Dottorato di Ricerca in
Computer Engineering and science
Scuola di Dottorato in
Information and Communication Technologies

XXVIII CICLO

Università degli studi di MODENA e REGGIO EMILIA

TESI PER IL CONSEGUIMENTO DEL TITOLO DI DOTTORE DI RICERCA

Confidentiality and integrity for cloud database services

Candidato:
Ing. Luca Ferretti

Relatore:
Prof. Michele Colajanni

Coordinatore:
Prof. Giorgio Matteo Vitetta

Direttore:
Prof. Giacomo Cabri

Contents

Contents i

List of Figures v

List of Tables vii

Introduction ix

1 System and security models 1
1.1 Scenario 1
1.2 Adversaries 2
1.2.1 External attackers 2
1.2.2 Cloud insiders 3
1.2.3 Tenant insiders 3
1.3 Attack models 4
1.3.1 Passive attacks 4
1.3.2 Active attacks 6

2 Related work 9
2.1 Confidentiality 9
2.2 Integrity 12

3 An encryption architecture for distributed access 17
3.1 Architecture design 18
3.1.1 Data management 20
3.1.2 Metadata management 22
3.1.3 Database setup 24
3.1.4 Operations execution 26
3.2 Concurrency management 27
3.2.1 Data manipulation 28
3.2.2 Structure modifications 29
3.2.3 Consistency management 31
3.2.4 Altering tables 33
3.2.5 Secure types modifications 34

3.2.6 Unrestricted operations . . . 37
3.2.7 Discussion . . . 37
3.3 Cryptographic access control enforcement . . . 38
3.3.1 Access control rules . . . 41
3.3.2 Plaintext database model . . . 42
3.3.3 Encrypted database model . . . 43
3.3.4 Cryptographic enforcement strategy . . . 46
3.3.5 Metadata management . . . 48
3.3.6 Query translation . . . 52
3.3.7 Credentials management . . . 53
3.4 Performance evaluation . . . 54
3.4.1 Prototype implementation . . . 54
3.4.2 Emulated environment . . . 57
3.4.3 Real-world environment . . . 63

4 Efficient authenticity guarantees in dynamic databases 71
4.1 Theoretical background . . . 72
4.2 Protocol design . . . 73
4.2.1 Insert operation . . . 73
4.2.2 Read operations . . . 75
4.2.3 Update operations . . . 76
4.3 Security analysis . . . 77
4.3.1 Attacks on the digest . . . 78
4.4 Sizing boundaries . . . 82
4.5 Overhead minimization . . . 84
4.5.1 Costs and network usage models . . . 84
4.5.2 Stateful protocol . . . 88
4.5.3 Stateless protocol . . . 92
4.6 Performance evaluation . . . 94
4.6.1 Micro-benchmarks . . . 96
4.6.2 Mixed operations . . . 96

5 Integrity guarantees for key-value databases 101
5.1 A protocol for mutual integrity guarantees . . . 102
5.1.1 System model and security guarantees . . . 103
5.1.2 Protocol operations . . . 105
5.1.3 Misbehavior detection . . . 109
5.1.4 Periodic signatures aggregation . . . 111
5.2 Correctness in bulk operations workloads . . . 115
5.2.1 Data model . . . 115
5.2.2 Setup and key generation . . . 119

5.2.3 Insert operations 121
5.2.4 Read operations 122

6 Conclusions 127

Bibliography 129

List of Figures

1.1 Actors of the Cloud database scenario. 2

3.1 MuteDB architecture. 19
3.2 Structure of a table metadata. 23
3.3 Organization of database metadata and table metadata in the metadata storage table. 24
3.4 Management of the encryption keys according to the field confidentiality parameter. 25
3.5 Architecture of MuteDB. 39
3.6 The poset representing a plaintext database. 43
3.7 Scheme of the structure of an encrypted database. 46
3.8 Scheme of the access and encryption groups of an encrypted database. 47
3.9 Encryption times of TPC-C benchmark operations grouped by transaction class. 58
3.10 Plain vs. encrypted SELECT and DELETE operations. 59
3.11 Plain vs. encrypted UPDATE and INSERT operations. 59
3.12 TPC-C performance (20 concurrent clients) 61
3.13 TPC-C performance (latency equal to 40 ms) 62
3.14 TPC-C performance (latency equal to 80 ms) 62
3.15 Distribution of the RTTs and ENC times for the 80 Planetlab clients. 64
3.16 Response times for the SQL operations in the TPC-C configurations. 65
3.17 Average response time of the most frequent TPC-C SELECT operation for different clients. 66
3.18 Response times for the SQL operations in the YCSB configurations. 67
3.19 Average response time of YCSB SELECT operation for different clients. 68

3.20 Throughput for increasing number of concurrent clients for different workloads and database configurations. 70

4.1 Estimated average network usage for each update operation as a function of the BF size m. 89
4.2 Estimated average network usage as a function of the BF size m for the stateless protocol. 94
4.3 Estimated average network usage for each update operation in terms of the BF size m. 98
4.4 Estimated average network usage for each update in terms of acceptable false positive rate ε. 98
4.5 BF size in terms of the acceptable false positive rate ε. 99

5.1 Candidate proxy-based architecture to implement Probus. . . 103
5.2 Message exchange for create operations. 106
5.3 Message exchange for read operations. 107
5.4 Message exchange for delete operations. 108
5.5 First verification of k within the current epoch. 114
5.6 Example of sets computed over a key-value database. 117

List of Tables

3.1 Notation for transactions and SQL queries. 30
3.2 Database tokens table. 49
3.3 Database encryption table. 49
3.4 Users tokens table. 50
3.5 Response times and overheads of SQL operations for different network latencies 60
3.6 YCSB workloads. 64

4.1 Database table enriched with a column of cryptographic digests. 73
4.2 Model parameters. 85
4.3 Storage and network usage comparison for VLH, TLH and EBF integrity solutions. 95
4.4 Analysis of the tables of a TPC-C compliant database 97

Introduction

Public cloud services represent an important opportunity for many enterprises and organizations attracted by high availability, scalability and elasticity guarantees arranged at pay-as-you-go prices. Their security is commonly guaranteed by the cloud provider by complying security regulations and certifications through adoption of best practices in terms of technology solutions and personnel policies. However, when critical information is placed in untrusted third parties' infrastructures, ensuring security of data is of paramount importance and should not be completely delegated to the service providers. Popular security incidents in terms of data breaches by external attackers, exfiltration by the cloud personnel, side-channels attacks by other cloud customers' virtual machines are open issues that prevent companies to trust public cloud services for storing confidential data. Foreign regulations for data management, access control and privacy laws pose additional problems to the cloud customers. This imposes clear data management choices: original plain data must be accessible only by trusted parties that do not include cloud providers, intermediaries, Internet; in any untrusted context data must be protected through cryptographic protocols. Satisfying these goals comes at the cost of increased performance overhead depending on the type of cloud service and on the security requirements, and needs complex architectures to distribute secret information to legitimate users.

In this thesis we focus on confidentiality and integrity and propose novel architectures and protocols that secure data stored in public cloud database services. Modern database softwares integrate a large number of security solutions, including data encryption, fine-grained access control enforcement, data replication, fault-detection and recovery. Although these solutions are adopted in cloud contexts by service providers, they are not able to satisfy strong security requirements by the service customers. In particular, we are interested in any scenario that considers cloud providers as untrusted parties, where even the system administrator of the cloud infrastructure might be interested in compromising security of the customers' data. For this reason, data owners must enforce confidentiality and integrity at their side through

cryptographic protocols that cannot be compromised even by a malicious cloud provider.

All the proposed architectures and protocols are tailored for cloud database services. They do not affect scalability of the cloud infrastructure and reduce overhead and with respect to state-of-the-art solutions. They can be adopted in real-world cloud database scenarios and avoid impractical cryptographic primitives and architectures that cause unacceptable performance and costs. As common in related literature, this thesis describes novel contributions to cloud database security in terms of their security guarantees. Let us summarize the main contributions of this thesis.

We describe a novel approach to design secure cloud database architectures, where all security information is distributed to legitimate clients without requiring the tenant to deploy additional in-house infrastructures or to trust intermediate brokers. We focus on confidentiality and integrate existing encryption protocols with original strategies that allow concurrent modifications by distributed clients and limit information leakage at the level of in-house databases. We demonstrate the feasibility of the proposal for public cloud database services through experimental results in Internet-based environments.

We propose a novel protocol that guarantees data authenticity and allows efficient detection of unauthorized data modifications. By adopting an original authentication structure, our protocol achieves low network overhead even in presence of read and update operations on small subsets of the authenticated data. We analyze the security guarantees of the protocol, the sizing boundaries to satisfy the required security level and a sizing methodology to minimize overhead with respect to the database characteristics and the operations workload. We demonstrate the benefits of the protocol by comparing its overhead to standard schemes based on message authentication codes through performance evaluation based on analytical models.

We propose two protocols to guarantee data integrity, including authenticity, completeness and freshness, in complex threat models and workloads that cannot be secured by adopting standard standard frameworks. We consider scenarios where tenants are able to produce cryptographic proofs of the cloud database misbehavior, and the cloud provider can detect and defend himself from false accusations. We describe a candidate architecture to implement the protocol and show that it produces efficient proofs with low response times. We also describe additional strategies that improve the protocol performance by periodically outsourcing most of the cryptographic proofs to the same cloud database without affecting the security guarantees of the scheme. Furthermore, we consider workloads that involve insertion of bulks of data and propose an original protocol that produces cryptographic

proofs of small size. We describe the high-level design of the original data model that is adopted by the protocol and an efficient candidate implementation.

This thesis consists of six chapters. In Chapter 1 we describe the scenario and the threat models considered in our proposals. Chapter 2 discusses related work and compare our proposals. Chapter 3 describes the design of the novel architecture to protect cloud database confidentiality. Chapter 4 describes the novel protocol for efficient authenticity guarantees. Chapter 5 describes two original protocols for efficient integrity guarantees in key-value databases. Finally, we discuss conclusion and future work in Section 6.

Chapter 1

System and security models

1.1 Scenario

In this thesis, we typically refer to the cloud database scenario represented in Figure 1.1. A company (*tenant*) arranges an agreement to store its data (*tenant data*) into a public cloud database service managed by a *cloud provider*. Both the tenant and the cloud provider include many employees that are delegated with different privileges on data and infrastructures depending on their role. We do not distinguish among the roles within the cloud provider because they are transparent to the tenant. Instead, we consider the main roles within the tenant company because they have implications in the design of secure architectures and protocols. We assume that the tenant includes a *database administrator* (DBA) that manages the setup of the cloud architecture and maintains the local infrastructure. He is in charge of installing and configuring the database and implements the *access control policies*. Many *tenant users* must access the cloud database by using client machines to insert, read, or execute computations on the tenant data. Tenant users are provided with unique *unique credentials* through which they may access to different portions of the data and have different access privileges on the cloud database. The tenant DBA manages the *user credentials* and distributes them to the tenant users. We assume that the *DBA* is the only role that has legitimate access to all tenant data. As in related literature, our threat models always assume that the DBA is trusted. Possible measures to verify the loyalty of the DBA, such as hashed logging, continuous monitoring and supervision, are outside the scope of this thesis. However, other cloud or tenant employees as well as people external to both companies might leak some information and compromise the security of the tenant data.

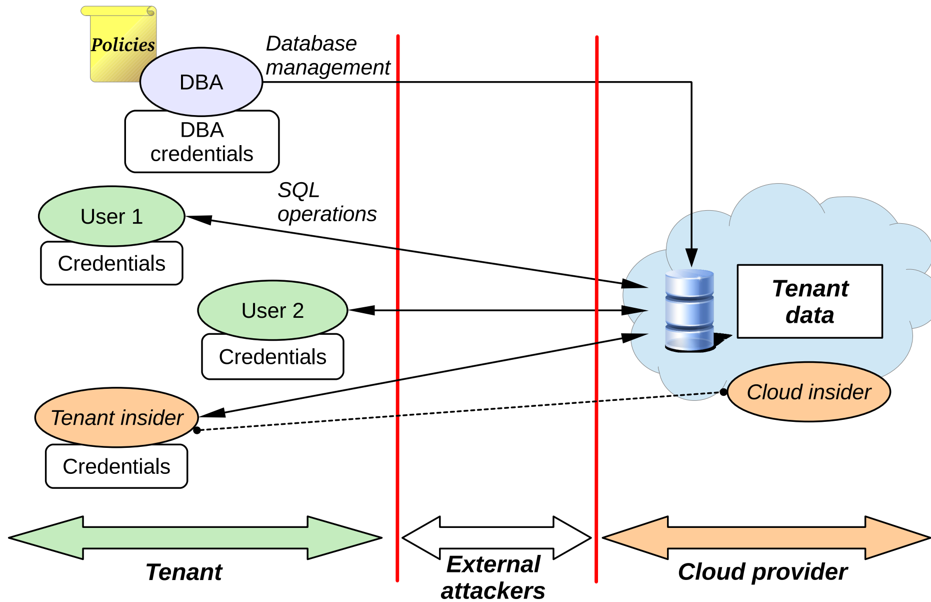


Figure 1.1: Actors of the Cloud database scenario.

1.2 Adversaries

Typical threat models in literature identify the possible issues related to the following roles: *external adversaries*, *tenant insiders* and *cloud insiders*.

1.2.1 External attackers

External adversaries have no legitimate access to the infrastructure and data of the tenant organization nor to those of the cloud provider. They can try to access tenant information through several types of attack: by eavesdropping data in motion between the tenant clients and the cloud servers, by compromising the cloud servers and/or the tenant clients. An external adversary can access tenant data by trying to eavesdrop or to manipulate data in motion between tenant clients and cloud servers; attacking cloud servers; attacking tenant clients. Communications between clients and servers can be secured against eavesdropping (and also other attacks such as Man-in-the-Middle) by using standard secure protocols, such as *Secure Socket Layer* (SSL). If an external attacker obtains unauthorized access to cloud servers, he can only access encrypted tenant data, because decryption keys are never stored unencrypted in the cloud database. Hence confidentiality of tenant data is guaranteed. On the other hand, if an attacker gains access to a tenant client that is actively using the secured database (e.g., a client that has an open

connection to the cloud database and has just issued some SQL queries) he can try to extract plaintext data and metadata that are temporarily cached in the client machine (data in use). If he is able to also access cryptographic keys and database credentials, the attacker can impersonate a legitimate customer. In this scenario, database access control mechanisms still apply, hence the attacker can only access data that are readable by the stolen identity.

1.2.2 Cloud insiders

The *cloud insiders* are employees of the cloud provider that have access to the cloud infrastructure hosting the database service of the tenant organization. They might have different access privileges on the tenant data, such as physical access to the disks, privileged accounts on the server machines or even complete access on the cloud infrastructure. The analysis on data confidentiality and integrity requires a differentiation among multiple adversaries profiles. Some adversaries are said *honest-but-curious* [57] (also *passive* or *semi-honest*) when they may be interested in accessing confidential data but never participate actively in the protocols, that is, they do not modify or delete data nor interact with authorized parties. Adversaries are said *malicious* (also *active*) when they aim to modify data or send incorrect messages/responses to legitimate parties of the protocol. An active attack might aim to let the authorized parties accept wrong results as well as to compromise data confidentiality [92]. Depending on the content of the database, a tenant might want to protect his data against passive or active attackers, or both. Assuming honest-but-curious cloud insiders is often considered realistic in related literature [36,49,62,95] because while reading data would remain unnoticed by a tenant, the detection of any data modification would penalize the trust and reputation of the cloud provider in the eyes of all its customers. However, the assumption might not be strong enough for small service providers and brokers or for critical data. Moreover, if a tenant publishes public information (e.g., winners of some contest, the list of his products) he has no interests in protecting the data but he must be sure that the data is never modified by untrusted parties.

1.2.3 Tenant insiders

Tenant insiders refer to tenant users having legitimate access to a subset of the tenant data stored in the cloud database. The portion of accessible data is defined by the access control policies of the tenant organization. Tenant insiders may try to gain access to more information by escalating their privileges through a violation of the access control policies.

The malicious operations of a tenant insider are limited by access control policies, but these policies cannot prevent the possibility that a tenant insider discloses its credentials including its secret key(s) to a cloud insider. The latter, that has access to all the tenant data, can bypass the access control policies enforced at the cloud side and violate confidentiality and integrity of the entire database by means of the key(s) received by the tenant insider. Moreover, a cloud insider may deliver some encrypted data to a tenant insider that is not authorized to access them. In this scenario, the tenant insider can leverage its credentials to decrypt data and violate the tenant access control policies. Recent cryptographic strategies [95] allow a tenant to store encrypted data thus preventing cloud insiders and external attackers from reading tenant data. Standard database access control mechanisms limit the operations of tenant insiders within their legitimate authorizations. Existing access control mechanisms at the database engine side guarantee confidentiality and isolation in traditional in-house deployments where the infrastructure is managed by trusted personnel.

In this thesis we describe an access control and key distribution strategies that fits encryption architectures that allow the computation of complex SQL operations on the database [48, 52]. As a result, our proposal limits information leakage even in presence of colluding tenant and cloud insiders, but maintains most of the capabilities of a plaintext database.

1.3 Attack models

We discuss possible attacks on the tenant data by an untrusted cloud insider with full privileges on the tenant data. We distinguish attacks in *passive attacks*, where we assume honest-but-curious cloud insiders, and *active attacks*, where cloud insiders are malicious and may participate actively in the database protocols and interact with legitimate users.

1.3.1 Passive attacks

We consider attacks on the confidentiality of data based on passive attacks, i.e. that are based on the analyses on the tenant data and on the protocols executed by legitimate parties. In this context, confidentiality of tenant data can be violated in two main scenario:

- by accessing the data while at rest, in motion or in use;
- by inferring information from the queries and from the data access patterns.

Preventing access to confidential information within data can be guaranteed by means of encryption. In this instance, we consider passive analyses on the encrypted data, where the adversary does not interact with authorized parties. The typical security level requirement for this attack model is to adopt encryption schemes that are semantically secure (also indistinguishable from random) under chosen plaintext attacks (IND-CPA) [66]. Informally, this requirement imposes that the attacker cannot infer any information on the plaintext content and data distribution by observing the ciphertext, even if he can get as many ciphertexts of known plaintext as he might need to analyze the encryption protocol, except those of interest. Please refer to established literature for details on the attack model, such as [66].

A cloud insider can read tenant data while they are stored (at rest) or are being accessed (in use) in the cloud database. If the tenant encrypts the cloud database and never send decryption keys to the cloud database, then confidentiality is guaranteed both for data at rest and in use. Hence, cloud insiders cannot interpret any tenant data stored in the database. However, if the tenant uses standard encryption algorithms [85] then the cloud database cannot execute any operations. The preferred solution is to use advanced encryption protocols that each supports some operations [11, 19, 43, 88, 89]. Some of these algorithms might leak some information about the data and are defined over attack models that represent weaker versions of the standard IND-CPA model (e.g., some encryption algorithms leaks the "order" of the plaintext data [19]). Literature considers these necessary trade-offs if the tenant needs practical solutions. As the main alternative is to send decryption keys to the cloud provider, then these trade-offs are by far the best solution to encrypt cloud databases. In this thesis we are not interested into the details and formal definitions of the security models for confidentiality as we do not propose novel encryption schemes. Instead, we propose an architecture that is able to leverage most of the existing encryption algorithms [11,19,43,88,89] in terms of confidentiality guarantees [49].

Even when data are encrypted, a family of sophisticated attacks exist that infer some information about the structure of the database through the analysis of the database operations (e.g., [27, 39]). As an example, it is possible to infer useful information by monitoring the nature and the order of SQL queries, as well as their frequency. Some information may leak if a cloud tenant leverages encryption schemes that sacrifice security to outsource computation over encrypted data. Moreover, access pattern analyses could reveal plain data distribution whose secrecy is critical for some encryption schemes [19, 64]). Some interesting solutions to prevent information leakage due to access patterns have been proposed in [5], [28], [42].

In this thesis we propose a novel encryption architecture that protects

data confidentiality at rest, in motion and in use, thus preventing information disclosure even to passive attacks by cloud insiders. Our proposal can be extended with techniques related to *private information retrieval* [28] and *query obfuscation* [74] that anonymize data retrieval patterns by introducing fake queries, but we leave integration details and implementation as future work. We do not focus on designing novel solutions to prevent information leakage due to access patterns. Nevertheless, there is no theoretical limit that prevents the integration of specific solutions into the proposed architectures.

1.3.2 Active attacks

Guaranteeing data integrity requires that legitimate tenant users are able to detect any form of unauthorized modifications, including incorrect results returned to users' queries. Modifications may be also caused by hardware or software failures, that should be avoided by the cloud provider, and represent violations of data integrity from the point of view of the tenant.

Enforcing cloud database integrity implies to make the tenant detect the database misbehaviors in terms of results *authenticity*, *completeness* and *freshness*.

- *Authenticity* ensures that the database returns only data that were previously inserted by the tenant, without including fake or manipulated values.
- *Completeness* ensures that the database returns all the data required by the tenant.
- *Freshness* ensures that the database returns the most updated version of the queries responses. As an example, freshness is not enforced if the answer of the database includes all (complete) and only (authentic) the due values for an old version of the database.

As common in literature, in this thesis we will refer to all completeness and freshness guarantees as *integrity* or *correctness* guarantees.

Authenticity guarantees are usually provided by means of message authentication codes (MAC) [12] and digital signatures respectively for symmetric and asymmetric scenarios. These algorithms applied to arbitrary data together with a secret key produces a *(cryptographic) digest*, that is a short representation of the input data. An attacker can violate data authenticity by forging a digest on behalf of authorized users. While the security level of signatures depends on the size of the key and on other parameters that are specific to the algorithm (e.g., the size of the primes used in RSA), the security level of a MAC depends separately on the key and the 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 [10]. The digest size 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 authorized parties.

Completeness and freshness for databases are usually guaranteed trough additional control structures that bind values stored in the database with each other (completeness) and with some timestamp or versioning information (freshness) by using, for example, authenticated Merkle hash trees or chained hashes. Although in static databases these are optimal solutions already implemented in many scenarios (e.g., to authenticate the content of Web mirrors), their adoption in dynamic databases cause high network overhead.

Chapter 2

Related work

We compare our proposals to state-of-the-art literature by distinguishing confidentiality and integrity solutions.

2.1 Confidentiality

Cryptographic file systems and secure storage solutions represent the earliest works in this field. We do not detail the several papers and products (e.g., Sporc [46], Sundr [69], Depot [72]) because they do not support computations on encrypted data. Different approaches guarantee some confidentiality (e.g., [1, 54]) by distributing data among different providers and by taking advantage of secret sharing [103]. In such a way, they prevent one cloud provider to read its portion of data, but information can be reconstructed by colluding cloud providers. A step forward is proposed in [64], that makes it possible to execute range queries on data and to be robust against collusive providers.

Our work is more related to proposals performing operations on encrypted databases [7, 62, 95] and enforcing access control at the encryption level [36, 112, 115], although the following reasons differentiate our architecture from the state of the art. Some DBMS engines and operative systems offer the possibility of encrypting data at the filesystem level through the so called Transparent Data Encryption feature [26, 73, 87], that makes it possible to build a trusted DBMS over untrusted storage. However, the DBMS is trusted and decrypts data before their use. Hence, these approaches are not applicable to the cloud database context, because we assume that the cloud provider is untrusted. Similarly, policy enforcement strategies [93] to enforce access control are not acceptable because modern threat models assume that a cloud provider employee could access tenant data. Other solutions, such

as [38], allow the execution of operations over encrypted data. These approaches preserve data confidentiality in scenarios where the DBMS is not trusted, however they require a modified DBMS engine and are not compatible with DBMS software (both commercial and open source) used by cloud providers. On the other hand, the proposed architecture is compatible with standard DBMS engines, and allows tenants to build secure cloud databases by leveraging services already available. For this reason, our work is more related to [62] and [95] that preserve data confidentiality in untrusted DBMSs through encryption techniques, allow the execution of SQL operations over encrypted data, and are compatible with common DBMS engines. However, the architecture of these solutions is based on an intermediate and trusted proxy that mediates any interaction between each client and the untrusted DBMS server. From the access control perspective, the proposed solutions are similar to that of an internally managed infrastructure where a trusted proxy stores all encryption and decryption keys, and clients access the encrypted database transparently.

Deploying a trusted *proxy* that mediates access of all the authorized clients to the cloud database in the tenant private network is a common solution in literature. This architectural choice may be feasible as, for example, many commercial solutions are based on a *cloud gateway* [55] that has to be deployed in premises, and trusted components may be co-located in pre-existing *hybric cloud* architectures [6]. However, if the tenant outsourced all his server infrastructure to the cloud, re-introducing in premises servers would cause high operational costs. Moreover, introducing intermediate servers could limit the scalability of the cloud database services, especially if the tenant must guarantee access to geographically distributed clients. The reliance on a trusted proxy that characterizes [62] and [95] facilitates the implementation of a secure cloud database service, and is applicable to multi-tier Web applications, which are their main focus. However, it causes several drawbacks. Since the proxy is trusted, its functions cannot be outsourced to an untrusted cloud provider. Hence, the proxy is meant to be implemented and managed by the cloud tenant. Availability, scalability, and elasticity of the whole secure cloud database service are then bounded by availability, scalability, and elasticity of the trusted proxy, that becomes a single point of failure and a system bottleneck. Since high availability, scalability and elasticity are among the foremost reasons that lead to the adoption of cloud services, this limitation hinders the applicability of [62] and [95] to the cloud database scenario. The proposed architecture solves this problem by letting clients connect directly to the cloud database, without the need of any intermediate component and without introducing new bottlenecks and single points of failure.

A proxy-based architecture requiring that any client operation should pass through one intermediate server is not suitable to cloud-based scenarios, in which multiple clients, typically distributed among different locations, need concurrent access to data stored in the same DBMS. On the other hand, the proposed architecture supports distributed clients issuing independent and concurrent SQL operations to the same database and possibly to the same data. The architecture in [7] avoids proxies, but adopts an access control mechanism that is based on a reference monitor within the cloud infrastructure and on a trusted authentication server. Some interesting solutions for enforcing access control policies on outsourced information are proposed in [36,60,112,115]. The encryption schemes in [115] allow a tenant company to outsource confidential information to the cloud, but they do not permit execution of SQL operations on encrypted data. The authors in [36] allow efficient key-value data retrieval in publish-subscribe scenarios where only one user is able to execute write operations. These architectures enforce access control through encryption at the record-level. However, they cannot be applied to a cloud database scenario where several users should be able to execute read and write operations as well as execute computations on encrypted data.

The approach proposed in [62] by the same authors that proposed the *database-as-a-service* model [63], works by encrypting blocks of data instead of each data item. Whenever a data item that belongs to a block is required, the trusted proxy needs to retrieve the whole block, to decrypt it, and to filter out unnecessary data that belong to the same block. As a consequence, this design choice requires heavy modifications of the original SQL operations produced by each client, thus causing significant overhead on both the DBMS server and the trusted proxy. Other works [70,80] introduce optimization and generalization that extend the subset of SQL operators supported by [62], but they share the same proxy-based architecture and its intrinsic issues. On the other hand, the proposed architecture allows the execution of operations over encrypted data through a mix of searchable, property-preserving and homomorphic encryption algorithms. This strategy, initially proposed in [95], makes it possible to execute operations over encrypted data that are similar to operations over plaintext data. In many cases, the query plan executed by the database for encrypted and plaintext data is mostly the same. Our architecture supports both static and adaptive encryption strategies, where layers of encryption are applied one on top of the other and removed on the cloud side when needed. In this thesis, we focus on static encryption strategies to while additional performance results are proposed in [50].

Our work shows that data consistency can be guaranteed for some operations by leveraging concurrency isolation mechanisms implemented in da-

tabase engines [47], and identifying the minimum isolation level required for those statements. Here, we present an architecture guaranteeing same security and confidentiality levels of an internally managed database in which the maximum information leakage that can be caused by a tenant insider is limited by his/her database access privileges [52]. We consider theoretically and experimentally a complete set of SQL operations represented by the TPC-C standard benchmark [108], in addition to multiple clients, and different client-cloud network latencies that were never evaluated in literature. Finally, it shows performance and scalability evaluations obtained in a real environment and for realistic workloads executed by clients that are dispersed over different geographical areas.

2.2 Integrity

In this thesis, we propose novel protocols to guarantee cloud database integrity for databases that supports data insertion and data retrieval. We do not consider the case of verifiable computation [91], where the tenant requires cryptographic proofs of correct computation of simple algebraic operations or generic programs, but we focus on databases where the main challenge is to achieve efficient solutions for data insertion and retrieval.

A well-known solution to guarantee integrity for files in cloud storage services is to associate a Message Authentication Code (MAC) tag to each file stored in the cloud. A cloud tenant can verify data integrity by downloading a file, recomputing the tag and comparing it to that returned by the database. 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. This approach may be viable for file storage services [98] because MACs are associated to data blocks of larger sizes that are accessed independently. The most viable design choice for cloud databases is to store each MAC tag to authenticate multiple attributes, because most primitive data types of databases are smaller than the tag. In particular, the common choice is to store a MAC tag with each row of database tables [90]. The disadvantage of this approach is 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 even if the cloud tenant wants to verify the integrity of only one of them. As a result, for certain types of databases and operation workloads the overhead of the additional transferred data causes higher network overhead than the transfer of the tags. This is quite common for relational databases where read operations often

request only small subsets of a table columns.

To comply with asymmetric settings, where some users must be able to verify authenticity but not to generate valid authentication structures, the same approach can be adopted by using digital signatures instead of MACs. However, a more interesting approach relies on cryptographic accumulators, proposed for the first time in [14], then implemented through different asymmetric cryptographic primitives (e.g., RSA accumulators [58], bilinear map accumulators [83]). They support set 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 [71] and source code repositories [23]. On the other hand, because of asymmetric cryptography they introduce much more computational overhead than MAC, that renders them inconvenient for symmetric settings, where the tenant users must be able both to generate and to verify valid authenticated structures.

Popular frameworks to guarantee authenticity and completeness are *authenticated data structures* [59, 106], which in turn often delegate authenticity guarantees to standard *digital signatures* (e.g., RSA, DSA, ECDSA [84]) or *message authentication codes* (e.g., HMAC [12]). We note that attacks against freshness represents a special case of replay attacks [66], a type of attacks well-known in communications security. Similar to secure communication protocols, cloud database integrity solutions can be integrated with secure versioning or timestamping protocols to guarantee freshness (e.g., persistent authenticated dictionaries [34], history trees [33]). On the one hand, by using standard frameworks one can design specific protocols for either authenticity, completeness or freshness, and then integrate them with existing solutions to guarantee database integrity as a whole security guarantee. On the other hand, these frameworks do not allow the design of efficient solutions when it comes to some types of databases, operation workloads and advanced threat models. For these reasons, one might need to design a novel integrity protocols that take into account all authenticity, completeness and freshness guarantees.

Very interesting results were presented in [34], where the authors compare persistent authenticated dictionaries (PADs) built upon RSA accumulators to PADs built upon symmetric MAC. The first solution has constant verification costs, while the latter are based on Merkle hash trees or skip lists and has asymptotic computational costs comparable to those of the adopted data structure. Despite their better asymptotic cost, experimental results based on software implementations conclude that RSA accumulators are never the preferable algorithm due to the high constant costs involved. In parallel

to our work and with motivations similar to ours, authors of [61] proposed a protocol based on *indistinguishable obfuscation* to avoid costly signatures generations. In their scenario, thin clients can generate authenticated structures by only executing symmetric encryption operations, while expensive verification operations are executed by more powerful semi-trusted servers. In this thesis we consider the symmetric setting and propose a secure variant of Bloom filters to verify data authenticity. By supporting set membership operations, our protocols allow the tenant to verify authenticity of subset of columns without retrieving all the other values stored at the same row. Our strategy improves network overhead of MACs and does not incur in the high computational costs of asymmetric cryptography.

The use of Bloom filters as cryptographic data structures exists in the literature, as in [41, 56, 86]. Authors of [56] design a variant of Bloom filters to build a secure index on encrypted data. The Bloom filter variant proposed by their scheme is similar to ours, however their protocol is designed for a different purpose and its security is analyzed in a different threat model. Hence, although the two proposals seem similar at a first glance, the purpose is very different and makes their work not comparable to ours. Authors of [86] analyze the use of Bloom filters as symmetric cryptographic accumulators. However, their computational cost is two orders of magnitude higher than that of RSA asymmetric accumulators [68] and thus they are not practical for any scenarios. Recent literature in the field of secure two-party computation [41] proposed a data structure based on Bloom filters and secret sharing to execute private set intersection protocols based on oblivious transfer. However, the size of the digest produced by their protocol is proportional to the square of the security level, while the size of the digest produced by our proposal is the same of standard Bloom filter algorithm, that is linear in the required security level. As a result, their proposal is not efficient in the cloud database scenario because it would cause unacceptable network and storage overhead. On the other hand, our proposal cannot be used as-is in the secure two-party computation setting because the parties cannot leverage set membership operations on the Bloom filters digests without decrypting them, that would compromise their security.

To the best of our knowledge, this is the first symmetric data authenticity scheme for cloud database services that guarantees efficient verification, low computational costs, and low overhead in terms of extra storage and network traffic thanks to the use of Bloom filters. By combining Bloom filters and 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 greatly reducing network and storage overhead.

