Efficient detection of unauthorized data modification in cloud databases

Luca Ferretti, Fabio Pierazzi, Michele Colajanni, Mirco Marchetti, and Marcello Missiroli
University of Modena and Reggio Emilia
Email: {luca.ferretti, fabio.pierazzi, michele.colajanni, mirco.marchetti, marcello.missiroli}@unimore.it

***Abstract*—Cloud services represent an unprecedented opportunity, but their adoption is hindered by confidentiality and integrity issues related to the risks of outsourcing private data to cloud providers. This paper focuses on integrity and proposes an innovative solution that allows cloud tenants to detect unauthorized modifications to outsourced data while minimizing storage and network overheads. Our approach is based on encrypted Bloom filters, and is designed to allow efficient integrity verification for databases stored in the cloud. We assess the effectiveness of the proposal as well as its performance improvements with respect to existing solutions by evaluating storage and network costs.**

I. Introduction

Cloud computing represents an important business opportunity for many enterprises and organizations that are attracted by high availability, scalability and elasticity properties characterizing cloud-based services. Their adoption is often hindered by the security issues related to outsourcing of data to providers that cannot be completely trusted by the tenants (e.g., [1]). Most literature related with cloud security assumes that the cloud provider is *honest-but-curious*, meaning that cloud service operations are always executed correctly, but the cloud provider may access tenants data without being authorized. This paper explores a more complex threat model not excluding that some cloud provider employees may modify or corrupt tenants data. In a similar scenario, it is important to devise integrity algorithms and protocols that make it possible for cloud tenants to detect unauthorized modifications to their data.

There are several solutions based on Message Authentication Codes (MAC) that can guarantee the integrity of tenant files stored in clouds [2]. Unfortunately, the application of these algorithms to cloud databases would cause storage and/or network overheads to the extent of preventing the convenience of cloud services.

This paper proposes a novel scheme relying on Bloom filters for the detection of unauthorized data modifications by the cloud employees. This solution is specifically optimized for cloud databases, and it is designed to be orthogonal with respect to data encryption strategies for data confidentiality proposed in literature [3]–[5]. Hence, it can be integrated with existing encryption algorithms so to achieve both confidentiality and integrity of data stored in cloud databases. The benefits of the proposed scheme are demonstrated by evaluating storage and network overheads, and comparing them against those related to MAC-based schemes under the TPC-C workload, that is a famous benchmark for database services [6].

The remaining part of the paper is organized as follows. Section II compares our solution with existing proposals. Section III offers a detailed description of the proposed solution, while its security guarantees are discussed in Section IV. Performance evaluation in terms of storage and network overheads, as well as comparisons with respect to other schemes for database integrity are provided in Section V. Finally, Section VI concludes the paper by summarizing its main contributions and future work.

II. Related work

Many cloud providers offer database services [7], but the security in terms of tenant's data confidentiality and integrity is an open research area. Some papers are oriented to guarantee data confidentiality [3]–[5] under the commonly adopted *honest-but-curious* cloud provider threat model. That is, some cloud provider employees may read tenant's data to try obtaining confidential information, but they do not modify them and they execute all protocols correctly. These proposals focus on the development of architectures and encryption strategies that allow the tenant to execute SQL operations on encrypted data. Other proposals also consider the issues related to data integrity in the context of cloud storage services, while few results exist that are related to data integrity of cloud database services.

The solutions that are commonly adopted to guarantee file integrity are based on the association of MACs to all the files that are stored in the cloud [2]. In such a way, a cloud tenant can verify data integrity by downloading a file, recomputing its MAC and comparing it with the MAC stored in the cloud. Since the cloud provider cannot fake a new MAC and cannot modify the file without altering the resulting MAC, data integrity is guaranteed if and only if the two MAC codes match. This approach could be immediately applied to cloud database services by associating a MAC with each attribute (that is, to each element of each tuple) stored in the database. However, since the size of a MAC is non-trivial (e.g., 256 bits for HMAC based on SHA256) and bigger than many primitive data types, this solution is not viable because of excessive storage overhead.

