Integrity for Join Queries in the Cloud

Sabrina De Capitani di Vimercati, *Senior Member*, *IEEE*, Sara Foresti, *Member*, *IEEE*, Sushil Jajodia, *Fellow*, *IEEE*, Stefano Paraboschi, *Member*, *IEEE*, and Pierangela Samarati, *Fellow*, *IEEE*

Abstract—We address the problem of providing users with the ability to assess the integrity of join results produced by external computational providers and computed over externally stored databases. Our approach relies on different mutually supporting techniques offering strong integrity protection guarantees at a limited cost. The application of the approach is completely transparent to the computational provider, against which data and query confidentiality are preserved. The paper introduces our techniques analytically, examining their protection guarantees and performance. It also illustrates experimental results, which confirm the effectiveness and efficiency of our solutions.

Index Terms—Cloud, query integrity, correctness, and completeness, twins, markers, salts and buckets

1 Introduction

Cloud technology has become the reference paradigm for the realization of large-scale storage and computational services. The significant economies of scale that cloud providers can enjoy in the procurement, configuration and administration of storage, networking, and computing infrastructures, allow them to offer the services at a price that is normally a fraction of the cost that users would otherwise face for building in-house the same level of availability, scalability, and elasticity.

However, although extremely appealing in terms of functionality and economic advantages, cloud technology can see effective and widespread exploitation only if users have guarantees on the correct and proper working of the external services. The problem of providing integrity and confidentiality guarantees in distributed settings has been receiving considerable attention in recent years. Most solutions have addressed the protection of data confidentiality, typically against honest-but-curious servers and assuming a single cloud provider [1], [2]. Work addressing integrity protection has mainly considered the problem of detecting possible misbehavior of the server storing the data, often assuming prior knowledge of the user on the data externally stored [3], [4], [5], [6].

Also, while research has focused on solutions assuming a single provider of cloud services, the market shows today a clear evolution toward the creation of a varied ecosystem, with the availability of a multitude of services managed by independent providers allowing selective adoption of functional abilities (e.g., Amazon Web Services differentiates between Amazon S3 dedicated to storage and Amazon EC2 dedicated to computation). This evolution enables distinguishing between: providers of *storage services*, which offer continuous availability of stored data, with high bandwidth and reliability guarantees; and providers of *computational services*, which offer access to inexpensive virtualized machines, with high elasticity and support for efficient execution of computationally intensive services. This separation and the increased use of cloud technology can enable the development of applications that integrate data and functions hosted by different service providers.

In this context, we consider a reference scenario integrating relational database technology with novel cloud infrastructures and address the execution of join queries delegated to potentially unreliable computational services, providing integrity guarantees on the result computed by the computational server, as well as ensuring data confidentiality. Our techniques rely on the cooperation of the user issuing the query and of the storage servers storing the data, which are assumed to be trusted, without requiring the user to have any knowledge of the data externally stored.

Providing protection guarantees, we enable users and companies to enjoy the benefit of the cloud marked and dynamically choose among available services the one that is less expensive for running their queries over external data. Our approach can also find application in a hybrid cloud scenario [7], enabling the cost benefits of public cloud providers and the level of trust that can be obtained from a private infrastructure.

This work is in line with recent analyses that emphasize how query optimization in cloud scenarios should move from an analysis of computational costs to an analysis of economic costs [8]. Detaching computation from storage, while ensuring security and privacy, our solution supports the realization of a flexible market in cloud computation services, enabling users to acquire services from different providers with a significant economic gain.

- *S. De Capitani di Vimercati, S. Foresti, and P. Samarati are with the Università degli Studi di Milano, Italy. E-mail: {Sabrina.DeCapitanidiVimercati, Sara.Foresti, Pierangela.Samarati}@unimi.it.*
- *S. Jajodia is with CSIS, George Mason University. E-mail: jajodia@gmu.edu.*
- *S. Paraboschi is with the Dipartimento di Ingegneria, Via Marconi 5, Dalmine, BG, 24044, Italy. E-mail: parabosc@unibg.it.*

Manuscript received 4 July 2013; revised 29 Oct. 2013; accepted 24 Nov. 2013; published online 10 Dec. 2013.
Recommended for acceptance by V. Varadharajan.
For information on obtaining reprints of this article, please send e-mail to: tcc@computer.org, and reference IEEECS Log Number TCC-2013-07-0125.
Digital Object Identifier no. 10.1109/TCC.2013.18.

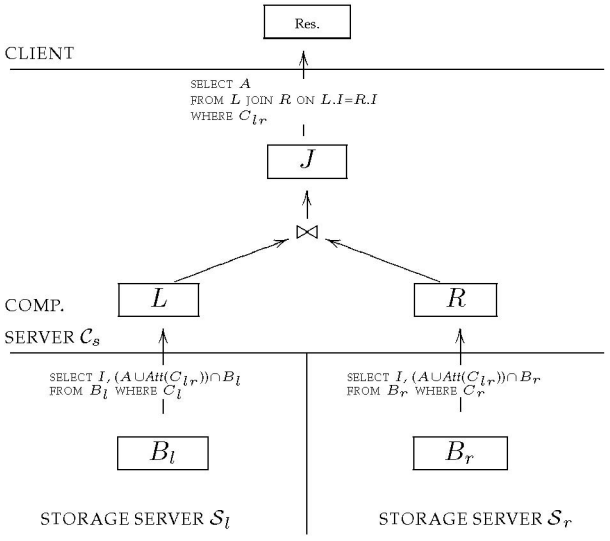


Fig. 1. Query evaluation process.

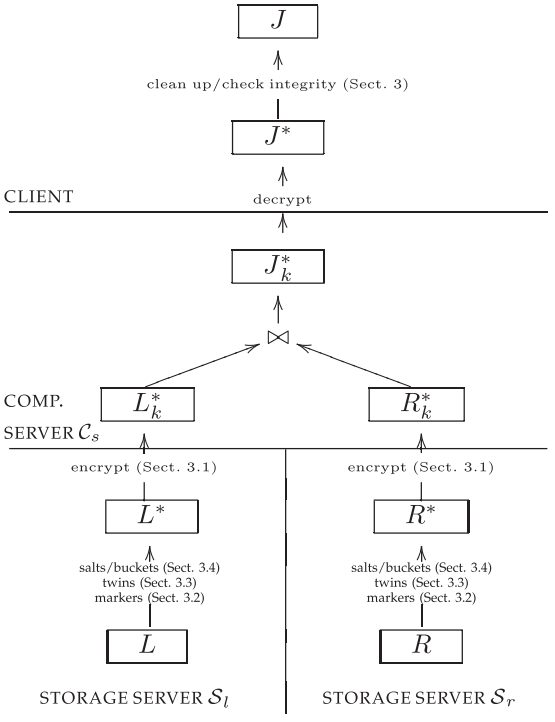


Fig. 2. Join execution with our techniques.

The remainder of the paper is organized as follows. Section 2 illustrates an overview of our approach. Section 3 describes the working of our protection techniques. Section 4 presents an analysis of the guarantees of integrity of query results ensured by our techniques. Section 5 evaluates the performance impact of our techniques. Section 6 shows experimental results. Section 7 discusses related work. Section 8 presents our conclusions.

2 Overview of Our Approach

We consider a cloud scenario where *storage* and *computational* services are with different providers. Clients execute queries over databases kept on storage servers by means of computational servers. For concreteness, in the paper, we assume two storage servers $\mathcal{S}_l$ and $\mathcal{S}_r$ each storing a relation (B_l and B_r, resp.), and a computational server $\mathcal{C}_s$. For simplicity, and without loss of generality, we assume B_l and B_r to be in cleartext, that is, the storage servers are fully trusted and have visibility of the data they store. The approach can, however, be applied in contexts where the storage servers should not have visibility on the data they store (*honest-but-curious*). In this case, the relations are stored in encrypted form together with indexes that are used for query execution [1], [2].

Clients can delegate to the computational server queries of the form "SELECT A FROM B_l JOIN B_r ON $B_l.I = B_r.I$ WHERE C_l AND C_r AND C_{lr}," where A is a subset of the attributes in $B_l \cup B_r$; I is the set of join attributes; and C_l, C_r, and C_{lr} are Boolean formulas of conditions over attributes in B_l, B_r, and $B_l \cup B_r$, respectively. In absence of security considerations, the query can be executed as usual by pushing down selections and projections at the storage servers and having the computational server compute the join J of the subquery results (L and R) and evaluate condition C_{lr} on it, producing the result to be returned to the client (see Fig. 1).

Such a classical execution plan does not allow the client to assess potential misbehavior (data tampering or omissions) of the computational server. Our approach allows the client, with the cooperation of the storage servers, to assess the integrity of the join performed by the computational server. Intuitively, instead of communicating to the computational server the results L and R of their subqueries, the storage servers first inject into them additional tuples (which will be used by the client to assess the integrity of the query result), producing relations L^* and R^*, which are then encrypted (relations L_k^* and R_k^*) and sent to the computational server (see Fig. 2). The computational server performs the join of such relations, producing relation J_k^* that is returned to the client, which can then retrieve the original join J and produce the result. J_k^* is such that the client can enjoy a certain degree of assurance of the integrity of the computation performed by the computational server. Also, operating on obfuscated versions of the data, the computational server can neither know the actual data values (confidentiality is protected) nor misbehave selectively. Relations L_k^* and R_k^* are computed independently by the storage servers without the need for them to communicate with each other, but simply according to a limited amount of information provided by the client within the subquery to be executed. Such subquery is transmitted through the computational server in encrypted form, thus remaining unintelligible to the computational server.

In the next section, we illustrate the different complementary techniques (also reported in Fig. 2) that we use for producing L_k^* and R_k^* from L and R. Within L and R we distinguish the common attributes I (on which the join has to be performed) and the other attributes *Attr*, treating each of them as a unit. Therefore, for simplicity, regardless of the degree of the schema of the two relations and the number of attributes involved in the join, we refer to L and R as having schemas $(I, Attr)$. We use generic table B to refer to L and R indistinguishably. Note that, we do not consider

many-to-many joins as the use of classical database design methodologies leads to relational schemas that directly support only one-to-one and one-to-many joins. For the same reason, we focus our analysis on the evaluation of equi-join conditions, which characterize most join operations in real-life scenarios.

