, with non-negligible probability. We will construct an adversary $\mathcal{A}'$ that breaks the $q$-**SBDH** assumption. For ease of notation we denote $\mathcal{C}_{S_i} = Q_i(r)$ and $\mathcal{C}_O = P(r)$.

Since $O \neq I$, either it contains an element $x$ s.t. $x \notin I$, or there exists element $x \in I$ s.t. $x \notin O$ (or both happen at the same time). Let us deal with the first case. Since $x \in O \wedge x \notin I$, there must exist set $S_j$ s.t. $x \notin S_j$. Therefore for the term $(x + r)$ it is true that $(x + r) \not| Q_j(r)$ and $(x + r) | P(r)$. It follows that there exist efficiently computable $F(r), \kappa$ s.t. $Q_j(r) = (x + r)F(r) + \kappa$. Also let $H(r)$ be polynomial s.t. $(x + r)H(r) = P(r)$. The following equalities must hold:

$$\begin{aligned} e(f_O, W_j) &= e(f_j, g) \\ e(g, W_j)^{P(s)} &= e(g, g)^{Q_j(s)} \\ e(g, W_j)^{(x+s)H(s)} &= e(g, g)^{(x+s)Q_j(s)+\kappa} \\ \left(e(g, W)^{H(s)} e(g, g)^{-Q_j(s)}\right)^{\kappa^{-1}} &= e(g, g)^{\frac{1}{x+s}}. \end{aligned}$$

It follows that $\mathcal{A}'$ can, by outputting the above value break the $q$-**SBDH** for point $x$. Hence, this case can happen only with negligible probability.

It remains to deal with the second case, conditioned on the first not happening. Namely, there exists $x \in I$ that is omitted by answer $O$, i.e. $O$ is a common subset of $S_i$ but not the maximal one. There must exist $x \in I$ s.t. $x \notin O$ therefore it must be that $x \in (S_i \setminus O)$ for all $i = 1, ..., t$. Let polynomials $R_i(r) = \mathcal{C}_{S_i \setminus O}$. Observe that because the verifier accepts, it must be that $e(g, W_i) = e(g, g)^{Q_i(s)}$, hence $W_i = g^{R_i(s)}$. From the above it must hold that $R_i(r) = (x + r)R'_i(r)$ for some $R'_i(r) \in \mathbb{Z}[r]$. The following

must be true:

$$\prod_{i=1}^{t} e(W_i, F_i) = e(g,g)$$
$$\left(\prod_{i=1}^{t} e(g^{R'_i(s)}, F_i)\right)^{x+s} = e(g,g)$$
$$\prod_{i=1}^{t} e(g^{R'_i(s)}, F_i) = e(g,g)^{\frac{1}{x+s}}.$$

From the above, $\mathcal{A}'$ can break the $q$-**SBDH** assumption for point $x$. It follows that $O$ is the maximal common subset of $S_i$'s with all but negligible probability.

If we denote the two cases as $E_1, E_2$, we showed that $\Pr[E_1], \Pr[E_2|E_1^c]$ are negligible probabilities. Since $E_1, E_2$ cover all possible cheating adversary strategies, the claim follows by a simple union bound.

**Union.** Let us assume that there exists $\mathcal{A}$ that outputs $S_1, ..., S_t, O, \Pi, f_O$ s.t. *verifyUnion* accepts and $O \neq U := \cap_{i=1}^{t} S_i$, with non-negligible probability. We will construct an adversary $\mathcal{A}'$ that either finds a collision in $h$, or breaks the $q$-**SBDH** assumption. For ease of notation we denote $\mathcal{C}_{S_i} = Q_i(r)$ and $\mathcal{C}_O = P(r)$. We begin by providing a proof for $t = 2$, i.e., a union of two sets $A \cup B$.

Upon receiving the output from $\mathcal{A}$, adversary $\mathcal{A}'$ runs the extractor $\mathcal{E}_{\mathcal{A}}$ (the existence of which is guaranteed by our analysis in the start of the proof of Theorem 1) for value $h_{I^*} \in \Pi$ to receive polynomial $R(r)$ s.t. $g^{R(s)} = h_{I^*}$ with overwhelming probability.

**Claim 1** *$R(r) \approx_a \mathcal{C}_I$ where $I = A \cap B$, with all but negligible probability.*

**Proof of Claim.** The following two relations must hold:

$$e(g, W_A)^{R(s)} = e(g,g)^{Q_A(s)}$$
$$e(g,g)^{Q_A(s)\cdot Q_B(s)} = e(g,g)^{R(s)\cdot P(s)}.$$

First we will prove that $R(r)$ can be written as a product of degree 1 polynomials. Assume there exists irreducible polynomial $R'(r)$ of degree$> 1$ and polynomial $J(r)$ s.t. $R(r) = R'(r)J(r)$. It follows that $R(r)P(r) \neq Q_A(r)Q_B(r)$ (since only one of them has irreducible terms of degree greater than 1), however from the above equality $h(R(r)P(r)) = h(Q_A(r)Q_B(r))$ therefore by outputting $R(r)\cdot P(r), Q_A(r)\cdot Q_B(r)$ (in coefficient form), $\mathcal{A}'$ finds a collision in the ECRH. This can happen with negligible probability hence $R(r)$ can be written as a product of degree 1 polynomials with all but negligible probability.

