Taking Authenticated Range Queries to Arbitrary Dimensions

Dimitrios Papadopoulos
Boston University
Boston MA, USA
dipapado@bu.edu

Stavros Papadopoulos
Intel Labs & MIT
Cambridge MA, USA
stavrosp@csail.mit.edu

Nikos Triandopoulos
RSA Laboratories &
Boston University
Cambridge MA, USA
nikolaos.triandopoulos@rsa.com

ABSTRACT

We study the problem of authenticated multi-dimensional range queries over outsourced databases, where an *owner* outsources its database to an untrusted *server*, which maintains it and answers queries to *clients*. Previous schemes either scale *exponentially* in the number of query dimensions, or rely on heuristic data structures without provable bounds. Most importantly, existing work requires an *exponential*, in the database attributes, number of structures to support queries on every possible combination of dimensions in the database. In this paper, we propose the first schemes that (i) scale *linearly* with the number of dimensions, and (ii) support queries on *any* set of dimensions with *linear* in the number of attributes setup cost and storage. We achieve this through an elaborate fusion of novel and existing *set-operation* sub-protocols. We prove the security of our solutions relying on the q-Strong Bilinear Diffie-Hellman assumption, and experimentally confirm their feasibility.

Categories and Subject Descriptors

H.2.7 [**Database Management**]: Database Administration—*security, integrity and protection*; C.2.4 [**Communication Networks**]: Distributed Systems—*client/server, distributed databases*

General Terms

Algorithms, Security, Verification

Keywords

authenticated range queries; authenticated data structures; delegation of computation; database outsourcing

1. INTRODUCTION

Database outsourcing [16] has lately emerged as a common practice for companies and institutions. It allows a data *owner* to delegate the maintenance and administration of its database to a powerful third-party *server*. *Clients* access the database by contacting the server instead of the owner. This paradigm reduces the owner's needs for building a sophisticated and potentially costly infrastructure, since storage and computationally intensive tasks are offloaded to the server. On the other hand, the server profits from accommodating a large number of owners. Despite its merits, database outsourcing poses the challenge that the server may be *untrusted* and, thus, tamper with the results (e.g., to bias the competition among rival serviced companies). Hence, it is vital that the server provides the client with proofs certifying the result *integrity*, i.e., that the query was executed correctly on the owner's data.

In this work we target the case where the client issues a *multi-dimensional range query*. We model the owner's database as a table T that contains n tuples with m attribute values. A range query is defined over d out of the m attributes, which we refer to as dimensions. It is expressed as d pairs of values l_i, u_i, each along a certain dimension a_i. Its result includes all the tuples whose value on a_i is in range $[l_i, u_i]$ for *all* dimensions a_i specified in the query. This query is fundamental in a vast variety of applications. For instance, it is a typical `select`...`where` query in conventional relational databases. Moreover, it is a frequent query in the emerging scientific databases (e.g., it is called `subarray` in SciDB [7]). Relational and scientific database systems manage numerous types of data, such as corporate, stock, astronomical, medical, etc. With the advent of "big data," such systems are commonly deployed by third party servers in massively parallel architectures, in order to address the issue of scalability. Integrity assurance is a desirable property that serves both as a guarantee against a possibly malicious server, but also as a tool for error detection.

Prior work. The most basic authentication problem is *set membership*, i.e., whether an element belongs in a data collection. Well-known example schemes include Merkle trees [19] and accumulation trees [22]. At the opposite extreme, there exist generic approaches (e.g., [11, 25]) that aim at authenticating any possible query on outsourced data. Although such protocols can address our problem, they incur an excessive proof construction overhead at the server, due to their generality. Therefore, there is a large variety of specialized constructions that have been proposed in the literature for the problem of authenticated multi-dimensional range queries.

Martel et al. [18] provide a generalization of Merkle trees, which captures the case of multi-dimensional range queries. Chen et al. [9] proposes a solution that is similar to [18], based on attribute domain partition and access control. For the restricted case of 1-dimensional and 2-dimensional queries, Goodrich et al. [14, 15] construct schemes based on cryptographic extensions of Merkle trees. Li et al. [17] propose a variant of the B$^+$-tree that incorporates hash values similarly to the Merkle tree, for processing 1-dimensional queries in external storage. Yang et al. [28] extend this idea to multiple dimensions, by transforming the R*-tree [4] into a

CCS'14, November 3–7, 2014, Scottsdale, Arizona, USA.
Copyright 2014 ACM 978-1-4503-2957-6/14/11 ...$15.00.
http://dx.doi.org/10.1145/2660267.2660373 .

Scheme	**Setup**	**Proof size**	**Proof construction**	**Verification**	**Update**
Martel et al. [18]	$O(\lvert T\rvert \log^{m-1} \lvert T\rvert)$	$O(\log^{d-1} \lvert T\rvert)$	$O(\log^{d-1} \lvert T\rvert)$	$O(\log^{d-1} \lvert T\rvert)$	$O(\log^{m-1} \lvert T\rvert)$
Chen et al. [9]	$O(\lvert T\rvert \log^{m} N)$	$O(\log^{d} N)$	$O(\log^{d} N)$	$O(\log^{d} N)$	$O(\log^{m} N)$
Our basic scheme (using [19])	$O(\lvert T\rvert \log n)$	$O(d \log n)$	$\tilde{O}(\sum_{i=1}^{d} \lvert R_i\rvert) + O(d \log n)$	$\tilde{O}(\lvert R\rvert) + O(d \log n)$	$O(\lvert T\rvert)$
Our basic scheme (using [22])	$O(\lvert T\rvert \log n)$	$O(d)$	$\tilde{O}(\sum_{i=1}^{d} \lvert R_i\rvert) + O(dn^{\epsilon} \log n)$	$\tilde{O}(\lvert R\rvert) + O(d)$	$O(\lvert T\rvert)$
Our update-efficient scheme (using [19])	$O(\lvert T\rvert \log n)$	$O(d \log n)$	$\tilde{O}(\sum_{i=1}^{d} \lvert R_i\rvert) + O(d \log n)$	$\tilde{O}(\lvert R\rvert) + O(d \log n)$	$O(m\sqrt{n})$
Our update-efficient scheme (using [22])	$O(\lvert T\rvert \log n)$	$O(d)$	$\tilde{O}(\sum_{i=1}^{d} \lvert R_i\rvert) + O(dn^{\epsilon} \log n)$	$\tilde{O}(\lvert R\rvert) + O(d)$	$O(m\sqrt{n})$

m: # attributes, n: # tuples, $|T|(= mn)$: database size, d: # dimensions, R_i: partial result at dimension a_i, R: query result, N: maximum domain size, $\epsilon \in (0, 1]$

Table 1: Comparison of asymptotic complexities of authenticated multi-dimensional range query schemes

Merkle R*-tree. There are also other cryptographically augmented data structures (e.g., [20, 10] based on signatures instead of hashes).

The existing literature suffers from the following critical problems. On the one hand, the schemes of [18, 9] that provide guaranteed (non-trivial) complexity bounds, scale *exponentially* with the number of dimensions d. On the other hand, the rest of the approaches rely on the heuristic R*-tree and fail to accommodate more than a limit of dimensions in practice (e.g., more than 8), as the performance and effectiveness of the index deteriorates with dimensionality. Most importantly, *all* methods require an *exponential* in m number of structures to support queries on every possible combination of dimensions in the database. This is because each structure is built on a *specific* set of dimensions, and different sets require separate structures.

Finally, there is a work by Xu [27] that, contrary to [18, 9], scales quadratically with d. However, this scheme falls within a different model, as it necessitates multi-round interaction between server and client (as opposed to our non-interactive setting). Its security is based on non-falsifiable "knowledge-type" assumptions. Moreover, this scheme makes use of functional encryption [6], considerably reducing its potential for implementation.

Our contributions. We introduce two schemes for authenticated multi-dimensional range query processing; a basic, and a update-efficient. Our solutions feature two novel and powerful properties:

- They are the first schemes where *all* costs (i.e., setup, storage, update, proof construction, verification, and proof size) grow only *linearly* with the number of dimensions, a huge improvement over the current literature. Table 1 provides a comparison of our asymptotic complexities against known schemes (with non-trivial bounds).
- They are the first to support an *exponential* in m number of range queries with *linear* in m setup cost and storage.

In that sense, the main result of this work is that it takes authenticated range query processing to *arbitrary* dimensions, both in terms of number and choice.

The central idea of our solutions is the reduction of the multi-dimensional range query to *set-operations* over appropriately defined sets in the database. In particular, in a one-time setup stage, the owner builds a novel authenticated structure over every database attribute *separately*, and then binds all structures using an existing set-membership structure (e.g., [19, 22]). Given a query involving *any* set of dimensions, the server decomposes it into its d 1-dimensional ranges, and processes them *individually* on the structure of each dimension, producing d proofs for the partial results $R_1, \ldots, R_d$. The main challenge is for these d proofs to (i) be *combinable* such that they verify the *intersection* of R_i, which is the final result R, and (ii) be *verifiable without the partial results*, so that the total proof size and verification cost are independent of their (potentially large) sizes. We address this challenge through an elaborate fusion of existing and novel *intersection*, *union*, and *set-difference* protocols, based on bilinear accumulators.

This particular treatment of the problem, i.e., the efficient authentication of a d-dimensional range query via the combination of d separate 1-dimensional proofs, would not be feasible without the recent advances in set-operation authentication (e.g., [24, 8]). We anticipate that future research will substantially improve the efficiency of the set-operation sub-protocols. Motivated by this, as an additional important contribution, we identify and abstract the set-operation sub-protocols needed as building blocks in our schemes, and formulate a general framework that can integrate any future improved machinery for set-operation authentication.

We formally prove our constructions secure under the q-Strong Bilinear Diffie-Hellman [5] assumption and the security of the underlying set-membership schemes. We also provide an experimental evaluation, demonstrating the feasibility of our schemes.

Roadmap. Section 2 contains the necessary cryptographic background. Section 3 formulates our problem. Section 4 presents our basic scheme for authenticated multi-dimensional range query processing, whereas Section 5 introduces an alternative construction with optimized updates. Section 6 provides a thorough experimental evaluation of our solutions. Finally, Section 7 concludes our paper with directions to future work.

2. CRYPTOGRAPHIC BACKGROUND

In the following, λ denotes the security parameter, $\nu(\lambda)$ a negligible function, and PPT a probabilistic polynomial time algorithm. In complexity analysis, we also use $\tilde{O}$ notation that hides a polylogarithmic multiplicative factor. Moreover, for proof sizes, we omit the factor imposed by the bit representation of group elements.