3 Protection Techniques

We describe our protection techniques ensuring integrity and confidentiality, and discuss how they complement each other. Encryption offers elementary protection making data unintelligible (see Section 3.1). The core of our protection relies on the use of markers (see Section 3.2) and twins (see Section 3.3) for signaling incompleteness of the query results and working in a complementary way providing effective protection in synergy. Salts and buckets (see Section 3.4) complement them for addressing the case of one-to-many joins.

3.1 Encryption on-the-Fly

The first basic protection technique we apply is *encryption on-the-fly*, which is based on a symmetric encryption schema. Encryption on-the-fly means that the storage servers encrypt the data before sending them to the computational server. In this way, the computational server only deals with encrypted values. We use the term encryption on-the-fly since symmetric encryption is performed at run time on subquery results by the storage servers and with encryption keys changing at every query.

We apply encryption at tuple level (i.e., each tuple is individually encrypted) and separately on the join attribute and on the whole tuple. The encrypted version of a relation B is a relation, denoted B_k, with two attributes: I_k contains the encryption of the join attribute and $B.Tuple_k$ contains the encryption of all attributes in B (including the join attribute). Before encryption a padding scheme is adopted to ensure that the plaintext data have the same length.

Definition 3.1 (Encrypted relation). *Let B be a relation, I be the join attribute, and k be a cryptographic key. The* encrypted *version of B is a relation over schema $B_k(I_k, B.Tuple_k)$ such that $\forall t \in B$, $\exists$ a distinct tuple $\tau \in B_k$, with $\tau[I_k] = E_k(t[I])$ and $\tau[B.Tuple_k] = E_k(t)$.*

For instance, the encrypted version of relations L^* and R^* in Fig. 3 is represented by encrypted relations L_k^* and R_k^*, respectively. (Note that while in the encrypted tables of our examples we preserve the order of tuples for readability, the storage servers shuffle tuples sending them in random order to the computational server). To simplify the notation, we use Greek letters to denote encrypted data. For instance, in Fig. 3, $E_k(a) = \alpha$, and $E_k(l_1) = \lambda_1$.

To ensure correctness of the join when operating on encrypted values, the encryption key used to encrypt the join attribute must be the same for the two storage servers. Such a key changes at every query and is communicated by the client to the storage servers together with their subqueries. By contrast, the key used for encrypting the whole tuple could be different for the two servers. This being said, for simplicity, in the following, we assume

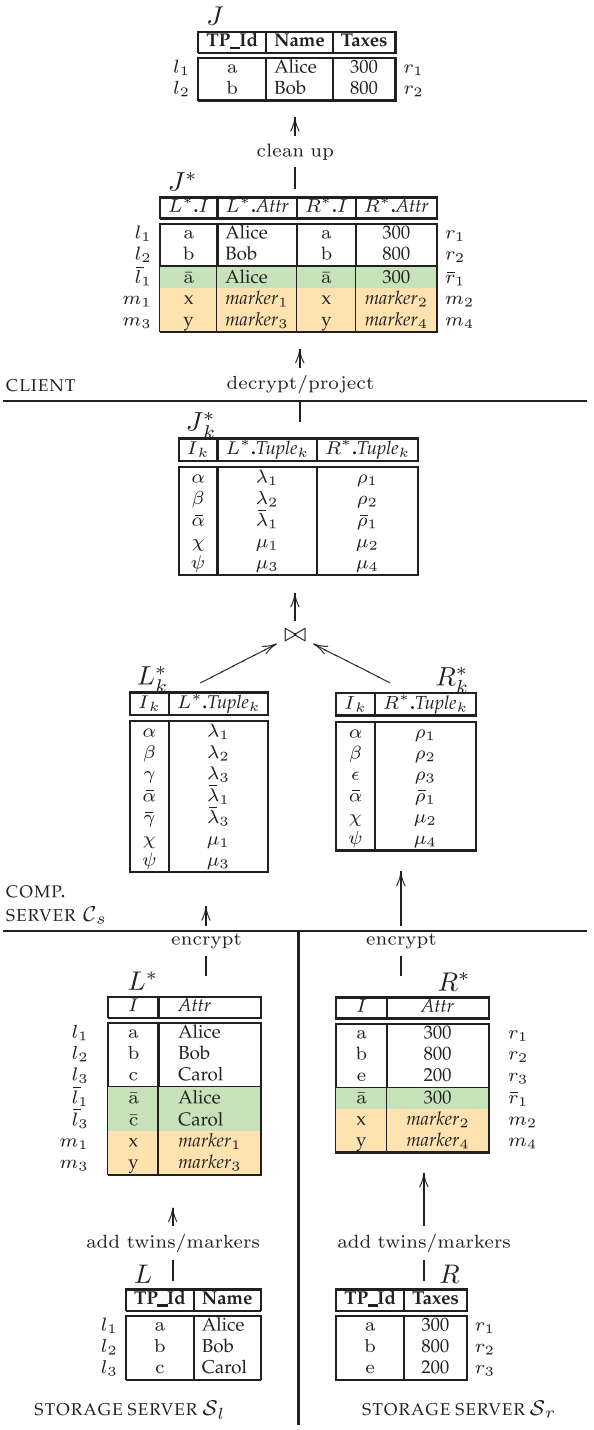


Fig. 3. An example of query evaluation process with twins on a and c and two markers.

tuples to be encrypted with the same key (the one also used for the join attribute) at the two storage servers.

Encryption protects confidentiality and offers integrity guarantee of the individual tuples in the join result (see Section 4).

3.2 Markers

Our second protection technique aims at ensuring query completeness and consists in injecting into the query computation artificial (but not recognizable as such by the

computational server) tuples, called *markers*. The absence of any of these tuples in the query result signals that the query result is not complete.

To guarantee the presence of marker tuples in the query result if the computational server behaves correctly, the storage servers insert the same number of markers in the relations obtained from their subqueries with the same distinct values for the join attribute. Since we do not want the storage servers to communicate with each other, the information about the markers to be inserted is provided directly by the client together with the subquery to be executed. Note that the client does not need to explicitly communicate the values of the join attribute for the markers that should appear in the results of the subqueries, but just ensure their presence. It is then sufficient for the client to communicate to the storage servers the number m of markers to be generated. The storage servers compute the same set of distinct values for the join attribute by simply using a progressive counter and a flag "Marker" that is set to 1. This flag is instead set to 0 for the genuine tuples, ensuring that the values of the join attribute for the markers do not collide with the ones for the genuine tuples. Markers can be formally defined as follows:

Definition 3.2 (Markers). *Let B be a relation, I be the join attribute, and D_I be the domain of I. A set $\mathcal{M}$ of* markers *for B is a set of tuples over schema $B^*(I, Attr)$, where: $\forall\langle v, rnd\rangle \in \mathcal{M}$, $v \notin D_I$; rnd is a random string; and $\forall\langle v, rnd\rangle, \langle v', rnd'\rangle \in \mathcal{M}$, $v \neq v'$.*

For instance, in Fig. 3 tables L^* and R^* include markers m_1 and m_3, and m_2 and m_4, respectively, which have been added assuming that the storage servers are requested to insert two markers for which they generate values x and y. Note that markers: 1) do not affect the correct execution of the join operation since the values of the join attribute are taken from a domain different from that of the original relation (see Definition 3.2) and therefore, they cannot be joined with original tuples (see Theorem 4.1); 2) are not recognizable by the computational server (see Theorem 4.2) and therefore the computational server cannot selectively ensure the presence of all markers in the query result while not returning some genuine tuples.

3.3 Twins

The effectiveness of markers tends to decrease as the size of the query result increases (since the probability of the computational server to omit a marker when returning an incomplete join result decreases if markers are very few with respect to original tuples). To avoid introducing a large number of markers, we complement markers with an additional technique for ensuring query completeness. This technique consists in *twinning* (i.e., creating duplicates of) original tuples in the query result. A tuple appearing "solo" (i.e., without a twin) signals that the result is not complete.

Like for markers, our approach to inject twins in the query result consists in having the storage servers twinning tuples in their subquery results. The values of the join attribute for twinned tuples are obtained by associating a "Twin" flag set to 1 with the values of the join attribute of the twinned tuples. The flag is instead set to 0 for the original tuples and for markers. Note that twinned tuples: 1) do not affect the correct computation of the join introducing spurious tuples (see Theorem 4.1); and 2) are not recognizable as such and the computational server cannot even distinguish what tuples are originals and what tuples have been added due to the twinning operation (see Theorem 4.2).

The degree of integrity protection offered by twinning clearly depends on the number of tuples that are twinned and appear in the result of the join operation. In principle, all tuples in the subquery results can be twinned. This strategy offers an extremely high degree of integrity protection at the price of a considerable overhead in communication, since the size of the relations communicated from the storage servers to the computational server, as well as the size of the join would be twice as much as the original size. However, as we will see in Section 4, a limited number of twins provide sufficient integrity guarantees. Therefore, only a subset of the original tuples in the subquery results needs to be twinned. We assume that the client defines, and communicates to the storage servers, a twinning condition C_{twin} that determines the tuples in the subquery results that have to be twinned. Note that if a tuple is twinned at one storage server, also the tuple joining it at the other storage server must be twinned. To this purpose, C_{twin} should depend only on the join attribute, which is the attribute that the storage servers have in common. The twin tuples added to the subquery results are formally defined as follows:

Definition 3.3 (Twins). *Let B be a relation, I be the join attribute, and C_{twin} be a twinning condition. A set of* twins *for B is a set $\mathcal{T}$ of tuples over schema $B^*(I, Attr)$ such that $\forall t \in B$ where $t[I]$ satisfies C_{twin}, $\exists$ a distinct tuple $t' \in \mathcal{T}$, with $t'[I] = t[I]$, twin-flag $(t') = 1$, and $t'[Attr] = t[Attr]$.*

For instance, tables L^* and R^* in Fig. 3 extend L and R, respectively, by including twins for the tuples with join attribute equal to a and c. We use the bar notation to denote twins. For instance, $\bar{l}$ is the tuple twin of l and $\bar{a}$ is the join attribute value twin of a.

