Scalable architecture for multi-user encrypted SQL operations on cloud database services

Luca Ferretti, Fabio Pierazzi, Michele Colajanni, and Mirco Marchetti

Abstract—The success of the cloud database paradigm is strictly related to strong guarantees in terms of service availability, scalability and security, but also of data confidentiality. Any cloud provider assures the security and availability of its platform, while the implementation of scalable solutions to guarantee confidentiality of the information stored in cloud databases is an open problem left to the tenant. Existing solutions address some preliminary issues through SQL operations on encrypted data. We propose the first complete architecture that combines data encryption, key management, authentication and authorization solutions, and that addresses the issues related to typical threat scenarios for cloud database services. Formal models describe the proposed solutions for enforcing access control and for guaranteeing confidentiality of data and metadata. Experimental evaluations based on standard benchmarks and real Internet scenarios show that the proposed architecture satisfies also scalability and performance requirements.

Index Terms—Database, Confidentiality, Encryption, Access Control

1 Introduction

The diffusion of cloud database services is being hindered by the perception of confidentiality risks when we store our information in cloud infrastructures [1]. Cryptographic solutions address this issue in the context of file storage when there is no need to perform computations over encrypted data. We aim, instead, to guarantee data confidentiality and data isolation for cloud databases that represent an open research area. There are three main related issues behind these two problems: execution of SQL operators over encrypted data; enforcement of access control mechanisms through selective encryption strategies; design of architectures not penalizing the performance and scalability that are typical of cloud-based services [2]. Existing proposals offer partial and separate solutions to data confidentiality and isolation. For example, architectures supporting SQL operations on encrypted data leave access control to the cloud provider [3] or enforce it through an intermediate trusted server [4]. Other proposed architectures solve the problem of access control without the intervention of the cloud provider, but they do not allow execution of SQL operations on encrypted data (e.g., [5]–[7]).

We propose the first architecture, called *Multi-User relaTional Encrypted DataBase* (MuteDB), that guarantees data confidentiality by executing SQL operations on encrypted data and by enforcing access control policies through selective encryption methods. By combining these two approaches MuteDB is the only solution ensuring confidentiality of data stored in the cloud even in the worst threat scenario where legitimate database users collude with cloud provider employees. This result is achieved through an innovative model that translates access control policies related to a plaintext database into selective encryption strategies that are applied to the corresponding encrypted database. Our solution works even in dynamic scenarios, in which users and access control policies change over time, without the need to renew and redistribute user credentials. The proposed architecture is specifically designed for cloud database scenarios where multiple users can access the cloud database through the Internet possibly from different geographical areas. Special attention in the architectural design is devoted to guarantee the same availability and scalability of a plaintext cloud database. For this reason, MuteDB does not rely on any intermediate trusted server that could become a system bottleneck and a single point of failure. Moreover, it adopts innovative solutions for guaranteeing efficient retrieval of database metadata that are stored in an encrypted form in the cloud database.

We can consider MuteDB as the first architecture that allows enterprises to leverage cloud database services while achieving the same confidentiality guarantees of a traditional in-house database and the same scalability of a cloud database service.

The performance and scalability of MuteDB are evaluated through a prototype that is subject to different query workloads based on standard (TPC-C) and recently proposed (YCSB) database benchmarks. We highlight that, as a further contribution, this paper reports the first performance evaluation studies related to encrypted cloud database services in real distributed environments where the clients are geographically distributed over the PlanetLab platform [8]. Experimental results show that MuteDB does not affect the scalability of the original cloud service, and its performance for geographically distributed clients are comparable to those of unencrypted cloud database services.

The remaining part of this paper is organized as following. Section 2 describes the main threats affecting

• *University of Modena and Reggio Emilia. E-mail: {luca.ferretti,fabio.pierazzi,michele.colajanni,mirco.marchetti}@unimore.it*

a cloud database service and that are addressed by our solution. Section 3 outlines the main features of the MuteDB architecture. Section 4 describes the novel encryption and access control enforcement schemes for guaranteeing data isolation. Section 5 describes the methods for managing encrypted metadata that are stored in the cloud database. Section 6 details the fundamental operations of MuteDB for SQL query execution and privilege management. Section 7 presents the experimental results obtained in a geographically distributed environment. Section 8 discusses related work. Section 9 summarizes the main conclusions and future work.