Bilinear pairings. Let $\mathbb{G}_1, \mathbb{G}_2$ be cyclic multiplicative groups of prime order p, generated by g_1, g_2 respectively. Let also $\mathbb{G}_T$ be a cyclic multiplicative group with the same order p and $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ be a bilinear pairing with the following properties: (1) *Bilinearity*: $e(P^a, Q^b) = e(P, Q)^{ab}$ for all $P, Q \in \mathbb{G}_1 \times \mathbb{G}_2$ and $a, b \in \mathbb{Z}_p$. (2) *Non-degeneracy*: $e(g_1, g_2) \neq 1$. (3) *Computability*: There is an efficient algorithm to compute $e(P, Q)$ for all $P, Q \in \mathbb{G}_1 \times \mathbb{G}_2$. In the sequel, for ease of presentation, we will assume a *symmetric* (Type I) pairing, where $\mathbb{G}_1 = \mathbb{G}_2 = \mathbb{G}$. We denote with $pub = (p, \mathbb{G}, \mathbb{G}_T, e, g)$ the bilinear pairings parameters, output by a PPT algorithm BilGen on input 1^λ. We will make use of the following assumption over bilinear groups:

ASSUMPTION (q-Strong Bilinear Diffie-Hellman (q-SBDH) [5]). *Let λ be a security parameter and let $pub \leftarrow \mathsf{BilGen}(1^\lambda)$. Given $(pub, g^s, \ldots, g^{s^q})$ where q is $\mathsf{poly}(\lambda)$, for some s chosen at random from $\mathbb{Z}_p^*$, there is no PPT algorithm that can output a pair $(c, e(g, g)^{1/(c+s)}) \in \mathbb{Z}_p^* \times \mathbb{G}_T$ except with probability $\nu(\lambda)$.*

Collision-resistant hash functions. A *collision-resistant hash function* (CRHF) H is a function randomly sampled from a function family, such that no non-uniform PPT algorithm can output x, x', such that $H(x) = H(x')$ and $x \neq x'$, except with probability $\nu(\lambda)$.

Set-membership authentication. Consider a data owner outsourcing a set X to an untrusted third-party server. Clients issue queries about a single element $x \in X$. A set-membership authentication protocol ($\mathcal{SMA}$) allows the server to prove to a client that x is indeed a member of X. An $\mathcal{SMA}$ is a collection of algorithms KeyGen, Setup, Prove, Verify and Update. The owner executes Keygen and Setup prior to outsourcing X. The former generates a secret and public key pair sk, pk, whereas the latter produces a digest δ that is a succinct cryptographic representation of X. The owner keeps sk and publishes pk and δ. Given a client query about $x \in X$, the server runs Prove to produce a proof of membership π. Given pk, δ, π and x, the client runs Verify to check the membership of x in X. In case the owner modifies X by inserting/deleting elements, it executes Update to produce a new digest δ reflecting the updates in X, and notifies the server about the changes. An $\mathcal{SMA}$ is *secure*, if the probability that ($\texttt{accept} \leftarrow \mathsf{Verify} \ \wedge \ y \notin X$) is negligible.

The most well-known $\mathcal{SMA}$ is the *Merkle tree* [19], which is a binary tree where (i) each leaf node contains an element $x \in X$, and (ii) each non-leaf node stores the hash of the values of its children, using a CRHF H. During Setup, the owner builds a Merkle tree on X, signs the hash value in the root, publishes it as the digest δ, and sends the tree to the server. During Prove, the server accesses the tree to find x, and includes in proof π all sibling hash values along the path from the root to the leaf storing x. In Verify, the client recursively performs the hash operations to reconstruct the root hash value, and checks it against δ. Producing an element $y \notin X$ and a convincing proof is known to be as hard as finding a collision for H. For n elements, the proof construction and size, verification, and update time are all $O(\log n)$, whereas the setup is $O(n)$. An alternative $\mathcal{SMA}$ is the *accumulation tree* [22], which features two main differences to the Merkle tree: (i) the fanout of each non-leaf node is $n^{1/\epsilon}$, where $\epsilon \in (0, 1]$ is a user-defined parameter, and (ii) each non-leaf node stores an *accumulation value* (discussed below) produced over the values of its children. The following lemma (originally from [23]; slightly informal here) describes the security of the bilinear-pairing based accumulation tree.

LEMMA 1. *Let λ be a security parameter. Under q-SBDH, no PPT adversary on input pk output by* Keygen *and with oracle access to algorithms* Setup*,* Update*,* Prove *and* Verify *can produce set X, element $x \notin X$ and proof π, such that* Verify *outputs* accept*, except with probability $\nu(\lambda)$.*

We refer the interested reader to [22] and [24] for a more detailed description of the accumulation tree. This $\mathcal{SMA}$ offers $O(1)$ proof size, verification, and update time, and $O(n)$ setup cost. The downside is the proof construction overhead, which is now $O(n^\epsilon \log n)$, and the costly operations as opposed to the Merkle tree (exponentiations vs. hashings).

Set operation authentication. Consider now an owner of a collection of sorted sets $\mathcal{X} = \{X_1, \ldots, X_m\}$, who outsources them to an untrusted server. Clients issue queries describing set operations over $\mathcal{X}$, consisting of *unions*, *intersections*, and *set-differences*. Example queries include $X_1 \cap X_5$, $(X_2 \cup X_3) \cap X_1$, and $X_1 \setminus X_2$. A set operation authentication protocol ($\mathcal{SOA}$) enables the server to prove the integrity of the result. Similarly to $\mathcal{SMA}$, it is comprised of algorithms KeyGen, Setup, Prove, Verify Update, and its security is defined as the inability of the server to present a false answer with an accepting proof.

Existing $\mathcal{SOA}$ constructions appear in [24] and [8]. The former can support queries expressed as a *single* set operation (for instance $X_i \cap \ldots \cap X_j$, i.e., one intersection over an arbitrary number of sets). On the other hand, the latter accommodates *any circuit* of set operations, e.g., $(X_i \cap X_j) \cup X_l$. Both schemes offer the same asymptotic overhead. Specifically, for a single operation on collection $\mathcal{X}_Q \subseteq \mathcal{X}$ of d sets that produces result R, computable in time $\Omega(\sum_{X \in \mathcal{X}_Q} |X|)$, the proof is generated in time $\tilde{O}(\sum_{X \in \mathcal{X}_Q} |X|)$ and has size $O(d)$. Note that proof construction incurs only a polylogarithmic overhead compared to result computation, and this is generalized naturally for circuits of multiple operations in [8]. The verification overhead is $\tilde{O}(|R|) + O(d)$, whereas the setup cost is $O(\sum_{X \in \mathcal{X}} |X|)$. Although [8] subsumes [24] in terms of functionality, its security relies on non-standard "knowledge-type" assumptions.

We next describe the intersection scheme of [24], as we utilize it in our constructions. This scheme employs the *bilinear accumulator* primitive [21]. Let X be a set with elements from $\mathbb{Z}_p$, and $s \leftarrow_R \mathbb{Z}_p^*$ a secret. The *accumulation value* of X is defined as:

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

This value is a succinct, collision-resistant cryptographic representation of X under q-SBDH. It is also computable (from scratch) even without s, by having access to the public pairing parameters pub, as well as a public key $(g^s, \ldots, g^{s^q})$, where q is a user-defined parameter that is an upper bound on the cardinality of X. In particular, we can write $\prod_{x \in X}(x+S) = P_X(S) = \sum_{i=0}^{|X|} c_i S^i$, where S is an undefined variable. The coefficients $c_0, \ldots, c_{|X|}$ can be computed in time $O(|X| \log |X|)$ using FFT interpolation. One can compute $acc(X) = g^{P_X(s)} = \prod_{i=0}^{|X|} (g^{s^i})^{c_i}$ using only the public information. Note that, with access to s, the bilinear accumulator can accommodate an insertion/deletion in X with $O(1)$ operations [21]. However, without s, the updated accumulation value must be computed from scratch.

In order to prove to a client with access to $acc(X_1)$, $acc(X_2)$ that a set I is the intersection $X_1 \cap X_2$, it suffices to prove that (i) $I \subseteq X_1$ and $I \subseteq X_2$, and (ii) $(X_1 \setminus I) \cap (X_2 \setminus I) = \emptyset$. Towards (i), the server must send *subset witnesses* W_1, W_2 to the client, where $W_i = acc(X_i \setminus I)$ for $i = 1, 2$. To verify (i), the client first computes $acc(I)$, and checks the following for $i = 1, 2$:

$$e(acc(I), W_i) \stackrel{?}{=} e(acc(X_i), g) \ .$$

For (ii), the server computes two *disjointness witnesses* F_1, F_2 as follows. Since $(X_1 \setminus I) \cap (X_2 \setminus I) = \emptyset$, $P_{X_1 \setminus I}(S) = \prod_{x \in X_1 \setminus I}(x + S)$ and $P_{X_2 \setminus I}(S) = \prod_{x \in X_2 \setminus I}(x + S)$ have greatest common divisor of degree zero. Hence, there exist polynomials $Q_1(S), Q_2(S)$ such that $Q_1(S) \cdot P_{X_1 \setminus I}(S) + Q_2(S) \cdot P_{X_2 \setminus I}(S) = 1$. These polynomials (also known as Bézout coefficients) are efficiently computable by the Extended Euclidean algorithm. The server calculates the disjointness witnesses as $F_1 = g^{Q_1(s)}, F_2 = g^{Q_2(s)}$. To verify (ii), the client simply checks

$$e(W_1, F_1) \cdot e(W_2, F_2) \stackrel{?}{=} e(g, g) \ .$$

This approach naturally generalizes for $d > 2$ sets X_i, with corresponding *intersection proof* $\pi_\cap = \{W_i, F_i\}_{i=1}^d$. In our security proofs, we use the following lemma from [24]:

LEMMA 2. *Let λ be a security parameter, and $pub \leftarrow \mathsf{BilGen}(1^\lambda)$. Under q-SBDH, on input $(pub, g^s, \ldots, g^{s^q}) \in \mathbb{G}$ for some s chosen at random from $\mathbb{Z}_p^*$, no PPT adversary can output sets $X_1, \ldots, X_d, I$*

with elements in $\mathbb{Z}_p$, where $d = \mathsf{poly}(\lambda)$, and $\pi_\cap = \{W_i, F_i\}_{i=1}^d$, such that $e(acc(I), W_i) = e(acc(X_i), g)$, $\prod_i e(W_i, F_i) = e(g, g)$, and $I \neq \bigcap_i X_i$, for $i = 1, \ldots, d$, except with probability $\nu(\lambda)$.

3. PROBLEM FORMULATION

In this section we describe our targeted setting, formulate our authentication protocol, and model its security.

Setting and query. Our setting involves three types of parties; an *owner*, a *server*, and a number of *clients*. The owner outsources to the server a dataset T that consists of n tuples, each having a set $A = \{a_1, \ldots, a_m\}$ of attributes. This dataset can be perceived as a table in traditional relational databases. It could also be a multi-dimensional array in scientific databases (e.g., SciDB [7]), where a subset of the attributes are the array dimensions (i.e., the array indices), and the rest are the array attributes (i.e., the array cell values). In addition, the server is responsible for maintaining the dataset, upon receiving tuple updates (modeled as insertion/deletion requests) from the owner.

Clients issue *multi-dimensional range queries* on T to the server, which return the tuples from T that satisfy certain range conditions over a set of attributes. More formally, a query Q is specified over *any subset* of d attributes $A_Q \subseteq A$, where $|A_Q| = d \leq m$, and encoded by the set of triplets $\{(i, l_i, u_i)\}_{a_i \in A_Q}$. The result of Q is a set $R \subseteq T$ that contains exactly those tuples $t \in T$ that satisfy $l_i \leq t.a_i \leq u_i$ for all $a_i \in A_Q$. This query corresponds to a `select...where` query in relational databases, and a `subarray` query in scientific databases. In our terminology, each $a_i \in A_Q$ represents a dimension in the multi-dimensional range query.