A trivial approach for the twinning criteria can request all actual values of the join attribute that satisfy some conditions to be duplicated. However, the client might not have information on the specific values occurring at the storage servers, and operating on actual values it might be difficult to control the amount of tuples eventually twinned (depending on the specific occurrences of actual values in the relation instances, a condition could result too selective, producing no twins, or too inclusive, producing a large number of twins). A better control on the twinning operation can be provided by operating on the result of a secure keyed hash function over the actual values of the join attribute. The common approach for secure keyed hash-functions relies on HMAC; SHA-3 (http://keccak.noekeon.org) can be used in a direct way, simply using the key as a prefix. Assuming such a keyed hash function h_k and assuming p_t the percentage of tuples that the client would like the storage servers to twin, the twinning condition can then impose to twin all the tuples such that $C_{\text{twin}} =$ "$h_k(t[I]) \bmod \lfloor \frac{1}{p_t} \rfloor = 0$ ". Since the hash function (unlike plaintext values) can be assumed to produce a

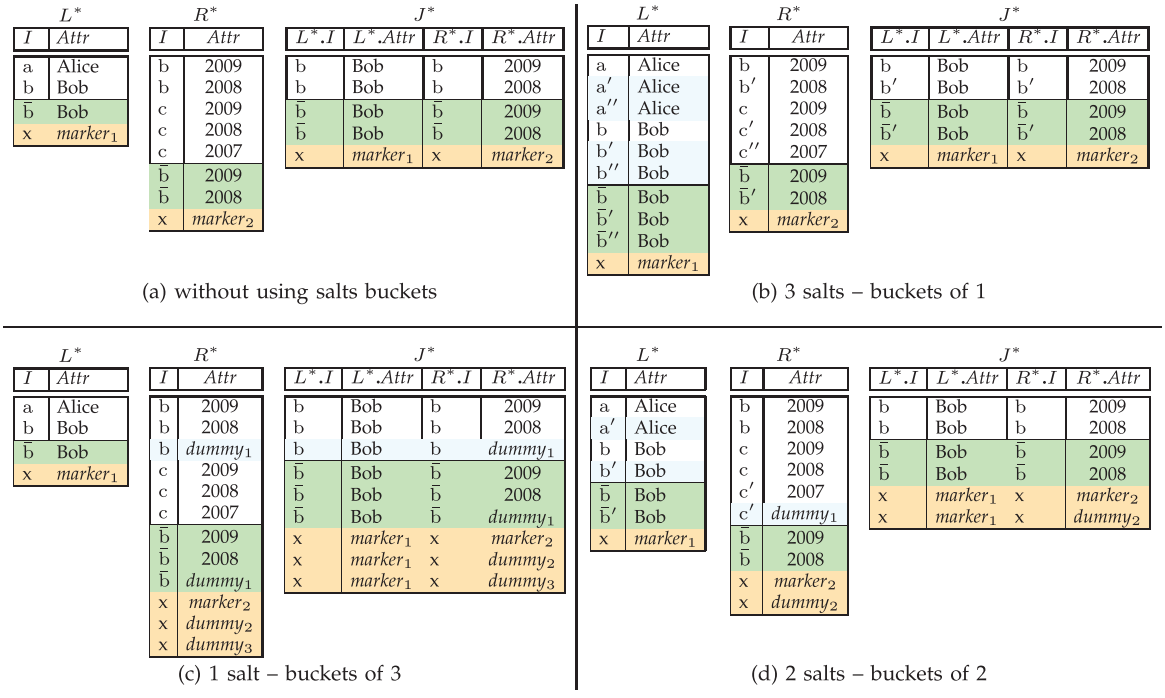


Fig. 4. Join with twins on b and one marker, varying the number of salts and the size of buckets.

uniform distribution of values, the condition on the modulo $\lfloor \frac{1}{p_t} \rfloor$ provides effective control on the percentage of twins actually inserted by the storage servers.

Example 3.1. Fig. 3 illustrates an example of application of our techniques. At the bottom of the figure, there are tables L and R resulting from the evaluation of the subqueries at $\mathcal{S}_l$ and $\mathcal{S}_r$, respectively. Insertion of two markers and twinning of tuples with join attribute equal to a and c produce relations L^* and R^*. Each storage server encrypts its relation with the query key k received from the client, producing encrypted relations L^*_k and R^*_k (according to Definition 3.1). The computational server executes the natural join between L^*_k and R^*_k and sends the result J^*_k to the client. The client projects over attributes $L^*.Tuple_k$ and $R^*.Tuple_k$, decrypts the result of projection, and checks if: the tuples in J^* have been correctly joined, (i.e., if $L^*.I = R^*.I$); the two expected markers belong to the join result; there is a twin for all the tuples with join attribute equal to a or c. Finally, the client removes markers and twins from J^* and projects over the requested attributes, obtaining the join result J.

3.4 Salts and Buckets

In the case of one-to-many joins, encryption on-the-fly, markers, and twins leave the number of occurrences of the different values of the join attribute (although not disclosing the specific values) visible to the computational server. The information on the number of occurrences of the join attribute could then be exploited by the computational server to identify twins and markers. For instance, multiple tuples with the same value for the join attribute will certainly not represent markers (since markers are all distinct). Also, uncertainty on twin tuples can be reduced by observing sets of tuples with the same number of occurrences for the join attribute. For instance, consider relations L^* and R^* in Fig. 4a where b has been twinned and one marker has been used. The encrypted view R^*_k of the computational server will have two join attribute values with two occurrences (β and $\bar{\beta}$, from b and $\bar{b}$), one with three occurrences (γ, from c), and one with one occurrence (χ, from x). The server can then: 1) infer that the only possible marker is χ and 2) exclude the fact that tuples with join attribute γ correspond to twins and markers. Tuples with encrypted join attribute equal to β and $\bar{\beta}$ could either be different genuine tuples with the same number of occurrences or twins.

Since frequency distribution of values can compromise the indistinguishability property of twins and markers, we need to destroy it. Two different approaches can be used, either in alternative or in combination. We illustrate each of them and then discuss how we apply them jointly. In the following, we assume L to be the relation on the side "one" and R to be the relation on the side "many" of the join.

Salts. The first approach applies salts in the encryption of values in R^* so that different occurrences of each join attribute value map to different encrypted values, producing for the computational server a view where all the values for the join attribute have one occurrence (i.e., reducing the view to be the same as in the case of one-to-one joins). For instance, with reference to relation R^* in Fig. 4a, the two occurrences of b, their twins $\bar{b}$, and the three occurrences of c will be made distinct by concatenating them with a different salt before their encryption. The view of the computational server (i.e., the encrypted version R^*_k of table R^* in Fig. 4b) will have eight tuples, all with distinct values for the join attribute. Here and in the following, given a join attribute value i, notations i' and i'' denote salted versions of value i, and notation $\simeq$ denotes that two values differ for a salt (i.e., $i \simeq i'$, and $i \simeq i''$). Note that (regardless of the value of the salt) $\simeq$ is reflexive,

symmetric, and transitive. To preserve the correctness of the join, also storage server $\mathcal{S}_l$ needs to modify its relation L^*. In fact, occurrences of values that have been salted in R^* will have to find a corresponding salted value in L^*. Assuming $nmax$ to be the number of occurrences of the join attribute value that appears more frequently in R, $\mathcal{S}_l$ creates, for each original or twin tuple, $nmax$ occurrences (one is the starting tuple and $nmax-1$ tuples are copies where different salts have been concatenated with the corresponding join attribute), to be potentially joined with tuples in R^*. Clearly, the salts used by storage server $\mathcal{S}_l$ for creating $nmax$ copies of the tuples in $L \cup \mathcal{T}$ are the same used by storage server $\mathcal{S}_r$ for salting the different occurrences of the same value in $R \cup \mathcal{T}$. For instance, assuming $nmax = 3$, L^* will need to have three occurrences, differing from a salt, for each value of the join attribute appearing in it, producing relation L^* in Fig. 4b.

Buckets. The second approach consists in flattening the frequency distribution, making all values of the join attribute in R^* have the same frequency $nmax$ (i.e., the maximum possible frequency). We call *bucket* a set of tuples with the same value for the join attribute. For values of the join attribute with a number of occurrences smaller than $nmax$, *dummy tuples* (i.e., tuples with the same value for the join attribute, but with dummy tuple content) are inserted. In this way, the encrypted view of the computational server on relation R^* will have all buckets with $nmax$ tuples. No change is needed for relation L^*. The insertion of dummy tuples will cause the presence of dummy tuples in the join (which the client can easily recognize and discard). To illustrate, consider the relations in Fig. 4a, assuming $nmax = 3$, storage server $\mathcal{S}_r$ will have to produce for table R^* all buckets of three tuples. This requires inserting some dummy tuples for values b, $\bar{b}$, and x of the join attribute (see Fig. 4c).

Salts and buckets. Salts and buckets have each advantages and disadvantages. The advantage of salts is that the size of relation R^* as well as of the join result J^* are not affected. The disadvantage is an increased size ($nmax$ times the original size) of relation L^*. The advantage of using buckets is that the size of the buckets can actually be established by the storage server storing relation R^* and therefore can be set to be exactly the actual $nmax$ in R (i.e., after the application of the selection condition), in contrast to the potential value that can be estimated by the client. The disadvantage is that the increased size of relation R^* affects also the size of the join J^*.

To enjoy the advantages of both techniques while minimizing their disadvantages, we combine the use of salts and buckets as follows: We assume that the client sets a maximum number s of salts that can be used (as we will discuss in Section 5, a good estimation for the number of salts is $\sqrt{nmax}$). Storage server $\mathcal{S}_l$ managing relation L will generate relation L^* by producing s tuples for each tuple in $L \cup \mathcal{T}$, as previously discussed, while leaving unchanged the marker tuples. Storage server $\mathcal{S}_r$ managing relation R will compute the maximum number $nmax$ of occurrences with which a value of join attribute appears. It will then generate relation R^* assuming buckets of size $b = \lceil \frac{nmax}{s} \rceil$, and using the same salt for all the tuples within a single bucket. For each value with a number of occurrences modulo b different from zero, the last bucket will be filled with dummy tuples. Note that the salts used by storage server $\mathcal{S}_l$ for creating s copies of the tuples in $L \cup \mathcal{T}$ and the salts used by storage server $\mathcal{S}_r$ for salting the at most s buckets originating from the same value of the join attribute must be the same.

To illustrate, consider relations L^* and R^* in Fig. 4a and assume the client sets the number s of salts to two. At storage server $\mathcal{S}_l$, table L^* is modified to have two occurrences (one original and one salted) of each non marker tuple, producing table L^* in Fig. 4d. At storage server $\mathcal{S}_r$, the maximum number of occurrences of the values of the join attribute is $nmax = 3$ (value c), and the size of the bucket is then $b = \lceil \frac{3}{2} \rceil = 2$. Enforcing buckets of size 2 on relation R^* in Fig. 4a produces the table in Fig. 4d. For b and $\bar{b}$, each having two occurrences, and x having one occurrence, only one bucket is needed. For c, the first two occurrences fall in the first bucket while a second bucket is needed for the third occurrence (which will be then salted). Such a second bucket and the bucket of x are then completed with a dummy tuple.

