Authenticated Online Data Integration Services

Qian Chen, Haibo Hu, and Jianliang Xu
Dept. of Computer Science, Hong Kong Baptist University
Kowloon Tong, Hong Kong
{qchen, haibo, xujl}@comp.hkbu.edu.hk

ABSTRACT

Data integration involves combining data from multiple sources and providing users with a unified query interface. Data integrity has been a key problem in online data integration. Although a variety of techniques have been proposed to address the data consistency and reliability issues, there is little work on assuring the integrity of integrated data and the correctness of query results. In this paper, we take the first step to propose authenticated data integration services to ensure data and query integrity even in the presence of an untrusted integration server. We develop a novel authentication code called homomorphic secret sharing seal that can aggregate the inputs from individual sources faithfully by the untrusted server for future query authentication. Based on this, we design two authenticated index structures and authentication schemes for queries on multi-dimensional data. We further study the freshness problem in multi-source query authentication and propose several advanced update strategies. Analytical models and empirical results show that our seal design and authentication schemes are efficient and robust under various system settings.

Categories and Subject Descriptors

H.2.0 [**DATABASE MANAGEMENT**]: General—Security, integrity, and protection

General Terms

Algorithms; Experimentation; Security

Keywords

Query Authentication; Data Integration; Data Integrity

1. INTRODUCTION

Data integration, which combines data from different sources and provides users with a unified query interface, has been an essential service in database and web applications. With the recent advances in social computing and internet of things, data integration is once again under the spotlight for "big-data" applications such as listing aggregation, social media analysis, and sensor fusion. Recent work focuses on online data integration that integrates data from distributed and heterogeneous sources into a single repository, where the sources' contents may change over time [16, 25, 26, 39]. Figure 1 illustrates the online data integration and query model, where data sources or owners periodically synchronize their data with the centralized data integrator.[1]

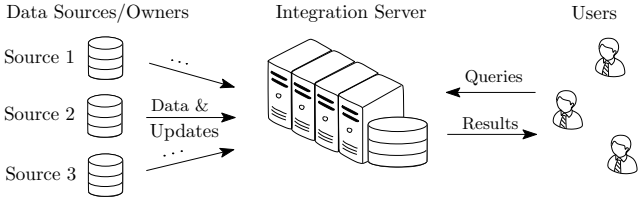


Figure 1: Multi-Source Data Integration and Query Processing

One key problem in online data integration is the data integrity [2, 5]. However, all existing work assumes that the data integrator can always be trusted and, hence, addresses data inconsistency and unreliability issues arising from the data source side. A variety of techniques on data cleaning and conflict resolution have been developed [22, 23, 26, 32, 45, 50]. Unfortunately, this assumption no longer holds in many real-life applications where the integrator might alter (leave out or forge) the integrated data and query results either intentionally or unintentionally, as shown in the following examples:

- Metasearch Engines: Internet-based price comparison websites, such as Qunar for airfares, Trivago for hotels, and Pricegrabber for consumer electronics, claim to crawl up-to-date price data from the actual service providers (e.g., airlines, travel agencies, online electronic stores, etc.). However, the query results from these search engines may not be correct or fresh because: (1) they may favor sponsors or partner service providers and hide the results of other competitors; and (2) they may fail to keep their integrated data up-to-date as promised due to a system failure or glitch. Both cases might be covered up by these engines without users' awareness.
- Meta-Analysis: Life science research, such as those on virus spread and disease control, usually requires the collection of disparate datasets for meta-analysis. Gov-

[1] We use "data source" and "data owner" interchangeably in this paper.

SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.

http://dx.doi.org/10.1145/2723372.2747649.

ernment initiatives such as DataNet have been put forward to provide a centralized repository to integrate scientific data produced by the scientists worldwide. However, in case the repository server is comprised by cyber-attacks, the query results returned by the server might be wrong.

- Collaborative Data Fusion: Online collaborative data platforms, such as Wikipedia (for documents), Wikisensing (for sensor data), and Wikidata (for structured data), allow data owners to feed large-volume data to their databases. They also allow other users to connect to their databases and build external applications based on these data. However, in many sectors such as politics and finance, the platforms may have motives to screen out or omit some critical data when exporting them to external applications.

To address these issues, in this paper we propose authenticated data integration services to ensure data integrity and query result correctness even in the presence of an untrusted data integrator. There is a substantial collection of literature on authenticated query processing, e.g., [9, 17, 21, 33, 34, 36, 42], and the core technique is the design of an *authenticated data structure* (ADS) that enables the untrusted server to prove to the users the correctness of query results. Since almost all previous studies assume only a single data source/owner, this ADS can always be constructed by the owner itself who has access to the entire dataset. However, as our problem concerns multiple sources and the integration server who has access to the entire dataset cannot be trusted, this centralized approach is no longer feasible. It calls for a distributed ADS scheme in which the ADS can be constructed based on individually authenticated data from each data owner.

The main challenge lies in the design of a mechanism that can aggregate the inputs from all data sources faithfully by the untrusted server for future query authentication. To deal with this challenge, we first propose *homomorphic secret sharing seal* (HS^3) as the underlying authentication code for integrity assurance of distributed data. Based on this, we design two ADS schemes, namely the HS^3-G-tree and HS^3-R-tree, for authenticating various types of queries on multi-dimensional data. We further study the freshness problem in multi-source query authentication and propose several advanced update strategies. To summarize, our contributions made in this paper are as follows:

- To the best of our knowledge, this is the first work that addresses the query integrity issue for large-scale, dynamic, and multi-source data integration.
- We design HS^3 authentication code that enables authentication on distributed data, based on which ADS and authentication schemes for various queries are proposed.
- We consider data updates and study the update strategies for the server to address the freshness issue in multi-source query authentication.
- We provide analytical models and empirical results to evaluate the performance and robustness of the proposed authentication schemes.

The rest of this paper is organized as follows. Section 2 introduces the research background and related work. Section 3 formally defines the problem and system model. Section 4 introduces the homomorphic secret sharing seal, followed by the query authentication schemes for multi-dimensional data in Section 5. Section 6 presents the cost model, and Section 7 studies the server update strategies. Section 8 shows the experiment results, followed by a conclusion.

2. BACKGROUND AND RELATED WORK

Inconsistency and Unreliability in Data Integration. In the data integration literature, a variety of conflict resolution techniques have been proposed to address the inconsistency and unreliability issues. Yin et al. [45] proposed the TruthFinder algorithm that resolves conflicting binary opinions during aggregation. The algorithm computes truth scores for predicates through probabilistic aggregation of independent user opinions. Pal et al. [32] studied the conflict resolution in the temporal domain and modeled the source data as observations of a hidden semi-Markovian process. Zhao et al. [50] took the confliction resolution problem from another perspective by modeling the quality of data sources. They use false positive and false negative errors as quality indicators, and their inference algorithm is based on sampling and a Bayesian probabilistic model. More recently, Li et al. [22] studied conflict resolution in a heterogenous data scenario and proposed a probabilistic source quality estimation algorithm. Besides probabilistic models, data fusion has been adopted for conflict resolution. Liu et al. [26] adopted data fusion techniques for online data aggregation. The system iteratively refines answers as more sources are probed and fusioned until it can assure that the unprobed sources cannot change the answer. Li et al. [23] also used data fusion to study the data inconsistency problem in deep web with low-accuracy sources. However, no study so far has addressed the integrity issue caused by the data integrator, which is the focus of this paper.

Query Authentication in Outsourced Databases. Existing query authentication techniques can be categorized into two groups according to the authentication data structure used: (i) chained digital signature and (ii) Merkle hash tree (MHT). The former is a public-key message authentication scheme based on asymmetric cryptography. With the signature and the public key, anyone can verify the authenticity of a message signed with a private key. Based on this scheme, early studies on query authentication impose a signature for every data value. The VB-tree [34] augments a conventional B^+-tree with a signature for each leaf entry. By verifying the signatures of all returned values, the user is convinced of the soundness of the results. To guarantee the completeness of query results further, Pang et al. [33] proposed signature chaining, which links the signatures of adjacent data values to ensure no result can be left out. Signature aggregation and chaining were adapted to multi-dimensional R-tree indexes by Cheng and Tan in [10].

The MHT was originally introduced in [28] to authenticate a large set of data values. It is a binary tree. Each leaf entry is assigned a digest based on its data value, and each internal node is assigned a digest derived from its two child nodes. The data owner signs the root digest of the MHT. For authentication purposes, besides the root signature, only the digests of those boundary nodes and query results are sent to the user. The notion of the MHT has been generalized to multi-way trees and widely adapted to various index structures. Typical examples include the Merkle-B tree and its variant the Embedded Merkle B-tree [21]. For multi-dimensional datasets and queries, similar techniques were proposed by Yang et al. to integrate an R-tree with

an MHT [42,43]. In addition to selection and range queries, recent studies have investigated the authentication of more complex queries, including aggregation queries [20], shortest paths [46], kNN queries [18, 47], top-k spatial keyword queries [41], location-based skyline queries [24], subgraph queries [15, 38] and privacy-preserving queries [9, 17]. However, in all these previous studies, only a single data owner is assumed and the database is outsourced as a whole.

Recent work has also investigated continuous query authentication over online data streams. Nath and Venkatesan [29] proposed a solution for grouped aggregation queries on data streams from a single data owner. Their solution does not require the users to be trusted and thus is publicly verifiable. Papamanthou et al. [37] investigated the problem of streaming verifiable computation, where a user (also the data owner) can later delegate some computation over the stream to the server. They designed a new authentication tree for efficient query verification on a stream. Papadopoulos et al. [35] considered linear algebraic queries, e.g., vector sums and dot products, on data streams. Their schemes are light-weight and offer strong security guarantees. However, all these solutions to online data streams only focus on a single owner and specific query types. In contrast, this paper considers the applications where the data are collectively owned by multiple sources. This makes the authentication problem more challenging since the authentication data structures need to be generated by data sources in a distributed manner.

Authenticated In-Network Aggregation in Distributed Systems. In distributed systems and sensor networks, a centralized server collects and aggregates data from the sensors. Existing studies on assuring the aggregation result accuracy can be broadly summarized into two categories: (i) verifiable sampling and (ii) commitment verification [6]. The former relies on sampling techniques, and thus offers only approximate results [3, 13, 14]. The latter usually gives precise query results. The key issue is the construction of the secure proof attached to the data. The user verifies the aggregation result according to the secure proof and/or tests the in-network aggregators/sensors with a series of challenges. Chan et al. [7] proposed a secure information aggregation (SIA) framework in which interactive proofs are used for result verification. In [8, 44], the SIA framework has been extended to support multiple hierarchical aggregators with a topology-following hash tree. To reduce the communication cost of testing, Nath et al. [30] and Zhang et al. [49] adopted one-way chains and homomorphic MACs, respectively, to make commitments. Papadopoulos et al. [36] further optimized the communication and verification costs via a secret sharing technique. However, these studies consider only application-tailored in-network aggregations such as sum and max/min, which cannot be extended to support general queries considered in this paper.