From this it follows that $\mathcal{A}'$ can, by running a probabilistic polynomial factorization algorithm, find roots $x_i$ s.t. $R(r) = \lambda \prod_{i\in[deg(R)]}(x_i + r)$. Note that upon input polynomial $R(r)$, value $\lambda$ can be efficiently computed correctly by a polynomial factorization algorithm, with all but negligible probability, and the value $\lambda^{-1}$ is also computable efficiently since $p$ is a prime.

Let $X$ be the set containing the additive inverses of the roots $x_i$[7] and observe that $\mathcal{C}_X = \lambda \prod_{i\in[deg(R)]}(x_i + r)$. If $X \neq I$, $\mathcal{A}'$ can output $\{A, B, X, Pi^* = (h_X^{\lambda^{-1}}, W_A^{\lambda}, W_B^{\lambda}, F_A^{\lambda^{-1}}, F_B^{\lambda^{-1}})\}$. It is easy to verify that the above is satisfying proof for the claim that $X = A \cap B$ (i.e., **verifyIntersection** accepts), while $X \neq I$. By our previous analysis for the intersection case, this can only happen with negligible probability. This concludes the proof of the claim. ■

[7]The case where $X$ has a root that is also a root of $\mathcal{I}$ but with cardinality $> 1$ can easily be dealt with as follows. Since the term $(x + s)$ appears in the exponent in both sides of the bilinear relation, $\mathcal{A}'$ can remove it from both hands, until at the end it remains only in one of them. After that happens, the consequent analysis holds. Similar argument can be made for the union case thus in the following we skip this part of the analysis.

Consequently, the following must be true:

$$e(g,g)^{Q_A(s)Q_B(s)} = e(g,g)^{R(r)P(r)}$$
$$e(g,g)^{\prod_{x\in A}(x+s)\prod_{x\in B}(x+s)} = e(g,g)^{P(s)\lambda\prod_{x\in A\cap B}(x+s)}$$
$$e(g,g)^{\prod_{x\in A\cup B}(x+s)} = e(g,g)^{\lambda P(s)}.$$

In case polynomials $\mathcal{C}_{A\cup B}$ and $\lambda P(r)$ are not equivalent, due to the above equality $\mathcal{A}$ can by outputting them find a collision in the ECRH. Therefore it must be that with overwhelming probability $\lambda P(r) = \mathcal{C}_U$. Again, if $\lambda \neq 1$ then the two polynomials form a collision for the ECRH, therefore with all but negligible probability, $O = U$.

Let us now turn our attention to the case of a generalised union over $k$ sets (assume wlog that $k$ is a power of 2). Consider the binary tree $\mathcal{T}$ that captures this union operation as described in Section 3.2.2. Observe that this tree consists only of $O(poly(l))$ nodes ($2t-1$ in practice) hence $\mathcal{A}'$ can efficiently run an extractor for all intermediate hash values corresponding to internal nodes of $\mathcal{T}$ (as per our former analysis) to compute the related polynomials correctly, with overwhelming probability.

We will prove that the polynomial $\mathcal{C}_O(r)$, corresponding to $h_O$, is an associate of $\mathcal{C}_U$ by showing that this is true for all intermediate polynomials and their corresponding sets. We will do this by an induction on the levels of $\mathcal{T}$.

**level-1** Let $P_i^{(1)}(r)$ be the extracted polynomials for all first level nodes. Let us assume that there exists node $v$ in the first level such that $P(r) := P_v^{(1)}(r) \not\approx_a \mathcal{C}_{U_i^{(1)}}$ where $U_i^{(1)}$ is the corresponding correct union of its two children nodes.
With a similar argument as above, $P(r)$ can be written as a product of degree 1 polynomials with all but negligible probability (otherwise a collision in the ECRH can be found). Let $X$ be the set containing the additive inverses of the roots $x_i$ of $P(r)$. It follows that $P(r) = \lambda\mathcal{C}_X$ for some efficiently computable $\lambda \in \mathbb{Z}_p^*$. Similar as above, if $X \neq U_i^{(1)}$, $\mathcal{A}'$ can output $\{A, B, X, \Pi^* = (h_X^{\lambda^{-1}}, h_I^{\lambda}, W_A^{\lambda^{-1}}, W_B^{\lambda^{-1}}, F_A^{\lambda}, F_B^{\lambda})\}$. It is easy to verify that this consists a satisfying proof for the claim $A \cup B = X$, which by our previous analysis can happen with negligible probability and the claim follows.