Relations L^* and R^* are formally defined as follows:

Definition 3.4 (L^*). *Let L be a relation, I be the join attribute, $\mathcal{M}$ be a set of markers for L, $\mathcal{T}$ be a set of twins for L, and s be the number of salts. Relation $L^*(I, Attr)$ is such that $\mathcal{M} \subseteq L^*$, and $\forall l \in L \cup \mathcal{T}$, $\exists\, l_1, \ldots, l_s \in L^*$ where a tuple corresponds to l and the other $s-1$ tuples are obtained with different salts in a way that: $l_{j_p}[I] \simeq l[I]$ and $l_{j_p}[Attr] = l[Attr]$, $p = 1, \ldots, s-1$.*

Relation L^* is obtained inserting a set $\mathcal{M}$ of markers and a set $\mathcal{T}$ of twins into the original relation L, and by possibly generating $s-1$ salted copies of each tuple in L and $\mathcal{T}$.

Definition 3.5 (R^*). *Let R be a relation, I be the join attribute, $\mathcal{M}$ be a set of markers for R, $\mathcal{T}$ be a set of twins for R, s be the number of salts, and $nmax$ be the maximum number of occurrences of a value of I in R. Relation $R^*(I, Attr)$ is such that:*

1. *$\forall v \in \pi_I(R \cup \mathcal{T} \cup \mathcal{M})$, $\exists Bu_1, \ldots, Bu_p \in R^*$ buckets of tuples of size $\lceil \frac{nmax}{s} \rceil$, with $p = \lceil \frac{freq(v)}{s} \rceil$, such that $\forall r, r' \in Bu_j, r[I] = r'[I]$ and $r[I] \simeq v$, $j = 1, \ldots, p$;*
2. *there exists a bijective function between the set of tuples in $R \cup \mathcal{T} \cup \mathcal{M}$ and the set of non-dummy tuples in R^*;*
3. *$\forall r, r' \in R^*$ such that r and r' are dummy tuples and $r[I] \simeq r'[I]$, then $r[I] = r'[I]$.*

Condition 1 states that, the original tuples of R, twin tuples in $\mathcal{T}$, and marker tuples in $\mathcal{M}$ are partitioned in the minimum number of buckets, such that for each bucket and each tuple in the bucket, the value for the join attribute has been salted with the same random value. Condition 2 states that, for each tuple in R, $\mathcal{T}$, and $\mathcal{M}$ there exists a corresponding non-dummy tuple in R^*. Condition 3 states that, for each value v of the join attribute of the tuples in R, $\mathcal{T}$, and $\mathcal{M}$, there is at most one bucket for v in R^* with dummy tuples.

With the adoption of our protection techniques, a client, which shares a key with the storage servers, sends to the

QUERY EVALUATION

/* q : user query SELECT A FROM B_l JOIN B_r ON $B_l.I$=$B_r.I$ WHERE C_l AND C_r AND C_{lr}
k_l : encryption key shared between C and $\mathcal{S}_l$
k_r : encryption key shared between C and $\mathcal{S}_r$
k : query encryption key set by C for query q
m : number of markers set by C for query q
p_t : percentage of twin tuples set by C for query q
h_k: cryptographic hash function
s : number of salts set by C for query q */

CLIENT

/* knows: q, k_l, k_r, k, encryption function decryption function, m, p_t, h_k, s */
c.01 /* sub-query to be evaluated by $\mathcal{S}_l$ */
c.02 q_l := "SELECT I, ($A \cup Att(C_{lr})) \cap B_l$ FROM B_l WHERE C_l"
c.03 *ToL* := Encrypt (q_l, k, m, p_t, s) with k_l
c.04 /* sub-query to be evaluated by $\mathcal{S}_r$ */
c.05 q_r := "SELECT I, ($A \cup Att(C_{lr})) \cap B_r$ FROM B_r WHERE C_r"
c.06 *ToR* := Encrypt (q_r, k, m, p_t, s) with k_r
c.07 J_k^* := Send "SELECT $*$ FROM *ToL* NATURAL JOIN *ToR*" to $\mathcal{C}_s$
c.08 J^* := Decrypt (J_k^*) with k
c.09 /* check if the tuples satisfy the join condition */
c.10 **if** $\exists t \in J^*$ s.t. $t[L^*.I] \neq t[R^*.I]$ **then return**(integrity error)
c.11 $\mathcal{M}_I$:= Generate m values of I for markers /* check markers */
c.12 **if** $\exists i \in \mathcal{M}_I$ s.t. $\nexists t \in J^*$, $t[L^*.I]=i$ **then return**(integrity error)
c.13 $\mathcal{M} := \{t \in J^* : t[L^*.I] \in \mathcal{M}_I\}$
c.14 *ToCheck* := $\{t' \in J^* : t'[L.I] \mod \lfloor \frac{1}{p_t} \rfloor = 0\}$, $\mathcal{T} := \emptyset$
c.15 **for each** $t \in$ *ToCheck* **do**
c.16 **if** $\nexists t' \in$ *ToCheck*, $t'[L^*.I]=t[L^*.I]$, twin-flag(t')=1,
c.17 $t'[L^*.Attr]=t[L^*.Attr]$, $t'[R^*.Attr]=t[R^*.Attr]$
c.18 **then return**(integrity error)
c.19 **else** *ToCheck* := *ToCheck* $\setminus \{t,t'\}$, $\mathcal{T} := \mathcal{T} \cup \{t\}$
c.20 /* check buckets */
c.21 Let b be the size of buckets in J^*
c.22 **if** $\exists i \in \pi_{L^*.I} J^*$ s.t. the number of tuples with $L^*.I=i$ is not b
c.23 **then return**(integrity error)
c.24 $J := J^* \setminus (\mathcal{M} \cup \mathcal{T} \cup \{t$: $t[R^*.Attr]$ is *dummy*$\})$ /* clean up */
c.25 Remove salts from J
c.26 *Res* := Evaluate "SELECT A FROM J WHERE C_{lr}"
c.27 **return**(*Res*)

COMPUTATIONAL SERVER

cs.1: Receive "SELECT $*$ FROM *ToL* NATURAL JOIN *ToR*" from the client
cs.2: L_k^* := Send *ToL* to $\mathcal{S}_l$
cs.3: R_k^* := Send *ToR* to $\mathcal{S}_r$
cs.4: J_k^* := Evaluate "SELECT $*$ FROM L_k^* JOIN R_k^* ON $L_k^*.I_k$=$R_k^*.I_k$"
cs.5: **return**(J_k^*)

STORAGE SERVER $\mathcal{S}_l$

/* knows: k_l, encryption function, h_k */
sl.01: Receive *ToL* from the computational server
sl.02: (q_l, k, m, p_t, s) := Decrypt *ToL* with k_l
sl.03: L := Evaluate q_l
sl.04: $\mathcal{M}_l$:= Generate m markers (Def. 3.2)
sl.05: $\mathcal{T}_l$:= for each $t \in L$ s.t. $t[I] \mod \lfloor \frac{1}{p_t} \rfloor = 0$ generate t' s.t.
sl.06: $t'[I]=t[I]$, twin-flag(t')=1, and $t'[Attr]=t[Attr]$ (Def. 3.3)
sl.07: L^* := for each $t \in (L \cup \mathcal{T}_l)$ generate $t_1, \ldots, t_s$ s.t.
sl.08: $\forall t' \in \{t_1, \ldots, t_s\}$, $t'[I] \simeq t[I]$ and $t'[Attr]=t[Attr]$ (Def. 3.4)
sl.09: $L^* := L^* \cup \mathcal{M}_l$
sl.10: L_k^* := Encrypt L^* with k
sl.11: **return**(L_k^*)

STORAGE SERVER $\mathcal{S}_r$

/* knows: k_r, encryption function, h_k */
sr.01: Receive *ToR* from the computational server
sr.02: (q_r, k, m, p_t, s) := Decrypt *ToR* with k_r
sr.03: R := Evaluate q_r
sr.04: $\mathcal{M}_r$:= Generate m markers (Def. 3.2)
sr.05: $\mathcal{T}_r$:= for each $t \in R$ s.t. $t[I] \mod \lfloor \frac{1}{p_t} \rfloor = 0$ generate t' s.t.
sr.06: $t'[I]=t[I]$, twin-flag(t')=1, and $t'[Attr]=t[Attr]$ (Def. 3.3)
sr.07: $nmax$:= maximum number of tuples with the same value for I
sr.08: $b := \lceil \frac{nmax}{s} \rceil$
sr.09: R^* := create groups of b tuples $\{t_1, \ldots, t_b\}$ s.t.
sr.10: $t_1[I]=\ldots=t_b[I]$; $\forall t \notin \{t_1, \ldots, t_b\}$, $t[I] \neq t_1[I]$;
sr.11: $\forall t \in \{t_1, \ldots, t_b\}$, $t \in (R \cup \mathcal{M}_r \cup \mathcal{T}_r)$ or $t[Attr]$ is dummy;
sr.12: $\nexists t, t' \in R^*$ s.t. $t[I] \simeq t'[I]$, $t[I] \neq t'[I]$,
sr.13: $t[Attr]$ and $t'[Attr]$ are dummy (Def. 3.5)
sr.14: R_k^* := Encrypt R^* with k
sr.15: **return**(R_k^*)

Fig. 5. The pseudo-code of the algorithms executed by the storage servers, the computational server, and the client for the evaluation of a query.

computational server a request to execute a join operation between the relations produced by storage servers $\mathcal{S}_l$ and $\mathcal{S}_r$. To this purpose, the client sends two strings encrypted with the key shared with the storage servers. Each string includes the subquery that the storage server should evaluate, the query key k, the number m of markers, the percentage p_t of twins, and the number s of salts. The query execution process then proceeds as already discussed and sketched in Fig. 2.

The pseudo-code of the algorithms executed by the storage servers, the computational server, and the client for the evaluation of a query is illustrated in Fig. 5.