Other schemes for guaranteeing data integrity in cloud database services were proposed in [8], where the key idea is to reduce the storage overhead by associating one control value with multiple attributes. For example, the authors in [8] present a scheme called *Condensed-RSA* (CRSA), in which a control value is associated with a combination of all the attributes of the same tuple. A drawback of similar approaches is that whenever a cloud tenant aims to verify the integrity of

a_1	a_c	a_C	
$v_{1,1}$	$\dots$	$v_{1,C}$	e_1
$\dots$	$v_{r,c}$	$\dots$	e_r
$v_{R,1}$	$\dots$	$v_{R,C}$	e_R

TABLE I: Database table enriched with encrypted Bloom filters.

one attribute, he has to retrieve all the other attributes that are needed to compute the control value. In the CRSA scheme, it is necessary to download all the attributes of a row, even if the cloud tenant needs only one of them. Hence, the reduction of the storage overhead is achieved at the expense of increased network overheads. RDAS is another proposal related to data integrity [9]. Since it is applicable only to the context of static databases where data are never updated, it is not viable in most real scenarios.

This paper proposes a novel scheme that detects unauthorized data modifications while minimizing both storage and network overheads through Bloom filters [10]. These filters are proposed to guarantee integrity in cloud storage contexts in [11], but the existing scheme cannot be immediately applied to cloud database services. To the best of our knowledge, this is the first paper that proposes a scheme for efficient integrity verification in cloud databases based on encrypted Bloom filters. We include an original evaluation of storage and network overheads based on the TPC-C workload benchmark that demonstrates the convenience of the proposal with respect to the state of the art.

III. Scheme design

In this section we describe the novel scheme that uses encrypted Bloom Filters to guarantee integrity of the cloud database while minimizing storage and network overheads.

Let us consider a table having R rows and C columns: we denote as $v_{r,c}$, where $r = [1, \dots, R]$ and $c = [1, \dots, C]$, the value stored at the r-th row and c-th column of the table. Similarly, V_r is defined as the set of all values that belong to the r-th row, that is $(v_{r,1}, \dots, v_{r,C})$. We extend the database schema by adding a new column to all the tables of the database. Each row of the new column stores an encrypted version of a control structure that allows the tenant to verify the integrity of all the data stored in the same row. The notation e_r identifies the encrypted control structure associated with the r-th row of a table. A representation of a table modified according to the proposed scheme is given in Table I, where a_c represents the name that is associated with the column c.

We compute e_r according to the following equation:

$$e_r = \mathcal{E}(s, b_r), \tag{1}$$

where $\mathcal{E}$ represents a symmetric encryption algorithm with IND-CPA security guarantees (*Indistinguishable under chosen-plaintext attack*, e.g. Blowfish [12] or AES [13] with random IV), s is the encryption key, and b_r is a Bloom filter computed over data that includes all values belonging to V_r. We assume that all client machines used by the cloud tenant to access and manipulate data stored on the cloud database share the same encryption key s, that can be distributed according to well known key distribution schemes [14], as well with more efficient strategies that are specific to the field of cloud databases [4], [15]. We assume that the cloud provider does not know the value of s.

The value b_r is computed as:

$$b_r = \bigvee_{c=1}^{C} \mathcal{B}(a_c | v_{r,c}), \tag{2}$$

where $\mathcal{B}$ is a function that computes a Bloom filter [16], the operator $|$ is the concatenation between two values, and $\bigvee$ represents the bitwise OR between the C Bloom filters computed through $\mathcal{B}$.

Our scheme can be easily integrated with previous proposals for cloud database confidentiality that are based on the encryption of all the values stored in the cloud database and that support the execution of SQL operations on encrypted data [3]–[5]. These related works protect tenant data against honest-but-curious cloud providers that do not modify the database. On the other hand, our scheme improves security by allowing cloud tenants to detect unauthorized modifications to its data by a malicious cloud provider. If each value $v_{r,c}$ stores an encrypted value computed according to any of these proposals, then the proposed scheme guarantees confidentiality, integrity and allows the execution of SQL operations.