In our setting, we consider that the server is *untrusted*, and may present to the client a tampered result. Our goal is to construct a protocol for authenticated multi-dimensional range queries, which allows the client to verify the integrity of the received result.

Authentication protocol. Let T_j denote the version of dataset T after j rounds of updates. An *authenticated multi-dimensional range query protocol* ($\mathcal{AMR}$) consists of the following algorithms:

1. $\mathsf{KeyGen}(1^\lambda)$: It outputs secret and public keys sk, pk.
2. $\mathsf{Setup}(T_0, sk, pk)$: It computes some authentication information $auth(T_0)$ and digest δ_0, given dataset T_0, sk and pk.
3. $\mathsf{Update}(upd, auth(T_j), \delta_j, sk, pk)$: On input update information upd on T_j, $auth(T_j)$, δ_j and sk, it outputs an updated dataset T_{j+1}, along with new $auth(T_{j+1})$, and δ_{j+1}.
4. $\mathsf{Prove}(Q, R, T_j, auth(T_j), pk)$: On input query Q on T_j with result R, and $auth(T_j)$, it returns R and proof π.
5. $\mathsf{Verify}(Q, R, \pi, \delta_j, pk)$: On input query Q, result R, proof π, digest δ_j and pk, it outputs either `accept` or `reject`.

In a pre-processing stage, the owner runs KeyGen and Setup. It publishes public key pk and digest δ_0, which is a succinct cryptographic representation of initial dataset T_0. Moreover, it sends $T_0, auth(T_0)$ to the server, where $auth(T_0)$ is some authentication information on T_0 that will be used by the server to construct proofs. The owners maintains its dataset by issuing Update when changes occur at the dataset. Specifically, an update is a tuple insertion or deletion, encoded by upd. An update on T_j produces a new version T_{j+1}, as well as new digest δ_{j+1} and $auth(T_{j+1})$. The owner sends to the server only the modified parts necessary for computing $T_{j+1}, auth(T_{j+1}), \delta_{j+1}$. The server responds to a query Q from the client by first computing the result R, and executes Prove that constructs the corresponding proof π. Finally, the client validates the integrity and freshness of R being a correct answer to Q on current T_j, by running Verify.

An $\mathcal{AMR}$ must satisfy the following two properties:

CORRECTNESS. *A $\mathcal{AMR}$ is* correct *if, for all $\lambda \in \mathbb{N}$, $(sk, pk) \leftarrow$ $\mathsf{KeyGen}(1^\lambda)$, all $(T_0, auth(T_0), \delta_0)$ output by one invocation of* Setup *followed by j' calls to* Update *on updates upd, where j' is* $\mathsf{poly}(\lambda)$*, any Q with correct result R, and π output by* $\mathsf{Prove}(Q, T_j, auth(T_j), pk)$, $\mathsf{Verify}(Q, R, \pi, \delta_j, pk)$ *returns* `accept` *with probability 1, for all $j \leq j'$.*

SECURITY. *Let $\lambda \in \mathbb{N}$ be a security parameter, key pair $(sk, pk) \leftarrow$* $\mathsf{KeyGen}(1^\lambda)$*, and $\mathcal{A}$ be a PPT adversary that possesses pk and has oracle access to all algorithms of $\mathcal{AMR}$. The adversary picks an initial state of the dataset T_0 and receives $T_0, auth(T_0), \delta_0$ through oracle access to* Setup*. Then, for $i = 0, \ldots, j' - 1 = \mathsf{poly}(\lambda)$, $\mathcal{A}$ issues an update upd_i for T_i and receives $T_{i+1}, auth(T_{i+1})$ and δ_{i+1} through oracle access to* Update*. At any point during these update queries, $\mathcal{A}$ can make polynomially many oracle calls to algorithms* Prove *and* Verify*. Finally, $\mathcal{A}$ picks an index $0 \leq j \leq j'$, a query Q, a result R^* and a proof π^*. We say that a $\mathcal{AMR}$ is* secure *if for all large enough $\lambda \in \mathbb{N}$ and all PPT adversaries $\mathcal{A}$, it holds that:*

$$\Pr\left[\begin{array}{c} (Q, R^*, \pi^*, j) \leftarrow \mathcal{A} \quad s.t \\ \mathtt{accept} \leftarrow \mathsf{Verify}(Q, R^*, \pi^*, \delta_j, pk) \\ \wedge R^* \neq R \end{array}\right] \leq \nu(\lambda) ,$$

where R is the correct result of Q on T_j, and the probability is taken over the randomness of the algorithms and the coins of $\mathcal{A}$.

As an additional remark, note that the above protocol falls within the framework of authenticated data structures, introduced in [26], but is tailored for the specific problem of range queries.

4. BASIC SCHEME

Section 4.1 presents a generalized methodology for constructing $\mathcal{AMR}$s, Section 4.2 introduces a concrete instantiation of this methodology, and Section 4.3 includes a set-difference sub-protocol that is used as part of our construction.

4.1 A General Framework

We present our proposed framework, outline its benefits, and highlight the challenges behind a secure and efficient instantiation.

Framework. Recall that the query result is a set of tuples, each consisting of m attribute values, and satisfying certain range conditions. For the sake of simplicity, we henceforth define result R of query Q on dataset T as a set containing exactly the *hash value* $h_i = H(t_i)$ of the binary representation of each tuple $t_i \in T$ that satisfies the query conditions, under a CRHF $H : \{0, 1\}^* \rightarrow \mathbb{Z}_p$. Our $\mathcal{AMR}$ constructions will focus on proving that R is the correct set of hash values corresponding to the tuples satisfying the query. Then, given these hash values along with the full result tuples, the client can validate the integrity of each result tuple t_i by testing $H(t_i) \overset{?}{=} h_i$. Due to collision-resistance of H, the server cannot return a falsified t_i^* such that $H(t_i^*) = h_i$, instead of the correct pre-image t_i of h_i. In the following, when clear from the context, we use term "tuple" for a table tuple $t_i \in T$ and its hash value $h_i = H(t_i)$ interchangeably.

We illustrate the main idea of our framework using Figure 1. Let $h_1, \ldots, h_n$ correspond to the hash values of the tuples $t_1, \ldots, t_n$ of T, respectively. We maintain a copy of the values for every attribute a_i, and *sort* the copy of a_i according to the attribute values of the tuples on a_i. For instance, in Figure 1, $h_3 = H(t_3)$ appears first in the ordering of a_1 because t_3 has the smallest value on attribute a_1 among the tuples in T.

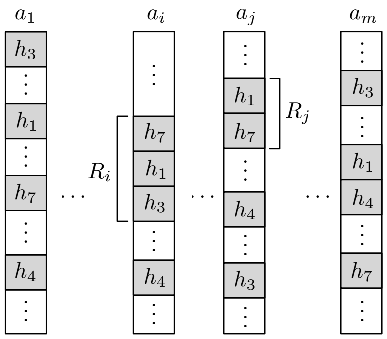


Figure 1: Illustrating the different tuple orders per attribute

A multi-dimensional range query Q is defined over an *arbitrary* set of dimensions (where, recall, each dimension is an attribute). Our framework "decomposes" a d-dimensional range query into d separate 1-dimensional queries. More specifically, our framework boils down to two steps:

- Step 1: **(1-D proofs)** For each dimension a_i involved in Q, compute the set R_i of all hash values of tuples that satisfy the condition on a_i. Formally, $R_i = \{h_j = H(t_j) \mid l_i \leq t_j.a_i \leq u_i\}$. Also compute proof π_{R_i} for the integrity of R_i.

- Step 2: **(Combination)** Compute the result $R \overset{\text{def}}{=} \bigcap_i R_i$ and proof π for its integrity, given pairs (R_i, π_{R_i}) for every a_i involved in Q.

For example, suppose in Figure 1 that a 2-dimensional query Q is defined over a_i and a_j. Our two-step framework first dictates the computation of $R_i = \{h_1, h_3, h_7\}$ that corresponds to the 1-dimensional result along dimension a_i, and $R_j = \{h_1, h_7\}$ along dimension a_j, as well as proofs π_{R_i}, π_{R_j}. It next requires the computation of result $R = \{h_1, h_7\}$ and a proof π.

Benefits. Our view of multi-dimensional range queries as a collection of 1-dimensional range queries offers multiple advantages over existing approaches: (i) We aim to support range queries over *any combination* of attributes. Thus, there are $O(2^m)$ possible different attribute combinations that can be involved in a query, where m is the total number of attributes in T. In order to support all of them, existing solutions must build $O(2^m)$ *separate* authenticated structures. On the contrary, our framework requires m such structures (one per attribute) constructed *once*, which suffice to capture *all* $O(2^m)$ possible subsets of attributes. (ii) As discussed in Section 1, the performance of all existing constructions deteriorates drastically with d. In contrast, the separate handling of each dimension allows our framework to scale with d gracefully. (iii) To address scalability issues that arise from the advent of "big data," data management systems typically employ multi-core CPU hardware, as well as cloud infrastructures involving multiple nodes. In our framework, the 1-dimensional sub-queries can be distributed across multiple cores/nodes, and run in parallel. The combination step can then take place using well-known in-network aggregation techniques (e.g., in a MapReduce fashion [12]).

The challenge. There are several efficient solutions for the problem of 1-dimensional range queries (e.g., [18, 17, 14, 15]), each of which can be used to instantiate Step 1 in our framework. The problem lies in Step 2, i.e., how to efficiently combine the separate proofs. In particular, for all known 1-dimensional solutions, Step 2 entails creating the proof π as the concatenation of all proofs π_{R_i} *and* the partial results R_i. This makes the proof size as large as the sum of the partial result cardinalities, which can be substantially larger than the final result R. In turn, this may lead to a prohibitive communication and verification cost for the client.

A fundamental requirement of our framework is the partial proofs π_{R_i} produced in Step 1 to be efficiently combinable to a short proof π in Step 2, whose size is independent of $\sum_i |R_i|$. More formally:

EFFICIENCY REQUIREMENT. *A $\mathcal{AMR}$ following our framework is* efficient*, if it outputs proofs π of size $o(\sum_i |R_i|)$.*

Based on our observation above, any existing 1-dimensional solution trivially conforms to our framework. However, *no* such solution satisfies the efficiency requirement. Essentially, the efficiency requirement motivates the design of $\mathcal{AMR}$s with non-trivial proof combination techniques. What has prevented the research community from devising such $\mathcal{AMR}$s is the combination of the lack of appropriate cryptographic tools, and the reduced need for range queries over arbitrarily many dimensions, and large quantities of data. However, the emergence of big data practices renders the problem timely and important, whereas the recent introduction of $\mathcal{SOA}$ techniques opens new directions towards efficient solutions.

4.2 Construction

We first outline the main idea of our scheme, and elaborate on some important implementation decisions. Subsequently, we present the instantiation of our algorithms.