4 CORRECTNESS AND ANALYSIS

We analyze the guarantees provided by our approach on the correctness and completeness of join results. We first note that, since encryption on-the-fly changes the encryption key at each query, the computational server cannot infer information by monitoring a sequence of queries. We then focus our analysis on a single query. (Proofs of theorems and lemmas are provided in the Appendix, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TCC.2013.18)

Before starting the analysis, we observe that the computational server is not able to recognize twins and markers from original tuples since all tuples in the encrypted relations received from the storage servers are indistinguishable (i.e., there is no information that the computational server can exploit to infer the nature of the encrypted tuples).

Proposition 4.1 (Encryption procedure). *Let B^* be a relation, I be the join attribute in B^*, and k be a cryptographic key. The encryption function E_k used for producing the encrypted version B_k^* of B^* satisfies the following two conditions:*

1. $\forall t, t' \in B^*$, $E_k(t[I]) = E_k(t'[I])$ *iff* $t[I] = t'[I]$;
2. $\forall t, t' \in B^*$, $E_k(t) \neq E_k(t')$.

We guarantee satisfaction of this proposition by using CBC with different bit configurations.

The *correctness* of the join result is guaranteed by the adoption of encryption on-the-fly. Intuitively, since the computational server does not know the encryption key, it cannot go undetected if it inserts fake tuples in the join result, modifies the tuples received from the storage servers, or badly combines tuples returning combinations that do not satisfy the join condition. Indeed the client, when decrypting the join result, would discover the misbehavior. Formally, the join result J_k^* is correct if it contains, in encrypted form, all the tuples in J^* and does not include spurious tuples. To prove the correctness, we first need to show that two plaintext tuples satisfy the equi-join condition on I iff their encrypted versions satisfy the equi-join condition on I_k, as stated by the following lemma.

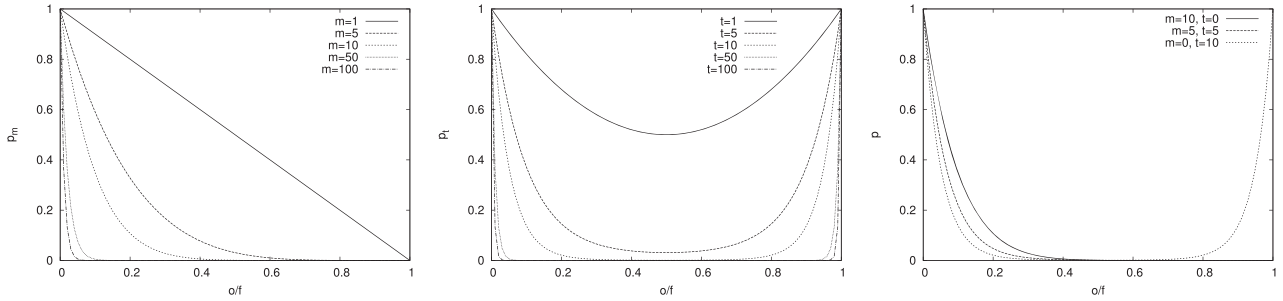


Fig. 6. Probability that the computational server omits a fraction $\frac{o}{f}$ of the tuples without being detected varying m (a), t (b), and both m and t (c).

Lemma 4.1. *Let L^* and R^* be two relations, I be the join attribute, L_k^* and R_k^* be the encrypted versions of L^* and R^*, respectively, $l \in L^*$ and $r \in R^*$ be a pair of tuples, and $\lambda \in L_k^*$ and $\rho \in R_k^*$ be the encrypted versions of l and r, respectively. We have that $l[I] = r[I]$ iff $\lambda[I_k] = \rho[I_k]$.*

Thanks to Lemma 4.1, we can prove that J_k^* is the encrypted representation of J^*, as formally stated by the following lemma.

Lemma 4.2. *Let L^* and R^* be two relations, L_k^* and R_k^* be their encrypted versions, J^* be the join $L^* \bowtie R^*$, and J_k^* be the join $L_k^* \bowtie R_k^*$. Tuple $\langle l,r \rangle \in J^*$ iff tuple $\langle \lambda,\rho \rangle \in J_k^*$, where λ is the encrypted version of l, and ρ is the encrypted version of r.*

Since J_k^* is the encrypted version of J^*, the correctness of J_k^* can be formulated on J^*. In the following theorem, we show that the join result does not include spurious tuples.

Theorem 4.1 (Correctness). *Let L^* and R^* be two relations, $\mathcal{M}_l$ and $\mathcal{M}_r$ be two sets of m markers for L and R, and $\mathcal{T}_l$ and $\mathcal{T}_r$ be two sets of twins for L and R. Relation J^* is equal to $L^* \bowtie R^* = (L \bowtie R) \cup (\mathcal{M}_l \bowtie \mathcal{M}_r) \cup (\mathcal{T}_l \bowtie \mathcal{T}_r) \cup (L^* \bowtie D)$, where $D = \{t \in R^* : t[R.Attr]$ is dummy$\}$, and $\mathcal{M}_l \bowtie \mathcal{M}_r$ contains m markers.*

The *completeness* of the join result is guaranteed by the adoption of markers and twins. We first observe that, for the computational server, all the values of the join attribute in an encrypted relation have the same number of occurrences. This observation is formally stated by the following lemma.

Lemma 4.3. *Let $B_k^*(I_k, B^*.Tuple_k)$ be an encrypted version of relation B^*. The frequency distribution of values in $B_k^*[I_k]$ is flat.*

The following theorem states that, as a consequence of Lemma 4.3 and of the encryption procedure adopted, tuples appearing in L_k^* (and R_k^*) received by the computational server are indistinguishable from each other.

Theorem 4.2 (Indistinguishability). *Let $B_k^*(I_k, B^*.Tuple_k)$ be an encrypted version of relation B^*. No inference can be drawn from the tuples in B_k^* about the corresponding tuples in B^* and therefore tuples in B_k^* are indistinguishable.*

The theorem above implies that the computational server cannot draw any inference about the correspondence between encrypted tuples and plaintext tuples and therefore cannot distinguish original tuples from markers and twins. Furthermore, if the computational server behaves correctly, the join result should include, without spurious tuples, all the m markers injected by the storage servers in their encrypted relations, and a twin for all the tuples whose join attribute satisfies the twinning condition (see Theorem 4.1). The guarantee of completeness offered by a join result with m markers and t twin pairs can then be measured as the probability $\wp$ that the computational server omits a given number o of tuples without being detected. Before proceeding with the analysis, we note that in the case of one-to-many joins, if the computational server omits only some of the tuples in a bucket, it will always be detected (since the result will have some incomplete buckets). Hence, when buckets are used, it is in the interest of the computational server to either preserve or omit buckets of tuples in their entirety. We can then consider all the tuples in a bucket as a single tuple, reducing the analysis to one-to-one joins.

The following theorem states the probability of the computational server to be undetected when omitting some tuples from the result.

Theorem 4.3. *Let J_k^* be a relation with cardinality f that includes m markers and t twin pairs. The probability that the computational server can omit o tuples (or equivalently a fraction $\frac{o}{f}$ of J_k^*) without being detected is $\wp < (1-\frac{o}{f})^m \cdot (1 - 2\frac{o}{f} + 2(\frac{o}{f})^2)^t$.*

Proof. The computational server can omit a fraction $\frac{o}{f}$ of J_k^* without being detected only if: 1) no marker is omitted, and 2) $\forall t_i,t_j$ of twin tuples, either t_i and t_j are both omitted or t_i and t_j are both preserved. The probability of omitting a given tuple t_i from J_k^* is $\frac{o}{f}$, while the probability of preserving it is $(1-\frac{o}{f})$. Hence, the probability $\wp_m$ that no marker is omitted is $(1-\frac{o}{f})^m$. Given a pair t_i,t_j of twin tuples, the probability that t_i and t_j are both omitted is $(\frac{o}{f})^2$, while the probability that t_i and t_j are both preserved is $(1-\frac{o}{f})^2$. The probability $\wp_t$ of either omitting or preserving every pair of twins without detection by the client is $((1-\frac{o}{f})^2 + (\frac{o}{f})^2)^t = (1 - 2\frac{o}{f} + 2(\frac{o}{f})^2)^t$. The probability of the omission being undetected by both markers and twins is then $\wp = \wp_m \cdot \wp_t = (1-\frac{o}{f})^m \cdot (1 - 2\frac{o}{f} + 2(\frac{o}{f})^2)^t$. □

Figs. 6a and 6b report the value of $\wp_m$ (probability that no marker is omitted) and $\wp_t$ (probability that every pair of

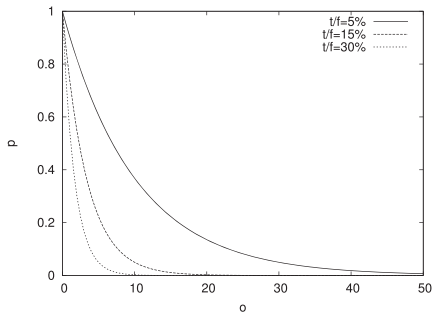


Fig. 7. Probability of the omission of o tuples to go undetected, varying the percentage $\frac{t}{f}$ of twins.

twins is either omitted or preserved) depending on the fraction $\frac{o}{f}$ of tuples omitted from the result for different values of m and t. It is interesting to note how just a few markers can provide effective protection when the join result is relatively small. However, when the fraction of omitted tuples is significantly small compared with the cardinality of the result, the detection ability of a fixed number of markers is limited. For twins, the probability of the server to be undetected is maximum when the fraction of omitted tuples is either zero or one (i.e., no tuple is omitted or all tuples are). The reason for the probability of being undetected to increase when $\frac{o}{f}$ is large is that clearly the greater the number of tuples the server omits, the more likely it is for the server to omit twins in pair (and therefore have the omission go undetected). In the extreme case, if only twins are used, the server could simply return an empty result and the omission will go undetected (since there would be no twin left "solo"). This observation strengthens our argument for the combined use of twins and markers. As a matter of fact, omitting a large fraction of the result, while increasing the probability of being undetected with respect to twin control, the server decreases the probability of being undetected with respect to marker control (since the greater the number of omissions, the greater the probability of omitting a marker). As a simple case, a single marker is sufficient to signal the fact that an empty result is incorrect. Fig. 6c reports the value of $\wp$ for different combinations of m and t, clearly showing that the best configuration is the one combining markers and twins (the configuration using only markers offers lower detection for low values of $\frac{o}{f}$; the configuration using only twins is not able to detect extreme omissions of tuples). A limited number of markers (independent from the cardinality f of the result, which the client cannot know a priori) together with a limited percentage of twins (adapting the number of twin pairs to the size of f) is sufficient for providing strong protection guarantees.