IV. Security analysis

We describe the security guarantees provided by the proposed scheme in two steps. In the former we do not consider the effect of false positives that are inherent in Bloom filters. In the latter we describe how our solution limits the detrimental effects of false positives.

A. Design choices

The use of Bloom filters instead of hash or MAC functions makes it possible to test whether a value belongs to a set without the need to retrieve all the other values that belong to the same set. This property is useful in the field of database integrity, since it allows a tenant to check whether a value belongs to a tuple by only downloading the required value and the Bloom filter related to the corresponding row of the database table. After having retrieved a value $v_{r,c}$ and the corresponding Bloom filter b_r computed according to Equation (2), then the tenant can verify that $v_{r,c}$ is a legitimate value of the tuple by checking if:

$$b_r \wedge \mathcal{B}(a_c | v_{r,c}) = \mathcal{B}(a_c | v_{r,c}) \tag{3}$$

where $\wedge$ is the bitwise AND operator. If Equation (3) is not verified, then the tenant has evidence that the $v_{r,c}$ has been modified after the computation of b_r. Let us consider the possible attacks that a malicious cloud provider could attempt to thwart a similar scheme.

The first family of attacks is based on possible modifications to the control structures used to check data integrity. Let us suppose that we store plaintext Bloom filters instead of encrypted Bloom filters in the cloud database (that is, b_r instead of e_r in Table I). Then a cloud provider could alter the value of b_r to its advantage. As an example, a cloud provider could alter any $v_{r,c}$ and then recompute the corresponding b_r to reflect the modified values. Another attack could be performed by setting more bits of the Bloom filter to one, thus increasing the probability that an integrity test succeeds even for values that are not included in the Bloom filter. Our proposal protects data integrity against all these attacks by storing on the cloud server a version of the Bloom filter that is encrypted with a secret key only known to the cloud tenant. Any modification to the encrypted Bloom filter would be easily detected by the tenant, because the resulting decrypted Bloom filter would have a completely random content due to our choice of using IND-CPA encryption algorithms (see Section III).

Another possible attack is to alter tenant data by switching the values belonging to different columns of the same row. To prevent this attack our solution computes the Bloom filter over a concatenation of the value $v_{r,c}$ and the label associated with its column a_c, as shown in Equation (2). Let us consider a table with two columns c_1 and c_2, and two values stored in the same row $v_{r,c1}$ and $v_{r,c2}$. The Bloom filter associated with the row is calculated at insertion time by the tenant as the OR of the Bloom filters $\mathcal{B}(a_{c1}|v_{r,c1})$ and $\mathcal{B}(a_{c2}|v_{r,c2})$. We assume that the cloud provider switches the two values. When the tenant requests any of the two values, he executes integrity checks as described in Equation (3) by computing the Bloom filters $\mathcal{B}(a_{c1}|v_{r,c2})$ and $\mathcal{B}(a_{c2}|v_{r,c1})$. Since these Bloom filters are different from those calculated at insertion time, the tenant can detect any unauthorized modifications.

B. *False positives*

A drawback of Bloom filters is that integrity checks may result in false positives and allow the cloud provider to modify the tenant database without being detected. By referring to Equation (3), it is always true that if this equation is not satisfied, then the value $v_{r,c}$ has been modified. But there is a small probability that a modified $v_{r,c}$ still satisfies this equation, thus making it impossible for the tenant to detect the unauthorized alteration. The probability of having false positives depends on the parameters used to build the Bloom filter b_r. Bigger Bloom filters guarantee lower false positive rates but may cause greater overheads in terms of storage and network usages, and consequently lower performance and increased costs of the cloud database service [17].