Authenticated top-k queries were recently investigated in [48] and [12]. Zhang et al. [48] investigated how to verify the authenticity and completeness of top-k queries in sensor networks. Their basic approach is to let each sensor prepare a MAC based on ordered data values. Choi et al. [12] developed several solutions for authenticated top-k aggregation in distributed databases, based on the classic Three-Phase-Uniform-Threshold algorithm. To guarantee the correctness of query results, a Skewed MHT is applied to the ordered data in each distributed database. Although both of these two studies consider distributed query processing, they assume that each processor is related to a single data source and do not address the issues arising from multiple data sources. Jain and Prabhakar [19] proposed a novel authentication mechanism to further guarantee the transactional integrity. Their solution allows multiple authorized users to run transactions on the outsourced database. All the authorized users can control the database on behalf of the (single) data owner. However, it cannot be applied to our problem where data are independently collected from multiple data sources.

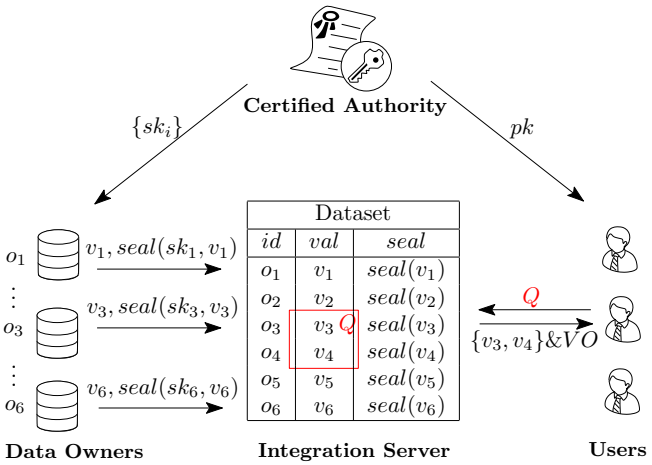


Figure 2: Query Authentication Example in Data Integration

In summary, most previous studies consider query authentication with a single data source/owner. For those studies on authenticating in-network queries in distributed systems, they can address only simple and specific aggregation types such as sum and min/max. To the best of our knowledge, there is no study on multi-source data authentication that can simultaneously support a wide range of query types.

3. PROBLEM DEFINITION AND SYSTEM MODEL

In this paper, we study the problem of authenticating query results in data integration services as shown in Figure 2. During system setup, the certified authority (CA) distributes a secret key sk_i to each data owner (DO) (for generating authentication codes) and a public key pk to the users (for verifying query results). The detailed constructions of sk_i and pk will be given in Section 4. To guarantee the freshness of dynamic data, the DOs need to update their secret keys, which may or may not involve the CA and will be further discussed in Section 7.1. Without loss of generality, we assume that each DO o_i owns a single data value v_i.[2] Upon request, each DO sends its data value v_i together with an authentication code (proposed as $seal(sk_i, v_i)$ in Section 4) to the integration server (IS), which answer queries from the users based on the collected values. Figure 2 illustrates a range query Q on a dataset of $D = \{v_1, v_2, \cdots, v_6\}$ collected independently from data owners $o_1, o_2, \cdots, o_6$. Assuming the data values $v_1, v_2, \cdots, v_6$ are sorted in the ascending order, apparently the query results are $R = \{v_3, v_4\}$.

Threat Model: While the CA and DOs are trusted, we assume that the IS is not trusted and may fabricate the

[2]For cases in which each DO owns a set of data values, we can introduce virtual DOs and associate each value with a virtual DO.

query results. Therefore, the problem in this paper is to design query authentication schemes on top of individual authentication codes so that the IS can prove to the users that the queries are executed faithfully and the results are correct. As with previous studies [21, 33], the correctness consists of two authenticity conditions: (i) *soundness*: the returned values are all genuine query results and are not tampered with; (ii) *completeness*: no genuine query results are missing. To facilitate the authentication, the IS will prepare a verification object (VO) to be returned to the users along with the query results. In Figure 2, the VO is used to verify the soundness and completeness of result set R with respect to range query Q as follows:

- No values in R are tampered with.
- All values in R are inside the query range of Q.
- All values not in R are outside the query range of Q.

Specifically, the first two conditions guarantee the soundness of query results, and the last one guarantees the completeness of query results.

Throughout this paper, we assume that no DO or CA colludes with the IS. Further, the computation and storage capacities of the untrusted IS, as an adversary, are polynomially bounded.

4. HOMOMORPHIC SECRET SHARING SEAL (HS^3)

Intuitively, the multi-source query authentication problem involves two main issues: (i) the design of the authentication code for each data value, and (ii) the construction of the VO for each query based on the collected data values and authentication codes. A naive solution is to use a digital signature $sig(v_i)$ as the authentication code for each value v_i. For any query, the IS simply returns the whole dataset, $\{(v_1, sig(v_1)), (v_2, sig(v_2)), \cdots\}$, to the user. To authenticate the query results, the user first verifies the integrity of the dataset through the signatures, and then computes the query results using the genuine dataset. Obviously, this solution has its computation and communication complexities both as high as $O(n)$, where n is the number of DOs.

To improve the performance, we propose *homomorphic secret sharing seal* (*HS^3*), a novel distributed authentication code for integrity verification in the multi-source environment. The design principle of HS^3 is two-fold: (1) individual codes can be gathered collectively by an untrusted IS; (2) to prove a set of values is not query results does not need the authentication codes of all these values, especially those whose values are faraway from the query range. Therefore, we "divide" the authentication code of a value into pieces and use only relevant ones for the authentication of a specific query. Further, by sharing common pieces of such codes, multiple values can be authenticated as a whole, which significantly improves the efficiency. In the rest of this section, we first introduce some basic cryptographic functions, followed by the detailed HS^3 design and query authentication process. Finally, we give some formal security analysis.

4.1 Preliminaries

In this subsection, we introduce some preliminaries used to construct the HS^3 code.

Cryptographic Hash Function: A cryptographic hash function $h(\cdot)$ takes an arbitrary-length string and returns a fixed-length bit string. It is collision-resistant, i.e., it is difficult to find two different messages m_1 and m_2 such that $h(m_1) = h(m_2)$. Some examples include MD5 and SHA-1.

Pseudorandom Function: Let $f_k(\cdot) : \{0,1\}^* \times \{0,1\}^* \rightarrow \{0,1\}^*$ be an efficient, length-preserving, keyed function, where k is a string chosen at random and can be regarded as a secret key. We say $f_k(\cdot)$ is a *pseudorandom function* if for all probabilistic polynomial-time (PPT) distinguishers, there exists only a negligible possibility to distinguish $f_k(\cdot)$ from a truly random string.

Secret Sharing Scheme: A secret sharing scheme [27] is related to key establishment. The idea of secret sharing is to distribute a secret s amongst n users such that only when all n parties together can the original secret s be reconstructed. The procedure is as follows. The key distributer first generates $n-1$ random values $ss_1, ss_2, \cdots, ss_{n-1}$ (called secret shares) using a pseudorandom function $f_k(\cdot)$ and distributes them to the first $n-1$ users. The last one is given as $ss_n = s - \sum_{i=1}^{n-1} ss_i$. The secret is recovered by $s = \sum_{i=1}^{n} ss_i$.

RSA-based Homomorphic Signature $\mathcal{G}(\cdot,\cdot)$: An RSA cryptosystem has a secret key $sk = (z, d)$ and a public key $pk = (z, e)$, where z is the product of two random large primes p, q. The keys d and e satisfy $de = 1 \mod \phi(z)$, where $\phi(z) = (p-1)(q-1)$. The RSA cryptosystem can be used to generate RSA *digital signatures*. Let $h(\cdot)$ denote a cryptographic hash function; the RSA digital signature over a message m is computed as $sig(m) = h(m)^d \mod z$. The signature can be verified by checking $h(m) = sig(m)^e \mod z$. The RSA-based homomorphic signature $\mathcal{G}(\cdot,\cdot)$ for a message m is defined as $\mathcal{G}(r, h(m)) = (r \cdot g^{h(m)})^d \mod z$, where r is a random value and g is a group generator.[3] Given r, g, and e, the signature can be verified by checking $r \cdot g^{h(m)} = \mathcal{G}(r, h(m))^e \mod z$. This signature function has a homomorphic property as follows: $\mathcal{G}(r_1, h(m_1)) \cdot \mathcal{G}(r_2, h(m_2)) = \mathcal{G}(r_1 \cdot r_2, h(m_1) + h(m_2))$.[4]

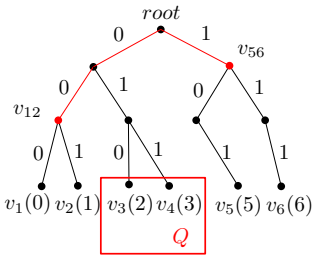
4.2 HS^3 Design and Query Authentication

Recall that the intuition behind the HS^3 design is to utilize some common properties of the data to support efficient authentication of a specific query. Without loss of generality, we assume that all data values are distinct integers. As such, the IS can sort them and build a prefix tree out of their binary forms. Figure 3(a) is a prefix tree for the dataset in Figure 2.

Based on the prefix tree, the basic idea of HS^3 is to merge the authentication codes of non-result values with a common prefix, thus allowing them to be verified as a whole. For example in Figure 3(a), with prefix '00', it can be inferred that v_1 and v_2 (under v_{12}) will not be in the range $[2, +\infty)$. Consider a specific range query $Q = [2, 3]$ in Figure 3(a) whose result set is $R = \{v_3(2), v_4(3)\}$. To authenticate result values v_3 and v_4, the IS can return their individual authentication codes to the user, who can therefore verify that they genuinely originate from DOs and have not been tampered with. To authenticate all other values are non-result values, i.e., they are outside Q, the IS can simply return the authentication codes of intermediate nodes v_{12} and v_{56}, because the values that belong to these nodes share common

[3] In group theory, a generator of a group G is an element $g \in G$ such that all elements in G can be generated as powers of g.

[4] Note that this is different from an additive homomorphic digital signature, which requires $sig(m_1) \otimes sig(m_2) = sig(m_1 + m_2)$.



(a) Prefix Tree

id	v	*seal*
v_1	000	$S_1 = \mathcal{G}(g^{ss_1}, h(000) \mid h(00) \mid h(0))$
v_2	001	$S_2 = \mathcal{G}(g^{ss_2}, h(001) \mid h(00) \mid h(0))$
v_3	010	$S_3 = \mathcal{G}(g^{ss_3}, h(010) \mid h(01) \mid h(0))$
v_4	011	$S_4 = \mathcal{G}(g^{ss_4}, h(011) \mid h(01) \mid h(0))$
v_5	101	$S_5 = \mathcal{G}(g^{ss_5}, h(101) \mid h(10) \mid h(1))$
v_6	111	$S_6 = \mathcal{G}(g^{ss_6}, h(111) \mid h(11) \mid h(1))$
S_{12}	$= S_1 \otimes S_2$	
	$= \mathcal{G}(g^{ss_1+ss_2}, h(000) + h(001) \mid 2h(00) \mid 2h(0))$	
S_{56}	$= S_5 \otimes S_6 = \mathcal{G}(g^{ss_5+ss_6}, \cdots \mid \cdots \mid 2h(1))$	

(b) Table of Seals

Figure 3: Prefix Tree Index

prefixes '00' and '1', respectively, which are outside Q. And since these two nodes already cover all non-result values, it suffices for the user to verify the authenticity of v_{12} and v_{56} only, thereby reducing the communication and computational overheads. In general, for a query Q, the IS finds the set of nodes to return by recursively descending the prefix tree until the values in every node are all non-result.

