# DISTRIBUTED AUTHORIZATION IN LOOSELY COUPLED DATA FEDERATION

WEI LI

A THESIS

IN

THE CONCORDIA INSTITUTE FOR INFORMATION SYSTEMS ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF APPLIED SCIENCE IN INFORMATION SYSTEMS SECURITY

CONCORDIA UNIVERSITY

MONTRÉAL, QUÉBEC, CANADA

FEBRUARY 2009

*Your file Votre référence*
*ISBN:* 978-0-494-63200-0
*Our file Notre référence*
*ISBN:* 978-0-494-63200-0

# ABSTRACT

Distributed Authorization in Loosely Coupled Data Federation

Wei Li

The underlying data model of many integrated information systems is a collection of interoperable and autonomous database systems, namely, a loosely coupled data federation. A challenging security issue in designing such a data federation is to ensure the integrity and confidentiality of data stored in remote databases through distributed authorization of users. Existing solutions in centralized databases are not directly applicable here due to the lack of a centralized authority, and most solutions designed for outsourced databases cannot easily support frequent updates essential to a data federation. In this thesis, we provide a solution in three steps. First, we devise an architecture to support fully distributed, fine-grained, and data-dependent authorization in loosely coupled data federations. For this purpose, we adapt the integrity-lock architecture originally designed for multilevel secure databases to data federations. Second, we propose an integrity mechanism to detect, localize, and verify updates of data stored in remote databases while reducing communication overhead and limiting the impact of unauthorized updates. We realize the mechanism as a three-stage procedure based on a grid of Merkle Hash Trees built on relational tables. Third, we present a confidentiality mechanism to control remote users' accesses to sensitive data

while allowing authorization policies to be frequently updated. We achieve this objective through a new over-encryption scheme based on secret sharing. Finally, we evaluate the proposed architecture and mechanisms through experiments.

# Acknowledgments

I would like to express my gratitude to all those who gave me the possibility to complete this thesis.

First of all, I would like to express my sincerely thanks to my supervisor, Dr. Lingyu Wang, on whose constant encouragement guidance and advices I have relied throughout my entitle research at Concordia University, especially to the published papers, journal and this manuscript.

Especially, I would like to give my special thanks to my wife, Yiyi Wang, whose patient love enabled me to complete this work.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Data integration and information sharing have attracted significant interests lately. Although web services play a key role in data integration as the interface between autonomous systems, the underlying data model of the integrated system can usually be regarded as a collection of inter-operable and autonomous database systems, namely, a *loosely coupled database federation* [32]. Among various issues in designing such a database federation, the authorization of users requesting for data located in remote databases remains to be a challenging issue in spite of existing efforts.

The autonomous nature of a loosely coupled federation makes it difficult to directly apply most centralized authorization models. The subject and object in an access request may belong to different members of a federation that are unaware of each other's user accounts, roles, or authorization policies. Simply duplicating such information across the members is generally not a feasible solution due to the confidential nature of such information. In addition, the members of a database federation usually lack full trust in each other, especially in terms of confidentiality and integrity of sensitive data. On the other hand, although there are similarities between a loosely coupled database and outsourced databases, a fundamental difference is that data in a federation of operational databases is subject to constant updates. This difference prevents direct application of most existing

security solutions in outsourced databases to a database federation.

In this thesis, we propose a solution for distributed authorization in loosely coupled database federations. We describe the solution in three steps. First, we devise an architecture to support fully distributed, fine-grained, and data-dependent authorization in loosely coupled database federations. For this purpose, we adapt the *integrity-lock architecture* originally designed for multilevel secure databases to database federations. Although intended for a different purpose, the integrity lock architecture has properties that are particularly suitable for a loosely coupled database federation. The architecture does not require the remote database to be fully trusted but instead supports *end-to-end* security between the creation of a tuple to the inquiry of that tuple. This capability is essential to a database federation where members do not fully trust each other for authorization. The architecture binds authorization polices to the data itself, which can avoid duplicating data or authorization policies across the federation and also allows for attribute-level authorizations and authorizations that depend on data content.

Second, we propose an integrity mechanism to detect, localize, and verify updates of data stored in remote databases while reducing communication overhead and limiting the impact of unauthorized updates. We realize the mechanism as a three-stage procedure. In the first stage, a database detects modifications of remote data when such data are involved in a query. Detected modifications are localized using a two-dimensional grid of Merkel Hash Trees (MHTs). In the second stage, the two involved databases follow a common protocol to verify the legitimacy of detected modifications. The modified data are accepted as the result of legitimate updates only if the remote database can provide sufficient evidence. Finally, the local database updates the MHTs on the legitimate portion of remote data by excluding any unauthorized modifications. To reduce performance overhead in recomputing MHTs, we propose two caching schemes that are suitable for different types of queries. We evaluate the performance of those schemes through experiments.

Third, we present a confidentiality mechanism to control remote users' accesses to sensitive data while allowing authorization policies to be frequently updated. We achieve this objective through a new secret sharing-based *over-encryption* scheme. The over-encryption scheme doubly encrypts sensitive data at both the local database and remote database. Access control policies are enforced through publishing tokens that enable users to derive the encryption keys to which they are authorized. The two independent layers of encryption allows a remote database to be only partially trusted, and it also enables efficient updates of access control policies, which is particularly important for database federations. Our secret sharing-based scheme improves the performance of over-encryption by reducing the number of public tokens. We evaluate different implementations of the proposed scheme through experiments.

The main contribution of the thesis is two fold. First, with the proposed architecture and mechanisms, we provide a practical security solution to many data integration applications as long as their data model can be abstracted as a loosely coupled database federation. Second, by adapting existing architecture and methods in multilevel and outsourced databases to a database federation, we establish interesting connections between those distinct areas of research. The rest of the thesis is organized as follows. Chapter 2 reviews previous work. Chapter 3 illustrates security issues addressed in this thesis through a motivating example and describes the adapted integrity lock architecture in database federation. Chapter 4 proposes a three-stage procedure for supporting legitimate updates of remote data while ensuring their integrity. Chapter 5 devises a secret sharing-based over encryption scheme for supporting access control on remote data and efficient policy updates. Chapter 6 presents experimental results on the performance of the proposed solution. Chapter 7 concludes the thesis.

# Chapter 2

# Related Work

## 2.1 Database Federation

A Federated Database System (FDBS) is a collection of cooperating yet autonomous member database systems [32]. Member databases are usually heterogeneous in many aspects such as data models, query languages, authorization policies, and semantics (which refers to the fact that the same or similar data items may have different meanings or distinct intended usages among member databases). According to the degree of integration, FDBSs are mainly classified as *loosely coupled FDBS* and *tightly coupled FDBS*. A loosely coupled FDBS is rather like a collection of inter-operable database systems. Most research efforts have focused on a tightly coupled FDBS, where the federation, as an independent component, is created at design time and actively controls all accesses to member databases [5, 6, 12, 19, 38]. Although designing a tightly coupled FDBS from scratches has obvious advantages, in many cases it may not be feasible due to the implied costs. Our study assumes the loosely coupled FDBS model, and we do not require major modifications to existing DBMSs. This makes our approach more attractive to today's data integration applications.

Security issues such as access control are more challenging in a loosely coupled FDBS

than in a centralized database or a tightly coupled database federation due to the lack of a central authority and the autonomy in authorization that allows member databases to have partial control over shared data. Depending on the degree of such autonomy, the access control models can be divided into three classes [5]. With *full authorization autonomy*, member databases authenticate and authorize federation users as if they are accessing member databases directly. To the other extreme, *low authorization autonomy* fully trusts and relies on the central federation component to authenticate and authorize federation users. Our work considers the compromise between the two, namely *medium authorization autonomy*, where member databases have partial control on shared resources. Most existing efforts on medium authorization autonomy in FDBSs, such as *subject switching* [38], require members to agree on a loose mapping between user accounts and privileges across different databases.

By excluding corrupted data from query results, our approach allows the database federation to continue normal operation in the presence of unauthorized modifications. This is similar to database recovery mechanisms, such as those based on trusted repair algorithms using read-from dependency information [1] and the extended model based on state transition graphs [36]. However, our focus is not on the isolation and recovery from intrusions, but rather on the interaction between local and remote databases in a database federation. Multilevel databases have received enormous interests in the past, as surveyed in [15, 16, 28]. Various architectures have been proposed for building multilevel databases from un-trusted database components [28]. Those work mainly focus on the prevention of information flow between different security level while supporting cover stories that are essential to military applications [15, 16]. We adapt the integrity lock architecture originally proposed for multilevel databases [28] to database federations.

In a database federation, the sharing of data between databases bears a similarity to data publication in outsourced databases. The security of outsourced databases has attracted

significant interests [10, 22, 25, 26, 29]. One of the major issues in outsourced databases is to allow clients to verify the integrity of query results, because the database service provider in this model is usually not fully trusted [26]. Various techniques based on cryptographic signature and Merkel Hash Tree (MHT) [24] have been proposed to address the integrity, completeness, freshness, and other desired properties of query results. The key challenge in applying those techniques in outsourced databases to the federation of operational databases is that data are relatively static in the former while they are constantly being updated in the latter. To allow legitimate updates of data without having to ship them back to the owner (local database), we propose a protocol for the automatic detection and verification of updates on remote data.

## 2.2 Merkle Hash Tree (MHT)

Our discussions on incrementally updating MHTs is related to algorithms for reducing the time or space cost of MHT traversal [17]. Those algorithms aim to achieve tradeoffs between storage and computational efficiency in sequential traversals of a MHT. The algorithm in [17] uses subtrees for traversal of a MHT and discard intermediate nodes of a subtree when they are found in existing subtrees. The algorithm in [35] improves the classic MHT traversal algorithm in terms of less space requirement. A hybrid of those approaches is introduced in [20]. However, those algorithms focus on visiting every node in a MHT structure, which is slightly different from the incremental updates of MHT used in the verification of data updates.

## 2.3 Metadirectories and Virtual Directories

*Metadirectories* and *virtual directories* technology are related to our work. They both allow users to access data from different repositories by using directory mechanisms such

as *Lightweight Directory Access Protocol (LDAP). Metadirectories* needs to create a new repository to synchronize data from multiple source directories storing the data. When data in source directories changes frequently, it would be expensive to keep data updated due to excessive storage and computation overhead. Instead of maintaining a separate information repository, *virtual directories* create a virtualization layer to access information indirectly. The virtual directories based on a directory protocol, such as LDAP, works well under a hierarchical structure. LDAP, which is optimized for read but not for write, is mainly designed for data sharing. For security, LDAP certificates the identities through authentication methods. However, our approach is based on a different assumption that the remote database is not fully trusted by the local database so authentication between the two databases cannot be relied on.

## 2.4 Over-Encryption and Secret Sharing

Over-encryption is a novel technique introduced for enforcing access control and the efficient management of policy updates in outsourced databases [7]. In over-encryption, resources are doubly encrypted at the base encryption layer (BEL) and the surface encryption layer (SEL). The BEL layer encryption is imposed by the owner for providing initial protection; the SEL layer encryption is imposed by the outsourced server to reflect policy modifications. One potential limitation of the over-encryption scheme is that it may require to publish too many tokens when the number of users is large. Instead of relying on key derivation function, we base our over-encryption scheme upon secret-sharing to reduce the number of public tokens [33]. A number of different proposals exist on secret sharing schemes [3, 4, 27, 31] among which we apply Shamir's scheme [31]. Another related area of research is the group key management [21, 37] and hierarchical key assignment [18, 30]. Those schemes classify data into different levels and generate a key for each level, with lower level keys dependent on higher level keys. However, those schemes are generally

based on tree architectures and are not suitable for over-encryption.

# Chapter 3

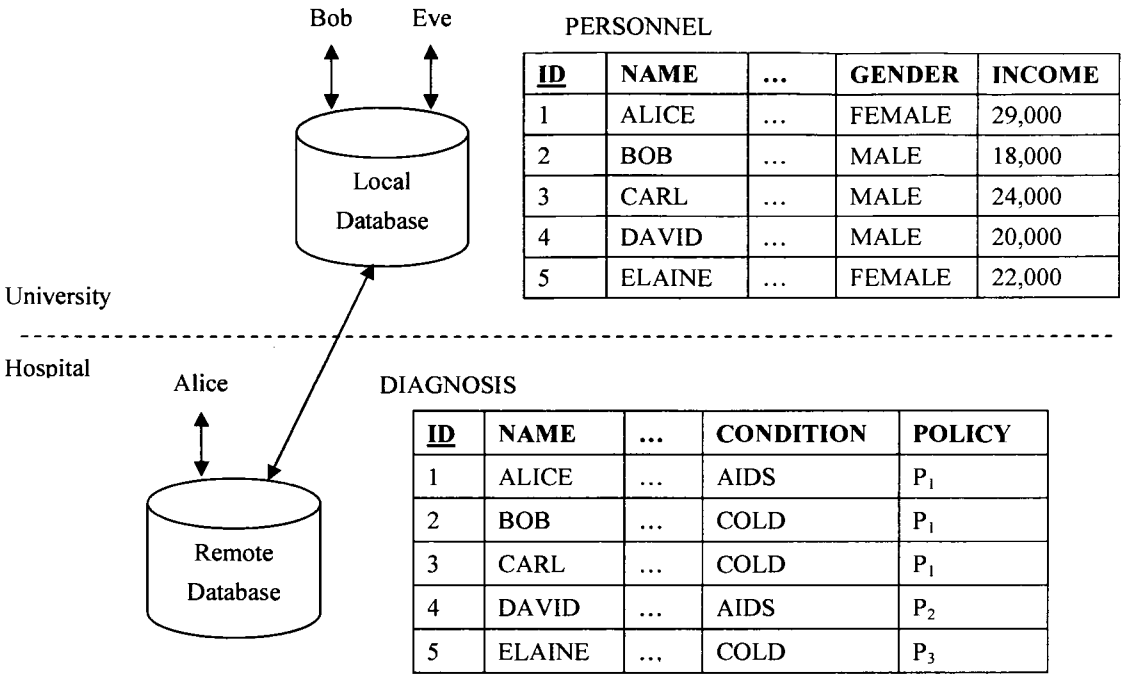
# Integrity Lock Architecture for Database Federation

Section 3.1 first illustrates security issues in a loosely coupled database federation. Section 3.2 then gives a high-level overview of our solution by adapting the integrity lock architecture to database federations.

