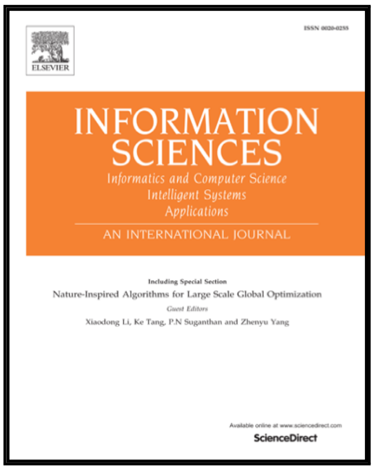
Accepted Manuscript



A symmetric cryptographic scheme for data integrity verification in cloud databases

Luca Ferretti, Mirco Marchetti, Mauro Andreolini, Michele Colajanni

PII: S0020-0255(17)30473-5
DOI: 10.1016/j.ins.2017.09.033
Reference: INS 13137

To appear in: *Information Sciences*

Received date: 14 February 2017
Revised date: 8 September 2017
Accepted date: 10 September 2017

Please cite this article as: Luca Ferretti, Mirco Marchetti, Mauro Andreolini, Michele Colajanni, A symmetric cryptographic scheme for data integrity verification in cloud databases, *Information Sciences* (2017), doi: 10.1016/j.ins.2017.09.033

A symmetric cryptographic scheme for data integrity verification in cloud databases

Luca Ferretti[a,*], Mirco Marchetti[a], Mauro Andreolini[b], Michele Colajanni[a]

[a] *University of Modena and Reggio Emilia, Department of Engineering "Enzo Ferrari"*
[b] *University of Modena and Reggio Emilia, Department of Physics, Informatics and Mathematics*

Abstract

Cloud database services represent a great opportunity for companies and organizations in terms of management and cost savings. However, outsourcing private data to external providers leads to risks of confidentiality and integrity violations. We propose an original solution based on encrypted Bloom filters that addresses the latter problem by allowing a cloud service user to detect unauthorized modifications to his outsourced data. Moreover, we propose an original analytical model that can be used to minimize storage and network overhead depending on the database structure and workload. We assess the effectiveness of the proposal as well as its performance improvements with respect to existing solutions by evaluating storage and network costs through micro-benchmarks and the TPC-C workload standard.

Keywords: authentication, integrity, Bloom filter, MAC, cloud, database

1. Introduction

Cloud database services represent an important opportunity for many enterprises and organizations attracted by high availability and scalability guarantees. However, their adoption is limited by the perceived risks about data confidentiality and integrity that can be violated by insiders and external attackers [32]. In this paper we consider the problem of data integrity in cloud databases, and we propose an efficient solution to assess the integrity of outsourced data while minimizing network and storage overhead.

We should consider that cloud service contracts do not oblige providers to notify tenants about data corruption, hence verifying data integrity remains a tenant's burden. Existing verification solutions are affected by prohibitive computational, storage and bandwidth overhead that have an impact on costs because additional network and storage usage increases cloud service expenses [22]. We propose a novel solution for integrity verification that allows a tenant to efficiently detect unauthorized modifications on outsourced data at reduced storage and network overhead. The contribution of this paper is threefold:

- it defines a novel integrity protocol and analyzes the guarantees that it offers;
- it proposes an analytical model that allows the tenant to optimize the parameters of the proposed integrity protocol with the goal of minimizing storage and network overhead as a function of database workload characteristics;
- it evaluates the performance of the proposed protocol through micro-benchmarks and the TPC-C standard database benchmark.

The literature on database outsourcing faces three types of correctness guarantees: *integrity*, *completeness* and *freshness* [10, 30, 46, 50]. Completeness and integrity are satisfied if the result of a query includes all and only the relevant data that an authorized party inserted in the database, thus guaranteeing that tenant's data is not altered, deleted or ignored by the outsourced database. Freshness ensures that a client receives the latest version of the requested data. This paper focuses on integrity and proposes a novel scheme that allows efficient detection of unauthorized modifications of data stored in cloud databases.

Existing solutions to guarantee data integrity leverage cryptographic digests that can be based on asymmetric or symmetric primitives. Asymmetric cryptographic accumulators [7, 39] ensure optimal asymptotic complexity in storage, computation, and network usage [35], but their excessive computational costs prevent their adoption in most database scenarios [15]. Symmetric Message Authentication Codes (MAC) can guarantee the integrity of tenant files stored in the cloud [2], but their adoption in cloud database services poses several challenges related to the granularity of the protected data: if every value is protected by a MAC, the database storage increases considerably; if the entire set of rows/tables are protected by one MAC, then the verification of each value requires to fetch an entire row/table, thus imposing an excessive network overhead that renders verification unfeasible.

The proposed scheme relies on a variant of Bloom fil-

*Corresponding author
Email addresses: luca.ferretti@unimore.it (Luca Ferretti), mirco.marchetti@unimore.it (Mirco Marchetti), mauro.andreolini@unimore.it (Mauro Andreolini), michele.colajanni@unimore.it (Michele Colajanni)

ters [8] for the detection of unauthorized modifications to outsourced data. This choice guarantees two benefits: the processes of integrity verification and update of the authentication structures cause low network and storage overhead; all cryptographic operations are based on symmetric schemes that do not introduce significant computation overhead. The performance of the proposed scheme depends on Bloom filter sizes that should be chosen on the basis of the integrity guarantees required by the tenant. We enrich the proposal by offering an analytical model that takes as its input the integrity requirement, the database characteristics and workload, and evaluates the optimal size of the Bloom filters that minimize overheads and related cloud service costs.

Our proposal is orthogonal to data encryption strategies for data confidentiality proposed in literature [20, 21, 23, 43], and can be integrated with encryption algorithms to achieve both confidentiality and integrity of data stored in cloud databases. Moreover, it can be used to design solutions that aim to ensure data completeness and freshness [47].

To the best of our knowledge, the proposed scheme is the most efficient solution for detecting unauthorized modifications in cloud database services. We demonstrate its benefits through micro-benchmarks and the standard TPC-C benchmark. All results show that the proposed scheme greatly reduces network and storage footprint with respect to existing proposals.

The remaining part of the paper is organized as follows. Section 2 describes the considered cloud database scenario and the threat model. Section 3 outlines theoretical background on Bloom filters. Section 4 presents the proposed solution and its security guarantees. Section 5 describes how the tenant must size the protocol parameters to achieve the required security level. Section 6 presents the analytical method to minimize overhead. Section 7 discusses performance in terms of storage and network overhead, and compare results against state-of-the-art schemes. Section 8 compares our solution with existing proposals. Finally, Section 9 concludes the paper by summarizing its main contributions and future work. For further implementation details, security analyses and proofs please refer to the Appendices.

2. Scenario and threat model

We consider a scenario where a *tenant* stores large amounts of data into a cloud database through clients that issue read and write operations. The tenant benefits from pay-per-use cloud prices with the goal of reducing his operational costs, but his data at rest, in motion and in use are subject to different security threats.

Our proposal aims to improve security of data in use against internal malicious attackers, and can be combined with additional solutions to protect other attack surfaces. For example, clients and database servers should adopt the SSL/TLS protocol suite to protect confidentiality and integrity of data in motion as well as the ability to detect replay and reflection attacks. The cloud provider must own a valid PKI certificate that avoids man-in-the-middle attacks. All clients must own valid credentials (e.g., API tokens, client-side PKI certificates) that allow the provider to identify and authenticate them, as well as grant proper access on the resources stored in the database.

In the proposed scheme, we assume that all clients share the same secret key, that can be distributed according to known key distribution schemes [9], as well through more efficient strategies that are specific to the field of cloud database services [16, 20]. We assume that the secret key is not known by the cloud provider nor by any other part that is not authorized to manipulate tenant data. We assume that all clients are trusted and will never send corrupted data to the cloud database, nor will leak confidential information to unauthorized parties (including the cloud provider). On the other hand, we assume that the cloud provider is not trusted and could alter tenant's data. Modification may be caused by hardware or software failures, as well as by deliberate attacks coming from external adversaries or from insiders within the cloud organization. From the tenant's point of view, any unauthorized modification represents a data integrity violation.

Data integrity and authenticity is a prominent research area of the cryptography community, and is usually guaranteed by means of message authentication codes (MAC) [5]. A MAC applied to arbitrary data together with a secret key produces a *(cryptographic) digest* (also called *tag*), that is a compressed representation of the input data. An attacker can violate integrity by forging a digest on behalf of the authorized users. The security level of a MAC depends on the key and digest sizes. The key size determines the security level against off-line brute-force attacks, in which the attacker knows some plaintext data and the corresponding digest, and tries to guess the key. Key length should always comply with standards recommendations [4]. The digest size, that is constant with regard to the size of the input, determines the probability of guessing a valid digest. Since only an authorized party can verify a digest, this attack can only be executed by participating to the secure protocol and interacting with the authorized parties.

The security threats against the integrity of data outsourced to a remote database are peculiar to this context. In other scenarios, such as protocols for secure communications, protecting data integrity requires scheme that withstand *adaptive attacks*, where attackers can adopt a trial-and-error strategy and attempt a large number of unsuccessful attacks. On the other hand, in the outsourced database scenario, the attackers cannot access the verification function because there is no reason to expose this service. As a consequence, one unsuccessful attack is sufficient for the tenant to detect an integrity violation. Hence, for this scenario it is sufficient that integrity solutions withstand *non-adaptive attacks*.

Among all proposed security models, our scheme must protect data against the so called *chosen ciphertext attacks (CCA)* assuming that an attacker tries to forge a ciphertext corresponding to a valid plaintext. The literature distinguishes between *non-adaptively chosen ciphertext attacks (CCA1)* and *adaptively chosen ciphertext attacks (CCA2)* depending on attacker capabilities. Cryptographic schemes are *CCA1-secure* or *CCA2-secure* if they are resistant against CCA1 and CCA2 attack models, respectively. We design a novel cryptographic scheme that ensures integrity of data outsourced to a cloud provider by allowing a tenant to detect whether data has been modified without authorization. The adopted scheme is robust against CCA1 attacks. While it is possible to improve its security guarantees, the proposed scheme should not be used to protect integrity in scenarios requiring CCA2-security guarantees.

The proposed protocol guarantees data integrity without incurring in excessive overheads affecting state-of-the-art schemes. For this reason, it perfectly suits the cloud scenario where network and storage overheads are costly metered resources.