We propose methodologies that allow the tenant to size the Bloom filters stored in his cloud database to choose a trade-off between an acceptable false positive rate and resources overheads. To this aim, we define the *acceptable false positive rate* ε as the highest false positive probability that a cloud tenant is willing to accept. For example, if the cloud tenant deems $\varepsilon = 0.01$ acceptable, then the cloud provider has only 1 chance out of 100 to modify a value without being detected. We also observe that the probabilities of detecting modifications of different values are independent of each other. Hence, if the cloud provider tampers with t different values of the database, the probability of not being detected can be computed as ε^t. In the previous example, if $\varepsilon = 0.01$ and the cloud provider modifies three values, the probability of not being detected drops to 10^{-6}. Our methodologies take as input an acceptable false positive rate chosen by the cloud tenant and allow the cloud tenant to size the Bloom filters.

Let us describe which parameters influence Bloom filters false positive rates. By using the notation established in the literature [10], we define m as the length of the Bloom filter expressed in bits, k as the number of hash functions used for the computation of the Bloom filter, n as the number of elements inserted in the Bloom filter and p as the false positive rate. The false positive rate p can be estimated as the probability function $\mathcal{P}(m, n, k)$ through the following equation [16]:

$$p = \mathcal{P}(m, n, k) = \left[1 - \left(1 - \frac{1}{m}\right)^{kn}\right]^k \approx (1 - e^{-kn/m})^k \tag{4}$$

We assume that the parameters m and k of the Bloom filter are defined at database design time and remain constant afterwards. Moreover, we define as $\bar{k}$ the number of hash functions that minimizes p. We can obtain $\bar{k}$ through the following equation [16]:

$$\bar{k} = \frac{m}{n} \cdot \ln(2) \tag{5}$$

Hence, we can define the optimum false positive rate $\bar{p}$ as:

$$\bar{p} = \bar{\mathcal{P}}(m, n) = \mathcal{P}(m, n, \bar{k}) \approx (1 - e^{-\bar{k}n/m})^{\bar{k}} \tag{6}$$

By substituting Equation (5) in Equation (6), we obtain:

$$\bar{p} \approx (1/2)^{\frac{m}{n} ln2} = e^{-\frac{m}{n} ln(2)^2} \tag{7}$$

Our methodologies translate the tenant acceptable false positive rate ε to an upper bound for the false positive probability $\bar{p}$ and for the number of elements n inserted in the Bloom filter, and provide as outputs the size of the Bloom filters m and the number of the optimal hash functions $\bar{k}$. The tenant should choose the methodology by considering the type of SQL operations that are issued to the database. In particular, we distinguish two scenarios: a database on which values are only created, read, and deleted (CRD); a database on which values may be created, read, updated, and deleted (CRUD).

In the CRD scenario, the false positive rate $\bar{p}$ depends only on the parameters m and n used to build the Bloom filter, and does not change during the lifetime of the database. We observe that n is equal to the number of columns C belonging to a table. In this scenario the tenant is interested in using the smallest possible Bloom filter that satisfies the upper bound on the acceptable false positive rate, thus reducing storage overhead. We define m_{min} as the optimal value for m in the CRD scenario such as:

$$m_{min} = min\{m \in \mathbb{N} : \bar{\mathcal{P}}(m, C) < \varepsilon\} \tag{8}$$

The tenant can compute m_{min} by using the following equation:

$$m_{min} = \left\lceil -\frac{C \cdot ln(\varepsilon)}{ln(2)^2} \right\rceil \tag{9}$$

that is the inverse of the Equation (7) in which $\bar{p} = \varepsilon$ and $n = C$. Then, the cloud tenant can compute $\bar{k}$ by substituting m_{min} and C to m and n in Equation (5).

In the CRUD scenario, we also need to handle update operations that represent authorized modifications of tenant data that are already stored in the cloud database. Update operations can be executed according to two strategies: *Always renew* and *Greedy renew*.

Always renew. Whenever a value needs to be updated, the tenant also retrieves all the other values of the same row and recomputes the corresponding encrypted Bloom filter. In this strategy the value of n does not change over the lifetime of a Bloom filter, hence the false positive rate p remains constant and the computation of m_{min} falls back to Equation (9) of the CRD scenario. However, the tenant has to retrieve unnecessary values every time that a single value is updated, thus incurring in a higher network overhead.