Main idea. Recall that Step 2 of our framework dictates that the result R is expressed as the *intersection* of *sets* R_i. We stress that $\mathcal{SOA}$ techniques appear to solve our targeted problem trivially as follows: The owner pre-computes a proof component $\pi_{R_i} = acc(R_i)$ for each R_i, where $acc(R_i)$ is the accumulation value of set R_i, and signs each π_{R_i}. According to our discussion in Section 2, given all R_i, π_{R_i}, the server computes and sends to the client a combined intersection proof $\pi_\cap$ for the integrity of R, along with all π_{R_i} and their corresponding signatures. Observe that this approach satisfies our efficiency requirement. However, there exist $O(n^2)$ possible R_i sets per dimension that can be involved in a query, which makes the pre-processing cost for the owner and the storage overhead for the server prohibitive. The main idea behind our scheme is to express *any possible* R_i as the result of an operation over a *fixed* number of "primitive sets," given the constraint that there are $O(n)$ such "primitive sets."

One possible way to derive R_i from "primitive sets" is illustrated in Figure 2(a). Let us focus on a_i and the ordering of the hash values h_j (of tuples t_j) according to the $t_j.a_i$ values. We define the *prefix set* $P_{i,j}$ to consist of all hash values appearing in positions $1, \ldots, j$ in the ordering. In the figure, $P_{i,1} = \{h_3\}$ and $P_{i,2} = \{h_3, h_1\}$. Similarly, we define *suffix set* $S_{i,j}$ to consist of all hash values appearing in positions $n - j + 1, \ldots, n$ in the ordering. In our example, $S_{i,1} = \{h_6\}$ and $S_{i,2} = \{h_5, h_6\}$. Now assume that $k'_i + 1, k_i$ are the two positions in this ordering corresponding to the first and last tuple satisfying the query on a_i. Observe that, in this case, $R_i = P_{i,k_i} \cap S_{i,k'_i+1}$, and, thus, there exist $2n$ "primitive sets" per dimension, i.e., all prefix and suffix sets. Let $\pi_{P_{i,k_i}} = acc(P_{i,k_i})$ and $\pi_{S_{i,k'_i+1}} = acc(S_{i,k'_i+1})$. Then, since $R = \bigcap_i R_i = \bigcap_i (P_{i,k_i} \cap S_{i,k'_i+1})$ can be computed with a single set intersection, we can utilize the $\mathcal{SOA}$ of [24] to create a proof π (consisting of $\pi_\cap$ and signatures on every $\pi_{P_{i,k_i}}, \pi_{S_{i,k'_i+1}}$) for the integrity of R, while satisfying both efficiency and $O(n)$ preprocessing/storage.

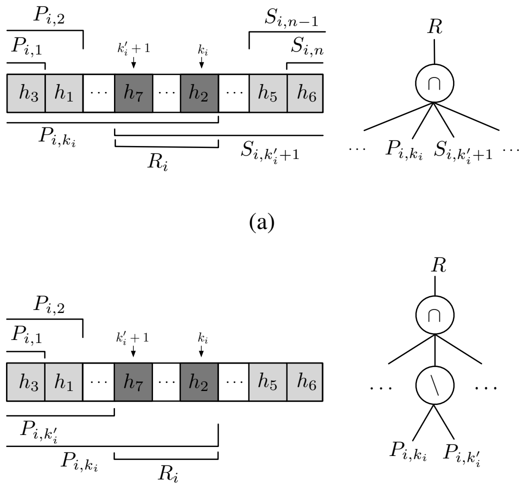


Figure 2: Set representation of R_i

Unfortunately, from the complexity analysis of [24] in Section 2, it follows that $\pi_\cap$ requires $\tilde{O}(d \cdot n)$ time for each query at the server, which makes this approach impractical. The reason is that the $\pi_\cap$ construction overhead is dictated by the cardinality of the input sets, which is $|P_{i,k_i}| + |S_{i,k'_i+1}| \in \Omega(n)$, along each attribute.

Motivated by the above, we propose an alternative solution which we demonstrate using Figure 2(b). We define sets R_i through *set-difference*. In particular, using the notation of the previous paragraph, it holds that $R_i = P_{i,k_i} \setminus P_{i,k'_i}$. Consequently, in this case the "primitive sets" are only the n prefix sets. Now $R = \bigcap_i R_i = \bigcap_i (P_{i,k_i} \setminus P_{i,k'_i})$ is no longer expressed as a single set operation. The only known $\mathcal{SOA}$ that can accommodate a circuit of set operations is [8]. Briefly stated, [8] allows the construction of π_{R_i} with $\tilde{O}(|R_i|)$, and $\pi_\cap$ with $\tilde{O}(\sum_i |R_i|)$, exponentiations. The downside is that its security relies on non-standard cryptographic assumptions. In Section 4.3, we construct our own sub-protocol for producing combinable proofs of set-difference, customized for the special case where the first participating set is a *strict superset* of the second. This particular constraint enables our sub-protocol to prove the validity of $R_i = P_{i,k_i} \setminus P_{i,k'_i}$ with $\tilde{O}(|R_i|)$ exponentiations, while being secure under a standard cryptographic assumption.

Our algorithms are comprised of a collection of set operation sub-protocols, bundled with a set membership scheme. For clarity of presentation, we will abstract the internal mechanics of these sub-protocols, and instead use the following conventions:

- By $\mathcal{SMA}$ we refer to either a Merkle tree [19] or an accumulation tree [22], along with all its algorithms.

- By ProveIntersection, VerifyIntersection, we refer to the corresponding algorithms of the $\mathcal{SOA}$ of [24]. The former computes an intersection proof on its input sets, and the latter verifies this operation.

- By ProveSetDiff, VerifySetDiff, we refer to the corresponding algorithms of our construction presented in Section 4.3. The former generates a set-difference proof, and the latter verifies this operation.

This presentation choice also highlights that our algorithms use elementary cryptographic tools as building blocks. Therefore, the overall performance of our scheme is highly dependent on that of the underlying tools, leaving potential for great improvement as novel tool instantiations are introduced in the literature.

Key generation. It outputs key pair pk, sk, which are simply the public and secret keys of the underlying $\mathcal{SMA}$ and $\mathcal{SOA}$ schemes, generated by their corresponding key generation routines.

Setup. Figure 3 visualizes the detailed authentication structure produced by the setup algorithm, whose pseudo code is shown in the next page. The owner computes the hash value $h = H(t)$ for every $t \in T$ (Line 1). It then produces the sorted orderings of the hash values along every attribute (Lines 2-3), and computes the prefix sets $P_{i,j}$ as explained in Figure 2(b). Next, it calculates prefix proof $\pi_{P_{i,j}}$ for each $P_{i,j}$ (Lines 4-5). For each $P_{i,j}$, it computes a triplet $\tau_{P_{i,j}} = (v_{i,j}, v_{v,j+1}, \pi_{P_{i,j}})$ in Lines 6-7. Values $v_{i,j}, v_{i,j+1}$ indicate the j^{th} and $(j+1)^{\text{th}}$ largest values on attribute a_i appearing in T. These values are necessary for guaranteeing the completeness of the result, and their purpose will become clear soon. We make the assumption that all triplets τ are distinct across attributes, i.e., there does not exist τ that appears in more than one attributes. This can be easily achieved in practice by including the attribute index i in each τ, however for simplicity of presentation we avoid it. Subsequently, it computes an $\mathcal{SMA}_i$ over the triples $\tau_{P_{i,j}}$ of every attribute a_i, producing digests $\delta_1, \ldots, \delta_m$ (Line 8). It then constructs a $\mathcal{SMA}$ over the (i, δ_i) pairs, and generates digest δ (Line 9). Finally, it sends the $m+1$ $\mathcal{SMA}$ structures to the server along with T (Line 10), and publishes pk, δ (Line 11).

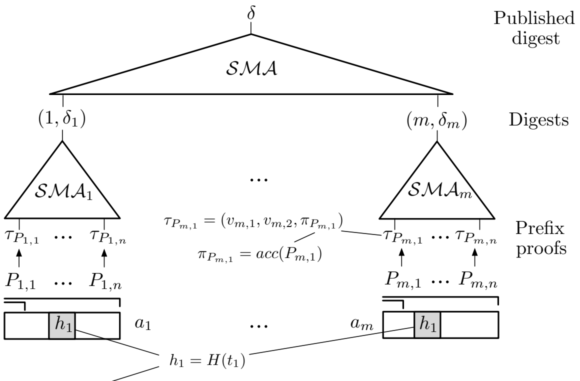

δ
Published digest
$\mathcal{SMA}$
$(1, \delta_1)$
(m, δ_m)
Digests
$\mathcal{SMA}_1$
$\cdots$
$\mathcal{SMA}_m$
$\tau_{P_{m,1}} = (v_{m,1}, v_{m,2}, \pi_{P_{m,1}})$
$\pi_{P_{m,1}} = acc(P_{m,1})$
$\tau_{P_{1,1}}$ $\cdots$ $\tau_{P_{1,n}}$
$\tau_{P_{m,1}}$ $\cdots$ $\tau_{P_{m,n}}$
Prefix proofs
$P_{1,1}$ $\cdots$ $P_{1,n}$
$P_{m,1}$ $\cdots$ $P_{m,n}$
h_1
a_1
$\cdots$
a_m
h_1
$h_1 = H(t_1)$
t_1
$\cdots$
t_n
T

Figure 3: Our authentication structure

Proof construction. For ease of presentation and without loss of generality, we assume that the requested query is upon the d first attributes of T and, hence, encode it as $Q = \{i, l_i, u_i\}$ for $i = 1, ..., d$. We provide the pseudo code of this algorithm below.

Given R, the server first computes π_R (Line 1), and calculates set R_i for each attribute a_i. It identifies prefixes P_{i,k_i}, P_{i,k'_i} such that

```
**Algorithm** Setup$(T, pk, sk)$

1.  **For** $j = 1, ..., n$, compute $h_j = H(t_j)$
2.  **For** $i = 1, ..., m$
3.      Sort $h_j$ in ascending order along $t_j.a_i$
4.      **For** $j = 1, ..., n$
5.          Compute $\pi_{P_{i,j}} = acc(P_{i,j})$
6.          Let $v_{i,j}$ be the $j^{\text{th}}$ largest value on attribute $a_i$ in $T$
7.          Construct triplet $\tau_{P_{i,j}} = (v_{i,j}, v_{i,j+1}, \pi_{P_{i,j}})$
8.      Build $\mathcal{SMA}_i$ with digest $\delta_i$ over $\tau_{P_{i,1}}, ..., \tau_{P_{i,n}}$
9.  Build $\mathcal{SMA}$ with digest $\delta$ over $(1, \delta_1), ..., (m, \delta_m)$
10. Send $T, auth(T) = (\mathcal{SMA}, \mathcal{SMA}_1, ..., \mathcal{SMA}_m)$ to the server
11. Publish $pk, \delta$
```

```
**Algorithm** Prove$(Q, R, pk, auth(T))$

1.  Compute $\pi_R = acc(R)$
2.  **For** $i = 1, ..., d$
3.      Compute $R_i$
4.      Identify $P_{i,k}$ and locate $\tau_{P_{i,k}} = (v_{i,k_i}, v_{i,k_i+1}, \pi_{P_{i,k_i}})$
5.      Identify $P_{i,k'}$ and locate $\tau_{P_{i,k'}} = (v_{i,k'_i}, v_{i,k'_i+1}, \pi_{P_{i,k'_i}})$
6.      Compute $\mathcal{SMA}_i$ proofs for $\tau_{P_{i,k}}, \tau_{P_{i,k'}}$
7.      Compute $\mathcal{SMA}$ proof for $(i, \delta_i)$
8.      $\pi_{R_i}$ = ProveSetDiff$(R_i, pk)$
9.  $\pi_\cap =$ ProveIntersection$(R, R_1, ..., R_d, \pi_R, \pi_{R_1}, ..., \pi_{R_d}, pk)$
10. Set $\pi = (\pi_R, \pi_\cap, (\pi_{R_i}, \tau_{P_{i,k}}, \tau_{P_{i,k'}}, \delta_i)_{i=1}^d$, all $\mathcal{SMA}$ proofs)
11. Send $\pi, R$ to the client
```

$R_i = P_{i,k_i} \setminus P_{i,k'_i}$ (as in Figure 2(b)), and locates the corresponding triplets $\tau_{P_{i,k_i}}, \tau_{P_{i,k'_i}}$ (Lines 4-5). Subsequently, it constructs the $\mathcal{SMA}_i$ proofs for $\tau_{P_{i,k_i}}, \tau_{P_{i,k'_i}}$ (Line 6) and $\mathcal{SMA}$ proofs for (i, δ_i), for $i = 1, \ldots, d$ (Line 7). It then invokes subroutines ProveSetDiff and ProveIntersection as defined in our main idea paragraph, and produces proofs $\pi_{R_1}, \ldots, \pi_{R_d}$ and $\pi_\cap$ (Lines 8-9), respectively. Finally, it puts together all proof components into a single proof π (Line 10), and sends it to the client along with result R (Line 11). We thoroughly describe the functionality of every proof component in π in the next paragraph.