3. Theoretical background

Besides the attack models and cryptographic protocols discussed in the previous Section 2, the main theoretical foundations used in this paper are related to *Bloom filters* [8, 11, 34], that are space-efficient data structures for representing sets. A Bloom filter (BF) can be described as an array of m bits (a *bit string*) built using k hash functions. Each hash function maps an arbitrary long string to an integer within the range $[m] = 0, \ldots, m-1$. To insert an element in the BF, we compute the hash functions of the element and set to one the corresponding bits in the BF bit string. A BF supports membership queries that allow false positives, in the sense that a query may return a positive answer even if an element is not stored in the Bloom filter (BF). As an advantage, queries never return false negatives. To execute a membership query for a given element, we compute the hash functions of the element and verify if all the corresponding bits in the BF bit string are set to one. A false positive occurs if the bits are equal to one even if the element was not inserted in the BF.

The probability of getting a false positive, namely the *false positive rate*, depends on the size m of the BF, on the number of hash functions k and on the number of values n stored in the BF. The false positive rate function $f(\cdot)$ can be computed as following:

$$f(m,n,k) = \left[1 - \left(1 - \frac{1}{m}\right)^{kn}\right]^k \approx (1 - e^{-kn/m})^k \quad (1)$$

The optimal number of hash functions $\bar{k}$ is the value of k that minimizes the false positive rate, and can be computed in terms of m and n [11]:

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

Moreover, the optimal false positive rate $\bar{f}(m,n) = f(m,n,\bar{k})$ can be computed as following:

$$\bar{f}(m,n) \approx (1 - e^{-\bar{k}n/m})^{\bar{k}} = 2^{-\bar{k}} = e^{-\frac{m}{n}\ln(2)^2} \quad (3)$$

Note that if the optimal amount of hash functions $k = \bar{k}$ is used, then the estimated amount of bits set to one is half the size of the BF and the bit string created by the BF function resembles a randomly generated string [11, 33, 34].

4. Protocol design

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

Table 1: Database table enriched with a column containing cryptographic digests.

We describe the solution based on encrypted BF to guarantee integrity of data outsourced in outsourced databases while minimizing computational, storage and network overhead. Let us consider a table having R rows and C columns. We denote as $v_{r,c}$ the value stored in the r-th row and c-th column of the table, where $r = [1, \ldots, R]$ and $c = [1, \ldots, C]$. Similarly, V_r is defined as the set of all values that belong to the r-th row, that is $(v_{r,1}, \ldots, v_{r,C})$.

We add to all database tables a new column storing a short control structure, namely a *cryptographic digest* (or just *digest*), that allows the tenant to verify the integrity of all the data stored in the corresponding row. The notation e_r identifies the digest associated to the r-th row of a table. Table 1 shows the modified database schema, where a_c is the name of column c. We assume that the tenant database administrator generates a secret key and distributes it to authorized clients and that the algorithms used in the protocol are public. We now describe how an authorized client executes *insert*, *read* and *update* operations.

4.1. Insert operation

An authorized client issues an insert operation by sending a tuple of values V_r and the associated digest e_r to the cloud database. The client computes e_r according to the following equation:

$$e_r = \mathcal{E}_{sk}(iv, b_r), \quad (4)$$

where $\mathcal{E}_{sk}(\cdot)$ is an *IND-CPA*-secure symmetric encryption algorithm, sk is the secret key and iv is the random initialization vector. The encryption algorithm takes as input the value b_r, that is a bit string computed as following:

$$b_r = \bigvee_{c=1}^{C} \mathcal{B}_\tau^m(a_c, v_{r,c}), \tag{5}$$

where a_c is the label of the column associated to the value $v_{r,c}$, $\bigvee$ is the bitwise *OR* operator and $\mathcal{B}_\tau^m(\cdot)$ is a BF function that outputs a bit string of size m that is independent of the size of the input values. The function $\mathcal{B}_\tau^m(\cdot)$ is computed by using keyed hash functions (e.g., HMAC algorithms [5]) that accept multiple inputs [36] and the secret key τ. For the sake of clarity in the rest of the paper we will refer to function $\mathcal{B}_\tau^m(\cdot)$ and to its outputs as *secret BF function* and *secret BFs*, respectively. As an example, we say that the value b_r is the *secret BF* associated to the row r and, as expressed by equation (5), that it stores the set of elements $\{(a_c, v_{r,c})\}_{c\in[1,\dots,C]}$. We propose a candidate implementation in Appendix A.

The use of keyed hash functions is uncommon, since BFs are usually implemented through public hash functions. However, in our protocol this is not a viable option. Let us consider an adversary that tries to insert a fake value in the row r. If BFs are based on public hash functions, an attacker that knows V_r can compute b_r and randomly generate fake values until a false positive is obtained. The average amount of trials that are necessary to find a valid fake value depends on the false positive rate of the BF and not on the security level of the encryption function $\mathcal{E}_{sk}(\cdot)$.

We observe that any data given in input to the function should be binary encoded as it is typical for most protocols in symmetric cryptography. The proposed protocol also supports missing (null) values inserted in the database. Whenever a client inserts a null value in the database, he inserts an encoding convention in the corresponding digest. All encoding functions and conventions are public and known to all the parties. They are also used in the verification phase.

4.2. Read operations

Let us assume that the client wants to retrieve one value $v_{r,c}$ from the cloud database and to verify its integrity. This can be accomplished by fetching the required value together with the corresponding digest e_r through a select query. We note that there is no need to retrieve all the other values of the row r because the BF supports efficient set membership. The client then decrypts e_r using sk as decryption key and obtains b_r. Now he can execute a membership test of the value $v_{r,c}$ on b_r by means of the secret key τ (see Appendix A for a candidate implementation). If the membership test fails, then an integrity violation has been detected. If it occurs, then one of the following is true:

- the value $v_{r,c}$ has not been tampered with and integrity holds;
- the integrity of $v_{r,c}$ has been compromised, but the membership test returned a false positive.

False positives are a well known drawback of BFs, but they can be limited by a careful choice of the BF parameters. A thorough analysis of the attacks against BFs and of how they can be prevented is proposed in Section 4.4.

We recall that the elements stored in the secret BF are a concatenation of values stored in the database and of the labels associated to their column. This design choice prevents attacks based on columns scrambling. As an example, we consider a table with two columns c_1 and c_2. An authorized client inserts two values $v_{r,c1}$, $v_{r,c2}$ and the digest e_r computed through Equations (5) and (4). The values inserted in the secret BF are obtained by concatenating the labels c_1 and c_2 to the corresponding values $v_{r,c1}$ and $v_{r,c2}$. Now, let us assume that an adversary swaps the two values. When the tenant requests any of the two values, he executes integrity checks. He obtains the values $v_{r,c1}$ and $v_{r,c2}$ for the columns c_2 and c_1, respectively. To verify integrity he concatenates the values with the associated columns and tests the membership of the results in the secret BF. Since the resulting elements are different, he detects an integrity error. This strategy can also be used with databases with complex hierarchical schema by substituting the name of the column with the unique path or identifier used to retrieve the value.

Another threat is represented by attacks based on rows scrambling. As an example, consider a client that issues the following select query: SELECT c_2 FROM *tablename* WHERE $c_1 = x$. Let us assume that only row $r2$ satisfies the WHERE clause, hence a correct cloud provider should return $v_{r2,c2}$ together with e_2. An adversary cloud provider may reply with a value of column c_2, but belonging to a row that does not satisfy the WHERE clause. Suppose that the cloud provider replies with $v_{r1,c2}$ and e_1. Our protocol is able to detect this incorrect result, because the client verifies that both $(c_2, v_{r1,c2})$ and (c_1, x) are stored in e_1. While the first check succeeds, the second fails, and the client is able to detect the row scrambling attack of the cloud provider. This design choice protects the database against row scrambling for any type of query, and similar examples may be provided for range queries.

4.3. Update operations

When a client issues an update operation to modify one or more values belonging to row r, he must also update the digest e_r. We distinguish two methods to issue update operations: the *always renew* strategy, and the *greedy renew* strategy.

The *always renew* strategy is the simplest one and implies the execution of update operations as insert operations. The client fetches all the row values from the database, even those that are not modified by the update, and computes a digest that stores the updated tuple as described in the previous paragraph, by using Equations (4), (5). The main drawback of this simple update strategy is that the tenant retrieves unnecessary values every time he updates even one value, thus causing high network overhead.

The *greedy renew* method requires the execution of update operations without having to download unnecessary data from the cloud database service, thus reducing the network overhead. In this strategy, the client that updates some values $V_r' \subset V_r$ retrieves only the values V_r' and the associated digest e_r. First, the clients verifies the integrity of the values. Then, it computes the new secret BF b_r' associated to the updated values by decrypting the retrieved digest e_r into b_r and adding the new values. Finally, it re-encrypts the value by using the secret key sk and a new random initialization vector iv'. The update of e_r can be summarized as following:

$$e_r' = \mathcal{E}_{sk}\left(iv', \mathcal{B}_\tau^m(V_r') \vee \mathcal{D}_{sk}(e_r)\right), \tag{6}$$

where $\vee$ denotes the bitwise OR operator.

Here we propose an informal description of the issues that must be addressed when sizing the protocol parameters, while a precise evaluation is proposed in Section 5. The false positive rate of the secret BFs increases after each update operation. Hence, when using the greedy renew strategy the following design choices must be taken into account. The digest must be able to store a number of values higher than the cardinality of columns without affecting security. Moreover, *renew update operations* must be executed periodically before affecting the security guarantees. In particular, since bigger BFs have lower false positive rates, the greedy renew strategy requires to oversize the digest stored in the database. By knowing the maximum amount of values that can be stored in the digests, one can estimate when to execute renew operations. Increasing the size of the BF introduces a trade-off between the storage and network overhead. We propose an analytical methodology to choose the best protocol parameters in Section 6.

4.4. Security analysis

We analyze the security guarantees of the proposed scheme by identifying possible attacks and by evaluating the sizing requirements to defend against them. All security analyses are discussed under the threat model described in Section 2. We distinguish two types of attacks.

Attacks on the plaintext values. The attacker modifies an existing tuple by inserting fake values without modifying the associated digest. In this case the attacker can leverage the BFs false positives by guessing an element whose membership query is answered positively even if the element was not inserted by the tenant.