2 THREAT MODEL

We propose an architecture guaranteeing confidentiality and isolation of data stored in cloud database infrastructures that are subject to two types of threats: those related to specific roles, and those deriving by the collusion of these roles. The literature focuses on the former threats, while our proposal aims to respond to both classes.

Typical threat models in literature identify the possible issues related to four roles: the tenant Database Administrator (DBA), the tenant database users, the cloud provider employees, and people external to tenant's and provider's organizations. We describe our assumptions based on the four roles and then we consider collusion.

The *DBA* is the only role that has access to all tenant data. He is in charge of installing and configuring the database, implementing the *access control policies* and managing the *users credentials*. As in related literature, our threat model assumes 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 paper.

External attackers 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.

The *cloud insiders* are employees of the cloud provider that have access to the cloud infrastructure hosting the database service of the tenant organization. Their behavior is *honest but curious* [9], that is, they may be interested in accessing tenant data, but they do not modify or delete them. This assumption is considered realistic in all related literature [3]–[5], [10] and the motivation should be clear. 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 of its customers.

Tenant insiders refer to database 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.

Guaranteeing data confidentiality in the cloud against external attackers, cloud insiders, and tenant insiders under the assumption that they do not collude can be achieved through some combinations of existing solutions. For example, best practices in the field of authentication and secure communication protocols hinder external attacks. Recent SQL-aware cryptographic strategies [3], [4] allow a tenant to store encrypted data thus preventing cloud insiders and external attackers from reading tenant data. Standard database access control mechanisms, such as privilege GRANTS and reference monitors [11], 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, but they do not work as well for cloud database services because they do not consider the main threats posed by a collusion between a cloud and a tenant insiders when data are encrypted through a global master key.

In a cloud database scenario, 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 decryption key(s) to a cloud insider. The latter, that has access to all the encrypted data and can bypass the access control policies enforced at the cloud side, can violate the confidentiality of the entire database by means of the key(s) received by the tenant insider. A second collusion scenario may happen if a cloud insider delivers 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 all encrypted data, thus violating the tenant access control policies.

Let us anticipate a summary of the design choices and novel solutions that allow MuteDB to protect data against *external attackers*, *cloud insiders* and *tenant insiders*, and against collusion between these roles. External attackers that eavesdrop network traffic cannot access any plaintext information because SQL operations issued to the cloud database are protected by using standard encryption protocols (e.g., SSL). Cloud insiders and external attackers that have breached the cloud servers cannot access confidential information, because MuteDB encrypts tenant data with SQL-aware encryption algorithms and the cloud provider never obtains the decryption keys. Tenant insiders cannot perform privilege escalation attacks on the encrypted database thanks to a novel scheme that translates and enforces the database access control policies defined by the tenant DBA on the plaintext database to the encrypted one. Even in the worst case of a collusion between tenant and cloud insiders, the proposed solution limits the data leakage to the amount of information that is accessible to

the colluding tenant insider, because MuteDB does not delegate the enforcement of access control policies to the cloud provider.

3 Architecture

In this section we outline the main solutions adopted in the MuteDB architecture that guarantee data isolation and confidentiality on any relational cloud database service rented by a tenant organization. The solutions and operational details are described in the following three sections.

In Fig. 1 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.

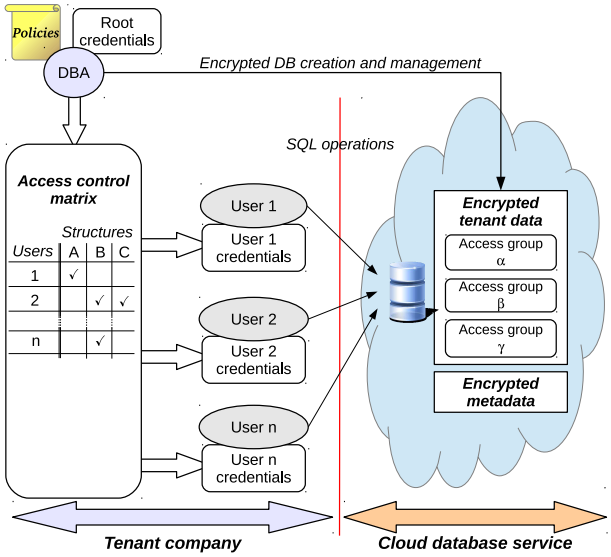


Fig. 1: Architecture of MuteDB.