Verification. We visualize the intuition in Figure 4, where we depict the authentication flow among the various proof components of the final proof π sent to the client. Specifically, if a component authenticates another, we draw an arrow from the former to the latter. The corresponding arrow labels represent information serving as "glue" between the components. The goal is to verify result R (top of the figure), but the only trusted information (in addition to pk) is δ (bottom of the figure). Verification proceeds bottom-up from level 0 to 5, maintaining the invariant that, at level ℓ, the server must have computed the components therein truthfully with respect to T and Q.

At level 0, δ is signed/published by the owner and, thus, it is trusted. At level 1, given the d $\mathcal{SMA}$ proofs and δ, we verify the integrity of components (i, δ_i). Likewise, at level 2, given the $2d$ $\mathcal{SMA}_i$ proofs along with (i, δ), we verify the integrity of $2d$ triplets $(\tau_{P_{i,k_i}}, \tau_{P_{i,k'_i}})$. Here, we reach a critical point in the verification process. We must prove that these particular $(\tau_{P_{i,k_i}}, \tau_{P_{i,k'_i}})$ correspond to the triplets for sets P_{i,k_i}, P_{i,k'_i}, such that $P_{i,k_i} \setminus P_{i,k'_i} = R_i$, where R_i is the truthful result of Q on dimension a_i. To do this, we parse Q as $(i, l_i, u_i)_{i=1}^d$ and check $v_{i,k'_i} < l_i \leq v_{i,k'_i+1}$ and $v_{i,k_i} \leq u_i < v_{i,k_i+1}$, where $v_{i,k_i}, v_{i,k_i+1}, v_{i,k'_i}, v_{i,k'_i+1}$ are included in $\tau_{P_{i,k_i}}, \tau_{P_{i,k'_i}}$. This guarantees that P_{i,k_i} is the smallest prefix set that contains the entire R_i, and P_{i,k'_i} is the largest prefix set that does not intersect R_i. Therefore, we verify that $(\tau_{P_{i,k_i}}, \tau_{P_{i,k'_i}})$ indeed correspond to the correct P_{i,k_i}, P_{i,k'_i}. Next, we retrieve $\pi_{P_{i,k_i}}, \pi_{P_{i,k'_i}}$ from $(\tau_{P_{i,k_i}}, \tau_{P_{i,k'_i}})$, respectively, and run routine VerifySetDiff to validate the truthfulness of π_{R_i} as the accumulation value of set R_i at level 3. Next, at level 4 we verify that π_R is the accumulation value of $\bigcap_i R_i$ with VerifyIntersection, using π_R, all π_{R_i}, and $\pi_\cap$. Observe that, at this point we know that π_R corresponds to the accumulation of the correct result of Q on T. At the last level 5, we verify that R is indeed this correct result by checking if $acc(R) = \pi_R$. We summarize this verification process in the pseudocode below.

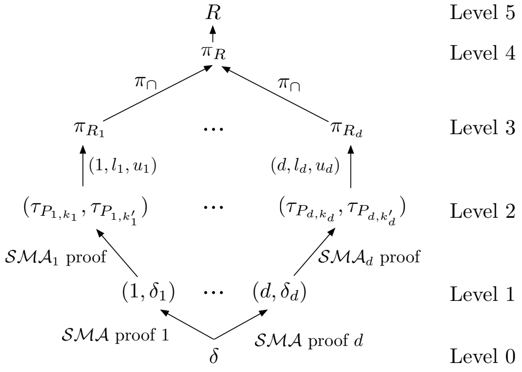


Figure 4: Authentication flow

```
**Algorithm** Verify$(Q, R, \pi, pk, \delta)$

1.  Parse $Q$ as $(i, l_i, u_i)_{i=1}^d$
2.  **For** $i = 1, ..., d$
3.      Verify $\delta_i$ with respect to $\delta$ with $\mathcal{SMA}$ proof
4.      Verify $\tau_{P_{i,k}}, \tau_{P_{i,k'}}$ with respect to $\delta_i$ with $\mathcal{SMA}_i$ proofs
5.      Verify $v_{i,k'_i} < l_i \leq v_{i,k'_i+1}$ and $v_{i,k_i} \leq u_i < v_{i,k_i+1}$
6.      Run VerifySetDiff$(\pi_{P_{i,k}}, \pi_{P_{i,k'}}, \pi_{R_i}, pk)$
7.  Run VerifyIntersection$(\pi_R, \pi_{R_1}, ..., \pi_{R_d}, \pi_\cap, pk)$
8.  Compute $acc(R)$ and verify $acc(R) = \pi_R$
9.  **If** verification in Lines 3-8 fails, **return** reject, **else return** accept
```

Updates. We focus on an insertion of a single tuple t (the case of deletions is similar). The process is easier to follow by revisiting Figure 3. The owner first computes $h = H(t)$. It then inserts h in the appropriate position in the ordering of each attribute a_i, and properly updates all the prefix sets it affects. Note that, if h is placed in position j for attribute a_i, the owner must change sets $P_{i,j'}$ for all $j' \geq j$, and modify their corresponding proofs $\pi_{P_{i,j'}}$ in $\tau_{P_{i,j'}}$. Furthermore, it must create a new $\tau_{P_{i,j}}$, and alter $v_{i,j}$ in $\tau_{P_{i,j-1}}$ (where it appears as the second element). Finally, it must propagate the changes of all τ triplets in all $\mathcal{SMA}_i$ and $\mathcal{SMA}$. Admittedly, the update process in this basic scheme can be quite expensive; in fact, it can be as costly as re-running the setup stage. In the Section 5, we introduce a solution that supports efficient updates, while maintaining all other asymptotic costs.

Complexity analysis. Table 1 in Section 1 summarizes our complexities. We will analyze our schemes considering $\mathcal{SMA}$s [19, 22]. The setup cost is $O(|T| \log n)$ regardless of the underlying $\mathcal{SMA}$, as it is dominated by m sorts of n hashes, and $O(mn)$ prefix proof computations. The proof size is $O(d \log n)$ if Merkle-trees are used as $\mathcal{SMA}$s, and $O(d)$ in the case of accumulation trees;

the proof is comprised of $O(d)$ prefix proofs, and d $\mathcal{SMA}$ proofs each with size $O(\log n)$ and $O(1)$ for the two alternatives, respectively. Proof construction entails $\tilde{O}(\sum_{i=1}^{d} |R_i|)$ cost for computing W_i, F_i encompassed in proof $\pi_\cap$, and $O(d \log n)$ / $O(dn^\epsilon \log n)$ $\mathcal{SMA}$ overhead for [19] / [22], respectively. Verification involves $\tilde{O}(|R|)$ time for verifying $\pi_\cap$, and $O(d \log n)$ / $O(d)$ for validating the $\mathcal{SMA}$ proofs. Finally, update is dominated by the recomputation of the $O(mn)$ prefix proofs and, thus, it can be done in time $O(|T|)$.

Correctness, efficiency, security. The correctness of our scheme results from the semantics of the proof generation and verification as thoroughly explained above. Moreover, since the proof size is either $O(d)$ or $O(d \log n)$, our construction satisfies the efficiency requirement. Finally, the next theorem states the security of the basic scheme. The proof is included in the Appendix.

THEOREM 1. *Our basic scheme is* secure *under q-SBDH and the security of the employed $\mathcal{SMA}$.*

4.3 A Set-Difference Sub-protocol

We present a sub-protocol for proving the correctness of a set-difference operation between two sets X_1, X_2, *under the constraint that the first is a proper superset of the second.* This constraint renders our sub-protocol conceptually simple and very efficient. It consists of two routines ProveSetDiff and VerifySetDiff. The former takes as input set $X_1 \setminus X_2$ and outputs a proof for its validity as the set-difference of X_1, X_2. The latter receives succinct representations $\pi_{X_1}, \pi_{X_2}, \pi_{X_1 \setminus X_2}$ of $X_1, X_2, X_1 \setminus X_2$, respectively, and returns `accept` if $X_1 \setminus X_2$ is the set-difference of X_1, X_2, and `reject` otherwise. Below is the pseudo codes of the two routines.

```
Algorithm ProveSetDiff($X_1 \setminus X_2, pk$)
1. Return $\pi_\setminus = acc(X_1 \setminus X_2)$

Algorithm VerifySetDiff($\pi_{X_1}, \pi_{X_2}, \pi_\setminus, pk$)
1. If $e(\pi_{X_2}, \pi_\setminus) = e(\pi_{X_1}, g)$, return accept, else return reject
```

Note that these routines are meaningful only as part of a more elaborate $\mathcal{SOA}$ scheme (e.g., [24, 8]), which utilizes bilinear accumulators as well, and relies on the same public key pk. More specifically, the caller $\mathcal{SOA}$ is enforced with the computation of input $X_1 \setminus X_2$ to ProveSetDiff. Therefore, this routine simply returns $\pi_\setminus$ as the accumulation value of $X_1 \setminus X_2$ in time $\tilde{O}(|X_1 \setminus X_2|)$. In addition, the $\mathcal{SOA}$ must first check that inputs π_{X_1}, π_{X_2} of VerifySetDiff are the accumulation values of X_1, X_2, such that X_1 is a proper superset of X_2, prior to calling the routine. In this case, the cost of VerifySetDiff is $O(1)$ pairings.

For example, in our scheme in Section 4.2, ProveSetDiff is called in algorithm Prove for each set R_i, *after* R_i has been computed. Moreover, VerifySetDiff is invoked in Verify using as inputs the *already verified* accumulation values of prefix sets P_{i,k_i}, P_{i,k'_i} that, by definition, satisfy the constraint $P_{i,k_i} \supset P_{i,k'_i}$. The following lemma is useful in our proofs included in the Appendix.