## 3.1 Motivating Example

Unlike a tightly coupled database federation, a loosely coupled database federation has no centralized federation component created at design time to actively control accesses to each member of the federation. Instead, the two databases are autonomous members of the federation that directly interact with each other. To illustrate security issues that may arise due to such interaction between federation members, we consider a concrete case in the following.

Figure 1 depicts a simplified scenario of the interaction between two databases in a loosely coupled database federation. In this example, we assume a fictitious university and its designated hospital are aiming to establish an integrated application to provide the




PERSONNEL

| ID | NAME | ... | GENDER | INCOME |
|---|---|---|---|---|
| 1 | ALICE | ... | FEMALE | 29,000 |
| 2 | BOB | ... | MALE | 18,000 |
| 3 | CARL | ... | MALE | 24,000 |
| 4 | DAVID | ... | MALE | 20,000 |
| 5 | ELAINE | ... | FEMALE | 22,000 |

DIAGNOSIS

| ID | NAME | ... | CONDITION | POLICY |
|---|---|---|---|---|
| 1 | ALICE | ... | AIDS | $P_1$ |
| 2 | BOB | ... | COLD | $P_1$ |
| 3 | CARL | ... | COLD | $P_1$ |
| 4 | DAVID | ... | AIDS | $P_2$ |
| 5 | ELAINE | ... | COLD | $P_3$ |

Figure 1: An Example of Interaction Between Federation Members

university's employees (as depicted in the $PERSONNEL$ table) direct accesses to their medical records hosted at the hospital (the $DIAGNOSIS$ table). Bob and Eve are two users of the university, and Alice belongs to the hospital. The two tables both contain facts about employees of the university and they have two common attributes $ID$ and $NAME$. As a normal employee of the university, Bob should not have free accesses to other employees' $CONDITION$ attribute values hosted at the hospital. On the other hand, another user at the university side, Eve, may be authorized to access records of a selected group of employees due to her special job function (for example, as a staff working at the university clinic or as a secretary in a department). At the hospital side, Alice is prohibited from accessing the $INCOME$ attribute of any university employee. However, as a doctor designated by the university, Alice is authorized to access and modify the $CONDITION$ attribute.

To realize the above scenario, a loosely coupled database federation has advantages over

centralized approaches. First, we can store the $CONDITION$ attribute in the university-side database and thus completely eliminate the hospital-side table. However, the attribute $CONDITION$ and other related medical data will most likely be frequently accessed and updated at the hospital side. Storing those attributes at the hospital is thus a more natural choice. Second, the university would certainly be reluctant to move or duplicate the table $PERSONNEL$ to the hospital side due to its sensitive nature. The above scenario is also different from the case of two separate organizations. In this case, the university is responsible for its employees' medical records even though the records are stored in the hospital. From this point of view, we can regard the *local database* at the university as outsourcing its data to the *remote database* at the hospital. However, different from a outsourced database which is relatively static, here the data are constantly subject to updates.

The above scenario shows the need for distributed authorization. The local database at the university apparently needs to verify the legitimacy of accesses and updates to data stored in the remote database at the hospital. Such verification is needed to ensure all updates to be in accordance with policies or contractual conditions that may have been agreed upon during the formation of the database federation. For example, only a doctor designated by the university is allowed to access and modify the $CONDITION$ attribute. We consider following possible approaches to such an authorization.

- We could choose to let the university trust the hospital in enforcing such a policy. However, this implies trust in not only the hospital as an organization, but also any user who gains accesses to the hospital-side data. The autonomous nature of a loosely coupled federation most likely will render such amount of trust unacceptable to the university. Another difficulty is that the university may have to export its employees' account information (for example, Eve is a secretary of a certain department) to the hospital so the latter can enforce access control based on such information. Again,

this fact is inconvenient since such account information usually includes sensitive data about the university's employees.

- Another possible approach is to enforce access control policies completely at the university side, with no trust in the hospital. This solution works fine for users at the university under policies that are either data-independent (for example, no user should ever access the $POLICY$ attribute) or only dependent on attributes in the university's table (for example, Bob should only access his own record). However, the solution cannot easily handle a policy that depends on attributes in the hospital's database, such as $CONDITION \neq AIDS$. If Bob has many records with different $CONDITION$ values, all with different policies, then even storing such policies in the university's database will be difficult. Also, for users at the hospital, this solution would require constant communication between the two databases.

- In this thesis, we adopt a distributed authorization approach that is based on the *trust but verify* principle. In this particular example, the university will trust the hospital in enforcing data dependent policies. However, whenever remote data are sent from the hospital to the university as query results, the university will attempt to verify the integrity of such data. The hospital must provide evidence to prove any detected modification to be the result of legitimate updates from authorized users. Such an approach allows the hospital to ensure the integrity of remote data without having to authorize every update. With the assumption that the hospital as an organization is trustworthy but all of its users are not, such a distributed authorization approach has advantages in terms of both security and performance.

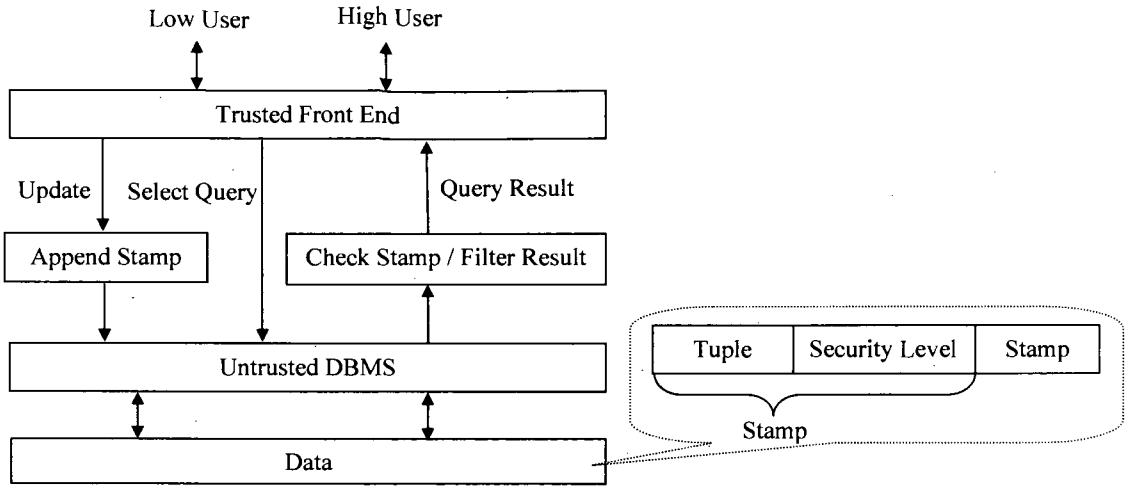



Figure 2: The Integrity Lock Architecture

## 3.2 Integrity Lock Architecture

At the architecture level, we need to decide where and when to enforce security policies. We borrow the *integrity lock architecture*, which is originally proposed for multi-level databases [28], to support distributed authorization in loosely coupled database federations. Unlike databases in commercial worlds, in multilevel databases, both users and data are classified with different security levels, such as top secret, secret, confidential, or unclassified. The primary concern is to prevent information from flowing downwards across the security levels. The main objective of the integrity lock architecture is to reduce costs by building secure multilevel databases from un-trusted off-the-shelf DBMS components.

Figure 2 illustrates a simplified integrity lock architecture where two security levels, *high* and *low*, are considered. The integrity lock architecture depends on a trusted front end (also called a filter) to mediate accesses between users and the un-trusted DBMS (the original model also has an un-trusted front end, which is omitted here for simplicity) [8,9,11,23]. Each tuple has two additional attributes, namely, a security level and a cryptographic stamp. The stamp is basically a message authentication code (MAC) computed over the whole tuple excluding the stamp using a cryptographic key known to the trusted front end.

When a tuple is to be inserted or updated by a legitimate user, the trusted front end will determine the security level of the new tuple, and it will compute the stamp and append it to the query. The trusted front end determines the security level of the new tuple and computes the stamp to append it to the query when a tuple is to be inserted or updated. The query is then forwarded to the DBMS for execution. When users submit a legitimate selection query, the trusted front end will simply forward the query to the DBMS. Upon receiving the query result from the latter, the trusted front end will verify all tuples in the result and their security levels by recomputing and matching the cryptographic stamps. If all the data check out, the trusted front end will then filter out prohibited tuples based on their security levels, the user's security level, and the security policy. For example, low users are not allowed to retrieve high tuples. The remaining tuples are then returned to the user as the query result.

Instead of relying on a secure DBMS, which incurs higher cost to build, the integrity lock architecture provides *end-to-end security* from the time a tuple is created (or modified) to the time it is returned in a query result. The un-trusted DBMS cannot alter any tuple or its associated security level without being detected. Such a capability naturally fits in the requirements of a database federation. More specifically, in Figure 1, we can regard the university-side database as the trusted front end, and the hospital-side database as an un-trusted DBMS in the integrity lock architecture. The security levels of users and tuples in the integrity lock architecture can be interpreted as users' credentials (for example, user IDs, groups, or roles) and the security policies associated with tuples, respectively. Suppose a user Eve of the university-side database wants to insert or update some records in the table stored at the hospital (for example, to create or update an account for an employee). The university-side database will compute and append a cryptographic stamp to the tuple to be inserted or updated.

As in the original integrity lock architecture, the cryptographic stamp is a MAC computed over all attributes of the tuple, including the access control policies associated with that tuple (which is provided by Eve). When a user of the university-side database wants to select tuples in the hospital-side database, the university database will enforce any policy that is locally stored through either rejecting or modifying the original query posed by the user. For example, if Bob is only allowed to ask about his own records, then his query will be modified by appending a `WHERE` clause `NAME='Bob'`. The university database then forwards the modified query to the hospital database for processing. Upon receiving query results from the latter, the university database will then verify the integrity of each returned tuple in the results through the cryptographic stamp in the tuple. It then filters out any tuple that Bob is not allowed to access according to the access control policy associated with that tuple.

In the context of multilevel databases, a known complication of the integrity lock architecture is its vulnerability to two kinds of inference attacks [9]. In particular, *Trojan horse leakage* refers to the covert channel that an un-trusted DBMS can signal a 0 or 1 bit by returning different tuples as the result of the same query. Such a threat is more of a concern to multilevel systems used by military or governmental organizations and we shall not consider it further. On the other hand, *user inference* allows a user to infer prohibited data from the result of legitimate queries. For example, in Figure 1 if Bob asks the following query: `SELECT ID, NAME FROM DIAGNOSIS WHERE NAME='ALICE and CONDITION='AIDS'`. The query will be allowed because the result returned by the hospital-side database, $(1, Alice)$, only includes data that Bob is allowed to access. However, Bob can then infer that Alice has AIDS. Denning gives a solution to such inference problem, namely, the commutative filter [9]. A commutative filter answers a query only if its result is the same as if it had been computed on a database with all prohibited data removed. In the rest of this thesis, we shall assume such solutions are in place.

The security of any architecture critically depends on its proper implementations. The adapted integrity lock architecture faces following implementation issues. First, the original architecture requires a whole tuple to be returned by the un-trusted DBMS, even if the query only asks for one or two attributes [8, 11], because the cryptographic stamp is computed over the whole tuple (excluding the stamp itself). This limitation may cause unnecessary communication overhead between databases in a federation, if queries involve projections. Second, the integrity lock architecture can only detect modified tuples but cannot detect the omission of tuples in a query result. That is, the completeness of query results is not guaranteed. Similar issues have recently been addressed in the context of outsourced databases (ODB) [10, 22, 29]. The solution typically involves implementing cryptographic stamps as the signature of root of a MHT on each tuple, with all attribute values being the leaves. Since the root of a MHT can be computed from any subset of the leaves plus a small number of sibling nodes, communication cost is reduced. Moreover, omitting tuples from query results will be detected when comparing a recomputed signature of the root to the stamp.

However, simply applying the aforementioned solutions in ODB to the integrity lock architecture in database federations is not practical. A fundamental difference between ODB and database federations is that the former usually assumes a relatively static database with no or infrequent updates. In the ODB model, the database service provider is generally not supposed to modify the outsourced data. Existing techniques in ODB thus mainly focus on the detection of modifications with pre-computed signatures given to users. Data updates usually imply significant computational and communication costs. In the case of MHT-based solutions, the signature of the root must be updated immediately after every update, because no future query can be verified before this update (the verification of all queries depends on the same signature). Such an overhead is not acceptable to database federations, because the members of such a federation are typically operational databases

where data are constantly being updated. We shall address such issues in next section.

# Chapter 4

# Ensuring Data Integrity while Supporting Frequent Updates

In this section, we present mechanisms for ensuring the integrity of data while allowing legitimate updates under the integrity lock architecture. First, Section 4.1 provides an overview of our approach. Section 4.2 then shows how to detect and localize modifications using a grid of MHTs. Section 4.3 presents a procedure for verifying modifications. Section 4.4 provides a solution to incrementally update the grid of MHTs upon legitimate updates of data. Finally, Section 4.5 evaluates the security of our approach.