Solutions that provide strong integrity, completeness and freshness guarantees have already been proposed in the field of secure management of audit logs [71, 102]. However all these solutions are designed for append only operations and cannot be extended to more complex workloads that also include update and delete operations. The authors in [107] propose an incremental and efficient data structure that combines Merkle Hash Trees and Bloom Filters for guaranteeing integrity, completeness and freshness in key-value databases. However, this paper (as well as all the aforementioned solutions) do not allow the tenant to demonstrate violations of data outsourced to cloud services to third parties, and do not permit the cloud provider to defend itself against false accusations by demonstrating its correct behavior. Closer to our paper is the proposal in [94] that allows a cloud tenant to obtain cryptographic proofs that testify violations of integrity, completeness and freshness of its data, and the cloud provider to defend against false accusations. One of the main ideas of their proposal is to exchange short authenticated messages for each operations, and to periodically build and exchange an authenticated Merkle Hash Tree on randomly sampled portions of the database. As a result, the number of values periodically selected for verification represents a trade-off between efficiency and security. We propose an original protocol [4] that improves over existing proposals in two aspects: our protocol also builds Merkle Hash Trees periodically, but they authenticate the operations executed within each time interval: this design choice makes it possible to achieve more efficient verification and faster epoch management; any tenant client can verify integrity, completeness and freshness of all results generated by the cloud service without any restriction on the trade-offs between the amount verified data and the achieved security level.

Another interesting issue is the efficiency of the protocol with respect to insertion and retrieval of large amounts of data. The common design choices in this field [24, 65, 79, 82, 94] do not provide any specific optimization for bulk operations. They transform bulk write operations into a set of write operations over a single key that are executed sequentially, thus incurring in protocol overheads for each value inserted in the key-value database. MHTs allow efficient range queries based on a pre-defined sorting convention. Proposals that rely on cryptographic accumulators [71] allows efficient bulk insert operations but, as for MHTs, support efficient reads of many records only if they are based on indexes. In this thesis we describe a new protocol that supports efficient read and write bulk operations with short cryptographic proofs. The size of the cryptographic proofs produced for arbitrary read operations are independent of the number of retrieved records. Closer to our

work is the scheme proposed in [24], that focuses on producing efficient proofs of correct execution for set operations. However, their proposal supports the memory checking setting that use position identifiers to build a N-ary tree of constant height (where only the values of the nodes change). Their protocol cannot be naively adopted in the cloud database scenario where new set of values are inserted for each operations, since the tree structure would require periodic costly re-balancing operations.

Chapter 3

An encryption architecture for distributed access

We propose a novel architecture that integrates cloud database services with data confidentiality named *MuteDB* (Multi User relaTional Encrypted DataBase). This architecture operates on an encrypted cloud database but it achieves the same level of confidentiality of an in-house database. It allows distributed clients to execute concurrent operations on the encrypted database [47], does not delegate the enforcement of access control policies to the cloud provider [48] and, even in the worst case of a collusion between a tenant and a cloud insider, limits the data leakage to the amount of information that is accessible to the colluding tenant insider. This result is achieved by combining original selective encryption strategies on the cloud database with the access control policies decided by the tenant [49, 52].

First, we propose the high-level design of the architecture in Section 3.1. Section 3.2 analyzes how independent clients can issue concurrent operations to the encrypted database, even those that modify the database schema. Section 3.3 describes how to extend the encrypted database architecture with cryptographic access control enforcement schemes that enable authentication and authorization of tenant users. Finally, Section 3.4 demonstrate the feasibility of the architecture in geographically distributed environments through experimental evaluation based on a software prototype in well-known database workloads.

3.1 Architecture design

Enforcing confidentiality of cloud databases needs design choices at the protocol level, such as the design of cryptographic or authentication protocols, and at the architectural level, for the management of the protocols metadata information and the distribution of cryptographic material to legitimate clients (e.g., security policies, cryptographic keys). MuteDB is designed to allow multiple and independent clients to connect directly to the untrusted cloud database without any intermediate server. Figure 3.1 describes the overall architecture. We assume that a *tenant* organization acquires a cloud database service from an untrusted database provider. The tenant then deploys one or more machines (*Client 1* through N) and install a *MuteDB client* on each of them. This client allows a user to connect to the cloud database to administer it, to read and write data, and even to create and modify the database tables after creation.

As we are interested in data confidentiality, we assume that the cloud provider is honest-but-curious (see Section 1.2.2), that is, cloud service operations are executed correctly, but tenant information confidentiality is at risk. In this first phase of the design we also assume that the tenant users are trusted and will not leak any confidential information. To protect confidentiality against honest-but-curious cloud insiders, tenant data, data structures, and metadata must be encrypted before exiting from the client.

To enforce data confidentiality we consider using a mix of encryption algorithm families: searchable [11], property preserving [19, 89] and partially homomorphic [43,88]. As first depicted in [95], each encryption algorithm allows the execution of some retrieval or algebraic operations on the encrypted data: the tenant selectively encrypts his data through some of the available encryption algorithms to support the operations workload and with respect to the security constraints [95]. Starting from this design choice, we build a novel architecture that allows completely distributed clients to execute SQL operations on a cloud database and enforces access control of the users at the encryption level.

The information managed by MuteDB includes *plaintext data*, *encrypted data*, *metadata*, and *encrypted metadata*. *Plaintext data* consist of information that a tenant wants to store and process remotely in the cloud database. To prevent an untrusted cloud provider from violating confidentiality of tenant data stored in plain form, MuteDB adopts multiple cryptographic techniques to transform plaintext data into *encrypted tenant data*, and *encrypted tenant data structures* because even the names of the tables and of their columns must be encrypted. MuteDB clients produce also a set of *metadata*

consisting of information required to encrypt and decrypt data as well as other administration information. Even metadata are encrypted and stored in the cloud database.

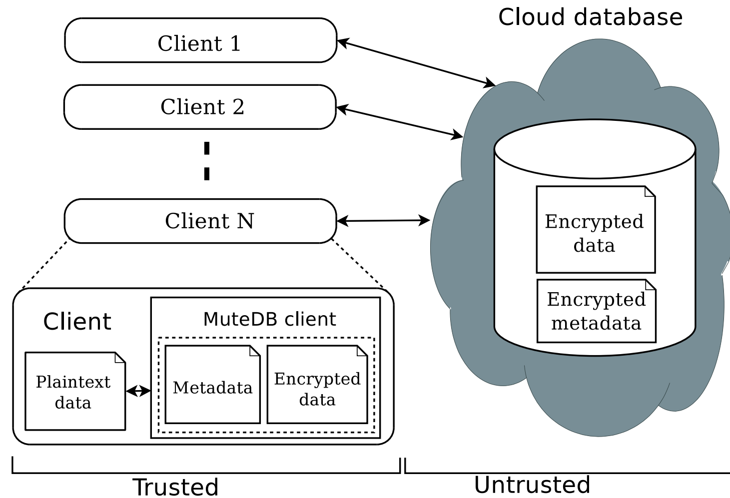


Figure 3.1: MuteDB architecture.

MuteDB moves away from existing architectures that store just tenant data in the cloud database, and save metadata in the client machine [62] or split metadata between the cloud database and a trusted proxy [95]. When considering scenarios where multiple clients can access the same database concurrently, these previous solutions are quite inefficient. For example, saving metadata on the clients would require onerous mechanisms for metadata synchronization, and the practical impossibility of allowing multiple clients to access cloud database services independently. Solutions based on a trusted proxy are more feasible, but they introduce a system bottleneck that reduces availability, elasticity and scalability of cloud database services.

MuteDB proposes a different approach where all data and metadata are stored in the cloud database. MuteDB clients can retrieve the necessary metadata from the untrusted database through SQL statements, so that multiple instances of the MuteDB client can access to the untrusted cloud database independently with the guarantee of the same availability and scalability properties of typical cloud database. Encryption strategies for tenant data, and innovative solutions for metadata management and storage are described in the following two subsections.

3.1.1 Data management

We assume that tenant data are saved in a relational database. We have to preserve the confidentiality of the stored data and even of the database structure because table and column names may yield information about saved data. We distinguish the strategies for encrypting the database structures and the tenant data.

Encrypted tenant data are stored through *secure tables* into the cloud database. To allow transparent execution of SQL statements, each plaintext table is transformed into a secure table because the cloud database is untrusted. The name of a secure table is generated by encrypting the name of the corresponding plaintext table. Table names are encrypted by means of the same encryption algorithm and an encryption key that is known to all the MuteDB clients. Hence, the encrypted name can be computed from the plaintext name. On the other hand, column names of secure tables are randomly generated by MuteDB, hence even if different plaintext tables have columns with the same name, the names of the columns of the corresponding secure tables are different. This design choice improves confidentiality by preventing an adversarial cloud database from guessing relations among different secure tables through the identification of columns having the same encrypted name.

MuteDB allows tenants to leverage the computational power of untrusted cloud databases by making it possible to execute SQL statements remotely and over encrypted tenant data, although remote processing of encrypted data is possible to the extent allowed by the *encryption policy*. To this purpose, MuteDB extends the concept of *data type*, that is associated to each column of a traditional database by introducing the *secure type*. By choosing a *secure type* for each column of a secure table, a tenant can define fine-grained encryption policies, thus reaching the desired trade-off between data confidentiality and remote processing ability. A *secure type* is composed by three fields: *data type*, *encryption type*, and *field confidentiality*. The combination of the *encryption type* and of the *field confidentiality* parameters defines the *encryption policy* of the associated column.

The **data type** represents the type of the plaintext data (e.g., int, string). The MuteDB client uses this information to decide how to encode data before encryption and how to decode them after decryption. As an example, if the MuteDB client encrypts strings and dates through order preserving encryption, that is usually designed on unsigned integers [19], it must know the data type to use a proper “order-preserving” encoding strategy (e.g., if we consider dates, the MuteDB client must encode dates to the integer domain and the encoded data must preserve their order). This information is used

in the decryption phase as well, because the MuteDB client must decode and cast the decrypted raw data to the due plaintext data type.

The **encryption type** identifies the encryption algorithm that is used to cipher all the data of a column. It is chosen among the algorithms supported by the MuteDB implementation. As in [95], MuteDB leverages several SQL-aware encryption algorithms that allow the execution of statements over encrypted data. It is important to observe that each algorithm supports only a subset of SQL operators (see Section 3.4.1 for details on the implemented algorithms in the current version of the software prototype). When MuteDB creates a secure table, the data type of each column of the encrypted table is determined by the encryption algorithm defined in the corresponding secure column. Two encryption algorithms are defined *compatible* if they produce encrypted data that require the same column data type.

As a default behavior, MuteDB uses a different encryption key for each column, hence equal values stored in different columns are transformed into different encrypted representations. (Even in the case of deterministic encryption algorithms, if same plaintext data are encrypted through different keys the resulting ciphertext data are indistinguishible from each other.) This design choice guarantees the highest confidentiality level, because it prevents an adversarial cloud provider to infer information from data that are repeated in different columns. However, to allow remote processing of SQL statements over encrypted data, sometimes it is required to encrypt different columns by means of the same encryption key. Common examples are the *join* queries and the *foreign key* constraint.

The **field confidentiality** parameter allows a tenant to define explicitly which columns of which secure table should share the same encryption key (if any). MuteDB offers three *field confidentiality* attributes:

- **Column** (COL) is the default confidentiality level that should be used when SQL statements operate on one column; the values of this column are encrypted through a randomly generated encryption key that is not used by any other column.

- **Multi-column** (MCOL) should be used for columns referenced by join operators, foreign keys, and other operations involving two columns; the two columns are encrypted through the same key.

- **Database** (DBC) is recommended when operations involve multiple columns; in this instance, it is convenient to use the special encryption key that is generated and implicitly shared among all the columns of the database characterized by the same *secure type*.

The choice of the proper field confidentiality levels make it possible to execute SQL statements over encrypted data while allowing a tenant to minimize key sharing.

3.1.2 Metadata management

Metadata generated by MuteDB contain all the information that is necessary to manage SQL statements over the encrypted database in a way transparent to the user. Metadata management strategies represent an original idea because MuteDB is the first architecture storing all metadata in the untrusted cloud database together with the encrypted tenant data. MuteDB uses two types of metadata.

- **Database metadata** are related to the whole database. There is only one instance of this metadata type for each database.
- **Table metadata** are associated with one *secure table*. Each table metadata contains all information that is necessary to encrypt and decrypt data of the associated secure table.

This design choice makes it possible to identify which metadata type is required to execute any SQL statement so that a MuteDB client needs to fetch only the metadata related to the secure table/s that is/are involved in the SQL statement. Retrieval and management of database metadata are necessary only if the SQL statement involves columns having the field confidentiality policy equal to *database*. This design choice minimizes the amount of metadata that each MuteDB client has to fetch from the untrusted cloud database, thus reducing bandwidth consumption and processing time. Moreover, it allows multiple clients to access independently metadata related to different secure tables, as we discuss in Sections 3.1.4 and 3.2.

Database metadata contain the encryption keys that are used for the secure types having the field confidentiality set to *database*. A different encryption key is associated with all the possible combinations of *data type* and *encryption type*. Hence, the database metadata represent a keyring and do not contain any information about tenant data.

The structure of a *table metadata* is represented in Figure 3.2. Table metadata contain the name of the related secure table and the unencrypted name of the related plaintext table. Moreover, table metadata include *column metadata* for each column of the related secure table. Each column metadata contain the following information.

- **Plain name**: the name of the corresponding column of the plaintext table.

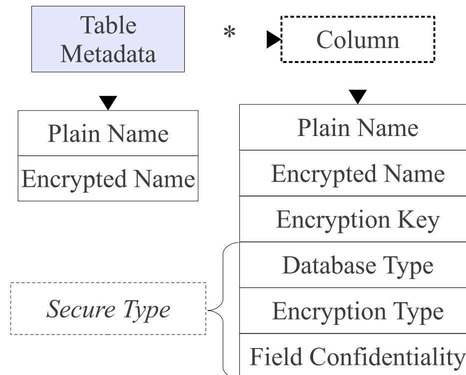


Figure 3.2: Structure of a table metadata.

- **Encrypted name**: the name of the column of the secure table. This is the only information that links a column to the corresponding plaintext column because column names of secure tables are randomly generated.
- **Secure type**: the secure type of the column, as defined in the previous Section 3.1.1. This allows a MuteDB client to be informed about the data type and the encryption policies associated to a column.
- **Encryption key**: the key used to encrypt and decrypt all the data stored in the column.

MuteDB stores metadata in the *metadata storage table* that is located in the untrusted cloud as the database. This is an original choice that augments flexibility, but opening two novel issues in terms of efficient data retrieval and data confidentiality. To allow MuteDB clients to manipulate metadata through SQL statements, we save database and table metadata in a tabular form. Even metadata confidentiality is guaranteed through encryption. The structure of the metadata storage table is shown in Figure 3.3. This table uses one row for the database metadata, and one row for each table metadata.

Database and table *metadata* are encrypted through the same encryption key before being saved. This encryption key is called *master key*. Only trusted clients that already know the master key can decrypt the metadata and acquire information that is necessary to encrypt and decrypt tenant data. Each metadata can be retrieved by clients through an associated *ID*, which is the primary key of the metadata storage table. This ID is computed by applying a Message Authentication Code (MAC) function to the name of the object (database or table) described by the corresponding row. The use of a deterministic MAC function allows clients to retrieve the metadata of a given table by knowing its plaintext name. This mechanism has the further

Metadata Storage Table

ID	*Encrypted Metadata*	*Control Structure*
MAC('.'+Db)	Enc(Db metadata)	MAC(Enc Db meta)
MAC(T1)	Enc(T1 metadata)	MAC(Enc T1 meta)
MAC(T2)	Enc(T2 metadata)	MAC(Enc T2 meta)

Figure 3.3: Organization of database metadata and table metadata in the metadata storage table.

benefit of allowing clients to access each metadata independently, which is an important feature in concurrent environments. In addition, MuteDB clients can use caching policies to reduce the bandwidth overhead.

3.1.3 Database setup

We describe how to initialize a MuteDB architecture from a cloud database service acquired by a *tenant* from a cloud provider. We assume that the DBA creates the metadata storage table that at the beginning contains just the database metadata, and not the table metadata. The DBA populates the database metadata through the MuteDB client by using randomly generated encryption keys for any combinations of *data types* and *encryption types*, and stores them in the *metadata storage table* after encryption through the *master key*. Then, the DBA distributes the *master key* to the legitimate users. User access control policies are administrated by the DBA through some standard data control language as in any unencrypted database.

In the following steps, the DBA creates the tables of the encrypted database. He must consider the three field confidentiality attributes (COL, MCOL, DBC) introduced at the end of the Section 3.1.1. Let us describe this phase by referring to a simple but representative example shown in Figure 3.4, where we have three secure tables named ST1, ST2 and ST3. Each table STi ($i = 1, 2, 3$) includes an *encrypted table* Ti that contains *encrypted tenant data*, and a *table metadata* Mi. (Although in the reality the names of the columns of the secure tables are randomly generated, for the sake of simplicity, this figure refers to them through C1-CN.)

For example, if the database has to support a join statement among the values of T1.C2 and T2.C1, the DBA must use the MCOL field confidentiality for T2.C1 that references T1.C2 (solid arrow). In such a way, MuteDB can retrieve the encryption key specified in the column metadata of T1.C2 from

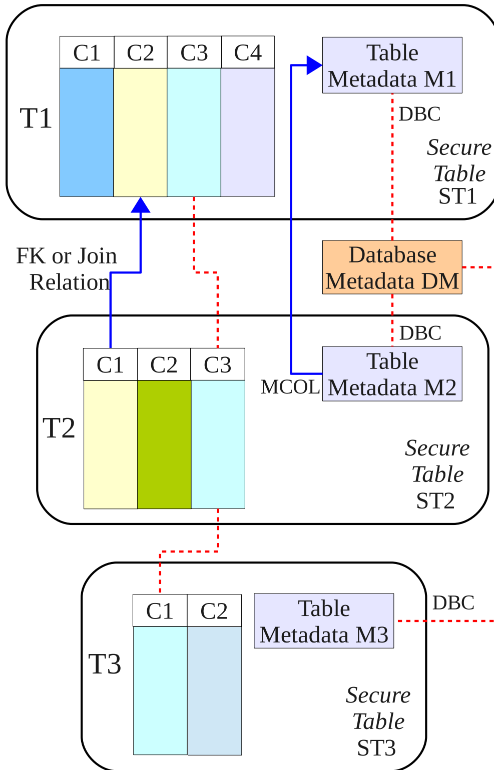


Figure 3.4: Management of the encryption keys according to the field confidentiality parameter.

the metadata table M1 and can use the same key for T2.C1. The solid arrow from M2 to M1 denotes that they explicitly share the encryption algorithm and the key.

When operations (e.g., algebraic, order comparison) involve more than two columns, it is convenient to adopt the DBC field confidentiality. This has a twofold advantage: we can use the special encryption key that is generated and implicitly shared among all the columns of the database characterized by the same *secure type*; we limit possible consistency issues in some scenarios characterized by concurrent clients (see Section 3.2). For example, the columns T1.C3, T2.C3 and T3.C1 in Figure 3.4 share the same *secure type*. Hence, they reference the *database metadata*, as represented by the dashed line, and use the encryption key associated to their data and encryption type. As they have the same data and encryption types, T1.C3, T2.C3 and T3.C1 can use the same encryption key even if no direct reference exists between them. The database metadata already contain the encryption key K associated with the data and the encryption types of the three columns, because

the encryption keys for all combinations of data and encryption types are created in the initialization phase. Hence K is used as the encryption key of the T1.C3, T2.C3 and T3.C1 columns and copied in M1, M2 and M3.

3.1.4 Operations execution

In this section we describe the execution of SQL operations on encrypted data in two scenarios: a naïve context characterized by a single client, and realistic contexts where the database services are accessed by concurrent clients.

Sequential operations

We describe the SQL operations in MuteDB by considering an initial simple scenario in which we assume that the cloud database is accessed by one client. Our goal here is to highlight the main processing steps, hence we do not take into account performance optimizations and concurrency issues that will be discussed in Sections 3.1.4 and 3.2.

The first connection of the client with the cloud database is for authentication purposes: MuteDB relies on standard authentication and authorization mechanisms provided by the original DBMS server. After the authentication, a user interacts with the cloud database through the MuteDB client. MuteDB analyzes the original operation to identify which tables are involved and to retrieve their metadata from the cloud database. The metadata are decrypted through the master key and their information is used to translate the original plain SQL into a query that operates on the encrypted database.

Translated operations contain neither plaintext database (table and column names) nor plaintext tenant data. Nevertheless, they are valid SQL operations that the MuteDB client can issue to the cloud database. Translated operations are then executed by the cloud database over the *encrypted tenant data*. As there is a one-to-one correspondence between plaintext tables and encrypted tables, it is possible to prevent a trusted database user from accessing or modifying some tenant data by granting limited privileges on some tables. User privileges can be managed directly by the untrusted and encrypted cloud database. The results of the translated query that includes encrypted tenant data and metadata are received by the MuteDB client, decrypted, and delivered to the user. The complexity of the translation process depends on the type of SQL statement.

Concurrent operations

The support to concurrent execution of SQL statements issued by multiple independent (and possibly geographically distributed) clients is one of the most important benefits of MuteDB with respect to state-of-the-art solutions. Our architecture must guarantee consistency among encrypted tenant data and encrypted metadata, because corrupted or out-of-date metadata would prevent clients from decoding encrypted tenant data resulting in permanent data losses. A thorough analysis of the possible issues and solutions related to concurrent SQL operations on encrypted tenant data and metadata is proposed in the following Section 3.2. Here, we remark the importance of distinguishing two classes of statements that are supported by MuteDB: SQL operations not causing modifications of the database structure, such as read, write, update; operations involving alterations of the database structure through creation, removal and modification of database tables (*Data Definition Layer* operators).

In scenarios characterized by a static database structure, MuteDB allows clients to issue concurrent SQL commands to the encrypted cloud database without introducing any new consistency issues with respect to unencrypted databases. After metadata retrieval, a plaintext SQL command is translated into one SQL command operating on encrypted tenant data. As metadata do not change, a client can read them once and cache them for further uses, thus improving performance.

MuteDB is the first architecture that allows concurrent and consistent accesses even when there are operations that can modify the database structure. In such cases, we have to guarantee the consistency of data and metadata through isolation levels, such as the snapshot isolation [15], that we demonstrate can work for most usage scenarios.

3.2 Concurrency management

The support to concurrent execution of SQL operations issued by multiple independent (possibly geographically distributed) clients is one of the most important benefits of the proposed architecture with respect to state-of-the-art solutions that require clients to issue queries to database through some intermediate (trusted) proxy. The drawback of scalability is that our architecture must guarantee consistency among encrypted tenant data and encrypted metadata accessed independently by multiple clients. Indeed, corrupted or out-of-date metadata would prevent clients from decoding encrypted tenant data with consequences of permanent data losses. In such a way, clients

can transform plaintext SQL statements into SQL operations that leverage transactions and isolation mechanisms provided by any relational database engine and cloud database.

Problems and solutions depend on the use of the database, that is, on the types of SQL operations that can be executed by clients. We present the adopted solutions to consistency issues in relation to five contexts:

- Data manipulation
- Structure modifications
- Altering tables
- Secure types modifications
- Unrestricted operations.

3.2.1 Data manipulation

In the Data Manipulation context, clients can read and write encrypted tenant data stored in the encrypted cloud database through the execution of SELECT, INSERT, DELETE and UPDATE commands. This set of SQL operations is indicated by the *DML* acronym. In this scenario, clients cannot modify the structure of the database by creating new tables or altering or dropping existing tables, hence we can assume that tables are created by the database administrator during a set-up period (see Section 3.1.3). Since only one client can access the cloud database while tables are being created, no concurrency issues arise here. Multiple and independent clients can access the cloud database and modify data only after the creation of all tables.

Plaintext SQL commands issued by users are translated by clients into queries that operate over encrypted tenant data stored in the cloud. The MuteDB client analyzes plaintext SQL commands to identify which plaintext tables are involved. Then, it issues a SELECT query over the metadata storage table to retrieve the table metadata of the corresponding secure tables. (We remark that MuteDB clients retrieve just the metadata related to the tables that are used in the plaintext SQL command issued by the users, thus minimizing the amount of information exchanged with the cloud database.) After that, clients generate exactly one translated SQL command for each plaintext SQL command issued by the users.

In this context, there are no consistency issues related to metadata management because metadata never change. However, multiple clients executing concurrent read and write commands over the same data set can lead to

inconsistencies over tenant data. These issues can be addressed by leveraging standard concurrency isolation mechanisms provided by the DBMS server used to provision the cloud database service. Each MuteDB client can enclose several SQL statements within a transaction by issuing BEGIN, COMMIT and ABORT commands. In this context, clients forward these commands to the cloud database without any modification. Hence, the cloud database executes concurrent transactions of translated queries in the same way as a traditional cloud database executes concurrent transactions of plaintext SQL commands. Consistency is guaranteed by the isolation level chosen by the database administrator among those implemented by the database, and are not influenced by the encryption and decryption operations.

3.2.2 Structure modifications

A tough context that has never been considered by state-of-the-art proposals about secure cloud databases is the possibility of modifying the structure of the database. On the other hand, MuteDB architecture supports even the execution of CREATE, DROP and DML commands on cloud databases. Unlike the previous scenario, in this context database metadata may change, hence clients cannot rely on the copy of metadata stored in their cache. For this reason, MuteDB translates each SQL command into a database transaction containing:

- the SQL queries necessary to retrieve the up-to-date metadata;
- the translated SQL commands that correspond to the original SQL command.

We analyze SQL commands execution by using the notation in Table 3.1, that is similar to that proposed in [15].

SQL commands translation. A plaintext SELECT query $R[T]$ that reads data from the table T is translated by the MuteDB client into the following transaction:

$$R[T] \Rightarrow B_t M_t^R[T] R_t[T_{enc}] C_t \tag{3.1}$$

MuteDB begins the transaction t (B_t), gathers up-to-date metadata for the desired table ($M_t^R[T]$), issues the translated SELECT query ($R_t[T_{enc}]$), and commits the transaction (C_t). Similarly, a plaintext write (INSERT, UPDATE or DELETE) command $W[T]$ to table T is translated by the MuteDB client into the following transaction:

$$W[T] \Rightarrow B_t M_t^R[T] W_t[T_{enc}] C_t \tag{3.2}$$

B_t	BEGIN operation of transaction t
C_t	COMMIT operation of transaction t
A_t	ABORT operation of transaction t
$R_t[T_{enc}, U_{enc}]$	Read (SELECT) operation on tables T_{enc},U_{enc} in transaction t
$W_t[T_{enc}, U_{enc}]$	Write (INSERT, UPDATE, DELETE) operation on tables T_{enc},U_{enc} in transaction t
$SM_t[T_{enc}]$	Structure Modification (CREATE or DROP) operation on table T_{enc} in transaction t
$ALT_t[T_{enc}]$	ALTER operation on table T_{enc} in transaction t
$M_t^R[T]$	Read operation on metadata related to table T in transaction t
$M_t^W[T]$	Write operation on metadata related to table T in transaction t

Table 3.1: Notation for transactions and SQL queries.

MuteDB begins the transaction t (B_t), gathers up-to-date metadata for the desired table ($M_t^R[T]$), issues the translated write command ($W_t[T_{enc}]$), and commits the transaction (C_t). $M_t^R[T]$ indicates a read operation belonging to the transaction t over the row of the *metadata storage table* that contains metadata related to the table T. If this operation involves metadata related to multiple tables, all of them are indicated. As an example, the notation $M_t^R[T, U]$ denotes a read query over metadata related to the tables T and U.

A plaintext DDL command $DDL[T]$ on the table T is translated into the following transaction:

$$DDL[T] \Rightarrow B_t M_t^R[T] M_t^W[T] DDL_t[T_{enc}] C_t \tag{3.3}$$

MuteDB begins the transaction t (B_t), gathers up-to-date metadata for the desired table ($M_t^R[T]$), modifies the metadata ($M_t^W[T]$), issues the translated DDL command ($DDL_t[T_{enc}]$) and commits the transaction (C_t). DDL operations over the encrypted database are always executed after write operations over the related metadata. This design choice is motivated by the behavior of many popular databases that perform an implicit commit after each DDL. Translated DDL operations are always executed at the end of an implicit transaction, hence for a client and a user MuteDB behaves as an unencrypted cloud database.

MuteDB allows also users to execute transactions on plaintext data. However, by following the described translation process, data consistency would require to translate each SQL command into an implicit transaction, and the SQL standard does not support nested transactions. We solve this problem by not translating each plaintext SQL command that belongs to the same explicit transaction into one implicit transaction, but by using only one translated explicit transaction on encrypted data. As an example, the explicit plain transaction $B_t R_t[T, U] W_t[T] C_t$ is translated into the following transaction over the encrypted database:

$$B_t M_t^R[T, U] R_t[T_{enc}, U_{enc}] W_t[T_{enc}] C_t \tag{3.4}$$

Since metadata related to the table T are already get before executing the read query, there is no need for reading them again in the same transaction before executing the write command on the table T_{enc}.

A possible performance optimization can be achieved by caching a local copy of metadata that has already been used in previous SQL commands. However, since metadata may change, a MuteDB client must guarantee that the cached metadata are not outdated before using it. This check is carried out by reading the current MAC associated to the cached metadata from the *metadata storage table*. If this MAC is equal to its cached copy, then cached metadata are up-to-date. Otherwise, the cached metadata are outdated and the MuteDB client needs to perform a second access to the *metadata storage table* to read up-to-date metadata and its MAC.

3.2.3 Consistency management

Each plaintext SQL command executed in unencrypted databases is an atomic operation. However, we must translate each atomic command into a sequence of multiple commands enclosed in a transaction. In such case, data consistency is guaranteed by choosing a sufficient transaction isolation level among those offered by the cloud database.

If the isolation level is insufficient, consistency issues may arise from the execution of operations belonging to different but concurrent transactions. If concurrent transactions operate just on encrypted tenant data, metadata are not modified and we return to the data manipulation context analyzed in Section 3.2.1, in which the database administrator can choose the isolation level among those provided by the DBMS. On the other hand, consistency issues may arise when a concurrent transaction contains commands that modify metadata. Among the considered SQL commands, only CREATE and DROP operations modify metadata, hence consistency issues may arise if concurrent executions of the following commands occur:

- DROP and DML;
- CREATE and DML;
- any concurrent CREATE and DROP.

DROP and DML. The database may generate errors if the DML command is executed after the table has been dropped. For example, let us consider the following two transaction histories.

The former represents the execution of a table DROP, while a data read is being executed on the same table:

$$B_1 B_2 M_1^R[T] M_2^R[T] M_2^W[T] SM_2[T_{enc}] C_2 R_1[T_{enc}] A_1 \quad (3.5)$$

Transaction 1 obtains metadata that are necessary to create the translated read command $R_1[T_{enc}]$ and to decrypt its result. The DROP command ($SM_2[T_{enc}]$) issued by transaction 2 is executed before the translated data read is issued by the transaction 1. The table T_{enc} does not exist anymore and the read command issued by transaction 1 fails.

We now consider the concurrent execution of a DROP and a write command:

$$B_1 B_2 M_1^R[T] M_2^R[T] M_2^W[T] SM_2[T_{enc}] C_2 W_1[T_{enc}] A_1 \quad (3.6)$$

In this context, the write command executed by transaction 1 fails because T_{enc} was deleted by the DROP command ($SM_2[T_{enc}]$).

CREATE and DML. The database may generate errors if the DML command is executed before the creation of the table. As an example, we consider the following two transaction histories.

The former represents the execution of a table CREATE, while a data read is being executed on the same table:

$$B_1 B_2 M_2^R[T] M_2^W[T] M_1^R[T] R_1[T_{enc}] A_1 SM_2[T_{enc}] C_2 \quad (3.7)$$

The read command executed by transaction 1 fails because T_{enc} was not yet created by the CREATE operation ($SM_2[T_{enc}]$).

In the concurrent execution of a CREATE and a write command, we have:

$$B_1 B_2 M_2^R[T] M_2^W[T] M_1^R[T] W_1[T_{enc}] A_1 SM_2[T_{enc}] C_2 \quad (3.8)$$

The write command executed by transaction 1 fails because T_{enc} was not yet created by the CREATE ($SM_2[T_{enc}]$).

In all these cases, the client software handles the error notification generated by the remote database. It is important to remark that that no considered error causes consistency issues to the encrypted tenant data or metadata in the cloud.

Any concurrent CREATE and DROP. The database may generate errors if two commands modifying the structure of the database are executed concurrently. For example, if two CREATE (DROP) commands insist on the same table, then an error is generated as soon as the second transaction inserts (deletes) the related metadata, as represented by the following history case.

$$B_1 B_2 M_1^R[T] M_2^R[T] M_1^W[T] M_2^W[T] A_2 SM_1[T_{enc}] C_1 \tag{3.9}$$

If a CREATE and a DROP are executed concurrently over the same table, an error can be generated because the DROP is executed on a table that does not exist yet, or because a client creates an already existing table. The following history represents a failed CREATE (DROP) command by the transaction 2 executed before the other DROP (CREATE) command by the transaction 1.

$$B_1 B_2 M_1^R[T] M_1^W[T] M_2^R[T] M_2^W[T] SM_2[T_{enc}] A_2 SM_1[T_{enc}] C_1 \tag{3.10}$$

Since the transaction 2 aborts, its previous modification on related metadata ($M_2^W[T]$) is rolled back (A_2).