Attacks on the digests. The attacker generates new digests for existing or new tuples without any knowledge about the secret key. In this case the attacker aims to take advantage of the malleability intrinsic to IND-CPA encryption functions to manipulate the underlying BF.

The straightforward design choice to defend against attacks on the digests is to protect their integrity. This is usually accomplished by attaching a MAC computed on the encrypted digest or by using an authenticated encryption scheme [37] instead of the IND-CPA encryption as described in Section 4.1. Both design strategies increase storage and bandwidth overhead. In Appendix B we demonstrate that in the considered scenario, the attacks on the digests are always less convenient than the attacks on plaintext values. As a result, the protocol parameters must be sized to defend against attacks on plaintext values and there is no need to protect the encrypted BF with a MAC or to use an authenticated encryption scheme.

we recall that the proposed approach does not work when an adversary can execute adaptive attacks (see Section 2), because he could exploit the malleability of the IND-CPA encryption to efficiently forge fake digests.

5. Sizing boundaries

In this section we show how to size the protocol parameters to guarantee the security levels required by the cloud tenant. To this purpose, we define the *acceptable false positive rate* ε as the highest false positive probability that a cloud tenant wants to tolerate. For example, if the cloud tenant deems $\varepsilon = 0.01$ acceptable, then the attacker 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 attacker alters t values of the database, the probability of not being detected is equal to ε^t. In the previous example, if $\varepsilon = 0.01$ and the attacker modifies 3 values, the probability of not being detected drops to 10^{-6}. The proposed model takes as its input the acceptable false positive rate chosen by the cloud tenant and the database workload, and then computes the smallest BF size that still satisfies the constraints on the false positive rate.

Starting from the Equation (3) we propose a model that can be used to compute the lower bound on the BF size satisfying the acceptable false positive rate ε. We distinguish two typical workload scenarios:

- a database in which values are only created, read, and deleted (CRD);
- a database in which values may be created, read, updated, and deleted (CRUD).

As in the CRD scenario there are no updates, only whole rows can be inserted or deleted. Hence, the number of elements n inserted in the BF is always equal to the number of columns c. In this scenario the goal is to minimize storage and network overhead by using the smallest BF that satisfies the upper bound on the acceptable false positive rate. We define $m_{\min}$ as the optimal value for m in the CRD scenario. By using the inverse of Equation (3) with

$\bar{f} = \varepsilon$ and $n = c$, the tenant can compute $m_{\min}$ as following:

$$m_{\min} = \left\lceil -\frac{c \cdot \ln(\varepsilon)}{\ln(2)^2} \right\rceil \tag{7}$$

Then, the cloud tenant can compute $\bar{k}$ by substituting $m_{\min}$ and c for m and n in Equation (2).

In the CRUD scenario, we also need to handle update operations that represent authorized modifications of tenant data stored in the cloud database. As discussed in Section 4.3, update operations can be executed according to two strategies: *always renew* and *greedy renew*.

5.1. Always renew

When a value needs to be updated in the always renew case, the tenant retrieves all the other values of the same row and recomputes the corresponding encrypted BF. Since the BF for a row is rebuilt from scratch after any update query, it always contains a number of values (n) that is equal to the number of columns of the table (c). As the value of n does not change over the lifetime of a BF, the false positive rate f remains constant and the computation of $m_{\min}$ falls back to Equation (7).

5.2. Greedy renew

The main goal of the greedy renew strategy is to execute updates without having to *always renew* the digest, thus reducing network overhead. If we want to perform several updates while avoiding that the false positive rate f exceeds the acceptable false positive rate ε, we need to accommodate for a larger BF, having $m > m_{\min}$. After inserting a row, the number of values stored in the BF n is equal to the number of columns c. Whenever a tenant updates a value, he also needs to add the new value to the BF attached to the row. This implies that n increases over the BF lifetime, thus causing an increase of f that may eventually become higher than ε. To avoid that an update causes $f > \varepsilon$, the tenant has to renew the BF to reset f to its original value. We define u as the maximum number of values that the tenant can update while keeping $f \leqslant \varepsilon$, and we propose a method that the tenant can use to compute u as a function of c, ε and m.

A fresh BF includes c values and, by definition, can tolerate up to u insertions before needing a renew. Thus, the maximum number of elements that can be inserted is $n_{\max} = c + u$. By using the inverse of Equation (3), where $\bar{f} = \varepsilon$, the tenant can compute $n_{\max}$ as follows:

$$n_{\max} = \left\lfloor -\frac{m \cdot \ln(2)^2}{\ln(\varepsilon)} \right\rfloor \tag{8}$$

Then, the tenant can compute $\bar{k}$ by substituting $n_{\max}$ to n in Equation (2). The number of updates left until the greedy renew is $u = n_{\max} - c$. Through Equation (8) we obtain:

$$u = \left\lfloor -\frac{m \cdot \ln(2)^2}{\ln(\varepsilon)} \right\rfloor - c \tag{9}$$

The only other parameter required to build the BF is m. A lower bound for m is represented by $m_{\min}$, as computed from Equation (7). For $m = m_{\min}$, the tenant has to perform a *greedy renew* for every update, thus falling back to the *always renew* strategy. Values of m higher than $m_{\min}$ reduce the network overhead for update operations, but they increase storage and network overhead for 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 overhead. In the following section we propose an analytical method to estimate the optimal BF size with respect to the tenant requirements.

6. Analytical model for overhead minimization

We have shown that a greedy renew strategy requiring larger BFs reduces network overhead in an update scenario at the cost of additional storage overhead. Now we propose an analytical model that takes as its inputs the acceptable false positive rate ε, the database characteristics and the database workload, and computes the best BF size, namely m_{best}, that minimizes the costs faced by a tenant. In Section 6.1, we introduce the cost models for cloud database services. In Section 6.2, we describe our cost minimization methodology for a proxy-based architecture. In Section 6.3, we extend the minimization methodology for an alternative architecture that does not leverage any proxy. Table 2 summarizes the main parameters used in the models.

6.1. Costs and network usage models

We assume that the tenant uses a typical pay-per-use cloud database service, where the costs are a function of storage, ingoing network and outgoing network traffic. We assume service costs that are independent of each other, that is, where the total cost of the service can be modeled by the following equation:

$$Cost = Cost(Storage) + Cost(NetIn) + Cost(NetOut) \tag{10}$$

We denote as φ_s, φ_i and φ_o the cost weights related to storage, ingoing and outgoing network, respectively. We define them as coefficients that are normalized with respect to the total cost ($\varphi_s + \varphi_i + \varphi_o = 1$):

$$\varphi_s = \frac{Cost(Storage)}{Cost} \tag{11}$$

$$\varphi_i = \frac{Cost(NetIn)}{Cost} \tag{12}$$

$$\varphi_o = \frac{Cost(NetOut)}{Cost} \tag{13}$$

Symbol	Description
ε	Acceptable false positive rate.
f	False positive rate of the bloom filter.
c	Number of columns.
t	Average tuple size.
m	Size of the encrypted Bloom filter.
d	Size of the initialization vector attached to the Bloom filter.
ω	Normalized frequency of execution of an operation.
p	Percentage of tuple size that is read or updated by select and update operations.
o	Amount of values updated in each update operation.
μ	Frequency of execution of a Bloom filter renew operation.
u	Amount of values that can be stored in Bloom filters before a renewal operation.
λ	Amount of greedy update operations between two renew operations.
φ_o	Percentage of costs that are due to outgoing network usage.

Table 2: Model parameters.

If the database is already deployed in the cloud, the tenant can determine φ_s, φ_i and φ_o by using the real costs of the basic service without our extensions. Otherwise, he can compute the costs and the corresponding weights by using estimation methodologies proposed in literature (e.g., [17, 22, 48]). In any case, it is important to evaluate the network usage that depends on type and number of operations executed on the database. We consider the most common operations: select, update and insert. We are not interested in delete operations because they do not transfer data between the tenant and the cloud provider.

We define the global workload W as the tuple $((\omega_S, S), (\omega_U, U), (\omega_I, I))$, where S, U, I are workload descriptions for select, update and insert operations, and ω_S, ω_U and ω_I represent the normalized execution frequencies of the operations within the workload W. We introduce the ingoing and outgoing network usage I and O for a given workload. For example, $\mathcal{O}_S(S)$ is the outgoing network usage for the select workload S, while $\mathcal{I}_I(I)$ is the ingoing network usage for the insert workload I.

Select operations affect only the outgoing network usage, since they only fetch data from the database service. On the other hand, insert operations affect only the ingoing network usage, because they push data to the database service. Finally, update operations affect both ingoing and outgoing network usage. Ingoing network usage is due to the upload of a new value; outgoing network usage is due to the download of the digests and, possibly, of all the other row values (in case of a digest renew).

We define the total outgoing ($NetIn$) and ingoing network usage ($NetOut$) as:

$$NetIn = \omega_I \cdot \mathcal{I}_I(I) + \omega_U \cdot \mathcal{I}_U(U) \quad (14)$$

$$NetOut = \omega_S \cdot \mathcal{O}_S(S) + \omega_U \cdot \mathcal{O}_U(U) \quad (15)$$

We model $\mathcal{I}_I(I)$ by assuming that any insert operation creates a new tuple whose average size is the sum of three components: the average size t of the tuple in the original database, the size of the digest m, and the size d of the initialization vector used to encrypt the BF. Hence, the ingoing network usage $\mathcal{I}_I(I)$ can be computed as follows:

$$\mathcal{I}_I(I) = t + m + d \quad (16)$$

The formula for the ingoing network usage $\mathcal{I}_U(U)$ due to updates is more complex. We assume that each update operation will push on average to the database an amount of data that is the sum of three components: the average amount of values transmitted, the size of the BF m, and the size d of the initialization vector used to encrypt the BF. We can define the workload U as a set of tuples (ω, p, o), each describing a single class of update operations. The parameter ω is the normalized frequency of execution ($\sum_{\omega \in U} \omega = 1$), p is the ratio between the size of the updated values and the size of the updated row, and o is the number of updated values ($o \leqslant c$). The average ratio of data modified by an update operation $\bar{p}_u$ can be computed as the weighted average of the sizes of updated values across the different classes of update operations, that is:

$$\bar{p}_u = \sum_{(\omega,p,o) \in U} \omega \cdot p \quad (17)$$

Multiplying $\bar{p}_u$ by the average tuple size t yields the average amount of data transmitted by all update operations. Thus, the ingoing network usage $\mathcal{I}_U(U)$ can be written as:

$$\mathcal{I}_U(U) = m + d + t \cdot \bar{p}_u \quad (18)$$

Note that the o values modeled in the (ω, p, o) tuple is used later to quantify the amount of values that can be updated without requiring a greedy renew.