## 4.1 Overview

First of all, we describe what we mean by *authorized users*. As mentioned earlier, we shall refer to the database hosting shared data as *remote database* and the other database *local database*. In forming the federation, each member database should be given the capability of authenticating users of a remote database, without the help of that remote database. Our solution will not depend on specific ways of implementing such authentication, although we shall consider a concrete case where a remote user possesses a public/private key pair,

so the user's query can be authenticated through digital signatures created using the private key.

To ensure the integrity of data stored in a remote database, two seemingly viable approaches are either to verify the update queries, or to verify the state of remote data immediately after each update. For example, in Figure 1, whenever Alice attempts to update a record, the hospital-side database can send the query and the records to be updated, which are both digitally signed by Alice, to the university-side database for verification. The latter will verify the legitimacy of the update by comparing Alice's credential to the access control policies stored in the records. However, this approach is not effective because the hospital-side database must be trusted in forwarding all update queries for verification and in incorporating all and only those legitimate updates after they are verified. As an example of the second approach, the university-side database can choose to verify the state of remote data after each update is made to the data. However, this approach faces two difficulties. First of all, it is difficult to know about every update if the remote database is not trusted since it may delay or omit reporting an update. Moreover, the approach may incur unnecessary performance overhead. For example, a doctor may need to make several temporary updates to a diagnosis record before a final conclusion can be drawn. The university-side database does not need to verify all those temporary updates.

We take a three-stage approach, as outlined below and elaborated in following sections. First, referring to the example in Figure 1, the university-side database will detect modifications in a *lazy* manner. More precisely, when Bob or Eve issues a selection query and the result is returned by the hospital-side database, the university-side database will attempt to detect and localize modifications in the tuples involved in the query result using a two-dimensional grid of MHTs. Second, if a modification is detected and localized, then the local database will request the remote database to provide proofs for the legitimacy of such updates. The remote database then submits necessary log entries containing digitally signed

| $\underline{A_1}$ | $A_2$ | $A_3$ | $A_4$ | $A_5$ | ... | $A_n$ | $A_{n+1}$ |
|---|---|---|---|---|---|---|---|
| $v_{1,1}$ | $v_{1,2}$ | ... | | | | $v_{1,n}$ | $y_1$ |
| $v_{2,1}$ | $v_{2,2}$ | ... | | | | $v_{2,n}$ | $y_2$ |
| ... | ... | ... | | | | ... | ... |
| $v_{m,1}$ | $v_{m,2}$ | ... | | | | $v_{m,n}$ | $y_m$ |
| $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | ... | $x_n$ | |

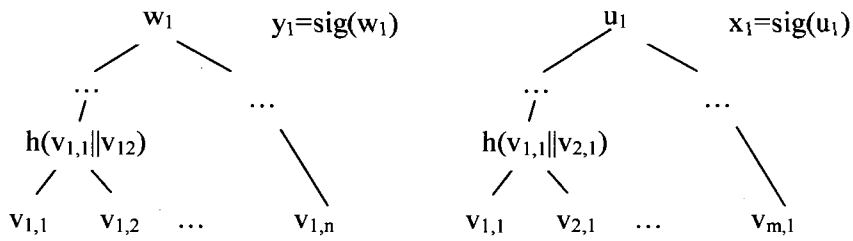



Figure 3: A Grid of Merkel Hash Trees on Tables

update queries corresponding to those updates. The local database will check whether the queries are made by those users who are authorized for such updates, and whether those queries indeed correspond to the modified data. Third, the local database will then disregard any tuples in the query result for which no valid proof can be provided by the remote database. To accommodate legitimate updates, the local database will incrementally compute new signatures and send them back to the remote database who will incorporate those new signatures into the tuples.

## 4.2 Detecting and Localizing Modifications

We compute a two-dimensional grid of MHTs on a table to detect and localize any modification to tuple or attribute level (a similar idea was applied to watermarks in [13]). In Figure 3, the attributes are denoted as $A_i (1 \leq i \leq n+1)$, among which we assume $A_1$ is the primary key and $A_n$ the access control policy for each tuple. The MHT is built with a

collision-free hash function $h()$ and a public key signature algorithm $sig()$. Each $y_i (1 \leq i \leq m)$ is the signature of the root $w_i$ of a MHT built on the tuple $(v_{i,1}, v_{i,2}, \ldots, v_{i,n})$. Similarly, each $x_i$ is a signature of the root $u_i$ of the MHT built on the column $(v_{1,i}, v_{2,i}, \ldots, v_{m,i})$. For example, in Figure 1, for the hospital-side table, the signatures will be created by the university-side (local) database using its private key. If a table includes tuples jointly *owned* by multiple local databases, then multiple signatures can be created and then aggregated (for example, using the Condensed RSA scheme [26]) as one attribute value, so any involved database can verify the signature.

When a user at the local database poses a selection-projection query whose result includes a set of values $V \subseteq \{v_{i,j} \mid 1 \leq i \leq m, 1 \leq j \leq n-1\}$, the remote database needs to return the set $V$, the policy $v_{i,n}$ and the signatures $x_i$ and $y_j$ for each $v_{i,j} \in V$. Moreover, the siblings needed for computing the root of the MHTs from which the signatures have been computed should also be returned. Upon receiving the query result, the local database will verify the signatures and values in $V$ by re-computing roots of corresponding MHTs. If all signatures are valid, then the local database is assured about the integrity of data. It will then examine the access control policies and filter out those tuples not allowed to be accessed by the user, and check the completeness of the query result based on the MHTs. If everything checks out, the query will be answered.

If some signatures do not match those included in query result, then modified data must first be localized based on following observations. If a value $v_{i,j}$ is updated, then signatures $y_i$ and $x_j$ will both mismatch. The insertion of a new tuple $(v_{i,1}, v_{i,2}, \ldots, v_{i,n})$ will cause signatures $x_1, x_2, \ldots, x_n$ and $y_i$ to mismatch, while all the $y_j (j \neq i)$ will still match. The deletion of a tuple $(v_{i,1}, v_{i,2}, \ldots, v_{i,n})$ will cause signatures $x_1, x_2, \ldots, x_n$ to mismatch, while all the $y_i (1 \leq i \leq n-1)$ will still match. The first three pictures in Figure 4 depict these cases.

The localization of modifications helps to reduce the amount of proofs that need to be

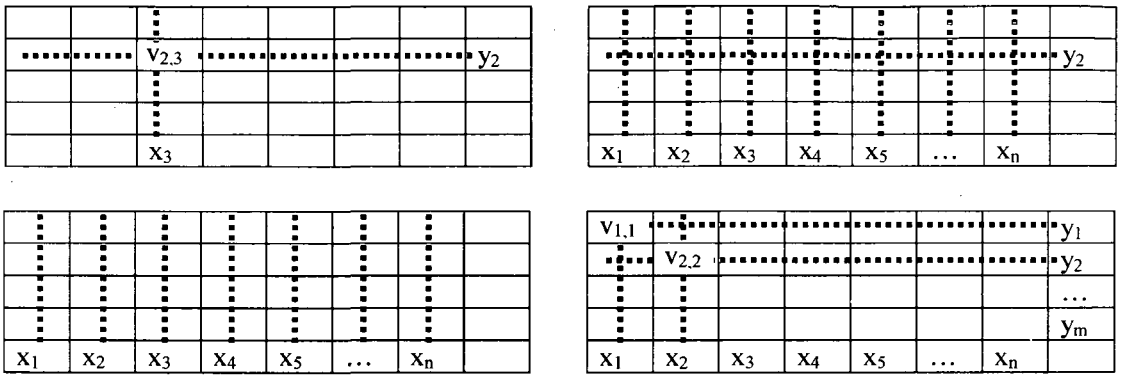



Figure 4: Localizing Updates With MHT Grid

provided (and thus the communication and computational cost) in the verification phase. However, this mechanism does not guarantee the precise identification of every update made to the data. For example, in the lower-left chart in Figure 4, we cannot tell how many (or which) tuples have been deleted from the mismatched signatures. Also, in the lower-right chart, we cannot tell whether two, three, or four values have been modified from the four mismatched signatures. Fortunately, as we shall show, the verification phase does not rely on this localization mechanism.

We notice that a query usually involves only a subset of tuples or attributes. An update of data thus may not be reflected in the result of every query. For example, in the lower-right chart in Figure 4, the query result will only include four signatures, so updates of any value not covered by the four dot lines will not be detected. This observation indicates another aspect of the *lazy* approach in detecting modifications (the first aspect is that we only detect modifications at query time). That is, an update may not affect the verification process of subsequent queries that do not involve the updated value. If we were not to compute a MHT on each column but instead build a single MHT over all the signatures $y_i$'s, then any update will always cause the verification of all subsequent queries to fail until the update has been either rejected or accepted in later phases, which implies unnecessary performance overhead.

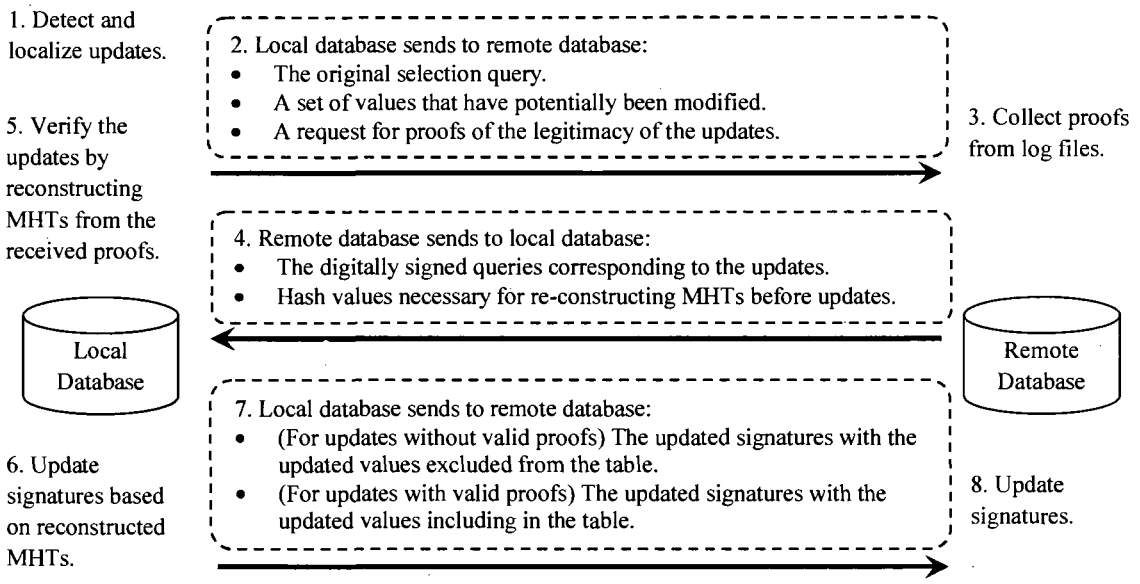



Figure 5: The Protocol for the Verification of Updates

## 4.3 Verifying the Legitimacy of Updates

As part of the protocol for verifying updates, we describe how a remote database handles updates. A remote database will need to record all the following into a log file: The update query, the signature of the query created with the user's private key, the current time, the current value before the update for deletion, and the current signatures involved by the update. Such information in the log file will allow the remote database to be rolled back to the last valid state without any update in effect [14]. The information will thus act as proofs for the legitimacy of updates. When updates are detected and localized, the local and remote databases will both follow the protocol shown in Figure 5 to automatically verify the legitimacy of those updates and to accommodate legitimate updates by updating signatures stored in the table.

In step 1, the local database detects mismatches in signatures and localizes the updates to a set of values that may have been updated (recall that the localization does not guarantee the precise set of modified values). The local database will then send to the remote database the potentially updated values and related information, such as the original selection query,

in step 2. In step 3, the remote database examines its log files to find each update query that involves the received values. For each such query, the remote database will attempt to reconstruct the mismatched signatures using values and signatures found in the log file, which are supposed to be in effect before the update. If a state is found in which all the mismatched signatures match again, then the involved queries will be collected as proofs and sent to the local database in step 4. Otherwise, the remote database will send to the local database a response indicating no proof for the updates is found.

In step 5, the local database will verify the signatures of the received update queries and ensure those queries are made by users who are authorized for such updates. The local database then attempts to reconstruct from the received queries a previous valid state in which all mismatched signatures match again. If such a state is found and all the update queries until that state are made by authorized users, then the detected updates are legitimate so the local database will recompute signatures by including the updated values (the details will be given in the next section) in step 6. Otherwise, the updates are unauthorized, so signatures are created by excluding the updated values in step 6. Upon receiving the updated signatures in step 7, the remote database will then update the received signatures in the table in step 8. The local database will only answer the original selection query if all the involved values are successfully verified.

A few subtleties of the protocol are as follows.

- It may seem to be a viable choice for the local database to stop after verifying that all the received update queries are made by authorized users, without having to go through again the reconstruction of a previous valid state. However, in this case, the remote database may actually make unauthorized modifications without being detected. Referring to the lower-right chart in Figure 4, suppose $v_{1,1}$ and $v_{2,2}$ are updated by authorized users. The remote database can freely modify $v_{1,2}$, $v_{2,1}$ and then send in the update queries corresponding to only $v_{1,1}$ and $v_{2,2}$. The local database will

not be able to detect the omitted updates from the mismatched signatures $x_1, x_2, y_1$ and $y_2$. However, since we let the local database to reconstruct each previous state from the received queries, it will detect the claimed updates to be unauthorized and then refuse to answer any selection query involving those four values.