Another possible scenario involves the creation of two different tables, that both contain a column with *database field confidentiality* policy. Since database metadata is created during database setup (as discussed in Section 4.1), both transactions do not modify database metadata. Hence, no consistency issue arises.

In this context, the use of implicit transactions is sufficient to guarantee data consistency, hence the database administrator can freely choose the preferred isolation level among those provided by the database.

3.2.4 Altering tables

In this context, we assume that users can execute any command considered in the previous contexts, and they can issue ALTER commands to delete (add) columns from (to) existing tables.

This scenario does not introduce new concurrency issues with respect to the concurrent execution of DDL and DML commands. However, DML commands may fail because they may try to access to newly created columns

that do not exist yet, or to a column that has already been deleted. These scenarios are similar to those discussed in Section 3.2.2 and represented by histories (3.5), (3.6), (3.7) and (3.8).

Concurrency issues may arise if two ALTER commands are executed concurrently. As an example, let us consider two ALTER commands issued by two clients A and B that are transformed as described by (3.3). A possible history of the two concurrent executions is the following:

$$B_1 B_2 M_1^R[T] M_2^R[T] M_1^W[T] M_2^W[T] ALT_1[T_{enc}] ALT_2[T_{enc}] C_1 C_2 \quad (3.11)$$

In this example each client modifies the metadata related to table T, and it executes its ALTER operation ($DDL_1[T_{enc}]$ and $DDL_2[T_{enc}]$). Both transactions commit successfully in our architecture. However, metadata related to table T reflect only the modifications written by the transaction 2, while table T_{enc} maintains modifications of both DDL commands. Hence, table T_{enc} and its metadata are inconsistent.

This concurrency issue is an instance of the *lost update* consistency phenomenon as defined in [15].

If multiple clients can execute DDL concurrently, consistency is guaranteed only if the isolation level provided by the remote database prevents *lost updates*. These guarantees are provided by a snapshot isolation level [15], that is now implemented in all popular DBMS engines.

3.2.5 Secure types modifications

MuteDB introduces new ALTER commands that modify the *secure type* of *encrypted tenant data*. We refer to these commands as *securetype modification language* (SML). SML commands can modify all the security type parameters and the encryption key used to encrypt tenant data. In particular, some SML commands can be executed without modifying the database structure, hence they do not require DDL commands. All the SML commands that can change the field confidentiality level and the encryption key belong to this category. Moreover, compatible data types and compatible encryption algorithms (as discussed in Section 3) can be converted without changing the structure of the database table storing encrypted tenant data.

In this context, users can execute all the commands allowed in Section 3.2.4, and any SML command that does not modify the database structure. Since these SML commands change just metadata and encrypted tenant data, the translation process can be modeled as following.

$$SML \Rightarrow BM^R[T]M^W[T]R[T_{enc}]W[T_{enc}]C \quad (3.12)$$

As an example, let us consider an SML command modifying the encryption key that is used to encrypt tenant data stored in the table T_{enc}. In such a case, MuteDB first reads the current metadata ($M^R[T]$) associated with the encrypted tenant data to retrieve all the information related to their encryption policy, including current encryption keys. Then, it updates the metadata ($M^W[T]$) according to the new encryption policy, by changing the encryption keys. As a consequence, MuteDB needs to read all data ($R[T_{enc}]$), to decrypt them through the old encryption keys, to encrypt them through the new encryption keys and to write new data to the encrypted table ($W[T_{enc}]$). Decryption and encryption operations must be performed locally by a trusted client because MuteDB never exposes plaintext data to the untrusted cloud database.

Consistency issues may arise in the following cases of concurrent executions:

- SML command and data read
- SML command and data write
- multiple SML commands.

SML command and data read. The database may return data that are not accessible by the client, if a data read command is executed concurrently with an SML command. We consider the case in which a data read command requires a set of data whose encryption key is being modified by a concurrent SML command, as represented by the following transaction history:

$$B_1 B_2 M_1^R[T] M_2^R[T] M_2^W[T] R_2[T_{enc}] W_2[T_{enc}] C_2 R_1[T_{enc}] C_1$$

In this example, the transaction 1 reads metadata ($M_1^R[T]$). Then, the transaction 2 executes sequentially all operations included in the SML command as defined in (3.12). Finally, the transaction 1 reads the set of data ($R_1[T_{enc}]$). However, it obtains data that are encrypted through a new encryption key, hence it cannot decrypt them. This concurrency issue is an instance of the well known *read skew* anomaly that was explained in [15].

SML command and data write. Inconsistent data may be written if a data write command and a SML command are executed concurrently. We consider the case in which a data write command stores a set of data whose encryption key is being modified by a concurrent SML command. This scenario is represented by the following transaction history.

$$B_1 B_2 M_1^R[T] M_2^R[T] M_2^W[T] R_2[T_{enc}] W_2[T_{enc}] C_2 W_1[T_{enc}] C_1$$

In this example, the transaction 1 reads metadata ($M_1^R[T]$), then the transaction 2 executes sequentially all operations included in the SML command, as defined in (3.12). Finally, the transaction 1 writes the set of data ($W_1[T_{enc}]$). The problem is that it writes data that are encrypted by means of the old encryption key that is not stored anymore in the metadata related to the table T_{enc}. As a consequence, these data are inaccessible. The consistency anomaly that affects the above history may differ on the basis of the considered write command. We distinguish two main cases: UPDATE or DELETE commands, and INSERT commands.

In the case of an UPDATE or a DELETE command, the data write command ($W_1[T_{enc}]$) insists on a set of data that is involved also by the SML command ($W_2[T_{enc}]$). Hence, the concurrency issue is an instance of the *lost update* phenomenon, as defined in [15].

In the INSERT case, the data write command insists on a set of data that did not exist when the SML command was executed, but that is included in the predicate of the update sequence of the SML command ($R_2[T_{enc}]W_2[T_{enc}]$). This concurrency issue is an instance of the so called *phantom anomaly* [15].

We highlight that an alternative example of the above transaction history is to swap the order of the last writes operations, as represented by the following history:

$$B_1 B_2 M_1^R[T] M_2^R[T] M_2^W[T] R_2[T_{enc}] W_1[T_{enc}] C_1 W_2[T_{enc}] C_2 \qquad (3.13)$$

In the case of an UPDATE or DELETE command, the database is still consistent and completely accessible. However, newly written data have been lost because of the *lost update* phenomenon.

Multiple SML commands. We consider the case in which two SML commands are executed on the same fields of the same table through the following history:

$$\begin{gathered} B_1 B_2 M_1^R[T] M_2^R[T] M_1^W[T] M_2^W[T] \\ R_1[T_{enc}] R_2[T_{enc}] W_1[T_{enc}] C_1 W_2[T_{enc}] C_2 \end{gathered} \qquad (3.14)$$

Since both transactions modify the same metadata ($M_1^W[T]\ M_2^W[T]$), this history may cause a *lost update* anomaly.

Finally, we can define the consistency requirements of the *secure-types-modifications* context, where the DBMS isolation level must avoid *read skew*, *lost update* and *phantom* concurrency anomalies. Since *lost update* is a subcase of a *read skew* [15], it is possible to trace back the two anomalies to only *read skew*.

MuteDB guarantees data consistency by leveraging the appropriate isolation level. The *read skew* anomaly is avoided by the snapshot isolation level, that does not guarantee consistency with respect to the *phantom* anomaly. Besides the highest ANSI *serializable*, no standard isolation level with similar guarantees has been defined yet. We can observe that several popular DBMS engines extend snapshot isolation through predicate locking mechanisms, thus avoiding also *phantom* anomalies. We call the set of snapshot isolation levels that avoid also *phantom* anomalies as the *snapshot isolation plus*.

If the required isolation level is set on the cloud database, MuteDB lets several clients execute DML commands concurrently while one client executes SML operations on a table with no consistency issues. An isolation level that suits our requirements with low overhead has been proposed in [114].

3.2.6 Unrestricted operations

This context does not pose any constraint to the nature of the commands that can be issued concurrently by the clients to the cloud database. For example, it is possible to execute any data definition language (DDL) command, as well as DML commands, and SML commands that modify the database structure and encryption policies. Since the behavior of DDL is not formalized in any database standard, each DBMS implements different DDL locking mechanisms and DDL transaction policies. Hence, it is impossible to identify one isolation level that does not depend on a specific database and that guarantees data consistency. A possible solution is to impose the isolation level *serializable* [15] together with the support to rollback of DDL operations that are included in the transactions.

If these constraints are satisfied, then the MuteDB architecture guarantees data consistency in any execution context. Since these constraints are not met by all the DBMS engines, another solution is to explicitly handle concurrency issues at the application level. This problem is out of the scope of this architecture because it would depend on the guarantee level provided by the specific cloud database.

3.2.7 Discussion

In scenarios characterized by a static database structure (as described in Section 3.2.1), the MuteDB architecture allows multiple, independent and possibly geographically distributed clients to issue concurrent SQL commands to read, write and update data stored in an encrypted cloud database. In summary, it is worth to observe that:

- in the data manipulation context, that is considered by multiple previous proposals (e.g., [37, 62, 95]), a negligible overhead is generated. Clients can read metadata and cache them locally without any consistency issue;

- in all contexts the proposed architecture does not introduce any additional consistency issue with respect to unencrypted databases;

- any underlying mechanism implementing database operations is transparent to the users.

Some inevitable overhead is caused by the computational cost related to data encryption and decryption operations. However, this cost is inherent in any encrypted database solution that does not want to expose plaintext data to the cloud provider insiders.

We highlight also that the MuteDB is the first solution that allows concurrent and direct accesses to the cloud database and that supports even modifications to the database structure. Depending on the type of modification, higher isolation levels are required with consequent overheads. If we have to support operations such as CREATE and DROP tables (Section 3.2.2) or SML commands (Section 3.2.5), then the proposed solution introduces some additional operations to implement implicit transactions, but in any case the data consistency is guaranteed even if multiple clients access the cloud database concurrently and independently from different geographic locations.

3.3 Cryptographic access control enforcement

In this section we extend the architecture described in Section 3.1 by considering stronger security guarantees that include the possibility of authorized database users colluding with a cloud provider. The novel architecture protects guarantee data isolation and confidentiality on any relational cloud database service rented by a tenant organization and maintains the possibility for distributed clients to access the encrypted database independently and concurrently.

In Fig. 3.5 we evidence a tenant organization in which a trusted DBA machine hosts the MuteDB DBA client, that is the application for the creation and management of the encrypted database. All tenant database users can issue SQL operations directly to the cloud database even from geographically distributed locations by executing a MuteDB client on their machines. The entire set of *tenant data* are stored in an encrypted form in the cloud

database. Thanks to the use of SQL-aware encryption strategies, the cloud database engine can execute queries on encrypted data without accessing any decryption keys. Even *metadata* that are necessary to manage encryption strategies are considered critical information, hence MuteDB stores them encrypted in the cloud database: the DBA and the tenant users can efficiently retrieve metadata through standard SQL queries. We refer to the encrypted forms of tenant data and metadata as *encrypted tenant data* and *encrypted metadata*, respectively.

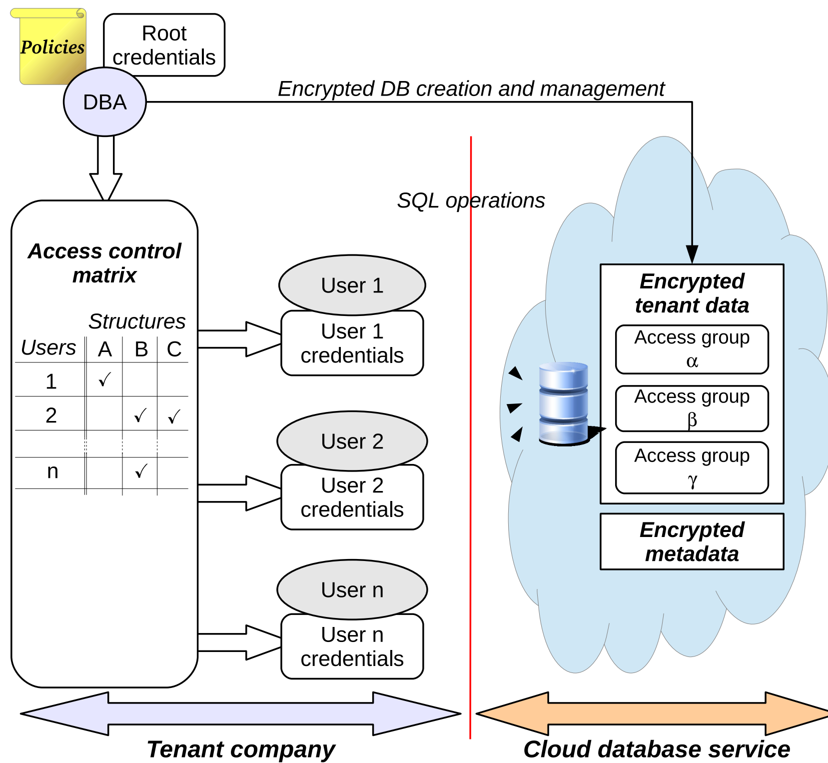


Figure 3.5: Architecture of MuteDB.

Unlike existing proposals, MuteDB does not use any trusted intermediate proxy [95] and key distribution server [7], nor it stores large amounts of cryptographic information and metadata in the client machines [37]. We assume that the DBA is the only subject that owns *root credentials* for the DBA client, and that no internal nor external attackers are able to access, steal or crack the credentials. The DBA manages user accounts, and enforces the tenant *access control policies*. These policies represent the set of rules adopted by the tenant organization to define which user can access to which subset of tenant data. The importance of data isolation through access control policies should be clear: the tenant users must access all and only authorized data

where authorizations are specified as if the database was maintained by the tenant. On the other hand, the mechanisms for implementing access control policies are complicated by the cloud database service scenario. MuteDB offers the following original solutions. Each user is provided with a set of *user credentials* including all information that allows him/her to access all and only the legitimate data. The encrypted data cannot maintain the same structure of the plaintext version, and the wide literature on enforcing access control policies on relational databases (e.g., [8, 16]) does not propose how to extend these policies on SQL-aware encrypted cloud databases. Hence, to the best of our knowledge, this model is the first addressing the issue of transforming authorization rules expressed on a plaintext database into rules enforced in the SQL-aware encrypted database.

The access control matrix is the most common solution for describing discretionary access control policies [7, 99, 100]. Each row is associated with a database user and each column is associated with a *structure* (e.g., column, table, database) that is defined as a subset of tenant data on which it is possible to apply an *authorization rule*. Each cell of the access control matrix defines whether a user can or cannot access the corresponding structure. For example, the access control matrix in Fig. 3.5 denotes that user 1 and user 2 are allowed to access the structure A, and the structures B and C, respectively. We propose an original model that maps the 1:1 correspondence between the sets of plaintext data and the encrypted data on which the tenant access control policies are defined. For example, in Fig. 3.5 MuteDB maps plaintext tenant data A, B, and C into encrypted tenant data α, β, and γ, respectively. The access control policies are satisfied by enforcing any authorization rule expressed over a plaintext structure on the corresponding *access group* (e.g., A and α). The details of our model and solution are described in Section 3.3.4.

A similar solution works for a database stored in-house, but it does not guarantee the confidentiality of data stored in the cloud because a cloud insider can access the storage devices. Hence, MuteDB enforces the access control policies through selective encryption strategies. Selective encryption requires the encryption of data through multiple encryption keys at a granularity that depends on the reference access control model. Since our target is a discretionary access control model that is expressed over database structures, we use a different encryption key for each structure of the encrypted database. Each user credentials include small cryptographic information consisting of a unique secret key that allows him/her to calculate the database decryption keys through derivation algorithms (e.g., [9, 32]). This choice avoids the generation and distribution of new credentials even if the access control policies change (Section 3.3.7). We note that our proposal

can be combined with symmetric or asymmetric SQL-aware encryption algorithms; moreover, the derivation scheme is designed to fit symmetric, private or public keys of different lengths.

We conclude this section by describing the main operations required to create and access the encrypted cloud database. The DBA is in charge of translating the access control policies into an *access control matrix* used by MuteDB. The DBA client takes as its input the original plaintext database, and produces the encrypted tenant data. The structures of the plain database are mapped to access groups within the encrypted tenant data. In the example of Fig. 3.5, the structure A is mapped to the encrypted access group α. Moreover, the DBA client produces metadata that are encrypted and then stored in the cloud database. The DBA distributes unique secret keys to the users at the creation of their accounts according to the access control matrix. These keys enable the users to access (decrypt) all and only the subsets of encrypted tenant data corresponding to the structures on which the users have legitimate access. In the example of Fig. 3.5, the user 2 credentials can only be used to decrypt data included in the access groups β and γ. Each user can execute SQL operations through the MuteDB client installed on his/her client machine. The client takes as its inputs the user credentials and the encrypted metadata stored in the cloud database, and translates plaintext SQL operations into encrypted SQL operations that can be executed on encrypted data. MuteDB guarantees the isolation of the tenant data and protects also the names of the database structures by enforcing access control on all these sets of information. This solution avoids that a cloud insider infers some information about the content of the database by knowing the names of the tenant database structures. Our choice of subjecting structure names to access control enforcement guarantees data isolation and confidentiality but it complicates the management of encrypted queries and metadata retrieval. In the following Sections 3.3.1, 3.3.2, 3.3.3 we describe models to represent access control rules, plaintext and encrypted database, respectively. The detailed solutions of MuteDB for access control through encryption and for metadata management are described in Section 3.3.4 and Section 3.3.5.

3.3.1 Access control rules

We now introduce the MuteDB models and schemes for combining encryption and key management to support data confidentiality and isolation in cloud databases. After the presentation of the models related to access control in plaintext (Section 3.3.2) and encrypted (Section 3.3.3) databases, we describe how MuteDB transforms an access control matrix for the plaintext model to

a matrix suitable for the encrypted database (Section 3.3.4), and how it generates user credentials (Section 3.3.4).

Let $\mathcal{R}$ be the set of resources that represent plaintext tenant data, $\mathcal{S}$ the set of plaintext database structures, $\mathcal{E}$ the set of encrypted tenant data, $\mathcal{U}$ the set of users, and $\mathcal{K}$ the set of encryption keys. We define $\mathcal{A}$ as the access control matrix where, for each user $u \in \mathcal{U}$ and for each structure $s \in \mathcal{S}$, there exists a binary authorization rule $a \in \mathcal{A}$ that defines whether an access to s by u is denied ($a_{u,s} = 0$) or allowed ($a_{u,s} = 1$).

The user u *capability list* cap_u denotes the set of structures accessible to u. We assume the existence of a decryption function $D : \mathcal{E} \times \mathcal{K} \longmapsto \mathcal{R}$ such that for each encrypted resource $e \in \mathcal{E}$, there exists a key $k \in \mathcal{K}$ that allows us to calculate $r = D(k, e)$, where $r \in \mathcal{R}$. For the sake of simplicity, we define $e_r \in \mathcal{E}$ and $k_r \in \mathcal{K}$ as the encrypted resource and the decryption key for the resource $r \in \mathcal{R}$, that is, $r = D(k_r, e_r)$. For each *user* $u \in \mathcal{U}$, we define the *keyring* $\mathcal{K}_u \subseteq \mathcal{K}$ as the set of all the decryption keys known by u, and the *user accessible resources* $\mathcal{R}_u$ as the set of all and only resources that u is able to decrypt through the keys included in $\mathcal{K}_u$. The idea is that an encryption scheme can enforce tenant access control policies if the users keyrings include the keys that decrypt all and only the resources belonging to their capability lists [36].

3.3.2 Plaintext database model

We model the plaintext database through the following triple:

$$\mathbb{P} := (\mathcal{S}, \succ, \mathcal{R}) \tag{3.15}$$

where $(\mathcal{S}, \succ)$ is the partially ordered set (*poset*) of the database structures, and $\mathcal{R}$ is the set of resources representing the tenant data. Each element $s \in \mathcal{S}$ is a structure of the database (e.g., a table, a column), and the ordering operator $x \succ y$ $(x, y \in \mathcal{S})$ denotes that x is an ancestor of y, and y is a descendant of x. If a third structure $z \in \mathcal{S} : x \succ z \succ y$ does not exist, then we use the notation $x \gtrdot y$, where x is a parent node of y, and y is a child node of x. We remark that a parent (child) is also an ancestor (descendant), while the opposite is not true. All inclusion relations between the database structures are represented as parent-child relations in the poset (e.g., the column c of the table t is represented by $t \gtrdot c$). Each element $r \in \mathcal{R}$ is the set of all information stored in a column of the database. If we model the structure poset as a hierarchical tree, there is a 1:1 correspondence between each resource $r \in \mathcal{R}$ and each leaf of the poset tree. As an example, we refer to Fig. 3.6 that represents the model of a plaintext database schema (s_1)

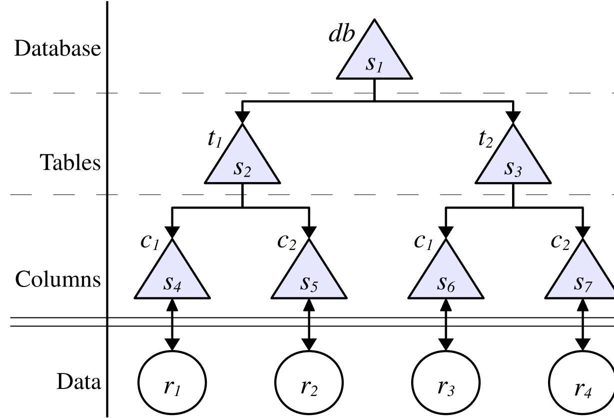


Figure 3.6: The poset representing a plaintext database.

containing two tables (s_2, s_3), each consisting of two columns (s_4, s_5, and s_6, s_7). The columns denote the leafs of the poset tree. The set of data stored in each column is represented as a resource, that is, r_1 represents the actual data stored in the column s_4. The labels associated with the structures are the actual names of the database structures that are concatenated to the absolute path from the root of the structure poset. For example, the label of the structure s_4 is denoted by '$db.t_1.c_1$'.

The proposed plaintext database model is a poset that extends the structure poset $(\mathcal{S}, \succ)$ with the resources R: a structure $s \in \mathcal{S}$ associated with a resource $r \in \mathcal{R}$ is a parent of the resource r $(s \gtrdot r)$; all structures $s^\star \in \mathcal{S}$ that are ancestors of s $(s^\star \succ s)$ are also ancestors of r $(s^\star \succ r)$.

We model the access control rules on the plaintext database through the triple $(\mathcal{U}, \mathcal{S}, \mathcal{A})$, where $\mathcal{U}$ is the set of users, $\mathcal{S}$ is the set of structures, and $\mathcal{A}$ is the access matrix [99]. An authorization rule on a structure also grants an access to all descendant structures and resources. For example, the rule $a_{u_1,s_3} = 1$ authorizes u_1 to access s_3 and all its descendant structures and resources, that is, s_6, s_7, r_3, and r_4.

3.3.3 Encrypted database model

Assuming that a tenant organization owns a plaintext relational database, the first goal is to preserve the confidentiality of the tenant data and even of the database structures because also the table and the column names may leak some information about tenant data. To these purposes, we encrypt tenant data through SQL-aware cryptographic schemes that allow SQL operations on encrypted data: different algorithms support different subsets of SQL operators.

Encrypted data are contained in encrypted tables stored in cloud database servers. For each plaintext table, the MuteDB DBA client generates the corresponding encrypted table and a unique encryption key. The name of the encrypted table is computed by encrypting the name of the plaintext table through that key. The encryption algorithm used for encrypting the table names is a standard AES algorithm in a deterministic mode (e.g., CBC with constant initialization vector). In such a way, only the users that know the plaintext table name and the corresponding encryption key are able to compute the name of the encrypted table. The deterministic scheme is preferred because it allows a 1:1 correspondence between plaintext and encrypted tables and improves the efficiency of the query translation process (see Section 3.3.6).

As a plaintext database column could correspond to multiple encrypted columns, MuteDB does not straightforwardly encrypt its name. Instead, the name of each encrypted column is computed by encrypting the concatenation of the names of the plaintext column and of the encryption algorithm through the deterministic encryption algorithm and by using the encryption key associated with the plaintext column.

We model the encrypted database through the set $\mathbb{E}$, that is an extension of the plaintext database model $\mathbb{P}$ (see Equation (3.15)):

$$\mathbb{E} := (\mathcal{S}, \succ, \mathcal{R}, \mathcal{G}, \mathcal{V}, \Phi, \mathcal{K}, \mathcal{E}, \mathcal{T}, \theta, \Gamma) \tag{3.16}$$

where:

- $(\mathcal{S}, \succ, \mathcal{R})$ is the poset that represents structures and resources belonging to the database, as modeled in the previous section;
- $\mathcal{G}$ is the set of the *access groups*, where each $g \in \mathcal{G}$ is a set of structures $\mathcal{S}_g \subseteq \mathcal{S}$;
- $\mathcal{V}$ is the set of *derivation keys* that are used to compute resource keys; each access group has exactly one derivation key, hence a user u that owns an authorization for the access group g is able to obtain the derivation key $v_g \in \mathcal{V}$ associated with g;
- Φ is the set of the SQL-aware encryption algorithms used to encrypt the resources $\mathcal{R}$;
- $\mathcal{K}$ is the set of resource keys used to encrypt plaintext resources;
- $\mathcal{E}$ is the set of *encryption groups*, where each group $e \in \mathcal{E}$ denotes a set of resources $\mathcal{R}_e \subseteq \mathcal{R}$ that are encrypted through the same encryption key k_e and the same SQL-aware encryption algorithm $\phi \in \Phi$;

- $\mathcal{T}$ is the set of *tokens*; each token $t \in \mathcal{T}$ is a public value that is used to compute derivation and resource keys;
- θ is a *derivation function* that allows the computation of derivation keys; it is defined as:

$$\theta : \mathcal{V} \times \mathcal{G} \times \mathcal{T} \longmapsto \mathcal{V} \tag{3.17}$$

$$\forall (a, b) \in \mathcal{G} \times \mathcal{G} : a \succ b \Rightarrow \exists!\, t : \theta(v_a, b, t) = v_b \tag{3.18}$$

  An implementation example of derivation function is proposed in [9].

- Γ is a function that allows the computation of resource keys for all the resources descending from structures included in an access group, and that is defined as:

$$\Gamma : \mathcal{V} \times \mathcal{G} \times \Phi^n \longmapsto \mathcal{S}^n \times \mathcal{K}^n \tag{3.19}$$

$$\forall (a, \Phi_B), a \in \mathcal{G}, B := \{b \in \mathcal{E} : b \prec a\}$$

$$\Rightarrow \Gamma(v_a, a, \Phi_B) = \{(c, k_b) : b \in B, c \in \mathcal{S}, c \succ b\} \tag{3.20}$$

  An implementation case using symmetric encryption and metadata is proposed in Section 3.3.5.

Let us explain the proposed model by referring to the example of the encrypted database shown in Fig. 3.7, where the encrypted database structures $(s_1, \ldots, s_{10})$ are represented by triangles, the access groups $(g_1, \ldots, g_7)$ by boxes with rounded corners, the encrypted resources $(r_1, \ldots, r_7)$ by circles, and the encryption groups $(e_1, \ldots, e_6)$ by boxes. In this example, there is one database schema (s_1) that contains two tables (s_2, s_3). The table s_2 contains four columns $(s_4, \ldots, s_7)$, and the table s_3 contains three columns $(s_8, \ldots, s_{10})$. Each column is associated with the corresponding set of encrypted resources (e.g., r_1 represents the actual data stored in column s_4). This scheme shows associations between access groups and structures, and between encryption groups and encrypted resources. The access group g_2 includes the structure s_2, and g_5 includes the structures s_5, s_6, s_7. Similarly, the encryption group e_1 contains r_1 and e_4 contains r_4, r_5.

Fig. 3.8 refers to the same encrypted database represented in Fig. 3.7, but it highlights the relations among access and encryption groups. Here, each access group $g_1, \ldots, g_7$ is associated with a derivation key $v_1, \ldots, v_7$. Similarly, each encryption group e is associated with an encryption key k and an encryption algorithm ϕ. As an example, the encryption group e_2 is associated with the algorithm ϕ_1 and the encryption key k_2. The definition of encryption groups is driven by cross-column operations. If multiple encrypted

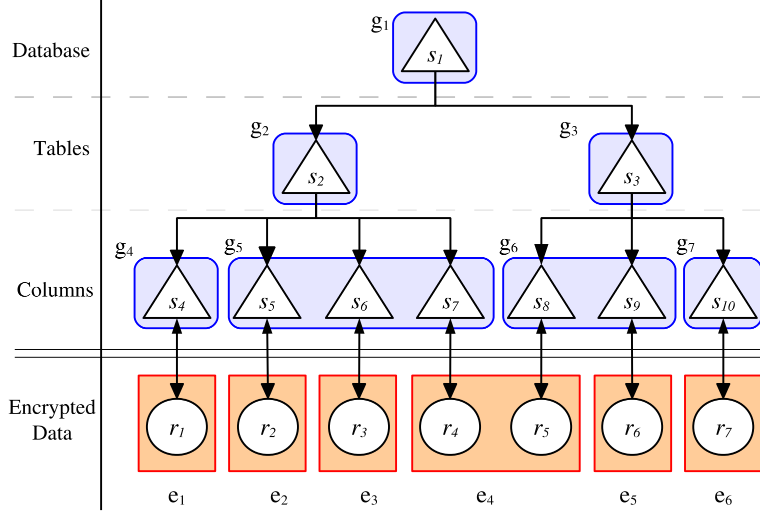


Figure 3.7: Scheme of the structure of an encrypted database.

columns are involved in cross-columns operations (e.g., JOIN), they must belong to the same encryption group because they must share the same resource key. For example, both resources r_4 and r_5 belong to the encryption group e_4, and are encrypted through the algorithm ϕ_3 using the key k_4. Each arrow represents a parent-child relationship between two access groups, or one access group and one encryption group. Each arrow that connects two access groups is associated with a token. As an example, a parent-child relationship $g_1 \succ g_2$ is associated with the token $t_{1,2}$.

3.3.4 Cryptographic enforcement strategy

For the sake of clarity, from now on we refer to the proposed models of plaintext (3.15) and encrypted (3.16) databases by using the following disambiguated notations.

$$\mathbb{P} := (\mathcal{S}_P, \succ, \mathcal{R}_P)$$
$$\mathbb{E} := (\mathcal{S}_E, \succ, \mathcal{R}_E, \mathcal{G}, \mathcal{V}, \Phi, \mathcal{K}, \mathcal{E}, \mathcal{T}, \theta, \Gamma)$$

We define that for each plaintext structure $s_i \in \mathcal{S}_P$, there exists an associated access group $g_i \in \mathcal{G}$ in the encrypted database. In particular, we highlight that the access group g_i is identified by the same name of the plaintext structure s_i. Each encrypted structure $s_e \in \mathcal{S}_E$ has an encrypted name. All and only users authorized to enter the access group g_i know the corresponding derivation key v_i, and are able to know the names of the encrypted structures s_e included in g_i.

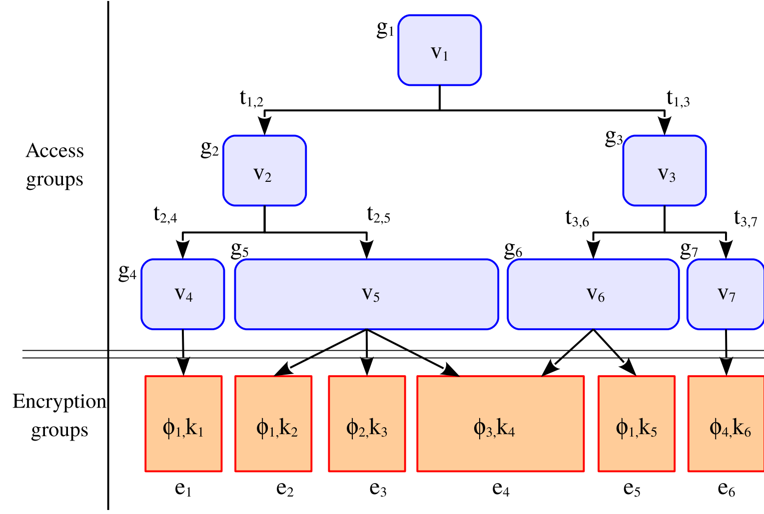


Figure 3.8: Scheme of the access and encryption groups of an encrypted database.

From this definition, it follows that the access control matrix $\mathcal{A}$ defined by the triple $(\mathcal{U}, \mathcal{S}_P, \mathcal{A})$ for the plaintext database $\mathbb{P}$ (Section 3.3.2) can also be applied to the triple $(\mathcal{U}, \mathcal{G}, \mathcal{A}_E)$ defined for the encrypted database $\mathbb{E}$. As a consequence, MuteDB transparently transforms an authorization rule $a_{u,s_i} \in \mathcal{A}$ defined on a plaintext structure s_i into the authorization rule $a_{u,g_i} \in \mathcal{A}_E$ which is defined on the corresponding access group g_i. The authorization rules are automatically enforced in the encrypted database because a user u is authorized to access s_e if and only if he/she is able to calculate the derivation key v_i associated with the corresponding access group g_i.