To define $\mathcal{O}_S(S)$ we model the workload of select operations S as a set of tuples $\{(\omega, p)\}$, each describing a single class of select operations. The parameter ω is the normalized frequency of execution and p is the ratio between the average size of the retrieved data and the size of a row. We estimate the average outgoing network usage due to select operations as the weighted average of network usage of the different types of select operations, that is:

$$\mathcal{O}_S(S) = m + d + t \cdot \sum_{(\omega,p) \in S} \omega \cdot p \quad (19)$$

Finally, for the model of the outgoing network usage $\mathcal{O}_U(U)$ due to update operations we must consider the greedy renew strategy used to periodically renew the BFs. An update operation that does not renew the digest only retrieves the subset of the row that will be updated. An update operation that renews the digest retrieves the whole row. We model the average outgoing network usage of update operations as the following weighted average:

$$\mathcal{O}_U(U) = m + d + t \cdot [1 - (1-\mu) \cdot (1-\bar{p}_u)], \tag{20}$$

where μ is the average frequency rate of the renew operations, that is directly proportional to the amount of values u that we can insert in the BFs (see Section 4) and inversely proportional to the amount of values o updated in each update operation. As u is inversely proportional to the BF size m (see Equation (9)), choosing greater values of m allows us to reduce the frequency of renew operations. However, this choice increases the network usage of both greedy and renew operations. The aim of our methodology is to estimate the value $m = m_{best}$ that minimizes Equation (20). Then, by using this optimal solution, we are able to minimize Equation (10). To optimize Equation (20) we must denote μ in terms of m. In this paper, we propose two variants of the protocol.

- The *stateful* variant assumes that the states of BFs are known. This scheme can be deployed on architectures based on a proxy that tracks the update operations issued by all clients. We discuss the scheme in Section 6.2.

- The *stateless* variant assumes that distributed clients operate on the cloud database without any intermediary servers and without knowing the states of BFs. This scheme can be easily deployed on client-side applications without any intermediate server, but offers weaker security guarantees in presence of some advanced attack scenarios. We discuss the scheme and its security guarantees in Section 6.3.

6.2. Stateful protocol

We initially consider a stateful protocol that is able to track the number of values inserted in all the encrypted BFs. A possible architecture leverages a trusted proxy that intercepts all operations issued to the cloud database service. This proxy manages a counter for each digest to track the number of values stored in each of them. A similar architecture is characterized by two drawbacks:

- it increases the tenant costs due to infrastructure management;

- if the tenant has geographically distributed clients, this scheme is not efficient because all client requests must pass through the proxy.

For these reasons, we also propose the optimization methodology for an alternative distributed architecture in Section 6.3.

Let λ be the number of greedy update operations between two consecutive renewals. The renew frequency rate

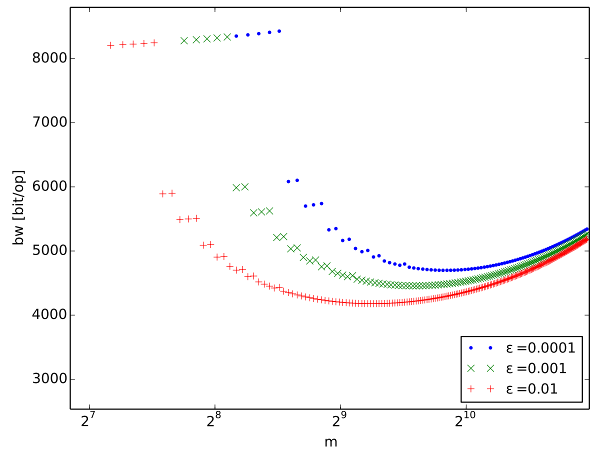


Figure 1: Estimated average network usage for each update operation as a function of the BF size m.

μ can be expressed as a function of λ as following:

$$\mu = \frac{1}{1+\lambda} \tag{21}$$

The parameter λ can be estimated as the ratio between the amount of values that can be inserted in the BF, that is u (see Equation (9)), and the expected amount of values modified by each update operations, namely $\bar{o}_u$:

$$\lambda = \frac{u}{\bar{o}_u} \tag{22}$$

where:

$$\bar{o}_u = \sum_{(\omega,o,p)\in U} \omega \cdot o \tag{23}$$

is the weighted average of the number of values updated in each update operation for each class. To obtain a good approximation of u in the most general scenario in which the update workload includes many classes of update operations, we define the following constraint: u should be chosen as the maximum linear combination of the values $\{o\}$ that is lower or equal to the maximum amount of values that can be stored in the BF. Hence, we adjust the definition of u in Equation (9) as following:

$$u = \max\left\{x \in \mathbb{N}_0 | x \leqslant -\frac{m \cdot \ln(2)^2}{\ln(\varepsilon)} - c, x = \sum_{(d,o)\in\mathbb{N}_0\times U} d \cdot o\right\} \tag{24}$$

The outgoing network usage due to update operations can be estimated by substituting Equations (24), (22) and (21) in (20).

For the sake of clarity, we describe an example by referring to Figure 1. This figure shows the behavior of Equation (20), where the y-axis represents the estimated average outgoing network usage $\mathcal{O}_U(U)$ for update operations, while the x-axis represents the BF size m (in log-scale). Points represent admissible BF sizes and the corresponding outgoing network usage due to update operations. The estimation refers to a table of $c = 15$ columns and in which the tuples have an average size of $t = 1KB$. The size of the initialization vector is $d = 64bit$. The update workload is characterized by two operations updating $o_1 = 5$ and $o_2 = 7$ values that transfer $p_1 = 30\%$ and $p_2 = 60\%$ of the tuple size. The execution frequencies are $\omega_1 = 0.7$ and $\omega_2 = 0.3$, respectively. Figure 1 reports also three scenarios that correspond to the acceptable false positive rates $\varepsilon = 1\%, 0.1\%, 0.01\%$. For each of them, through Equation (7), we can compute the minimum BF sizes, that are $m_{\min} = 144, m_{\min} = 216$ and $m_{\min} = 288$ bits, respectively. Choosing these values requires an always renew strategy and, as previously discussed and confirmed by the figure, they do not allow to minimize network usage. We need to compute the value of m for which the outgoing network usage is minimal. To this purpose, in Figure 1 we define "segment" a set of close, aligned points of the same curve. Each point corresponds to incremental values of u (the first point occurs for $u = 0$, the second one for $u = 1$, and so on). The goal of our analysis is to compute the local minimum in each segment; then, we compute the global minimum among all the local minima.

We note that for increasing BF sizes, the network usage increases within the same segment. We obtain this behavior when increasing the BF size does not incur in additional greedy update operations, due to the constraint imposed on u by Equation (24). As an example, let us consider the first "segment"., where the first value corresponds to the bandwidth usage for $m = m_{\min}$, for which no greedy update operation is possible. Choosing any other value of m within the first segment implies an u in range $1, \ldots, 4$ which is always less then any admissible number of updated values in the considered workload (we recall that $o = \{o_1 = 5, o_2 = 7\}$). Thus, no greedy update operation can be executed as well. As a result, the local minimum of segment is the minimum value of m for that segment.

We define M as the set of the local minima:

$$M = \left\{ m \mid \left\lfloor -\frac{m \cdot \ln(2)^2}{\ln(\varepsilon)} - c \right\rfloor \in \left\{ \sum d \cdot o \right\}_{(d,o) \in \mathbb{N}_0 \times U} \right\} \tag{25}$$

Here, the goal is to define which value of M is the global minimum of the estimation function. To this aim, we compute the minimum of the continuous function that intersects the elements of M, namely $\dot{m}_0$. Then, the absolute minimum of the target function is the nearest neighbor of $\dot{m}_0$ in M.

$$\dot{m}_0 = -\frac{(c - \bar{o}_u) \cdot \ln \varepsilon}{\ln(2)^2} + \frac{\sqrt{\bar{o}_u \cdot t \cdot \ln \varepsilon \cdot (\bar{p}_u - 1)}}{\ln(2)} \tag{26}$$

The proposed methodology estimates the best value of m that reduces outgoing network usage due to the update workload. Computing the value of m that minimizes the overall cloud service cost is an immediate extension of Equation (26). As described by the Equations (14)—(18), ingoing network usage due to insert and update, and outgoing network usage due to select is linearly dependent on m. Hence, we can use the costs weight φ_o (see Equation (13)) and the workload frequency ω_u (see Equation (14)) to compute the minimum of the continuous form of the Equation (10) as following:

$$\hat{m}_0 = -\frac{(c - \bar{o}_u) \cdot \ln \varepsilon}{\ln(2)^2} + \frac{\sqrt{\omega_u \cdot \varphi_o \cdot \bar{o}_u \cdot t \cdot \ln \varepsilon \cdot (\bar{p}_u - 1)}}{\ln(2)} \tag{27}$$

The optimal BF size m_{best} is the nearest neighbor of $\hat{m}_0$ in the set M, computed through Equation (25).

In Appendix D, we propose a closed-form equation to compute the optimal value of m in a simple yet common scenario characterized by an update workload that includes only one class of update operation.

6.3. Stateless protocol

We now discuss how to minimize the proposed verification mechanism in a stateless protocol variant. In this scenario each client operates directly on the cloud database service with no knowledge about the state of the BFs. Hence, it is necessary to decide when to renew the BFs without knowing the actual amount of values inserted in them. We propose the following implementation: each update operation is greedy with a given probability, otherwise it renews the BF. This approach suffers from two drawbacks:

- a client might renew a BF even if the acceptable amount of update operations has not been reached yet;
- a client might not renew a BF even if the acceptable amount of update operations has already been reached.

The former issue does not impact the ability of a system to verify integrity, although the outgoing network usage might be higher than necessary. On the other hand, the latter issue may lead to a false positive rate higher than ε. Even worse, this cannot be completely prevented when using the stateless protocol variant. We assume that the tenant chooses a probability of not exceeding ε, then we analyze the behavior of this protocol and minimize its overhead.