**level-j** Assuming that this holds for the polynomials on level $j$ we will show that it also holds for level $j+1$. Let us assume that this not the case. It follows that there must exist node $v$ of the tree on level $j+1$ the children of which have extracted polynomials $Q_A(r), Q_B(r)$, the corresponding extracted output polynomial is $P(r)$ and the corresponding extracted polynomial for the intersection be $H(r)$. Assuming $P(r)$ is not an associate of $\mathcal{C}_U$ we will construct an adversary that finds a collision in the ECRH similar to above.
By assumption, $Q_A(r) = \lambda_A \prod_{i\in[|A|]}(x_i + r)$ and likely for $Q_B(r)$ (recall that these are associate polynomials of the correctly computed corresponding set at level $j$) for sets $A, B$. If $P(r)$ contains an irreducible factor of degree $> 1$, our previous analysis shows that a collision for the ECRH is found.
Therefore $P(r)$ can be written as a product of degree 1 polynomials and a scalar and there exist an efficiently computable set $X$ and $\lambda \in \mathbb{Z}_p^*$ s.t. $P(r) = \lambda\mathcal{C}_X$. Similar as above, if $X \neq A \cup B$, $\mathcal{A}'$ can output $\{A, B, X, \Pi^* = (h_X^{\lambda^{-1}}, h_I^{\lambda/\lambda_A\cdot\lambda_B}, W_A^{\lambda_B}, W_B^{\lambda_A}, F_A^{\lambda_B^{-1}}, F_B^{\lambda_A^{-1}})\}$. It is easy to verify that this consists a satisfying proof for the claim $A \cup B = X$, which by our previous analysis can happen with negligible probability and the claim follows.
Since this holds for every node of level $j+1$, this concludes our induction proof.

Hence with all but negligible probability, the claim holds for the value $h_O$. As per the intersection case, it must be that with all but negligible probability $O = U$. □

For the proof of our main result we make use of Lemmas 4 and 3. Let $\mathcal{A}_{ADS}$ be a poly-size adversary that upon input the public key $pk$ of our ECRH construction, is given oracle access to all algorithms of $\mathcal{AHSO}$. $\mathcal{A}_{ADS}$ picks initial state $D_0$ for the data structure and computes $auth(D_0), d_0$ through oracle access to **setup()**. Consequently he chooses a polynomial number of updates and with oracle access to **update()** computes $D_{i+1}, auth(D_0), d_{i+1}$ for $i = 0, ..., h$. Also, he receives oracle access to algorithms **query**,**verify**,**refresh**. Finally, $\mathcal{A}_{ADS}$ outputs $\{\alpha', \Pi, q, D_k, auth(D_k), d_k\}$ where $k$ is between 0 and $h+1$ and denotes the snapshot of the data structure to which the query $q$ is to be applied. We want to measure the probability that **verify**$(\alpha', \Pi, q, pk, d_k)$ outputs accept and algorithm **check**$(D_k, q, \alpha')$ outputs reject (i.e., $\alpha'$ is not equal to the set produced by applying operations in $q$ on dataset $D_k$).

Assuming $\mathcal{A}_{ADS}$ can succeed in the above game with non-negligible probability $\epsilon$, we will use him to construct $\mathcal{A}'$ that finds a collision in the ECRH with non-negligible probability. $\mathcal{A}'$ works as follows. Upon input $pk$ of ECRH, he sends it to $\mathcal{A}_{ADS}$. Following that, he provides oracle interface to $\mathcal{A}$. Finally, he receives $\{\alpha', \Pi, q, D_k, auth(D_k), d_k\}$ from $\mathcal{A}$ and runs corresponding extractor $\mathcal{E}_{\mathcal{A}_{ADS}}$ to receive hash pre-images for all hash vales in $\Pi$.

Let $S_1, ..., S_t$ be the sets in $D_k$ over which $q$ is defined. First $\mathcal{A}'$ computes honestly $q$ over $S_i$, and receives the correct output $\alpha$ and all intermediate sets. Then he runs **verify** on the received tuple and checks if $\alpha \neq \alpha$. If verification fails or $\alpha = \alpha'$ he aborts (i.e. he only proceeds if $\mathcal{A}_{ADS}$ wins the ADS game). Following that, $\mathcal{A}'$ checks if $f(\mathcal{C}_{S_i}) = f_i$ for $i = 1, ..., t$. If any of the checks fails, he aborts. Then $\mathcal{A}'$ compares the correctly computed set for each node $v \in \mathcal{T}$ and the corresponding extracted polynomial which we denote by $P_v(r)$. Given polynomial $P_v(r)$ for each node, $\mathcal{A}'$ checks if it is an associate polynomial of the characteristic polynomial of the corresponding honestly computed set. If this does not hold for some node $v$, he aborts. Finally, he outputs the pair of polynomials $P_{root}(r), \mathcal{C}_{\alpha'}$.

First, note that $\mathcal{A}'$ runs in time polynomial in the security parameter, since both $\mathcal{A}_{ADS}$ and $\mathcal{E}_{\mathcal{A}_{ADS}}$ run in polynomial time, the set computations can be done in polynomial time and polynomial associativity is also decidable in polynomial time by long division. Regarding, his success probability in finding a collision we argue as follows.