Let us give an example by referring to Figs. 3.6, 3.7 and 3.8. If a user u is authorized for the database structure s_2 of the plaintext database (as in Fig. 3.6) by the access control matrix $\mathcal{A}$, then he/she is also authorized for the access group g_2 of the encrypted database (see Fig. 3.7) by the access control matrix $\mathcal{A}_E$. Hence, this user is able to access all the descendant access groups by using the public tokens and the derivation key v_2 (see Fig. 3.8). The user is also implicitly authorized to access the encryption groups descending from g_2, and can decrypt all the encrypted resources that are included in these encryption groups. In the considered example, the user u owns an implicit authorization to g_4 and g_5. Hence, he/she is also implicitly authorized to access e_1, e_2, e_3, e_4, and can decrypt the resources r_1, r_2, r_3, r_4, r_5.

Just to give a detailed example, we describe how u is able to decrypt r_3. Since u is authorized for g_2, he/she already knows the derivation key v_2 and also the token $t_{2,5}$ because all the tokens are public, and g_5 be-

cause it is a descendant of g_2. Hence, u can compute v_5 through Equation (3.18): $v_5 = \theta(v_2, g_5, t_{2,5})$. After having computed v_5, u can employ Equation (3.20) to compute the set of keys associated with the encryption groups e_2, e_3, e_4 and the encrypted names of the associated structures s_5, s_6, s_7: $\Gamma(v_5, g_5, \{\phi_{e_2}, \phi_{e_3}, \phi_{e_4}\}) = \{(s_5, k_2), (s_6, k_3), (s_7, k_4)\}$. As the information included in the encrypted resource r_3 belongs to the encryption group e_3, it can be decrypted through the key k_3.

Generation of credentials The DBA can generate and deliver secret keys to enforce the access rules included in the access control matrix $\mathcal{A}_E$ by using the following model $\mathbb{D}$:

$$\mathbb{D} := (\mathcal{U}, \mathcal{G}, \mathcal{A}_E, \mathcal{V}, \mathcal{T}, \theta) \quad (3.21)$$

where $(\mathcal{U}, \mathcal{G}, \mathcal{A}_E)$ represents the access control rules applied to the encrypted database, $\mathcal{V}$ and $\mathcal{T}$ are the sets of derivation keys and tokens as described in Section 3.3.3, θ is the derivation function defined in Equation (3.18).

Each user $u \in \mathcal{U}$ owns a single derivation key $v_u \in \mathcal{V}$, and a set of public tokens $\mathcal{T}_u \subset \mathcal{T}$. The user u is able to calculate the derivation keys $v_g \in \mathcal{V}$ through the function θ if and only if there exists an associated token $t_{v_u,v_g} \in \mathcal{T}_u$. In order to enforce the access rules in the access control matrix $\mathcal{A}_E$, the DBA client randomly generates the derivation key v_u for each user u, where v_u represents the secret key that is included in the credentials of the user. After that, the DBA client scans the access control matrix by rows, thus obtaining the capability list of each user. For each access group g that is included in the capability list cap_u, the DBA client computes a token t_{v_u,v_g}, and inserts it in $\mathcal{T}_u$.

3.3.5 Metadata management

Database metadata include all information allowing a MuteDB client to translate plaintext SQL operations into operations working on the encrypted database.

We describe the original solutions adopted by MuteDB to manage metadata. Existing proposals use trusted infrastructures to store and distribute metadata information [95, 110] or require database users to maintain them locally [37]. These schemes simplify metadata management, but they limit scalability and availability of a cloud database service. The MuteDB alternative is to store metadata in the cloud database together with encrypted tenant data. This approach allows each client to access metadata directly and concurrently through standard SQL operations, thus avoiding system

StructureID	DBToken
$MAC(v_1, \text{'}db\text{'})$	$\{Enc(v_1, \text{'}db.t_1\text{'}), t_{1,2}\}$, $\{Enc(v_1, \text{'}db.t_2\text{'}), t_{1,3}\}$
$MAC(v_2, \text{'}db.t_1\text{'})$	$\{Enc(v_2, \text{'}db.t_1.c_1\text{'}), t_{2,4}\}$, $\{Enc(v_2, \text{'}db.t_1.c_2\text{'}), t_{2,5}\}$
$MAC(v_3, \text{'}db.t_2\text{'})$	$\{Enc(v_3, \text{'}db.t_2.c_1\text{'}), t_{3,6}\}$, $\{Enc(v_3, \text{'}db.t_2.c_2\text{'}), t_{3,7}\}$

Table 3.2: Database tokens table.

ColumnID	Enc
$MAC(v_4, \text{'}db.t_1.c_1\text{'})$	$\{Enc(v_4, \text{'}\phi_1\text{'}), Enc(v_4, k_1)\}$
$MAC(v_5, \text{'}db.t_1.c_2\text{'})$	$\{Enc(v_5, \text{'}\phi_1\text{'}), Enc(v_5, k_2)\}$, $\{Enc(v_5, \text{'}\phi_2\text{'}), Enc(v_5, k_3)\}$, $\{Enc(v_5, \text{'}\phi_3\text{'}), Enc(v_5, k_4)\}$
$MAC(v_6, \text{'}db.t_2.c_1\text{'})$	$\{Enc(v_6, \text{'}\phi_1\text{'}), Enc(v_6, k_5)\}$, $\{Enc(v_6, \text{'}\phi_3\text{'}), Enc(v_6, k_4)\}$
$MAC(v_7, \text{'}db.t_2.c_2\text{'})$	$\{Enc(v_7, \text{'}\phi_4\text{'}), Enc(v_7, k_6)\}$

Table 3.3: Database encryption table.

bottlenecks and single point of failures at the tenant side. Metadata contain sensitive information, hence it is necessary to store them in an encrypted form. MuteDB proposes a new metadata management strategy that enforces access control policies at the encryption level, by generating a different encryption key for each user and by ensuring that each user is able to decrypt all and only encrypted tenant data on which he/she has legitimate access.

The naïve solution of using the same encrypted metadata structure and to enforce access control policies by replicating metadata for each user has several drawbacks: metadata replication causes storage overhead and requires some consistency management scheme. This requires locking and synchronization mechanisms that increase concurrency conflicts and lower database performance as the number of users increases. The proposed metadata management strategy guarantees the following benefits: each user is provided with unique credentials that allow him/her to encrypt and decrypt only information on which he has legitimate access; MuteDB clients can perform all operations supported by the SQL-aware algorithms in the encrypted database concurrently and independently; the DBA is the only subject authorized to modify database metadata in order to enforce changes of the access control matrix such as granting and revoking access authorizations.

Independently of the number of users, MuteDB stores all metadata in three tables. The *database tokens table* contains all information related to

UserID	UToken
$MAC(v_{u1}, \text{‘}u1\text{’})$	$\{Enc(v_{u1}, \text{‘}db.t_1\text{’}), t_{u1,v_2}\}$, $\{Enc(v_{u1}, \text{‘}db.t_2\text{’}), t_{u1,v_3}\}$
$MAC(v_{u2}, \text{‘}u2\text{’})$	$\{Enc(v_{u2}, \text{‘}db.t_2\text{’}), t_{u2,v_3}\}$
$MAC(v_{u3}, \text{‘}u3\text{’})$	$\{Enc(v_{u3}, \text{‘}db\text{’}), t_{u3,v_1}\}$
$MAC(v_{u4}, \text{‘}u4\text{’})$	$\{Enc(v_{u4}, \text{‘}db.t_2.c_2\text{’}), t_{u4,v_7}\}$

Table 3.4: Users tokens table.

the encryption enforcement scheme. The *database encryption table* contains all information related to the algorithms and keys used to encrypt resources. These two tables include all information required by the encrypted database model proposed in Section 3.3.3. The *users tokens table* stores all information related to the users credentials (see Section 3.3.4). Each of these tables has two columns: the first column is used as an index to access the actual metadata that are stored in the second column.

In the database tokens table, each row is associated with a structure, namely s, of the plaintext database. The index column is the result of a deterministic MAC function applied to the name of the structure by using the derivation key associated with s as its encryption key. The metadata column memorizes the set of data associated with all the children of s. Each child is represented by two values: the former is an encrypted version of the child name, obtained by using a symmetric encryption algorithm and the derivation key associated with s; the latter is a public token that links s to the child. Structures described in this table are not leafs (i.e., columns) of the hierarchical representation of the plaintext database. Table 3.2 is an example of database tokens table associated with the encrypted database represented in Fig. 3.7. The *StructureID* is the index column, and *DBToken* is the metadata column. The first row includes information related to the structure s_1 that represents the database schema. The StructureID stores an encrypted version of its name ($MAC(v_1, \text{‘}db\text{’})$), and DBToken contains the information related to the two children tables ‘*db.t*$_1$’ and ‘*db.t*$_2$’. For example, for ‘*db.t*$_2$’ it stores $Enc(v_1, \text{‘}db.t_2\text{’})$ which is the encrypted version of its name, and $t_{1,3}$ which is the public token that allows users that know v_1 and ‘*db.t*$_2$’ to compute the derivation key associated with ‘*db.t*$_2$’ (v_3) by means of Equation (3.18).

The database encryption table represents the relationships between columns in the encrypted and plaintext databases. Each row is associated with a column, namely c, of the plaintext database. The index column of this table has the same structure of the index column of the database tokens table.

The metadata column stores the set of data associated with all the encrypted columns related to c. Each encrypted column is represented by two values: the former is an encrypted version of the name of the SQL-aware encryption algorithm, obtained through the symmetric encryption algorithm algorithm and the derivation key associated with c; the latter value is an encrypted version of the resource key used to cipher data stored in the encrypted column. This key is encrypted by using the derivation key of c. Table 3.3 is an example of database encryption table, where *ColumnID* is the index column, and *Enc* is the metadata column. The second row includes information related to the plaintext column '$db.t_1.c_2$'. The Enc column includes metadata associated with the three encrypted columns s_5, s_6, s_7 within the access group g_5 (Fig. 3.7). As an example, the resource r_4 included in s_7 is encrypted through the algorithm ϕ_3 and the resource key k_4. It is worth to observe that also r_5 is encrypted through the algorithm ϕ_3 and the resource key k_4, because r_4 and r_5 belong to the same encryption group and hence they share the same encryption algorithm and resource key.

The users tokens table contains information that is necessary to each user to derive his/her resource encryption keys. Each row is associated with a user. The index column stores a MAC computed over the user identifier with the user derivation key. The metadata column memorizes a set of data in which each element represents an explicit authorization to access a structure of the plaintext database. Each authorization includes two values: the former is the name of the structure encrypted through a symmetric encryption algorithm and the user derivation key; the latter is the public token that allows the user to compute derivation key associated with the encrypted structure.

Let us consider an example in which four users ($u1, \ldots, u4$) have legitimate access to different structures of the plaintext database of Fig. 3.6. The user $u1$ has an explicit authorization for '$db.t_1$' and '$db.t_2$'; $u2$ for '$db.t_2$'; $u3$ for 'db'; $u4$ for '$db.t_2.c_2$'. We recall from the Section 3.3.2 that users are implicitly authorized to access all the descendant structures and resources. Table 3.4 shows the content of the users tokens table in the corresponding encrypted database.

An important objective of the metadata table design is to avoid disclosure of any association between the encrypted database structures and the metadata, and between the users and the metadata information. To this purpose, MuteDB uses MAC functions and IND-CPA symmetric encryption algorithms with random initialization vectors. As a result, the same metadata or structure identifier is never encrypted to the same ciphertext value, thus making each of them indistinguishable to a cloud insider even if he colludes with a legitimate database user.

3.3.6 Query translation

We describe how a plaintext SQL operation is translated into an encrypted operation by taking as an example that the user $u1$ has to execute the following operation: *SELECT SUM(c_2) FROM* t_1 *WHERE* $c_1 > 10$. We assume that the encryption algorithm ϕ_1 used to encrypt r_1 is order preserving [19], and the algorithm ϕ_2, which is used to encrypt r_2, is homomorphic with respect to sums [88]. We also assume that this is the first execution of the MuteDB client, hence no metadata is cached locally, but the only information available is v_{u1}, that is the u_1 derivation key included in the user credentials. The MuteDB client of $u1$ retrieves the $u1$ tokens from the user tokens table (*ut-table*) by executing the following query: *SELECT UToken FROM ut-table WHERE UserID* = $MAC(v_{u1}$, *'u1'*). This operation returns all the structures for which $u1$ is explicitly authorized and the related tokens. The MuteDB client decrypts the structure names by using its own derivation key v_{u1} and computes the derivation key v_2 by using the public token t_{u1,v_2} because the query requires an access to the table t_1. A second query is executed to the database tokens table (*db-table*): *SELECT DBToken FROM db-table WHERE StructureID=*$MAC(v_2$, *'db.t_1'*). This operation returns encrypted column names and their tokens. By using v_2, the $u1$ client decrypts these names and computes the derivations keys v_4 and v_5 required to operate over encrypted versions of the columns $t_1.c_1$ and $t_1.c_2$. The MuteDB client executes the third query on the database encryption table (*enc-table*): *SELECT Enc FROM enc-table WHERE ColumnID=*$MAC(v_4$, *'db.$t_1.c_1$'*) *OR ColumnID=*$MAC(v_5$, *'db.$t_1.c_2$'*). The results include resource keys and encryption algorithms of all the encrypted columns corresponding to the plaintext columns $t_1.c_1$ and $t_1.c_2$. The MuteDB client decrypts algorithms names and resources keys. Since $t_1.c_2$ has three encrypted representations, the client chooses ϕ_2 as its encryption algorithm and k_3 as its resource key. Now, the client owns all the information required to translate the plaintext query into the encrypted query. First it computes the names of the encrypted table s_2 and of the encrypted columns s_4 and s_6: $s_2 = AES_{det}(v_2, \text{'}db.t_1\text{'})$, $s_4 = AES_{det}(v_4, \text{'}db.t_1.c_1\text{'}|\text{'}\phi_1\text{'})$, and $s_6 = AES_{det}(v_5, \text{'}db.t_1.c_2\text{'}|\text{'}\phi_2\text{'})$, where AES_{det} represents deterministic AES encryption using a constant initialization vector. Moreover, the client encrypts the constant value '10' as $y = \phi_1(k_3, 10)$. The encrypted query is: *SELECT HSUM(s_6) FROM* s_2 *WHERE* $s_4 > y$, where *HSUM* is a remote stored procedure that executes homomorphic sums [88]. Metadata are cached by the MuteDB clients, hence the successive executions of SQL operations using the same metadata do not require a metadata retrieval from the cloud database. In most workloads metadata caching allows the client to directly encrypt queries. In the use case

scenarios that include database structure modifications, MuteDB can leverage standard isolation mechanisms to guarantee consistency of encrypted data and metadata as proposed in Section 3.2.

3.3.7 Credentials management

We describe the operations to provision a new user with access privileges and to revoke him of existing privileges.

User creation and privilege provisioning Whenever a new user is created or when access control policies change by giving more privileges to an existing user, the DBA has to update metadata reflecting the new access control policies. The creation of a new user implies the generation of a new derivation key, and the insertion of a new row in the users tokens table. The index field of the new row is the deterministic MAC computed over the user identifier through the user derivation key. Since the metadata field of the row related to the new user is empty, at this point the user cannot access any structure of the encrypted cloud database. To provision a new privilege to an existing user, the DBA updates the metadata field of the user tokens table row related to that user by inserting all metadata information related to the new authorization. This information includes the encrypted version of the plaintext structure for which the user is authorized, and the new public token that the user needs to compute the structure derivation key. We highlight that MuteDB is able to provision new privileges with no necessity of distributing new credentials to the users. This necessity represents one of the main disadvantages of existing architectures for access control enforcement that store encryption keys and complex metadata structures in client machines (e.g., [36]).

User removal and privilege revocation When a database user is removed or when some of his access privileges are revoked, we have to invalidate all information related to the revoked privileges because the user should not be able to decrypt information for which he/she is no longer authorized. These operations include the renewal of metadata, and the re-encryption of encrypted information through download/upload operations of encrypted tenant data from/to the cloud database. They are among the most expensive processes of any architecture that enforces access control of outsourced data through encryption. Indeed, other countermeasures (e.g., access limitation to the database, updating just tokens or derivation keys) that do not include data re-encryption do not guarantee confidentiality because the

user may have maintained locally a private copy of resource keys and use them to collude with a cloud insider. In addition to resource re-encryption, MuteDB updates metadata by renewing all the encryption keys of the revoked resources, and the tokens and derivation keys that were used to obtain these encryption keys. We describe the metadata update process by considering as an example the revocation of access privileges on table t_2 for user $u1$ (see Figs. 3.7 and 3.8). Renewing resources encryption keys require the DBA client to identify all encryption groups that are descendant of the access group related to t_2. In this example, the access group is g_3 and all descendant encryption groups are e_4, e_5, e_6. The DBA client generates a new random resource key for each encryption group, and generates new random derivation key for g_3 and for the descendant access groups g_6 and g_7. Then, it computes all tokens that point to or that exit from any access groups for which a new derivation key has been generated, that are $t_{1,3}, t_{3,6}, t_{3,7}$, and between users and access groups, that are t_{u2,v_3}, t_{u4,v_7}. These operations are efficiently executed by MuteDB thanks to the fine-grained storage granularity of the access control enforcement scheme and of metadata tables.

3.4 Performance evaluation

In this section we describe performance evaluation based on a software prototype. In Section 3.4.1 we give the details of the software implementation. In Sections 3.4.2 and 3.4.3 we evaluate the performance of the architecture in emulated and real distributed environments by using the TPC-C benchmark and the YCSB stress test.

3.4.1 Prototype implementation

MuteDB does not require modifications of database services, but just of the client software. We implemented a software prototype based on *Python* and *C/C++* that acts as a database connector between the client application and the plaintext database connector. The software receives standard SQL commands, encrypt them depending on the encryption policies, and forward the translated commands to the plaintext database connector. The current version of the software allows users to access standard and cloud database solutions by ensuring and managing confidentiality of all data stored in a untrusted cloud database. We tested it with PostgreSQL, MySql and SQL Server relational databases. We can observe that porting MuteDB to different DBMS required minor changes related to the software components more associated to the plaintext database connector, and minimal modifica-

tions of the codebase. The software supports the main data manipulation (SELECT, INSERT, UPDATE, DELETE) and data definition (CREATE, DELETE) operations of the SQL language with no required modification of the cloud database service. It also supports transactions (BEGIN, COMMIT and ROLLBACK commands) to execute multiple concurrent operations consistently, and it can be ported to any relational DBMS and to any commercial cloud database service. The software consists of five logical components that implement the architecture logic and of *encryption algorithms* that are used transparently to encrypt and decrypt data. In the following we describe the MuteDB software prototype components, the encryption algorithms supported in the last version of the software and the support for existing databases and cloud database services.

Software components. The five software components are: *operation parser*, *encryption engine*, *metadata manager*, *query writer*, *standard database connector*.

- The *operation parser* software modules receives plaintext SQL commands from users and translates them into an intermediate form that is processed by the other software modules. Each plaintext command is analyzed to identify its main components, such as command type, table and column names, operators and constant values.

- The *encryption engine* handles all encryption and decryption operations by applying the encryption policies defined in MuteDB metadata. Although the current version of the software already implements the encryption algorithms to execute most SQL operations (see *Encryption algorithms* below), the encryption engine module can be easily extended with new algorithms.

- The *metadata manager* handles local copies of metadata on the client, and guarantees that the encryption engine always uses consistent and up-to-date metadata information. This component adapts the metadata caching policies with respect to the use context defined in Section 3.2. Moreover, the metadata manager administrates the metadata serialization and de-serialization processes. To achieve interoperability and platform independence the metadata are serialized into JSON data structures that are easily readable. Metadata elements, such as encryption keys and strings, are encoded through open standards.

- The *query writer* is the software component that translates plaintext commands processed by the operation parser into SQL commands over

encrypted data that will be executed by the untrusted cloud database. It leverages the encryption engine to execute all the encryption operations. Moreover, it interacts with the metadata manager to check whether all the operators contained into the plaintext commands are supported by the encryption policies applied to the relevant customer data. Translated SQL commands are forwarded to the *standard database connector*.

- The *standard database connector* represents the interface between the MuteDB client and the remote DBMS. It is implemented through the SQLAlchemy libraries [105] that are compatible with Python DBAPI2 standard [97]. SQLAlchemy handles database connections and SQL dialects, thus allowing MuteDB clients to abstract from the specific DBMS implementation.

Encryption algorithms. The current implementation of the MuteDB prototype includes all the encryption algorithms that are necessary to support each SQL operation of the TPC-C and YCSB workloads on the encrypted database columns. *Equality check* is supported by deterministic symmetric encryption algorithms (DET) that preserve data equality (the user can choose AES [35] or BlowFish [101] in CBC-mode with constant initialization vectors). *Order comparison* operations, that is, $=$, $<$, $>$, $\leqslant$, $\geqslant$, can be executed through Order Preserving Encryption (OPE) [19] that preserves the same order of unencrypted data. *Sum of integers* is made available through the Paillier algorithm [88] that is homomorphic with respect to the sum operator. Other operations, such as string match and multiplication, are feasible through Search algorithms [25, 104] and RSA, respectively. The database columns not requiring any computation can be encrypted through standard ciphers and operation modes such as AES [35] or Blowfish [101] in CBC-mode with random initialization vectors. It is important to observe that the MuteDB architecture is modular so it can integrate other encryption algorithms. Moreover, we do not consider the use of authenticated encryption algorithms because data integrity should be addressed at the architectural level through integration with additional integrity protocols.

Cloud databases support. We tested the prototype implementation on different cloud database providers. Experiments are carried out in Postgres Plus Cloud Database [44], Windows SQL Azure [76], and also on an IaaS provider, such as Amazon EC2 [3], that requires a manual setup of the database. The first group of cloud providers offer ready-to-use solutions to

tenants, but they do not allow a full access to the database system. For example, Xeround provides a standard MySql interface and proprietary APIs that simplify scalability and availability of the cloud database, but do not allow a direct access to the machine. This prevents the installation of additional software, the use of tools, and any customization. On the positive side, MuteDB using just standard SQL commands can encrypt tenant data on any cloud database service. Some advanced computation on encrypted data may require the installation of custom libraries on the cloud infrastructure. This is the case of Postgres Plus Cloud that provides SSH access to enrich the database with additional functions.

3.4.2 Emulated environment

In this section we test performance of the software prototype in a controlled environment with emulated network latencies. We use the Emulab [113] testbed that provides us a controlled environment with several machines, assuring repeatability of the experiments for the variety of scenarios to consider in terms of workload models, number of clients and network latencies.

As the workload model for the database, we refer to the TPC-C benchmark [108]. The DBMS server is PostgreSQL9.1 deployed on a quad-core Xeon having 12GB of RAM. Clients are connected to the server through a LAN where we can introduce arbitrary network latencies to emulate WAN connections that are typical of cloud services. The experiments evaluate the overhead of encryption, compare the response times of plain vs. encrypted database operations, and analyze the impact of network latency. We consider two TPC-C compliant databases with 10 warehouses that contain the same number of tuples: plain tuples consist of 1046MB data, while MuteDB tuples have size equal to 2615MB because of encryption overhead. Both databases use repeatable read (snapshot) isolation level [15].

In the first set of experiments, we evaluate the overhead introduced when one MuteDB client executes SQL operations on the encrypted database. Client and database server are connected through a LAN where no network latency is added.

To evaluate encryption costs, the client measures the execution time of the 44 SQL commands of the TPC-C benchmark. Encryption times are reported in the histogram of the Figure 3.9 that has a logarithmic Y-axis. TPC-C operations are grouped on the basis of the class of transaction: Order Status, Delivery, Stock Level, Payment, New Order. From this figure, we can appreciate that the encryption time is below 0.1ms for the majority of operations, and below 1ms for almost all operations but two. The exceptions are represented by two operations of the *Stock level* and *Payment*

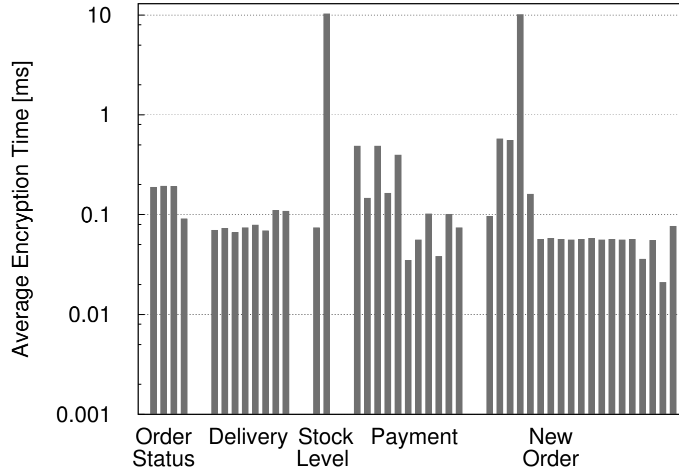


Figure 3.9: Encryption times of TPC-C benchmark operations grouped by transaction class.

transactions where the encryption time is two orders of magnitude higher. This high overhead is caused by the use of the order preserving encryption that is necessary for range queries [19].

To evaluate the performance overhead of encrypted SQL operations, we focus on the most frequently executed SELECT, INSERT, UPDATE and DELETE commands of the TPC-C benchmark. In the Figures 3.10 and 3.11, we compare the response times of SELECT and DELETE, and UPDATE and INSERT operations, respectively. The Y-axis reports the boxplots of the response times expressed in ms (at a different scale), while the X-axis identifies the SQL operations. In SELECT, DELETE, and UPDATE operations, the response times of MuteDB SQL commands is almost doubled, while the INSERT operation is, as expected, more critical from the computational point of view and it achieves a tripled response time with respect to the plain version. This higher overhead is motivated by the fact that an INSERT command has to encrypt all columns of a tuple, while an UPDATE operation encrypts just one or few values.

The second set of the experiments is oriented to evaluate the impact of network latency and concurrency on the use of a cloud database from geographically distant clients. To this purpose, we emulate network latencies through the traffic shaping utilities available in the Linux kernel by introducing synthetic delays from 20ms to 150ms in the client-server connection. These values are representative of round-trip times in continental (in the ranges 40-60ms) and inter-continental (in the ranges 80-150ms) con-

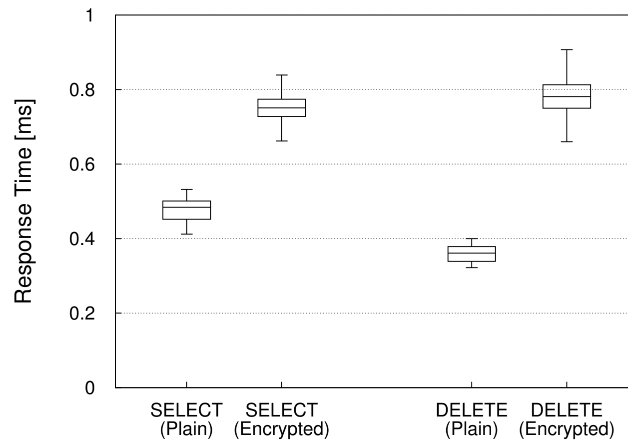


Figure 3.10: Plain vs. encrypted SELECT and DELETE operations.

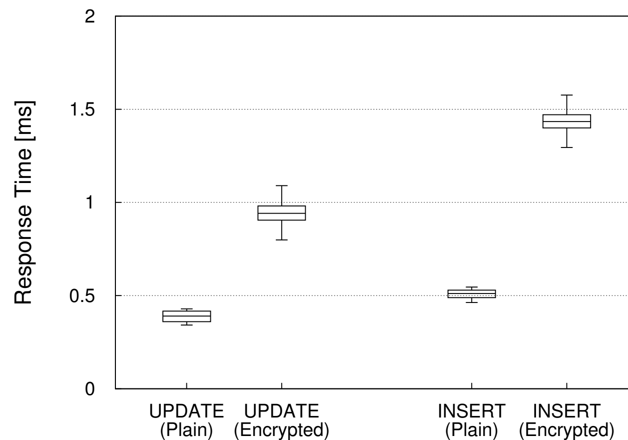


Figure 3.11: Plain vs. encrypted UPDATE and INSERT operations.

nections [111], that are expected when a cloud-based solution is deployed. Table 3.5 reports the response times of the most frequent SQL operations in the plain and encrypted cases for 20ms, 40ms and 80ms latencies. The last column of this table also reports the absolute and percentage overhead introduced by MuteDB. These experimental results demonstrate that the response times of the SQL operations issued to a remote database is dominated by network latencies even in well connected regions. Each response time is two orders of magnitude higher than the corresponding time of a plain SQL operation in a LAN environment. Thanks to this effect, the overhead of MuteDB for the most common SELECT operation falls from 57% to 1.31%

Network delay	SQL command	Plaintext response time	Encrypted response time	Overhead (absolute and percentage)
LAN	SELECT	0.478 ms	0.753 ms	0.275 ms 57%
	DELETE	0.369 ms	0.783 ms	0.414 ms 112%
	UPDATE	0.397 ms	0.951 ms	0.554 ms 140%
	INSERT	0.517 ms	1.442 ms	0.925 ms 179%
20 ms	SELECT	20.67 ms	20.94 ms	0.27 ms 1.31%
	DELETE	20.66 ms	20.97 ms	0.31 ms 1.50%
	UPDATE	20.67 ms	21.12 ms	0.45 ms 2.18%
	INSERT	20.85 ms	21.61 ms	0.76 ms 3.65%
40 ms	SELECT	40.64 ms	40.90 ms	0.26 ms 0.64%
	DELETE	40.65 ms	40.92 ms	0.27 ms 0.66%
	UPDATE	40.62 ms	41.08 ms	0.46 ms 1.13%
	INSERT	40.82 ms	41.56 ms	0.74 ms 1.81%
80 ms	SELECT	80.76 ms	80.97 ms	0.21 ms 0.26%
	DELETE	80.67 ms	81.01 ms	0.34 ms 0.42%
	UPDATE	80.65 ms	81.09 ms	0.44 ms 0.55%
	INSERT	80.86 ms	81.63 ms	0.77 ms 0.95%

Table 3.5: Response times and overheads of SQL operations for different network latencies

and to 0.26% in correspondence of network latencies equal to 20 ms and 80 ms, respectively.

The last set of experiments assess the performance of MuteDB in realistic cloud database scenarios, as well as its ability to support multiple, distributed and independent clients. The testbed is similar to that described previously, but now the runs are repeated by varying the number of concurrent clients (from 1 to 40) and the network latencies (from plain LAN to delays reaching 150 ms). All clients execute concurrently the benchmark for 300 seconds. The results in terms of throughput refer to three types of database operations:

- **Original TPC-C**: the standard TPC-C benchmark;
- **Plain-MuteDB**: MuteDB that use plain encryption, that is, all MuteDB functions and data structures with no encryption; it allows us to evaluate the overhead of MuteDB without the cost of cryptographic operations;
- **MuteDB**: MuteDB referring to the highest confidentiality level.

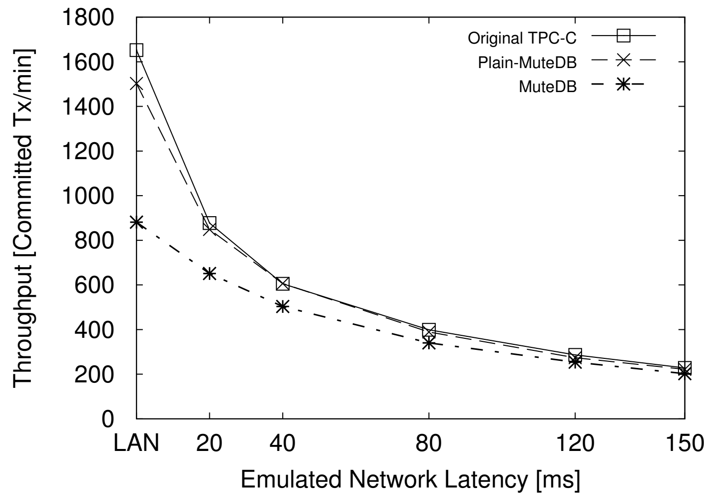


Figure 3.12: TPC-C performance (20 concurrent clients)

Figure 3.12 shows the system throughput referring to 20 clients issuing requests to MuteDB as a function of the network latency. The Y-axis reports the number of committed transactions per minute during the entire experiment. This figure shows two important results:

- if we exclude the cryptographic costs, MuteDB does not introduce significant overheads. This can be appreciated by verifying that the throughput of Plain-MuteDB and Original TPC-C overlies for any realistic Internet delay (>20ms);
- as expected, the number of transactions per minute executed by MuteDB are lower than those referring to Original TPC-C and Plain-MuteDB, but the difference rapidly decreases as the network latency increases to the extent that is almost nullified in any network scenario that can be realistically referred to a cloud database context.

Figures 3.13 and 3.14 show the throughput for increasing numbers of concurrent clients in contexts characterized by 40ms and 80ms network latencies, respectively. These measures are optimistic representations of continental and intercontinental delays. The Y-axis represents the number of committed TPC-C transactions per minute executed by the clients. The trends of the MuteDB lines are close to those of the Original TPC-C database, thus demonstrating that MuteDB encrypted database does not affect scalability with respect to the plain database. Even more important, the network latencies tend to mask cryptographic overheads for any number of clients. For example, the overheads of MuteDB with 40 concurrent clients decreases from

20% in a 40ms scenario to 13% in a realistic scenario where the client-server latency is equal to 80ms. This result is important because it confirms that MuteDB is a valid and practical solution for guaranteeing data confidentiality in real cloud database services.