Unlike existing proposals, MuteDB does not use any trusted intermediate proxy [4] and key distribution server [7], nor it stores large amounts of cryptographic information and metadata in the client machines [12]. 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., [11], [13]) does not propose how to extend these policies on SQL-aware encrypted cloud databases. Hence, to the best of our knowledge, this paper 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], [14], [15]. 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. 1 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. 1 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 4.3.

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., [16], [17]). This

choice avoids the generation and distribution of new credentials even if the access control policies change (Section 6). 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. 1, 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. 1, 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. The detailed solutions of MuteDB for access control through encryption and for metadata management are described in Section 4 and Section 5, respectively.

4 Access control and data encryption

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 4.1) and encrypted (Section 4.2) databases, we describe how MuteDB transforms an access control matrix for the plaintext model to a matrix suitable for the encrypted database (Section 4.3), and how it generates user credentials (Section 4.4).

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 [5].

4.1 Plaintext database model

We model the plaintext database through the following triple:

$$\mathbb{P} := (\mathcal{S}, \succ, \mathcal{R}) \tag{1}$$

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 $x \succ\!\!\!\cdot\, y$, 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 $t \succ\!\!\!\cdot\, c$). 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. 2 that represents the model of a plaintext database schema (s_1) 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 '$db.t_1.c_1$'.

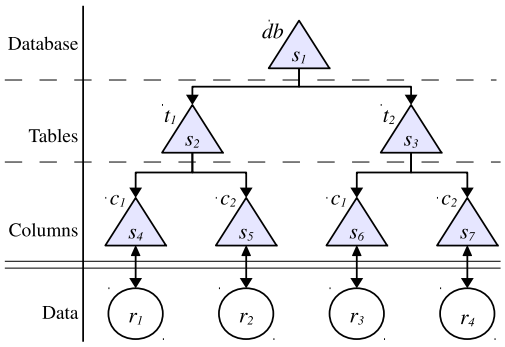


Fig. 2: The poset representing a plaintext database.

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 ($s \succ r$); 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 [15]. 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.

4.2 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 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 standard deterministic AES function 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 (1)):

$$\mathbb{E} := (\mathcal{S}, \succ, \mathcal{R}, \mathcal{G}, \mathcal{V}, \Phi, \mathcal{K}, \mathcal{E}, \mathcal{T}, \theta, \Gamma) \tag{2}$$

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:

$$\theta : \mathcal{V} \times \mathcal{G} \times \mathcal{T} \longmapsto \mathcal{V} \tag{3}$$

$$\forall (a, b) \in \mathcal{G} \times \mathcal{G} : a \succ b \Rightarrow \exists! \, t : \theta(v_a, b, t) = v_b \tag{4}$$

  An implementation example of derivation function is proposed in [16].
- 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:

$$\Gamma : \mathcal{V} \times \mathcal{G} \times \Phi^n \longmapsto \mathcal{S}^n \times \mathcal{K}^n \tag{5}$$

$$\forall (a, \Phi_B), a \in \mathcal{G}, B := \{b \in \mathcal{E} : b \prec a\}$$
$$\Rightarrow \Gamma(v_a, a, \Phi_B) = \{(c, k_b) : b \in B, c \in \mathcal{S}, c \succ b\} \tag{6}$$

  An implementation case using the AES algorithm and metadata is proposed in Section 5.

Let us explain the proposed model by referring to the example of the encrypted database shown in Fig. 3, 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

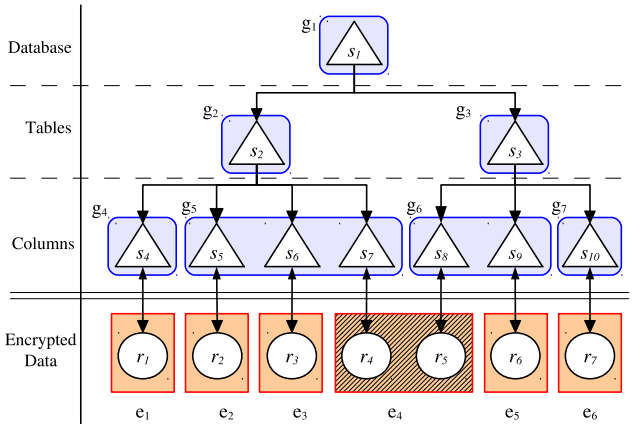


Fig. 3: Scheme of the structure of an encrypted database.