Note how for large values of f (i.e., for small fractions $\frac{o}{f}$ of omitted tuples) $\wp_t = (1 - 2\frac{o}{f} + 2(\frac{o}{f})^2)^t \approx (1 - 2\frac{o}{f})^t$, showing how twins are twice as effective as markers (recall that $\wp_m = (1 - \frac{o}{f})^m$). Also, $\wp_t \approx (1 - 2\frac{o}{f})^t = (1 - 2\frac{o}{f})^{\frac{t}{f}f} \approx e^{-2\cdot\frac{t}{f}o}$, with the crucial property that if we select twins as a percentage p_t $(= \frac{t}{f})$ of the tuples as illustrated in Section 3.3, the formula is independent from the size of the join result. In other words, the probability of the computational server to be undetected when omitting o tuples decreases exponentially with the percentage of twins inserted in the result and with the number of omitted tuples (see Fig. 7). We note that a very limited percentage of twins suffices for achieving strong guarantees. For instance, with just 5 percent of twins, the probability of the server to be undetected when omitting 50 tuples is 0.007, be they 50 tuples out of 10,000 or 50 out of 10,000,000.

5 Performance Analysis

We are interested in estimating the overhead produced by our techniques on the response time seen by the client. Such a response time depends on two main components, the computational time and the time taken by the transfer of data on the network. The computational overhead at the client as well as at the storage servers is limited. The client only needs to decrypt tuples and check the presence of markers and twins. The storage servers need to introduce markers, twins, and dummy tuples and encrypt tuples before sending them, in randomly shuffled order, to the computational server. The major computational time is with the computational server, where the join is executed. The computational server, however, can rely on a considerable amount of computational power and can apply traditional join optimization techniques ensuring efficiency of the computation. As testified by the experiments (see Section 6), the computational time is dominated by the data transfer time. We then focus our analysis on the communication overhead, due to the additional data communicated from the storage servers to the computational server and from the computational server to the client. Such an overhead Δ can be expressed as: $\Delta = \Delta_L \cdot \frac{size_L}{Cap_L} + \Delta_R \cdot \frac{size_R}{Cap_R} + \Delta_J \cdot \frac{size_J}{Cap_C}$, where: Δ_L, Δ_R, and Δ_J are the increased number of tuples in L_k^*, R_k^*, and J_k^* with respect to L, R, and J; $size_L$, $size_R$, and $size_J = (size_L + size_R)$ are the sizes of the tuples in L_k^*, R_k^*, and J_k^*, respectively; and Cap_L, Cap_R, and Cap_C are the capacities of the network channels between the computational server and storage server $\mathcal{S}_l$, storage server $\mathcal{S}_r$, and client C, respectively.

The additional tuples communicated from $\mathcal{S}_l$ are: one tuple for each marker, s tuples for each twin (one for the twinning and $s-1$ for their salting) and $s-1$ copies of each of the original tuples in L. Hence, $\Delta_L = |\mathcal{M}| + s \cdot |\mathcal{T}_l| + (s-1) \cdot |L|$, where $\mathcal{M}$ is the set of markers and $\mathcal{T}_l$ is the set of twins.

The additional tuples communicated from $\mathcal{S}_r$ are the dummy tuples potentially inserted for making all buckets of equal cardinality. In average, we can assume buckets with dummy tuples to contain $\frac{b-1}{2}$ dummy tuples. Every value occurring in the relation will have at most one bucket with dummy tuples. Assuming uniform distribution of values, the number of distinct values in $R[I]$ is $\frac{2\cdot|R|}{nmax}$ and the number of distinct twins is $\frac{2\cdot|\mathcal{T}_r|}{nmax}$. Hence, the additional tuples communicated from $\mathcal{S}_r$ are: b tuples for each marker (i.e., one marker and $b-1$ dummy tuples), one tuple plus possibly $\frac{b-1}{nmax}$ dummy tuples for each twin, and possibly $\frac{b-1}{nmax}$ dummy tuples for each tuple in the original relation. Therefore, $\Delta_R = b \cdot |\mathcal{M}| + (|\mathcal{T}_r| + |\mathcal{T}_r| \cdot \frac{b-1}{nmax}) + |R| \cdot \frac{b-1}{nmax}$, where $\mathcal{M}$ is the set of markers and $\mathcal{T}_r$ is the set of twins.

The additional tuples communicated from the computational server to the client are: b for each marker, and a percentage (depending on the selectivity σ of the join

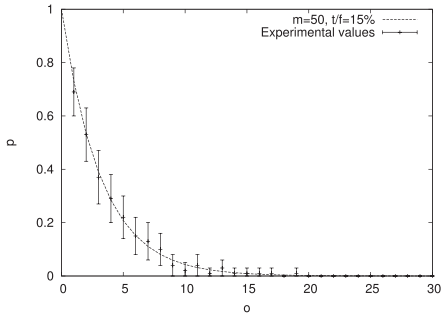


Fig. 8. Frequency of successful checks by the client after the removal by the computational server of tuples from the join result: comparison between the model in Theorem 4.3 and the average over 100 runs.

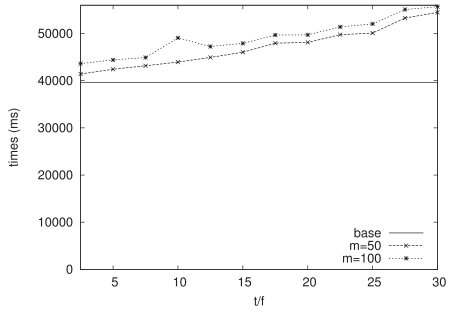


Fig. 9. Response time for $m = 50$ and $m = 100$ with a varying percentage of twins.

condition) of the additional tuples in the right-hand side of the join. Hence, $\Delta_J = b \cdot |\mathcal{M}| + \sigma \cdot (|\mathcal{T}_r| + |\mathcal{T}_r| \cdot \frac{b-1}{nmax}) + \sigma \cdot |R| \cdot \frac{b-1}{nmax}$.

As we can see from the formulas, the overhead due to markers and twins is very limited while the potential explosion in the amount of data transmitted is due to salts and buckets (to be paid for one-to-many joins), which then need to be set with care. Luckily, the formulas above provide also a possible good estimate for the number of salts and buckets able to minimize the communication overhead. In fact, the formula expressing the network overhead Δ due to the adoption of our techniques can be reformulated as a function of s and b as $c_s \cdot s + c_b \cdot b + c$, with the constraint that $s \cdot b \geq nmax$. To provide a good estimate for s and b, it is worth considering two possible scenarios depending on whether: 1) all communication channels have uniform bandwidth, or 2) the channel reaching the client has limited bandwidth. The first scenario is the typical scenario for distributed systems, while the second scenario covers applications where the client connects from mobile devices.

Uniform channels. Since all communication channels have the same bandwidth, we can minimize the network overhead Δ by simply minimizing function $\Delta \cdot Cap = \Delta_L \cdot size_L + \Delta_R \cdot size_R + \Delta_J \cdot size_J$, where $Cap = Cap_L = Cap_R = Cap_C$. To this aim, we can consider a common configuration where markers and twins represent a small fraction of L^* and R^* and their contribution can be disregarded, thus obtaining $\Delta \cdot Cap = (s-1) \cdot |L| \cdot size_L + |R| \cdot \frac{b-1}{nmax} \cdot size_R + \sigma \cdot |R| \cdot \frac{b-1}{nmax} \cdot size_J$. Since the only parameter that the client can set is s, we substitute b with $\lceil \frac{nmax}{s} \rceil$ in $\Delta \cdot Cap$ and compute the value for s that minimizes it, which is equal to

$$\sqrt{(|R|/|L|) \cdot ((size_R + \sigma(size_R + size_L))/size_L)}.$$

If the client has all the information required in the formula ($nmax$ can be estimated using classical tools of query optimization), it will be able to identify the value for s that maximizes efficiency in query evaluation. For instance, in a join where the ratio $\frac{|R|}{|L|}$ is equal to $nmax$, σ is close to zero and the tuples in R and L have identical size, the optimal value of s will be $\sqrt{nmax}$.

Slow client-channel. Since in this scenario the bottleneck is represented by the communication with the client, we only consider overhead Δ_J. This overhead is minimized when $b = 1$, and therefore when $s \geq nmax$. The optimal strategy is then to use a high number of salts, potentially equal to the maximum cardinality $nmax$ of the join attribute in table R. This choice multiplies by s times the size of table L_k^* sent to the computational server, but it also does not require the introduction of any dummy tuple in R_k^*.

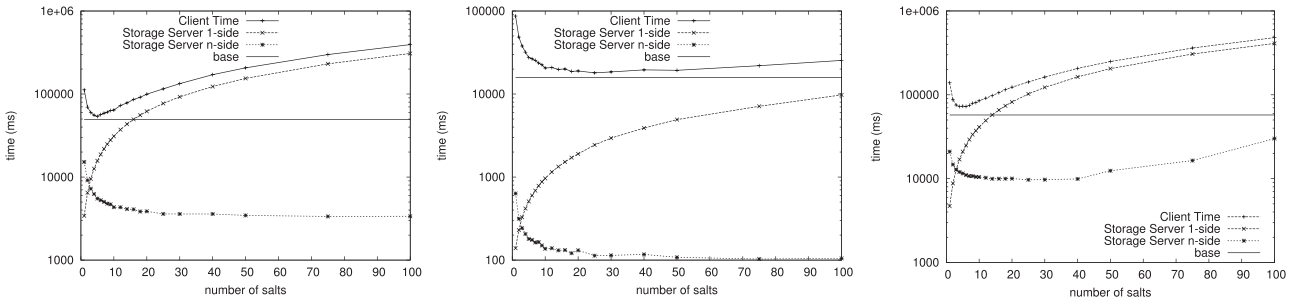
6 Experimental Results

To assess the integrity and performance guaranteed by our techniques, we run experiments: evaluating the ability of the techniques to detect omissions by the computational server, evaluating the performance impact due to the use of the techniques considering several parameters, and finally analyzing the economic benefits.