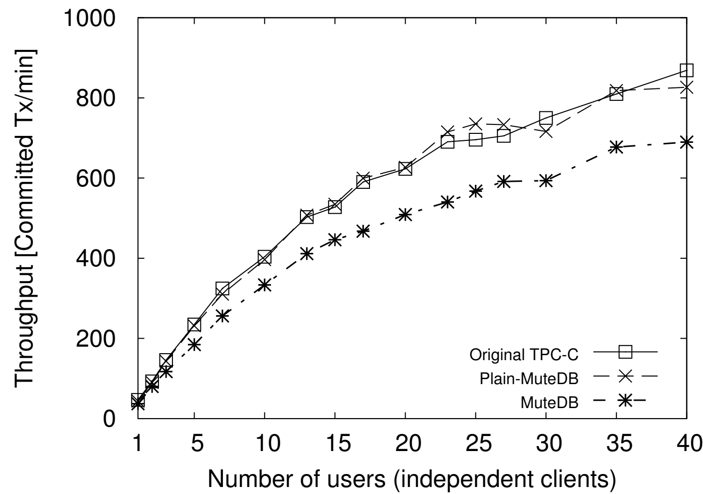


Figure 3.13: TPC-C performance (latency equal to 40 ms)

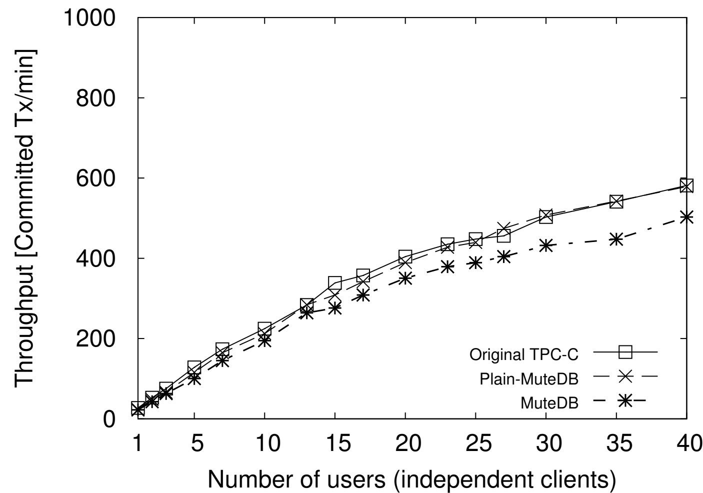


Figure 3.14: TPC-C performance (latency equal to 80 ms)

3.4.3 Real-world environment

In this section we evaluate the performance and scalability of the proposed architecture by using workloads based on the standard database benchmark TPC-C and on the cloud database stress test YCSB [30] executed by concurrent clients that are geographically distributed over ten different countries of the Planetlab platform [29].

The experimental testbed is composed by a PostgreSQL database server located in Europe and by up to 80 clients geographically distributed over ten countries of Planetlab Europe [29]. We highlight that this setting not considering clients located in other continents represents a worst case for the performance of the MuteDB architecture: we have experimentally verified that network latencies higher than 100ms introduce a unrealistic positive bias favoring our solution because they mask the overheads introduced by the encryption, access control and concurrency management of MuteDB.

As we present the first thorough experimental evaluation of an encrypted cloud database service subject to real Internet dispersed clients, we had to carry out some preliminary experiments that aimed to evaluate the characteristics of the Planetlab clients having different network latencies and computational capabilities. For each client, we evaluated its average Round Trip Time with respect to the cloud database server (*RTT* in ms), and the average time required for an OPE encryption (*ENC time* in ms) that is the most computationally expensive algorithm in our prototype. The ENC times of the 80 Planetlab clients with respect to their RTT are represented in Fig. 3.15. The RTTs of most clients concentrate between 30÷40ms (Central Europe) and 50÷60ms (West and North Europe), with some clients between 15÷20ms and around 70ms. The majority of clients have similar computational capabilities as demonstrated by the concentration of the ENC times in a range between 8ms and 13ms with the exception of a few outliers.

The first set of experiments aims to compare the performance of MuteDB and a plaintext database that receive realistic SQL operations. To this purpose, we use a workload based on the standard TPC-C benchmark and two TPC-C compliant database configurations with 100 warehouses that we denote as:

- *TPC-C Standard* (TPCC-STD), in which the TPC-C workload is executed over a plaintext database not using MuteDB;
- *TPC-C MuteDB* (TPCC-MuteDB), in which the TPC-C workload is executed on a database encrypted through MuteDB. All columns are encrypted with the most secure encryption algorithm supporting the SQL operations of the TPC-C workload.

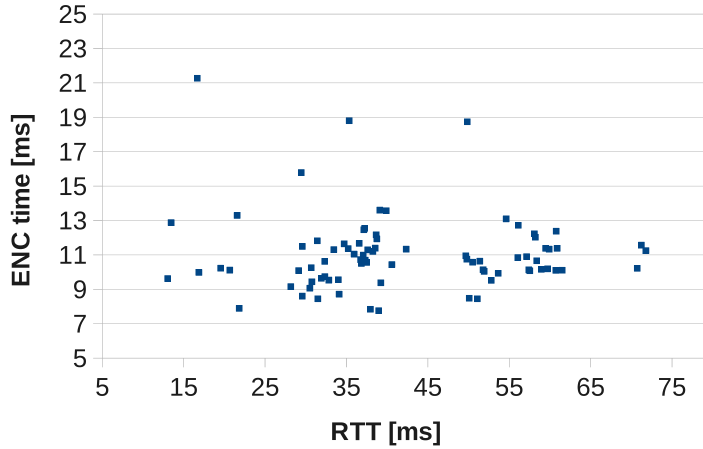


Figure 3.15: Distribution of the RTTs and ENC times for the 80 Planetlab clients.

Type	Name	Query Ratios
A	*Update Heavy*	50% READ, 50% UPDATE
B	*Read Mostly*	95% READ, 5% UPDATE
C	*Read Only*	100% READ
D	*Read Latest*	95% READ, 5% INSERT

Table 3.6: YCSB workloads.

We also perform several experiments based on YCSB [30], that is a stress test for cloud database services recently proposed by Yahoo. YCSB emulates various workloads by executing different mixes of SQL operations (Table 3.6). They are complementary to the TPC-C evaluations because they allow us to estimate the impact of different encryption algorithms on the performance perceived by the clients.

In the reported experiments, we consider YCSB-compliant databases each consisting of one table composed by 11 columns: one primary key and 10 data columns. The table contains one million tuples, each having a size of about 1 KB. We design the following three configurations:

- *YCSB Standard* (YCSB-STD), where the columns of the YCSB table are not encrypted.

- *YCSB MuteDB - Best Case* (MuteDB-Best), where the primary key of the YCSB table is encrypted with DET that is the fastest encryption algorithm supported by MuteDB.

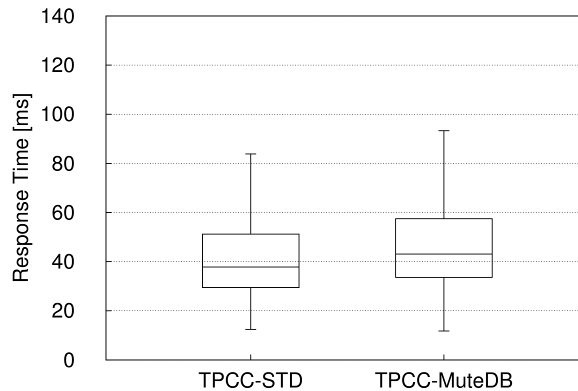


Figure 3.16: Response times for the SQL operations in the TPC-C configurations.

- *YCSB MuteDB - Worst Case* (MuteDB-Worst), where the primary key of the YCSB table is encrypted with OPE that is the most computationally expensive encryption algorithm supported by MuteDB.

The data columns on which no computation is required are encrypted through AES with a random initialization vector. We observe that each query of any YCSB workload requires the execution of at least one operation on the primary key column. For the encrypted configurations, it means that each query requires at least one encryption using the algorithm associated with the primary key. Hence, the overhead introduced by MuteDB for a realistic workload will fall between the overheads of MuteDB-Best and MuteDB-Worst scenarios.

In the first set of experiments, we execute several TPC-C tests with the 80 concurrent distributed clients for all database configurations. Each test lasts twelve minutes, of which we report the stable state results of ten minutes in the middle. We monitor the TPC-C SQL operations response times in order to evaluate the performance overhead of MuteDB with respect to the network latencies that are intrinsic to any cloud environment.

Fig. 3.16 reports the response times of the 80 clients of the testbed with respect to all the SQL operations of the TPC-C scenarios. The two boxplots represent the distribution of the response times (Y-axis) experienced by clients in the TPCC-STD (left boxplot) and TPCC-MuteDB (right boxplot) configurations. This figure shows that clients experience similar performance in the two configurations: the median response time for the plaintext database is slightly lower than 40ms, and the overhead added by MuteDB is less than 6ms. The distribution of the response times is similar as well:

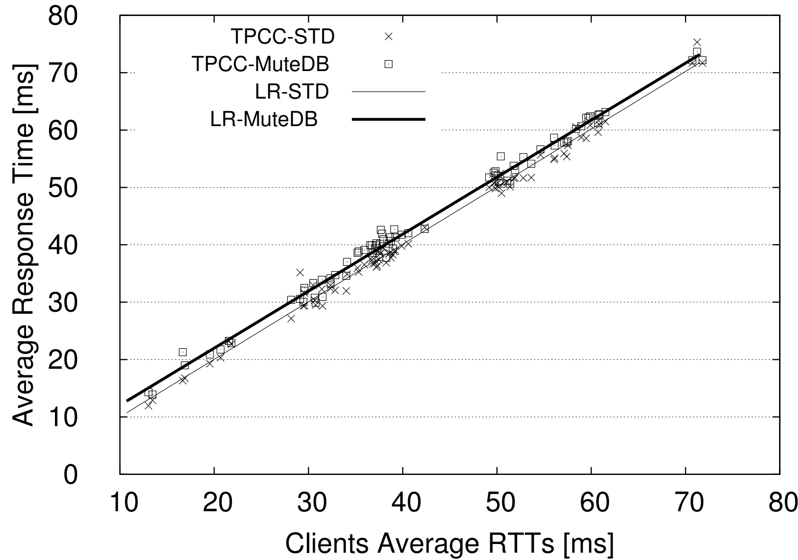


Figure 3.17: Average response time of the most frequent TPC-C SELECT operation for different clients.

the interquartile range differs of about 3ms and the whiskers distance of about 10ms. These experiments carried out for a realistic OLTP workload and geographically distributed clients characterized by different computational capabilities and round trip times show that the overhead expected by a cloud tenant using MuteDB is limited and compatible with real use cases.

We then investigate the details of the presented cumulative results. For space reasons, we report how the network RTT influences the response times by focusing on the most frequent SELECT, UPDATE, INSERT and DELETE SQL operations included in the TPC-C workload. The scatterplot in Fig. 3.17 represents the average response time of the most frequent SELECT operation executed by all clients in both TPC-C configurations with respect to their average RTTs. The X-axis represents the clients average RTTs while the Y-axis is the average response time. To facilitate the interpretation of the results, we draw two linear regression lines denoted by LR-STD and LR-MuteDB, for the TPCC-STD and the TPCC-MuteDB configurations respectively. The low performance overhead introduced by MuteDB is highlighted by the overlap between the clouds of points related to the TPCC-STD and TPCC-MuteDB configurations. The linear regressions show that the overhead introduced by MuteDB is approximately constant and independent of the RTT. Indeed, while MuteDB overhead may be not negligible for clients with very low RTT (e.g., from 13ms to 15ms for a client having an

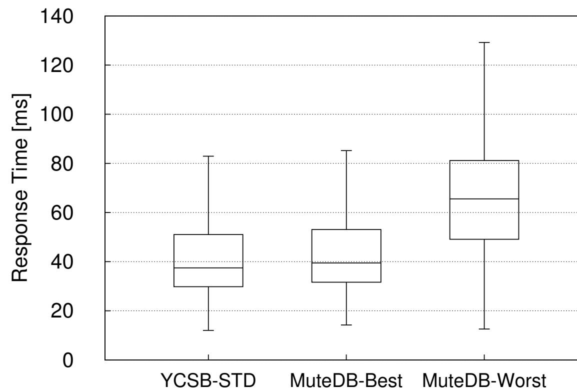


Figure 3.18: Response times for the SQL operations in the YCSB configurations.

average RTT of 13ms), it becomes less significant for clients characterized by higher RTTs (e.g., from 61ms to 64ms for a client having an average RTT of 60ms). Analogous charts related to the most frequent INSERT, UPDATE and DELETE operations of the TPC-C workload confirmed the same results.

We now investigate the effects of different encryption configurations on performance through several experiments based on the YCSB stress test. In particular, we consider 80 concurrent clients executing the YCSB workloads A, B, C and D (Table 3.6), and we analyze the distribution of the response times considering all the SQL operations composing YCSB. Fig. 3.18 compares the response times of the clients in the YCSB-STD (leftmost boxplot), MuteDB-Best (central boxplot) and MuteDB-Worst (rightmost boxplot) configurations. We observe that the performance of YCSB-STD and MuteDB-Best are almost equal. On the other hand, in the MuteDB-Worst scenario, the response times are approximately 25÷30ms higher, and the interquartile range and the whiskers distance increase. The higher variability is caused by the different computational capabilities of the Planetlab clients that compose our testbed. A breakdown of these results is presented by the scatterplot in Fig. 3.19 where the average response time of each client is plotted as a function of its average RTT. Similarly to Fig. 3.17 we draw three linear regressions (LR-STD, LR-Best and LR-Worst) to highlight the trends of the three clouds of points that correspond to the YCSB-STD, MuteDB-Best and MuteDB-Worst configurations. The linear regressions related to the YCSB-STD and MuteDB-Best configurations are similar and the scatterplots denote narrow clouds of points. As expected, the linear regression of the MuteDB-Worst response time is higher and its scatterplot is characterized by a high disper-

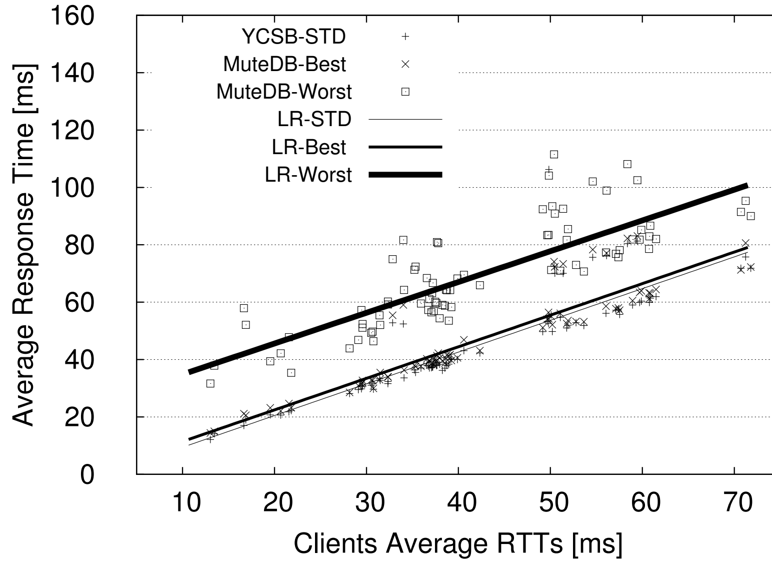


Figure 3.19: Average response time of YCSB SELECT operation for different clients.

sion of the results. The interesting result is that the overall overhead of the worst case scenario remains stable for any RTT between the clients and the cloud service.

Scalability evaluation

In the following set of experiments we evaluate the scalability of the proposed architecture subject to different workloads with respect to increasing number of concurrent clients. Since we are working on a real platform consisting of clients that differ in terms of RTT and computational capability, for the sake of fairness it is important to add at each new iteration of the scalability tests a set of clients that are relatively uniform to the previous set. To this purpose, we divide the 80 Planetlab clients in ten groups where each group consists of eight clients with similar RTT. The tests are repeated for increasing number of geographically distributed and concurrent clients, by adding one client from each group at every iteration. The first iteration of each test has 10 clients, the second iteration 20 clients, and so on. Each iteration lasts twelve minutes of which we report the stable state results of ten minutes in the middle. The results of the most significant scalability experiments are reported in Figs. 3.20.

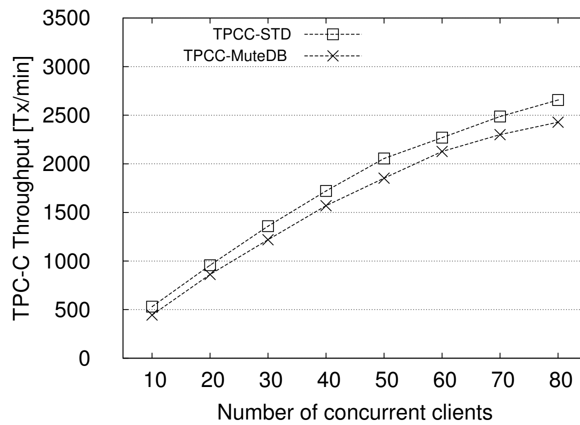
The *TPC-C throughput* denotes the number of TPC-C transactions committed per minute on the database server. In Fig. 3.20a, we report on the

Y-axis the TPC-C throughput of the TPCC-STD and TPCC-MuteDB configurations for increasing number of concurrent clients represented on the X-axis. We are mainly interested in evaluating the scalability of the proposed architecture and the impact of cryptography. Although the absolute values of the TPC-C throughputs are less important for the scope of performance evaluation, we observe that the proposed results are affected by network latencies and hence they cannot be compared to those of typical TPC-C evaluations obtained in local deployments. From Fig. 3.20a we can appreciate that both the TPCC-STD and TPCC-MuteDB throughputs scale linearly for up to 40 clients and slightly sub-linearly for higher numbers of clients. Even more importantly, the *throughput slowdown*, which is defined as the difference between the plaintext and the encrypted configuration throughputs, remains rather constant for any number of clients. This is an important result because it shows that the scalability of the cloud database service is not affected by the solutions adopted by MuteDB.

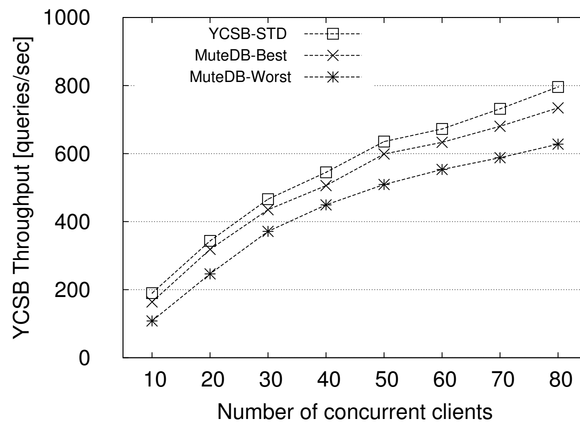
Similar conclusions can be drawn by analyzing the results obtained by using the Update Heavy workload (A) of YCSB reported in Fig. 3.20b. The X-axis represents the number of concurrent clients, and the Y-axis reports the *YCSB throughput* as the total number of SQL operations executed per second on the database server. The three lines represent the YCSB throughput of the YCSB-STD, MuteDB-Best and MuteDB-Worst configurations, respectively. In all the three scenarios, the scalability is linear up to 30 clients, and then sub-linear.

Different results are obtained for the Read-Only (C) YCSB workload. Fig. 3.20c shows that the system scales linearly for up to 80 clients in all the three database configurations. We observe that a read-only workload is rather unrealistic but it is interesting as a term of comparison. In such scenario, where the throughputs keep scaling linearly because there are no database consistency issues due to additional concurrent clients, the throughput slowdown of the MuteDB-Worst configuration tends to be more evident. In any case, we remark that this represents a worst case scenario and in realistic workloads the throughput would fall between those of MuteDB-Best and MuteDB-Worst.

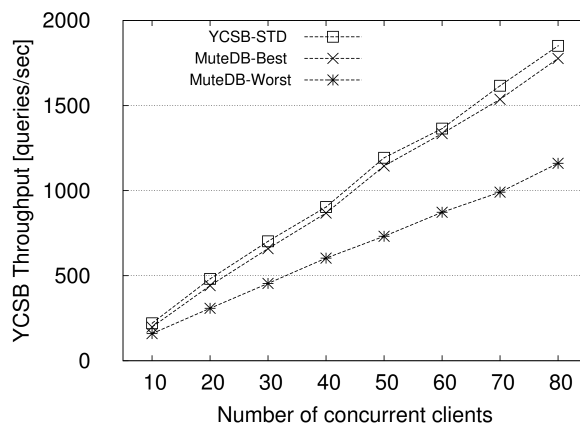
All results confirm that the solutions adopted by the MuteDB architecture are efficient and do not affect the scalability of the native cloud database services.



(a) TPC-C Workload



(b) YCSB Workload A (Update Heavy)



(c) YCSB Workload C (Read Only)

Figure 3.20: Throughput for increasing number of concurrent clients for different workloads and database configurations.

Chapter 4

Efficient authenticity guarantees in dynamic databases

We propose a novel protocol that allows a tenant to efficiently guarantee authenticity of the data stored in a cloud database. By relying on a variant of Bloom filters [18] for the detection of unauthorized modifications, the protocol protects authenticity of the data and reduces storage and network overhead compared to state-of-the-art proposals. 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. Resource overhead in terms of storage and network depends on Bloom filter sizes that should be chosen depending on the security level required by the cloud tenant. To minimize network and storage overhead we propose an analytical model that allows to determine the optimal size of the Bloom filters and to minimize the cloud service costs for the given security level, database characteristics and operations workload. Moreover, we demonstrate the benefits of the proposed protocol through micro-benchmarks and standard mixed-operations workloads. We claim that the protocol fits well cloud databases as it is robust against non-adaptive attacks (see Chapter 1), can be adopted both in proxy-based and in distributed architectures, and can be used to integrate the proposed encryption architecture (see Chapter 3) with integrity guarantees.

The chapter is organized as following. Section 4.1 describes theoretical background on Bloom filters used to design the protocol. Sections 4.2 and 4.3 describe the novel integrity protocol and its security guarantees. Section 4.4 describes how to size the protocol parameters in terms of the security re-

quirements. Section 4.5 proposes an analytical methodology to minimize the resource overhead depending on the workload of the operations executed on the database. Section 4.6 compares performance of the protocol to standard message authentication code protocols through micro-benchmarks and mixed operations workloads.

4.1 Theoretical background

Bloom filters [18,22,78] are space-efficient data structures used for representing sets. They support 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. A BF can be described as an array of m bits (or for short, 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. 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 happens if such bits are equal to one even if the element had never been 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:

$$f(m,n,k) = \left[1 - \left(1 - \frac{1}{m}\right)^{kn}\right]^k \approx \left(1 - e^{-kn/m}\right)^k \tag{4.1}$$

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 [22]:

$$\bar{k} = \frac{m}{n} \cdot \ln(2) \tag{4.2}$$

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

$$\bar{f}(m,n) \approx \left(1 - e^{-\bar{k}n/m}\right)^{\bar{k}} = 2^{-\bar{k}} = e^{-\frac{m}{n}\ln(2)^2} \tag{4.3}$$

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 [77,78].

4.2 Protocol design

In this section we describe the novel solution that uses encrypted BFs to guarantee integrity of data stored in cloud database services 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 propose adding 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 4.1 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. In the following we describe how an authorized client executes *insert*, *read* and *update* operations.

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

Table 4.1: Database table enriched with a column of cryptographic digests.

4.2.1 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:

$$e_r = \mathcal{E}_{sk}(iv, b_r), \tag{4.4}$$

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:

$$b_r = \bigvee_{c=1}^{C} \mathcal{B}_{\tau}^{m}(a_c \parallel v_{r,c}), \tag{4.5}$$

where a_c is the label of the column associated to the value $v_{r,c}$, $\|$ is the concatenation operator, $\bigvee$ is the bitwise *OR* operator and $\mathcal{B}_\tau^m(\cdot)$ is a BF function that outputs a bit string of size m. The function $\mathcal{B}_\tau^m(\cdot)$ is computed by using keyed hash functions (e.g., HMAC algorithms [12]) and the secret key τ. For the sake of clarity 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 (4.5), that it stores the set of elements $\{a_c \parallel v_{r,c}\}_{c\in[1,\ldots,C]}$.

The secret Bloom filter function $\mathcal{B}_\tau^m(\cdot)$ is implemented as follows:

$$\mathcal{B}_\tau^m(x) = \bigvee_{i\in[k]} \left[1 \ll H_{\tau_i}^m(x)\right], \tag{4.6}$$

where $\mathcal{H}_{\tau_i}^m(\cdot)$ denotes a keyed hash function that maps arbitrary length inputs to the range $[0, \ldots, m-1]$, $\ll$ denotes the *bitwise left shift* operator 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 and depends on the secret key τ_i and on the input data x. We note that τ_i denotes a portion of the secret key τ distributed to the authorized clients. All keys τ_i are independent from each other.

From a security perspective, function $\mathcal{H}_{\tau_i}^m(\cdot)$ should be implemented such that its output is pseudo-random, i.e. its distribution is uniform and independent of the distributions of the input value x and of the secret key τ_i. Candidate implementations are keyed pseudo-random functions that accept variable length inputs [13].

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 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 $\mathcal{E}_{sk}(\cdot)$.

As a final comment, we observe that 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. This convention is known to all the parties and it is also used in the verification phase.

4.2.2 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 executes a membership test of the value $v_{r,c}$ on b_r using the secret key τ:

$$\mathcal{D}_{sk}(e_r) \wedge \mathcal{B}^m_\tau(a_c \parallel v_{r,c}) \stackrel{?}{=} \mathcal{B}^m_\tau(a_c \parallel v_{r,c}) \tag{4.7}$$

If the membership test fails, then a violation of integrity has been detected. On the other hand, if it succeeds, then one of the following facts 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 can be limited by a careful choice of BF parameters. A thorough analysis of the attacks against BFs and of how they can be prevented is proposed in Section 4.3.

We recall that 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 (4.5) and (4.4). 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. We highlight that 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 $c_2 \parallel v_{r1,c2}$ and $c_1 \parallel x$ 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.

4.2.3 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 strategies 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 (4.4), (4.5), and (4.6). The main drawback of this simple update strategy is that the tenant retrieves unnecessary values every time that even a single value is updated, thus incurring in a higher network overhead.

The *greedy renew* strategy implies 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 it verifies their integrity. 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 it by using the secret key sk and a new random initialization vector iv'. The update of e_r can be summarized as following:

$$e'_r = \mathcal{E}_{sk}\left(iv', \mathcal{B}^m_\tau(V'_r) \vee \mathcal{D}_{sk}(e_r)\right), \tag{4.8}$$

where $\vee$ denotes the bitwise OR operator.

In the following we propose an informal description of the aspects that must be taken into account when sizing the protocol parameters. A precise estimation is given in Section 4.4.

We observe that 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. First, the digest

must be able to store more values than the cardinality of columns without affecting the security requirement. Second, *renew update operations* must be executed periodically before affecting the security guarantees. In particular, since bigger BFs have lower false probability 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. We note that increasing the size of the BF introduces a trade-off between the storage and network overhead. We propose an analytical methodology to compute an optimal estimation of the protocol parameters in Section 4.5.

4.3 Security analysis

In this section we analyze the security guarantees of the proposed protocol: we identify and evaluate the potential attacks on the scheme and the sizing requirements to defend against them. We remind that all security analyses are discussed under the threat model described in Section 1.3.2. Given the objective of the attacker of modifying the tenant database without being detected, we distinguish two family 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 takes advantage of the BFs false positives by guessing an element whose membership query is answered positively even if the element was never inserted by the tenant.
- **Attacks on the digests**. The attacker generates new digests for existing or new tuples without having any knowledge about the secret key. In this case the attacker tries 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 algorithm [81] that are secure in the chosen-ciphertext attacks models, instead of the IND-CPA encryption algorithm as described in Section 4.2.1. However, both design strategies increase storage and bandwidth overhead.

We highlight that security threats faced when protecting integrity of data outsourced to a remote database are different than those faced in other scenarios. In other contexts, such as protocols for secure communications, attackers can attempt a high number of unsuccessful attacks. Any scheme

that aims at providing security guarantees in similar scenarios have to withstand *adaptive attacks*, since attackers can adopt a trial-and-error strategy and adapt their attacks. In a cloud database scenario, a single unsuccessful attack is enough for the tenant to detect an integrity violation, and to detect attacker activities. Hence, integrity solutions for this scenario have to withstand *non-adaptive attacks*. In this thesis we adopt the non-adaptive chosen plaintext unforgeability attack model proposed in [67]. Informally, this attack model assumes that the attack has success he the adversary is able to generate a ciphertext for a plaintext of choice. It is very important to note that the proposed approach should never be used when an adversary can execute adaptive attacks (see Section 1.3.2), because an adversary could exploit the malleability of the IND-CPA encryption to efficiently forge fake digests.

4.3.1 Attacks on the digest

In this attack scenario an adversary tries to attack the integrity of some data by forging its cryptographic digest. This attack is formally modeled in the symmetric encryption literature as *chosen plaintext forgery* [67]. 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:

- the 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 the values of the tuple;
- the modification of an existing digest: an adversary tries to update an existing tuple with a forged value and to modify the existing digest by adding the new value to it.

Creation of a new digest

In the first 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 [78]. 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.

However, for this attack to succeed the adversary has to generate 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 second 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 goal 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 with 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 modification cannot be detected.

Our analysis focuses on the worst case scenario, and assumes the security guarantees of the most malleable *IND-CPA* encryption algorithm, based on stream ciphers. (We advise against the use of stream ciphers in the proposed protocol due to well known implementation issues that could make them vulnerable to other attacks.) We also assume that the attacker knows all the algorithms and the parameters used in the protocol except for 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 the following 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.

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 underlying secret Bloom filter. In this section we analyze how the false positive rate of the BF varies after an attacker modifies one or more bits of the digest. In particular, we are interested in computing which is the number of bits that the adversary should modify to gain the best advantage.

We define s as the amount of bits that the adversary modifies. The value of s is within the range $0, \dots, m-1$, where for 0 the adversary does not modify the digest, thus falling back to a plaintext attack, and for $m-1$

he 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.) We recall that, as described in Section 4.1, 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:

$$\Pr[\mathit{fp}\,|\#\ \mathit{mod\ bits} = s] = \\ = \sum_{i=1}^{m-s} \left(\Pr[\mathit{fp} \mid s\ \mathit{succ\ mod}] \cdot \Pr[s\ \mathit{succ\ mod} \mid \#\mathit{1bits} = i] \cdot \Pr[\#\mathit{1bits} = i] \right) \tag{4.9}$$

where:

- $\Pr[\mathit{fp} \mid s\ \mathit{succ\ mod}]$ denotes the false positive rate of the BF after s successful modifications (i.e. 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:

$$\Pr[\mathit{fp} \mid s\ \mathit{succ\ mod}] = \Pr[\mathit{fp} \mid \#\mathit{1bits}{=}(i{+}s)] = \left(\frac{i+s}{m}\right)^{\bar{k}} \tag{4.10}$$

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

- $\Pr[s\ \mathit{succ\ mod} \mid \#\mathit{1bits} = i]$ 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:

$$\Pr[s\ \mathit{succ\ mod} \mid \#\mathit{1bits} = i] = \prod_{j=0}^{s-1} \frac{(m-i-j)}{m-j} = \frac{(m-i)!\,(m-s)!}{(m-i-s)!\,m!} \tag{4.11}$$

- $\Pr[\#\mathit{1bits} = i]$ denotes the probability of having exactly i bits set to 1 in the Bloom filter bit string. As described above, we use the optimal number of hash functions $\bar{k}$ to build the secret Bloom filters. Thus, the probability distribution of zeros and ones in the unmodified BF bit string is uniform (see Section 4.1). Hence, this probability can be computed as:

$$\Pr[\#\mathit{1bits} = i] = \frac{\binom{m}{i}}{2^m} = \frac{m!}{2^m i!(m-i)!} \tag{4.12}$$

By substituting Equations (4.10), (4.11) and (4.12), in (4.9) we obtain the following formula:

$$\begin{aligned}
&\Pr[fp \mid \#\ \mathit{mod\ bits} = s] = \\
&= \sum_{i=1}^{m-s} \left(\frac{i+s}{m}\right)^{\bar{k}} \frac{(m-i)!\,(m-s)!}{(m-i-s)!\,m!} \frac{m!}{2^m i!\,(m-i)!} = \\
&= \frac{1}{2^m m^{\bar{k}}} \sum_{i=1}^{m-s} (i+s)^{\bar{k}} \frac{(m-s)!}{(m-s-i)!\,i!} = \\
&= \frac{1}{2^m m^{\bar{k}}} \sum_{i=1}^{m-s} (i+s)^{\bar{k}} \binom{m-s}{i} = \\
&= \frac{1}{2^m m^{\bar{k}}} \sum_{j=1+s}^{m} j^{\bar{k}} \binom{m-s}{j-s}
\end{aligned} \tag{4.13}$$

We observe that this function is monotonic decreasing, as it can be verified as following:

$$\begin{aligned}
&\Pr[fp \mid \#\ \mathit{mod\ bits} = s+1] - \Pr[fp \mid \#\ \mathit{mod\ bits} = s\ ] = \\
&= \frac{1}{2^m m^{\bar{k}}} \left[\sum_{i=2+s}^{m} i^{\bar{k}} \binom{m-s-1}{i-s-1} - \sum_{i=1+s}^{m} i^{\bar{k}} \binom{m-s}{i-s} \right] = \\
&= -\frac{1}{2^m m^{\bar{k}}} \left[(1+s)^{\bar{k}} + \sum_{i=1+s}^{m} i^{\bar{k}} \binom{m-s-1}{i-s} \right]
\end{aligned} \tag{4.14}$$

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

$$\Pr[fp \mid s=0] - \Pr[fp \mid s=1] = \frac{1}{2^m m^{\bar{k}}} \left[1 + \sum_{i=1}^{m} i^{\bar{k}} \binom{m-i}{i} \right] \tag{4.15}$$

We can conclude that an adversary cannot gain any advantage by tampering with the encrypted Bloom filter. Actually a rational adversary will never try to tamper with the encrypted Bloom filter since any modification decreases the attack success rate.