Let $E'$ be the event that $\mathcal{A}'$ succeeds in finding a collision and $B$ the event that $\mathcal{A}_{ADS}$ wins the ADS game. By assumption $\Pr[B] > \epsilon$ for non-negligible $\epsilon$, a function of $l$. Observe that, conditioned on not aborting, the probability of $\mathcal{A}'$ to find a collision is at least $(1-\nu^*(l))$ where $\nu^*(l)$ is the sum of the negligible errors in the output of the extractor and the randomized factorization algorithm, which by a union bound is an upper bound for the total error probability. This holds because, since $\mathcal{A}'$ did not abort, the verification succeeded and $\mathcal{A}_{ADS}$ provided a false answer which implies that the polynomials output are not equivalent yet they have the same hash values. Overall $\Pr[E'] = \Pr[E'|\neg\text{abort}]\Pr[\neg\text{abort}] \geq (1-\nu^*(l))\Pr[\neg\text{abort}]$.

Let $E_V$ be the event that **verify** accepts during the first step of $\mathcal{A}'$ and $\alpha \neq \alpha'$. Also, let $E_1$ be the event that all $f(\mathcal{C}_{S_i}) = f_i$ for $i = 1, ..., t$ given that **verify** accepts and $E_2$ be the event that all extracted polynomials are of the form $P_v(r) \approx_a \mathcal{C}_O$ also given that **verify** accepts. Also, let $E_3$ be the event that the polynomials $\mathcal{C}_{\alpha^*}(r)$ and $\sum_{i=0}^{\delta-1} c_i r^i$ are equivalent given that **verify** accepts. By Lemma 3, $\Pr[E_1] > 1 - \nu_1(l)$ and $\Pr[E_3] > 1 - \nu_3(l)$ since, by the Schwartz-Zippel lemma [31], the probability that two non-equivalent polynomials of degree $\delta$ agree on a point chosen uniformly at random is $\leq d/2^l$ in this case, which is negligible in $l$. Also, by assumption $\Pr[E_V] \geq \epsilon$.

We argue about $\Pr[E_2]$ as follows:

**Claim 2** $\Pr[E_2] > 1 - \nu_2(l)$ .

**Proof of Claim.** Equivalently, we will prove that for all internal nodes $v \in \mathcal{T}$, with corresponding extracted polynomial $P_v(r)$, it must be that $P_v(r) \approx_a \mathcal{C}_O$ where $O$ is the correctly computed set corresponding to $v$ when computing $q$ over $S_i$, with all but negligible probability.

As in the proof of Lemma 4, we will prove this by an induction on the levels of $\mathcal{T}$ (in fact, since $\mathcal{T}$ is not a balanced tree, the induction is over the nodes themselves in the order they are accessed by a DFS traversal).

**level-1** If the operation for $v$ is a union, the claim immediately holds from Claim 1 above, for the tree $\mathcal{T}_v$ corresponding to the union operations defined in $v$ over its children.
If the operation for $v$ is an intersection, then if $P_v(r)$ has a factor that is an irreducible polynomial of degree $> 1$, then let $R_i(r), \tilde{R}_i(r)$ be the corresponding extracted polynomials for the pair of values $W_i, \tilde{W}_i$ in the proof. Since the verification process succeeds, it follows that $e(f_O, W_i) = e(f_i, g)$. Since by assumption, $f(\mathcal{C}_{R_i}) = f_i$, (slightly abusing the notation, we assume that $S_i = R_i$) it follows that the polynomials $\mathcal{C}_{R_i}(r), P_v(r) \cdot R(r)$ form a collision for the ECRH for some index $i$. On the other hand, if $P_v(R)$ can be written as a product of degree 1 polynomials, it follows that it can be written as $\lambda\mathcal{C}_X$ for some set $X$ and $\mathcal{A}'$ could output appropriate proof for the claim $\cap_{i=1}^{t_v} R_i = X$, in the exact same manner as we demonstrated in proof of Lemma 4, which can only happen with negligible probability and this concludes the base case of the induction.

**general step** Let us assume that the statement holds for all the children of node $v$, we show it also holds for $v$. Assuming there exists such node $v$, we can separate into two cases.
If the operation at $v$ is an intersection, then let $Q_1(r), ..., Q_{t_v}(r)$ be the extracted polynomials corresponding to its children nodes. By assumption $Q_i(r) = \lambda_i\mathcal{C}_{O_i}$ where $O_i$ are the correctly computed sets up to that point according to $q$. Similar as for the case for level-1, if $P_v(r)$ contains a factor that is an irreducible polynomial of degree $> 1$, $\mathcal{A}'$ can find a collision in the ECRH. Therefore, with all but negligible probability, $P_v(r)$ can be written as $\lambda\mathcal{C}_X$ for some efficiently computable set $X = \{x_1, ..., x_{|X|})\}$. Hence $\mathcal{A}'$ can output $\{O_1, ..., O_{t_v}, X, \Pi^* = (h_X^{\lambda^{-1}}, W_i^{\lambda/\lambda_i}, F_i^{\lambda_i/\lambda}; i = 1, ..., t_v)\}$. It is easy to verify that the above is a satisfying proof for the claim $X = \cap_{i=1}^{t_v} O_i$ which by Lemma 4 can happen with negligible probability.
If the operation at $v$ is a union, then we argue as follows. Let $\mathcal{T}_v$ be the tree corresponding to the union operations defined in $v$ over its children. Observe that the only difference between this case and the case analysed previously in the proof of Lemma 4 is that the polynomials at the leafs of tree $\mathcal{T}_v$ are not characteristic polynomials necessarily. However, by assumption, they are polynomials of the form $\lambda_i\mathcal{C}_{O_i}$ where $O_i$ are the correctly computed sets up to that point according to $q$. $\mathcal{A}'$ can produce a satisfying proof for an incorrect set, in the *exact* same manner as described in the general step of our induction proof for Claim 1 above. Hence, with all but negligible probability, $P_v(r) \approx_a \mathcal{C}_O$, which concludes our induction proof.