As HS^3 enables a value to be authenticated by prefix, traditional digital signatures cannot work as authentication codes here. As such, we design *seal* as a new authentication code. In the following, we first present the seal data structure and its generation process. Then, we discuss how to build the authenticated prefix tree based on seals, followed by the detailed authentication process for range and aggregation queries.

4.2.1 Seal Data Structure

The seal of a value should satisfy the following requirements: (i) all prefixes of the value are embedded and verifiable individually; (ii) seals can be merged (as we call "folded") by their common prefixes; and (iii) the completeness of the dataset can be verified through a single seal that folds all individual seals. To satisfy the first two requirements, a seal includes not only the digest of the value as a signature does, but also the digests of all its prefixes. Thus, we define the digest set of a value as follows:

Definition 1. *Digest Set. Let $v_{i,(j)}$ be the prefix of a value v_i with length j, $h(\cdot)$ be a cryptographic hash function, and '$|$' denote string concatenation. The digest set dig_i of a value v_i is defined as:*

$$dig_i = h(v_{i,(l)}) | \cdots | h(v_{i,(2)}) | h(v_{i,(1)}).$$

The upper part of Figure 4 illustrates the detailed data structure of dig_i. Note that for each digest, $log_2 n$ bits are reserved at the beginning to prevent additive overflow in seal folding (detailed in Section 4.2.3), where n is the number of DOs.

Based on the digest set, we define the seal data structure as follows:

Definition 2. *Seal Data Structure. Denote the homomorphic signature function introduced in Section 4.1 by $\mathcal{G}(\cdot,\cdot)$ and a group generator by g. Let sk_i be a secret key, including the signing key of $\mathcal{G}(\cdot,\cdot)$ and a powered secret share g^{ss_i}, and dig_i be the digest set. The seal of a data value v_i is defined as:*

$$S_i = seal(sk_i, v_i) = \mathcal{G}(g^{ss_i}, dig_i).$$

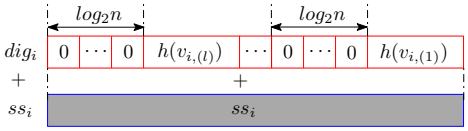


Figure 4: Content of Seal S_i

The secret shares are used to fulfill the last requirement mentioned above: each DO binds its secret share to its own seal and thus the secret s of the whole dataset can be recovered correctly only if there is no missing seal from any DO. Note that each secret share ss_i is given in a powered form of g^{ss_i} so that the plaintext of ss_i can be protected against the adversary, due to the hardness of the discrete logarithm problem [27]. To verify whether n secret shares agree with the original secret s, one can check $\prod_{i=1}^{n} g^{ss_i} = g^s \mod z$. Figure 3(b) shows the seals $S_1, S_2, \cdots, S_6$ for the data values $v_1, v_2, \cdots, v_6$; e.g., $S_1 = \mathcal{G}(g^{ss_1}, h(000)|h(00)|h(0))$.

4.2.2 Key Distribution and Seal Generation

After defining the seal data structure, we are ready to describe the key distribution process. During system setup, the CA distributes a secret key sk_i, including the signing key of the homomorphic signature function $\mathcal{G}(\cdot,\cdot)$ and the (powered) secret share g^{ss_i}, to each DO; it also distributes a public key pk, including the verifying key of $\mathcal{G}(\cdot,\cdot)$ and the (powered) total secret g^s to the users for result verification. During data collection, each DO o_i uses sk_i to generate its $seal(sk_i, v_i)$ for data value v_i. It is worth noting that the CA is not involved in the data collection process and can be offline after the key distribution for the static data case (the dynamic data case will be further discussed in Section 7).

4.2.3 Seal Folding and Authenticated Prefix Tree

Once the IS collects individual data values and their seals from the DOs, it can build the authenticated data structure for the prefix tree by *folding* the common prefixes of seals in a bottom-up manner. Note that the construction of the authenticated prefix tree only involves the IS, thanks to the homomorphic property of $\mathcal{G}(\cdot,\cdot)$. For example, as v_1 and v_2 share common prefixes '0' and '00', S_{12} is computed as:

$$\begin{aligned}
S_{12} &= S_1 \otimes S_2 \\
&= \mathcal{G}(g^{ss_1}, h(000)|h(00)|h(0)) \cdot \mathcal{G}(g^{ss_2}, h(001)|h(00)|h(0)) \\
&= (g^{ss_1} \cdot g^{h(000)|h(00)|h(0)})^d \cdot (g^{ss_2} \cdot g^{h(001)|h(00)|h(0)})^d \mod z \\
&= (g^{ss_1+ss_2} \cdot g^{h(000)|h(00)|h(0)+h(001)|h(00)|h(0)})^d \mod z \\
&= (g^{ss_1+ss_2} \cdot g^{h(000)+h(001)|2\cdot h(00)|2\cdot h(0)})^d \mod z \\
&= \mathcal{G}(g^{ss_1+ss_2}, h(000) + h(001)|2 \cdot h(00)|2 \cdot h(0)).
\end{aligned}$$

Similarly, $S_{56} = S_5 \otimes S_6 = \mathcal{G}(g^{ss_5+ss_6}, \cdots | \cdots | 2 \cdot h(1))$.

4.2.4 Authenticating Range Queries

After the IS builds the authenticated prefix tree, it can provide query services to the users. We use Figure 3 as an example to illustrate the query authentication process. To authenticate a range query Q, the verification object (VO) returned to the user should include: (i) the seals of result values v_3 and v_4, i.e., S_3 and S_4; (ii) the (folded) seals of all

non-result values, i.e., S_{12} and S_{56}; and (iii) all the components necessary to verify the seals, i.e., each seal's (folded) secret share, count of folded seals, corresponding prefix, and other prefix digests. As for (ii), individual seals are folded by their longest common prefixes that do not overlap with Q, i.e., '00' in node v_{12} and '1' in node v_{56} (as shown in light grey in Figure 3(b)).

The user verifies the correctness of query results in terms of two aspects: (i) the completeness, by multiplying up the (folded) secret shares from all returned seals (as shown in dark grey in Figure 3(b)) and comparing it against the original secret g^s, which is known to the user; (ii) the soundness, by verifying that each result value lies inside the query range and that each non-result value lies outside the query range. For (ii), the user first checks whether each received prefix (e.g., '010', '00') is truly a prefix of the corresponding node (e.g., v_3, v_{12}). Take v_{12} as an example; the VO received by the user includes the following components: the folded secret share (i.e., $g^{ss_1+ss_2}$), the count of folded seals (i.e., 2), the corresponding prefix (i.e., '00'), and the other prefix digests (i.e., $h(000) + h(001)$, $2 \cdot h(0)$). For verification, the user computes the digest of the prefix '00' (i.e., $h(00)$) itself and compares $g^{ss_1+ss_2} \cdot g^{h(000)+h(001)|2\cdot h(00)|2\cdot h(0)}$ with $(S_{12})^e$ in the module domain. If they agree, the user is assured that S_{12} has common prefix '00' and therefore is outside Q. Similarly, the user verifies that prefixes '010' and '011' are truly prefixes of v_3 and v_4, respectively, and thus v_3 and v_4 are inside Q; and prefix '1' is truly a prefix of v_{56} and thus v_5 and v_6 are outside Q.

4.2.5 Authenticating Aggregation Queries

HS^3 can also support a wide range of aggregation queries. In what follows, we discuss the authentication schemes for *sum* (*count*) and *top-k* queries, from which many other aggregation queries can be derived. For example, an *average* query can be decomposed into a sum query and a count query, and a *max* query is equivalent to a top-1 query.

- Sum (count) query. A sum query aggregates all the values in a query range. To support authenticating such a query, each DO augments the digest set dig_i with $v_i \cdot dig_i$ as if the DO folds the original seal v_i times. As such, the authentication process is the same as that for a range query, except that the user additionally sums up the values in relevant digests as the sum result. A count query is a special case of a sum query, in which the counts of result values are summed up.
- Top-k query. A top-k query returns the largest k values among all DOs. Since the values are sorted in the prefix tree at the IS, the top-k results are simply the right-most k values. As such, a top-k query is equivalent to a range query with the range $[v_{n-k}, +\infty]$.

4.3 Security Analysis

In this subsection, we analyze the security of the proposed authentication code HS^3. We first prove that the seal in HS^3 is secure against forgeries. Then, we show that the seal folding in HS^3 is at least as secure as the *batch verification* [4]. At last, we prove that HS^3 guarantees the correctness of query results.

LEMMA 1. *The signature $\mathcal{G}(\cdot,\cdot)$, i.e., seal$(\cdot,\cdot)$, is secure against forgeries, if $r \cdot g^{h(\cdot)}$ can be modeled as a random oracle and the RSA problem is hard.*

PROOF. See Appendix A for the proof. □

LEMMA 2. *The batch verification (also called* fast screening *[4]) of the seals in HS^3 is secure, if $r \cdot g^{h(\cdot)}$ can be modeled as a random oracle and $RSA_{(e,n)}$ is a one-way function.*

PROOF. See Appendix B for the proof. □

LEMMA 3. *The seal folding in HS^3 is at least as secure as the batch verification.*

PROOF. See Appendix C for the proof. □

THEOREM 3. *The HS^3 scheme guarantees the correctness of query results in presence of a PPT adversary.*

PROOF. The theorem is proved from the two aspects:

- Soundness. According to Lemmas 1 and 3, the (folded) seal in HS^3 is secure against forgeries.
- Completeness. The completeness of HS^3 derives from the secret sharing scheme. Since each secret share is generated by a pseudorandom function, it is indistinguishable from random and unpredictable by the adversary. Suppose some DOs' data values are dropped, the secret shares of those DOs will also be missing, which would result in the impossibility of final secret recovery.

□

5. AUTHENTICATING MULTI-DIMENSIONAL DATA

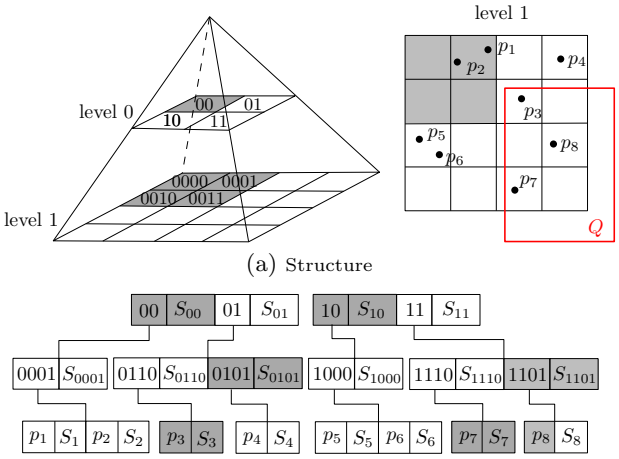
This section extends the HS^3 authentication code to multi-dimensional data. Typical examples of multi-dimensional data include object locations and multi-attribute review scores. We propose two authenticated data structures (ADS), also known as authenticated indexes or simply indexes, based on HS^3. For ease of presentation, we assume the dimensionality is two and thus use "points" and "values" interchangeably, while the extension to higher dimensions follows unless otherwise stated. Also following the previous section, all coordinates are integers, and a data point can be represented by a string that concatenates its binary representation in both dimensions.