4.4 Sizing boundaries

In this section we show how to size the protocol parameters to guarantee the security levels required by the cloud tenant. As we showed in the previous Section 4.3, the adversary's best attack is to try taking advantage of the Bloom filter false positive rate by inserting a fake value in a database table, without modifying the associated digest. In this section we show how to size the parameters of the protocol, including the size of the digests, to protect data integrity with respect to the tenant required security level. We build our methodology on established Bloom filters false positive analytical estimations described in Section 4.1.

We define the *acceptable false positive rate* ε as the highest false positive probability that a cloud tenant is willing 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 lower BF size that still satisfies the constraints on the false positive rate.

Starting from Equation (4.3) we develop a model that can be used to compute the lower bound on the BF size that satisfies 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).

In the CRD scenario there are no updates. This means that only whole rows can be inserted or deleted, thus 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 (4.3) with $\bar{f} = \varepsilon$ and $n = c$, the tenant can compute $m_{\min}$ as follows:

$$m_{\min} = \left\lceil -\frac{c \cdot \ln(\varepsilon)}{\ln(2)^2} \right\rceil \tag{4.16}$$

Then, the cloud tenant can compute $\bar{k}$ by substituting $m_{\min}$ and c for m and n in Equation (4.2).

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 4.2.3, 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 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). Therefore, the value of n does not change over the lifetime of a BF, hence the false positive rate f remains constant and the computation of $m_{\min}$ falls back to Equation (4.16).

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 without making the false positive rate f exceed 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 that ε. Before an update causes f to exceed ε, the tenant renews the BF, thus restoring f to its original value. We define u as the maximum number of values that the tenant can update while keeping $f \leqslant \varepsilon$. We now 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 the Equation (4.3)

where $\bar{f} = \varepsilon$, the tenant can compute $n_{\max}$ as follows:

$$n_{\max} = \left\lfloor -\frac{m \cdot \ln(2)^2}{\ln(\varepsilon)} \right\rfloor \tag{4.17}$$

Then, the tenant can compute $\bar{k}$ by substituting $n_{\max}$ to n in the Equation (4.2). The number of updates left until the greedy renew is $u = n_{\max} - c$. Through Equation (4.17) we obtain:

$$u = \left\lfloor -\frac{m \cdot \ln(2)^2}{\ln(\varepsilon)} \right\rfloor - c \tag{4.18}$$

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 (4.16). 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 methodology to estimate the optimal BF size with respect to the tenant requirements.

4.5 Overhead minimization

We have shown that a greedy renew strategy requiring larger BFs reduces network overhead in an update scenario, at the cost of an 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 4.5.1, we introduce the cost models for cloud database services. In Section 4.5.2, we describe our cost minimization methodology for a the stateful version of the protocol in which authorized clients always know the amount of values stored in the BFs. In Section 4.5.3, we extend the minimization methodology for an alternative stateless version of the protocol. Table 4.2 summarizes the main parameters used in the models.

4.5.1 Costs and network usage models

We assume that the tenant is using a typical pay-per-use cloud database service [2, 45, 75], where the costs are a function of storage, ingoing network

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 4.2: Model parameters.

and outgoing network traffic. Hence, the total cost of the service can be modeled by the following equation:

$$Cost = Cost(Storage) + Cost(NetIn) + Cost(NetOut) \tag{4.19}$$

Let us define φ_s, φ_i and φ_o as the cost weights related to storage, ingoing and outgoing network, normalized with respect to the total cost ($\varphi_s + \varphi_i + \varphi_o = 1$):

$$\varphi_s = \frac{Cost(Storage)}{Cost} \tag{4.20}$$

$$\varphi_i = \frac{Cost(NetIn)}{Cost} \tag{4.21}$$

$$\varphi_o = \frac{Cost(NetOut)}{Cost} \tag{4.22}$$

If the tenant database is already deployed in the cloud, he 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., [40, 51, 109]). In

any case, it is important to evaluate the network usage that depends on the type and number of operations executed on the database. Let us 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 and ingoing network usage $NetIn$ and $NetOut$ as:

$$NetIn = \omega_I \cdot \mathcal{I}_I(I) + \omega_U \cdot \mathcal{I}_U(U) \tag{4.23}$$

$$NetOut = \omega_S \cdot \mathcal{O}_S(S) + \omega_U \cdot \mathcal{O}_U(U) \tag{4.24}$$

We now proceed to explain the components of Equations (4.23) and (4.24). In the definition of $\mathcal{I}_I(I)$ we assume that any insert operation creates a new tuple whose size is the average tuple size, that 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 modeled as follows:

$$\mathcal{I}_I(I) = t + m + d \tag{4.25}$$

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 define the workload U as a set of tuples (ω, p, o), each describing a single class of update operations. 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 then be expressed as the weighted average of the sizes of updated values across the different classes of update operations, that is:

$$\bar{p}_u = \sum_{(\omega,p,o)\in U} \omega \cdot p \tag{4.26}$$

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:

$$\mathcal{I}_U(U) = m + d + t \cdot \bar{p}_u \tag{4.27}$$

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

We now 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. 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:

$$\mathcal{O}_S(S) = m + d + t \cdot \sum_{(\omega,p)\in S} \omega \cdot p \tag{4.28}$$

Finally, the outgoing network usage $\mathcal{O}_U(U)$ due to update operations is more difficult to model because it 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:

$$\mathcal{O}_U(U) = m + d + t \cdot [1 - (1 - \mu) \cdot (1 - \bar{p}_u)]\,, \tag{4.29}$$

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 4.2) 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 (4.18)), choosing greater values of m allows us to reduce the frequency of renew operations. However, it 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 (4.29). Then, by using this optimal solution, we are able to minimize Equation (4.19).

To optimize Equation (4.29) we must express μ in terms of m. In the following we discuss how μ depends on m depending on the capability of clients to track the number of values stored in the BFs. In particular, we consider two variants of the protocol:

- a *stateful* variant in which the states of BFs are known (as an example, an architecture based on a *proxy* that tracks the update operations issued by all clients, see Section 4.5.2);
- a *stateless* variant for distributed clients that operate on the cloud database *without any intermediary servers* and without knowing the states of BFs (see Section 4.5.3).

4.5.2 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 because of the 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 4.5.3.

Let λ be the number of greedy update operations between two consecutive renewals. The renew frequency rate μ can be expressed as a function of λ as following:

$$\mu = \frac{1}{1 + \lambda} \tag{4.30}$$

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 (4.18)), and the expected amount of values modified by each update operations, namely $\bar{o}_u$:

$$\lambda = \frac{u}{\bar{o}_u} \tag{4.31}$$

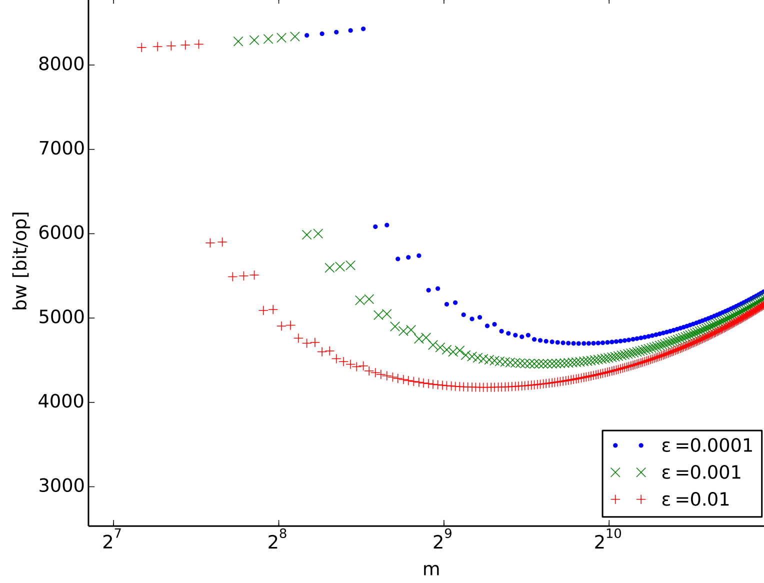


Figure 4.1: Estimated average network usage for each update operation as a function of the BF size m.

where:

$$\bar{o}_u = \sum_{(\omega,o,p)\in U} \omega \cdot o \tag{4.32}$$

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 (4.18) as following:

$$u = \max\left\{x \in \mathbb{N}_0 | x \leqslant -\frac{m \cdot \ln(2)^2}{\ln(\varepsilon)} - c, x = \sum_{(d,o)\in\mathbb{N}_0\times U} d \cdot o\right\} \tag{4.33}$$

The outgoing network usage due to update operations can be estimated by substituting Equations (4.33), (4.31) and (4.30) in (4.29).

For the sake of clarity, we describe an example by referring to Figure 4.1. This figure shows the behavior of Equation (4.29), 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 $t = 8000\,(1KB)$. 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 4.1 reports also three scenarios that correspond to the acceptable false positive rates $\varepsilon = 1\%, 0.1\%, 0.01\%$. For each of them, through Equation (4.16), 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 4.1 we define "segment" a set of close, aligned points of the same color. 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 (4.33). As an example, let us consider the first "segment". The first value of the segment 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 $1, \dots, 4$ 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:

$$M = \left\{ m \mid \left\lfloor -\frac{m \cdot \ln(2)^2}{\ln(\varepsilon)} - c \right\rfloor \in \left\{ \sum d \cdot o \right\}_{(d,o) \in \mathbb{N}_0 \times U} \right\} \tag{4.34}$$

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.

$$\dot{m}_0 = -\frac{(c - \bar{o}_u) \cdot \ln \varepsilon}{\ln (2)^2} + \frac{\sqrt{\bar{o}_u \cdot t \cdot \ln \varepsilon \cdot (\bar{p}_u - 1)}}{\ln (2)} \tag{4.35}$$

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 minimize the overall cloud service cost is an immediate extension of Equation (4.35). As described by Equations (4.23) – (4.27), 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 (4.22)) and the workload frequency ω_u (see Equation (4.23)) to compute the minimum of the continuous form of the Equation (4.19) as following:

$$\hat{m}_0 = -\frac{(c - \bar{o}_u) \cdot \ln \varepsilon}{\ln (2)^2} + \frac{\sqrt{\omega_u \cdot \varphi_o \cdot \bar{o}_u \cdot t \cdot \ln \varepsilon \cdot (\bar{p}_u - 1)}}{\ln (2)} \tag{4.36}$$

The optimal BF size m_{best} is the nearest neighbor of $\hat{m}_0$ in the set M, computed through Equation (4.34).

Scenario with one class of update. If a single class of update operations is defined in the workload, then it is possible to define a closed-form equation to compute the optimal Bloom filter size m_{best} if the operations workload includes only one class of update. In this case, the number of values inserted in the BF for each update operation is constant. Hence, the amount of operations between two renewal λ is constant as well, and can be computed as following:

$$\lambda = \left\lfloor \frac{u}{o} \right\rfloor \tag{4.37}$$

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

Thanks to Equation (4.37), the set of the local minima M can be defined by the following sequence:

$$M = \left\{ \left\lceil -\frac{(\lambda \cdot o + c) \cdot \ln \varepsilon}{\ln(2)^2} \right\rceil \right\}_{\lambda \in \mathbb{N}_0} \tag{4.38}$$

We obtain this equation by substituting Equations (4.18) and (4.31) in (4.29) and inverting.

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 happens at 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 (4.36) in (4.38) and computing the inverse as following:

$$\lambda_{best} = \left\| -\frac{\hat{m}_0 \cdot \ln(2)^2}{\omega_u \cdot \varphi_o \cdot o \cdot \ln \varepsilon} + \frac{c}{\bar{o}_u} \right\| \tag{4.39}$$

where $\|$ represents the round operator.

4.5.3 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, without knowing 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.

Here we propose the following implementation: each update operation is greedy with a given probability, otherwise it renews the BF. This approach suffers 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, analyze the behavior of this protocol and minimize its overhead.

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 (4.29), 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:

$$\mu = 1 - \sqrt[\lambda]{1-q} \tag{4.40}$$

One can obtain the equation estimating the outgoing network usage due to updates by substituting Equation (4.40) in (4.29) as following:

$$\mathcal{O}_U(U) = m + d + t \cdot \left[1 - (1 - \bar{p}_u) \cdot \sqrt[\lambda]{1-q}\right] \tag{4.41}$$

Let us compare the behavior of this function with that of the stateful protocol in Section 4.5.2 by referring to Figure 4.2. 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 4.1. 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 (4.40). One can estimate the outgoing network usage caused by update operations by substituting the Equation (4.31) in (4.41). We can compute the optimum BF size $\dot{m}_0$ of the continuous form of the Equation (4.41) as following:

$$\dot{m}_0 = -\frac{c \cdot \ln(\varepsilon)}{\ln(2)^2} - \frac{\bar{o} \cdot \ln(1-q) \cdot \ln(\varepsilon)}{2 \cdot \ln(2)^2 \cdot W(x)}, \tag{4.42}$$

$$x = -\frac{1}{2} \cdot \sqrt{\frac{\bar{o} \cdot \ln(1-q) \cdot \ln(\varepsilon)}{t \cdot (1-\bar{p}) \cdot \ln(2)^2}}, \tag{4.43}$$

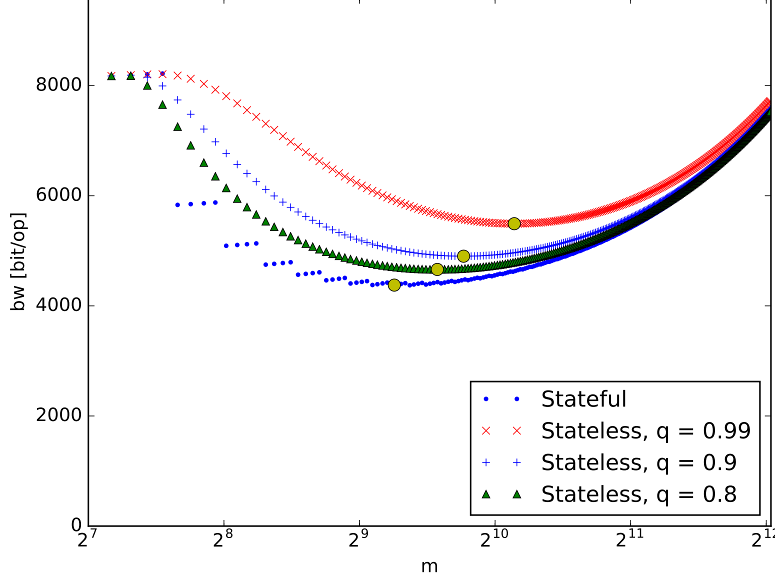


Figure 4.2: Estimated average network usage as a function of the BF size m for the stateless protocol.

where $W(\cdot)$ is the *Lambert W function* [31].

We can extend this evaluation to compute the best value of $m = \hat{m}_0$ to minimize overall costs (see Equation (4.19)) as done for the stateful protocol (see Equation (4.36)):

$$\hat{m}_0 = -\frac{c \cdot \ln(\varepsilon)}{\ln(2)^2} - \frac{\bar{o} \cdot \ln(1-q) \cdot \ln(\varepsilon)}{2 \cdot \ln(2)^2 \cdot W(\hat{x})}, \tag{4.44}$$

$$\hat{x} = -\frac{1}{2} \cdot \sqrt{\frac{\bar{o} \cdot \ln(1-q) \cdot \ln(\varepsilon)}{\varphi_o \cdot \omega_u \cdot t \cdot (1-\bar{p}) \cdot \ln(2)^2}}, \tag{4.45}$$

4.6 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 to each value stored in the database. In particular, we adopt HMAC-SHA256. This solution causes a

		Average tuple size $t = 500$		
Operation workloads		Select $p = 10\%$	Select $p = 50\%$	Insert
Plaintext		50	250	500
VLH (overhead)		82 (64%)	410 (64%)	820 (64%)
TLH (overhead)		532 (964%)	532 (112%)	532 (6.4%)
EBF (overhead)	$\varepsilon = 10^{-3}$	70 (40%)	270 (8%)	520 (4%)
	$\varepsilon = 10^{-4}$	76 (52%)	276 (10.4%)	526 (5.2%)
	$\varepsilon = 10^{-5}$	82 (64%)	282 (12.8%)	532 (6.4%)

Table 4.3: Storage and network usage comparison for VLH, TLH and EBF integrity solutions.

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 associates a MAC to a set of values. In particular, we associate a HMAC-SHA256 to all the values of the same row. The resulting scheme has the same structure of Table 4.1, but the last column is used to store a HMAC rather than an encrypted BF. 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 all the values stored in the same row. The retrieval of unnecessary values causes an increase in 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. In particular, we investigate the performance of both the stateful and stateless protocols by considering acceptable false positive rates $\varepsilon = 10^{-2}, 10^{-3}, 10^{-4}$ and maximum false positive thresholds $q = 0.9, 0.99$. In the following we refer to the proposed solution as *EBF* (Encrypted BF).

We first analyze the performance of micro-benchmarks with different database characteristics and workloads, then we analyze the storage and network overhead of a realistic scenario by referring to the TPC-C workload [108].

4.6.1 Micro-benchmarks

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 [101] 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 (4.16), 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 4.3 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 shows the bytes downloaded in the case of a select that retrieves only one value. The performance of EBF and VLH is optimal, since the tenant only needs to retrieve 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 since the tenant has to retrieve one control structure for each value.

4.6.2 Mixed operations

In this section we consider the TPC-C standard OLTP benchmark [108], commonly adopted for evaluating database performance. The contribution of the section is twofold: first, it shows how to leverage the proposed overhead minimization methodology (see Section 4.5); second, it demonstrates the performance advantages of using EBF in real-world 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 4.4. We distinguish two kinds of parameters: those that describe the database schema, and those that describe the workload. The amount of columns c and the average size of the tuples t are described in the first two rows of the table. Moreover,

	Schema parameters		Workload parameters			
Tables	*# cols* (c)	*Avg tuple size* ($t/8$)	*# updated values* $\{o\}$	*Perc size* $\{p\}[\%]$	*Exec freq* $\{\omega\}[\%]$	*Update freq* $(\omega_u)[\%]$
warehouse	9	99	1	9.1	100	32.8
district	11	107	1.1	8.41, 3.7	48.8, 51.2	50
item	5	87	(none)	(none)	(none)	0
customer	21	681	1, 4, 3	1.3, 76.8, 3.38	8.53, 10.7, 80.8	33.8
history	8	53	(none)	(none)	(none)	0
stock	17	318	1	1.57	100	47.9
order	8	34	1	11.8	100	7.02
new_order	3	12	(none)	(none)	(none)	0
order_line	10	62	1	6.5	100	6.56

Table 4.4: Analysis of the tables of a TPC-C compliant database

the TPC-C standard [108] 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 consider 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 update operations divided by the frequencies of all the operations executed on the table.

We now investigate the performance of integrity strategies applied to table *customer* by referring to Figures 4.3 and 4.4 (for space reasons, we limit our analysis to this table). 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 4.3 details network usage for increasing BF sizes when the acceptable false positive rate ε is equal to 10^{-3}. Figure 4.4 details network usage for decreasing acceptable

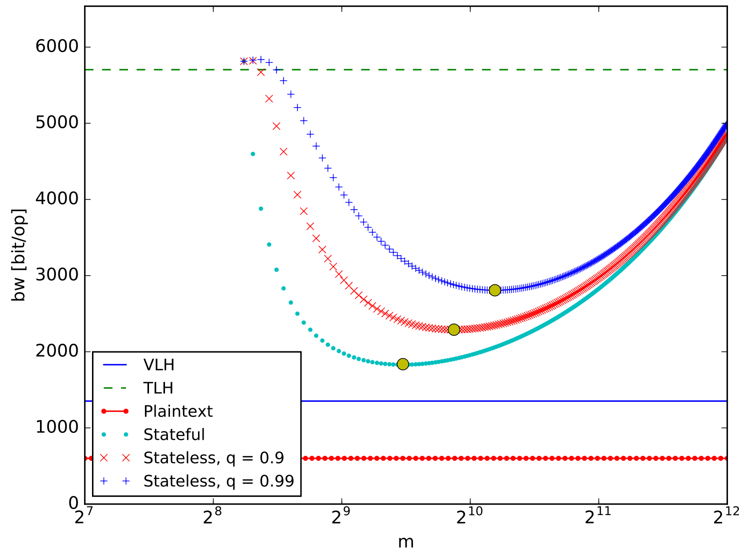


Figure 4.3: Estimated average network usage for each update operation in terms of the BF size m.

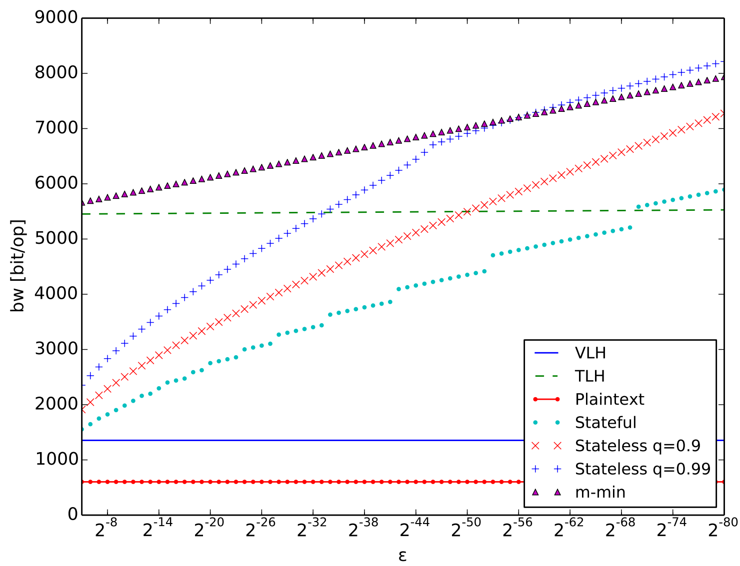


Figure 4.4: Estimated average network usage for each update in terms of acceptable false positive rate ε.

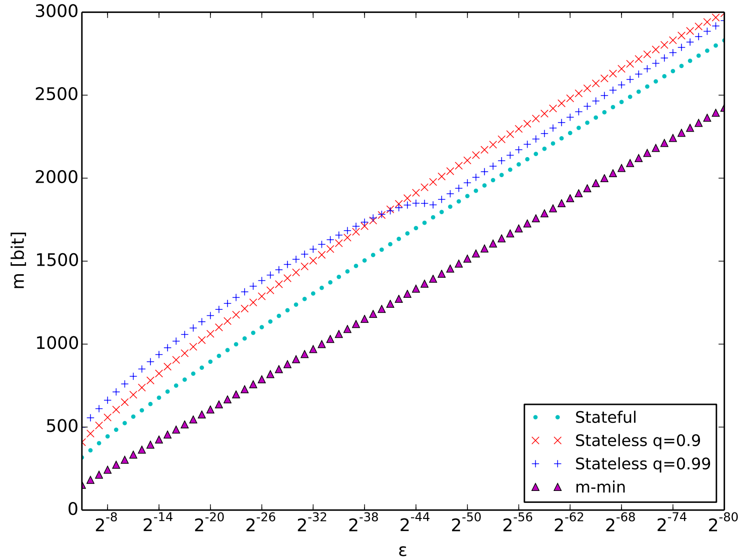


Figure 4.5: BF size in terms of the acceptable false positive rate ε.

false positive rates in the range $2^{-5}, \ldots, 2^{-80}$. In Figure 4.3 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 to achieve results comparable to that of VLH. In Figure 4.4 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 4.4 and 4.4 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 4.5 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.

Chapter 5

Integrity guarantees for key-value databases

We propose two novel protocols to detect integrity violations in cloud key-value databases, including authenticity, completeness and freshness guarantees. The former protocol, named *Probus*[1], considers an advanced threat model in which tenants are not trusted by the cloud provider. A tenant can detect any unauthorized modifications to his data as well as incorrect results generated by the cloud service in the context of cloud key-value database services. The cloud provider can defend him-self against false accusations by a malicious tenant. The latter protocol, named *Bulkopt*, considers particular operations in which data are inserted and read in bulks. This protocol produces efficient proofs of integrity whose size is independent of the number of records involved in the operations.

The chapter describes *Probus* and *Bulkopt* in Sections 5.1 and 5.2.

[1]From the latin word *prŏbus* that means *honest*, *good*.

5.1 A protocol for mutual integrity guarantees

Common scenarios in which a tenant adopts integrity solutions assumes that the cloud provider is an untrusted actor and the tenant is trusted. These are acceptable assumptions until the tenant has no benefits in behaving maliciously. However, a cloud provider might be interested in offering integrity guarantees within the service level agreements (e.g., by asking additional fees). In this scenario, the tenant would use an integrity protocol to detect incorrect results by the database and might be allowed to ask for refunds if he receives incorrect results or corrupted data. This scenario raises threats for the cloud provider: a tenant could falsify integrity violations and build malicious accusations to gain from the agreed refunds. Thus, the cloud provider needs to defend against false accusations.

Probus is a protocol that allows a tenant to detect integrity violations on his data stored in a cloud database, and the cloud provider to proof his correct behavior in case of false accusations by a dishonest tenant. Probus supports full CRUD workloads (Create, Read, Update and Delete operations) that characterize cloud key-value databases and generates cryptographic proofs both for the tenant and the cloud provider for each correctly executed operation. Efficiency of the proposal is guaranteed by using cryptographic proofs generated for each operation (online) that are very efficient and of small size, while most computations are executed as periodic batch operations, not affecting the cloud database service response times. Although similar design choices were proposed in literature for file storage [94], Probus is the first protocol that can be used in the context of cloud databases (see Chapter 2.2 for details on related work). The protocol requires clients to store locally cryptographic proofs exchanged with the cloud provider. Depending on the clients storage capabilities, certain thresholds are fixed for the execution of the periodic batch operations. These operations aggregate proofs in authenticated data structures, optimizing the client storage usage, and store most of the proofs to the same cloud database in a secure way. Our approach has many advantages: on-line and off-line network overhead are maintained low and computational overhead that affect on-line operations is very low as well; proofs are given both to the tenant and to the cloud provider; periodic batch operations can be adapted depending on the characteristic of the clients; the protocol design is modular and can be implemented with the most convenient protocols for authenticity guarantees, such as our proposal described in Chapter 4.

Let us summarize the structure of the section. Section 5.1.1 describes the

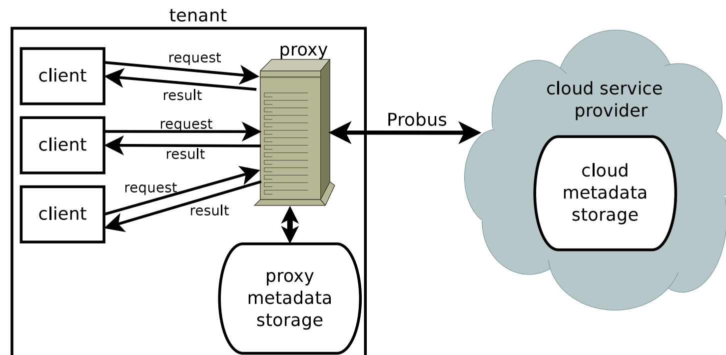


Figure 5.1: Candidate proxy-based architecture to implement Probus.

system model in which Probus operates. Sections 5.1.2 and 5.1.3 detail the protocols operations and how they enable the tenant and the cloud provider to detect misbehaviors. Section 5.1.4 describes how Probus achieves low performance overhead by periodically aggregating cryptographic proofs.

5.1.1 System model and security guarantees

We describe the protocol by referring to a candidate proxy-based architecture, as represented in Figure 5.1. The proxy is a server deployed within the tenant private network and has access to local storage (*proxy metadata storage*) that uses to maintain all the due metadata to execute the protocol, such as cryptographic keys and proofs. Tenant clients can issue insert, read, update and delete operations to the proxy that mediates all interactions between clients and the cloud key-value database. The proxy handles clients' requests, executes the Probus protocol with the database, and delivers results to the clients. Operations are defined over a typical key-value framework, as following:

- *create operations* insert a new *(key,value)* couple in the database;
- *read operations* retrieve the value that is associated to a given key;
- *update operations* substitute the value currently associated to a given key with a different, more recent value;
- *delete operations* remove a given key and the associated value from the database.

These operations can be executed concurrently by one or more tenant clients, and are issued to the local proxy. The proxy has to forward these requests

to the cloud key-value database, as well as to execute the Probus protocols, that must also be supported by the cloud provider.

We assume that all clients and the proxy within the tenant private network are trusted. In particular, addressing insider threats that have legitimate access to tenant's assets and try to compromise the security of tenant's data outsourced to the cloud is beyond the scope of this protocol. We also assume that the proxy behaves correctly by executing the protocol without altering or ignoring requests issued by legitimate clients. We assume that the tenant and the cloud provider do not trust each other. We highlight that this assumption is stronger than that commonly adopted in related literature (see Section 2.2), that usually focuses on malicious cloud providers and honest tenants. Hence, both the tenant proxy and the cloud provider may execute incorrectly the Probus protocols in their favor. In particular, a tenant may falsely accuse the cloud provider of incorrect behavior even if the cloud provider operates correctly. As an example, let us consider a tenant employee that has to execute data entry and fails in performing this task by forgetting to insert some data in the key-value database, or by inserting incorrect data. The tenant can then accuse the cloud provider of providing incomplete or incorrect results. Without the proposed protocol, the cloud provider could not demonstrate its correct behavior to third parties, and its reputation would be damaged by false accusations. Probus allows the tenant to validate all results generated by the cloud provider by verifying three main properties: *completeness*, *authenticity* and *freshness* (see Section 2.2). In the operations framework described above, these properties translate to the following guarantees:

- *omitting results*; e.g., the cloud provider produces an empty or partial result;
- *producing false results*; e.g., the cloud provider returns a fake result, that has never been inserted by the tenant in the past;
- *producing stale results*; e.g., the cloud provider returns a value that was previously inserted by a client, but that has since being updated.

Probus does not make any assumption about the causes of these incorrect operations. They may be due to any events, such as hardware or software failures, external attacks to the cloud provider's infrastructure, or attacks from a cloud insider. By using Probus the tenant can verify the result of each Read operation to ensure that these three properties hold. Whenever a violation of one of them is detected, the tenant can generate a cryptographic *proof of misbehavior* that is verifiable by any third party, including

the misbehaving cloud provider. On the other hand, if the tenant challenges the completeness, authenticity or freshness of a result generated by the cloud provider, the provider can defend itself by producing a proof of its correctness that is verifiable by any third party, including the tenant. Hence, through the novel protocol proposed in this section it is possible to enforce correct behavior of both the tenant and the cloud provider without requiring any trust among the two parties. Probus makes it possible to formally introduce guarantees of authenticity, completeness and freshness in the SLAs of cloud services.

5.1.2 Protocol operations

To execute Probus, the proxy and the cloud provider must have a public and a secret keys. We denote pk_p and sk_p as the public and secret keys of the proxy, while pk_c and sk_c represent the public and secret keys of the cloud service. The proxy and the cloud provider maintain also a set of metadata that are necessary to prove that an operation was executed, to reconstruct the operation history and to generate cryptographic proofs of misbehavior. We identify two main types of metadata: *chain hashes* and *attestations*.

We define ch_p as the *chain hash* that is maintained by the proxy. The proxy updates ch_p whenever a tenant client issues an operation that modifies the values stored in the key-value database (that is, Create, Update and Delete operations). Similarly, ch_c denotes the chain hash that is maintained and stored by the cloud provider. This chain hash is updated by the cloud provider for every operation, including Reads that do not modify the content of the key-value database. This design choice makes it possible to create a verifiable and ordered history of all the operations executed by all the tenant clients.

Attestations are digital signatures that the proxy and the cloud provider generate to provide *authenticity* and *non-repudiability* guarantees on protocol messages and on their results. The proxy generates an attestation by signing each operation that modifies data through its secret key sk_p. Cloud attestations are digital signatures generated by the cloud provider for all its responses (including results of Read operations) through its secret key sk_c. Depending on the size of the cryptographic algorithms used to generate signatures and hashes, the size of each attestation may vary from about 100 Bytes (in case of ECDSA signatures) to about 500 Bytes (in case of RSA signatures). Since attestations have to be maintained by the tenant proxy, the use of Probus becomes convenient for the use cases in which the values stored in a key-value database have a non-trivial size (more than a few kilobytes), such as large text documents, images and any multimedia content.

Probus requires an initialization phase before the execution of any other operation. In this phase, the proxy initializes a storage space, defined as the *proxy metadata storage*, that will be used to maintain a local copy of all the cloud attestations. Moreover the proxy initializes its chain hash ch_p with a random value, and signs it with its secret key sk_p. Both the chain hash and the signature are sent to the cloud service. The cloud service verifies the signature and initializes its chain hash ch_c to the same value of ch_p. Moreover, it initializes a *cloud metadata storage* that will be used to store the two chain hashes and all proxy attestations.