- The protocol requires two rounds of communications between the two databases in addition to sending query result, that is, to request for proofs of legitimate updates and to update stamps. This protocol can certainly be optimized as follows. Instead of waiting for the local database to request for proofs about updates, the remote database can proactively detect updates by itself, identify proofs for the updates, then piggyback the query result and the proofs.

- We do not consider potential denial of service attacks. Such an attack is clearly possible since any unauthorized modification may cause queries to be denied. However, we regard answering queries with modified values to be a greater threat than denying the queries since the former may lead to misleading results. Moreover, considering the collaborative nature of a database federation, the local database may request the remote database to initiate an investigation after a certain number of queries are denied due to unauthorized updates.

- We choose not to lock a modified value after it is detected and before it is either verified or known to be the result of an unauthorized update, which may seem to be a viable approach. The reason is that the remote database is fully trusted, so whether it locks a modified value cannot be counted on.

- We do not treat the access control policy attribute in a special way since some users of a remote database may be authorized to update such policies. If this is not the case, the local database can simply regard any update to $x_n$ (the signature computed on the access control policy attribute) as unauthorized.

- The proofs may include not only the updated values but also other values indirectly involved in the updates. It is the remote database's responsibility to provide sufficient proofs so that the local database can roll back the current database state as reflected in the query result to a previous valid state using the update queries (the roll back of database states is well studied in transaction processing and is out of the scope of this thesis).

## 4.4 Accommodating Legitimate Updates

To accommodate updates that are successfully verified to be made by authorized users, the local database needs to compute new signatures by including the updated values so the remote database can update the signatures in the table. Similarly, updates of signatures are also required for newly inserted tuples. Recomputing signatures for each tuple does not incur a significant performance overhead because the number of attributes in a table is limited. However, the signature of a column may be computed over a large number of tuples, and its computation is thus costly. Moreover, any insertion or update of a tuple will cause at least one of the signatures of columns to be updated.

To reduce the computational cost of updates, an obvious solution is to divide the table into smaller subtables with fewer tuples and then apply the aforementioned grid of MHTs to each subtable independently, or equivalently, to simply change the way the grid of MHTs is computed. At first glance, having near-square subtables may seem to be the optimal choice. Moreover, we can compute one additional stamp over all the values in each subtable. As showed in figure 6, a $q$-column and $p$-row table ($p$ is much bigger than $q$, since in the database system the number of the attributes of the table is much smaller than the number of the records), and each sub-table has its signatures. In the verification phase, only the stamp of sub-table needs to be sent over if the remote database finds no value is updated inside this particular subtable. However, in reality the optimal choice of subtable sizes is a

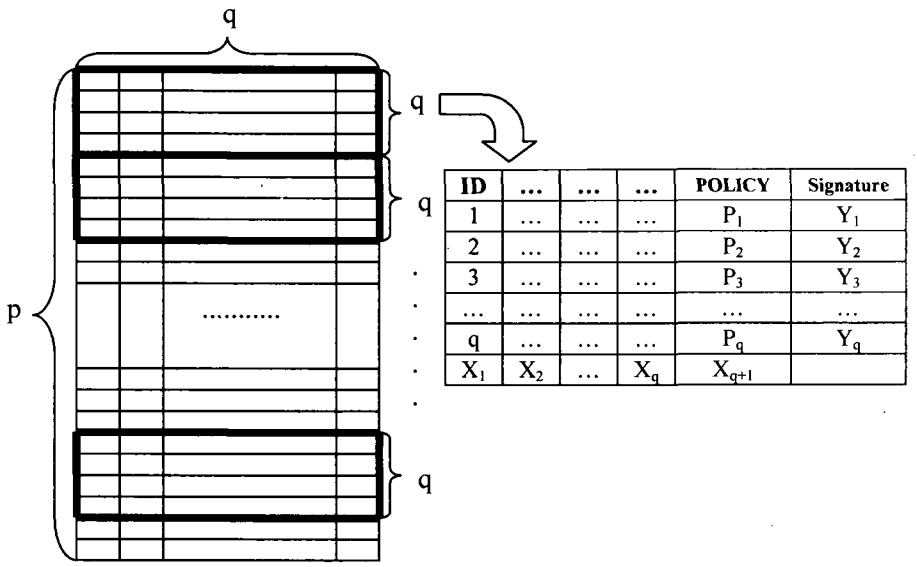



Figure 6: Partition Table into Sub-tables

little more complicated. As the size of subtable decreases, less computation is needed for recomputing the stamp over each column of the subtable, but a typical selection query may actually involve more subtables.

**Definition 1** *The involved sub-tables are all the sub-table which contains the return records.*

**Definition 2** *The stamp $\mathcal{Z}$ of a sub-table is a hash value that built based on all stamps on each column and each row of such table.*

When the user on remote database executes an update query, remote database will generate the hash of this query as the head, and send it to tell local database that there is an update on remote database. If the user has more than one update queries, then remote database will create a hash chain based on such set of queries and send the final hash result to local database. Hash chain bring a quick solution to check the remote database's integrity after a set of update. With the hash chain, any modification of the set of the queries will be detected at local database side. The hash chain will change when a malicious user adds a query to the set, deletes query from the set or reorders the set. Next time when user

send a new update query, the new header of the hash chain will be calculated and sent to local database. Local database can ensure freshness of the update that remote database user made. For example, if user Alice has N update queries on one record, remote database will record all of these queries and compute the hash chain. Suppose these queries are $M_1$, $M_2$,... $M_n$, the head of hash chain will be $h(M_1) \oplus h^2(M_2) \oplus h^3(M_3) \ldots \oplus h^n(M_n)$. This head will signatured by Alice and send it to local database together with return result.

In verification, we just check the integrity of the involved sub-tables. When local send a SELECT query, remote database first check the whether these return records modified, if not, just send the siblings and stamp Z of involved sub-tables. Otherwise send each row and each column's stamps of involved tables. If table is small enough, stamps numbers will reduced. However, the involved table's number maybe grow. So we adopt some experiment to get the suitable size of sub-table. We partition the database table into several small sub-tables, for each sub-table we use grid based stamps, and moreover we add one stamp $\mathcal{Z}$ for each sub-table as mentioned in definition 2. According to the algorithm 1, when executing a select query, remote database can check the stamp of each involved sub-table to see if there are modified records are involved in the result. If no then send the stamp $\mathcal{Z}$ and the siblings of such sub-table to local database, otherwise remote database should not only send $\mathcal{Z}$ and sibling but the stamps of modified tuples as well. We shall study this tradeoff through experiments in Chapter 6.

Another possible solution is to incrementally update the MHTs. As illustrated in Figure 7, to update the hash value 3, the local database only needs the hash values 1, 2 in the MHT of each column, instead of all the leaves. To balance the MHT over time, for insertion of new tuples, we should choose to insert each value at an existing hash value that has the shortest path to the root (this may not be feasible for ordered attributes where the order of MHT leaves is used for ensuring the completeness of query results). The next question, however, is where to obtain the required hash values 1 and 2, given that recomputing them

**Algorithm 1** Communication Algorithm for Sub-table

```
1: previousid ⇐ 0 {the sub-table id for previous record}
2: updated ⇐ 0 {the number of sub-table with updated record}
3: noupdated ⇐ 0 {the number of sub-table without updated record}
4: while is select query result do
5:    id ⇐ subtable id
6:    if tmpid = id then
7:       break
8:    else
9:       tmpid ⇐ id
10:      ischange ⇐ false
11:      for all tuples in sub-table do
12:         if tuple is updated then
13:            ischange ⇐ true
14:            break
15:         end if
16:      end for
17:      if ischange is true then
18:         updated ⇐ updated+1
19:      else
20:         noupdated ⇐ noupdated+1
21:      end if
22:   end if
23: end while
24: Communication Cost ⇐ updated*(sibling+1+updated records) + noupdated*(sibling+1)
```

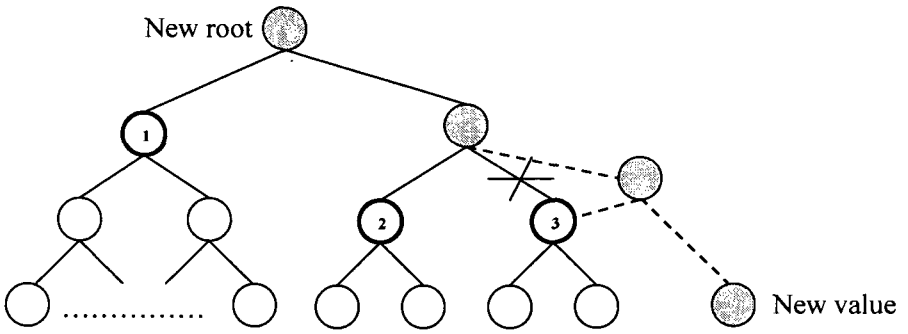



Figure 7: Update the Root of a MHT

from the leaves is not an option. One possibility is to keep a cache of all or part of the non-leaf hash values in the MHT. If we keep all the non-leaf values in a cache, then a direct lookup in the cache will be sufficient for computing the root, which has a logarithm complexity in the cardinality of the table (or subtable).

Considering the fact that the number of all non-leaf values is comparable to the number of leaves, the storage overhead is prohibitive. Instead, we can choose to cache only part of the MHT based on available storage. Two approaches are possible. First, we can use a static cache for a fixed portion of the MHT. If we assume a query will uniformly select any tuple, then clearly the higher a hash value is in the MHT, the more chance it will be useful in recomputing the new root of the MHT. For example, in Figure 7, value 1 will be needed in the update of twice as many values as value 2 will. Given a limited storage, we thus fill the cache in a top-down manner (excluding the root).

The assumption that queries uniformly select tuples may not hold in many cases. Instead, subsequent queries may actually select adjacent tuples in the table. In this case, it will lead to better performance to let the queries to drive the caching of hash values. We consider the following dynamic caching scheme. We start with the cache of a top portion of the MHT. Each time we update one tuple, we recompute the new root with the updated value by using as many values as possible from the cache. However, for each non-leaf value we need to recompute due to its absence in the cache, we insert this value into the

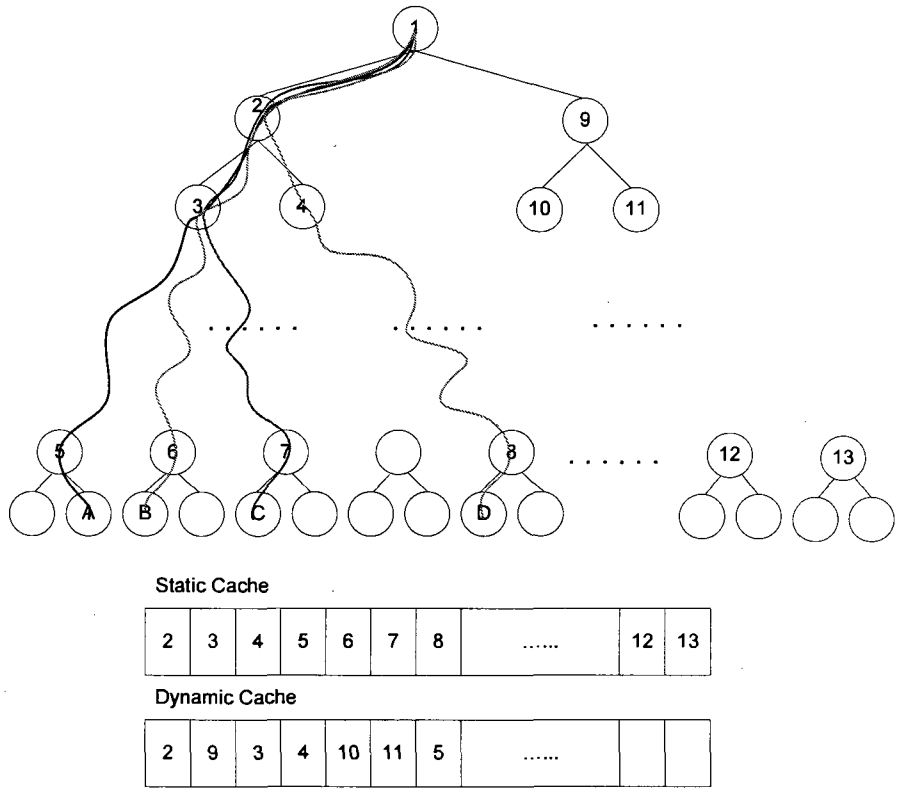



Figure 8: Static Cache and Dynamic Cache

cache by replacing a value that is least recently used (other standard caching schemes can certainly be used). Among those that have the same time stamp for last use, we replace the value that has the longest path from the root. Figure 8 illustrates the use of both static cache (that holds all non-leaf hashes) and dynamic cache where most queries involve only the leftmost leaves.

## 4.5 Security Analysis

In the following, we briefly discuss how the proposed scheme prevents various attacks.

- Suppose a malicious user of a remote database inserts, deletes, or modify tuples or attributes. Such modifications will cause mismatches between the recomputed

signature of MHT roots and the signatures stored in the table, by which the local database will detect modifications.

- The malicious user may attempt to modify the log entries to hide his activities by masquerading as users authorized for the updates. However, the local database can authenticate remote users' queries through their signatures and such signatures cannot be created by the malicious user without the private key of an authorized user.

- The malicious user can prevent the remote database from sending proofs or reporting the absence of proofs, but this does not help him/her to avoid detection (a timeout scheme can be used for the case of not receiving proofs in a timely fashion).

- The malicious user can also reorder or mix updates made by authorized users with his/her unauthorized updates. However, this will also be detected when the local database attempts to rebuild a previous valid state of data but fails.