Integrity protection. We performed a set of experiments where the computational server returned to the client the result of the join without a set of randomly chosen tuples. We recorded, among a series of 100 experiments, the number of times that the client was able to detect the integrity violation. Fig. 8 considers the use of 15 percent of twins and 50 markers ($\frac{t}{f} = 0.15$, $m = 50$). The graph shows the percentage of times the omission was undetected, for a number of missing tuples in the range from 1 to 30, over a result of the join containing 1,000 tuples. The graph presents the observed detection probability and 95 percent-confidence error bars. The observed behavior confirms the probabilistic analysis described in Section 4, represented by the dotted line.

Performance. In the second set of experiments, we evaluated the performance impact of markers, twins, salts and buckets. We used for the computational server a machine with 2 Intel Xeon Quad 2.0 GHz, 12-GB RAM. The client machine was a standard PC running an Intel Core 2 Duo CPU at 2.4 GHz, with 4-GB RAM, connected to the computational server through a WAN connection with a 4 Mbps throughput. For the scenario with uniform channels, the storage servers were machines with the same characteristics as the client machine. For the scenario with slow client-channel, the storage servers were machines with Intel i7 CPU, 8-GB RAM, connected to the computational server through a 100-Mbps LAN. The values reported have been obtained as the average over six runs.

The first series of experiments was used to evaluate the overall response time due to the use of markers and twins. The experiments considered a one-to-one join over a synthetic database containing 1,000 tuples in both join



(a) uniform channels (synthetic data) (b) slow client-channel (synthetic data) (c) uniform channels TPC-H database

Fig. 10. Overall response time and data transfer time from $\mathcal{S}_l$ (1-side) and $\mathcal{S}_r$ (n-side).

operands and in the result. Fig. 9 shows the response time observed when executing the query without using our techniques (base) and when using 50 ($m = 50$) and 100 ($m = 100$) markers, varying the percentage $\frac{t}{f}$ of twins. The experiments show that the increase in response time due to markers is proportional to their number, and that the increase in response time due to the use of t twin pairs is, as expected, a fraction $\frac{t}{f}$ of the overall response time. A detailed analysis of the source of the response time shows that the computational time represents less than 10 percent of the overall time, for all the tested configurations. The computational time is mainly due to the execution at the computational server side of the join operation, which represents more than 50 percent of the computational time. Encryption, decryption, and integrity check operations represent altogether less than 15 percent of the computational time.

We then focused the analysis on the use of salts and buckets. We first considered the uniform channels scenario and tested configurations with $nmax$ equal to 50 and used a number of salts s varying between 1 and 100. Fig. 10a shows the overall client response time, the contribution to the overall client response time deriving from the transfer to the computational server of L_k^* and R_k^*, and the time observed when executing the same query without using our techniques (base). The graph clearly shows that the best performance is obtained close to values for s equal to 7 ($\lfloor\sqrt{50}\rfloor$). It also confirms that low values of s increase the size of R^* and high values of s increase the size of L^*. We executed the same queries in the slow client-channel scenario, where the storage servers were using a fast channel to communicate with the computational server. The experiments in Fig. 10b confirm the results described in Section 5 for the slow client-channel scenario, where the minimal network overhead is achieved with small values of b and high values of s.

To obtain a further verification of the behavior of our techniques, we performed experiments over the well-known TPC-H database [10]. Fig. 10c shows the overall response time observed by the client when executing a join between relations CUSTOMERS and ORDERS. We considered 9,000 customers and the maximum number of orders per customer ($nmax$) is 41. The experiments used 50 markers, 15 percent of twins, and considered a variety of values for s. The graph shows the overall client response time, the contribution to the client response time deriving from the transfer of L_k^* and R_k^* from the storage servers to the computational server, and the time observed when executing the same query without using our techniques (base). For the configurations with values close to $\sqrt{41}$, the overhead due to markers, twins, salts, and buckets is 30 percent. Considering the inevitable 15 percent time increase due to twins, the response time appears reasonable.

Another set of experiments was dedicated to evaluate the performance of the approach for large databases (up to 2 GB in each table) and the possible impact of latency on the computation, comparing the response times for queries over local networks (local client configuration) with those obtained with a client residing on a PC at a distance of 1,000 Km connected through a shared channel that in tests demonstrated to offer a sustained throughput near to 80 Mbps (remote client configuration). The experiments used a synthetic database with two tables each with a number of tuples between 10^4 and 10^6 and tuples with size equal to 100 bytes. Each data point is the average of six runs of the experiments, which compute one-to-one joins using 10 percent of twins. The results of these experiments are reported in Fig. 11. The results demonstrate that our techniques can be applied over large tables with millions of tuples without a significant overhead. Also, the impact of latency is shown to be modest, as the comparison between local client and remote client configurations of the response times for the same query shows a limited advantage for the local client scenario, consistent with the limited difference in available bandwidth.

Economic analysis. We focused on evaluating the economic advantage of our solution when executing queries [8]. In fact, while implying a slight performance overhead, our solution in the majority of cases can be less

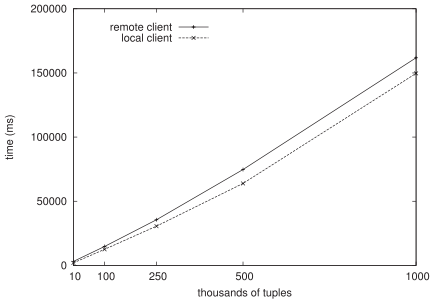


Fig. 11. Response time with large databases and variable latency.

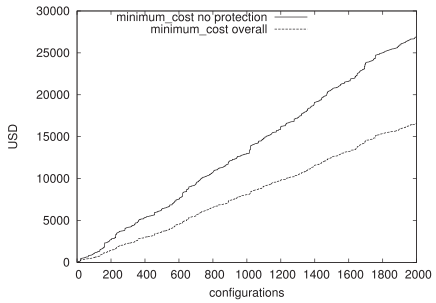


Fig. 12. Total economic cost of executing 2,000 join queries without (continuous line) or with (dotted line) availability of our techniques.

expensive, since it enables the use of available low-cost computational servers.

In our analysis, we assumed economic costs varying in line with available solutions (e.g., Amazon S3 and EC2, Windows Azure, GoGrid), number of tuples to reflect realistic query plans, and twin and markers varying like for the performance analysis. In particular, we considered the following parameters:

1. cost of transferring data out of each storage server (from 0.00 to 0.30 USD per GB) and of the computational server (from 0.00 to 0.10 USD per GB);
2. cost of transferring data to the client (from 0.00 to 2.00 USD per GB; we did not consider the cost of input data for the storage and computational servers since all the price lists we accessed let in-bound traffic be free);
3. cost of CPU usage for each storage server (from 0.05 to 2.50 USD per hour) and for the computational server (from 0.00 to 0.85 USD per hour);
4. bandwidth of the channel reaching the client (from 4 to 80 Mbps);
5. size of the join attribute $size_I$ (from 1 to 50 bytes);
6. number of tuples in L (from 10 to 1,000) and the size of the other attributes $size_L - size_I$ (from 1 to 2,000 bytes);
7. number of tuples in R (from 10 to 10,000,000) and the size of the other attributes $size_R - size_I$ (from 1 to 2,000 bytes);
8. number m of markers (from 0 to 50);
9. percentage p_t of twins (from 0 to 0.30);
10. number s of salts (from 1 to 100);
11. maximum number $nmax$ of occurrences of a value in $R.I$ (from 1 to 60);
12. selectivity σ of the join operation (from 0.01 to 1.00).

We used a Monte Carlo method to generate 2,000 simulations varying the parameters above. For each simulation, we evaluated the cost of executing a join operation at one of the storage servers without adopting our protection techniques (considering the cheaper alternative obtained relying on each of the storage servers for join execution), and at the computational server adopting our protection techniques. The query optimizer can assess the best strategy to use for performing a query, adopting the approach that provides the best economic advantage. Fig. 12 illustrates the total costs for executing a query with (and without respectively) the availability of our techniques, assuming their adoption whenever economically convenient. As visible in the figure, without availability of our techniques the total cost (continuous line) reaches 26,922 USD, while with the availability of our techniques (dotted line) it remains at 16,559 USD, corresponding to a total saving of 38.49 percent.

7 Related Work

Previous work is related to the data outsourcing scenario, where data are assumed to be stored at an external *honest-but-curious server* that correctly manages the data and guarantees their availability, but that is not trusted to access the data content (e.g., [3], [4], [5], [10], [11], [12], [13], [14], [15], [16], [17]). To protect data confidentiality, data are encrypted before outsourcing and indexing information, stored together with the encrypted data, are used for the evaluation of conditions (e.g., [18], [19]) and of join operations (e.g., [10]) at the server side. In many real world scenarios, the assumption that the external server is honest-but-curious is not applicable. Different techniques have then been proposed to provide *data integrity*, in terms of: 1) correctness, 2) completeness, and 3) freshness of query results. Correctness is traditionally provided by means of signature techniques, adequately revised to permit efficient signature composition (e.g., [14], [20]). Completeness can instead be provided either by means of authenticated data structures (e.g., [15], [21], [22]) or probabilistic approaches (e.g., [3], [4], [5]). Freshness is obtained by including a periodically updated timestamp in authenticated data structures or by periodically changing the function that generates the data used for integrity verification [22].

The approach in [3] stores at the external server a copy of the tuples that satisfy a predefined condition, encrypted using a different secret key. The completeness of the query result is guaranteed by the presence in the query result of two instances for all the tuples that satisfy the query selection condition and the selection condition used to define the additional tuples. The solution in [4] defines a deterministic function that generates a set of fake tuples that are inserted into the data set stored at the external server at initialization time. While our proposal and the approaches in [3], [4] share the idea of replicating tuples or including fake tuples, there are many crucial differences. First, our work does not assume that the client has knowledge of the data externally stored. Also, our work accommodates the dynamic generation of markers and twins: the computational server cannot accumulate information from the analysis of a sequence of queries (two executions of the same query are associated with two indistinguishable structures for integrity verification); also, the number of markers and twin pairs can change at each query. The combined use of markers and twins grants us, for the same number of additional tuples, a higher probabilistic guarantee of completeness and a more efficient adaptation to the desired protection requirements. Also, in [3] joins are not considered. In [4] the authors extend their solution to join operations by inserting fake tuples in both the relations participating in the join, but their solution may produce spurious tuples and may cause a high network overhead. An interesting technique designed to guarantee the completeness of a join result has been illustrated in [5]. This technique builds a Merkle hash tree on the join attributes of at least one

of the two relations involved in the join and uses this data structure to provide a completeness guarantee of the result. The approach is quite different from the one proposed by us, as it is deterministic whereas our technique adopts a probabilistic approach.