LEMMA 3. *Let λ be a security parameter, $pub \leftarrow$ BilGen(1^λ), and elements $(g, g^s, ..., g^{s^q}) \in \mathbb{G}$, computed for some s chosen at random from $\mathbb{Z}_p^*$. Let X_1, X_2 be sets with elements in $\mathbb{Z}_p$, such that $X_1 \supset X_2$. For an element $y \in \mathbb{G}$, it holds that $y = acc(X_1 \setminus X_2)$, iff $e(acc(X_2), y) = e(acc(X_1), g)$.*

5. UPDATE-EFFICIENT SCHEME

Section 5.1 presents an update-efficient construction that builds upon the basic scheme of the previous section. Section 5.2 includes a set union sub-protocol that is used as part of our construction.

5.1 Construction

Similar to our basic solution, the update-efficient scheme views the query result as a combination of "primitive set" operations. It then allows the server to compute a small set of proof elements, which can be aggregated by the client in a bottom-up fashion (similar to Figure 4). It adopts the same idea of computing proofs for the partial R_i results along each dimension a_i, and then combining them through a set intersection protocol into a single proof that verifies the final result R. It also adopts the idea of performing set-difference operations over prefix sets. The primary difference with the basic scheme is that we now organize the hash values in the ordering of each dimension into *buckets*, and compute prefix sets over both the buckets, as well as the hashes in each bucket. As we shall see, this twist *isolates* the effect of an update, thus, reducing the update cost complexity. However, it also mandates the modification of the overall authentication structure, proof generation and verification processes, and creates the need for a new *set union sub-protocol* (presented in Section 5.2). In the following, we only describe the main ideas behind the construction, omitting the tedious algorithmic details.

Figure 5 depicts the authentication structure created by the owner during the setup stage, focusing on attribute a_i. As before, the owner sorts the hash values of the n tuples of T in ascending order of the a_i values of the tuples. It then creates b buckets, enumerated as $B_{i,1}, \ldots, B_{i,b}$ (bottom left in the figure). For clarity of presentation, we assume that the partitioning of hashes into buckets is publicly known (e.g., each bucket may correspond to a specific range of the domain of a_i), and that each bucket has n/b hashes.

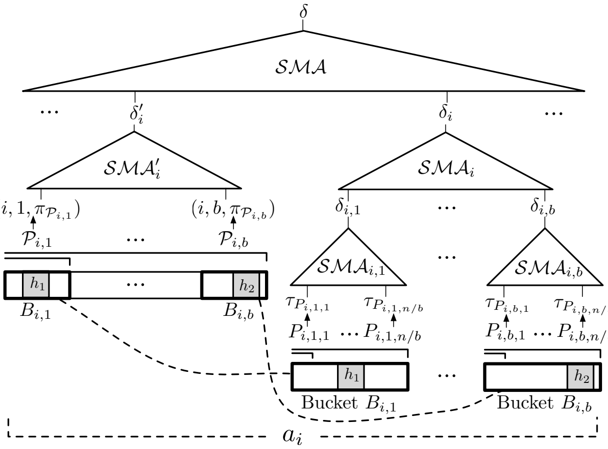


Figure 5: Authentication structure for efficient updates

We define as $\mathcal{P}_{i,j}$ the prefix set over buckets $B_{i,1}, \ldots, B_{i,j}$, i.e., the set of hashes included in $B_{i,1}, \ldots, B_{i,j}$ (we use calligraphic $\mathcal{P}$ for bucket prefixes to distinguish them from hash prefixes denoted by P). The owner computes a proof $\pi_{\mathcal{P}_{i,j}} = acc(\mathcal{P}_{i,j})$ for every $\mathcal{P}_{i,j}$. In addition, for every bucket $B_{i,j}$, it computes prefixes $P_{i,j,l}$ for the hashes therein (bottom right in the figure), as well as proofs $\pi_{P_{i,j,l}} = acc(P_{i,j,l})$. Subsequently, the owner creates a triplet $(i, j, \pi_{\mathcal{P}_{i,j}})$ for every $\mathcal{P}_{i,j}$, as well as tuple $\tau_{P_{i,j,l}}$ for every $P_{i,j,l}$. Note that $\tau_{P_{i,j,l}}$ is similar to the case of the basic scheme (i.e., it encompasses $\pi_{P_{i,j,l}}$ along with two a_i values), but now also incorporates the index j of the bucket. The owner feeds $(i, j, \pi_{\mathcal{P}_{i,j}})$ to the leaf level of $\mathcal{SMA}'_i$ with digest δ'_i. It also feeds $\tau_{P_{i,j,l}}$ to $\mathcal{SMA}_{i,j}$ with digest $\delta_{i,j}$. It then superimposes another $\mathcal{SMA}_i$ over digests $\delta_{i,j}$ which has digest δ_i. Finally, it builds $\mathcal{SMA}$ over all δ'_i, δ_i with final digest δ that is published. The various $\mathcal{SMA}$s will later allow the server to construct proofs validating that $\pi_{\mathcal{P}_{i,j}}$, $\pi_{P_{i,j,l}}$ were indeed computed by the owner specifically for bucket $B_{i,j}$; this is conceptually similar to their usage in the basic scheme.

We explain the proof construction and verification process using Figure 6, focusing on R_i. In our example, R_i *fully covers* buckets $B_{i,\kappa'+1}, \ldots, B_{i,\kappa}$, and *partially covers* buckets $B_{i,\kappa'}$ and $B_{i,k+1}$. Observe that we can decompose R_i into three sets, let ①, ②, ③ (so that we alleviate our notation and allow an easy reference to the figure), such that $R_i = \textcircled{1} \cup \textcircled{2} \cup \textcircled{3}$ and ①, ②, ③ are *pairwise disjoint* (the importance of the latter property will become clear in Section 5.2). Observe also that $\textcircled{1} = P_{i,\kappa',k} \setminus P_{i,\kappa',k'}$, $\textcircled{2} = \mathcal{P}_{i,\kappa} \setminus \mathcal{P}_{i,\kappa'}$, and $\textcircled{3} = P_{i,\kappa'+1,k}$. Therefore, the server builds the proof π by including proof components $\pi_{P_{i,\kappa',k}}$, $\pi_{P_{i,\kappa',k'}}$, $\pi_{\mathcal{P}_{i,\kappa}}$, $\pi_{\mathcal{P}_{i,\kappa'}}$, $\pi_{P_{i,\kappa'+1,k}}$. Moreover, it includes $(i, j, \pi_{\mathcal{P}_{i,j}}), \tau_{P_{i,j,l}}$, their proper proofs from $\mathcal{SMA}'_i, \mathcal{SMA}_{i,j}, \mathcal{SMA}_i, \mathcal{SMA}$, as well as $\pi_{\textcircled{1}} = acc(\textcircled{1})$, $\pi_{\textcircled{2}} = acc(\textcircled{2})$, $\pi_{\textcircled{3}=acc(\textcircled{3})}$. With all the above, the client can verify that $\pi_{\textcircled{1}}$, $\pi_{\textcircled{2}}$, $\pi_{\textcircled{3}}$ are the truthful proofs for sets ①, ②, ③.

The client next needs to combine $\pi_{\textcircled{1}}$, $\pi_{\textcircled{2}}$, $\pi_{\textcircled{3}}$ in order to verify that proof $\pi_{R_i} = acc(R_i)$, also included in the final π by the server, indeed corresponds to the R_i that is the *union* of ①, ②, ③. After that point, the client can proceed to prove the final result R in an identical way to the basic scheme. For this particular task, we utilize our own customized *set union sub-protocol*, which is included in Section 5.2. This sub-protocol is motivated by similar reasons that motivated our set-difference sub-protocol in Section 4.3; we need it to be executed in time $\tilde{O}(|R_i|)$, and be secure under standard cryptographic assumptions. What enables us to do this is the extra constraint that the participant sets must be a priori proven *pairwise disjoint*. At a high level, its ProveUnion routine outputs a proof $\pi_\cup$ on input sets ①, ②, ③, which later facilitates the VerifyUnion routine invoked on $\pi_{\textcircled{1}}$, $\pi_{\textcircled{2}}$, $\pi_{\textcircled{3}}$, π_{R_i}.

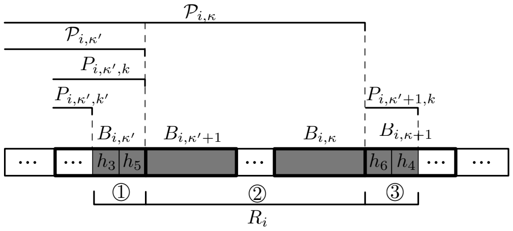


Figure 6: Representation of R_i through sets

Consider that tuple t is inserted in bucket $B_{i,j}$ (deletions are handled similarly). This insertion affects all b bucket prefixes in the worst case, and all n/b hash prefixes in $B_{i,j}$. It is important to observe that t does not affect any hash prefix of any other bucket; in that sense, the buckets isolate the effect of the update within their boundaries. Setting $b = \sqrt{n}$, the owner must update $O(\sqrt{n} \cdot m)$ prefixes in overall, each with a single exponentiation. Moreover, it should propagate the changes of the corresponding proofs inside the $\mathcal{SMA}$ structures, whose cost is asymptotically the same as in the case of the basic scheme. Therefore, the total update time in this construction reduces from $O(n \cdot m)$ to $O(\sqrt{n} \cdot m)$. Interestingly, the asymptotic complexities of all other algorithms and the proof size remain unaffected. However, the absolute costs slightly increase due to the extra bucket prefixes, as confirmed by our experiments in Section 6. Due to space limitations, we delegate the security and correctness proofs, as well as the detailed algorithm descriptions and complexity analysis to the long version of our work.

5.2 A Set Union Sub-protocol

We present a sub-protocol for proving the correctness of a union operation among a number of sets X_i *under the constraint that they are pairwise disjoint*. We focus on the case of three input sets, as this is the way it is utilized in Section 5. The sub-protocol consists of two routines, ProveUnion and VerifyUnion. The former receives sets X_1, X_2, X_3, and outputs a proof $\pi_\cup$ for the integrity of the union operation $X = X_1 \cup X_2 \cup X_3$. The latter receives succinct descriptions $\pi_{X_1}, \pi_{X_2}, \pi_{X_3}, \pi_X$ of X_1, X_2, X_3, X, respectively, as well as a proof $\pi_\cup$, and returns `accept` if X is the union of the three sets, and `reject` otherwise. We provide the pseudo codes of the two routines below.

```
Algorithm ProveUnion($X_1, X_2, X_3, pk$)
1. Output $\pi_\cup = acc(X_1 \cup X_2)$

Algorithm VerifyUnion($\pi_{X_1}, \pi_{X_2}, \pi_{X_3}, \pi_X, \pi_\cup, pk$)
1. Verify $e(\pi_{X_1}, \pi_{X_2}) = e(\pi_\cup, g)$
2. Verify $e(\pi_\cup, \pi_{X_3}) = e(\pi_X, g)$
3. If verification in Lines 1-2 fails, return reject, else return accept
```

Similar to the set-difference sub-protocol, these routines are meaningful only as part of a $\mathcal{SOA}$ scheme based on bilinear accumulators. ProveUnion runs in time $\tilde{O}(|X_1| + |X_2| + |X_3|)$. For VerifyUnion, it is the responsibility of the caller to check that π_{X_1}, π_{X_2}, π_{X_3} are the accumulation values of pairwise disjoint X_1, X_2, and X_3, prior to calling the routine. Its cost is $O(1)$ pairings.