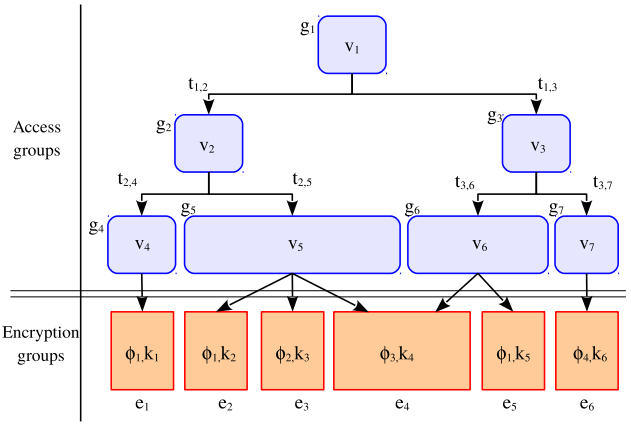


Fig. 4: Scheme of the access and encryption groups of an encrypted database.

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. 4 refers to the same encrypted database represented in Fig. 3, 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 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 $g_1 \succcurlyeq g_2$ is associated with the token $t_{1,2}$.

4.3 Access control enforcement strategy

For the sake of clarity, from now on we refer to the proposed models of plaintext (1) and encrypted (2) 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.

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 4.1) 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. 2, 3 and 4. If a user u is authorized for the database structure s_2 of the plaintext database (as in Fig. 2) 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) 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. 4). 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 because it is a descendant of g_2. Hence, u can compute v_5 through the Equation (4): $v_5 = \theta(v_2, g_5, t_{2,5})$. After having computed v_5, u can employ the Equation (6) to compute the set of keys associated with the encryp-

tion 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.

4.4 Generation of credentials

We now describe the credentials distribution scheme $\mathbb{D}$ used to generate and deliver secret keys to tenant database users. The DBA client applies this scheme to enforce the access rules included in the access control matrix $\mathcal{A}_E$:

$$\mathbb{D} := (\mathcal{U}, \mathcal{G}, \mathcal{A}_E, \mathcal{V}, \mathcal{T}, \theta) \quad (7)$$

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 4.2, θ is the derivation function defined in Equation (4).

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 $\mathcal{T}_u$.

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 [4], [18] or require database users to maintain them locally [12]. 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 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. Unlike the proposals of the same authors in which all users are provided with the same master encryption key [3], 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.

StructureID	DBToken
$MAC(v_1, \text{'db'})$	$\{AES(v_1, \text{'db.t}_1\text{'}), t_{1,2}\}$, $\{AES(v_1, \text{'db.t}_2\text{'}), t_{1,3}\}$
$MAC(v_2, \text{'db.t}_1\text{'})$	$\{AES(v_2, \text{'db.t}_1\text{.c}_1\text{'}), t_{2,4}\}$, $\{AES(v_2, \text{'db.t}_1\text{.c}_2\text{'}), t_{2,5}\}$
$MAC(v_3, \text{'db.t}_2\text{'})$	$\{AES(v_3, \text{'db.t}_2\text{.c}_1\text{'}), t_{3,6}\}$, $\{AES(v_3, \text{'db.t}_2\text{.c}_2\text{'}), t_{3,7}\}$

TABLE 1: Database tokens table.

ColumnID	Enc
$MAC(v_4, \text{'db.t}_1\text{.c}_1\text{'})$	$\{AES(v_4, \text{'}\phi_1\text{'}), AES(v_4, k_1)\}$
$MAC(v_5, \text{'db.t}_1\text{.c}_2\text{'})$	$\{AES(v_5, \text{'}\phi_1\text{'}), AES(v_5, k_2)\}$, $\{AES(v_5, \text{'}\phi_2\text{'}), AES(v_5, k_3)\}$, $\{AES(v_5, \text{'}\phi_3\text{'}), AES(v_5, k_4)\}$
$MAC(v_6, \text{'db.t}_2\text{.c}_1\text{'})$	$\{AES(v_6, \text{'}\phi_1\text{'}), AES(v_6, k_5)\}$, $\{AES(v_6, \text{'}\phi_3\text{'}), AES(v_6, k_4)\}$
$MAC(v_7, \text{'db.t}_2\text{.c}_2\text{'})$	$\{AES(v_7, \text{'}\phi_4\text{'}), AES(v_7, k_6)\}$

TABLE 2: Database encryption table.