5.1 HS^3-G-tree

The first ADS integrates HS^3 with a multi-layer grid system and is therefore called an HS^3-G-tree. A multi-layer grid system partitions the space into multiple levels of grid cells in a recursive manner. Figure 5 shows an example for dataset $\{p_1, \cdots, p_8\}$. In this paper, we assume every cell is partitioned into four sub-cells in the next level of the grid. For example, the grey cell encoded by '00' in Figure 5(a) is further partitioned into cells of '0000', '0001', '0010', and '0011'. Therefore, an HS^3-G-tree is a 4-ary tree. Without loss of generality, we do not presume any specific partitioning algorithm for this multi-layer grid. Nonetheless, we assume that this partitioning is known to all the DOs, the IS, as well as the querying user.

The detailed data structure of an HS^3-G-tree is illustrated in Figure 5(b). Each tree node corresponds to a grid cell, and is associated with a value v (e.g., '00') and a seal (e.g., S_{00}). Here v denotes the space encoding of the cell, and the seal, according to HS^3, only applies to the cells that contain data points. For example, the cell encoded by '0000' in Figure 5(a) does not have a corresponding seal and, hence, is suppressed in Figure 5(b).

The structure of a seal is defined similar to HS^3 as in Figure 4. Specifically, the seal of a leaf entry p_i (i.e., a



(a) Structure

(b) Index

Figure 5: HS^3-G-tree

data point) is defined as $\mathcal{G}(g^{ss_i}, dig_i)$, where dig_i concatenates the digests of its space encoding in decreasing precisions, together with $h(p_i)$. For example, the seal of p_1 is $S_1 = \mathcal{G}(g^{ss_1}, h(p_1)|h(0001)|h(00))$. And the seal of a non-leaf entry can be computed by the IS by folding the common prefix of its child entries. For example, p_1 and p_2 share a common prefix of '0001'; so the seal of node '0001' is $S_{0001} = S_1 \otimes S_2 = \mathcal{G}(g^{ss_1+ss_2}, h(p_1)+h(p_2)|2 \cdot h(0001)|2 \cdot h(00))$. As such, the IS can build the HS^3-G-tree in a bottom-up manner after the DOs send the value and seal of each data point.

5.1.1 Authenticating Range Queries

Algorithm 1 ServerQuery(range query Q, node t)

```
1: if t partially intersects with Q then
2:   for each child c of t do
3:     ServerQuery(Q, c)
4: else
5:   if t is an internal node then
6:     append v_t and seal S_t to the VO
7:     if t is fully contained in Q then
8:       append all points {p_i} under t to the VO
9:   else
10:    append p_t and seal S_t to the VO
11:  append g^{ss_t}, cnt_t, and other digests to the VO
```

Upon receiving a range query, the IS prepares a verification object (VO) by a depth-first traversal (see Algorithm 1). The VO consists of two types of entries: (i) node t that does not partially intersect with Q (Line 6); and (ii) leaf entry t that is visited (Line 10). For the former type of entries, if t is fully contained in Q, all data points under it are results and thus appended to the VO (Lines 7-8). For the query Q in Figure 5, the algorithm traverses the nodes and entries in order of '00', '01', '0110', p_3, '0101', '10', '11', '1110', p_7, '1101'; hence, the VO consists of ('00',S_{00}), (p_3, S_3), ('0101', S_{0101}), ('10', S_{10}), (p_7, S_7), ('1101', S_{1101}, $\{p_8\}$) (highlighted in grey color in Figure 5(b)). Here we do not list the secret shares, seal counts, and other digests in the interest of space.

After the user receives the VO, it verifies the correctness of query results by Algorithm 2. It first verifies the completeness (i.e., all data values have been included) by checking whether the powered secret g^s returned by this algorithm is the same as the public, genuine one. Then, the user verifies the returned seals for soundness, i.e., they are indeed inside or outside the query range (Lines 6–7). Followed by a checking of the result points (Lines 8–12), the returned results are verified not to be tampered with. In Figure 5, the verification procedure is explained as follows. The user first extracts $g^{\sum_{i=1}^{2} ss_i}$, $g^{\sum_{i=5}^{6} ss_i}$ from the VO components associated with S_{00}, S_{10}, and $g^{ss_3}, g^{ss_4}, g^{ss_7}, g^{ss_8}$ from the VO components associated with $S_3, S_{0101}, S_7, S_{1101}$; and then recovers $g^s = g^{\sum_{i=1}^{8} ss_i}$. The user further checks that all VO entries are genuinely from the DOs by the seals S_{00}, S_{10} and $S_3, S_{0101}, S_7, S_{1101}$. It also checks that p_8 matches the digest in the folded seal S_{1101}. Finally, the user verifies that '0101', '00', and '10' are completely outside Q while p_3, p_7, p_8, and '1101' are inside Q.

Algorithm 2 UserVerify(verification object VO)

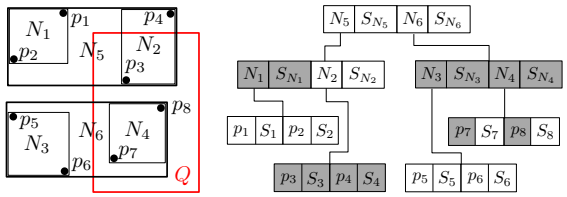
```
1: g^s := 1
2: while VO has next entry do
3:   get next entry
4:   parse v_t, S_t, g^{ss_t}, cnt_t, and other prefix digests from this
     entry (t is the current tree node)
5:   update g^s := g^s · g^{ss_t}  mod z
6:   if g^{ss_t} · g^{...|cnt_t·h(v_t)|...} ≠ (S_t)^e  mod z or v_t partially in-
     tersects with Q then
7:     return −1
8:   if t is fully contained in Q then
9:     extract h({p_i}) from this entry and set h_t := 0
10:    for each p_i under t do
11:      update h_t := h_t + h(p_i)
12:    if h_t ≠ h({p_i}) then return −1
13: return g^s
```

The security analysis of HS^3-G-tree is given in Appendix D.

5.2 HS^3-R-tree

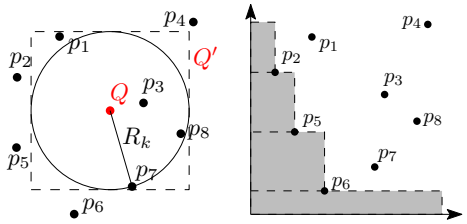
One possible disadvantage of HS^3-G-tree is that the data points may share few common prefixes due to the dimensionality curse. In the worse case, all points may reside in one leaf node, which makes the grid topology useless. To address the skewness of the point distribution, we propose to adopt a clustering-based multi-dimensional index, and a typical example is an R^*-tree. The main difference is that, instead of sharing common prefixes, data points share common Minimum Bounded Boxes (MBBs) and have their seals folded by common MBBs. Figure 6 shows the R^*-tree for the same dataset as in Figure 5. In this example, data points p_1 and p_2 share common MBBs N_1 and N_5. Based on this, we propose HS^3-R-tree that integrates R^*-tree with the HS^3 scheme.

The detailed data structure of an HS^3-R-tree is illustrated in Figure 6. Similar to the HS^3-G-tree, each tree node is associated with an MBB (e.g., N_1) and a seal (e.g., S_{N_1}). The seal of a leaf entry p_i (i.e., a data point) is defined as $\mathcal{G}(g^{ss_i}, dig_i)$. Specifically, given a data point p_i, dig_i is given as $h(p_i)|h(N_{i,(l)})|\cdots|h(N_{i,(1)})$, where $N_{i,(j)}$ denotes the MBB of the j-th level node in the R^*-tree. For example, the seal of p_1 is $S_1 = \mathcal{G}(g^{ss_1}, h(p_1)|h(N_1)|h(N_5))$. And the seal of an internal node can be computed by the IS by folding the common prefixes of its child entries. For instance in Figure 6, p_1, p_2 share common MBBs N_1, N_5; so the seal of N_1 is $S_{N_1} = S_1 \otimes S_2 = \mathcal{G}(g^{ss_1+ss_2}, h(p_1)+h(p_2)|2 \cdot h(N_1)|2 \cdot h(N_5))$. As such, the IS can build the HS^3-R-tree in a bottom-up manner after the DOs send in the value and seal of each data point. The difference from an HS^3-G-tree, however, is



(a) Structure (b) Index

Figure 6: HS^3-R-tree



(a) kNN Query (b) Skyline Query

Figure 7: Complex Queries

that since the R^*-tree can only be built after gathering all data points, the construction of an HS^3-R-tree must be performed in two phases. In the first phase, each DO sends its data point to the IS, who builds the R^*-tree and returns the common MBBs. In the second phase, each DO sends the seal computed upon the common MBBs to the IS, who then computes the folded seals for all R^*-tree nodes. Since a DO has to retrieve the common MBBs before a seal can be generated, this would incur more communication overhead and defer the computation of folded seals at the IS. It is worth noting that, no matter how the IS builds the R^*-tree, it will not affect the correctness of the verification result but only the verification efficiency.

5.2.1 *Authenticating Range Queries*

Upon receiving a range query Q, similar to the HS^3-G-tree, the IS prepares a verification object (VO) with a depth-first traversal on the HS^3-R-tree as in Algorithm 1, except that v_t is replaced with N_t. In Figure 6(a), the IS will visit $N_5, N_1, N_2, p_3, p_4, N_6, N_3, N_4$ in order; so the VO includes $(N_1, S_{N_1}), (p_3, S_3)$, (p_4, S_4), (N_3, S_{N_3}), $(N_4, S_{N_4}, \{p_7, p_8\})$, as shown by the shaded entries in Figure 6(b).

After the user receives the VO, it verifies the correctness of query results similarly to Algorithm 2, except that v_t is replaced with N_t. In Figure 6, the user first extracts $g^{\sum_{i=1}^{2} ss_i}$, g^{ss_3}, g^{ss_4}, $g^{\sum_{i=5}^{6} ss_i}$, and $g^{\sum_{i=7}^{8} ss_i}$ from the VO components associated with $S_{N_1}, S_3, S_4, S_{N_3}$, and S_{N_4}, from which it recovers $g^s = g^{\sum_{i=1}^{8} ss_i}$. The user then checks that all VO entries are genuine by the seals $S_{N_1}, S_{N_3}, S_{N_4}$ and S_3, S_4. Finally, the user verifies p_4, N_1, and N_3 are completely outside Q while p_3, N_4 are inside Q.

The security analysis follows that of the HS^3-G-tree.

5.3 Other Types of Queries

By converting to range queries, we can also authenticate other types of queries on the HS^3-G-tree or HS^3-R-tree, including *k-nearest-neighbors* (kNN) and *skylines* queries.

- kNN query. Given a query point Q, a kNN query returns the k nearest points to Q. To process this query, the IS first performs a kNN search and then reduces it to a range query Q' with range R_k, which is the circumscribed square of the circle that centers at Q and touches the kth-NN. Figure 7(a) shows a 3NN example, where the query is reduced to a range query Q' with range R_k. Once the user verifies that only p_3, p_7, and p_8 are in the range of R_k, it can confirm that they are the 3NN results.
- Skyline query. A skyline query returns all points that are not dominated by the others in the dataset. A point p_i dominates another point p_j if and only if in all dimensions, the value of p_i is no larger than that of p_j. The skyline in Figure 7(b) includes p_2, p_5, and p_6. These skyline results can be verified by four range queries in the shaded area of Figure 7(b). Once the user verifies that no other points except p_2, p_5, and p_6 are in this area, it can confirm that these three points constitute the skyline results.