6. EXPERIMENTS

We performed our experiments on a 64-bit machine with Intel Core i5 CPU 2.5GHz, running Linux. We measured the performance of all schemes implementing the necessary cryptographic primitives in C++, using the following libraries: DCLXVI [2] for fast bilinear pairing computations, Flint [3] for modular arithmetic, and Crypto++ [1] for SHA-256 hash operations. DCLXVI employs a 256-bit BN elliptic curve and an asymmetric optimal ate pairing, offering bit-level security of 128 bits. We represent elements of $\mathbb{G}_1$ with 768 bits using Jacobi coefficients, which yield faster operations. Elements in $\mathbb{G}_2$ are roughly twice as large as those of $\mathbb{G}_1$. We chose an asymmetric pairing for efficiency reasons, but we note that this choice does not introduce any redundancy to our schemes as presented with symmetric pairings. We instantiate all $\mathcal{SMA}$s with Merkle trees [19] and bilinear accumulator trees [22]. Table 2 summarizes all primitive costs involved in our schemes.

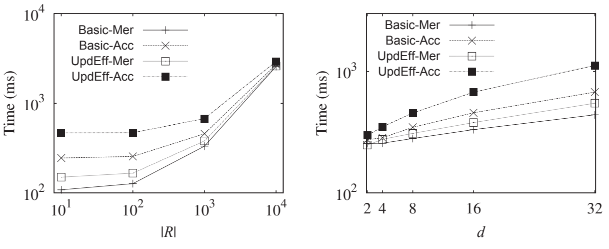
Operation		**Cost**
Exp. in $\mathbb{G}_1$ / $\mathbb{G}_2$		0.55 / 0.94 ms
Mult. in $\mathbb{Z}_p$ / $\mathbb{G}_T$		7 μs / 0.09 ms
SHA-256 / Bilinear pairing		5 μs / 1.41 ms
Quicksort in $\mathbb{Z}_p$	(100/1000/10000 elems.)	0.1 / 0.9 / 4.6 ms
Acc. in $\mathbb{G}_1$	–॥–	25.3 / 236 / $2,628$ ms
Acc. in $\mathbb{G}_2$	–॥–	32.6 / 338 / $3,471$ ms
Polynom. Mult in $\mathbb{Z}_p[r]$	(100/1000/10000 coeffs.)	0.4 / 7.3 / 92.9 ms
XGCD in $\mathbb{Z}_p[r]$	–॥–	8.4 / 599 / $108,093$ ms

Table 2: Costs of primitive operations

We test four possible configurations: (i) our basic scheme with Merkle trees (Basic-Mer), (ii) our basic scheme with accumulator trees (Basic-Acc), (iii) our update-efficient scheme with Merkle trees (UpdEff-Mer), and (iv) our update-efficient scheme with accumulator trees (UpdEff-Acc). For each configuration, we assess the performance at the client, owner and server, varying several parameters. We run each experiment 10 times and report the average costs. Note that the performance of all schemes does not depend on the data distribution, but rather on the table schema and the result selectivities, hence we chose to use synthetic data in our evaluation. We stress that our goal here is not to construct an optimized prototype, but rather to demonstrate the feasibility of our schemes. As such, we have left numerous optimizations regarding database storage and query handling as future work.

Client. Figure 7 depicts the verification cost at the client. This overhead is mainly affected by the result size $|R|$ and the number of query dimensions d. Figure 7(a) shows the CPU time (in ms) as a function of $|R|$, fixing $d = 32$, $n = 10^6$ and $m = 64$. The verification cost increases with $|R|$ in all schemes. Basic-Mer is the fastest for $|R| \leq 1,000$. This is because the Merkle-based schemes are faster than the accumulator-based ones, as they entail hash operations for the $\mathcal{SMA}$ proofs, which are much cheaper than the pairings needed in accumulation trees. Moreover, the overhead in the update-efficient schemes is slightly larger than that in their basic counterparts, due to the extra proof verifications of the bucket prefixes and the taller $\mathcal{SMA}$ hierarchy. Nevertheless, observe that, for $|R| = 10,000$ the performance of all schemes converges. The reason is that the computation of $\pi_R = acc(R)$ that is common to all techniques becomes the dominant factor, which effectively hides the costs of the $\mathcal{SMA}$ proofs and all set-operation verifications.

Figure 7(b) illustrates the CPU time versus d, when $|R| = 1,000$, $n = 10^6$ and $m = 64$. The performance of the schemes is qualitatively similar to Figure 7(a) for the same reasons. Once again, all costs increase linearly with d because the verification overhead of the set-differences and intersections is also linear in d. However, the effect of d on the total CPU time is not as significant as that of $|R|$, since the common accumulation cost for R emerges as the dominant cost when $|R| = 1,000$. In both figures, the verification time for all constructions is between 20 ms and 3.36 seconds.



(a) Time vs. $|R|$ (b) Time vs. d

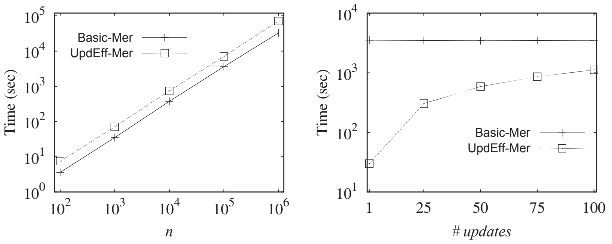
Figure 7: Verification overhead at client

Table 3 includes the proof sizes for the four schemes when varying d. We make three observations. First, all sizes increase with d, since the proof includes components for every dimension. Second, the basic schemes have smaller proofs than their counterparts, again because of the extra bucket prefix proofs and taller $\mathcal{SMA}$ hierarchy. Third, although Basic-Acc outperforms Basic-Mer , this is not true for UpdEff-Acc and UpdEff-Mer. This is because, although accumulators provide asymptotically smaller proofs than Merkle trees, this does not hold in practice for the database sizes we tested. Overall, the proofs for all schemes are quite succinct, ranging from 4.5 to 153.5 KBs, which are independent of the result size that could easily be in the order of MBs.

d	**2**	**4**	**8**	**16**	**32**
Basic-Mer	4.5	9.1	18.1	36.3	72.5
Basic-Acc	3.6	7.2	14.3	28.8	57.5
UpdEff-Mer	9.2	18.4	36.9	73.8	147.5
UpdEff-Acc	9.6	19.2	38.4	76.8	153.5

Table 3: Proof size in KB $(n = 10^6, m = 64)$

Owner. Figure 8 assesses the performance of the owner for the setup stage (which includes the key generation), and updates. In this set of experiments, we focus only on the Merkle-based schemes that have a clear performance advantage over the accumulator-based, as evident also from our evaluation for the client above. Figure 8(a) plots the pre-processing cost when varying n and fixing $m = 64$. Naturally, the overhead increases linearly with n in both schemes. This overhead is dominated by the computation of $\pi_{P_{i,j}}$ for all i, j, which completely hides the sorting and hashing costs (see also Table 2). As expected, UpdEff-Mer is more than twice as slow as Basic-Mer. Although the pre-processing time can reach up to three hours for $n = 10^6$, recall that this is a one-time cost for the owner.



(a) Setup time vs. n (b) Update time vs. # updates

Figure 8: Setup and update overhead at owner

Figure 8(b) evaluates the update time as a function of the number of updates performed in a single batch operation, where $n = 10^5$ and $m = 64$. Note that we report the respective *worst case* in both schemes. For Basic-Mer, the CPU time is practically unaltered and, in fact, is as bad as the setup overhead. On the contrary, UpdEff-Mer is greatly benefited by the bucket isolation and becomes up to more than two orders of magnitude more efficient than Basic-Mer. As the number of updates in the batch increase, the performance gap between the two schemes closes, since the updates in the batch are likely to affect more buckets. For the tested settings, the update time ranges between 30 seconds and one hour.

Server. Figure 9 reports the proof generation time at the server. As explained in our complexity analysis, the dominant factor here is $\sum_{i=1}^{d} |R_i|$. Therefore, due to the lack of real-world data and query workloads, it suffices to vary $|R_i|$ and fix it across all dimensions, rather than varying d and setting an arbitrary partial result size per dimension. Figure 9 depicts the CPU time at the server, when varying $|R_i|$ and setting $n = 10^5$, $m = 64$, $d = 32$ and $|R| = 0.1 \cdot |R_i|$ (i.e., 10% of a 1-dimensional result). At every point of the curve, we also provide the percentages of the three dominant computational costs, namely the construction of W_i, F_i (for the intersection proof) and π_{R_i}. The performances of two schemes differ marginally. This is because the generation of the extra set union proof of UpdEff-Mer incurs negligible cost compared to the large burden of computing the three types of elements mentioned above. The most interesting observation is that, for $|R_i| = 10$, the cost for W_i is 51% and for F_i is 15%, whereas for $|R_i| = 10,000$, the two costs become 7% and 87%, respectively. This is because computing F_i requires running the Extended Euclidean (XGCD) algorithm, whose time increases drastically with the degree of the polynomials (as shown in Table 2), dictated by $|R_i|$. The server's total overhead ranges between 650 ms and 25 minutes.

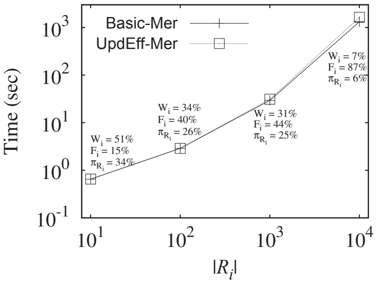


Figure 9: Proof construction cost at server

Summary and future improvements. Our experimental evaluation confirms the feasibility of our schemes. Specifically, it demonstrates that the verification cost at the client in all schemes is in the order of a few seconds in the worst case, even for moderate result sizes, whereas the proof size is up to a few hundred of KBs. At the owner, we illustrated the benefits of our update-efficient scheme over the basic one in terms of updates, which come at the cost of a more expensive setup and client verification. Finally, the server is the most impacted party in our constructions. The proof generation cost takes from several ms to several minutes, for small and moderate partial result sizes and dimensionality.

Nevertheless, it is important to stress that the defining costs at the server account for exponentiations and modular polynomial arithmetic. These operations are at the core of numerous applications and, thus, there is huge potential for improvement in the near future. In addition, there are works (e.g., [13]) that have substantially boosted such operations with modern hardware, which we did not possess in our experimentation. Being instantiations of a general framework, our schemes feature the attractive property that are easily upgradeable with future advances in such cryptographic tools.

7. CONCLUSION

We proposed schemes for authenticated multi-dimensional range queries over outsourced databases. Contrary to existing literature, our solutions scale *linearly* with the number of dimensions, and can support queries on *any* set of dimensions with *linear* in the number of database attributes setup cost and storage. The central idea of our methods is the reduction of the multi-dimensional range query to set-operations over appropriately defined sets in the database. We provided a detailed asymptotic and empirical performance evaluation, which confirms the feasibility of our schemes in practice.