This methodology assumes a weaker security model with respect to the stateful protocol. By keeping track of the operations executed by the tenant, the storage provider knows if the tenant exceededs the amount of acceptable updates. Thus, he is able to attack the scheme when it has security guarantees lower than ε. Hence, this variant is secure only against attackers that have read/write access to the tenants data for a short amount of time (referring to literature terminology, we could call it as a *snapshot active adversary* [25]).

We define q as the *false positive threshold* that is the probability of keeping $f \leqslant \varepsilon$. We assume that a client executes a renew operation with probability μ and a greedy operation with probability $1-\mu$. Then, q can be computed as the probability of not exceeding the acceptable amount of update operations λ that is, the CDF of the geometric probability distribution with mean μ and number of failures λ. We note that our construction allows to estimate the outgoing network usage due to updates by using Equation (20), however the estimation of μ is different from that of the stateful protocol. The value μ can be estimated as the inverse of the CDF of the geometric distribution:

$$\mu = 1 - \sqrt[\lambda]{1-q} \tag{28}$$

One can obtain the equation estimating the outgoing network usage due to updates by substituting Equation (28) in (20) as following:

$$\mathcal{O}_U(U) = m + d + t \cdot \left[1 - (1-\bar{p}_u) \cdot \sqrt[\lambda]{1-q}\right] \tag{29}$$

Let us compare the behavior of this function with that of the stateful protocol in Section 6.2 by referring to Figure 2. It compares the network usage estimation between the stateful and the stateless protocols for the following values of false positive threshold: $q = 0.8, 0.9, 0.99$. The database characteristics and the workload parameters are the same of the scenario previously proposed in Figure 1. This figure shows that the performance of the stateless protocol is similar to that of the stateful protocol for $q = 0.8$, although it is unrealistic to consider $q = 0.8$ as an acceptable choice for a cloud tenant. (It would mean that the tenant accepts a 20% probability of exceeding the required ε.) Even more important, we note that the best BF size changes with respect to the stateful protocol and also for different values of q.

The optimal BF size m_{best} can be computed through the approach already shown for the stateful protocol:

- we first compute the set of local minima M;
- we compute a continuous function that intersects all the local minima;
- we identify the global minimum $\dot{m}_0$ of this continuous function;
- finally we select as m_{best} the local minimum that is the nearest neighbor of $\dot{m}_0$.

The evaluation of λ and M use the same models and equations adopted for the stateful protocol, but for the estimation of the renew frequency rate μ we use Equation (28). One can estimate the outgoing network usage caused by update operations by substituting the Equation (22) in (29). We can compute the optimum BF size

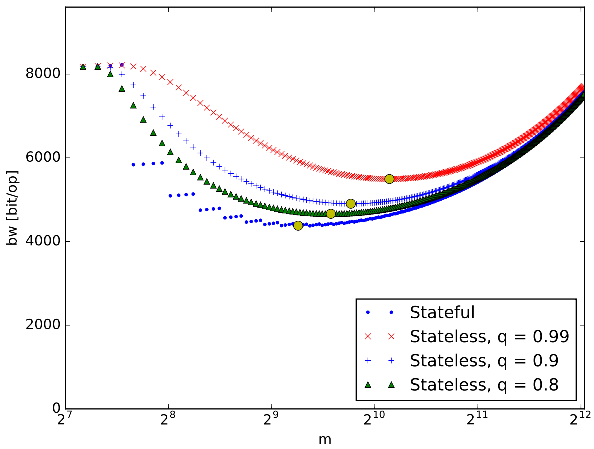


Figure 2: Estimated average network usage as a function of the BF size m for the stateless protocol.

$\dot{m}_0$ of the continuous form of the Equation (29) as following:

$$\dot{m}_0 = -\frac{c \cdot \ln(\varepsilon)}{\ln(2)^2} - \frac{\bar{o} \cdot \ln(1-q) \cdot \ln(\varepsilon)}{2 \cdot \ln(2)^2 \cdot W(x)}, \tag{30}$$

$$x = -\frac{1}{2} \cdot \sqrt{\frac{\bar{o} \cdot \ln(1-q) \cdot \ln(\varepsilon)}{t \cdot (1-\bar{p}) \cdot \ln(2)^2}}, \tag{31}$$

where $W(\cdot)$ is the *Lambert W function* [14].

We can extend this evaluation to compute the best value of $m = \hat{m}_0$ to minimize the overall costs (see Equation (10)) as done for the stateful protocol (see Equation (27)):

$$\hat{m}_0 = -\frac{c \cdot \ln(\varepsilon)}{\ln(2)^2} - \frac{\bar{o} \cdot \ln(1-q) \cdot \ln(\varepsilon)}{2 \cdot \ln(2)^2 \cdot W(\hat{x})}, \tag{32}$$

$$\hat{x} = -\frac{1}{2} \cdot \sqrt{\frac{\bar{o} \cdot \ln(1-q) \cdot \ln(\varepsilon)}{\varphi_o \cdot \omega_u \cdot t \cdot (1-\bar{p}) \cdot \ln(2)^2}}, \tag{33}$$

7. Performance evaluation

To analyze the performance of the proposed solution we compare its storage and network overhead 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 associate a MAC tag to each value stored in the database by using HMAC-SHA256. This solution causes a high storage overhead because a 256-bit digest 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 cloud database service. In the performance evaluation we refer to this solution as *VLH* (value-level HMAC). The second solution, proposed in [44], associates an HMAC-SHA256 to all the values of

		Plain [bytes]	VLH [bytes] (overhead)	TLH [bytes] (overhead)	EBF [bytes] (overhead)		
					$\varepsilon = 0.01$	$\varepsilon = 0.001$	$\varepsilon = 0.0001$
Storage Overhead	Tuple size	500	820 (64%)	532 (6.4%)	520 (4%)	526 (5.2%)	532 (6.4%)
Network overhead	Select $p = 10\%$	50	82 (64%)	532 (964%)	70 (40%)	76 (52%)	82 (64%)
	Select $p = 50\%$	250	410 (64%)	532 (112%)	270 (8%)	276 (10.4%)	282 (12.8%)
	Insert	500	820 (64%)	532 (6.4%)	520 (4%)	526 (5.2%)	532 (6.4%)

Table 3: Storage and network usage comparison for VLH, TLH and EBF integrity solutions.

the same row. The resulting scheme has the same structure of that shown in Table 1, but the last column is used to store a HMAC rather than an encrypted BF. 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 wants to verify the integrity of one value, he has to retrieve all the values stored in the same row. The retrieval of unnecessary values increases the network overhead.

To compare the performance of our approach against VLH and TLH we compute the optimal BF size m_{best} for the acceptable false positive rate ε of the tenant, and we consider both the stateful and stateless protocols as a function of different acceptable false positive rates ($\varepsilon = 10^{-2}, 10^{-3}, 10^{-4}$) and two maximum false positive thresholds $q = 0.9, 0.99$. We refer to our solution as *EBF* (Encrypted BF). We initially analyze the performance related to micro-benchmarks with different database characteristics and workloads. Then we evaluate the storage and network overhead of a realistic scenario by referring to the TPC-C workload.

7.1. Micro-benchmark workloads

We refer to a table in which each row is $t = 500$ bytes long, and contains $c = 10$ values. We assume that all values have the same size. We encrypt the BF by using a standard Blowfish 64-bit cipher [45] with a $d = 64$-bit initialization vector.

We consider a scenario that includes only select and insert operations. As described in Section 4.4, in this scenario the optimal choice is the minimum BF size $m = m_{\min}$. By using Equation (7), we compute $m_{\min} + d$ equal to $160, 208, 256$ bits for ε equal to $10^{-2}, 10^{-3}, 10^{-4}$, respectively. For insert operations, all the values of the row and all the corresponding digests (HMACs for VLH and TLH, Encrypted Bloom Filters for EBF) have to be transmitted from the client to the cloud database. Moreover, we consider three types of select operations depending on the amount of retrieved values.

Table 3 summarizes the number of bytes stored and transmitted for the different integrity strategies. The first row shows that VLH has the greatest storage overhead, while TLH and all EBF configurations have comparable overhead. The second row reports the bytes downloaded in the case of a select that retrieves only one value. The performance of EBF and VLH is optimal, because the tenant needs to retrieve just one value and the associated digest. If the select accesses only a subset of data, then EBF has a clear advantage over both TLH and VLH, as shown by the third row in which the tenant needs 5 out of the 10 values of a row. When select queries read all the values of a row, the performance of EBF and TLH is optimal, while VLH incurs in a high network overhead because the tenant has to retrieve one control structure for each value.

7.2. Mixed operations of the TPC-C workload

Now we consider the TPC-C standard OLTP benchmark, that is commonly adopted for evaluating database performance. The contribution is twofold: we show how to leverage the proposed overhead minimization methodology described in Section 6; then, we demonstrate the performance advantages of using EBF in realistic workload scenarios.