6. ANALYTICAL MODELS

In this section, we develop cost models for the proposed seal and indexes. It gives us valuable insights to the expected performance and proper choice of certain system settings.

6.1 Cost Model for Seal

Recall that the seal of a value v_i is defined as $seal(sk_i, v_i) = \mathcal{G}(g^{ss_i}, dig_i)$ consisting of two parts: the secret share part g^{ss_i} and the digest set dig_i. Let M_{digest} denote the length of a digest, respectively. Let $C_{pf}, C_{pow}, C_{hash}, C_{enc}, C_{dec}$ denote the CPU costs of a pseudorandom function operation, a power operation, a hash operation, encryption and decryption operations, respectively. Let n be the upper bound of the DO population in the system and l be the number of digests in the digest set of a seal.

Although the seal length is determined by the function $\mathcal{G}(\cdot,\cdot)$, it is essentially at least larger than the plaintext of the seal content. Thus, the space cost for a seal can be derived as:

$$M_{seal} = l \cdot (log_2^n + M_{digest}).$$

The CPU cost of generating a seal consumes at preparing and encrypting the seal content, which can be modeled as:

$$C_{gen} = C_{pf} + l \cdot C_{hash} + C_{dec} + C_{pow}.$$

To verify a folded seal, the user first decrypts it and then checks the correctness of the content. Note that the seal verification procedure differs according to whether the seal contains a result or not. Suppose the seal is folded based on m results, the total CPU cost is:[5]

$$C_{verify} = C_{enc} + m \cdot C_{hash} + C_{hash} + C_{pow}.$$

6.2 Cost Model for HS^3-G-tree

We focus on the indexing cost at the server and the VO cost incurred for a range query. For ease of presentation, the whole data space is normalized into a d-dimensional unit space $[0,1]^d$. For simplicity, we further assume the values from the DOs are drawn from a uniform distribution, and estimate the height of the HS^3-G-tree, which partitions each dimension equally, as $w = \lceil log_{2^d}(n) \rceil$.

The space cost at the server includes the individual seals and the seals of all non-leaf entries, which is modeled as:

$$M^G = (n + \sum_{i=0}^{w-1} \alpha_i \cdot 2^{i \cdot d}) \cdot M_{seal}, \quad (1)$$

where α_i represents the ratio of tree nodes occupied by data points at level i and can be estimated as $\frac{min(n, 2^{i \cdot d})}{2^{i \cdot d}}$.

[5]If the seal contains no results, only the common prefix is needed and, thus, $m \cdot C_{hash}$ is not necessary.

The CPU cost for the server to construct the HS^3-G-tree consumes only at seal folding, which is modeled as:

$$C^G_{con} = (\sum_{i=0}^{w-1} \beta_i \cdot 2^{i \cdot d}) \cdot C_{\otimes},$$

where β_i is the average entry number of a tree node at level i and $C_{\otimes}$ is the folding cost. For a leaf node, the average entry number is $max(0, \lceil \frac{n}{2^{w \cdot d}} \rceil - 1)$. For an internal node at level i, it has $max(0, \lceil \frac{\alpha_{i+1}}{\alpha_i} \cdot 2^d \rceil - 1)$ entries on average.

Next, we try to analyze the number of visited entries for a range query Q. According to [31], the probability that two random rectangles R_1, R_2 overlap is:

$$Pr_{overlap}(R_1, R_2) = \prod_{j=1}^{d} (R_1.L_j + R_2.L_j), \qquad (2)$$

where $R.L_j$ means the length of R in dimension j. Since the HS^3-G-tree partitions the space equally, the space length of an entry at level i is 2^{-i}. Given a uniform distribution of the DOs, the space length of a leaf entry is estimated as $\sqrt[d]{1/n}$. Thus, the number of entries visited can be modeled as:

$$N^G_{vis} = \sum_{i=0}^{w-1} 2^{i \cdot d} \prod_{j=1}^{d} (2^{-i} + Q.L_j) + n \cdot \prod_{j=1}^{d} (\sqrt[d]{\frac{1}{n}} + Q.L_j). \quad (3)$$

The costs of transmitting, preparing, and verifying the VO are all proportional to the number of visited nodes. Specifically, they can be derived as $M_{VO} = N^G_{vis} \cdot M_{seal}, C_{preVO} = N^G_{vis} \cdot C_{access}, C_{vrfVO} = N^G_{vis} \cdot C_{verify}$, respectively, where C_{access} is the CPU cost to access a tree node.

6.3 Cost Model for HS^3-R-tree

Let u_l denote the average fanout of a leaf node and u_i the average fanout of an internal node at level i. Denote by $\bar{u}_i$ the average of u_i's across different levels. Then, the height of the HS^3-R-tree is $w = 1 + \lceil log_{\bar{u}_i}(n/u_l) \rceil$.

The space cost at the server includes the seals of all leaf and non-leaf entries, which is modeled as:

$$M^R = (n + \sum_{i=0}^{w-2} U_i + n/u_l) \cdot M_{seal}, \qquad (4)$$

where U_i is the total number of entries at level i and is estimated as $U_i = \prod_{j=0}^{i} u_j$.

The CPU cost for the server to construct the HS^3-R-tree consumes only at seal folding, which is modeled as:

$$C^R_{con} = (\sum_{i=1}^{w-1} (U_{i-1} \cdot (u_i - 1)) + U_{w-1} \cdot (u_l - 1)) \cdot C_{\otimes}.$$

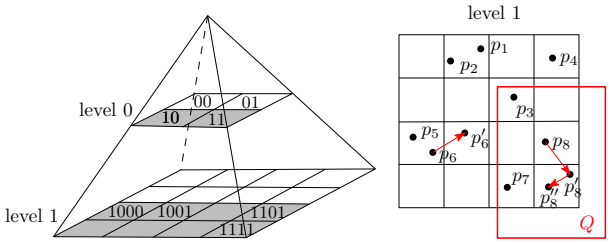
Similar to the model of HS^3-G-tree, we next estimate the entries visited in the HS^3-R-tree for a range query Q. Following [40], we assume all the nodes at the same level partition the whole space equally; thus the length of a non-leaf entry is $\sqrt[d]{1/U_k}$. The length of a leaf entry is the same as that of the HS^3-G-tree. We model the number of entries visited as:

$$N^R_{vis} = \sum_{i=0}^{w-1} U_i \prod_{j=1}^{d} (\sqrt[d]{\frac{1}{U_i}} + Q.L_j) + n \cdot \prod_{j=1}^{d} (\sqrt[d]{\frac{1}{n}} + Q.L_j). \quad (5)$$

The VO size, the VO preparing time, and the verification time are linear to the number of visited nodes in the tree. Based on Eqs. (1) and (4), when the HS^3-G-tree has a smaller fanout than the HS^3-R-tree, it results in more internal nodes and thus higher storage overhead. Since the HS^3-R-tree is generally more compact and thus fewer nodes to be visited, it is expected to have a lower authentication cost.

7. HANDLING DATA UPDATES

In this section, we study the update strategies for both of the proposed HS^3-G-tree and HS^3-R-tree. We start by considering a single-value update and propose a basic update scheme. We then present the optimized schemes of HS^3-G-tree and HS^3-R-tree for batch updates.



(a) Update on HS^3-G-tree

Case for p_6
① $S_{10} = S_5 \otimes S_6, \mathcal{L}_{10} = \{S_5, S_6\}, \mathcal{U}_{10} = \{\}$
④ $\mathcal{U}_{10} = \{S'_6\}$
$S_{10} = S_{10} \otimes S_6^{-1} \otimes S'_6, \mathcal{L}_{10} = \{S_5, S'_6\}, \mathcal{U}_{10} = \{\}$

Case for p_8
② $S_{11} = S_7 \otimes S_8, \mathcal{L}_{11} = \{S_7, S_8\}, \mathcal{U}_{11} = \{\}$
③ $\mathcal{U}_{11} = \{S'_8\}$
④ $\mathcal{U}_{11} = \{S''_8\}$
$\mathcal{U}_{11} = \{\}, \mathcal{U}_{1111} = \{S''_8\}, \mathcal{U}_{1101} = \{S''_8\}$
$S_{1111} = S''_8, \mathcal{L}_{1111} = \{S''_8\}, \mathcal{U}_{1111} = \{\}$

① $p_6 \to p'_6$ ② $p_8 \to p'_8$ ③ $p'_8 \to p''_8$ ④ Q is issued

(b) Update Sequence

Figure 8: Lazy Update

7.1 Basic Update Scheme

To support updates, we augment the definition of secret share with the latest update time. Formally, when a DO o_i updates its data at time τ, the secret share g^{ss_i} of this DO becomes $g^{ss_i(\tau)} = g^{f_k(i|\tau)}$, where $f_k(\cdot)$ is a pseudorandom function, and accordingly the latest total secret g^s becomes $g^{s(\tau)} = \prod_{j=1}^{i-1} g^{ss_j} \cdot g^{ss_i(\tau)} \cdot \prod_{j=i+1}^{n} g^{ss_j}$. During result verification, the latest total secret $g^{s(\tau)}$ is used to verify the correctness and freshness of query results.

There are two possible secret update models that can be adopted based on the data update frequency. For highly dynamic data such as airfares, the IS can schedule in advance an update agenda for all the DOs. For instance, if there are 3 DOs where o_1 updates twice as frequently as o_2 and o_3, a possible data update agenda can be $\{U_1, U_2, U_1, U_2, \cdots\}$ for time points $\tau_1, \tau_2, \tau_3, \tau_4, \cdots$, where $U_1 = \{o_1, o_2\}$ and $U_2 = \{o_1, o_3\}$. Then, instead of sending the secret share $g^{ss_i(\tau_j)}$ to each updated DO at every time point, the CA sends each DO the group generator g and a seed k_i during system setup so that the DOs can generate the secret shares locally as scheduled. Specifically, at time τ_j, a DO computes a new secret share $g^{ss_i(\tau_j)} = g^{f_{k_i}(i|\tau_j)}$ if and only if there is a scheduled update. Since the total secret at any time can be precomputed from the secret share of each DO at that time, the CA can publish it to the users in advance and is therefore no longer involved after system setup. For infrequently-updated data, however, the CA needs to be involved in the update process. Once a DO needs to update its value and seal, it requests a new secret share from the CA, who then updates the total secret share to the users and revokes the previous one. Nevertheless, as updates are infrequent, the overhead of the CA is expected to be low.