- The only damage that can be realistically caused by malicious users is a denial of service when too many tuples are excluded due to unauthorized modifications. However, as mentioned before, a database member may request the remote database to initiate an investigation when the number of such tuples exceeds a threshold.

One security issue not addressed by the proposed scheme is the freshness of query results. That is, a remote database controlled by malicious users may never execute the last legitimate update query, which will not be detected since the database state is old, but valid. To ensure freshness of query result, it is essential for the user-side application that updates data at the remote database to communicate with the local database. A simple approach is for each user-side application to send the head of a hash chain formed by the hash values of update queries issued by that user. Holding the head of the hash chain, the local database can easily detect any omission of update queries in the proofs sent by the remote database. Another potential issue is the use of random functions in an update query.

The above verification technique will not work properly in this case because a different result may be yielded by each execution of the same query. A solution is for the user-side application to sign the query after the random function has already been executed at the remote database. Notice that although the remote database may potentially lie to the user about the result of that random function, this misbehavior will later be detected by the local database since the proof is based on the update query signed by the user.

# Chapter 5

# Ensuring Data Confidentiality Through Over-Encryption

In this section, we present mechanisms for achieving the confidentiality of remote data. Section 5.1 firsts provides an overview of over-encryption. Section 5.2 details our new secret sharing-based over-encryption scheme. Section 5.3 provides the way to query on the encrypted database. Section 5.4 presents a case study to further illustrate how the mechanisms works.

## 5.1 Overview

We have so far assumed that data are stored in clear text in a remote database. The remote database thus must be trusted in correctly enforcing access control policies so only authorized users of the remote database have accesses to sensitive data. Such amount of trust may not be feasible in practice due to the autonomy of a database federation. To address this issue, we apply the over-encryption model [7] to our application by proposing a new key derivation scheme based on secret sharing.

In over-encryption, resources such as tuples are divided into different sets based on

access control lists. All the resources in each set are encrypted individually with the same encryption key. To give a user access to a resource, a token is published for allowing the user to derive the resource's encryption key from the user's own key. For example, the token $t = k_e \oplus h(k_u)$ will allow a user knowing $k_u$ to derive the encryption key $k_e$ [2] (other users knowing $k_e$ cannot derive $k_u$ due to the hash function $h()$). For example, in Figure 1, the hospital-side user Alice is given a secret key by the university-side database. By publishing a token that enables Alice to derive encryption keys, Alice can be given accesses to selected tuples in the hospital-side database.

However, an apparent limitation of the above simple approach is that tuples must be shipped back to the local database for re-encryption in order to grant or revoke users from accessing tuples, which incurs significant communication overhead. The over-encryption approach removes this limitation through a second layer of encryption at the remote database. More precisely, resources are doubly encrypted at the *base encryption layer* (BEL) and the *surface encryption layer* (SEL). Initially, both layers enforce the same access control policies. Upon an update to the policies, such as a grant or revoke, resources will be re-encrypted at the SEL layer by excluding revoked users, and new tokens will be published at the BEL layer for granted users. In any case, no resource needs to be sent back to the BEL layer for re-encryption.

In our application of database federations, the BEL layer encryption is imposed by the local database, and the SEL layer encryption by the remote database. When a user at the local database inserts a tuple, the tuple will be encrypted by the local database first and then sent to the remote database for a second encryption (notice that an integrity stamp will also be appended based on previous discussions). The remote database does not have the BEL encryption keys, so malicious users cannot access the data even if they are in control of the remote database. Only those remote database users who are authorized by the local database can have accesses to the original data. Those authorized users can derive the

encryption key at both layers by using his own secret key with the public tokens provided by the local database.

## 5.2 Secret Sharing-Based Over-Encryption

To introduce our new over-encryption scheme, we first start from a straightforward scheme and point out its limitations; we then extend this simple scheme to two variations, which are to be applied to the BEL and SEL layer, respectively. Each user of the remote database is assigned a *key pair* $\kappa$: $(X, Y)$ where $X$=$K$ and $Y$=$h(K)$ ($h()$ is a hash function). A token $\tau$ is public information which enables the user to derive an encryption key from his/her key pair $\kappa$.

Suppose we have the resource $R$, such as a tuple or an attribute value, encrypted by the key $K_{ab}$, and according to the access matrix list, user A and B can access resource $R$. Assume user A has the key pair $\kappa_a$: $(X_a, Y_a)$ and user B has $\kappa_b$: $(X_b, Y_b)$. They can derive the encryption key $K_{ab}$ by using the secret sharing function $f(x) = \alpha x + K_{ab}$. That is, $Y_a$=$\alpha X_a$+$K_{ab}$ and $Y_b$=$\alpha X_b$+$K_{ab}$. We can pick any $(X_{pab}, Y_{pab})$ such that $Y_{pab}$=$\alpha X_{pab}$+$K_{ab}$, and publish the pair $(X_{pab}, Y_{pab})$ as the token for user A and B so each of them can derive the encryption key $K_{ab}$ using his/her own key pair. We can see that user A and B's key pair and the public token are on (the line corresponding to) the same linear function. Each user can thus use his/her key pair together with the public token to generate this function and the encryption key $K_{ab}$ and then access the resource.

This simple scheme has limitations when one resource is shared by more than two users. To derive a key shared by $n$ users, we should use a $(n-1)$-degree function. However, each single user will need at least $n-1$ public tokens, which is against the very motivation of reducing the number of public tokens. However, a linear function itself is not sufficient, either. For example, suppose user A, B, C can access resource $R$, and each user has a pair of key $\kappa$, which may not be on the same linear function. Even though we can somehow

find a function to satisfy more than two users, consider two sets {A, B, C} and {B, C, D}. From {A, B, C}, we have $f_{abc}(x) = \alpha_{abc}x + K_{abc}$. Now, if we want to share the resource for {B, C, D}, we need another function $f_{bcd}$. However, we already have ($X_b$, $Y_b$) and ($X_c$, $Y_c$) fixed, so ($X_d$, $Y_d$) is not necessarily on the same function and user B, C and D may not be able to access the same resource.

To remove the limitation, we extend the above scheme in two ways. First, suppose we want to grant the access to resource $R$ to user A, B and C. We randomly choose two pairs of keys ($X_a$, $Y_a$) and ($X_b$, $Y_b$) as the *master keys*, and assign them to users A and B. Next, we generate the derivation function based on these two pairs as $f(x) = \alpha_{ab}x + K_{ab}$. We now can randomly choose ($X_c$, $Y_c$) on this function and assign it as a key pair to user C. For other users who need to share the resource $R$, we simply repeat this procedure and choose more points. Second, we randomly pick two users, say A and B, to establish a linear function $f(x) = \alpha x + K_{ab}$ as usual. We call the pairs ($h(K_{ab})$, $h^2(K_{ab})$) the *transfer key*. For another user C sharing the same resource, we use the transfer key together with user C's key pair $\kappa_c$: ($X_c$, $Y_c$) to establish another function $g(x) = \beta x + K_{abc}$ as usual. We can then pick any token ($X_{pabc}$, $Y_{pabc}$) satisfying that $Y_{pabc} = \beta X_{pabc} + K_{abc}$ and use $K_{abc}$ as the encryption key of the resource. User A, B and C all can access that resource by deriving the encryption key through the public token ($X_{pabc}$, $Y_{pabc}$) together with their own key pairs. Figure 9 and figure 10 illustrate this second scheme.

Among the above two extended schemes, the first is more effective in reducing the number of tokens at initialization time, which makes it a better choice at the BEL layer because the initial encryption is handled by local databases. More specifically, at initialization time, we just need one function and one public token for each subset of users sharing the same resources, which does not depend on how many users are in the set. On the other hand, the second scheme is more effective for policy updates (that is, granting or revoking users), which makes it a better choice at the SEL layer where policy update is the main concern.

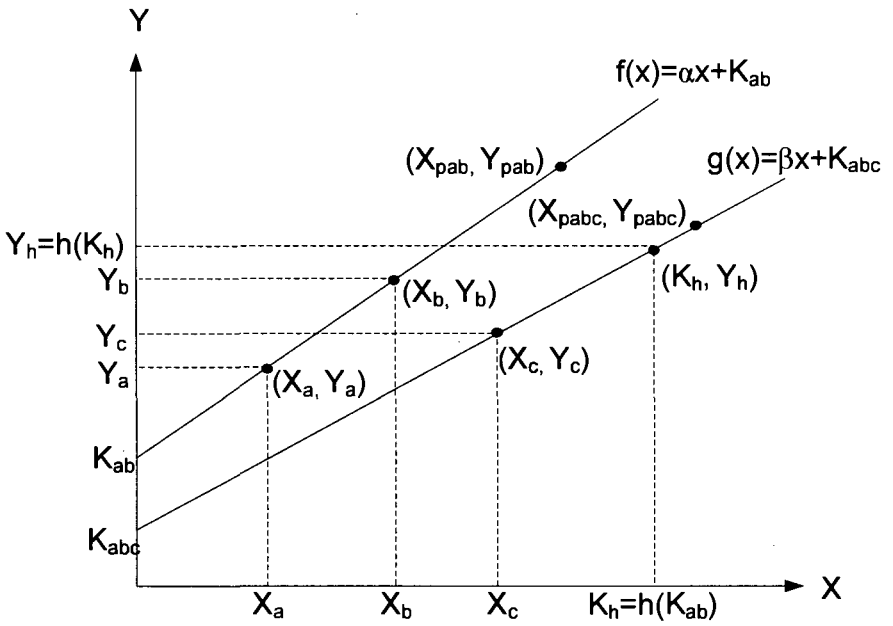



Figure 9: An Example Function of the Second Scheme

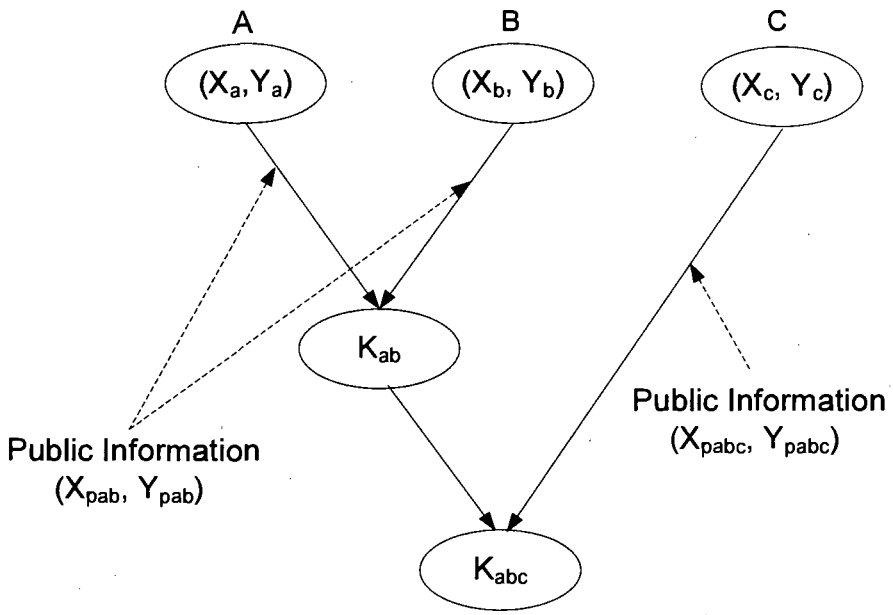



Figure 10: An Example Key Derivation Tree of the Second Scheme

| | | The Number of Tokens | |
|---|---|---|---|
| | | Original Method | Our Method |
| **BEL** | Initialization | $n$ | 1 |
| | Grant | $n+1$ | 2 |
| | Revoke | - | - |
| **SEL** | Initialization | $n$ | $n-1$ |
| | Grant | $n+2$ | $n$ |
| | Revoke | $2n-1$ | $\frac{3n^2-3n-2}{2n}$ |

Table 1: Comparison Between the Two Schemes

For instance, suppose users A, B, C, and D share the same resource, and we would like to revoke user C's access. With the second scheme, we can use A and B's transfer key together with D's key to establish a new function and generate a new encryption key. In this operation, just one token is to be published.

To integrate over-encryption into a database federation, we regard tuples or attribute values as resources, which depends on the desired granularity of access control. Each user of remote databases is assigned secret key pairs as credentials for authorization. Upon inserting or updating data stored in a remote database, the local database will generate encryption keys according to the above scheme for BEL-layer encryption of the data. The local database will also create MHT-based stamps for integrity as aforementioned. The encrypted data are then sent to the remote database, which will doubly encrypt the data at the SEL layer. Public tokens are provided to authorized users to enable them deriving corresponding encryption keys for accessing resources.

Table 1 compares our over-encryption scheme with the original method [7] in both BEL and SEL layers. We consider the number of tokens required in three situation: initialization, granting, and revoking with $n$ users.