Using the proposed methodology to estimate the best BFs size requires to translate database workloads into the parameters of the proposed model. Let us refer to a TPC-C compliant database represented in Table 4. We distinguish two kinds of parameters: those describing the database schema, and those representing the workload. The number of columns c and the average size of the tuples t are described in the first two rows of the table. Moreover, the TPC-C standard defines a workload in which five transactions are executed with certain probabilities. Each transaction is a set of mixed operations executed on many tables. Let us focus on the table *district* (we limit to it for space reasons, although the same approach can be applied to any other table). This table has 11 columns and the average size of the tuples is 107 bytes. Two update operations are executed on the table, and both of them update only one value. The size fraction p of a tuple transferred by an update can be obtained by computing the ratio between the sum of the sizes of the columns interested by the operation, and the average size of the tuple t. The update operations are executed within the *new order* and the *payment* transactions, that have probabilities of execution equal to 45% and 43%, respectively. The frequencies of execution $\{\omega\}$ within the UPDATE workload can be computed by normalizing their values, that is $\omega = {}^{43}/_{(43 + 45)}$ and $\omega = {}^{45}/_{(43 + 45)}$. The frequency of execution ω_U of the update workload can be obtained by computing the sum of all the transaction frequencies of the update operations

Table	warehouse	district	item	customer	history	stock	order	new_order	order_line
Number of columns c	9	11	5	21	8	17	8	3	10
Avg. tuple size $t/8$ [bytes]	99	107	87	681	53	318	34	12	62
# values updated $\{o\}$	1	1, 1	(none)	1, 4, 3	(none)	1	1	(none)	1
Perc. size $\{p\}$ [%]	9.1	8.41, 3.7	(none)	1.3, 76.8, 3.38	(none)	1.57	11.8	(none)	6.5
Execution freq. $\{\omega\}$ [%]	100	48.8, 51.2	(none)	8.53, 10.7, 80.8	(none)	100	100	(none)	100
UPDATE freq. ω_u [%]	32.8	50	0	33.8	0	47.9	7.02	0	6.56

Table 4: Analysis of the tables of a TPC-C compliant database

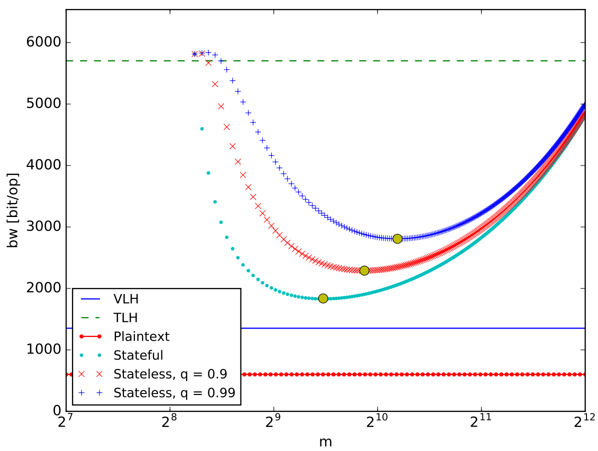


Figure 3: Estimated average network usage for each update operation in terms of the BF size m.

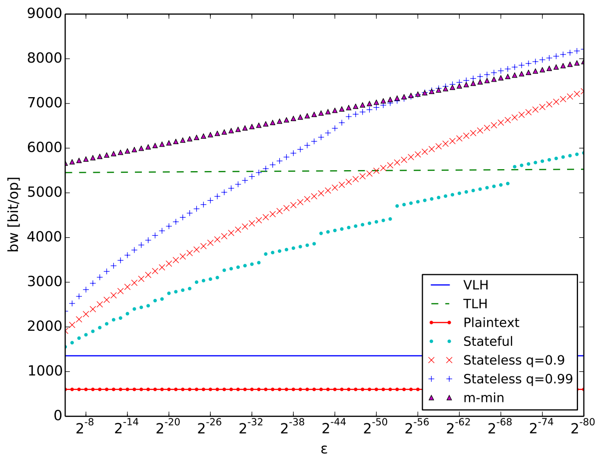


Figure 4: Estimated average network usage for each update in terms of acceptable false positive rate ε.

divided by the frequencies of all the operations executed on that table.

We now investigate the performance of integrity strategies applied to table *customer* by limiting the analysis to Figures 3 and 4 for space reasons. The figures compare the estimation of average network usage due to update operations in databases that adopt VLH, TLH and EBF integrity strategies and with the plaintext database. Figure 3 details network usage for increasing BF sizes when the acceptable false positive rate ε is equal to 10^{-3}. Figure 4 details network usage for decreasing acceptable false positive rates in the range $2^{-5}, \ldots, 2^{-80}$. In Figure 3 we note that the estimated network usage for the VLH, TLH and plaintext databases is constant, because they do not leverage BFs. The figure shows also that EBF greatly improves network overhead over TLH. Moreover, the stateful protocol allows us to achieve results comparable to that of VLH.

In Figure 4 we note that the proposed protocols are convenient even for much stronger security guarantees. The stateless protocol with a very high false positive threshold ($q = 0.99$) is convenient up to an acceptable false positive rate equal to about 2^{-34}, and the stateful version is convenient up to about 2^{-70}. We highlight that the convenience of the protocol depends on the workload. In both Figures 4 and 4 we note that if the tenant chooses an always renew strategy, then the network usage overhead is increased both with respect to the greedy renew strategy (due to the retrieval of all the values of the tuple) and to the VLH solution (due to bigger BF sizes). Finally, in Figure 5 we show the sizes of the BFs stored in database as a function of the acceptable false positive rate ε. We note that as ε decreases, the choice of the best BF size becomes closer to the minimum BF size.

8. Related work

Many cloud providers offer database services [26], but their security in terms of tenant's data confidentiality and integrity remains an open research area. Literature proposes architectures and protocols aimed at improving service dependability by guaranteeing data confidentiality [20, 21, 43]. They typically assume the *honest-but-curious* threat model where the main threat, besides external attackers, is represented by a cloud provider employee that may snitch on tenant's data, without modifying it. These architectures rely on encryption strategies that allow the tenant to execute SQL operations on encrypted

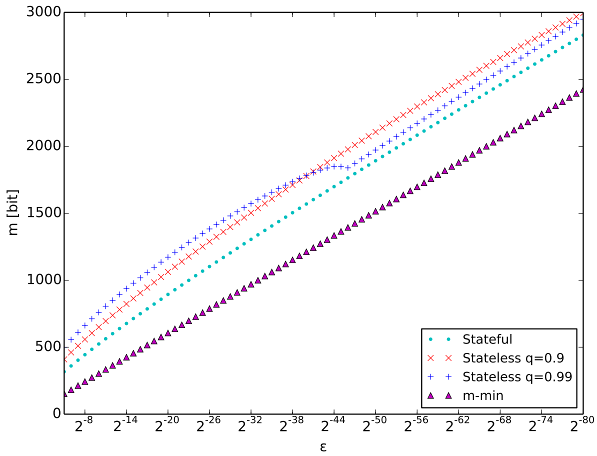


Figure 5: BF size in terms of the acceptable false positive rate ε.

data. In this paper we consider a different threat model in which data outsourced to the cloud may become corrupted, due to incorrect or faulty operations of the cloud provider or to deliberate attacks. Protocols guaranteeing verification of computations over outsourced data [42] are affected by high computational costs, especially on the server, and they are not convenient for database supporting CRUD workloads.

The scenario considered in this paper is a case of the more general data publishing scenario, in which the objective of the tenant is to leverage some third party infrastructure to distribute information to many users. In the data publishing scenario there are two classes of actors: *owners*, that can write data, and *users*, that only execute read operations. All solutions for guaranteeing integrity in this scenario [10, 31, 39] are based on asymmetric cryptography, and allow users to verify data integrity by leveraging digital signatures generated by owners. Similar approaches could guarantee integrity even in our database outsourcing scenario, but they are affected by high computation overhead [15]. For this reason we propose a novel protocol ensuring data integrity based on symmetric encryption primitives.

Research efforts on guaranteeing integrity for files in cloud storage services tend to associate a Message Authentication Code (MAC) to each file stored in the cloud. In such a way, a cloud tenant can verify data integrity by downloading a file, recomputing the MAC and comparing it with the tag stored in the cloud. Since we assume that only legitimate users know the symmetric key required to compute a MAC, any adversary that tries to modify data without authorization cannot compute a new valid MAC. These approaches could be extended to the cloud database services by associating a MAC to each attribute stored in the database (for example, each element in any tuple). However, the storage size of a MAC is non-trivial (e.g., 256 bits for HMAC based on SHA256) and bigger than many primitive data types commonly stored in a database. This causes two problems that make a similar approach inapplicable to cloud database services: the storage of a MAC for each database attribute causes an excessive storage overhead; the transmission of MACs causes an excessive network overhead. Since storage and network usage is metered in cloud services, solutions based on MACs lead to excessive cost increases for the tenant.

Storage overhead can be reduced by using one MAC to authenticate multiple attributes [41] (for example, one MAC for each tuple or table), but a similar approach implies that whenever a cloud tenant aims to verify the integrity of one attribute, he has to retrieve all the other attributes related to the same MAC. For example, it would be necessary to download all the attributes of a row or of a table even if the cloud tenant wants to verify the integrity of only one of them. This approach may be viable for file storage services [44] because MACs are associated to data blocks of large sizes that are accessed independently, but it causes high network overhead in database workloads, especially when queries often requests a small table subset.

A more intersecting approach relies on provable data possession protocols [3, 19] and proofs of retrievability [27], that allow tenants to verify integrity of very large data stored in outsourced databases when there is no need to retrieve them. In this scenario, the database provider is able to prove that he still maintains the tenant outsourced data by only returning a small amount of data [38]. Provable data possession protocols offer low communication costs for workloads that involve few or no read operations and tight integrity guarantees, such as replicated backups or ledgers. However, they are not convenient for CRUD workloads involving frequent read and update operations. Proofs of retrievability cannot be applied on unencrypted data, thus limiting their scope to specific applications.

The approach most related to our proposal is based on cryptographic accumulators [7] and asymmetric cryptography (e.g., RSA accumulators [24], bilinear accumulators [39], and aggregate signatures [13]). These methods support membership operations, that is, the tenant can associate a cryptographic accumulator to each tuple of the database, and can test the integrity of each value without having to download the entire tuple. Cryptographic accumulators are effective and efficient in guaranteeing integrity for some scenarios, such as log databases [31] and source code repositories [12]. On the other hand, because of asymmetric cryptography, they introduce a high computational overhead that makes them unsuitable to the majority of database workloads, especially when data is managed by cloud providers. In [15] the authors compare PADs (Persistent Authenticated Dictionaries) built upon RSA accumulators against PADs built upon other cryptographic data structures such as hashes and traditional digital signatures. They conclude that RSA accumulators are never the preferable algorithm despite their superior asymptotic complexity, due to the high computational costs.

Other results based on Bloom filters to efficiently detect accidental and malicious modifications of files stored in the cloud are reported in [1, 49]. The authors of [49] proposed a scalable checksum algorithm based on Bloom filters that reduce processing times in highly-parallel environments. Their solution allows the detection of data modifications in the case of accidental data corruption, but it cannot guarantee data authenticity in the case of adversarial modifications by malicious attackers in a data outsourcing scenario. The protocol proposed in [1] can be applied to adversarial scenarios, but it focuses on medium-to-large file sizes. Hence, it seems not practicable for the cloud database context that is typically characterized by a high volume of small-sized data.

The use of Bloom filters as cryptographic data structures has been investigated by other authors [18, 40]. In [40], the researchers propose the use of Bloom filters as symmetric cryptographic accumulators, but their computational cost is two orders of magnitude higher than that of RSA asymmetric accumulators [29]. Recent literature in the field of secure two-party computation protocols [18] has proposed a data structure based on Bloom filters and secret sharing to execute private set intersection protocols based on oblivious transfer. Since their data structure has a size much higher than that of common Bloom filters, their proposal is inapplicable to the cloud database scenario because it causes unacceptable network and storage overhead.