Acknowledgments

We thank all the anonymous reviewers for their detailed comments and suggestions. Research supported in part by NSF grants CNS-1012798 and CNS-1012910.

8. REFERENCES

[1] The Crypto++ Library. `http://www.cryptopp.com/`.

[2] The DCLXVI Library. `http://cryptojedi.org/crypto/`.

[3] The Flint Library. `http://www.flintlib.org/`.

[4] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, 1990.

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

[6] D. Boneh, A. Sahai, and B. Waters. Functional encryption: Definitions and challenges. In *TCC*, 2011.

[7] P. G. Brown. Overview of SciDB: Large scale array storage, processing and analysis. In *SIGMOD*, 2010.

[8] R. Canetti, O. Paneth, D. Papadopoulos, and N. Triandopoulos. Verifiable set operations over outsourced databases. In *PKC*, 2014.

[9] H. Chen, X. Ma, W. Hsu, N. Li, and Q. Wang. Access control friendly query verification for outsourced data publishing. In *ESORICS*, 2008.

[10] W. Cheng and K.-L. Tan. Query assurance verification for outsourced multi-dimensional databases. *J. Computer Security*, 17(1):101–126, 2009.

[11] K.-M. Chung, Y. T. Kalai, F.-H. Liu, and R. Raz. Memory Delegation. In *CRYPTO*, 2011.

[12] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[13] P. Emeliyanenko. High-performance polynomial GCD computations on graphics processors. In *HPCS*, 2011.

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

[15] 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.

[16] H. Hacigümüs, S. Mehrotra, and B. R. Iyer. Providing database as a service. In *ICDE*, 2002.

[17] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Dynamic authenticated index structures for outsourced databases. In *SIGMOD*, 2006.

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

[19] R. C. Merkle. A certified digital signature. In *CRYPTO*, 1989.

[20] E. Mykletun, M. Narasimha, and G. Tsudik. Authentication and integrity in outsourced databases. *TOS*, 2(2):107–138, 2006.

[21] L. Nguyen. Accumulators from bilinear pairings and applications. In *CT-RSA*, 2005.

[22] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Authenticated hash tables. In *CCS*, 2008.

[23] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Cryptographic accumulators for authenticated hash tables. *IACR Cryptology ePrint Archive*, 2009:625, 2009.

[24] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Optimal verification of operations on dynamic sets. In *CRYPTO*, 2011.

[25] B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly Practical Verifiable Computation. In *IEEE Symposium on Security and Privacy*, 2013.

[26] R. Tamassia. Authenticated data structures. In *ESA*, 2003.

[27] J. Xu. Authenticating aggregate range queries over dynamic multidimensional dataset. *IACR Cryptology ePrint Archive*, 2010:244, 2010.

[28] Y. Yang, S. Papadopoulos, D. Papadias, and G. Kollios. Authenticated indexing for outsourced spatial databases. *VLDB J.*, 18(3):631–648, 2009.

APPENDIX

Proof of Lemma 3

($\Rightarrow$) Let $X_1 = \{x_1, ..., x_{l'}\}$ and $X_2 = \{x_1, ..., x_l\}$ for $l, l' \in \mathbb{N}$ with $l < l'$. If $e(acc(X_2), y) = e(acc(X_1), g)$, then we have $e(g^{\prod_{i=1}^{l}(x_i+s)}, y) = e(g^{\prod_{i=1}^{l'}(x_i+s)}, g)$. Hence, it holds $e(y, g) = e(g^{\prod_{i=l+1}^{l'}(x_i+s)}, g) \overset{\text{def}}{=} e(acc(X_1 \setminus X_2), g)$ and it follows that $y = acc(X_1 \setminus X_2)$.

($\Leftarrow$) If $y = acc(X_1 \setminus X_2) = g^{\prod_{i=l+1}^{l'}(x_i+s)}$, then it holds that $e(g^{\prod_{i=1}^{l}(x_i+s)}, y) = e(g^{\prod_{i=1}^{l}(x_i+s)}, g^{\prod_{i=l+1}^{l'}(x_i+s)}) = e(g^{\prod_{i=1}^{l'}(x_i+s)}, g) = e(acc(X_1), g)$.

■

Proof of Theorem 1

Let us assume there exists PPT adversary $\mathcal{A}$ that wins the $\mathcal{AMR}$ security game with non-negligible probability. Also, let Q, R^*, π^*, j be the cheating tuple output by $\mathcal{A}$ and let T denote the data structure's state at index j, and $auth(T), \delta$ the corresponding authentication information and digest . In the following, we annotate by $*$ any element of π^*. Moreover, if an event is denoted by $\mathcal{E}$, then its complement is denoted by $\mathcal{E}'$. Consider the following events:

$\mathcal{E}_1$**:** $\mathcal{A}$ wins the $\mathcal{AMR}$ game.

$\mathcal{E}_2$**:** π^* contains a tuple τ^* or $(i, \delta_i)^* \notin auth(T)$.

Recall that $auth(T)$ consists of $m + 1$ $\mathcal{SMA}$ structures; each of the m first is built over n tuples τ containing sequential values and prefix accumulations for attribute a_i, and the last is built over m pairs of the form (i, δ_i), i.e., containing the attribute index and corresponding digest. Note that, in the $\mathcal{AMR}$ game the values $auth(T), d$ are computed correctly by the challenger for T.

By the law of total probability, we have:

$$\begin{aligned}\Pr[\mathcal{E}_1] &= \Pr[\mathcal{E}_2]\Pr[\mathcal{E}_1|\mathcal{E}_2] + \Pr[\mathcal{E}_2']\Pr[\mathcal{E}_1|\mathcal{E}_2'] \\ &\leq \Pr[\mathcal{E}_1|\mathcal{E}_2]] + \Pr[\mathcal{E}_1|\mathcal{E}_2'] \ .\end{aligned}$$

Intuitively, the first term in the right hand of the above relation corresponds to an adversary that wins by breaking the security of the underlying $\mathcal{SMA}$ and the second term with breaking the q-SBDH.

CLAIM 1. $\Pr[\mathcal{E}_1|\mathcal{E}_2]$ *is negligible in* λ.

Proof. Let us assume it is non-negligible in λ. Without loss of generality, we will assume that the non-existing tuple is of the form τ^*, i.e., it should fall under some of the first m $\mathcal{SMA}$ structures, e.g., $\mathcal{SMA}_i$. Since $\mathcal{A}$ wins, it follows that the $\mathcal{AMR}$ verification succeeds however $\tau^* \notin \mathcal{SMA}_i$. We now distinguish between the chosen $\mathcal{SMA}$ instantiation:

- **Merkle tree.** We will construct adversary $\mathcal{A}'$ that finds a collision in the CRHF H used to implement the Merkle tree as follows. $\mathcal{A}'$ runs $\mathsf{BilGen}(1^\lambda)$ to compute bilinear parameters pub, chooses $s \leftarrow_R \mathbb{Z}_p^*$ and $q \in poly(\lambda)$ and computes values $g^s, \ldots, g^{s^q}$. Finally, he runs $\mathcal{A}$ on input $(pub, g^s, \ldots, g^{s^q})$. He then proceeds to provide oracle access for all the $\mathcal{AMR}$ algorithms. For the necessary computations of H (as part of the Merkle tree construction and verification) he queries his CRHF challenger. After the setup and each update call from $\mathcal{A}$, database T_η for $\eta = 0, \ldots, j$ is produced and $\mathcal{A}'$ stores all triplets $(T_\eta, auth(T_\eta), \delta_\eta)$. When $\mathcal{A}$ outputs his challenge tuple for index j, $\mathcal{A}'$ parses π^*, checking for each tuple whether it appears in $auth(T_j)$. If any of them does not appear in $auth(T_j)$, there must exist triplet τ in the corresponding $\mathcal{SMA} \in auth(T_j)$ such that $\tau \neq \tau^*$ and $H(\tau) = H(\tau^*)$ (for the challenge sample key of H). This holds since the verification process for τ^* under a Merkle tree in $auth(T_j)$ succeeds, yet τ^* is not in the tree. By assumption this will happen with non-negligible probability, hence $\mathcal{A}'$ breaks the collision resistance of H, and the claim follows.
- **Accumulation tree.** The reduction proceeds in the same manner is in the previous case. The difference is that $\mathcal{A}'$ is now playing against an accumulation tree challenger, he receives as input a public key that coincides perfectly with the $\mathcal{AMR}$ game and he does not need to issue any queries to his challenger before he sees the challenge tuple by $\mathcal{A}$, since everything can be computed with access to the public key only (this follows from the properties of the bilinear accumulator used to build the tree). After $\mathcal{A}$ issues his challenge, $\mathcal{A}'$ constructs the tree $\mathcal{SMA}_i$ by issuing consecutive update queries to his challenger. Finally, he outputs τ^* and the part of π^* that corresponds to proving (the false statement) that $\tau^* \in \mathcal{SMA}_i$. By Lemma 1 this can only happen with negligible probability, which contradicts our original assumption, and the claim follows.

□

Now we prove that the second term of the inequality, namely $\Pr[\mathcal{E}_1|\mathcal{E}_2']$, is negligible, by contradiction. Assume that $\Pr[\mathcal{E}_1|\mathcal{E}_2']$ is non-negligible. Since $\mathcal{E}_2$ does not happen, all triplets τ^* and pairs $(1, \delta_1), \ldots, (m, \delta_m)$ in π^* appear in $auth(T)$.

This immediately implies that the two values $v_{i,l}^*, v_{i,l+1}^*$ in each triplet are consecutive along their dimension and each digest matches its corresponding dimension. By construction, along each dimension there exist *exactly two distinct* τ^*, τ'^* for which verification of Q succeeds; one corresponds to the lower bound of the 1-dimensional range of the query (l_i) and one for the upper (u_i). Furthermore, if a triplet correctly formed for $\mathcal{SMA}_i^*$ of attribute a_i, is used as part of the proof of an $\mathcal{SMA}_j^*$ corresponding to $a_j \neq a_i$, then it can be used to break the $\mathcal{SMA}$ security as shown in the proof of Claim 1, which can only happen with negligible probability.

From the above, it follows that, for all i, i', the triplet $\tau_{i,i'}^* \in \pi^*$ in dimension a_i contains the accumulation value of the correctly computed prefix set $P_{i,i'}$ with all but negligible probability. Assuming this holds, by Lemma 3 and because verification succeeds, it follows that $\pi_{R_i}^* \in \pi^*$ is the accumulation value of the correctly computed set R_i for query Q on T. Therefore, the values $W_i^*, F_i^* \in \pi_\cap^*$, along with sets R_i and cheating answer $R^* \neq R$ (where R is the correct result of Q) as output by $\mathcal{A}$, contradict Lemma 2, breaking the q-SBDH assumption. Therefore, the probability $\Pr[\mathcal{E}_1|\mathcal{E}_2']$ must be negligible.

Since $\Pr[\mathcal{E}_1|\mathcal{E}_2] + \Pr[\mathcal{E}_1|\mathcal{E}_2']$ is negligible, $\Pr[E_1]$ must be negligible as well, contradicting our original assumption that there exists PPT adversary $\mathcal{A}$ that breaks our scheme with non-negligible probability.

■