Greedy renew. The tenant builds the Bloom filter with a value of $m > m_{min}$. Hence, the false positive rate p remains smaller than the ε chosen by the tenant even after the insertion of new values in the Bloom filter. The tenant can then update a value by adding the new value to the Bloom filter, without the need to retrieve all the other values of the same row. Since the tenant does not need to download unnecessary values, the network overhead is minimized. However, the value of n increases over the Bloom filter lifetime, thus causing an increase of p. To satisfy the acceptable false positive requirement imposed by the tenant, it is necessary to renew the Bloom filter before p exceeds the maximum acceptable false positive rate ε. We assume that the tenant knows how many values have been updated for each row, hence he also knows the false positive rate p associated with each row of the database. Whenever a new update would cause p to exceed ε, the tenant has to renew the Bloom filter, thus restoring p to its original value. We define u as the maximum number of values that the tenant can update while keeping $p < \varepsilon$.

We now propose a method that the tenant can use to compute u as a function of C, ε and m. A fresh Bloom filter corresponding to a row already includes C values, hence the tenant can perform exactly $u = n_{max} - C$ updates before executing a greedy renew. The tenant can compute n_{max} using the following equation:

$$n_{max} = \left\lfloor -\frac{m \cdot ln(2)^2}{ln(\varepsilon)} \right\rfloor \tag{10}$$

that is the inverse of the Equation (7) in which $\bar{p} = \varepsilon$. Then, the cloud tenant can compute $\bar{k}$ by substituting n_{max} to n in the Equation (5).

The only other parameter that the tenant has to define to build the Bloom filter is m. A lower bound for m is represented by m_{min}, as computed from Equation (9). In particular, for $m = m_{min}$, the tenant has to perform a *greedy renew* for every update, thus falling back to the *always renew* strategy. As an example, in this scenario the cloud tenant can estimate a value of m_{min} by multiplying C by a factor of 10 for an acceptable false positive rate ε of 0.01 [16]. On the other hand, higher values of m reduce the network overhead for update operations, but increase both storage and network overheads in the case of select and insert operations. The choice of the best value for m depends on the acceptable false positive rate, the workload, the database structure, and on the trade-off between storage and network overheads. In the following section we propose extensive performance analyses and we demonstrate that our proposal leads to a good trade-off between storage and network overheads in realistic workloads.

V. Performance analysis

To analyze the performance of the proposed solution we compare its storage and network overheads to those of other two solutions for the integrity of outsourced databases that are proposed in literature and commonly adopted in practice.

The first solution is to extend the approaches that are devised for remote file storage [11] to the cloud database scenario. The integrity of files stored in the cloud is usually achieved by associating to each file a HMAC computed through standard hash functions, such as SHA-256. This approach can be trivially applied to the cloud database scenario by computing a HMAC for each value stored in the database. This solution causes a high storage overhead because a 256-bit HMAC is bigger than many primitive data types. Its main benefit is that the cloud tenant can verify the integrity of a value without having to retrieve other unnecessary values from the remote database. In the performance evaluation we refer to this solution as *VLH* (value-level HMAC).

The second solution is an optimization of VLH for the cloud database context that has been proposed in [8], [9]. The main idea is to reduce the storage overhead by associating a control structure (such as a SHA-256 or a HMAC) to a set of values, rather than to a single value. In the cloud database context, a HMAC is computed over all the values that belong to the same row. The resulting scheme has the same structure of Table I, but the last column is used to store a HMAC rather than an encrypted Bloom filter. In the following we refer to this solution as *TLH* (tuple-level HMAC). This approach has a clear benefit in terms of storage overhead with respect to the VLH solution. However, whenever the tenant wishes to verify the integrity of a single value, he has to retrieve also all the other values stored in the same row to compute the HMAC. The need to retrieve unnecessary values causes an increase in the network overhead.