UserID	UToken
$MAC(v_{u1}, \text{'u1'})$	$\{AES(v_{u1}, \text{'db.t}_1\text{'}), t_{u1,v_2}\}$, $\{AES(v_{u1}, \text{'db.t}_2\text{'}), t_{u1,v_3}\}$
$MAC(v_{u2}, \text{'u2'})$	$\{AES(v_{u2}, \text{'db.t}_2\text{'}), t_{u2,v_3}\}$
$MAC(v_{u3}, \text{'u3'})$	$\{AES(v_{u3}, \text{'db'}), t_{u3,v_1}\}$
$MAC(v_{u4}, \text{'u4'})$	$\{AES(v_{u4}, \text{'db.t}_2\text{.c}_2\text{'}), t_{u4,v_7}\}$

TABLE 3: Users tokens table.

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 novel metadata management strategy proposed in this paper 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 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 4.2. The *users tokens table* stores all information related to the users credentials (see Section 4.4). 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 the AES 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 1 is an example of database tokens table associated with the encrypted database represented in Fig. 3. 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 $AES(v_1, \text{'}db.t_2\text{'})$ 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 the Equation (4).

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 AES 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 through the AES algorithm and the derivation key of c. Table 2 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). 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 AES 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. 2. 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 4.1 that users are implicitly authorized to access all the descendant structures and resources. Table 3 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 AES, MAC functions and 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.

6 Operations

In this section we describe how database operations are performed by the MuteDB clients. By referring to the same encrypted database, users and access control policies presented in Section 5, we consider the three most important use cases from the point of view of this paper: translation of a plaintext SQL operation into an encrypted operation; provisioning a new user with access privileges; revocation of existing privileges.

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 [20]. 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 =* $MAC(v_{u1}, \text{'}u1\text{'})$. 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=$MAC(v_2, \text{'}db.t_1\text{'})$. 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=*$MAC(v_4, \text{'}db.t_1.c_1\text{'})$ *OR ColumnID=*$MAC(v_5, \text{'}db.t_1.c_2\text{'})$. 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 algorithm 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 [20]. 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 [3].

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., [5]).

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 and 4). 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.

7 Experimental evaluation

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 [21] executed by concurrent clients that are geographically distributed over ten different countries of the Planetlab platform [8]. The experimental results on a real setting represent an additional contribution of this paper.

7.1 Experimental testbed

The MuteDB prototype is implemented in Python. It 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, and it can be ported to any relational DBMS and to any commercial cloud database service.

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. For example, *equality check* is supported by deterministic ciphers (DET) that preserve data equality [22]–[24]; *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 [20] that is homomorphic with respect to the sum operator. Other operations, such as string match and multiplication, are feasible through Search algorithms [25], [26] and RSA, respectively. The database columns not requiring any computation can be encrypted through standard algorithms such as AES [23] or Blowfish [24] with random initialization vectors. It is important to observe that the MuteDB architecture is modular so it can integrate other encryption algorithms.

The experimental testbed is composed by a PostgreSQL 9.3 database server located in Europe and by up to 80 clients geographically distributed over ten countries of Planetlab Europe [8]. 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. 5. The RTT 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:

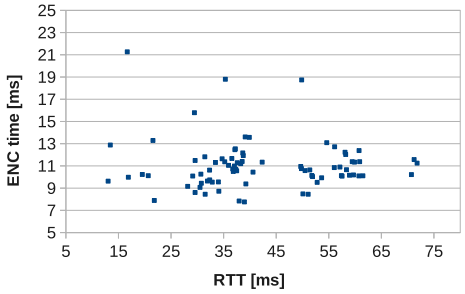


Fig. 5: Distribution of the RTTs and ENC times for the 80 Planetlab clients.

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 4: YCSB workloads.

- *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.

We also perform several experiments based on YCSB [21], 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 4). 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.
- *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

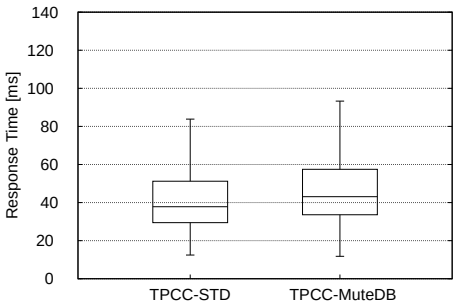


Fig. 6: Response times for the SQL operations in the TPC-C configurations.