We now describe the basic update algorithm for the HS^3-G-tree. When a DO o_i updates its value from p_i to p_i', it first obtains a new secret share and then generates a seal S_i' for p_i'. On the IS side, upon receiving an update from the DO, it first finds the leaf node N where p_i is located and removes the entry p_i from it. Then, the IS locates the leaf node N' for p_i' and inserts p_i' into that node as a new entry. Afterwards, the update is propagated to the parents $parent(N)$ and $parent(N')$. The IS removes the S_i component from the folded seal of $parent(N)$ by applying $\otimes S_i^{-1}$, the inverse of S_i with respect to operator $\otimes$, and adds the S_i' component to the folded seal of $parent(N')$ by applying $\otimes S_i'$. This propagation continues until the seals of all affected nodes are updated. In Figure 8(a), where o_6 moves from p_6 to p_6', the affected nodes are '10', '1000', and '1001'. Consequently, the IS removes the entry p_6 from '1000', inserts a new entry p_6' into '1001', and updates the following folded seals: $S_{10} = S_{10} \otimes S_6^{-1} \otimes S_6'$, $S_{1000} = S_{1000} \otimes S_6^{-1}$, $S_{1001} = S_{1001} \otimes S_6'$.

This update algorithm also applies to the HS^3-R-tree, except that an update may cause the R^*-tree to reconstruct. The DOs whose values fall in any of the reconstructed nodes need to send the new seals to the IS after the reconstruction.

We analyze the security of the update scheme as follows. While the soundness and completeness of query results can be easily derived from the static data case, we focus on the freshness guarantee of dynamic data. If an adversary can compromise the freshness at time τ, it must find a subset of seals received prior to τ whose secret shares sum up to the total secret at τ. Since each secret share is indistinguishable from a random value, the adversary can do no better than a random selection. As the probability of choosing two subsets whose sums are equal is approximately $\frac{1}{2^{|ss|}}$, where $|ss|$ is the length of a secret share and is set to at least 2048 bits in practice, this probability is negligible.

Note that in the basic update algorithm above, the IS responds to an update immediately, which may incur a lot of handling overhead even if many DO updates do not affect any query results. In the following, we propose lazy update strategies for both the HS^3-G-tree and HS^3-R-tree. The main idea is to defer actual seal updates as late as possible.

7.2 Lazy Update in HS^3-G-tree

We add two hash tables $\mathcal{L}$,$\mathcal{U}$ at each node: $\mathcal{L}$ keeps track of all seals with their latest values (for the inverse operator $\otimes^{-1}$); $\mathcal{U}$ keeps track of the seals that are yet to be updated. With the hash tables, the update of o_6 from p_6 to p_6' in Figure 8(b) can be buffered in $\mathcal{U}$ of node '10', until a query Q arrives. At that time, since node '10' is sufficient to prove p_8 is outside Q, only the seal S_{10} needs to be updated as $S_{10} \cdot S_6^{-1} \cdot S_6'$. This update evicts S_6' from $\mathcal{U}_{10}$ (and thus it is empty) and inserts S_6' to $\mathcal{L}_{10}$.

As another example, o_8 updates from p_8 to p_8' then to p_8'' in Figure 8. During this process, only the hash table of $\mathcal{U}_{11}$ needs to be updated twice. Then, when a query Q arrives, only the seal S_{1111} is updated when Q descends to the leaf level; the seals S_{11} and S_{1101} do not need to be updated at all as they are not required for the authentication of Q.

7.3 Loosely-Bounded HS^3-R-tree

The challenge of applying the above lazy update to the R^*-tree is that buffering a large set of updates in a node may cause to reconstruct when a query arrives, which requires all affected DOs to update their seals. This will incur a lot of time overhead in query response. To address this issue, we propose to build the R^*-tree in such a way that a data value is allowed to update within a certain range without leaving the leaf node. The result is a loosely-bounded R^*-tree.

The loosely-bounded R^*-tree stores a bounding region of each data value, so that any update within this region does not cause any reconstruction. The main challenge, however, is a proper setting of the region size — a larger region increases the probability of keeping future updates within this region, but at the cost of a higher chance of being unfolded (and thus needs to be verified) when a query comes.

To derive the optimal bounding-region size, we develop an analytical model for the overall CPU cost C attributed to a DO update. To estimate the new value of the DO, we assume it follows a Gaussian distribution centered at the current value with a variance of σ^2. We further simplify the region as a square and thus only consider a single dimension. For simplicity, we do not consider any node reconstruction caused by DO updates. As such, the CPU cost attributed to a DO update include new seal generations, index updates, and query verifications, denoted by C_{gen}, $C_{\otimes}$, and C_{verify}, respectively, as in Section 6.1:

$$C = C_{gen} \cdot num_u + w \cdot C_{\otimes} \cdot num_u^* + C_{verify} \cdot num_q^*, \quad (6)$$

where num_u is the total number of DO updates, num_u^* is the number of updates on the index, num_q^* is the number of query verifications, and w is the height of the index tree. Let num_q be the total number of queries within this period. We can approximate num_u^* as:

$$\begin{aligned} num_u^* &\approx num_u \cdot 2 \int_L^{+\infty} \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{x^2}{2\sigma^2}} dx \\ &= num_u (1 - erf(\frac{L}{\sqrt{2\sigma^2}}), \end{aligned} \quad (7)$$

where L is the half-side length of the bounding square. We further estimate num_q^* according to Eq. (2) as:

$$num_q^* \approx num_q \cdot (2L + Q.L), \quad (8)$$

where $Q.L$ is the average query length. Putting Eqs. (7) and (8) back to Eq. (6), taking the derivative of L on the left side, and making it zero, we can obtain that C reaches the minimum when L satisfies the following condition:

$$w \cdot C_{\otimes} \cdot num_u \frac{\sqrt{2}}{\sqrt{\pi\sigma^2}} e^{\frac{-L^2}{2\sigma^2}} = 2C_{verify} \cdot num_q.$$

Solving this equation, we have

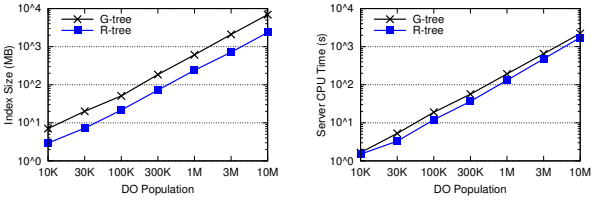
$$L = \sqrt{-2\sigma^2 \ln \frac{\sqrt{2\pi\sigma^2} \cdot C_{verify} \cdot num_q}{w \cdot C_{\otimes} \cdot num_u}}. \quad (9)$$

8. PERFORMANCE EVALUATION

In this section, we evaluate the performance of the proposed HS^3-G-tree and HS^3-R-tree on two real-life datasets, Gowalla [11] and Weather [1]. The Gowalla dataset from the Standford Large Network Dataset Collection contains $6,442,890$ user check-ins at $1,280,969$ unique locations [11]. These locations are simulated as the spatial objects in the experiments. Besides the location, each object is associated with a non-spatial value, which is randomly generated according to a normal distribution with a mean of 100 and a standard deviation of 30. The objects in this dataset are in the form of <*latitude*, *longitude*, *value*>. The Weather dataset is from the National Weather Service

Parameter	Symbol	Value/Range
# DOs	n	$1,280,969$
Hashtable size	$\|\mathcal{U}\|$	$[50, 250]$
# queries	$\|Q\|$	$[10, 250]$
Range query cube size	q	$[0.5 \times 10^{-3}, 512 \times 10^{-3}]$
R^*-tree node capacity	u	10
Update rate	γ	$[1000, 5000]$

Table 1: Parameter Settings for Experiments



(a) Index Size (b) Index Construction Time

Figure 9: Server Construction Cost vs. DO Population

(NWS) Cooperative Observer Program (COOP) [1], where more than 10,000 volunteers report daily weather observations, in the form of <*latitude*, *longitude*, *temperature*, *precipitation*, *snowfall*, *snowdepth*>, in urban and suburban areas. While we use Gowalla as the default dataset, Weather is used to evaluate the performance when the data dimensionality is varied. For convenience, all attribute values are converted to integers.

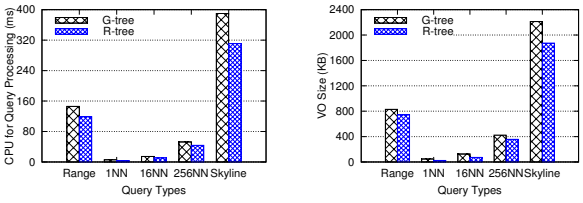
Both the DOs and user side are set up on a laptop computer, with an entry-level Intel Core i3 CPU and 4GB RAM, running Windows XP x64 SP3, while the IS is set up on an HP Proliant DL360 server with a Dual 6-core Intel Xeon X5650 2.66GHz CPU and 32GB RAM, running GNU/Linux. The system is written in Java and runs on 64-bit OpenJDK 1.6. The hash function $h(\cdot)$ used is 160-bit SHA-1, and the pseudorandom function $f_k(\cdot)$ adopts AES-256. As for $\mathcal{G}(\cdot, \cdot)$, RSA-2048 is used as the underlying cryotosystem.

For the homomorphic secret sharing seal, we set the number of prefixes in a seal to 6. Furthermore, we set the node capacity of the R^*-tree to 10 so that the depths of both HS^3-G-tree and HS^3-R-tree are 6. In the experiments, we evaluate range queries, k-nearest-neighbor (kNN) queries, and skyline queries. For range queries, since Gowalla is a 3-dimensional dataset, we vary the cube size of the query range from 0.5×10^{-3} to 512×10^{-3} of the whole space. The query workload is controlled by $|Q|$, the number of queries issued during each epoch; while the update workload is controlled by γ, the number of DOs updating their new values to the server during each epoch. The new value is randomly generated within $\pm 10\%$ of the current value. We adopt $\gamma = 3,000$ as the default setting. Table 1 summaries the settings for the default Gowalla dataset used in the experiments.

For performance evaluation, we measure two authentication costs: (i) the computational cost in terms of the server and user CPU time, and (ii) the communication cost in terms of the size of data transmitted between the DO and IS & between the IS and user. For brevity of presentation, we denote the two HS^3 schemes as G-tree and R-tree, and the lazy-update G-tree and loosely-bounded R-tree as Lazy-G-tree and Loose-R-tree, respectively.

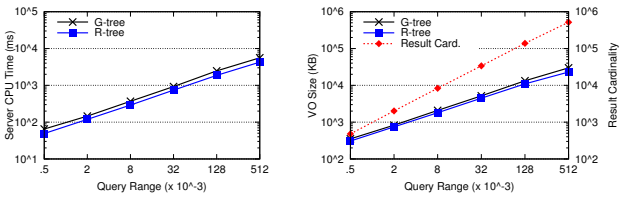
8.1 Authenticated Data Structure Construction

We measure (i) the seal generation cost (on the DO side), (ii) the seal size (i.e., the communication cost between the DO and IS), (iii) the HS^3 index size, and (iv) the HS^3 index construction time. For (i), the CPU time is 120.1ms and 124.3ms for G-tree and R-tree, respectively; for (ii), the seal size is 256 bytes for both schemes. As for the latter two



(a) Server CPU Time (b) VO Size

Figure 10: Basic Query Authentication Performance



(a) Server CPU Time (b) VO Size

Figure 11: Performance of Varying Query Ranges

metrics, since they depend on the DO population n, we vary n from $10,000$ to $10,000,000$ by randomly drawing n objects (possibly with updated objects) from the Gowalla dataset. Figure 9 plots the results under different n settings. Both metrics increase proportionally to n and are less than 6.9GB and 36 mins in the worst case. As our HS^3 schemes are highly parallelizable, we expect the server CPU cost can be further reduced with cloud or GPU acceleration.