To compare performance of our approach against those of VLH and TLH we need to pick realistic values for the acceptable false positive rate (ε) and for the size of the Bloom filter (m). In this evaluation, we assume that $\varepsilon = 0.01$ is an acceptable value for the tenant. We estimate m for the greedy renew strategy by multiplying the number of columns C by a factor of 20. Storage overhead introduced in the greedy renew strategy may double that of the always renew strategy. However, it allows the cloud tenant to greatly reduce the network overhead. The computation of the optimal m is left as a future work. In the following we refer to the proposed solution as *EBF* (encrypted Bloom filter).

We first analyze the performance of different SQL queries by referring to a synthetic example, and then we analyze the storage and network overheads of a realistic scenario by referring to the TPC-C workload [6].

A. Single operations workload

In this analysis we refer to a database table in which each row is 500 bytes long, and contains 10 values. For the sake of simplicity, we assume that all values have the same size, and we choose $m = 192$. Since we encrypt the Bloom filter using a standard Blowfish 64-bit cypher [12] with a 64-bit initialization vector, the resulting encrypted Bloom filter is $192 + 64 =$

256-bit long. We highlight that this is the same size of an HMAC computed using SHA-256. By using Equation (10) we can compute $n_{max} = 20$. Moreover, through Equation (5) we compute that the optimal number of hash function to build the Bloom filter in this scenario is $\bar{k} = 7$. Hence, the Bloom filter needs to be refreshed after the insertion of 20 values. In the proposed example, $C = 10$, hence it is possible to perform $u = n_{max} - C = 10$ updates to values stored in the same row without having to renew the corresponding Bloom filter. By using Equation (6), we can estimate that the false positive rate p for $n = C = 10$ is 0.000248, and for $n = n_{max} = 20$ is $0.00997 < \varepsilon = 0.01$. In this example, the storage overhead introduced by EBF is exactly the same of TLH, and 35.12% lower than VLH. We now investigate the network overhead introduced by the four basic CRUD operations.

CREATE. It is an INSERT SQL operation in which the tenant creates a new row in a table. In this scenario, all the values of the row and all the corresponding control structures (HMACs for VLH and TLH, encrypted Bloom filters for EBF) have to be transmitted from the client to the cloud database. The network bandwidth consumed by EBF is the same of TLH, while there is a 35.12% reduction with respect to VLH.

READ. It is a SELECT SQL operation used by the tenant to access values stored in the cloud database. Table II summarizes the number of bytes that need to be downloaded by the tenant for different strategies and types of SELECT queries. The first row shows the bytes downloaded in the case of a SELECT query in which the tenant needs only one value. The performance of EBF and VLH are optimal, since the tenant only needs to retrieve the needed value and the associated control structure. The other corner case is represented by SELECT queries used to read all the values of a row. In this case the performance of EBF and TLH are optimal, while VLH incurs in a high network overhead since the tenant has to retrieve one control structure for each value. If the SELECT query is used to access only a subset of data, then EBF has a clear advantage over both TLH and VLH, as shown by the second row of Table II in which the tenant needs 5 of the 10 values that compose a row.

SELECT	TLH	VLH	EBF	$\frac{EBF}{TLH} - 1$	$\frac{EBF}{VLH} - 1$
One value	532	82	82	82.59%	0.00%
Half values	532	410	282	46.99%	31.22%
All values	532	820	532	0.00%	35.12%

TABLE II: Bytes transmitted and overheads of SELECT queries

UPDATE. It is an UPDATE SQL operation, that the tenant uses to modify one or more values belonging to the same row. As already shown for READ operations, if TLH is used the tenant always has to read all the values of the row independently of the number of values that are actually modified, because he has to compute the new HMAC. On the other hand, if VLH is used, the tenant has to upload a new HMAC for each value. EBF reduces the network overhead since the tenant does not need to download useless values and always uploads one control structure. However, we must take into account that, since $u = 10$, after ten updates it is necessary to renew the Bloom filter. This operation requires the tenant to download all the values that belong to the same row. Table III shows how many bytes have to be transmitted to perform UPDATE queries using TLH, VLH and EBF strategies. If updates always modify a single value, the worst scheme is TLH. EBF performs better than TLH (73.07%) but worse than VLH (-74.72%), because EBF has to periodically renew the encrypted Bloom filter. If updates always modify all values in the row, then TLH and EBF have the better network usage reduction (35.12%). Finally, if update queries modify a subset of values, EBF outperforms both VLH and TLH.