A different, though related, line of works considered the design of mechanisms providing integrity guarantees on the results of large computation tasks outsourced to the cloud (e.g., [23], [24], [25], [26]), and the enforcement of restrictions in task provisioning (e.g., [27]) or data outsourcing (e.g., [28], [29], [30]).

8 Conclusions and Future Directions

Cloud technology is evolving today at a quick pace, and we see many applications where a client may ask a computational service to support the evaluation of a join over collections of data kept in separate storage servers. In such scenarios, it is important to provide the client with the ability to assess the integrity of the result returned by the computational server. Our proposal responds to this need, providing complementary techniques that operate in synergy effectively and efficiently. The extensive experimental results performed make us confident that our techniques can be applied in real scenarios, offering clear protection guarantees at a limited configurable overhead and providing economic benefit. There are several interesting directions in which our work can be extended. A first direction concerns the consideration of joins involving not only equality conditions; for instance approximate joins could be enforced with our proposal via domain discretization and postprocessing. Another direction relates to the diversification of trust assumptions over the storage servers which, for instance, could be considered not fully trustworthy for performing their portion of the query. Our techniques can be extended to operate in this context by assuming encrypted tables and generating separate twins and markers for controlling the storage servers. A further interesting direction is the involvement in the query of multiple computational and/or storage servers and the consideration of MapReduce scenarios. In these contexts, our techniques can be applied to parallel joins or chains of joins also investigating different strategies for the join executions and parallelization of the tasks.

Acknowledgments

This work was supported in part by: EC 7FP project PoSecCo (257129), Italian MIUR PRIN project "GenData 2020," a Google Research Award, US National Science Foundation (NSF) grants CT-20013A and IIP-1266147, ARO grant W911NF-13- 1-0317, and ONR grant N00014-13-1-0703.

References

[1] A. Ceselli, E. Damiani, S. De Capitani di Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati, "Modeling and Assessing Inference Exposure in Encrypted Databases," *ACM Trans. Information and System Security,* vol. 8, no. 1, pp. 119-152, Feb. 2005.

[2] H. Hacigümüs, B. Iyer, and S. Mehrotra, "Providing Database As a Service," *Proc. 18th Int'l Conf. Data Engineering (ICDE '02),* Feb. 2002.

[3] H. Wang, J. Yin, C. Perng, and P. Yu, "Dual Encryption for Query Integrity Assurance," *Proc. 17th ACM Conf. Information and Knowledge Management (CIKM '08),* Oct. 2008.

[4] M. Xie, H. Wang, J. Yin, and X. Meng, "Integrity Auditing of Outsourced Data," *Proc. 33rd Int'l Conf. Very Large Data Bases (VLDB '07),* Sept. 2007.

[5] Y. Yang, D. Papadias, S. Papadopoulos, and P. Kalnis, "Authenticated Join Processing in Outsourced Databases," *Proc. ACM Int'l Conf. Management of Data (SIGMOD '09),* June/July. 2009.

[6] V. Varadharajan and U.K. Tupakula, "TREASURE: Trust Enhanced Security for Cloud Environments," *Proc. IEEE 11th Int'l Conf. Trust, Security and Privacy in Computing and Comm. (TrustCom '12),* June 2012.

[7] K.Y. Oktay, V. Khadilkar, B. Hore, M. Kantarcioglu, S. Mehrotra, and B.M. Thuraisingham, "Risk-Aware Workload Distribution in Hybrid Clouds," *Proc. IEEE Fifth Int'l Conf. Cloud Computing,* June 2012.

[8] D. Kossmann, T. Kraska, and S. Loesing, "An Evaluation of Alternative Architectures for Trans. Processing in the Cloud," *Proc. ACM Int'l Conf. Management of Data (SIGMOD '10),* June 2010.

[9] "The Transaction Processing Performance Council (Tpc) Benchmark H," http://www.tpc.org/tpch/, 2011.

[10] B. Carbunar and R. Sion, "Toward Private Joins on Outsourced Data," *IEEE Trans. Knowledge and Data Engineering,* vol. 24, no. 9, pp. 1699-1710, Sept. 2012.

[11] C. Curino et al., "Relational Cloud: A Database Service for the Cloud," *Proc. Fifth Biennial Conf. Innovative Data Systems Research (CIDR '11),* Jan. 2011.

[12] E. Curtmola, A. Deutsch, K. Ramakrishnan, and D. Srivastava, "Load-Balanced Query Dissemination in Privacy-Aware Online Communities," *Proc. ACM Int'l Conf. Management of Data (SIGMOD '10),* June 2010.

[13] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati, "Encryption Policies for Regulating Access to Outsourced Data," *ACM Trans. Database Systems* , vol. 35, no. 2, pp. 12:1-12:46, Apr. 2010.

[14] E. Mykletun, M. Narasimha, and G. Tsudik, "Authentication and Integrity in Outsourced Databases," *ACM Trans. Storage* , vol. 2, no. 2, pp. 107-138, May 2006.

[15] H. Pang and K. Tan, "Verifying Completeness of Relational Query Answers from Online Servers," *ACM Trans. Information and System Security,* vol. 11, no. 2, pp. 5:1-5:50, May 2008.

[16] K. Ren, C. Wang, and Q. Wang, "Security Challenges for the Public Cloud," *IEEE Internet Computing,* vol. 16, no. 1, pp. 69-73, Jan.-Feb. 2012.

[17] C. Wang, Q. Wang, K. Ren, N. Cao, and W. Lou, "Toward Secure and Dependable Storage Services in Cloud Computing," *IEEE Trans. Services Computing,* vol. 5, no. 2, pp. 220-232, Jan. 2012.

[18] H. Hacigümüs, B. Iyer, S. Mehrotra, and C. Li, "Executing SQL over Encrypted Data in the Database-Service-Provider Model." *Proc. ACM Int'l Conf. Management of Data (SIGMOD '02),* June 2002.

[19] H. Wang and L.V.S. Lakshmanan, "Efficient Secure Query Evaluation over Encrypted XML Databases," *Proc. 32nd Int'l Conf. Very Large Data Bases (VLDB '06),* Sept. 2006.

[20] H. Hacigümüs, B. Iyer, and S. Mehrotra, "Ensuring Integrity of Encrypted Databases in the Database-As-A-Service Model," *Proc. IFIP WG11.3 Working Conf. Data and Application Security (DBSec '03),* Aug. 2003.

[21] P. Devanbu, M. Gertz, C. Martel, and S. Stubblebine, "Authentic Third-Party Data Publication," *Proc. IFIP WG11.3 Working Conf. Database and Application Security (DBSec '00),* Aug. 2000.

[22] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin, "Dynamic Authenticated Index Structures for Outsourced Databases," *Proc. ACM Int'l Conf. Management of Data (SIGMOD '06),* June 2006.

[23] V. Vu, S. Setty, A.J. Blumberg, and M. Walfish, "A Hybrid Architecture for Interactive Verifiable Computation," *Proc. IEEE Symp. Security and Privacy (SP '13),* May 2013.

[24] C. Wang, K. Ren, and J. Wang, "Secure and Practical Outsourcing of Linear Programming in Cloud Computing," *Proc. INFOCOM,* Apr. 2011.

[25] M.J. Atallah and K.B. Frikken, "Securely Outsourcing Linear Algebra Computations," *Proc. Fifth ACM Symp. Information, Computer and Comm. Security (ASIACCS '10),* Apr. 2010.

[26] Z. Xu, C. Wang, Q. Wang, K. Ren, and L. Wang, "Proof-Carrying Cloud Computation: The Case of Convex Optimization," *Proc. INFOCOM,* Apr. 2013.
[27] R. Jhawar, V. Piuri, and P. Samarati, "Supporting Security Requirements for Resource Management in Cloud Computing," *Proc. 15th IEEE Int'l Conf. Computational Science and Eng.,* Dec. 2012.
[28] E. Damiani, S. De Capitani di Vimercati, S. Foresti, S. Jajodia, and P. Samarati, "Key Management for Multiuser Encrypted Databases," *Proc. Int'l Workshop Storage Security and Survivability,* Nov. 2005.
[29] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, and G. Livraga, "Enforcing Subscription-Based Authorization Policies in Cloud Scenarios," *Proc. IFIP WG11.3 Working Conf. Data and Application Security and Privacy,* July 2012.
[30] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, G. Pelosi, and P. Samarati, "Encryption-Based Policy Enforcement for CloudStorage," *Proc. First Workshop Security and Privacy in Cloud Computing,* June 2010.

Sabrina De Capitani di Vimercati is a professor at the Computer Science Department, Università degli Studi di Milano, Italy. Her research interests include the area of security, privacy, and data protection. She has been a visiting researcher at SRI International, California, and George Mason University, Virginia. She is a chair of the IFIP WG 11.3 on Data and Application Security and Privacy. She is a senior member of the IEEE. http://www.di.unimi.it/decapita.

Sara Foresti is an assistant professor at the Computer Science Department, Università degli Studi di Milano, Italy. She has been a visiting researcher at George Mason University, Virginia. She has been serving as a PC chair and member of several conferences. Her research interests include the area of security and privacy. http://www.di.unimi.it/foresti. Her PhD thesis received the ERCIM STM WG 2010 award. She is a member of the IEEE.

Sushil Jajodia is a professor and the director of CSIS at George Mason University, Virginia. His research interests include information security and privacy. He has authored or coauthored six books and more than 425 papers, and edited 41 books and conference proceedings. He holds 12 patents. He has received several awards. He has been named IEEE fellow in 2013. http://csis.gmu.edu/jajodia.

Stefano Paraboschi is a professor and a deputy-chair at the Dipartimento di Ingegneria of the Università degli Studi di Bergamo, Italy. He has been a visiting researcher at Stanford University and IBM Almaden, California, and George Mason University, Virginia. His research interests include information security and privacy, web technology for data intensive applications, XML, information systems, and database technology. He is a member of the IEEE. http://cs.unibg.it/parabosc.

Pierangela Samarati is a professor at the Computer Science Department, Università degli Studi di Milano, Italy. She has published more than 230 papers in journals, conference proceedings, and books. She has received several awards. Her main research interests are in data protection, security and privacy. She has been named an ACM distinguished scientist, in 2009, and IEEE fellow, in 2012. http://www.di.unimi.it/samarati.