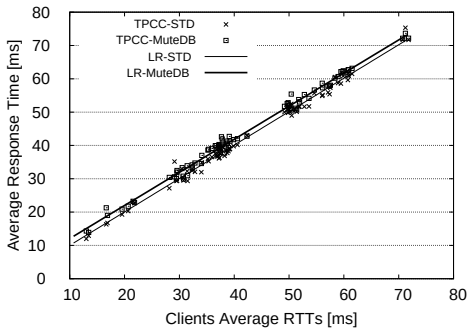


Fig. 7: Average response time of the most frequent TPC-C SELECT operation for different clients.

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.

7.2 Performance evaluation

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. 6 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: 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. 7 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 RTT. 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

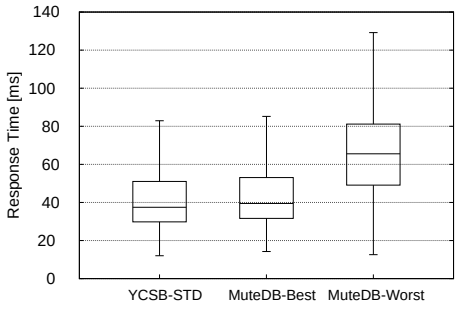


Fig. 8: Response times for the SQL operations in the YCSB configurations.

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 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 confirm the same results and are not included in this paper due to space limitations.

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 4), and we analyze the distribution of the response times considering all the SQL operations composing YCSB. Fig. 8 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

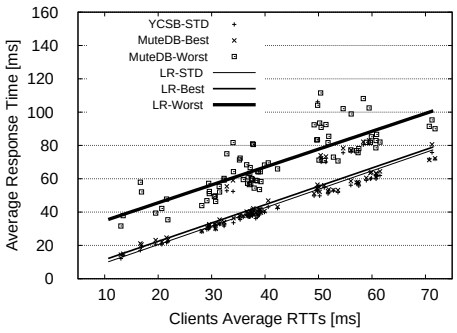


Fig. 9: Average response time of YCSB SELECT operation for different clients.

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. 9 where the average response time of each client is plotted as a function of its average RTT. Similarly to Fig. 7 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 dispersion 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.

7.3 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. 10.

The *TPC-C throughput* denotes the number of TPC-C transactions committed per minute on the database server. In Fig. 10a, 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 this paper, 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. 10a we can appreciate that the TPCC-STD and TPCC-MuteDB throughputs both 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.

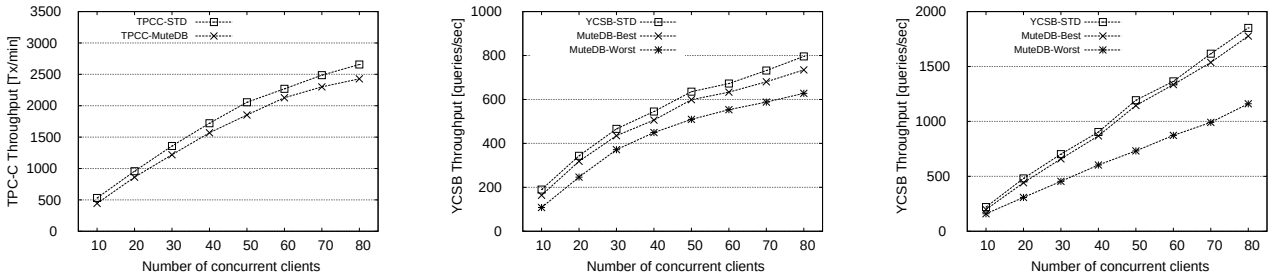
Similar conclusions can be drawn by analyzing the results obtained by using the Update Heavy workload (A) of YCSB reported in Fig. 10b. The X-axis represents the number of concurrent clients, and the Y-axis reports the *YCSB throughput* as the total number of SQL operations executed per second on the 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. 10c 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 in the MuteDB architecture are efficient and do not affect the scalability of cloud database services.

8 Related work

Many confidentiality solutions exist for cloud storage services [27], [28] but they do not support the execution of SQL operations on encrypted data. Other techniques guaranteeing data confidentiality through encryption managed by the cloud provider, standard database methods [29] and policy enforcement strategies [30] are not acceptable because modern threat models assume that a cloud provider employee could access tenant data. MuteDB is more related to proposals performing operations on encrypted databases [3], [4], [7], [10], and



(a) TPC-C Workload (b) YCSB Workload A (Update Heavy) (c) YCSB Workload C (Read Only)

Fig. 10: Throughput for increasing number of concurrent clients for different workloads and database configurations.

enforcing access control at the encryption level [5], [6], [31], although the following reasons differentiate our architecture from the state of the art.