## 5.3 Query over Encrypted Database

The use of encryption enables distributed authorization but it may complicate query processing when selection conditions involve encrypted data. Queries over encrypted data can be supported through existing techniques [34]. Alternatively, the local database (or user-side application for users at the remote database) may first encrypt attribute values involved in a query at the BEL layer before sending the query to the remote database. When remote database receives the query, it first decrypts the doubly encrypted data using the SEL layer encryption keys and then sends the result to the local database. The local database can then obtain the original data using the BEL layer encryption key. One complication is that resources accessible to a user may be encrypted with different keys and the local database does not know which of the keys corresponds to the particular resource being requested. In such a case, the local database must create multiple versions of the same query using different encryption keys. For example, on university side, suppose user Bob poses a selection-projection query for the disease attribute that store on the remote database, let's say `SELECT disease FROM table2 WHERE name='Bob'`, according to our approach, university-side database will send the query to remote database. However, because of the new feature of our model, university-side database should encrypt the value that appears in the user's query by the BEL encryption keys, and execute the query by using this encrypted value instead of the original one [34]. Since university-side database does not know which resources will be the results, it can not determine which encryption key should be used. University-side database will find all the derived keys for user $Bob$ according to access control matrix table 2. we can know that user $Bob$ can access the resources $s1$, $s2$, $s3$, $s4$, $s5$, $s8$ and $s9$, which means he has three keys at BEL layer, his own private key with BEL encryption keys $K_{B1}$ and $K_{B3}$. Now there will be three encrypted valued, they are $e_1$=$E_{K_b}('Bob')$, $e_2$=$E_{K_{B1}}('Bob')$ and $e_3$=$E_{K_{B3}}('Bob')$, where $E()$ is an encryption function. University-side database will replace the value 'Bob' in the query by these encrypted

| | s1 | s2 | s3 | s4 | s5 | s6 | s7 | s8 | s9 | s10 |
|---|---|---|---|---|---|---|---|---|---|---|
| **Alice** | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 |
| **Bob** | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| **Carl** | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| **David** | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| **Elaine** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |

Table 2: An Access Control Matrix

values and modify the query to three queries:

`SELECT disease FROM table2 WHERE name='`$e_1$`',`

`SELECT disease FROM table2 WHERE name='`$e_2$`',`

`SELECT disease FROM table2 WHERE name='`$e_3$`'.`

Remote database will answer these queries and send the relevant results back to university. After decrypting with the corresponding BEL encryption keys and filtering the result according to the policy of user *Bob*, university-side database outputs the query result to *Bob*. Since the result should only be encrypted by one of three keys, only one query will get the result. To simplify the query request on university sides, we consider that all data are updated, and no mismatch stamps occurs in this process.

## 5.4 A Case Study

Following the example in Figure 1, Table 2 shows an access control matrix of five users and ten resources in the hospital-side database. The confidentiality requirement is modeled in the access control matrix with each 1-entry representing a positive authorization and each 0-entry a prohibition of accesses. Each column of the table thus shows the status of a resource accessible to users. In our example, we assume the attribute $CONDITION$ is the sensitive resource and should only be accessible to authorized users.

According to the above access control matrix, Figure 11 and 12 depict the initial key derivation structure at the BEL and SEL layers. For example, an $CONDITION$ attribute

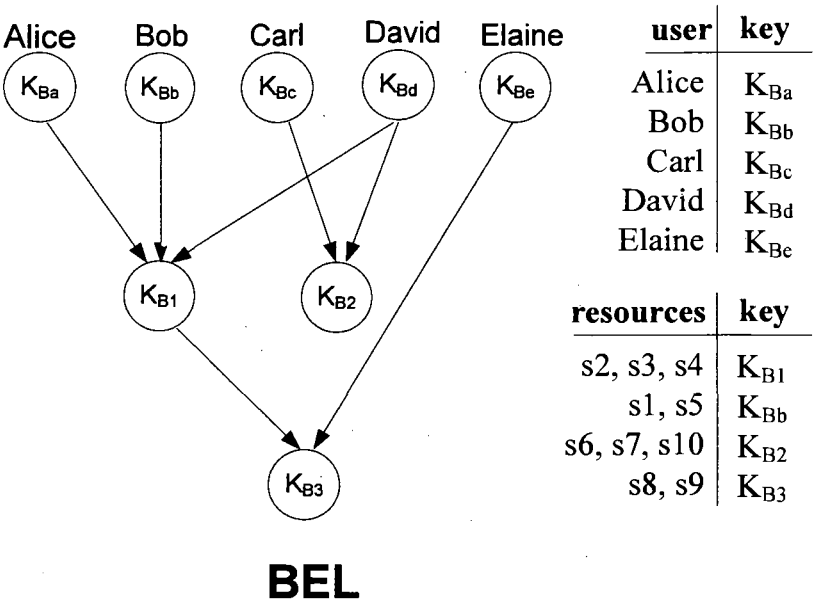



Figure 11: Initial Key Derivation on the BEL Layer

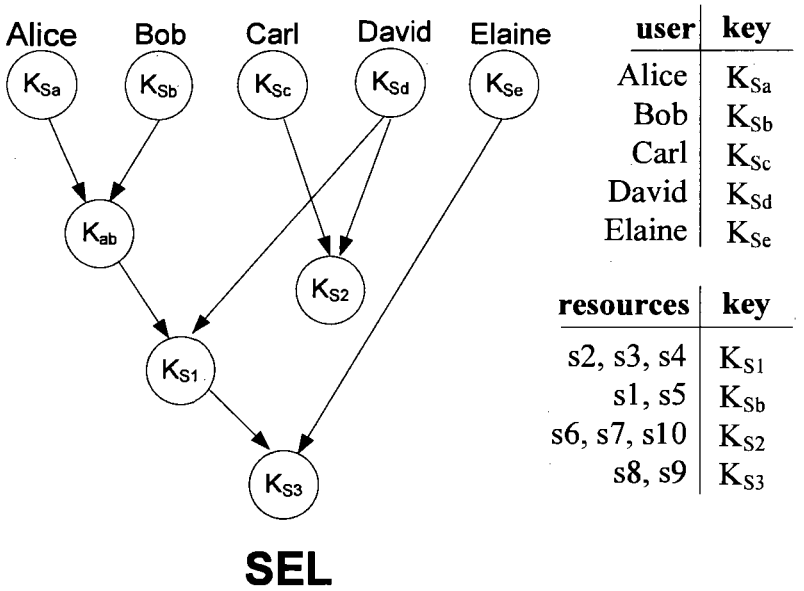
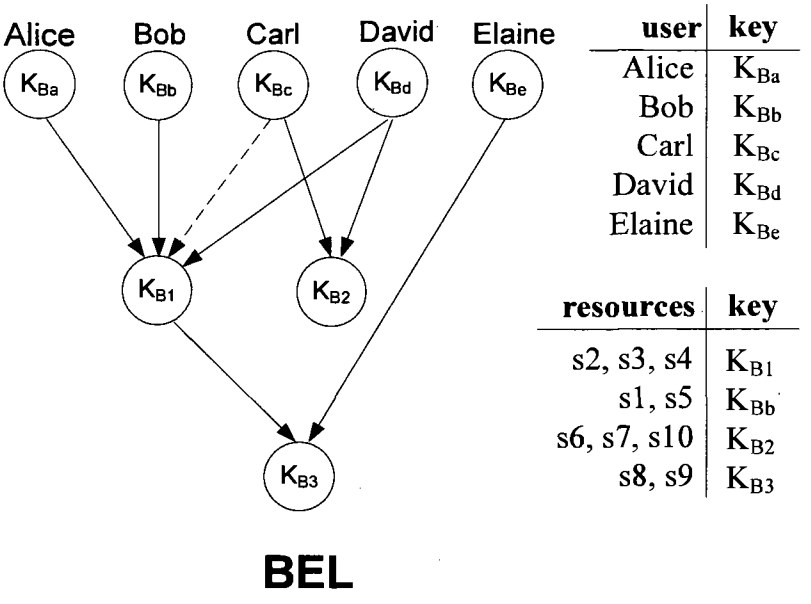



Figure 12: Initial Key Derivation on the SEL Layer

$s4$ is first encrypted using the BEL-layer encryption key $K_{B1}$, which is derived by the local database from the key pairs of users *Alice*, *Bob* and *David*. The encrypted resource is sent to the remote database. The remote database again encrypts the resource using the SEL layer encryption key $K_{S1}$. Now the resource has been encrypted using two different keys $K_{B1}$ and $K_{S1}$. Since user Alice is authorized to access the resource $s4$, her user-side application can derive both encryption keys using her own key pair and the public tokens, and then decrypt data to obtain the original results.

Suppose Alice is also authorized to update the resource $s4$. Alice's user-side application will first encrypt the new value using the BEL layer encryption key $K_{B1}$ and then send the result to the remote database. The remote database will over-encrypt the received value by using the SEL encryption key $K_{S1}$. Since the received value is already encrypted using the BEL layer encryption key, malicious users in control of the remote database cannot access the updated value. Neither can those users skip or alter the update because our integrity mechanisms, as described in previous sections, will detect such misbehaviors.

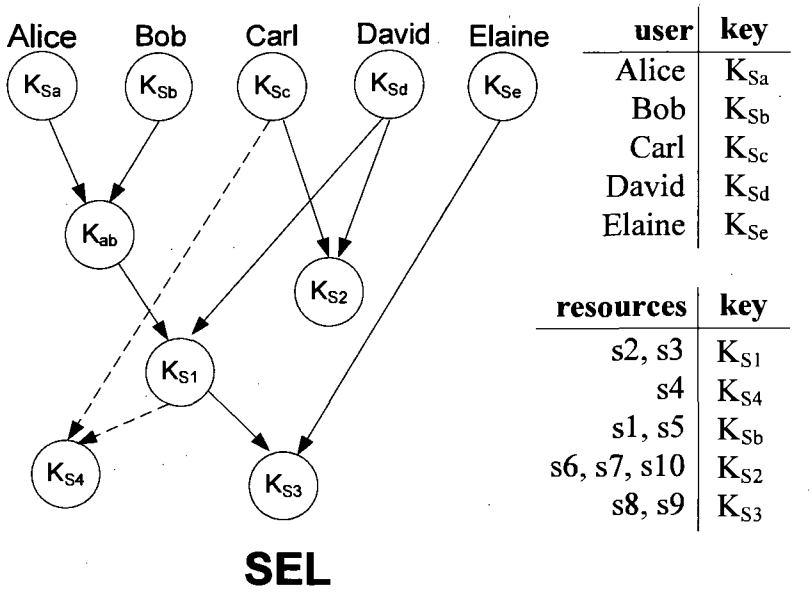
While a data update requires re-encryption, an update of access control policies, such as a grant or revoke, can be efficiently processed through over-encryption. For example, if local database wants to assign resource $s4$ to user $Carl$, it should link $s4$'s encryption key $K_{B1}$ to user $Carl$'s secret key by publishing a new token at the BEL layer; the remote database will derive a new encryption key to re-encrypt the resource (which is still encrypted by the same BEL key $K_{B1}$). Therefore, user $Carl$ can now access the resource $s4$, but not $s2$ or $s3$. Figure 13 and 14 show the key derivation structure for the BEL and SEL layers. For a revoke, the local database will not do anything but the remote database re-encrypts resources using new encryption keys.




| user | key |
|---|---|
| Alice | $K_{Ba}$ |
| Bob | $K_{Bb}$ |
| Carl | $K_{Bc}$ |
| David | $K_{Bd}$ |
| Elaine | $K_{Be}$ |

| resources | key |
|---|---|
| s2, s3, s4 | $K_{B1}$ |
| s1, s5 | $K_{Bb}$ |
| s6, s7, s10 | $K_{B2}$ |
| s8, s9 | $K_{B3}$ |

Figure 13: Key Derivation On BEL for Granting User Carl the Access to $s4$




| user | key |
|---|---|
| Alice | $K_{Sa}$ |
| Bob | $K_{Sb}$ |
| Carl | $K_{Sc}$ |
| David | $K_{Sd}$ |
| Elaine | $K_{Se}$ |

| resources | key |
|---|---|
| s2, s3 | $K_{S1}$ |
| s4 | $K_{S4}$ |
| s1, s5 | $K_{Sb}$ |
| s6, s7, s10 | $K_{S2}$ |
| s8, s9 | $K_{S3}$ |

Figure 14: Key Derivation On SEL for Granting User Carl the Access to $s4$

# Chapter 6

# Implementation and Experiments

We tested the performance of our proposed techniques on machines equipped with Intel Pentium M 1.80GHz processor, 1024MB RAM, Windows XP operating system, and Oracle 10g DBMS. The main objective of the experiments is to compare the performance of different caching schemes, to find the optimal subtable size, and to study the performance overhead of over-encryption. As a proof of concept, we have also implemented a demo system as a fictitious web application that integrates a university's web portal with a hospital's database. The web application is written in PHP version 5.2.5 and runs on the Apache 2.0 web server and Mysql 5.0 database system.