To the best of our knowledge, this is the first paper that proposes and evaluates a practical scheme for the verification of data integrity in cloud database services with the guarantee of efficient verification, low computational costs, and low overhead in terms of extra storage and network traffic. By combining Bloom filters and symmetric encryption, our scheme protects data against unauthorized modification by cloud employees and external attackers while minimizing computational costs related to Bloom filter verification and update. Moreover, it has the capability of testing set membership of an element without retrieving all the other attributes of the same row, thus minimizing network and storage overhead.

9. 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 scheme that allows cloud tenants to detect unauthorized modifications to data outsourced to untrusted cloud providers. We demonstrate that the solution, based on encrypted Bloom filters, is attractive especially in the case of metered network traffic and storage, that is common in cloud database service offers. The proposed solution allows the tenant to tune the trade-off between the probability of detecting unauthorized data modifications and storage and network overhead. We show that the protocol is applicable to architectures that are based on an intermediate trusted proxy and to distributed independent clients. We propose analytical methodologies that allow us to calculate the best size of the Bloom filters to minimize storage and network overhead and the cloud service costs. We demonstrate that the proposed scheme and methodologies are effective and that they reduce resource usage in realistic scenarios. The storage overhead of the proposed protocol is comparable or smaller than that of other solutions for database integrity and its network overhead is consistently lower.

As future work we are studying the integration of the proposed Bloom filter strategy with mechanisms for guaranteeing data completeness and freshness.

References

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

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

[3] Ateniese, G., Di Pietro, R., Mancini, L. V., Tsudik, G., 2008. Scalable and efficient provable data possession. In: Proc. Fourth ACM Int'l Conf. Security and privacy in communication networks.

[4] Barker, E., Barker, W., Burr, W., Polk, W., Smid, M., 2007. Nist special publication 800-57. Tech. Rep. 57.

[5] Bellare, M., Canetti, R., Krawczyk, H., 1996. Keying hash functions for message authentication. In: Advances in Cryptology. Springer.

[6] Bellare, M., Canetti, R., Krawczyk, H., 1996. Pseudorandom functions revisited: The cascade construction and its concrete security. In: Proc. 37th IEEE Annual Symp. Foundations of Computer Science.

[7] Benaloh, J., De Mare, M., 1994. One-way accumulators: A decentralized alternative to digital signatures. In: Advances in Cryptology. Springer.

[8] Bloom, B. H., 1970. Space/time trade-offs in hash coding with allowable errors. Comm. ACM.

[9] Blundo, C., Cresti, A., Vaccaro, U., Jun.-Jul. 1994. Key distribution schemes. In: Proc. IEEE Int'l Symp. Information Theory.

[10] Brandenburger, M., Cachin, C., Knežević, N., 2015. Don't trust the cloud, verify: integrity and consistency for cloud object stores. In: Proc. 8th ACM Int'l Conf. Systems and Storage.

[11] Broder, A., Mitzenmacher, M., 2004. Network applications of bloom filters: A survey. Internet Mathematics.

[12] Camenisch, J., Lysyanskaya, A., 2002. Dynamic accumulators and application to efficient revocation of anonymous credentials. In: Advances in Cryptology. Springer.

[13] Chen, X., Li, J., Weng, J., Ma, J., Lou, W., 2016. Verifiable computation over large database with incremental updates. IEEE Trans. Computers.

[14] Corless, R. M., Gonnet, G. H., Hare, D. E., Jeffrey, D. J., Knuth, D. E., 1996. On the lambertw function. Advances in Computational mathematics.

[15] Crosby, S. A., Wallach, D. S., 2011. Authenticated dictionaries: Real-world costs and trade-offs. ACM Trans. Information and System Security.

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

[17] Deelman, E., Singh, G., Livny, M., Berriman, B., Good, J., 2008. The cost of doing science on the cloud: the montage example. In: Proc. ACM/IEEE Conf. Supercomputing.

[18] Dong, C., Chen, L., Wen, Z., 2013. When private set intersection meets big data: an efficient and scalable protocol. In: Proc. ACM SIGSAC conference on Computer & communications security.

[19] Erway, C. C., Küpçü, A., Papamanthou, C., Tamassia, R., 2015. Dynamic provable data possession. ACM Trans. Information and System Security.

[20] Ferretti, L., Colajanni, M., Marchetti, M., Dec. 2013. Access control enforcement of query-aware encrypted cloud databases. In: Proc. Fifth IEEE Int'l Conf. Cloud Computing Technology and Science.

[21] Ferretti, L., Colajanni, M., Marchetti, M., 2014. Distributed, concurrent, and independent access to encrypted cloud databases. IEEE Trans. Parallel and Distributed Systems.

[22] Ferretti, L., Pierazzi, F., Colajanni, M., Marchetti, M., 2014. Performance and cost evaluation of an adaptive encryption architecture for cloud databases. IEEE Trans. Cloud Computing.

[23] Ferretti, L., Pierazzi, F., Colajanni, M., Marchetti, M., 2014. Scalable architecture for multi-user encrypted sql operations on cloud database services. IEEE Trans. Cloud Computing.

[24] Goodrich, M. T., Tamassia, R., Hasić, J., 2002. An efficient dynamic and distributed cryptographic accumulator. In: Information Security. Springer.

[25] Grubbs, P., McPherson, R., Naveed, M., Ristenpart, T., Shmatikov, V., 2016. Breaking web applications built on top of encrypted data. In: Proc. ACM SIGSAC Conf. Computer and Communications Security.

[26] Hacigümüş, H., Iyer, B., Mehrotra, S., Feb. 2002. Providing database as a service. In: Proc. 18th IEEE Int'l Conf. Data Engineering.

[27] Juels, A., Kaliski Jr, B. S., 2007. Pors: Proofs of retrievability for large files. In: Proc. 14th ACM Conf. Computer and communications security.

[28] Katz, J., Yung, M., 2001. Unforgeable encryption and chosen ciphertext secure modes of operation. In: Fast Software Encryption. Springer.

[29] Kumar, A., Lafourcade, P., Lauradoux, C., et al., 2014. Performances of cryptographic accumulators. In: Proc. 39th IEEE Int'l Conf. Local Computer Networks.

[30] Li, F., Hadjieleftheriou, M., Kollios, G., Reyzin, L., 2006. Dynamic authenticated index structures for outsourced databases. In: Proc. ACM SIGMOD Int'l Conf. Management of data.

[31] Ma, D., Tsudik, G., 2009. A new approach to secure logging. ACM Trans. Storage.

[32] Mather, T., Kumaraswamy, S., Latif, S., 2009. Cloud security and privacy: an enterprise perspective on risks and compliance. O'Reilly Media, Inc.

[33] Mitzenmacher, M., 2002. Compressed bloom filters. IEEE/ACM Trans. Networking.

[34] Mitzenmacher, M., Upfal, E., 2005. Probability and computing: Randomized algorithms and probabilistic analysis. Cambridge University Press.

[35] Mykletun, E., Narasimha, M., Tsudik, G., 2006. Authentication and integrity in outsourced databases. ACM Trans. Storage.

[36] Namprempre, C., Rogaway, P., Shrimpton, T., 2014. Reconsidering generic composition. In: Proc. Int'l Conf. Theory and Applications of Cryptographic Techniques. Springer.

[37] Namprempre, C., Rogaway, P., Shrimpton, T., 2014. Reconsidering generic composition. In: Advances in Cryptology. Springer.

[38] Naor, M., Rothblum, G. N., 2009. The complexity of online memory checking. Journal of the ACM.

[39] Nguyen, L., 2005. Accumulators from bilinear pairings and applications. Topics in Cryptology.

[40] Nyberg, K., 1996. Fast accumulated hashing. In: Fast Software Encryption. Springer.

[41] Pang, H., Zhang, J., Mouratidis, K., 2009. Scalable verification for outsourced dynamic databases. Proc. VLDB Endowment.

[42] Parno, B., Howell, J., Gentry, C., Raykova, M., 2013. Pinocchio: Nearly practical verifiable computation. In: Proc. 2013 IEEE Int'l Conf. Security and Privacy.

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

[44] Rodriguez-Henriquez, L. M., Chakraborty, D., 2013. Rdas: A symmetric key scheme for authenticated query processing in outsourced databases. In: Security and Trust Management. Springer.

[45] Schneier, B., 1993. Description of a new variable-length key, 64-bit block cipher (blowfish). In: Proc. Fast Software Encryption.

[46] Stefanov, E., van Dijk, M., Juels, A., Oprea, A., 2012. Iris: A scalable cloud file system with efficient integrity checks. In: Proc. 28th ACM Computer Security Applications Conference.

[47] Tamassia, R., 2003. Authenticated data structures. In: Algorithms – ESA. Springer.

[48] Truong, H.-L., Dustdar, S., 2010. Composable cost estimation and monitoring for computational applications in cloud computing environments. Procedia Computer Science.

[49] Xiong, S., Wang, F., Cao, Q., 2016. A bloom filter based scalable data integrity check tool for large-scale dataset. In: Proc. First IEEE Joint Int. Workshop Parallel Data Storage & Data Intensive Scalable Computing Systems.

[50] Yang, Y., Papadias, D., Papadopoulos, S., Kalnis, P., 2009. Authenticated join processing in outsourced databases. In: Proc. ACM SIGMOD Int'l Conf. Management of data.

Appendix A. Implementation of the Secret Bloom filter function

The secret Bloom filter function $\mathcal{B}_\tau^m(\cdot)$ is implemented through the following formula:

$$\mathcal{B}_\tau^m(l, x) = \bigvee_{i \in [k]} \left[1 \ll H_{\tau_i}^m(l \parallel x)\right], \tag{A.1}$$

where $\mathcal{H}_{\tau_i}^m(\cdot)$ is a keyed hash function that maps arbitrary length inputs to the range $[0, \ldots, m-1]$, $\ll$ denotes the *bitwise left shift* operator, $\parallel$ is an operator for the secure concatenation of the two inputs [36], and k denotes the number of functions $\mathcal{H}_{\tau_i}^m(\cdot)$ used in the computation of $\mathcal{B}_\tau^m(\cdot)$. The output of the function is deterministic; it depends on the input data $l \parallel x$ and the secret key τ_i denoting a portion of the secret key τ distributed to the authorized clients. All keys τ_i are independent of each other.