UPDATE	TLH	VLH	EBF	$\frac{EBF}{TLH} - 1$	$\frac{EBF}{VLH} - 1$
One value	532	82	143	73.07%	-74.72%
Half values	532	410	325	38.89%	20.71%
All values	532	820	532	0.00%	35.12%

TABLE III: Bytes transmitted and overheads of UPDATE operations

DELETE. It does not involve data transfer, so there are no differences in using TLH, VLH or EBF.

B. Mixed operations of the TPC-C workload

To analyze how EBF performs in real-world scenarios, we consider the query distribution of TPC-C [6], that is a standard OLTP benchmark for databases. To determine proper sizes for the Bloom filters we analyze the structure of all the tables of the TPC-C database. For each table we set a value for m that allows to modify a number of values equal or greater than the number of columns. As an example, let us consider the table *history* that contains 8 columns. For this table we choose a value for m that makes it possible to update at least 8 values. In this example, m is set to 192 bits, thus leading to a $192+64 = 256$-bit encrypted Bloom filter for all the rows of the table. We then compute the tuple size in the following configurations: no integrity (that is, the plain TPC-C database scheme), tuple-level HMAC (TLH), value-level HMAC (VLH) and encrypted Bloom filter (EBF). These values are summarized in Table IV. By computing a synthetic storage overhead for all the TPC-C tables we have that storage needs for EBF are slightly higher than TLH (+4%), and much smaller than VLH (-59%).

To assess network overheads we consider the average traffic generated by TPC-C. We take into account all queries and weigh their network usage according to their probability distribution. The results are shown in Table V. The network usage of INSERT queries is only influenced by the size of the control structures used to check data integrity. Hence, EBF has a slight advantage with respect to TLH (network consumption is 7.5% lower), but a huge advantage with respect to VLH (75% lower). In the case of SELECT operations, traffic is dominated by almost complete row transfers of relatively large tuples; in particular, SELECT operations that request one attribute are infrequent and their traffic contribution is low. As a result, EBF performs better than TLH (by 27%) and VLH (by 42%). For UPDATE operations EBF reduces the amount of network traffic by 72% with respect to TLH, while causing a 50% increment with respect to VLH. If we consider the total network traffic by aggregating all results, our solution

Table		warehouse	district	item	customer	history	stock	orders	new_orders	order_line
Number of values		9	11	5	21	8	17	8	3	10
Allowed updates ($\varepsilon = 1\%$)		11	9	8	19	12	23	12	3	10
EBF size (bytes)		32	32	24	56	32	56	32	16	32
Tuple size (bytes)	VLH	421	491	272	1415	304	950	284	104	384
	TLH	165	171	144	775	80	438	60	40	96
	EBF	165	171	136	799	80	462	60	24	96
Storage Reduction (EBF vs)	VLH	61%	65%	50%	44%	74%	51%	79%	77%	75%
	TLH	0%	0%	0%	6%	-3%	0%	-5%	40%	0%

TABLE IV: Analysis of the tables of a TPC-C compliant database

Traffic	INSERT	SELECT	UPDATE	TOTAL
TLH	95.60	1713.59	1122.27	2928.74
VLH	350.32	2166.01	209.32	2715.89
EBF	88.40	1252.05	313.16	1671.09
EBF/TLH-1	7.5%	27%	72%	43%
EBF/VLH-1	75%	42%	50%	38%

TABLE V: Network consumption (bytes) and overheads of the TPC-C workload

has an average traffic reduction of 38% against VLH, and 43% against TLH. In summary, EBF outperforms any other integrity solution whenever the distribution of database operations is similar to those outlined in TPC-C.