Therefore, the claim follows. ■

It follows by the way we defined these events that the overall abort probability of $\mathcal{A}'$ is (using a union bound) $\Pr[\text{abort}] \leq \Pr[E_V^c] + \Pr[E_1^c] + \Pr[E_2^c] + \Pr[E_3^c] = 1 - \epsilon + \nu'(l)$ where $\nu(l)'$ is the sum of the three negligible probabilities. Hence $\Pr[\neg\text{abort}] \geq 1 - 1 - \epsilon + \nu'(l) = \epsilon - \nu'(l)$. We showed above that $\Pr[E'] \geq (1 - \nu^*(l)) \Pr[\neg\text{abort}] \geq \epsilon(1 - \nu(l))$ (for an appropriately defined negligible function $\nu(l)$) which is non-negligible. This contradicts the collision resistance of the ECRH $h$ and the security of $\mathcal{AHSO}$ follows. □

**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 $m^\epsilon$ proofs $\pi_i$ 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.*

For Corollary 1 the following modifications are made to the scheme:

- The server upon receiving $D, auth_D, d, pk$ computes and stores $m$ proofs $\pi_1, .., \pi_m$ by running the algorithm *VerifyTree* for each value $f_i$ corresponding to $S_i$. These values are computed in time $m^{1+\epsilon} \log m$.
- Upon receiving a query request, the server performs $t$ lookups to find the corresponding proofs $\pi_i$ (instead of computing them on-the-fly) and includes them in the proof.
- Upon receiving an update, modifying $f_i \rightarrow f_i^*$, let $\pi_1, ..., \pi_{m^\epsilon})$ be the proofs that corresponds to the value $f_i$ and its $m^\epsilon - 1$ siblings in the accumulation tree. The server computes updated proofs $\pi_1^*, ..., \pi_{m^\epsilon}^*$ by running *QueryTree* $m^\epsilon$, hence this takes overall time $m^{2\epsilon} \log m$.

Likewise for Corollary 2:

- Upon receiving query $q$, the server runs **query** skipping step (4).
- Upon receiving $\alpha, \Pi$, the source computes $\prod_{i=1}^{\delta}(x_i + s)$ in time $O(\delta)$ using the secret key $s$. He then runs **verify** replacing steps (3),(4) with a single check of the equality $e(g^{\prod_{i=1}^{\delta}(x_i+s)}, g) = e(f_a, g)$.

### 3.4 Complexity Analysis for the algorithms of the scheme

Recall that we are using the *access complexity* model and we are measuring primitive operations in $\mathbb{Z}_p^*$ ignoring the cost (that in practice is $\tilde{O}(l)$ for a security parameter $l$).

#### Intersection

This is the most complicated argument in terms of asymptotic analysis and it will be useful for the consecutive ones, therefore we will provide an elaborate analysis. The algorithm *proveIntersection* consists of the following steps:

1. Compute values $W_i$ for $i = 1, ..., t$.
2. Compute polynomials $q_i(r)$.
3. Compute values $F_i$.

For simplicity of presentation, we will assume without loss of generality that all $t$ sets have cardinality $n$ and we denote $N = tn$. From Lemma 2 step (1) can be done with $\sum_{i\in[t]} n \log n$ operations which can be bound by $O(N \log N)$.[8]

For the greatest common divisor computation, we will be making use of the *extended Euclidean algorithm* presented in [33] which, for two polynomials $a(r), b(r)$ of degree $n$ runs in time $O(n \log^2 n \log \log n)$. The algorithms outputs three polynomials $u(r), v(r), g(r)$ s.t. $u(r)a(r) + v(r)b(r) = g(r)$ and $g(r)$ is the $gcd(a(r), b(r))$ and $u, v$ are known as Bezout coefficients of $a, b$. Observe that $g(r)$ can be at most of degree $n$ and by the analysis of the algorithm, $deg(u) < deg(b) - deg(g)$ and $deg(v) < deg(a) - deg(g)$. In our case, it is thus true that the degrees of polynomials $u, v, g$ are all upper bounded by $n$.