From a security perspective the function $\mathcal{H}_{\tau_i}^m(\cdot)$ must act as a pseudo-random function (PRF) that is, a function whose output distribution is uniform and independent of the distributions of the input value x and of the secret key τ_i. Candidate implementations are keyed PRFs that accept variable length inputs [6] such as the truncated HMAC functions.

Appendix B. Attacks on the digest

This attack refers to a scenario where an adversary tries to attack the integrity of some data by forging its cryptographic digest. This attack is formally modeled in symmetric encryption literature as *chosen plaintext forgery* [28]. Intuitively, the attacker has to generate a couple of plaintext-ciphertext values such that the plaintext is the decryption of the ciphertext. We identify two types of attacks on the digests that are relevant to our protocol:

- *creation of a new digest*: an adversary tries to insert a forged tuple in the database and to generate a new digest that includes all values of the tuple;
- *modification of an existing digest*: an adversary tries to update an existing tuple through a forged value and to modify the existing digest by adding the new value to it.

Creation of a new digest. In the former attack scenario, the adversary creates a new tuple and a new digest. The digest will be decrypted by an authorized client and the resulting secret Bloom filter will be used to verify the values. The proposed protocol always uses BFs built with the optimal number of hash functions, hence the probability of having a bit equal to zero is the same of having a bit equal to one for all the bits [34]. As a result, whenever a single value is tested against a completely random bit string, the false positive rate is exactly the same of a proper Bloom filter of the same length that does not contain the tested value. The success of this attack requires that the adversary generates a new digest associated to all the values in a database row. Since the false positive probability of all the values are independent of each other, the probability of having false positives for all values in the row against the same random bit string is equal to the conjunction of the false positive rates. Hence, the success probability of this attack decreases exponentially with the number of values within a row.

Modification of an existing digest. In the latter type of attack, the adversary alters one value of an existing row and tries to tamper with the existing digest to increase the false positive rate. This can be achieved by flipping some bits of a Bloom filter from zero to one. Since secret Bloom filters are built through keyed hash functions and encrypted through an IND-CPA algorithm, the attacker does not know which bits in the Bloom filter are set to zero and which bits are set to one. However, since we do not authenticate the digest, flipping a bit at a certain position in the ciphertext may cause some bits at random position in the plaintext to flip as well, and these modifications cannot be detected.

Let us consider the worst case scenario where the security is guaranteed by the IND-CPA encryption algorithm based on stream ciphers that is the most malleable cipher. (We advise against the use of stream ciphers in the proposed protocol due to known implementation issues that could make it vulnerable to other attacks.) Let us also assume that the attacker knows all the algorithms and the parameters used in the protocol but the secret keys. In this scenario, by flipping a bit at a given position in the ciphertext, the attacker knows that he is flipping the bit at the same position in the plaintext Bloom filter. However, the attacker still does not know the values of the plaintext Bloom filter. In Appendix C we give an algebraic proof showing that an adversary gains no benefits from modifying a digest. As any other IND-CPA encryption algorithm is less malleable than stream ciphers, our proof holds for all IND-CPA algorithms.

Appendix C. Digest modification attack

The objective of the attacker is to increase the false positive rate of an existing digest by flipping from 0 to 1 one or more bits of the secret Bloom filter. We analyze how the false positive rate of the BF varies if an attacker modifies one or more bits of the digest. We are especially interested to compute which is the number of bits that the adversary should modify to gain the best advantage.

Let s be the amount of bits that the adversary modifies. The value of s is in $[0,\ldots,m-1]$, where $s=0$ denotes that the adversary does not modify the digest thus falling back to a plaintext attack, and $s=m-1$ denotes that the attacker modifies the whole digest except for a bit. Since the attacker is only interested in flipping bits from 0 to 1, and since at least 1 bit in the BF is always set to one, it makes no sense for the attacker to try to flip all m bits. As described in Section 3, in this paper we always consider an optimal number of hash functions $\bar{k}$ to compute the secret Bloom filter.

The adversary does not know which are the positions of the bits that correspond to the fake value, thus he chooses them at random. The false positive rate of the BF with s different random bits flipped by the attacker can be computed as following:

$$\begin{aligned}
&\Pr[fp \mid \#\ mod.\ bits = s] = \\
&= \sum_{i=1}^{m-s} \big(\Pr[fp \mid s\ succ\ mod] \cdot \\
&\qquad \cdot \Pr[s\ succ\ mod \mid \#1bits = i] \cdot \\
&\qquad \cdot \Pr[\#1bits = i] \big) \qquad \text{(C.1)}
\end{aligned}$$

where:

- $\Pr[fp \mid s\ succ\ mod]$ denotes the false positive rate of the BF after s successful modifications (that is, s bits have been flipped from 0 to 1). It can be computed as the false positive rate of a BF with s additional bits equal to one. We denote it as:

$$\begin{aligned}
&\Pr[fp \mid s\ succ\ mod] = \\
&= \Pr[fp \mid \#1bits{=}(i{+}s)] = \left(\frac{i+s}{m}\right)^{\bar{k}} \qquad \text{(C.2)}
\end{aligned}$$

  where i is the number of bits set to 1 in the original BF.

- $\Pr[s\ succ\ mod \mid \#1bits = i]$ denotes the probability of not flipping any bit from one to zero that is, the probability of randomly selecting s bits equal to 0 in the secret Bloom filter. We highlight that flipping just one bit from 1 to 0 would allow an authorized client to detect the integrity violation because at least one of the values in the row would fail verification against the tampered Bloom filter. Intuitively, this probability decreases as the original amount of bits equal to one i and the number of modifications s increase. We denote this probability as:

$$\Pr[s\ succ\ mod \mid \#1bits = i] = \\ = \prod_{j=0}^{s-1} \frac{(m-i-j)}{m-j} = \frac{(m-i)!\,(m-s)!}{(m-i-s)!\,m!} \tag{C.3}$$

- $\Pr[\#1bits = i]$ denotes the probability of having exactly i bits set to 1 in the Bloom filter bit string. As we use the optimal number of hash functions $\bar{k}$ to build the secret Bloom filters, then the probability distribution of zeros and ones in the unmodified BF bit string is uniform (see Section 3), and it can be computed as:

$$\Pr[\#1bits = i] = \frac{\binom{m}{i}}{2^m} = \frac{m!}{2^m i!(m-i)!} \tag{C.4}$$

By substituting Equations (C.2), (C.3) and (C.4) in (C.1), we obtain the following formula:

$$\begin{aligned}
&\Pr[fp \mid \#\ mod.\ bits = s] = \\
&= \sum_{i=1}^{m-s} \left(\frac{i+s}{m}\right)^{\bar{k}} \frac{(m-i)!\,(m-s)!}{(m-i-s)!\,m!} \frac{m!}{2^m i!\,(m-i)!} = \\
&= \frac{1}{2^m m^{\bar{k}}} \sum_{i=1}^{m-s} (i+s)^{\bar{k}} \frac{(m-s)!}{(m-s-i)!\,i!} = \\
&= \frac{1}{2^m m^{\bar{k}}} \sum_{i=1}^{m-s} (i+s)^{\bar{k}} \binom{m-s}{i} = \\
&= \frac{1}{2^m m^{\bar{k}}} \sum_{j=1+s}^{m} j^{\bar{k}} \binom{m-s}{j-s}
\end{aligned} \tag{C.5}$$

This function is monotonic decreasing. It can be verified by observing that the following has always a negative value:

$$\begin{aligned}
&\Pr[fp \mid \#\ mod.\ bits = s+1] - \Pr[fp \mid \#\ mod.\ bits = s\ ] = \\
&= \frac{1}{2^m m^{\bar{k}}} \left[\sum_{i=2+s}^{m} i^{\bar{k}} \binom{m-s-1}{i-s-1} - \sum_{i=1+s}^{m} i^{\bar{k}} \binom{m-s}{i-s} \right] = \\
&= -\frac{1}{2^m m^{\bar{k}}} \left[(1+s)^{\bar{k}} + \sum_{i=1+s}^{m} i^{\bar{k}} \binom{m-s-1}{i-s} \right]
\end{aligned} \tag{C.6}$$

Thus, the false positive rate of a manipulated digest is always lower than the false positive rate of the original BF. For example, let us compute how the false positive rate changes after modifying one bit ($s = 1$):

$$\Pr[fp \mid s = 0] - \Pr[fp \mid s = 1] = \\ = \frac{1}{2^m m^{\bar{k}}} \left[1 + \sum_{i=1}^{m} i^{\bar{k}} \binom{m-i}{i} \right] \tag{C.7}$$

This appendix shows that an adversary cannot gain any advantage by tampering with the encrypted Bloom filter. Hence a rational adversary will never try to tamper with the encrypted Bloom filter because any modification decreases the attack success rate.

Appendix D. Scenario with one class of update

We detail how to define a closed-form equation to compute the optimal Bloom filter size m_{best} if the database operations include only one class of update. In this case, the number of values inserted in the BF for each update operation is constant, and the amount of operations between two renewals λ is constant as well. As a result, λ can be computed as following:

$$\lambda = \left\lfloor \frac{u}{o} \right\rfloor \tag{D.1}$$

where o is the amount of values updated by the operation.

Thanks to Equation (D.1), the set of the local minima M can be defined by:

$$M = \left\{ \left\lceil -\frac{(\lambda \cdot o + c) \cdot \ln \varepsilon}{\ln(2)^2} \right\rceil \right\}_{\lambda \in \mathbb{N}_0} \tag{D.2}$$

This equation is obtained by substituting Equations (9) and (22) in (20) and inverting it.

As expected, the first element of the sequence M is equal to $m_{\min}$. The optimal BF size m_{best} can be computed as the BF size belonging to M that minimizes the update outgoing network usage. This occurs at the element λ_{best} of the sequence M. The value of λ_{best} can be computed through the closed-form equation obtained by substituting the Equation (27) in (D.2) and computing the inverse as following:

$$\lambda_{best} = \left\| -\frac{\hat{m}_0 \cdot \ln(2)^2}{\omega_u \cdot \varphi_o \cdot o \cdot \ln \varepsilon} + \frac{c}{\bar{o}_u} \right\| \tag{D.3}$$

where $\|$ represents the round operator.