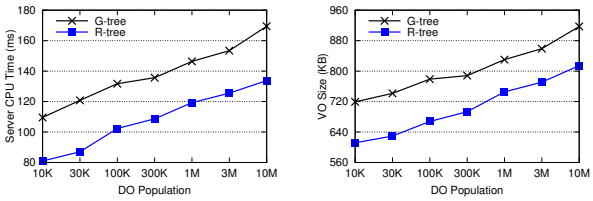
8.2 Query Authentication Performance

In this experiment, we evaluate the query authentication performance of the two HS^3 schemes. We test various queries including range query ($q = 2 \times 10^{-3}$), kNN (k is set to 1, 16, 256), and skyline query under the default Gowalla dataset. The results of server CPU time and VO size are plotted in Figure 10. The user CPU performance is omitted since it is related to the VO size and is below 430ms for all queries tested. We also evaluate the naive scheme mentioned in Section 4 that sends the whole dataset to the user for result verfication. However, its VO size and user CPU time are as large as 450MB and 10 hours (not shown in the figure for presentation clarity), which are more than three orders of magnitude worse than our proposed schemes. It is observed from Figure 10 that R-tree outperforms G-tree in all cases tested. This is consistent with our cost analysis in Section 6 that R-tree is more compact than G-tree.

Next, we focus on the range query and increase the query range from 0.5×10^{-3} to 512×10^{-3} of the whole space. We show the server CPU time, the VO size, and the cardinality of query results in Figure 11. As can be seen, even when the query range is as large as 512×10^{-3}, the server CPU time is still below 5.6 sec and the VO size is still below 29.7MB. This is a significant reduction from sending the whole dataset. While the cardinality of query results increases linearly with the query range, it is interesting to observe that the VO size in both G-tree and R-tree enlarges at a slower rate. This validates that the overhead of authenticating non-result values is not much affected by the query range.

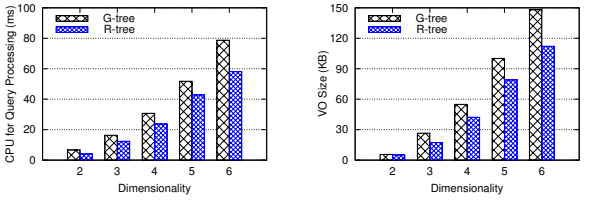
We then increase the DO population n from $10,000$ to $10,000,000$ while keeping the same result size as above when $q = 2 \times 10^{-3}$. As shown in Figure 12, both the server CPU time and VO size are only slightly increased with DO population and are less than 180ms and 960KB in the worst case. This demonstrates the scalability of our proposed schemes.

We further investigate the impact of dimensionality on the authentication performance using the Weather dataset. Fig-



(a) Server CPU Time (b) VO Size

Figure 12: Performance vs. Data Population



(a) Server CPU Time (b) VO Size

Figure 13: Performance vs. Data Dimensionality

ure 13 shows the costs incurred for range queries when the dimensionality is varied from 2 to 6. Both schemes become worse with increasing dimensionality, while G-tree deteriorates faster than R-tree. This indicates that the clustering property of R-tree is particularly important when the dimensionality is high.

8.3 Data Update Performance

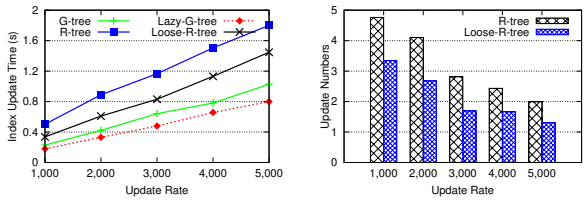
In the following, we first evaluate the basic update scheme for the two HS^3 indexes and then show the performance after applying the optimizations introduced in Section 7.

In the basic scheme, since G-tree is a space-partitioned index, its updating is expected to be more efficient than R-tree, a data-partitioned index. Figure 14(a) verifies our expectation, where the former consistently outperforms the latter for any update rate γ. We also plot the actual number of seal updates on the R-tree per DO update in Figure 14(b). It decreases as the update rate increases, because the seal updates incurred by node reconstructions can be amortized by more frequent DO updates. Next, we analyze the performance of the proposed optimizations, i.e., lazy-update G-tree and loosely-bounded R-tree. Figure 14(a) shows that on average lazy-G-tree is 20% more efficient than G-tree and loose-R-tree is 30% more efficient than R-tree.

We further evaluate the two optimizations under various parameter settings. First, we study how the query workload $|Q|$ will affect the server's CPU time used by lazy-G-tree for index updating and query processing. We fix the hash table size $|\mathcal{U}|$ at 150 and vary $|Q|$ from 10 to 250. Figure 15 shows that as $|Q|$ increases, the CPU time decreases because the update cost is amortized over more queries.

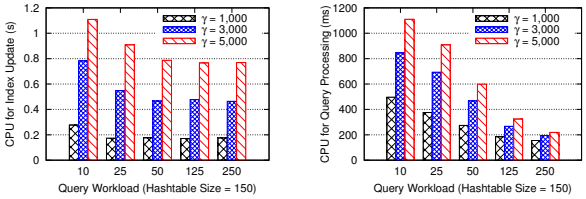
We also study the impact of L, the half-side length of the bounding square, on the performance of loose-R-tree. We set the query workload $|Q|$ at 50 and the update rate γ at $3,000$, and obtain the theoretical optimal L^* from Eq. (9). Then we vary L from $0.25L^*$ to $4L^*$ and plot in Figure 16 the total CPU time attributed to each DO for new seal generation, index updating, and query authentication. We observe that this real optimal value L' is close to the theoretical L^*. We also plot the number of seal updates of loose-R-tree with L^* in Figure 14(b), where the cost is significantly reduced by up to 35% from the original R-tree.

Finally, we compare the performance of G-tree and R-tree on mixed workloads. We mix p% of range queries with



(a) Index Update Time (b) # HS^3-R-tree Updates

Figure 14: Index Update Performance



(a) CPU for Index Updating (b) CPU for Query Processing

Figure 15: Impact of Query Workload on Server

$(100 - p)$% of data updates as the transactional workload and turn on all the optimizations of both trees. The server CPU time is plotted in Figure 17. We observe that when the updates dominate (i.e., $p < 40$), G-tree is the winner since it is a space-partitioned index and is therefore more efficient in updating. In contrast, when the queries dominate (i.e., $p > 60$), R-tree is the winner since it is more compact than G-tree and hence performs better in query processing.

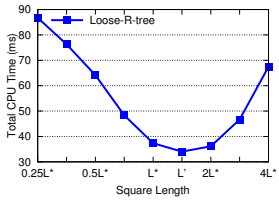


Figure 16: Impact of Bounding-Square Size

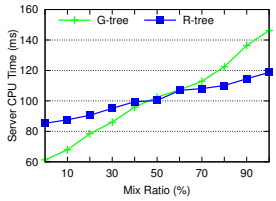


Figure 17: Performance under Mixed Workload

9. CONCLUSION

In this paper, we have studied the problem of integrity assurance and query result authentication for online data integration services. We developed a novel distributed authentication code named HS^3 that can aggregate the input from individual sources faithfully even by an untrusted server. Based on this, we designed two authenticated data structures, namely HS^3-G-tree and HS^3-R-tree, and proposed authentication schemes for various queries on multi-dimensional data. We also developed several advanced update strategies to address the freshness problem in multi-source query authentication. Analytical models and empirical results show our HS^3 code and schemes are efficient and robust under various system settings.

As for future work, we plan to extend the proposed authentication schemes to more complex query types, especially those involve distance computation such as distance-based joins. As there are many applications that adopt distance metrics other than the Euclidean distance, a more general version of secret sharing seal in the metric space is yet to be designed.

Acknowledgments

We would like to thank all reviewers for their valuable suggestions. This work was supported by RGC/GRF Grants 12202414, 12200114, 12200914, 210612 and FRG2/13-14/064.

10. REFERENCES

[1] Cooperative observer network (coop). http://www.ncdc.noaa.gov/data-access/land-based-station-data/land-based-datasets/cooperative-observer-network-coop.

[2] M. Arenas, L. E. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. In *Proc. PODS*, 1999.

[3] B. Babcock, S. Chaudhuri, and G. Das. Dynamic sample selection for approximate query processing. In *Proc. SIGMOD*, pages 539–550, 2003.

[4] M. Bellare, J. A. Garay, and T. Rabin. Fast batch verification for modular exponentiation and digital signatures. In *EUROCRYPT*, pages 236–250, 1998.

[5] A. CalÃň, D. Calvanese, G. D. Giacomo, and M. Lenzerini. Data integration under integrity constraints. *Inf. Syst.*, 29(2):147–163, 2004.

[6] H. Chan, H.-C. Hsiao, A. Perrig, and D. Song. Secure distributed data aggregation. *Found. Trends databases*, 3(3):149–201, 2011.

[7] H. Chan, A. Perrig, B. Przydatek, and D. Song. SIA: Secure information aggregation in sensor networks. *J. Comput. Secur.*, 15(1):69–102, 2007.

[8] H. Chan, A. Perrig, and D. Song. Secure hierarchical in-network aggregation in sensor networks. In *Proc. CCS*, pages 278–287, 2006.

[9] Q. Chen, H. Hu, and J. Xu. Authenticating top-k queries in location-based services with confidentiality. In *Proc. VLDB*, pages 49–60, 2014.

[10] W. Cheng and K. Tan. Query assurance verification for outsourced multi-dimensional databases. *Journal of Computer Security*, 17(1), 2009.

[11] E. Cho, S. A. Myers, and J. Leskovec. Friendship and mobility: User movement in location-based social networks. In *Proc. SIGKDD*, pages 1082–1090, 2011.

[12] S. Choi, H.-S. Lim, and E. Bertino. Authenticated top-k aggregation in distributed and outsourced databases. In *Proc. SOCIALCOM-PASSAT*, pages 779–788, 2012.

[13] J. Considine, F. Li, G. Kollios, and J. Byers. Approximate aggregation techniques for sensor databases. In *Proc. ICDE*, pages 449 –460, 2004.

[14] G. Cormode and M. Hadjieleftheriou. Finding frequent items in data streams. In *Proc. VLDB*, pages 1530–1541, 2008.

[15] Z. Fan, Y. Peng, B. Choi, J. Xu, and S. Bhowmick. Towards efficient authenticated subgraph query service in outsourced graph databases. *TSC*, 7(4):696–713, Oct 2014.

[16] O. Hassanzadeh, K. Q. Pu, S. H. Yeganeh, R. J. Miller, L. Popa, M. A. Hernández, and H. Ho. Discovering linkage points over web data. In *Proc. VLDB*, 2013.

[17] H. Hu, J. Xu, Q. Chen, and Z. Yang. Authenticating location-based services without compromising location privacy. In *Proc. SIGMOD*, pages 301–312, 2012.

[18] L. Hu, W.-S. Ku, S. Bakiras, and C. Shahabi. Spatial query integrity with voronoi neighbors. *IEEE TKDE*, 25(4):863–876, 2013.

[19] R. Jain and S. Prabhakar. Trustworthy data from untrusted databases. In *Proc. ICDE*, 2013.

[20] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Authenticated index structures for aggregation queries. *ACM TISSEC*, 13(32):1–35, 2010.