The $gcd(P_1, ..., P_t)$ can be recursively computed as $gcd(gcd(P_1, ..., P_{t/2}), gcd(P_{t/2+1}, ..., P(t))$ and this can be applied repeatedly all the way to first computing the pairwise $gcd$ of all consecutive pairs of polynomials and following that the $gcd$ of each pair of $gcd's$ all the way to the top. In order to better analyse the complexity of step (2), let us introduce the following conceptual construction that captures exactly this recursive approach. Let $\mathcal{T}$ be a binary tree with polynomials $\mathcal{C}_{S_i \setminus I}$ at the $t$ leafs. Each internal node is associated with one major polynomial which is the $gcd$ of the major polynomials of its two children nodes, and two minor polynomials, which are the corresponding Bezout coefficients. The tree must be populated (all polynomials of internal nodes computed) as follows. For the nodes that are parents of leafs, compute the $gcd$ of their children nodes and the corresponding Bezout coefficients. Following that, for each level of the tree all the way up to the root, the nodes are populated by computing the $gcd$ of the $gcd$'s stored in their two children nodes. It follows that the root of $\mathcal{T}$ stores the $gcd(\mathcal{C}_{S_1 \setminus I}, ..., \mathcal{C}_{S_t \setminus I})$.

[8] A tighter bound would be $O(N \log n)$. However we do not wish to capitalize on the fact that we assumed all sets are of the same size, since this is an assumption for ease of notation. Hence we provide this more general bound.

Let us now analyse how long it takes to populate the nodes of $\mathcal{T}$. By the analysis of the extended Euclidean algorithm, it follows that each of the nodes that are parents of leafs can be populated in time $O(n \log^2 n \log\log n)$. Since the degrees of the *gcd* polynomials higher in $\mathcal{T}$ can only be lower, it follows that the same bound holds for all nodes. Since there exist $O(t)$ nodes, $\mathcal{T}$ can be populated in time $O(N \log^2 N \log\log N)$.

Following that, we need to compute polynomials $q_i(r)$. Observe that each such polynomial can be computed after populating $\mathcal{T}$ as the product of exactly $O(\log t)$ polynomials each of which can be at most of degree $n$. We start by proving the following.

**Claim 3** *Having populated $\mathcal{T}$, all the polynomials $q_i(t)$ for $i = 1, ..., t$ can be computed by $2t-2$ polynomial multiplications.*

**Proof of Claim.** We will prove the above by induction on the number of sets $t$. For $t = 2$, having populated the tree, polynomials $q_1(r), q_2(r)$ are already stored at the root. Hence we need $2 \cdot t - 2 = 0$ multiplications. If this is true for $t = j$ we will show it is true for $2j$. Observe that for two sibling sets, the polynomials $q_i(r), q_{i+1}(r)$ can be written as $q_i = h(r)u(r)$ an $q_{i+1} = h(r)v(r)$ where $u, v$ are the corresponding Bezout coefficients stored in their parent. The polynomials $h_k(r)$ for $k = 1, ..., j$ (each associated with one grand-parent node of the leafs in $\mathcal{T}$) can be computed with $2j - 2$ multiplications by the assumption. Hence each polynomial $q_i(r)$ can be computed with one additional multiplication for a total of $2j$ additional multiplications. Thus the overall number of multiplications to compute $q_1(r), ..., q_{2j}(r)$ is $4j - 2 = 2t - 2$, which concludes our proof of the claim. ■

Since each of $q_i(r)$ can be at most of degree $O(n \log t)$, an upper bound on the complexity of each of these multiplications is $O((n \log t) \log(n \log t))$, by using fast multiplication with FFT interpolation. By the above claim, there are $O(t)$ such multiplication therefore the overall complexity for the computation of the polynomials $q_i(r)$ is $O(N \log N \log t \log\log t)$. Finally, the output of this procedure is the polynomial coefficients of the $q_i$'s hence the values $F_i$ can be computed in time $O(N \log t)$ since each $q_i$ has degree at most $n \log t$. Since $t \leq N$, from the above analysis the overall complexity of *proveIntersection* is $O(N \log^2 N \log\log N)$.

Algorithm *verifyIntersection* consists of $O(t)$ bilinear pairings. Finally the size of the proof $\Pi$ is $O(t)$ group elements (in practice $2t$ elements).

### Union

We begin with the proof $\Pi$ for a union of two sets $A, B$ with cardinalities $n_A, n_B$ (denote $N = n_A + n_B$). The intersection argument for $I = A \cap B$ can be computed in time $O(N \log^2 N \log\log N)$ from the above analysis. The value $h_U$ can be computed in time $O(N \log N)$ from Lemma 2, hence the algorithm *proveUnion* for two sets runs is time $O(N \log^2 N \log\log N)$.

For the general case, let use denote with $n_i$ the cardinality of each set $S_i$ and let $N = \sum_{i \in [t]} n_i$. Finally we denote with $N_v$ the sum of the cardinalities of the sets of its children nodes of each node $v \in \mathcal{T}$. Each of the first level nodes is related to value $N_i$ for $i = 1, ..., t/2$ s.t. $\sum_{i=1}^{t/2} N_i \leq N$). Hence computing the proofs for all first level nodes of $\mathcal{T}$ can be done in time $\sum_{i=1}^{t/2} N_i \log^2 N_i \log\log N_i$ which can be upper bound by $O(N \log^2 N \log\log N)$. Moreover, this bound is true for all levels of $\mathcal{T}$ since due to the commutativity of the union operation, no elements will be left out (in the worst case the sets are disjoint, hence $|U| = N$) and since we have exactly $\log t$ levels in the tree, the algorithm *proveUnion* in general runs in time $O(N \log^2 N \log\log N \log t)$.