The solutions in [4] and [10] require that clients issue SQL queries through one trusted proxy managing all encryption and decryption operations, and forwarding them to the encrypted cloud database. We avoid a similar approach because any architecture relying on one intermediate server limits the availability and elasticity of a cloud database service. Moreover, 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.

The proposals in [3], [7] avoid the need of an intermediate proxy server. The architecture in [7] adopts an access control mechanism that is based on a reference monitor within the cloud infrastructure and on a trusted authentication server. The solution proposed in [3] by the same authors solves client concurrency management problems for write/read accesses to encrypted data in the cloud, but it does not guarantee data isolation and confidentiality against the collusion threats considered in this paper. Indeed, all tenant users are provided with the same master key, and access control policies are implemented by leveraging the standard database access control mechanisms at the cloud provider side. 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.

Some interesting solutions for enforcing access control policies on outsourced information are proposed in [5], [6], [31], [32]. The encryption schemes in [31] 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 [5] 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 proposal of hierarchical attribute-based encryption schemes [6] to enforce access control policies may be applied to a cloud storage service, but not to a cloud database service because they do not support SQL operations. As theoretically introduced in [33], our proposal combines for the first time standard access control models of relational databases with the execution of SQL operations on encrypted data stored in the cloud. As a further original contribution, we remark that this paper includes for the first time performance and scalability evaluations obtained in a real environment and for realistic workloads executed by clients that are dispersed over different geographical areas.

9 Conclusions

In this paper we propose MuteDB, a novel architecture for cloud database services that guarantees for the first time data confidentiality through SQL-aware encryption algorithms and data isolation through access control enforcement based on encryption and key derivation techniques. These solutions allow MuteDB to address threat issues that are relevant for cloud services including risks of information leakage due to collusions between cloud provider employees and tenant users. The most important solutions are described through formal models, while the feasibility, performance and scalability of the proposed architecture are demonstrated through a large set of experiments carried out through a prototype deployed in a real Internet-based environment where cloud database services are accessed concurrently by geographically distributed clients. All results confirm that for realistic workloads, the MuteDB architecture achieves performance and scalability comparable to those of unencrypted cloud database services. Ongoing work is focused on integrating private information retrieval solutions in MuteDB with the goal of preventing information leakage caused by access pattern analyses, and novel architectural solutions for hybrid cloud environments.

References

[1] S. Pearson and A. Benameur, "Privacy, security and trust issues arising from cloud computing," in *Proc. 2010 IEEE Int'l Conf. Cloud Computing Technology and Science*, Nov.-Dec. 2010, pp. 693 – 702.

[2] L. M. Vaquero, L. Rodero-Merino, and R. Buyya, "Dynamically scaling applications in the cloud," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 1, pp. 45–52, 2011.