[21] F. Li, G. Kollios, and L. Reyzin. Dynamic authenticated index structures for outsourced databases. In *Proc. SIGMOD*, pages 121–132, 2006.

[22] Q. Li, Y. Li, J. Gao, B. Zhao, W. Fan, and J. Han. Resolving conflicts in heterogeneous data by truth discovery and source reliability estimation. In *Proc. SIGMOD*, 2014.

[23] X. Li, X. L. Dong, K. Lyons, W. Meng, and D. Srivastava. Truth finding on the deep web: is the problem solved? In *Proc. VLDB*, pages 97–108, 2013.

[24] X. Lin, J. Xu, H. Hu, and W.-C. Lee. Authenticating location-based skyline queries in arbitrary subspaces. *TKDE*, 26(6):1479–1493, June 2014.

[25] S. Liu, S. Wang, F. Zhu, J. Zhang, and R. Krishnan. Hydra: large-scale social identity linkage via heterogeneous behavior modeling. In *Proc. SIGMOD*, 2014.

[26] X. Liu, X. L. Dong, B. C. Ooi, and D. Srivastava. Online data fusion. In *Proc. VLDB*, pages 932–943, 2011.

[27] A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot. *Handbook of Applied Cryptography*. 1996.

[28] R. C. Merkle. A certified digital signature. In *Proc. Crypto*, pages 218–238, 1989.

[29] S. Nath and R. Venkatesan. Publicly verifiable grouped aggregation queries on outsourced data streams. In *Proc. ICDE*, pages 517–528, April 2013.

[30] S. Nath, H. Yu, and H. Chan. Secure outsourced aggregation via one-way chains. In *Proc. SIGMOD*, pages 31–44, 2009.

[31] B.-U. Pagel, H.-W. Six, H. Toben, and P. Widmayer. Towards an analysis of range query performance in spatial data structures. In *Proc. PODS*, pages 214–221, 1993.

[32] A. Pal, V. Rastogi, A. Machanavajjhala, and P. Bohannon. Information integration over time in unreliable and uncertain environments. In *Proc. WWW*, 2012.

[33] H. Pang, A. Jain, K. Ramamritham, and K.-L. Tan. Verifying completeness of relational query results in data publishing. In *SIGMOD*, pages 407–418, 2005.

[34] H. Pang and K.-L. Tan. Authenticating query results in edge computing. In *Proc. ICDE*, 2004.

[35] S. Papadopoulos, G. Cormode, A. Deligiannakis, and M. Garofalakis. Lightweight authentication of linear algebraic queries on data streams. In *Proc. SIGMOD*, pages 881–892. ACM, 2013.

[36] S. Papadopoulos, A. Kiayias, and D. Papadias. Exact in-network aggregation with integrity and confidentiality. *IEEE TKDE*, 24(10):1760–1773, 2012.

[37] C. Papamanthou, E. Shi, R. Tamassia, and K. Yi. Streaming authenticated data structures. In *Proc. EUROCRYPT*, volume 7881, pages 353–370. 2013.

[38] Y. Peng, Z. Fan, B. Choi, J. Xu, and S. Bhowmick. Authenticated subgraph similarity search in outsourced graph databases. *TKDE*, accepted to appear.

[39] T. Rekatsinas, X. L. Dong, and D. Srivastava. Characterizing and selecting fresh data sources. In *Proc. SIGMOD*, 2014.
[40] Y. Theodoridis and T. Sellis. A model for the prediction of r-tree performance. In *Proc. PODS*, pages 161–171, 1996.
[41] D. Wu, B. Choi, J. Xu, and C. Jensen. Authentication of moving top-k spatial keyword queries. *TKDE*, 27(4):922–935, April 2015.
[42] Y. Yang, S. Papadopoulos, D. Papadias, and G. Kollios. Spatial outsoucing for location-based services. In *Proc. ICDE*, pages 1082–1091, 2008.
[43] Y. Yang, S. Papadopoulos, D. Papadias, and G. Kollios. Authenticated indexing for outsourced spatial databases. *VLDBJ*, 18(3):631–648, 2009.
[44] Y. Yang, X. Wang, S. Zhu, and G. Cao. SDAP: A secure hop-by-hop data aggregation protocol for sensor networks. In *Proc. MobiHoc*, pages 356–367, 2006.
[45] X. Yin, J. Han, and P. S. Yu. Truth discovery with multiple conflicting information providers on the web. In *Proc. SIGKDD*, 2007.
[46] M. L. Yiu, Y. Lin, and K. Mouratidis. Effcient verification of shortest path search via authenticated hints. In *Proc. ICDE*, pages 237–248, 2010.
[47] M. L. Yiu, E. Lo, and D. Yung. Authentication of moving knn queries. In *ICDE*, pages 565–576, 2011.
[48] R. Zhang, J. Shi, Y. Liu, and Y. Zhang. Verifiable fine-grained top-k queries in tiered sensor networks. In *Proc. INFOCOM*, pages 1199–1207, 2010.
[49] R. Zhang, J. Shi, Y. Zhang, and C. Zhang. Verifiable privacy-preserving aggregation in people-centric urban sensing systems. *IEEE JSAC*, 31(9):268–278, 2013.
[50] B. Zhao, B. I. P. Rubinstein, J. Gemmell, and J. Han. A bayesian approach to discovering truth from conflicting sources for data integration. In *Proc. VLDB*, pages 550–561, 2012.

APPENDIX

A. PROOF OF LEMMA 1

We prove Lemma 1 by contradiction. If there were a PPT adversary $\mathcal{A}$ who can forge the seal $seal(sk, v)$ for a value v, we can design an algorithm $\mathcal{A}'$ for the RSA problem. Consider an instance of the RSA problem that, given a public key (e, n) and an element $y \in \mathbb{Z}_n^*$, attempts to find x such that $x^e = y \mod z$. In $\mathcal{A}'$, we design a random oracle as follows. When $\mathcal{A}$ issues p random oracle queries with $v_1, \ldots, v_p$, $\mathcal{A}'$ responds with $r_1 \cdot g^{h(v_1)}, \cdots, r_{i-1} \cdot g^{h(v_{i-1})}, y, r_{i+1} \cdot g^{h(v_{i+1})}, \cdots, r_p \cdot g^{h(v_p)}$, i.e., for the query v_i it responds with y in the given instance of the RSA problem, while for other queries it responds with $r \cdot g^{h(\cdot)}$, where p is polynomially upper-bounded. If $\mathcal{A}'$ is requested by $\mathcal{A}$ to return the seal of some value in $\{v_1, \ldots, v_p\} - \{v_i\}$, it responds accordingly. When $\mathcal{A}$ forges and outputs the seal $seal(sk, v)$ of some randomly chosen v in $\{v_1, \ldots, v_p\}$, $\mathcal{A}'$ can use $seal(sk, v)$ to solve the given instance of the RSA problem with a probability of $1/p$, which is the probability that v is chosen as v_i. In that case, $seal(sk_i, v_i)^e = y \mod z$; thus $\mathcal{A}'$ finds $x = seal(sk_i, v_i)$. It contradicts the assumption that there is no PPT algorithm for the RSA problem.

B. PROOF OF LEMMA 2

With the seal in Lemma 1, the batch verification function can be defined: $batch\_verify(\{v_1, \cdots, v_t\}, \{S_1, r_1, \cdots, S_t, r_t\})$. It returns 1 if $\mathcal{G}^{-1}(\otimes_{i=1}^{t} S_i) = \prod_{i=1}^{t} r_i \cdot g^{h(v_i)}$, otherwise 0. Similar to Theorem 4.1 in [4], the batch verification is secure when $\text{RSA}_{(e,n)}$ is a one-way function.

More specifically, since Lemma 1 guarantees the unforgery of the seal of a new value, the only practical attack is to forge a seal based on the existing seals. Suppose an adversary succeeds to make a forgery of (v_t', r_t', S_t') based on two existing seals for v_i, v_j by $S_t' = S_i \otimes S_j$. However, it requires to satisfy the equation $r_{t'} \cdot g^{h(v_t')} = r_i \cdot g^{h(v_i)} \cdot r_j \cdot g^{h(v_j)}$, which is impossible if $r \cdot g^{h(\cdot)}$ can be modeled as a random oracle.

C. PROOF OF LEMMA 3

Different from the batch verification, the principle of seal folding is to aggregate the seals before sending them to the verifier, i.e., the seals in $\{S_1, \cdots, S_t\}$ will be folded as $S = \otimes_{i=1}^{t} S_i$. When the verifier receives $\{v_1, \cdots, v_t\}$, the folded seal S, and the product of the random values $\prod_{i=1}^{n} r_i$, it verifies the seal by checking $\mathcal{G}^{-1}(S) = \prod_{i=1}^{t}(r_i \cdot g^{h(v_i)})$.

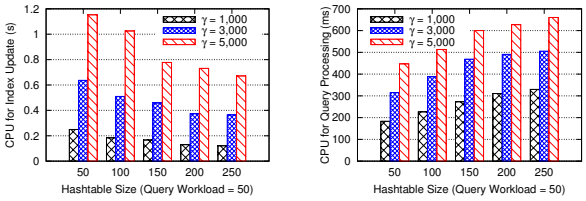
We prove Lemma 3 by contradiction. Suppose there exists a PPT eavesdropper adversary that observes a set of messages $\{v_1, \cdots, v_t\}$ and a folded seal S, it successfully makes a forgery (v_t', r_t', S') such that $\mathcal{G}^{-1}(S') = (\prod_{i=1}^{t-1} r_i \cdot g^{h(v_i)}) \cdot (r_t' \cdot g^{h(v_t')})$. Now let us consider the batch verification, where the adversary is required to return individual seals. The seals for the first $t-1$ values are the genuine ones, and the forged one can be computed as $S_t' = S' \otimes (\otimes_{i=1}^{t-1} S_i)^{-1}$, which can pass the batch verification and make a forgery of (v_t', r_t', S_t'). It breaks the security of the batch verification, which causes a contradiction to Lemma 2.

D. SECURITY ANALYSIS OF HS^3-G-TREE

We analyze the security of HS^3-G-tree from two aspects:

- Soundness. The basic security guarantee follows that of HS^3. The only difference here is that if an internal node is fully covered by the query, we do not need to return the individual seals of the points under this node (e.g., S_8 in Figure 5). Nevertheless, the authenticity of these result points can be guaranteed by the folded seal in the internal node since it is secure against forgeries according to Lemmas 1 and 3.
- Completeness. The completeness guarantee follows the secret sharing scheme.

E. ADDITIONAL EXPERIMENT ON THE IMPACT OF HASHTABLE SIZE



(a) CPU for Index Updating (b) CPU for Query Processing

Figure 18: Impact of Hashtable Size on Server

This experiment analyzes how the size of hashtable $\mathcal{U}$ will affect the server's CPU time used by lazy-G-tree for index updating and query processing. We fix $|Q|$ at 150 and vary $|\mathcal{U}|$ from 50 to 250. Figure 18(a) shows that a larger hashtable size reduces the index updating time, because it is more likely that a new DO update only replaces a buffered item in the hashtable. On the other hand, Figure 18(b) shows that a larger hashtable size increases the server's query processing time, because the server needs to take more CPU time in examining the hashtables of accessed nodes.