VI. Conclusions

Public cloud databases are appealing services that allow companies to outsource data management infrastructures, but their adoption is hindered by concerns about confidentiality and integrity of information managed by a third subject.

We propose a novel strategy that allows cloud tenants to detect unauthorized modifications to data outsourced to untrusted cloud providers. This solution is based on encrypted Bloom filters. We demonstrate that its storage overhead is comparable or smaller than other solutions for database integrity and its network overhead in realistic usage scenarios is consistently smaller. Our analysis also demonstrates that encrypted Bloom filters are attractive especially in the case of metered network traffic, that is very common in current commercial cloud storage offers. The proposed solution allows the tenant to tune the trade off between the probability of detecting unauthorized data modifications and storage and network overheads. A formal analysis of the best choice of the parameters on the basis of tenant preferences is left to future works.

References

[1] W. Jansen and T. Grance, "Guidelines on security and privacy in public cloud computing," Tech. Rep. NIST Special Publication 800-144, 2011.

[2] S. A. Almulla and C. Y. Yeun, "Cloud computing security management," in *Proc. Second IEEE Int'l Conf. Engineering Systems Management and Applications*, Mar.-Apr. 2010.

[3] L. Ferretti, M. Colajanni, and M. Marchetti, "Distributed, concurrent, and independent access to encrypted cloud databases," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 2, 2014.

[4] ——, "Access control enforcement of query-aware encrypted cloud databases," in *Proc. Fifth IEEE Int'l Conf. on Cloud Computing Technology and Science*, Dec. 2013.

[5] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan, "CryptDB: protecting confidentiality with encrypted query processing," in *Proc. 23rd ACM Symp. Operating Systems Principles*, Oct. 2011.

[6] TPC-C, "Transaction processing performance council," http://www.tpc.org, Apr. 2014.

[7] H. Hacigümüş, B. Iyer, and S. Mehrotra, "Providing database as a service," in *Proc. 18th IEEE Int'l Conf. Data Engineering*, Feb. 2002.

[8] E. Mykletun, M. Narasimha, and G. Tsudik, "Authentication and integrity in outsourced databases," *ACM Transactions on Storage*, vol. 2, no. 2, 2006.

[9] R. Accorsi and S. Ranise, "Rdas: A symmetric key scheme for authenticated query processing in outsourced databases," in *Proc. Ninth Int'l Work. Security and Trust Management*, Sep. 2013.

[10] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Comm. of the ACM*, vol. 13, no. 7, 1970.

[11] T. Adtiya, P. Baruah, and R. Mukkamaka, "Space-efficient bloom filter for enforcing integrity of outsourced data in cloud environments," in *Proc. Fourth IEEE Int'l Conf. Cloud Computing*, Jul. 2011.

[12] B. Schneier, "Description of a new variable-length key, 64-bit block cipher (blowfish)," in *Proc. Cambridge Security Work. Fast Software Encryption*, Dec. 1993.

[13] J. Daemen and V. Rijmen, *The design of Rijndael: AES – the advanced encryption standard*. Springer, 2002.

[14] C. Blundo, A. Cresti, and U. Vaccaro, "Key distribution schemes," in *Proc. 1994 IEEE Int'l Symp. Information Theory*, Jun.-Jul. 1994.

[15] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati, "Over-encryption: management of access control evolution on outsourced data," in *Proc. 33rd Int'l Conf. Very Large Data Bases*, Sept. 2007.

[16] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," *Internet Mathematics*, vol. 1, no. 4, 2004.

[17] L. Ferretti, F. Pierazzi, M. Colajanni, and M. Marchetti, "Performance and cost evaluation of an adaptive encryption architecture for cloud database services," *IEEE Trans. on Cloud Computing*, vol. Preprint, 2014, Doi: 10.1109/TCC.2014.2314644.