After the initialization phase clients can issue any CRUD operation to the cloud database through the proxy. We describe the Probus protocols for the supported operations by assuming that both the proxy and the cloud provider operate correctly. We then describe how to detect incorrect behaviors in Section 5.1.3.

Create operations

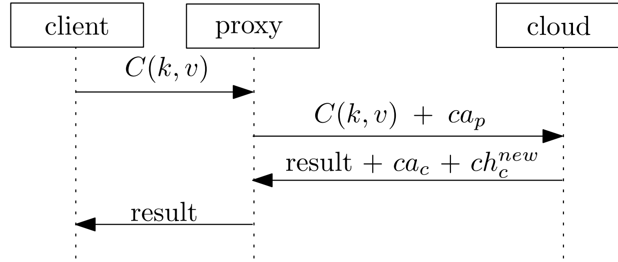


Figure 5.2: Message exchange for create operations.

The exchange of the messages that are necessary for the execution of Create operations is shown in Figure 5.2. To insert a new association between a key k and a value v in the cloud key-value database, a client issues a Create operation $C(k, v)$ to the local proxy. The proxy handles this operation by retrieving the last value of the proxy chain hash ch_p from the *proxy metadata storage*. The proxy updates the value of the chain hash according to the following equation:

$$ch_p^{new} = H(ch_p \parallel H(v)) \tag{5.1}$$

where $H(\cdot)$ denotes a cryptographic hash function and $\parallel$ denotes string concatenation. Moreover, the proxy computes the *signature* is_k for the key k, defined as Equation (5.2):

$$is_k = sign(H(\text{`}C\text{'} \parallel k \parallel H(v)), sk_p) \tag{5.2}$$

where '*C*' is a tag associated to the Create operation. Then the proxy computes the *proxy create attestation* ca_p, defined as Equation (5.3):

$$ca_p = is_k \parallel sign(ch_p^{new}, sk_p) \tag{5.3}$$

After these operations, the proxy sends the Create operation $C(k, v)$ and the attestation ca_p to the cloud service. The cloud service computes ch_p^{new} as shown in Equation (5.1) and updates the cloud chain hash according to the following equation:

$$ch_c^{new} = H(ch_c \parallel H(v)) \tag{5.4}$$

The cloud service then verifies the attestation ca_p. If it is correct, the cloud service stores ca_p, $H(v)$ and ch_p^{new} in the *cloud metadata storage*, and executes the operation requested by the client, thus generating its result. In the case of a Create operation, the two possible results are a code confirming the creation of the tuple or an error code if the key is already included in the key-value database. At this point, the cloud provider can compute a *cloud create attestation* ca_c, defined as Equation (5.5):

$$ca_c = sign(H(\text{'}C\text{'} \parallel k \parallel H(v) \parallel ch_p^{new} \parallel ch_c^{new}), sk_c) \tag{5.5}$$

Finally, the cloud provider sends the result, ca_c and ch_c^{new} to the proxy. The proxy verifies the signature included in ca_c and stores it in the *proxy metadata storage* together with the hash of the created value $H(v)$, ch_p^{new} and ch_c^{new}. Finally, the proxy delivers the result to the client that issued the Create operation.

Read operations

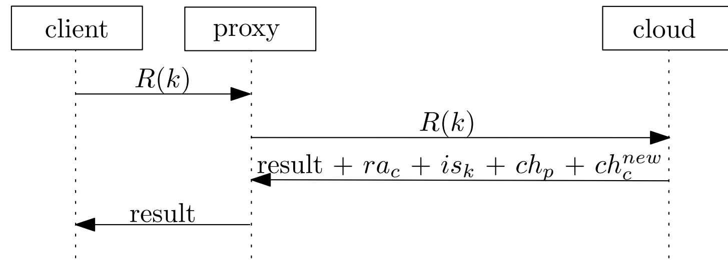


Figure 5.3: Message exchange for read operations.

The execution of a Read operation is represented in Figure 5.3. A client issues to the proxy a Read operation $R(k)$ to retrieve the value associated to a given key k. The proxy relays this operation to the cloud service that

executes it and generates the corresponding result. It then retrieves the *proxy create attestation* ca_p related to k from the cloud metadata storage and extracts the signature is_k. After these operations, it updates the cloud chain hash ch_c^{new} according to Equation (5.4) and computes a *cloud read attestation* defined in the following equation:

$$ra_c = sign(H(\text{‘}R\text{’} \parallel k \parallel H(v) \parallel is_k \parallel ch_p \parallel ch_c^{new}), sk_c) \tag{5.6}$$

Finally, the cloud provider sends the result, ra_c, is_k, ch_p and ch_c^{new} to the proxy. The proxy verifies the correctness of ra_c and is_k, stores ra_c and a hash of the corresponding value $H(v)$ in the proxy metadata storage and forwards the result to the client that initiated the Read operation.

Delete operations

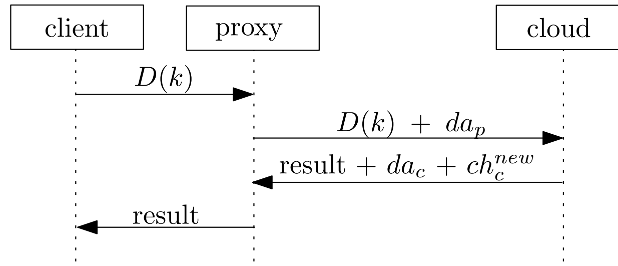


Figure 5.4: Message exchange for delete operations.

Figure 5.4 shows the protocol interactions related to the execution of a Delete operation. A client issues to the proxy a Delete command $D(k)$ to delete a key and the corresponding value from the cloud key-value database. The proxy retrieves the latest value of the proxy chain hash ch_p from the *proxy metadata storage*. The proxy updates the value of the chain hash according to the following equation:

$$ch_p^{new} = H(ch_p \parallel k) \tag{5.7}$$

and computes a *proxy delete attestation* da_p, defined by Equation (5.8):

$$da_p = sign(H(\text{‘}D\text{’} \parallel k), sk_p) \parallel sign(ch_p^{new}, sk_p) \tag{5.8}$$

Then, the proxy sends the Delete operation $D(k)$ and the attestation da_p to the cloud service. The cloud service computes ch_p^{new} as shown in Equation (5.7) and updates the cloud chain hash according to Equation (5.9):

$$ch_c^{new} = H(ch_c \parallel k) \tag{5.9}$$

The cloud service verifies whether da_p is correct. In the positive case, it stores da_p and ch_p^{new} in the *cloud metadata storage*, and executes the Delete operation. After that the cloud provider computes a *cloud delete attestation* da_c, defined by (5.10):

$$da_c = sign(H(\text{'}D\text{'} \parallel k \parallel ch_p^{new} \parallel ch_c^{new}), sk_c) \tag{5.10}$$

The result, da_c and ch_c^{new} are sent to the proxy, that verifies da_c, stores it together with ch_p^{new} and ch_c^{new} in the proxy metadata storage, and delivers the result to the client that issued the Delete operation.

We highlight that a new attestation is stored by the proxy and the cloud provider for each operation, thus the metadata storage size increases linearly with the number of operations. This characteristic would make the proposed solution impractical for any Read- and Update-intensive workloads. To solve this issue, we introduce the concept of *epoch*, that we detail in Section 5.1.4.

5.1.3 Misbehavior detection

We describe how the proxy can leverage the metadata generated by Probus to verify authenticity, completeness and freshness of all the results received by the cloud provider.

Authenticity. Authenticity verification is executed automatically by the proxy whenever the result of a Read operation is received. The proxy verifies the correctness of the signature is_k (see Equation (5.2)). If this signature is verified, the proxy is assured that the value retrieved by the cloud storage has been inserted in the key-value database by a Create operation issued in the past by the proxy itself. This guarantees data authenticity but not its freshness. On the other hand, if is_k is not verified, then the proxy knows that the cloud service operated incorrectly. In this case, the proxy can leverage ra_c (see Equation (5.6)) to prove the misbehavior of the cloud provider. Since ra_c is signed by the cloud provider, it can be used to demonstrate to a third party that the value returned by the cloud provider has not been previously inserted by the tenant. We remark that if the cloud provider operates correctly, he can always retrieve the signature is_k related to any key included in the key-value database, thus disproving any tenant that falsely accuses the cloud provider of having tampered with its data.

Completeness. Whenever the tenant wishes to verify the completeness for a given key, the proxy analyzes all the *cloud read attestations* stored in the *proxy metadata storage* looking for those related to Read operations for which

the cloud provider did not return any value. For each of these operations, the proxy looks for the most recent Create or Delete operation performed on the same key. If the proxy is unable to find any Create operation for the same key, then the cloud provider is correct, and the key does not exist in the key-value database. Similarly, the cloud provider is correct if the most recent operation is a Delete, since it shows that the tenant removed the key from the key-value database. On the other hand, the proxy can detect a violation of completeness if the most recent operation performed on the key is a Create. Since this operation has not been followed by a Delete for the same key, the key has to exist in the key-value database, hence the cloud provider should have returned the value associated to it. In order to accuse the cloud provider, the proxy can produce a *proof of misbehavior* including:

- all metadata related to the Read operation that did not return any value and that is challenged by the tenant (*challenged Read*);
- all metadata related to the Create operation performed over the same key that precedes the *challenged Read* (*challenged Create*);
- all metadata related to all the operations executed between the *challenged Create* and the *challenged Read.*

As all metadata include the cloud chain hash, this proof demonstrates that the Read operation was preceded by a Create and that no Delete for the same key was performed between the two. Hence the cloud provider violated the property of completeness. Similarly, if the cloud provider operates correctly, he can always demonstrate that a Read for which no result was returned is either related to a key that was never created by the tenant, or to a key that was Deleted and for which no new Create operation has been issued after the Delete. Hence the cloud provider can disprove a tenant that falsely accuses it of violating completeness.

Freshness. We describe the operations performed when the tenant wants to verify the freshness of the result returned by the cloud service for a given key. In this case the proxy analyzes all the *cloud read attestations* ra_c stored in the *proxy metadata storage* related to Read operations that returned a value different from *null* (we remark that the correctness of Read operations that returned a *null* value are verified by completeness checks). For all of them, the proxy looks for the most recent Create or Delete operation performed on the same key. If such operation is a Create, and if the attestations related to that Create and to the Read that is being checked have the same value $H(v)$, then the cloud service behaved correctly. On the other hand, we

distinguish two situations which identify a violation of freshness performed by the cloud provider. In the first case a violation occurs when the Read is preceded by a Delete. If there are no Create operations between them, then the cloud provider did not remove the key from the key-value database. If the cloud service behaved correctly, it would have returned a *null* value. To accuse the cloud provider, the proxy can produce a *proof of misbehavior* with the following elements:

- all the metadata related to the Read operation that returned a stale value and that is challenged by the tenant (*challenged Read*);
- all the metadata related to the Delete operation that precedes the *challenged Read* (*challenged Delete*);
- all metadata associated to all the operations performed by the proxy between the *challenged Read* and the *challenged Delete*.

These elements demonstrate that the Read was preceded by a Delete on the same key and all the operations performed between them did not modify the key. The second case occurs when the proxy detects a Read associated to an $H(v)$ that is different of the $H(v)$ of the Create previously performed on the same key. This implies that the cloud provider ignored an Update operation (that is executed as a Delete followed by a Create) and returned a stale value. The *proof of misbehavior* generated by the proxy is similar to the previous one, with the only difference that now the *challenged Delete* is replaced by the Create operation having a different value of $H(v)$. In both cases, a honest cloud provider can prove its correctness by demonstrating that the *challenged Read* is preceded by a Create performed by the proxy associated to the same $H(v)$ and that between these two operations the value associated to the key was not modified by the tenant. In such a way, the cloud provider can always disprove false accusations generated by a malicious tenant.

5.1.4 Periodic signatures aggregation

Probus allows the tenant proxy to verify authenticity, completeness and freshness of the data outsourced to the cloud key-value database. However, the metadata stored in the proxy and in the cloud metadata storage areas grow linearly with the numbers of operations executed by the clients. To limit this overhead, we improve the protocol by introducing the concept of *epochs*. An epoch is defined as a set of contiguous operations, whose length is configurable and defined by the tenant. Within the same epoch, operations are

executed according to the protocol described in Sections 5.1.2. At the end of an epoch, the proposed protocol enforces a verification of authenticity, completeness and freshness of all the operations belonging to this epoch, as described in Section 5.1.3. If these properties are verified, the proxy and the cloud service delete the attestations related to the epoch that ended, and substitute them with a small authenticated data structure. After this verification, a new epoch can begin. This operation makes it possible to greatly reduce space overheads related to the storage of proxy and cloud metadata. In particular, storage overheads for the tenant proxy is reduced to about 250 Bytes for each epoch, independently of the size of the key-value database and the number of operations performed.

Each Probus epoch is uniquely identified by a number. In the initialization phase (see Section 5.1.2) the tenant and the cloud provider agree to start the first epoch that is identified by number 0. Subsequent epochs are identified by increasing the epoch number by one. Let us assume that the tenant triggers the end of an epoch. The proxy analyzes all attestations stored in the *proxy metadata storage* and verifies authenticity, completeness and freshness of all the related operations. After that the proxy builds a *Merkle Hash Tree* (MHT) and a *Bloom Filter* (BF).

The MHT is built by using as leaves the attestation of the latest Create or Delete operation executed on all the keys modified within the considered epoch. We remark that keys that were only subject to Read operations are not included in the MHT. This design choice allows to build $MHTs$ that are much smaller with respect to the number of values stored in the key-value database, especially in Read-intensive workloads. Moreover, it represents an improvement with respect to similar solutions proposed in the literature [94] (see Section 2.2). Each leaf contains a tuple of three values: the key k, a tag that identifies the type of operation ('C' for Create or 'D' for Delete) last performed on k; the hash of the value $H(v)$ currently associated to k ($H(\text{'}null\text{'})$ if the operation is a Delete). The leaves are then sorted by the key k. The internal nodes of the tree are built in the classical way, so that each node contains the hash of the concatenation of all the values stored in its child nodes.

The BF associated to a MHT contains all the keys stored in the leaves of the MHT. Using BFs allows Probus to reduce the data that the cloud provider and the proxy exchange to demonstrate that a given key has not been modified within an epoch. MHT and BF are used to compute the *proxy epoch attestation* ea_p as described in Equation (5.11):

$$ea_p = sign(H(n \parallel root \parallel BF), sk_p) \tag{5.11}$$

where n is the number of the epoch that ended and $root$ is the root of the

MHT.

The proxy sends ea_p to the cloud provider. After having received ea_p, the cloud provider retrieves all attestations from the *cloud metadata storage* and uses them to compute MHT and BF as done by the proxy. Then, the cloud service can verify the correctness of ea_p. If ea_p is correct, then the cloud provider removes all the metadata of the current epoch with the only exception of the latest signatures is_k of the keys included in the key-value database. Then, the cloud service stores the MHT and the BF in the *cloud metadata storage*. Finally, the cloud service computes the *cloud epoch attestation* ea_c as described in Equation (5.12), sends ea_c to the proxy and increases the epoch number by one.

$$ea_c = sign(H(n \parallel root \parallel BF), sk_c) \tag{5.12}$$

The proxy then verifies ea_c. If it is correct, the proxy removes all the metadata related to the current epoch, including MHT and BF, and stores ea_c. Finally, the proxy increases the epoch number by one.

We highlight that by removing all attestations related to the epoch that just ended, the proxy is no longer able to verity completeness and freshness of results generated by the cloud service at the beginning of a new epoch. We introduce two new data structures to address this issue: the *Merkle verification object* (mvo) and the *epoch verification object* for epoch n (ev_n).

The mvo is a data structure that makes it possible to efficiently verify whether a given key k is included within a MHT or not. If k is not included in BF, then we know that k has not been modified within the epoch. Hence, mvo is equal to the root of the MHT belonging to the same epoch. On the other hand, if k is included in BF, there are two possibilities. In the former hypothesis, k has been modified within the epoch, hence it must be included among the leaves of the corresponding MHT. In this case, mvo is built as a subset of MHT including the leaf corresponding to k and all intermediate nodes that are needed to compute the root of the MHT. In the latter hypothesis, BF generated a false positive, hence k is not included in MHT. In this case, mvo is built by considering the two leaves storing the two adjacent keys that are lower and higher than k (we recall that leaves are sorted on the basis of k), as well as all intermediate nodes that are needed to compute the root of the MHT.

The *epoch verification object* ev_n is defined as following:

$$ev_n = n \parallel mvo \parallel BF \parallel ea_p \tag{5.13}$$

where n is the number of an epoch and mvo, BF and ea_p are related to the epoch n.

If the proxy reads a key for the first time in the current epoch, and if it wishes to check its completeness and freshness, it executes the protocol shown in Figure 5.5. This protocol is initiated by the proxy, that sends to the cloud service a message $get_{ev}(k)$ to retrieve the set of *epoch verification objects* that is needed to verify completeness and freshness for the key k.

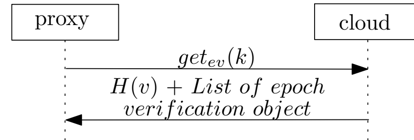


Figure 5.5: First verification of k within the current epoch.

After having received this message, the cloud service determines how many *epoch verification objects* are needed. If k has been modified in the previous epoch, then its *epoch verification object* alone can be used as the cryptographic proof. In the other case, the cloud provider iterates backwards until the most recent epoch in which k has been modified. Let us assume that the current epoch has $n = 4$. If k was modified in the previous epoch, then ev_3 is sufficient. If k was last modified in the epoch $n = 1$, then ev_1, ev_2 and ev_3 are all needed. In the worst case in which the proxy wishes to verify completeness and freshness of a key that has never been created, the cloud service has to go back up to the first epoch. Then the cloud service sends all the needed *epoch verification objects* to the proxy, together with the hash of the value v corresponding to k.

When the proxy receives all the *epoch verification objects* it verifies all the ea_p, thus confirming that k was last modified in a previous epoch or that it has never been created. If the cloud provider violated completeness or freshness, the proxy can use the list of *epoch verification objects* as a proof of misbehavior. On the other hand, by exhibiting the list of *epoch verification objects*, a honest cloud provider can always defend itself against false accusations of a tenant.

5.2 Correctness in bulk operations workloads

We describe Bulkopt, a protocol that guarantees authenticity, completeness and freshness of results produced by the cloud database. Bulkopt recasts the problem of verifying the correctness of results produced by the cloud provider in terms of set operations and produces short cryptographic proofs based on *aggregate signatures* and *extractable collision resistant* (ECR) hash functions [17,24]. This design choice optimizes communication overhead and verification costs of both bulk inserts and read operations of large amounts of records.

We organize the description of the protocol as following. Section 5.2.1 describes the formal model used by Bulkopt to represent data and how to express authenticity and completeness guarantees as set operations. Next sections detail the protocol implementation based on short aggregate signatures and extractable collision resistant hash functions. Section 5.2.2 describes the setup and the key generation phases and Sections 5.2.3 and 5.2.4 describe insert and read operations, including signatures and proofs generation protocols, that combines the benefit of aggregate bilinear signatures and ECR hash functions to produce efficient proofs of correctness for the data outsourcing scenario.

5.2.1 Data model

We model the key-value database as a set of tuple $D = \{(k, v)\}$, where k is the key and v is the value associated to k. The owner populates the key-value database by executing one or more insert operations. For each insert operation the owner sends a set of tuples $B_i = \{(k, v)\}$, where i is an incremental counter that uniquely identifies an insert operation. The set B_i contains at least one tuple, and may contain several tuples in case of bulk insertions. Without loss of generality, in the following we refer to each set of tuple B_i as a *bulk*. We define as K_i the set of keys included in B_i. Hence, after n insertions the database D_n can be modeled as $D_n = \cup_{i=1}^{n} B_i$.

We assume that the server has access to a lookup function that given a set of keys $\{k\}$ allows him to retrieve the set of insert operation identifiers $\{i\}$ in which these keys were sent by the owner. Such function can be obtained by deploying any standard indexing data structure of preference (e.g., a B-tree).

Any client (including the owner) can issue a read operation requesting an arbitrary set of keys $X = \{k\}$. If the server behaves correctly he must return the subset of the database A, defined as:

$$A = \{(k, v)\} = D_n \cap \{(k, v) \in D_n | k \in X\} \tag{5.14}$$

We define R as the set of keys included in A, that is:

$$\forall(k, v) \in A, k \in R \tag{5.15}$$

While executing read operations issued by clients, the server distinguishes two different sets of keys: T and $\bar{T}$.

T is the union of all sets K_i that contain at least one key among those requested by a client:

$$T = \bigcup K_i \mid K_i \cap X \neq \varnothing \tag{5.16}$$

Within each K_i we identify two subsets of keys: $R_i = K_i \cap X$ and $Q_i = K_i \backslash R_i$. We define Q as the union of all sets Q_i, and we note that the union of all sets R_i is equal to set R (see Equation (5.15)). Thus, set Q is the complement of R in T.

$\bar{T}$ is the union of all sets K_i that do not contain any key among those requested by a client:

$$\bar{T} = \bigcup K_i \mid K_i \cap X = \varnothing \tag{5.17}$$

To better explain how these sets are built and the relationships among them, we refer to a simple example shown in Figure 5.6. In this example we have a key-value database on which the owner already executed five bulk insert operations, each involving a different amount of tuples. The keys included in the database are represented by sets K_1 to K_5. We assume that a legitimate clients executes a bulk read operation, asking to retrieve six keys belonging to three different bulk insert. The set of keys requested is represented by X. Since X includes keys belonging to bulks K_1, K_3 and K_4, all keys of these bulks belong to T, while $\bar{T}$ includes all keys belonging in the remaining bulks (K_2 and K_5). Sets R_1, R_3 and R_5 include only the keys requested by the client and belonging to K_1, K_3 and K_5, respectively. Set R includes all the keys belonging to the union of R_1, R_3 and R_5. Sets Q_1, Q_3 and Q_5 include only the keys that were not requested by the client and that belong to K_1, K_3 and K_5, respectively. Finally, set Q includes all the keys belonging to the union of Q_1, Q_3 and Q_5.

Sets Q and $\bar{T}$ are the main building blocks that Bulkopt leverages to identify a violation of the security properties or to prove the correctness of results produced by the server.

Authenticity

Bulkopt builds proofs of authenticity by demonstrating that:

$$R \cup Q \cup \bar{T} = K_D \tag{5.18}$$

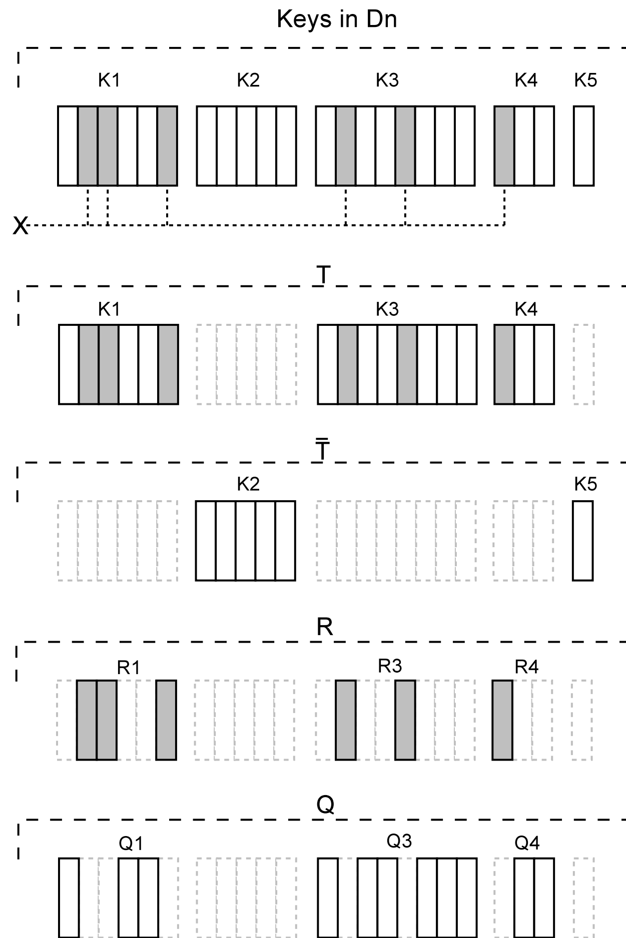


Figure 5.6: Example of sets computed over a key-value database.

where K_D represents the set of keys included in D_n. We recall from Section 5.2.1 that authenticity is violated if the server produces a result containing a key that has not been inserted by the owner. Let us assume that R includes a fake key k_f that has been created by the server but does not belong to K_D. Then it is obvious that Equation (5.18) does not hold, since R is not a subset of K_D.

An obvious solution to demonstrate that R is a subset of K_D would be for the client to have the complete set K_D. Of course this is not applicable, since it would require all clients to maintain a local copy of the whole key-value database. To overcome this issue, Bulkopt requires the owner to maintain a cryptographic accumulator $\sigma(K_D)$ that represent the state of the keys stored in the database D_n. This accumulator is updated after each insert operation and has to be available to all clients. Moreover, the server builds two witness

data structures W_Q and $W_{\bar{T}}$ that represent the sets Q and $\bar{T}$, and sends them to the client together with its response A. We remark that cryptographic accumulators and witnesses are small and fixed-size data structures, that can be transmitted with minimal network overhead [14, 20].

To verify Equation (5.18) a client can extract the set of keys R from A, and use accumulators verification functions to check whether the witness data structures received by the database correctly validates the results w.r.t. the requested data and the current state of the database that is maintained locally. By abstracting verification functions *verify*, the verification process can be represented as following:

$$\textit{verify}\left(\textit{verify}\left(R, W_Q\right), W_{\bar{T}}\right) \stackrel{?}{=} \sigma(K_D) \tag{5.19}$$

If Equation (5.19) is verified, then the client knows that the two witnesses produced by the server are correct and that Equation (5.18) is also verified. Hence R is a subset of K_D and authenticity holds. On the other hand, if Equation (5.19) is not verified, either the witnesses produced by the server are not correct or R is not a subset of K_D. In both cases, the client is able to efficiently detect a misbehavior of the server.

Completeness

Bulkopt builds proofs of completeness by demonstrating that:

$$X \cap (K_D \backslash R) \stackrel{?}{=} \varnothing \tag{5.20}$$

that is, the set of keys requested by the client X and the set of keys not returned by the server $K_D \backslash R$ share no common keys. We recall that $K_D \backslash R$ is equal to $Q \cup \bar{T}$, hence Equation (5.20) can be expressed as following:

$$X \cap (Q \cup \bar{T}) = \varnothing \tag{5.21}$$

Bulkopt proves such conditions by leveraging properties of ECR hash functions. In particular, as shown by [24], ECR hash functions can be used to efficiently express set intersections by using polynomial representations of sets. That is, an empty intersection between sets correspond to polynomials having *greatest common divisor* (*gcd*) equal to 1 (that is, informally we say that since the sets do not share any common elements, the corresponding polynomials do not have common roots).

Let us denote as $\mathcal{C}_M(s)$ a polynomial representation of a generic set M w.r.t. variable s [24, 53], and a set $P = Q \cup \bar{T}$. To prove that the *gcd* of polynomials is 1, the server must generate two polynomials $\dot{p}, \dot{x}$ such that:

$$\mathcal{C}_P \cdot \dot{p} + \mathcal{C}_X \cdot \dot{x} = 1, \tag{5.22}$$

The server sends witnesses W_P, $W_{\dot{p}}$ and $W_{\dot{x}}$ in addition to W_Q and $W_{\bar{T}}$ that were already sent to prove authenticity. The client can now exploit homomorphic properties of ECR hash functions to verify Equations (5.22). If Equation (5.22) is verified, then the client knows that the witnesses produced by the server are correct and that Equation (5.20) is also verified. Hence R includes all keys X requested by the client that are available in the server database, and completeness holds. On the other hand, if Equation (5.22) is not verified, either the witnesses produced by the server are not correct or X shares common elements with sets of keys Q or $\bar{T}$ that were not sent by the server, thus violating completeness. In both cases, the client is able to efficiently detect a misbehavior of the server.

5.2.2 Setup and key generation

Let g be a generator of the cyclic multiplicative group $\mathbb{G}$ of prime order p, $\mathbb{G}_T$ a cyclic multiplicative group of the same order and $\hat{e} : \mathbb{G} \times \mathbb{G} \rightarrow \mathbb{G}_T$ be the pairing function that satisfies the following properties: bilinearity: $\hat{e}(m^a, n^b) = \hat{e}(m, n)^{ab} \, \forall m, n \in \mathbb{G}, a, b \in \mathbb{Z}_p$; non-degeneracy: $\hat{e}(g, g) \neq 1$; computability: there exists an efficient one-way algorithm to compute $\hat{e}(m, n)$, $\forall m, n \in \mathbb{G}$.

Let h be a collision resistant hash function and $h_z(\cdot), h_g(\cdot)$ be two full domain hash functions (FDH) defined as following:

$$h_z : \{0,1\}^* \rightarrow \mathbb{Z}_p^* \tag{5.23}$$

$$h_g : \{0,1\}^* \rightarrow \mathbb{G} \tag{5.24}$$

Let us denote as $C_M(s)$ the polynomial representation of the set M generated by using as roots of the polynomial the sum opposite of the elements of the set [96] and as variable the secret key s. Polynomial $C_M(s)$ can be computed as following:

$$C_M(s) = \prod_{m \in M} (m + s) \tag{5.25}$$

Let $F_M = (f(M), f'(M))$ be the output of an extractable collision resistant (ECR) hash function [17] with secret key $(s, \alpha) \in \mathbb{Z}_p^* \times \mathbb{Z}_p^*$ and public key $[g, g^s, \ldots, g^{s^q}, g^{\alpha}, g^{\alpha s}, \ldots, g^{\alpha s^q}]$, where M denotes a set of values $m \in \mathbb{Z}_p^*$. The output of the function can be computed through two different algorithms depending on the knowledge of the secret key s. For this reason, we denote as $(f_{sk}(M), f'_{sk}(M))$ the computation of $(f(M), f'(M))$ with knowledge of the secret key and $(f_{pk}(M), f'_{pk}(M))$ the computation of $(f(M), f'(M))$ with only knowledge of the public key. We will use notation F_M to identify the output

data of the functions and thus we will distinguish if it was computed with or without knowledge of the secret key. Function $f_{sk}(M)$ can be computed by using straightforwardly the polynomial $C_M(s)$ shown in Equation (5.25) as following:

$$f_{sk}(M) = g^{C_M(s)} = g^{\prod_{i=1}^{|M|}(m_i+s)}, \tag{5.26}$$

$$f'_{sk}(M) = g^{\alpha C_M(s)} = g^{\alpha \prod_{i=1}^{|M|}(m_i+s)}, \tag{5.27}$$

Function $f_{pk}(M)$ can be computed by using the coefficients of the polynomial $C_M(s)$. That is, if we consider the set of the coefficients $\{a_i\}_{i=[1,\ldots,|M|]}$ of the polynomial $C_M(s)$ such that $C_M(s) = \sum_{i=1}^{|M|} a_i \cdot s^i$, function $f_{pk}(M)$ can be computed as following:

$$f_{pk}(M) = \prod_{i=1}^{|M|} \left(g^{s^i}\right)^{a_i} \tag{5.28}$$

$$f'_{pk}(M) = \prod_{i=1}^{|M|} \left(g^{\alpha s^i}\right)^{a_i} \tag{5.29}$$

We note that the coefficients $\{a_i\}_{i=[1,\ldots,|M|]}$ can be computed by using a polynomial interpolation algorithm [96]. Functions f_{sk} and f_{pk} have the same behavior, however the computation of f_{sk} is more efficient due to the computation of only one exponentiation in the group $\mathbb{G}$. Without knowledge of the secret key, ECR hash functions can be verified as following:

$$\hat{e}(f(M), g^{\alpha}) \stackrel{?}{=} \hat{e}(f'(M), g) \tag{5.30}$$

Otherwise, the secret key allows a more efficient verification:

$$f(M)^{\alpha} \stackrel{?}{=} f'(M) \tag{5.31}$$

Although knowledge of the secret key improves the algorithm efficiency, it allows one to cheat in the computation of the hash function. Hence, it cannot be given to parties that have advantages in breaking the security of ECR hash function.

We denote the owner's private and public keys as sk and pk and generate them as follows:

$$sk = (u, s, \alpha),\ u \stackrel{R}{\leftarrow} \mathbb{Z}_p,\ (s, \alpha) \stackrel{R}{\leftarrow} \mathbb{Z}_p^* \times \mathbb{Z}_p^* \tag{5.32}$$

$$pk = (U, [g^s, \ldots, g^{s^q}, g^{\alpha}, g^{\alpha s}, \ldots, g^{\alpha s^q}]),\ U = g^u \tag{5.33}$$

where $q \in \mathbb{N}$ must be greater than or equal to the maximum number of records involved for each insert or read operation.