## 6.1 Static and Dynamic Caching

Figure 15 shows the computational cost of updating a tuple in databases of different sizes, when all non-leaf values are cached. We can see that at the cost of extra storage, only a small performance overhead is incurred in updating tuples by recomputing cryptographic stamps under the static cache scheme, in contrast to updating tuples without recomputing any stamps (that is, ignoring the security requirement). On the other hand, recomputing the stamps from scratch is proved to be costly. In all cases, the performance overhead increases

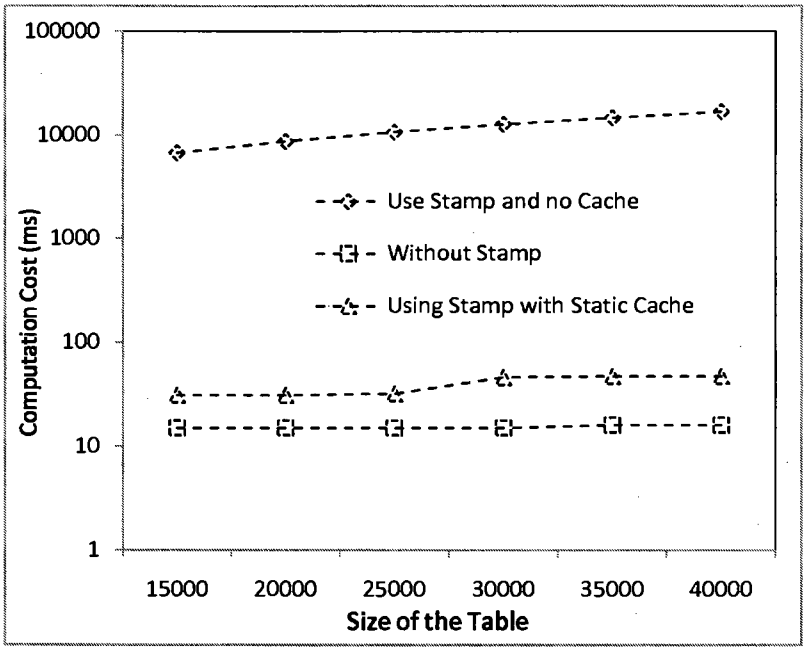



Figure 15: The Performance of Static Cache Scheme

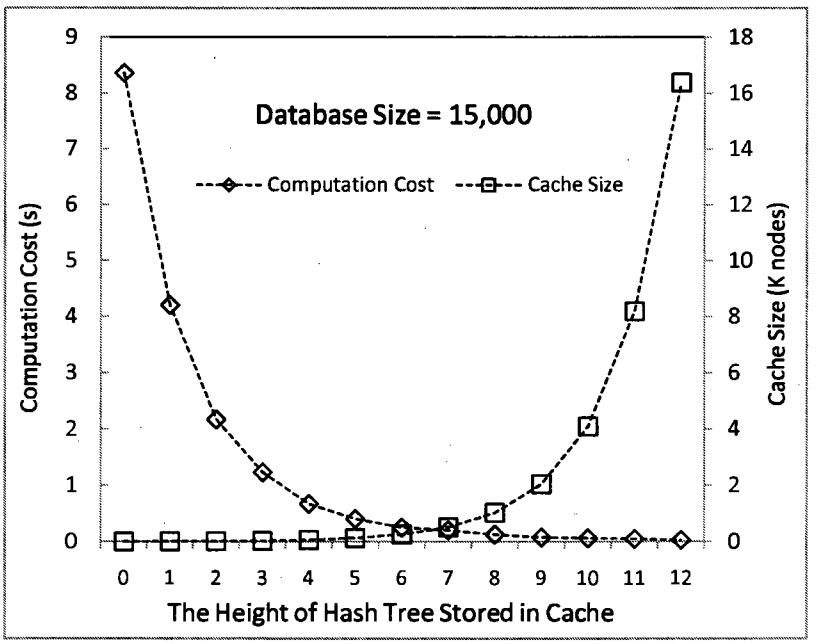



Figure 16: The Static Cahce Scheme Performance with Different Cache Size

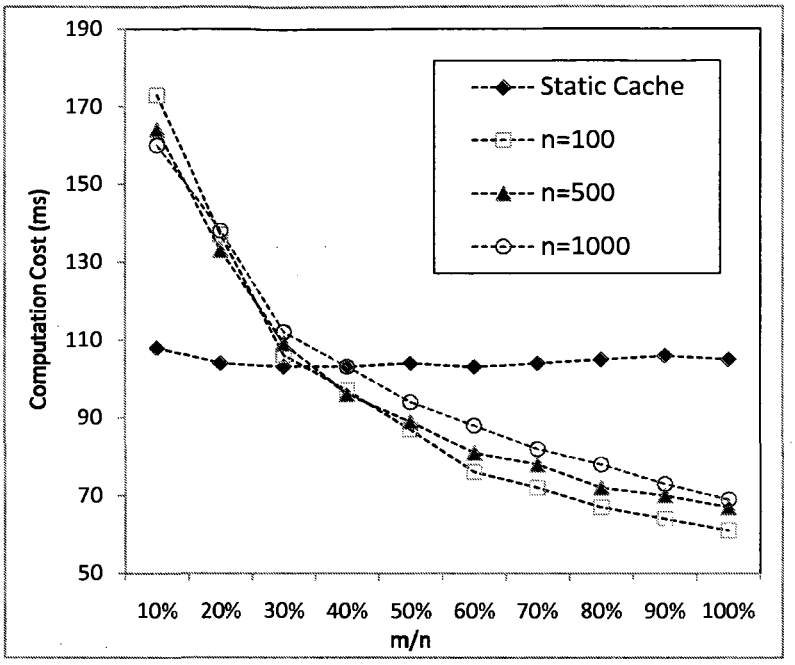



Figure 17: The Performance of LFU Dynamic Cache Schemes

as the table size increases because a larger table means the stamps will be computed over larger MHTs.

Figure 16 shows both the storage requirement and the performance of static caches of different sizes, which all hold a top portion of the MHT. We update one tuple in a database with 15,000 tuples with the height of MHT being 12. We reduce the size of caches by removing each level of the MHT in a bottom-up fashion, as reflected by different heights on the $x$-axis. The curve with square dots shows the number of values in the cache, that is, the storage requirement for caching. The other curve shows the computational cost. We can see the overall performance is optimal when the height of MHTs in the cache is between 3 and 9 where both the storage requirement and the computational cost are relatively low.

Figure 17 and 18 compare the computational cost of different dynamic cache schemes with that of the static cache scheme under the same storage limitation. The database size is 15,000 tuples, and the cache is limited to store only 500 MHT nodes. To simulate queries

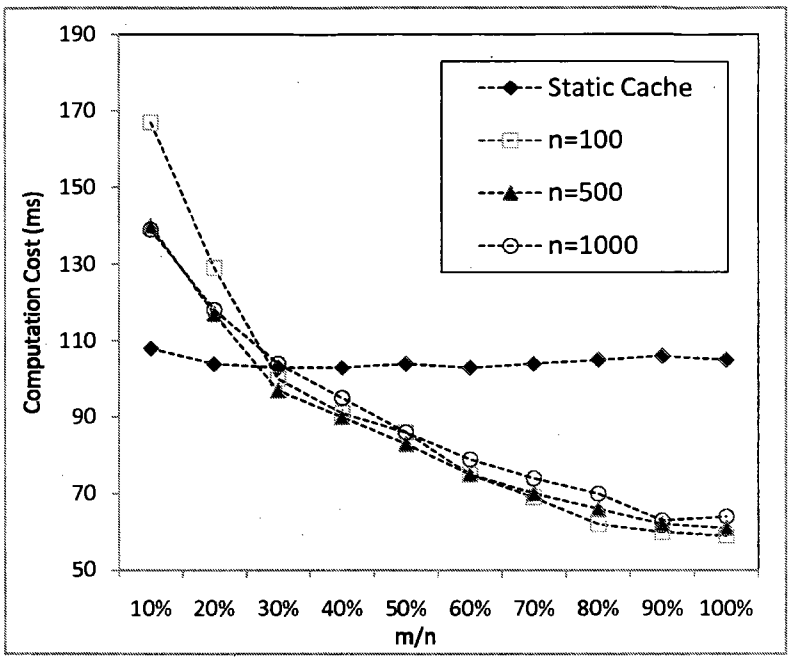



Figure 18: The Performance of LRU Dynamic Cache Schemes

that select adjacent tuples, we randomly pick tuples inside a window and repeat the experiment over different sizes of the window. In Figure 17 and 18, $n$ is the size of this window, and $m$ the number of tuples involved in a query, the $x$-axis is the percentage of to-be-updated values inside the window. There is only a negligible difference between different cache schemes. Figure 17 shows the LFU (Least Frequently Used) cache scheme and Figure 18 shows the LRU (Least Recently Used) scheme. We can see that as more values are updated, the performance of dynamic caching will improve as the cache hit rate will increase. There is only a negligible difference between the two different cache schemes. The size of the window only has a small effect on this result, indicating that the dynamic cache is generally helpful as long as subsequent queries focus on adjacent tuples.

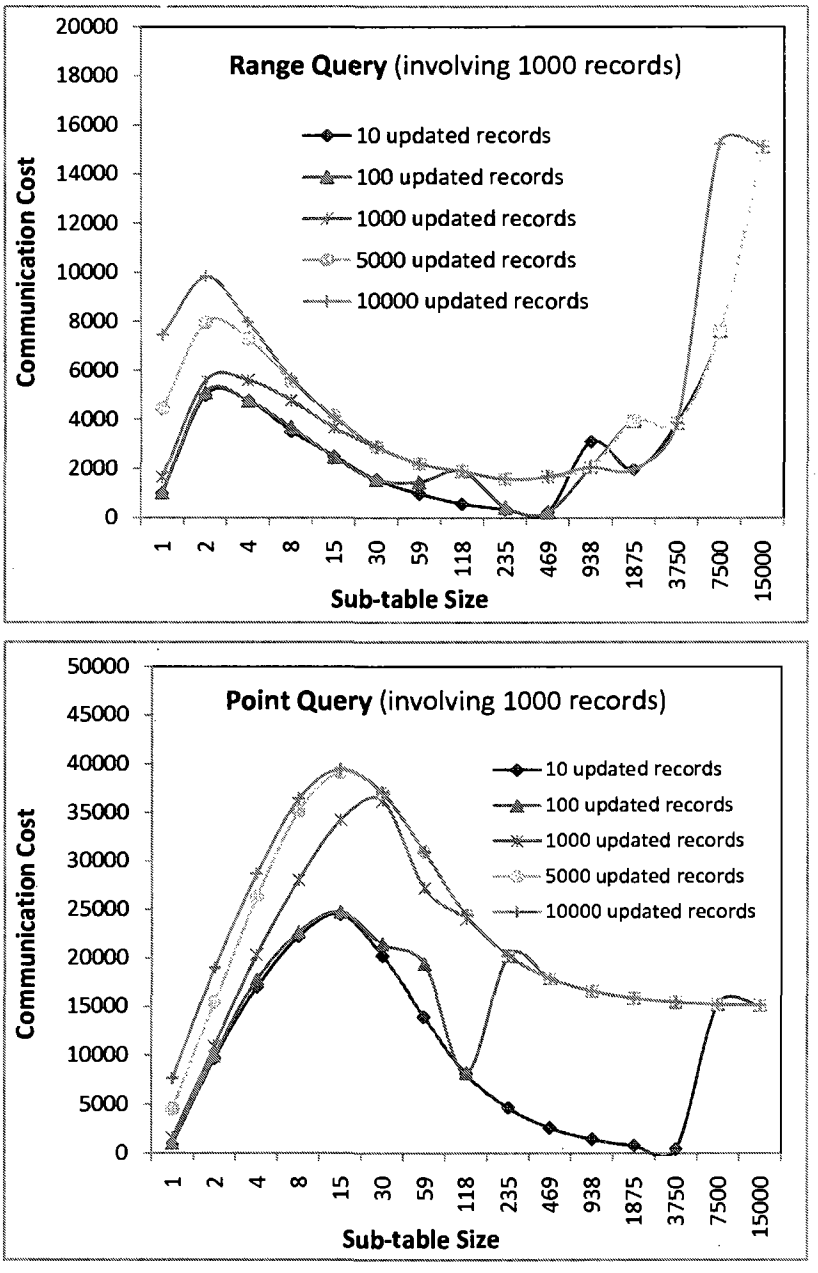



Figure 19: The Communication Cost under Different Sizes of Subtables

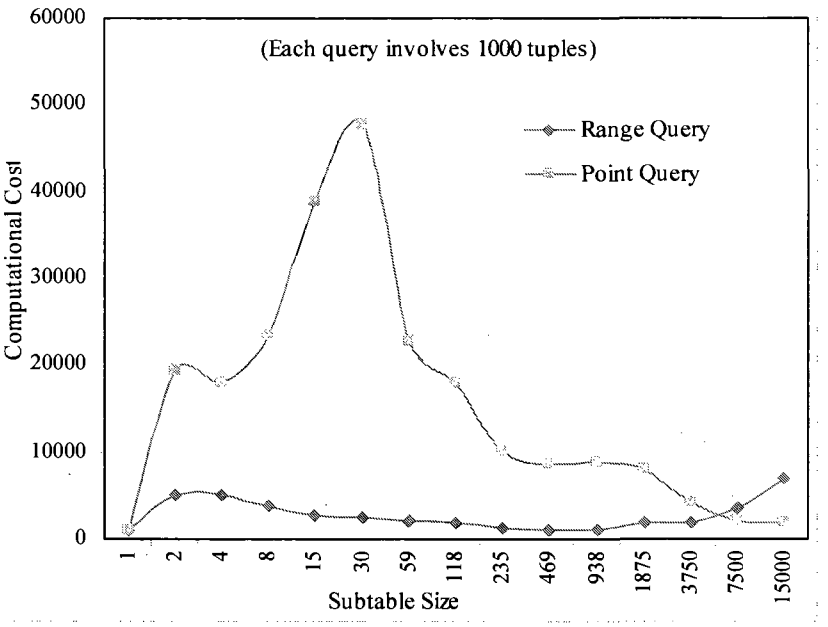



Figure 20: The Computational Cost of Partitioning

## 6.2 Table Partitioning

To study the communication and computational cost under different sizes of subtables, we execute selection queries each of which involves a random set of 1000 tuples in a database with totally 15000 tuples while we randomly update tuples. In Figure 19, the $y$-axis indicates the number of stamps that need to be sent to the local database as proofs, which comprise the major factor of the communication cost. Different lines correspond to different number of updated tuples. The $x$-axis reflects different sizes of subtables, which increases in the power of two. We consider two types of selection queries. First, range queries involves continuous tuples with respect to the partitioning process used to obtain subtables. Second, point queries involve random tuples chosen uniformly from the whole table regardless of the partitioning process.