Each proof for a pair of sets can be verified by checking $O(1)$ bilinear equalities and since there are exactly $t - 1$ such arguments, the runtime of *verifyUnion* is $O(t)$. The proof for each node $v$ consists of 8 group elements and there are $t - 1$ such arguments, hence the size of the argument is $O(t)$ (in practice, $8(t - 1)$ elements).

**Hierarchical Set Operations**

Observe that (similar to the generalised union case) the proof construction and verification consists of constructing (and verifying) a series of proofs as dictated by the structure of $\mathcal{T}$. Hence the complexity of the algorithms will be characterized from the complexity of the algorithms for the single operation case. As before we denote $N_v$ for each node $v \in \mathcal{T}$ as the sum of the cardinalities of the sets of its children nodes and $t_v$ as the number of its children nodes. Also let $N = \sum_{v \in \mathcal{T}} N_V$. The construction of the argument for each node can be made in time $O(N_v \log^2 N_v \log\log N_v \log t_v)$. If $t$ is the length of the set operation formula corresponding to $q$, it follows that $t_v \leq t$ hence the above can be also bound as $O(N_v \log^2 N_v \log\log N_v \log t)$. Finally the cost to compute $\Pi$ is equal to the sum of computing all of the respective proofs, which can be written as $O(N \log^2 N \log\log N \log t)$. Also, each of the proofs $\pi_i$ is computed in time $O(m^\epsilon \log m)$ and since there are $t$ of them, the overall complexity for **query** is $O(N \log^2 N \log\log N \log t + tm^\epsilon \log m)$.

Each proof can be verified by checking $O(t_v)$ bilinear equalities. Since each node has a single parent it follows that the runtime of **verify** is $O(|\mathcal{T}|)$. However, $|\mathcal{T}| \leq 2t$ since all operations are defined over at least two sets, hence **verify** consists of $O(t)$ operations. Each atomic proof in $\Pi$ consists of $O(t_v)$ group elements and therefore the total size of $\Pi$ is $O(t + \delta)$.

# 4 Extensions and Implementation Decisions

**Reducing the proof size.** The size of proof $\Pi$ can be reduced to being independent of the size of the final answer $\alpha$. Observe that what makes the proof be of size $O(t + \delta)$ is the presence of coefficients $\mathbf{c}$. However, given $\alpha$ itself, coefficients $\mathbf{c} = (c_0, ..., c_{\delta-1})$ can be computed using an FFT algorithm in time $O(\delta \log \delta)$. An alternative to the above scheme would be to omit $\mathbf{c}$ from the proof and let the verifier upon input $\alpha$ compute the coefficients by himself to run the last step of **verify**. That would give a proof size of $O(t)$ and verification time of $O(t + \delta \log \delta)$. Since in most real world applications $\delta \gg t$, a proof that has size independent of $\delta$ is useful, especially if one considers that the additional overhead for verification is logarithmic only. Of course the communication bandwidth is still $O(t + \delta)$ because of the answer size, but it does not extend to the proof size.

**A note on the public key size.** A downside of our construction -and all other constructions that are provably secure under a $q$-type assumption- is the large public key size. More specifically, the public key $pk$ is of size linear to the parameter $q$ where $q$ is an upper bound on the size of the sets that can be hashed. This holds not only for the original sets $S_1, ..., S_m$ but for any set that can result from hierarchical set operations among them thus a natural practical bound for $q$ is $|D|$. While computing this public key cannot be avoided and it is necessary for proof computation at the *server*, a *client* that needs to verify the correctness of query $q$ with corresponding answer $\alpha$ of size $\delta$, only needs the first $max\{t, \delta\}$ elements of the public key. By deploying an appropriate authentication mechanism (digital signatures, Merkle trees, accumulation trees etc.) to validate the elements of $pk$, a scheme that relieves clients from the necessity to store a long public key can be constructed. Ideally the necessary public key elements should be transmitted alongside proof $\Pi$ and cached or discarded at the behest of the client.

**Symmetric vs. Asymmetric pairings.** Throughout the presentation of our scheme, we assumed implicitly that the pairing $e(\cdot, \cdot)$ is symmetric (i.e., Type-1 pairing). For example for the construction of the union argument for the operation $A \cup B$, the value $f_B$ appears both in term $e(f_A, f_B$ and term $e(f_B, g)$ and we assumed that in both cases the same value is used as input for the pairing, as is the case if $e$ is symmetric. However, many times asymmetric pairings are preferable for implementation purposes since they are much faster than asymmetric ones in terms of computation. This is not a serious problem for our scheme as there is an easy way to circumvent it.

A pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is asymmetric if $\mathbb{G}_1 \neq \mathbb{G}_2$ but both are of prime order $p$ and let $g_1, g_2$ be respective generators. Observe that $e(g_1^{P(s)}, g_2) = e(g_2, g_2^{P(s)})$ is an efficiently checkable equality that

verifies that two hash values (their first parts) $f^1 = g_1^{P(s)}, f^2 = g_2^{P(s)}$ have the same pre-image but are computed in $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively. Therefore, by including both values $f_A^1, f_A^2$ in the proof, the case of an asymmetric pairing can be accommodated. By verifying the above equality a prover can be sure that both values refer to the same characteristic polynomial and use either one of them selectively, as dictated by the argument verification algorithm. By using the naive approach of including the "dual" hash value of each element in the proof, we see that the proof size can at most double but maintains the same asymptotic behaviour, i.e., proofs have size $O(t+\delta)$ and the same holds for the runtime of the verification algorithm. In practice, a smarter approach can be taken where only necessary elements are added (the ones that participate in union arguments and, of these, half can be "routed" through $\mathbb{G}_1$ and the other half through $\mathbb{G}_2$). Another by-product of using an asymmetric pairing it that the public key size is doubled $(g_1, ..., g_1^{s^q}, ..., g_2, ..., g_2^{s^q})$ and likewise for the setup phase cost for the source. Note that no isomorphism between $\mathbb{G}_2$ and $\mathbb{G}_1$ is explicitly used in the above process, hence our construction can work both with Type-2 and Type-3 pairings.

# References


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

[2] G. Ateniese, E. D. Cristofaro, and G. Tsudik. (if) size matters: Size-hiding private set intersection. In *Public Key Cryptography*, pages 156–173, 2011.

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

[4] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza. Snarks for c: Verifying program executions succinctly and in zero knowledge. In *CRYPTO (2)*, pages 90–108, 2013.

[5] E. R. Berlekamp. Factoring polynomials over large finite fields*. In *Proceedings of the second ACM symposium on Symbolic and algebraic manipulation*, SYMSAC '71, pages 223–, New York, NY, USA, 1971. ACM.

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

[7] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. Recursive composition and bootstrapping for snarks and proof-carrying data. *IACR Cryptology ePrint Archive*, 2012:95, 2012.

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

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

[10] D. Boneh and X. Boyen. Short signatures without random oracles. In *EUROCRYPT*, pages 56–73, 2004.

[11] D. Boneh and B. Waters. Conjunctive, subset, and range queries on encrypted data. In *TCC*, pages 535–554, 2007.

[12] E. Boyle and R. Pass. Limits of extractability assumptions with distributional auxiliary input. Cryptology ePrint Archive, Report 2013/703, 2013.

[13] J. Camenisch and A. Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In *CRYPTO*, pages 61–76, 2002.

[14] S. Chatterjee and A. Menezes. On cryptographic protocols employing asymmetric pairings - the role of revisited. *Discrete Applied Mathematics*, 159(13):1311–1322, 2011.

[15] K.-M. Chung, Y. T. Kalai, F.-H. Liu, and R. Raz. Memory delegation. In *CRYPTO*, pages 151–168, 2011.

[16] I. Damgård. Towards practical public key systems secure against chosen ciphertext attacks. In *CRYPTO*, pages 445–456, 1991.

[17] I. Damgård, S. Faust, and C. Hazay. Secure two-party computation with low communication. In *TCC*, pages 54–74, 2012.

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

[19] M. J. Freedman, K. Nissim, and B. Pinkas. Efficient private matching and set intersection. In *EUROCRYPT*, pages 1–19, 2004.

[20] M. T. Goodrich, R. Tamassia, and N. Triandopoulos. Efficient authenticated data structures for graph connectivity and geometric search problems. *Algorithmica*, 60(3):505–552, 2011.

[21] J. Groth. Short pairing-based non-interactive zero-knowledge arguments. In *ASIACRYPT*, pages 321–340, 2010.

[22] L. Kissner and D. X. Song. Privacy-preserving set operations. In *CRYPTO*, pages 241–257, 2005.

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

[24] R. C. Merkle. A certified digital signature. In *CRYPTO*, pages 218–238, 1989.

[25] M. Naor and K. Nissim. Certificate revocation and certificate update. *IEEE Journal on Selected Areas in Communications*, 18(4):561–570, 2000.

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

[27] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Authenticated hash tables. In *ACM Conference on Computer and Communications Security*, pages 437–448, 2008.

[28] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Optimal verification of operations on dynamic sets. In *CRYPTO*, pages 91–110, 2011.

[29] B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symposium on Security and Privacy*, pages 238–252, 2013.

[30] F. Preparata, D. Sarwate, and I. U. A. U.-C. C. S. LAB. *Computational Complexity of Fourier Transforms Over Finite Fields*. Defense Technical Information Center, 1976.

[31] J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, 27(4):701–717, 1980.

[32] R. Tamassia. Authenticated data structures. In *ESA*, pages 2–5, 2003.

[33] J. von zur Gathen and J. Gerhard. *Modern computer algebra (2. ed.)*. Cambridge University Press, 2003.

[34] Y. Yang, D. Papadias, S. Papadopoulos, and P. Kalnis. Authenticated join processing in outsourced databases. In *SIGMOD Conference*, pages 5–18, 2009.

[35] M. L. Yiu, Y. Lin, and K. Mouratidis. Efficient verification of shortest path search via authenticated hints. In *ICDE*, pages 237–248, 2010.

[36] Q. Zheng, S. Xu, and G. Ateniese. Efficient query integrity for outsourced dynamic databases. *IACR Cryptology ePrint Archive*, 2012:493, 2012.