[3] L. Ferretti, M. Colajanni, and M. Marchetti, "Distributed, concurrent, and independent access to encrypted cloud databases," *IEEE Trans. Parallel and Distributed Systems*, vol. 25, no. 2, pp. 437–446, 2014.
[4] 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, pp. 85–100.
[5] 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, pp. 74 – 83.
[6] G. Wang, Q. Liu, J. Wu, and M. Guo, "Hierarchical attribute-based encryption and scalable user revocation for sharing data in cloud servers," *Computers & Security*, vol. 30, no. 5, pp. 320–331, 2011.
[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, pp. 77–88.
[8] 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*, vol. 33, no. 3, pp. 3–12, 2003.
[9] O. Goldreich, *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge university press, 2004.
[10] 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'l Conf. Management of data*, Jun. 2002, pp. 216–227.
[11] E. Bertino, P. Samarati, and S. Jajodia, "Authorizations in relational database management systems," in *Proc. First ACM Conf. Computer and communications security*, Nov. 1993, pp. 130–139.
[12] E. Damiani, S. D. C. di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati, "Metadata management in outsourced encrypted databases," in *Proc. Second VLDB Int'l Conf. Secure Data Management*, 2005, pp. 16–32.
[13] 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*, vol. 1, no. 2, pp. 97–137, 1976.
[14] R. S. Sandhu and P. Samarati, "Access control: principle and practice," *IEEE Comm. Magazine*, vol. 32, no. 9, pp. 40–48, 1994.
[15] P. Samarati and S. De Capitani di Vimercati, "Access control: Policies, models, and mechanisms," in *Foundations of Security Analysis and Design*, 2001, pp. 137–196.
[16] 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*, vol. 12, no. 3, pp. 1–43, 2009.
[17] J. Crampton, K. Martin, and P. Wild, "On key assignment for hierarchical access control," in *Proc. 19th IEEE Workshop Computer Security Foundations*, Jul. 2006, pp. 98–111.
[18] S. Tu, M. Kaashoek, S. Madden, and N. Zeldovich, "Processing analytical queries over encrypted data," in *Proc. 39th Int'l Conf. Very Large Data Bases*, Aug. 2013, pp. 289–300.
[19] A. Boldyreva, N. Chenette, and A. O'Neill, "Order-preserving encryption revisited: Improved security analysis and alternative solutions," in *Proc. Advances in Cryptology*, Aug. 2011, pp. 578–595.
[20] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Proc. Advances in Cryptology*, May 1999.
[21] 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, pp. 143–154.
[22] M. Bellare, A. Boldyreva, and A. ONeill, "Deterministic and efficiently searchable encryption," in *Prof. Advances in Cryptology*. Springer, 2007, pp. 535–552.
[23] J. Daemen and V. Rijmen, *The design of Rijndael: AES – the advanced encryption standard*. Springer, 2002.
[24] B. Schneier, "Description of a new variable-length key, 64-bit block cipher (blowfish)," in *Proc. Cambridge Security Workshop Fast Software Encryption*, Dec. 1993, pp. 191–204.
[25] D. Song, D. Wagner, and A. Perrig, "Practical techniques for searches on encrypted data," in *Proc. IEEE Symp. Security and Privacy*, May 2000, pp. 44 – 55.
[26] 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*, vol. 25, no. 1, pp. 222–233, 2014.
[27] 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, pp. 337–350.
[28] 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*, vol. 29, no. 4, 2011.
[29] 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'l Conf. E-Commerce Technology*, 2005, pp. 559–565.
[30] 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'l Conf. Cloud Computing Technology and Science*, Nov.-Dec. 2011, pp. 689–703.
[31] S. Yu, C. Wang, K. Ren, and W. Lou, "Achieving secure, scalable, and fine-grained data access control in cloud computing," in *Proc. IEEE Conf. Computer Communications*, Mar. 2010.
[32] 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*, 2006, pp. 89–98.
[33] L. Ferretti, M. Colajanni, and M. Marchetti, "Access control enforcement of query-aware encrypted cloud databases," in *Proc. Fifth IEEE Int'l Conf. on Cloud Computing Technology and Science*, Dec. 2013, pp. 717–722.

Luca Ferretti is a Ph.D. student at the International Doctorate School in Information and Communication Technologies (ICT) of the University of Modena and Reggio Emilia, Italy. He received the Master Degree in computer engineering from the same University in 2012. His research focuses on information security, and cloud architectures and services. Home page: http://weblab.ing.unimo.it/people/ferretti

Fabio Pierazzi is a Ph.D. student at the International Doctorate School in Information and Communication Technologies (ICT) of the University of Modena and Reggio Emilia, Italy. He received the Master Degree in Computer Engineering from the same University in 2013. His research interests include security analytics and performance evaluation of cloud services. Home page: http://weblab.ing.unimo.it/people/fpierazzi

Michele Colajanni is full professor in computer engineering at the University of Modena and Reggio Emilia since 2000. He received the Master degree in computer science from the University of Pisa, and the Ph.D. degree in computer engineering from the University of Roma in 1992. He manages the Interdepartment Research Center on Security and Safety (CRIS), and the Master in "Information Security: Technology and Law". His research interests include security of large scale systems, performance and prediction models, Web and cloud systems. Home page: http://weblab.ing.unimo.it/people/colajanni

Mirco Marchetti received his Ph.D. in Information and Communication Technologies (ICT) in 2009. He holds a post-doc position at the Interdepartment Center for Research on Security and Safety (CRIS) of the University of Modena and Reggio Emilia. He is interested in intrusion detection, cloud security and in all aspects of information security. Home page: http://weblab.ing.unimo.it/people/marchetti