5.2.3 Insert operations

The owner issues an insert operation by sending the tuple $(B_i, \sigma_i, \Gamma_i)$, where:

- $i \in \mathbb{N}$ is the *operation identifier*, that is the incremental counter maintained locally by the owner and by the server that identifies the insert operation (see Section 5.2.1);

- $B_i = \{(k, v)\}$ is the set of keys and records inserted in the database at operation i. We also denote as K_i the set of the keys $\{\mathrm{k}\}$ inserted in this operation;

- σ_i is the *bulk signature* of the set of keys K_i inserted at operation i. It is computed by the tenant as:

$$\begin{aligned}\sigma_i(K_i) &= \left([h_g(i) \cdot f_{sk}(K_i)]^u, [h_g(i) \cdot f'_{sk}(K_i)]^u\right) = \\ &= \left(\left[h_g(i) \cdot g^{\prod_{k \in K_i}(k+s)}\right]^u, \left[h_g(i) \cdot g^{\alpha \prod_{k \in K_i}(k+s)}\right]^u\right) \end{aligned} \tag{5.34}$$

  We note that the adopted algorithm is similar to that of bilinear map accumulators [83]. The original scheme would compute the signature of ECR hash function $f_{sk}(K_i)$ as $f_{sk}(K_i)^u$. Our scheme differs for the factor $h_g(i)^u$, that could be seen as a BLS signature of the operation identifier i. This slight modificaiton allows to bind the bulk signature $\sigma_i(K_i)$ to the operation identifier i in which the insert operation is executed. We leave details to the next section, where we use this design choice to verify correctness of the server answers.

- Γ_i is the set of the *record signatures* of the records B_i, computed by using a BLS aggregate signature scheme [21]:

$$\Gamma_i(B_i) = \{\gamma_i(k, v)\}_{(k,v) \in B_i} \tag{5.35}$$

$$\gamma(k, v) = h_g(k \parallel v)^u \tag{5.36}$$

  where $\parallel$ denotes the concatenation operator.

Both the owner and the server keep track of the operation identifier i locally, without exchanging it in each insert operation. After each insert operation, the server stores all records B_i, the bulk signatures σ_i and the record signatures Γ_i in the database associated to the operation identifier i.

The owner does not store any bulk signature σ_i or record Γ_i, but he maintains a cryptographic structure of constant size to keep track of the state of the database. We call it the *database signature* $\mathcal{D} = (\sigma^\star{}_{last}, F_{D_{last}})$, where *last* represent the value of the operation identifier i for the last insert

operation executed on the server, and $\sigma^{\star}_{last}$ and $F_{D_{last}}$ are the bulk signature and ECR hash function of all the keys inserted in the database.

The owner computes the bulk signature $\sigma^{\star}_{last}$ as following:

- after the first insertion ($i = 1$) he sets the initial value of the database signature as $\sigma^{\star}_1 = \sigma_1$;
- after any other insert operation ($i > 1$), the owner computes the database signature $\sigma^{\star}_i$ by computing the product of the current version of the database signature $\sigma^{\star}_{i-1}$ and the bulk signature σ_i of the last executed insert operation: $\sigma^{\star}_i = \sigma^{\star}_{i-1} \cdot \sigma_{i-1}$.

As a result, the value of the database signature $\sigma^{\star}_{last}$ is equal to the product of all the bulk signatures σ_i ever sent by the owner to the server:

$$\sigma^{\star}_{last} = \prod_{i=1}^{i=last} \sigma_i \tag{5.37}$$

The owner computes the database ECR hash function $F_{D_{last}}$ as following:

- after the first operation ($i = 1$), the database accumulator is equal to the ECR hash function of the keys included in the first bulk of data, that is $F_{D_1} = f_{sk}(K_1)$;
- after any other operation ($i > 1$), the database accumulator is computed as $F_{D_i} = F_{D_{i-1}}^{C_{K_i}(s)}$.

As a result, the value of $F_{D_{last}}$ after the last insert operation is the following:

$$F_{D_{last}} = g^{\prod_{i=1}^{last} C_{K_i}(S)} \tag{5.38}$$

5.2.4 Read operations

To execute a read operation a client must send a set of keys $X = \{k\}$ to the server. The server returns the following tuple:

$$\mathit{response}(X) := (I, A, \pi_{auth}, \pi_{comp}, \pi_{rec}) \tag{5.39}$$

where $I = \{i\}$ is the set of the operation identifiers associated to the bulks that include at least one of the keys X requested by the client; $A = \{A_i\}_{i \in I}$ is the set of the key-value records that compose the actual response to the client, grouped by the corresponding operation identifier i from which the server retrieved it; π_{auth} and π_{comp} are π_{rec} are the *keys authenticy proof*,

the *keys completeness proof* and the *records authenticity proof* used to prove keys authenticity, completeness for the returned keys and authenticity for the values associated to the keys, respectively. We decide to denote proofs for authorization and completeness with π_{auth} and π_{comp} separately for the sake of clarity. As we describe later in this section, the two proofs are strictly interrelated. We also observe that we only need to guarantee records authenticity because the correctness of the answers in a key-value database only depends on the returned keys. We recall from Section 5.2.1 that the elements of each set of the response A_i is a key-value tuple (k, v), and we denote as R_i the set of the keys included in the set A_i. In the following we describe separately the generation and the verification processes for the proofs of authenticity and completeness for keys, and the authenticy proof for records.

Keys authenticity. The keys authenticity proof is a tuple that includes the following values:

$$\pi_{auth} = (\{F_{Q_i}\}_{i \in I}, F_T, W_{\bar{T}}), \tag{5.40}$$

where $\{F_{Q_i}\}_{i \in I}$ is the set of the *bulk witnesses*, F_T is the aggregate ECR hash function of bulks that include at least one of the keys requested by the client, $W_{\bar{T}}$ is the aggregate bilinear signature of the bulks that do not include any of the keys requested by the client.

The server generates each bulk witness F_{Q_i} by computing the ECR hash function f_{pk} (see Equation (5.29)) on the set complement Q_i of R_i w.r.t. K_i, as following:

$$\begin{aligned} F_{Q_i} &= (f_{pk}(Q_i), f'_{pk}(Q_i)) = (f_{pk}(K_i \backslash R_i), f'_{pk}(K_i \backslash R_i)) = \\ &= \left(g^{C_{K_i \backslash R_i}(s)}, g^{\alpha \cdot C_{K_i \backslash R_i}(s)}\right), \; \forall i \in I \end{aligned} \tag{5.41}$$

Moreover, the server computes the aggregate bilinear signature $W_{\bar{T}}$ as the witness for bulks that do not include any keys requested by the client by aggregating the owner signatures as following:

$$W_{\bar{T}} = \prod_{i \in I} \sigma_i(K_i) = \left[\prod_{i \in I} h_g(i) g^{C_{K_i}}\right]^u \tag{5.42}$$

The client can verify authenticity of the keys $\{R_i\}$ returned by the server by using values included in the authentication proof π_{auth} and the database signature $\sigma^{\star}_{last}$ stored locally (see Equation (5.37)). The client verifies correctness of the ECR hash function F_T by using Equation (5.30). Then, the client verifies correctness that the ECR hash function F_T is built correctly

w.r.t. the aggregate bilinear signature $W_{\bar{T}}$ by using the locally maintained database signature $\sigma^{\star}_{last}$, as following:

$$\hat{e}\left(F_T, U\right) \stackrel{?}{=} \hat{e}\left(\frac{\sigma^{\star}_{last}}{W_{\bar{T}}}, g\right) \tag{5.43}$$

Finally, the client can use F_T to verify authenticity of the returned records $\{R_i\}_{i\in I}$ by using the bulk witnesses $\{F_{Q_i}\}_{i\in I}$, as following:

$$\hat{e}\left(\prod_{i\in I} h_g(i), g\right) \prod_{i\in I} \hat{e}\left(f_{pk}(R_i), F_{Q_i}\right) \stackrel{?}{=} \hat{e}\left(F_T, g\right) \tag{5.44}$$

We highlight that after this verification process the client is sure about the following guarantees: (a) F_T is a valid witness for the bilinear aggregate signature $W_{\bar{T}}$, as the probability of generating or extracting any other owner signature would break the non-extractability guarantees of aggregate bilinear signatures [21]; (b) all the returned keys $\{R_i\}_{i\in I}$ are authentic, because the server proved existence of the witnesses Q_i w.r.t. bulks aggregate hash function F_T and generating false witnesses would break extractable collision resistance (ECR) guarantees of the ECR hash function $f(\cdot)$ [24]; (c) all the operation identifiers $i \in I$ sent by the client are authentic, as generating identifiers that satisfy Equation (5.44) would break either the FDH function $h_g(\cdot)$ or the collision resistance guarantees of aggregate bilinear signatures [21].

Keys completeness. As described in Section 5.2.1, to prove completeness of the response the server must produce witnesses that prove disjunction of set of the requested keys X and the complement sets Q and $\bar{T}$. The completeness proof is a tuple that includes such witnesses, and additional values that allow the client to verify that the server generated them correctly:

$$\pi_{comp} = (F_P, F_{\dot{p}}, F_{\dot{x}},), \tag{5.45}$$

where F_P is the ECR hash function of the set union including the complement sets Q and $\bar{T}$, ($F_{\dot{q}}$ and $F_{\dot{x}}$) the witnesses that prove disjunction of the set of the requested keys X w.r.t. sets $\bar{T}$ and Q.

First, the server computes the ECR function of $F_{Q\cup\bar{T}}$ as:

$$F_P = \left(f_{pk}(Q \cup \bar{T}), f'_{pk}(Q \cup \bar{T})\right) = \left(g^{C_{Q\cup\bar{T}}(s)}, g^{\alpha \cdot C_{Q\cup\bar{T}}(s)}\right) \tag{5.46}$$

The two witnesses $F_{\dot{p}}$ and $F_{\dot{x}}$ of polynomials $\dot{x}$ and $\dot{p}$ are generated by the server to show that the *gcd* between the characteristic polynomials C_X and

$C_{Q\cup\bar{T}}$ of sets X and $Q \cup \bar{T}$ is 1, that is equivalent to prove disjunction of sets X, Q and $\bar{T}$, as shown in [24]:

$$\dot{x}, \dot{p} : C_X(s) \cdot \dot{x} + C_P(s) \cdot \dot{p} = 1 \tag{5.47}$$

$$F_{\dot{p}} = \left(g^{\dot{p}}, g^{\alpha \cdot \dot{p}}\right) \tag{5.48}$$

$$F_{\dot{x}} = \left(g^{\dot{x}}, g^{\alpha \cdot \dot{x}}\right) \tag{5.49}$$

The client verifies correctness of the ECR hash functions sent by the server $F_P, F_{\dot{q}}$ and $F_{\dot{x}}$ by using Equation (5.30). Then, he verifies whether the ECR hash functions F_P represent the set complement of R w.r.t. D by checking the value of F_P against the database accumulator $F_{D_{last}}$ (see Equation (5.38)) publicly distributed by the owner:

$$\hat{e}\left(f_{pk}(R), F_P\right) \stackrel{?}{=} \hat{e}\left(F_{D_{last}}, g\right) \tag{5.50}$$

Now that the client verified the correct generation of the witnesses F_P, he can verify disjunction of X, Q and $\bar{T}$ by testing Equations (5.47) as following:

$$\hat{e}(f_{pk}(X), F_{\dot{x}}) \cdot \hat{e}(F_P, F_{\dot{p}}) \stackrel{?}{=} \hat{e}(g, g) \tag{5.51}$$

Records authenticity. The server computes proofs of authenticity π_{rec} by aggregating all the record signatures $\gamma_{k,v} = \gamma(k, v)$ previously received by the owner for all the records returned to the client, as following:

$$\pi_{rec} = \prod_{(k,v)\in A_i, \forall A_i \in A} \gamma_{k,v} \tag{5.52}$$

The client verifies authenticity of the response A given the server integrity proof π_{int} and the owner public key U by verifying the following condition:

$$\hat{e}\left(\prod_{(k,v)\in A_i, \forall A_i \in A} h_g(k \parallel v), U\right) \stackrel{?}{=} \hat{e}(\pi_{rec}, g) \tag{5.53}$$

This concludes the description of the protocol: any client that is enabled to query the database and that knows the owner's public key pk and the state of the database $\mathcal{D}$ can verify correctness of the results by using the described verification operations. We recall that if a client knows the secret key sk, such as in symmetric settings, he can verify results correctness more efficiently by using the secret exponents u and α. More work is needed to improve efficiency of the proposal by using data structures to cache partial proofs at the server side. Although the proposal have small network and storage overhead, a

software implementation is needed to evaluate constant computation costs and compare them to related literature. We note that the current version of the protocol seems especially promising to guarantee correctness of log data, and could be integrated as-it-is with Probus to implement the periodical signatures aggregation (see Section 5.1.4).

Chapter 6

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 an external party. This thesis takes a novel approach to the definition of security solutions by integrating novel protocols, theoretical analyses and architectural solutions that allow the design, implementation and comparison of alternative approaches evaluated through analytical and experimental results. Our proposals span over confidentiality and integrity guarantees, aiming to design practical solutions that can be adopted either selectively or as a whole system to guarantee security of public cloud databases in real-world scenarios. The main original contributions are outlined below:

1. an architecture that allows distributed and concurrent operations to encrypted cloud databases and cryptographic access control enforcement;

2. a new protocol that guarantees data authenticity by means of Bloom filters and symmetric encryption schemes;

3. a protocol for database integrity that is demonstrated secure even in the presence of malicious tenant insiders;

4. an efficient integrity scheme for database characterized by bulk insert and retrieval operations.

The proposed innovative architecture guarantees confidentiality of data stored in public cloud databases. Unlike state-of-the-art approaches, our solution does not rely on an intermediate proxy that we consider a single point of failure and a bottleneck limiting availability and scalability of typical cloud database services. The encryption architecture supports concurrent

SQL operations on encrypted data, including statements modifying the database schema and the security policies, issued by heterogeneous and possibly geographically dispersed clients, and novel cryptographic enforcement strategies limit information leakage from tenant insiders at the level of in-houses databases. The proposed encryption architecture does not require modifications to the cloud database, and it is immediately applicable to existing public cloud services. Experimental results show that the performance impact of encryption on response times becomes negligible because it is masked by network latencies. Moreover, the architecture scalability is comparable to that of unencrypted cloud database services. Our results pave the way for future improvements such as new encryption algorithms to support more operations on encrypted data, the integration of private information retrieval solutions to prevent information leakage caused by access pattern analyses, novel components to leverage hybrid cloud environments.

As a second main contribution, the protocol based on Bloom filters and symmetric encryption schemes focuses on data authenticity of cloud databases. We prove security of the protocol and, by using analytical costs estimations, we shoe its efficiency in different types of architectures such as those based on completely distributed systems and those based on intermediate trusted proxies. Storage overhead is comparable to state-of-the-art literature and network overhead is consistently lower for standard database workloads. As a consequence, we are confident that this protocol can be applied to realistic scenarios. Further research can extend it to be resistant against adaptive attacks without affecting its efficiency.

The original integrity protocol for malicious tenants insiders is able to protect the cloud provider from false accusations. The protocol improves the state-of-the-art proposals by reducing online verification costs by offloading computational expensive operations to bath operations and by avoiding weaker verification strategies.

As a fourth contribution, the integrity scheme for efficient bulk operations reduces network overhead by producing small size proofs of correct computation that are based on an original combination of different signature aggregation protocols. While this thesis describes the main characteristics of the novel protocol, future work can extend it enabling cryptographic access control enforcement and forward integrity guarantees.

The architectural solutions, theoretical models and performance results reported in this thesis evidence the feasibility of protecting companies information managed by cloud database providers. We envision research efforts to improve computation capabilities, security and performance trade-offs of cryptographic protocols. Other efforts are required to bring research solutions in real production environments.

Bibliography

[1] D. Agrawal, A. El Abbadi, F. Emekci, and A. Metwally. Database management as a service: Challenges and opportunities. In *Proc. of the 25th IEEE International Conference on Data Engineering*, Mar.-Apr. 2009.

[2] Amazon RDS Pricing. Amazon Relational Database Pricing. `http://aws.amazon.com/rds/pricing`, Feb. 2016.

[3] Amazon Web Services (AWS). Amazon elastic compute cloud (amazon ec2). `http://aws.amazon.com/ec2`, Feb. 2016.

[4] A. Andreoli, L. Ferretti, M. Marchetti, and M. Colajanni. Enforcing correct behavior without trust in cloud key-value databases. In *Proc. IEEE Second Int. Conf. Cyber Security and Cloud Computing*, pages 157–164, 2015.

[5] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. Orthogonal security with cipherbase. In *Proc. Sixth Conf. Innovative Data Systems Research*, Jan. 2013.

[6] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4), 2010.

[7] M. R. Asghar, G. Russello, B. Crispo, and M. Ion. Supporting complex queries and access policies for multi-user encrypted databases. In *Proc. 2013 ACM Workshop on Cloud computing security*, Nov. 2013.

[8] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, et al. System r: relational approach to database management. *ACM Trans. Database Systems*, 1(2):97–137, 1976.

[9] M. J. Atallah, M. Blanton, N. Fazio, and K. B. Frikken. Dynamic and efficient key management for access hierarchies. *ACM Trans. Information and System Security*, 12(3), 2009.

[10] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid. Nist special publication 800-57. Technical Report 57, 2007.

[11] M. Bellare, A. Boldyreva, and A. ONeill. Deterministic and efficiently searchable encryption. In *Advances in Cryptology – Crypto*, pages 535–552. Springer, 2007.

[12] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In *Advances in Cryptology – Crypto*, pages 1–15. Springer, 1996.

[13] M. Bellare, R. Canetti, and H. Krawczyk. Pseudorandom functions revisited: The cascade construction and its concrete security. In *Proc. 37th IEEE Symp. Foundations of Computer Science*, pages 514–523, 1996.

[14] J. Benaloh and M. De Mare. One-way accumulators: A decentralized alternative to digital signatures. In *Advances in Cryptology – Crypto*, pages 274–285. Springer, 1994.

[15] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. In *Proc. of the ACM SIGMOD*, June 1995.

[16] E. Bertino, P. Samarati, and S. Jajodia. Authorizations in relational database management systems. In *Proc. First ACM Conf. Computer and communications security*, pages 130–139, 1993.

[17] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *Proc. 3rd ACM Int. Conf. Innovations in Theoretical Computer Science*, pages 326–349, 2012.

[18] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Comm. of the ACM*, 13(7), 1970.

[19] A. Boldyreva, N. Chenette, and A. O'Neill. Order-preserving encryption revisited: Improved security analysis and alternative solutions. In *Proc. Advances in Cryptology – Crypto*, Aug. 2011.

[20] D. Boneh, C. Gentry, B. Lynn, and H. Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In *Advances in cryptology – Crypto*, pages 416–432. Springer, 2003.

[21] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the weil pairing. In *Advances in Cryptology – Crypto*, pages 514–532. Springer, 2001.

[22] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4), 2004.

[23] J. Camenisch and A. Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In *Advances in Cryptology – Crypto*, pages 61–76. Springer, 2002.

[24] R. Canetti, O. Paneth, D. Papadopoulos, and N. Triandopoulos. Verifiable set operations over outsourced databases. In *Public-Key Cryptography*, pages 113–130. Springer, 2014.

[25] N. Cao, C. Wang, M. Li, K. Ren, and W. Lou. Privacy-preserving multi-keyword ranked search over encrypted cloud data. *IEEE Trans. Parallel and Distributed Systems*, 25(1):222–233, 2014.

[26] G. Cattaneo, L. Catuogno, A. D. Sorbo, and P. Persiano. The design and implementation of a transparent cryptographic file system for unix. In *Proc. FREENIX Track: 2001 USENIX Annual Technical Conference*, Apr. 2001.

[27] A. Ceselli, E. Damiani, S. D. C. D. Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati. Modeling and assessing inference exposure in encrypted databases. *ACM Trans. Information and System Security*, 8(1), 2005.

[28] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan. Private information retrieval. *Journal of the ACM*, 45(6), 1998.

[29] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. Planetlab: an overlay testbed for broad-coverage services. *ACM SIGCOMM Computer Communication Review*, 33(3), 2003.

[30] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proc. First ACM Symp. Cloud Computing*, 2010.

[31] R. M. Corless, G. H. Gonnet, D. E. Hare, D. J. Jeffrey, and D. E. Knuth. On the lambertw function. *Advances in Computational mathematics*, 5(1):329–359, 1996.

[32] J. Crampton, K. Martin, and P. Wild. On key assignment for hierarchical access control. In *Proc. 19th IEEE Computer Security Foundations Workshop*, Jul. 2006.

[33] S. A. Crosby and D. S. Wallach. Efficient data structures for tamper-evident logging. In *Usenix Security Symp.*, pages 317–334, 2009.

[34] S. A. Crosby and D. S. Wallach. Authenticated dictionaries: Real-world costs and trade-offs. *ACM Trans. Information and System Security*, 14(2):17, 2011.

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

[36] E. Damiani, S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Key management for multi-user encrypted databases. In *Proc. ACM Workshop Storage Security and Survivability*, Nov. 2005.

[37] E. Damiani, S. D. C. di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Metadata management in outsourced encrypted databases. In *Secure Data Management*. Springer, 2005.

[38] E. Damiani, S. D. C. Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati. Balancing confidentiality and efficiency in untrusted relational dbmss. In *Proc. 10th ACM Conf. Computer and communications security*, October 2003.

[39] J. L. Dautrich Jr and C. V. Ravishankar. Compromising privacy in precise query protocols. In *Proc. 16th ACM Int. Conf. Extending Database Technology*, Mar. 2013.

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

[41] C. Dong, L. Chen, and Z. Wen. When private set intersection meets big data: an efficient and scalable protocol. In *Proc. 2013 ACM SIGSAC Conf. Computer & communications security*, pages 789–800. ACM, 2013.

[42] M. Duckham and L. Kulik. A formal model of obfuscation and negotiation for location privacy. In *Proc. Third Int. Conf. Pervasive Computing.* Springer, March 2005.

[43] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Advances in cryptology*, pages 10–18. Springer, 1985.

[44] EnterpriseDB. Postgres Plus Cloud Database. `http://enterprisedb.com/cloud-database`, Feb. 2016.

[45] EnterpriseDB. Postgres Plus Cloud Database Pricing. `http://www.enterprisedb.com/cloud-database/pricing-amazon`, Feb. 2016.

[46] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. Sporc: group collaboration using untrusted cloud resources. In *Proc. Ninth USENIX Conf. Operating Systems Design and Implementation*, Oct. 2010.

[47] L. Ferretti, M. Colajanni, and M. Marchetti. Supporting security and consistency for cloud database. In *Proc. of the 4th International Symposium on Cyberspace Safety and Security (CSS 2012)*, Melbourne, Australia, December 2012. Springer.

[48] L. Ferretti, M. Colajanni, and M. Marchetti. Access control enforcement on query-aware encrypted cloud databases. In *Proc. IEEE Fifth Int. Conf. Cloud Computing Technology and Science*, volume 1, pages 717–722, Dec 2013.

[49] L. Ferretti, M. Colajanni, and M. Marchetti. Distributed, concurrent, and independent access to encrypted cloud databases. *IEEE Transactions on Parallel and Distributed Systems*, 25(2):437–446, 2014.

[50] L. Ferretti, F. Pierazzi, M. Colajanni, and M. Marchetti. Security and confidentiality solutions for public cloud database services. In *Secureware 2013, The Seventh Int. Conf. on Emerging Security Information, Systems and Technologies*, pages 36–42, Aug 2013.

[51] L. Ferretti, F. Pierazzi, M. Colajanni, and M. Marchetti. Performance and cost evaluation of an adaptive encryption architecture for cloud databases. *IEEE Trans. Cloud Computing*, 2(2):143–155, April 2014.

[52] L. Ferretti, F. Pierazzi, M. Colajanni, and M. Marchetti. Scalable architecture for multi-user encrypted sql operations on cloud database services. *IEEE Trans. Cloud Computing*, 2(4):448–458, Oct. 2014.

[53] M. J. Freedman, K. Nissim, and B. Pinkas. Efficient private matching and set intersection. In *Advances in cryptology – Crypto*, pages 1–19. Springer, 2004.

[54] V. Ganapathy, D. Thomas, T. Feder, H. Garcia-Molina, and R. Motwani. Distributing data for secure database services. In *Proc. Fourth ACM Int. Workshop Privacy and Anonymity in the Information Society*, Mar. 2011.

[55] Gartner. Cloud Security Gateways. `http://www.gartner.com/it-glossary/cloud-security-gateways`, Feb. 2016.

[56] E.-J. Goh. Secure indexes. *IACR Cryptology ePrint Archive*, 2003:216, 2003.

[57] O. Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge university press, 2004.

[58] M. T. Goodrich, R. Tamassia, and J. Hasić. An efficient dynamic and distributed cryptographic accumulator. In *Information Security*, pages 372–388. Springer, 2002.

[59] M. T. Goodrich, R. Tamassia, and A. Schwerin. Implementation of an authenticated dictionary with skip lists and commutative hashing. In *Proc. Second IEEE Information Survivability Conference & Exposition*, volume 2, pages 68–82, 2001.

[60] V. Goyal, O. Pandey, A. Sahai, and B. Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *Proc. 13th ACM Conf. Computer and communications security*, pages 89–98, 2006.

[61] C. Guan, K. Ren, F. Zhang, F. Kerschbaum, and J. Yu. Symmetric-key based proofs of retrievability supporting public verification. In *Computer Security – Esorics 2015*, pages 203–223. Springer, 2015.

[62] H. Hacigümüş, B. Iyer, C. Li, and S. Mehrotra. Executing sql over encrypted data in the database-service-provider model. In *Proc. 2002 ACM SIGMOD Int. Conf. Management of data*, Jun. 2002.

[63] H. Hacigümüş, B. Iyer, and S. Mehrotra. Providing database as a service. In *Proc. 18th IEEE Int. Conf. Data Engineering*, Feb. 2002.

[64] M. Hadavi, E. Damiani, R. Jalili, S. Cimato, and Z. Ganjei. AS5: A secure searchable secret sharing scheme for privacy preserving database outsourcing. In *Proc. Fifth Int. Workshop Autonomous and Spontaneous Security*, Sep. 2013.

[65] J. Hong, T. Wen, Q. Gu, and G. Sheng. Query integrity verification based-on mac chain in cloud storage. In *Proc. 13th IEEE/ACIS Int. Conf. Computer and Information Science*, June 2014.

[66] J. Katz and Y. Lindell. *Introduction to modern cryptography.* CRC Press, 2014.

[67] J. Katz and M. Yung. Unforgeable encryption and chosen ciphertext secure modes of operation. In *Fast Software Encryption*, pages 284–299. Springer, 2001.

[68] A. Kumar, P. Lafourcade, C. Lauradoux, et al. Performances of cryptographic accumulators. 2014.

[69] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (sundr). In *Proc. of the 6th USENIX conference on Opearting Systems Design and Implementation*, October 2004.

[70] J. Li and E. Omiecinski. Efficiency and security trade-off in supporting range queries on encrypted databases. In *Proc. of the 19th annual IFIP WG 11.3 working conference on Data and Applications Security*. Springer, August 2005.

[71] D. Ma and G. Tsudik. A new approach to secure logging. *ACM Trans. Storage*, 5(1), 2009.

[72] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. *ACM Trans. Computer Systems*, 29(4), 2011.

[73] U. T. Mattsson. A practical implementation of transparent encryption and separation of duties in enterprise databases: protection against external

and internal attacks on databases. In *Proc. Seventh IEEE Int. Conf. E-Commerce Technology*, 2005.

[74] C. Mavroforakis, N. Chenette, A. O'Neill, G. Kollios, and R. Canetti. Modular order-preserving encryption, revisited. In *Proc. 2015 ACM SIGMOD Int. Conf. on Management of Data*, pages 763–777, 2015.

[75] Microsoft. Windows Azure SQL Database Web & Business Pricing. `http://www.windowsazure.com/en-us/pricing/details/sql-database/#service-webandbusiness`, Feb. 2016.

[76] Microsoft corporation. Windows azure. `http://www.windowsazure.com`, Feb. 2016.

[77] M. Mitzenmacher. Compressed bloom filters. *IEEE/ACM Trans. Networking*, 10(5):604–612, 2002.

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

[79] E. Mykletun, M. Narasimha, and G. Tsudik. Authentication and integrity in outsourced databases. *ACM Trans. Storage*, 2(2):107–138, May 2006.

[80] E. Mykletun and G. Tsudik. Aggregation queries in the database-as-a-service model. In *Proc. of the 20th annual IFIP WG 11.3 working conference on Data and Applications Security.* Springer, July-August 2006.

[81] C. Namprempre, P. Rogaway, and T. Shrimpton. Reconsidering generic composition. In *Advances in Cryptology – Eurocrypt*, pages 257–274. Springer, 2014.

[82] M. Narasimha and G. Tsudik. Dsac: Integrity for outsourced databases with signature aggregation and chaining. In *Proc. 14th ACM Int. Conf. Information and Knowledge Management*, pages 235–236, 2005.

[83] L. Nguyen. Accumulators from bilinear pairings and applications. *Topics in Cryptology – CT-RSA 2005*, pages 275–292.

[84] Nist. Digital signature standard. Technical report, Jul. 2013.

[85] NIST. Cryptographic toolkit. `http://csrc.nist.gov/groups/ST/toolkit/`, Feb. 2016.

[86] K. Nyberg. Fast accumulated hashing. In *Fast Software Encryption*, pages 83–87. Springer, 1996.

[87] Oracle corporation. Oracle advanced security. `http://www.oracle.com/technetwork/database/options/advanced-security`, Feb. 2016.

[88] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proc. of the Advances in Cryptology – Eurocrypt.* Springer, May 1999.

[89] O. Pandey and Y. Rouselakis. Property preserving symmetric encryption. In *Advances in cryptology – Eurocrypt*, pages 375–391. Springer, 2012.

[90] H. Pang, J. Zhang, and K. Mouratidis. Scalable verification for outsourced dynamic databases. *Proc. VLDB Endowment*, 2(1):802–813, 2009.

[91] B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *Proc. 34th IEEE Symp. Security and Privacy*, pages 238–252, 2013.

[92] K. G. Paterson and A. Yau. Padding oracle attacks on the iso cbc mode encryption standard. In *Topics in Cryptology – CT-RSA*, pages 305–323. Springer, 2004.

[93] S. Pearson, M. C. Mont, L. Chen, and A. Reed. End-to-end policy-based encryption and management of data in the cloud. In *Proc. Third IEEE Int. Conf. Cloud Computing Technology and Science*, Nov.-Dec. 2011.

[94] R. A. Popa, J. R. Lorch, D. Molnar, H. J. Wang, and L. Zhuang. Enabling security in cloud storage slas with cloudproof. In *Proc. Usenix Annual Technical Conf.*, 2011.

[95] 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.

[96] F. P. Preparata and D. V. Sarwate. Computational complexity of fourier transforms over finite fields. *Mathematics of Computation*, 31(139):740–751, 1977.

[97] Python Software Foundation. Python database API specification v2.0. `http://www.python.org/dev/peps/pep-0249`, Feb. 2016.

[98] L. M. Rodriguez-Henriquez and D. Chakraborty. Rdas: A symmetric key scheme for authenticated query processing in outsourced databases. In *Security and Trust Management*, pages 115–130. Springer, 2013.

[99] P. Samarati and S. De Capitani di Vimercati. Access control: Policies, models, and mechanisms. In *Foundations of Security Analysis and Design*. Springer, 2001.

[100] R. S. Sandhu and P. Samarati. Access control: principle and practice. *IEEE Comm. Magazine*, 32(9):40–48, 1994.

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

[102] B. Schneier and J. Kelsey. Secure audit logs to support computer forensics. *ACM Trans. Information and System Security*, 2(2):159–176, May 1999.

[103] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11), 1979.

[104] D. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Proc. 2000 IEEE Symp. on Security and Privacy.*, May 2000.

[105] SQLAlchemy authors and contributors. Sqlalchemy - the database toolkit for python. `www.sqlalchemy.org`, Feb. 2016.

[106] R. Tamassia. Authenticated data structures. In *Algorithms-ESA 2003*, pages 2–5. Springer, 2003.

[107] Y. Tang, T. Wang, L. Liu, X. Hu, and J. Jang. Lightweight authentication of freshness in outsourced key-value stores. In *Proc. 30th ACM Conf. Computer Security Applications*, pages 176–185, 2014.

[108] TPC-C. Transaction processing performance council. `http://www.tpc.org`, Feb. 2016.

[109] H.-L. Truong and S. Dustdar. Composable cost estimation and monitoring for computational applications in cloud computing environments. *Procedia Computer Science*, 1(1):2175–2184, 2010.

[110] S. Tu, M. Kaashoek, S. Madden, and N. Zeldovich. Processing analytical queries over encrypted data. In *Proc. 39th Int. Conf. Very Large Data Bases*, Aug. 2013.

[111] Verizon. IP Latency Statistics. `http://www.verizonbusiness.com/about/network/latency`, Feb. 2016.

[112] G. Wang, Q. Liu, and J. Wu. Hierarchical attribute-based encryption for fine-grained access control in cloud storage services. In *Proc. 17th ACM Conf. Computer and Communications Security*, Oct. 2010.

[113] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. Fifth USENIX Conf. Operating Systems Design and Implementation*, Dec. 2002.

[114] M. Yabandeh and D. Gómez Ferro. A critique of snapshot isolation. In *Proc. of the 7th ACM european conference on Computer Systems*, April 2012.

[115] S. Yu, C. Wang, K. Ren, and W. Lou. Achieving secure, scalable, and fine-grained data access control in cloud computing. In *Proc. 2010 IEEE Conf. Computer Communications*, Mar. 2010.