The upper side chart in Figure 19 shows that for range queries, there exists an optimal subtable size with the lowest communication cost around the middle of the $x$-axis. On the other hand, the following chart shows that for point queries, the communication cost

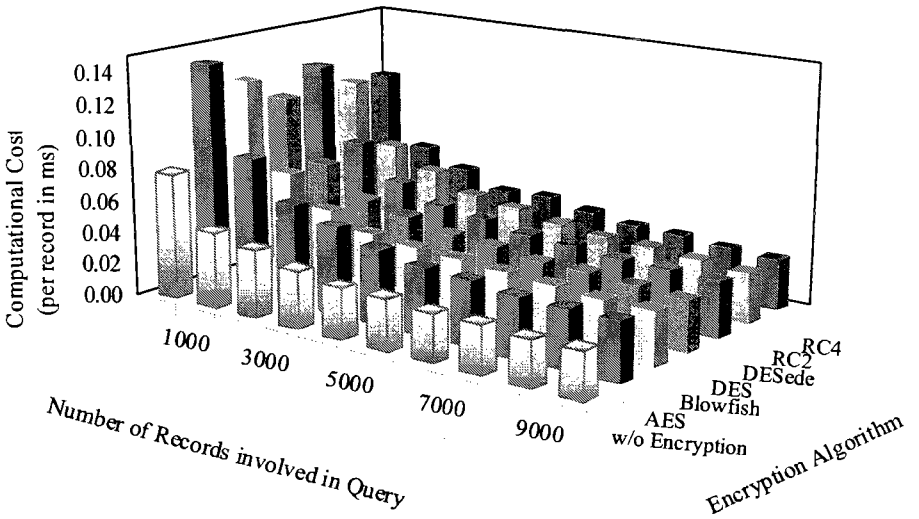



Figure 21: The Computational Cost of Search on Encrypted Database

is the lowest when each tuple itself is a subtable, or when the table is not partitioned. However, considering the fact that smaller subtables imply extra storage cost for stamps over columns, partitioning turns out to be an effective solution for range queries but not for point queries.

Figure 20 shows the computational cost for both range queries and point queries in different sizes of subtables. We can see the trend is similar to that of the communication cost. The reason is that with smaller subtables, although the computational cost of each subtable is lower, a greater number of subtables will be involved in a query.

## 6.3 Over Encryption with Caching

We study the performance overhead of over-encryption under different popular encryption algorithms, including AES, Blowfish, DES, DESede, RC2, and RC4. Figure 21 shows the computational cost per tuple for selection queries involving different number of tuples. The result shows that in order to provide confidentiality, the proposed over-encryption mechanism only incurs a small performance overhead. We also evaluate the computational

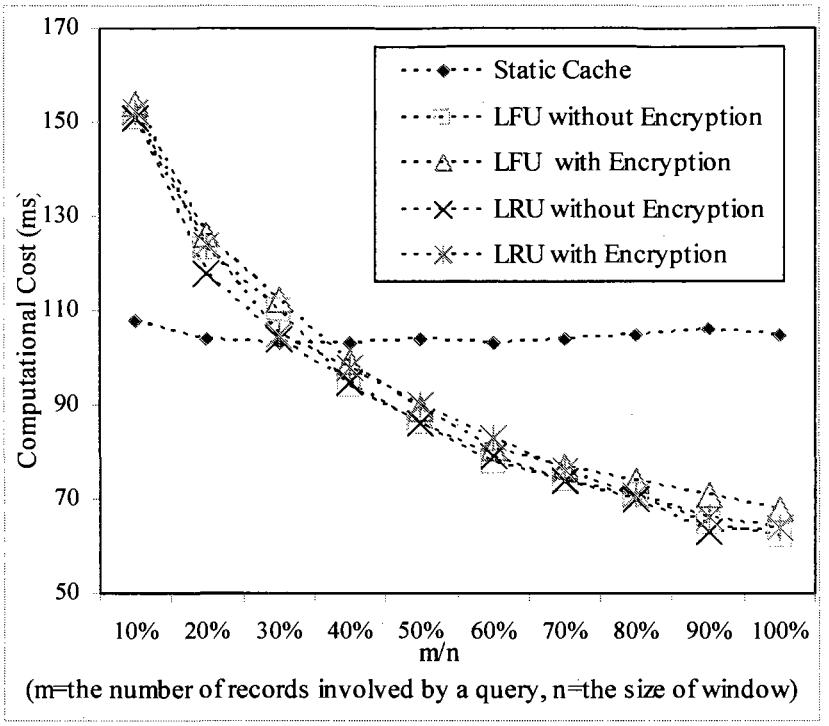



Figure 22: The Computational Cost of Over-Encryption with Dynamic Cache Scheme

cost of updates under different caching schemes with or without over-encryption in place. Figure 22 compares the computational cost of updates in encrypted and non-encrypted databases under the LFU and LRU-based dynamic caching schemes. Figure 23 compares the overall computational cost of updating different number of tuples under different encryption algorithms. Theses charts both show that the cost of over-encryption is acceptable and the per-tuple cost will decrease when more tuples are involved in a query.

## 6.4 A Demo System

We implemented a web application that provides a fictitious university's users with direct accesses to their medical records hosted in a hospital database. The web application implements the aforementioned security mechanisms. For simplicity, we assume two levels of users' privileges are enforced in the system, and each user can access records according to

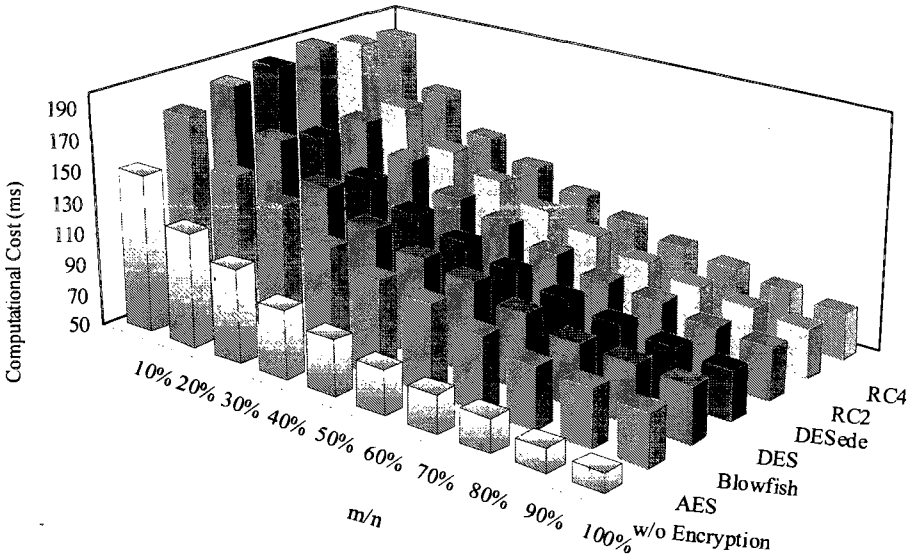



Figure 23: The Computational Cost of Over-Encryption with Different Algorithms

specific policies.

Figure 24 shows the web interface when a user check his status on the university side. The user with low privilege can only query the database according to the policy and the high user has the ability to modify the users' records. Figure 25 and Figure 26 are the web interfaces on the hospital side. Figure 25 displays all records in the hospital database, and in figure 26 there is one modified value with mismatched stamps which are marked in red color.

When a user on the university site sends a query, our web system demonstrates the verification process in Figure 27 step by step. After the university database receives results from the hospital, it will check the policy to determine whether the user has the privilege to read this result. If yes, the web system will show the page in Figure 28 and then to recompute and verify the stamps of the returned result. Otherwise a warning page will be presented in Figure 29.

The process of rebuilding and verifying stamps will be started if the user has privilege to access the results according to the policy. Figure 30 and Figure 31 show two situations

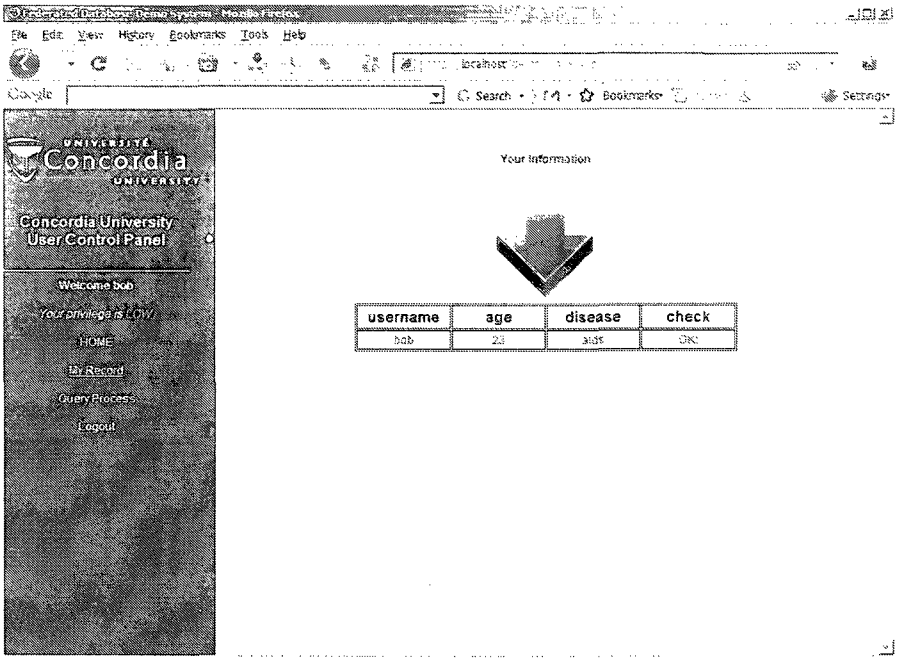



Figure 24: User Interface on the University Side

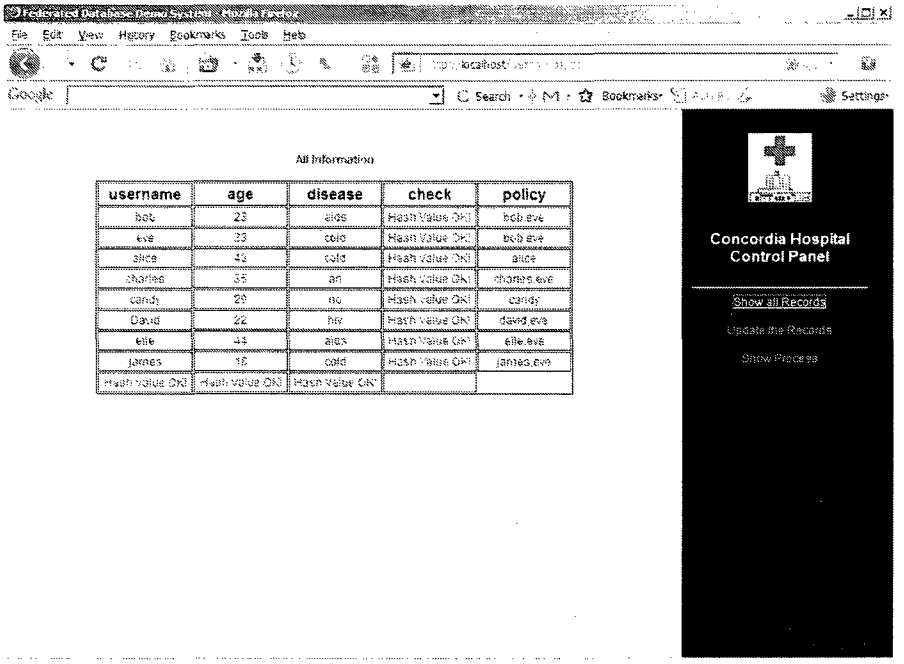



Figure 25: User Interface on the Hospital Side

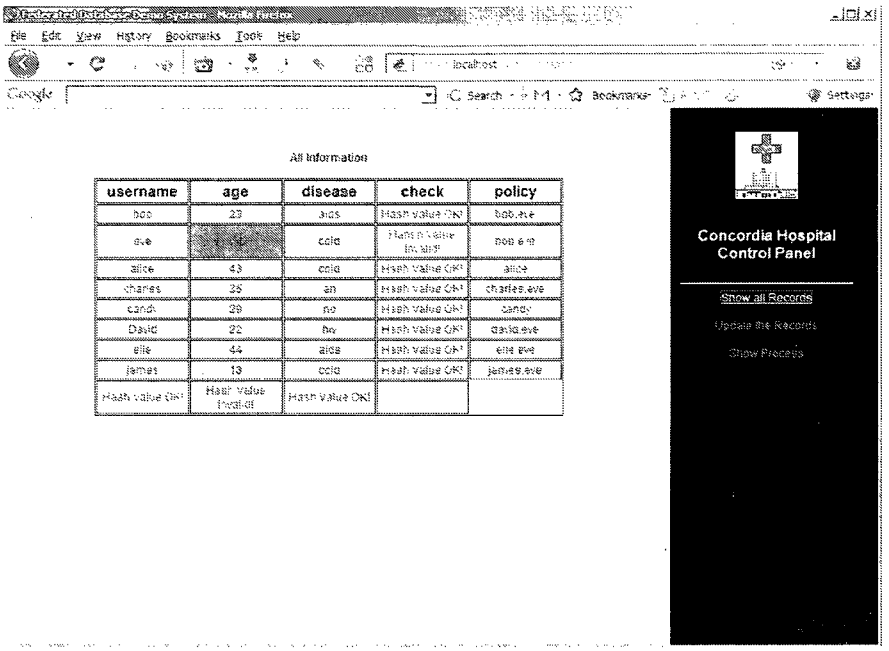



Figure 26: User Interface on the Hospital Side with Mismatched Stamps

where the returned results are involved in the databases shown in Figure 25 and Figure 26, respectively.


Figure 27: The Query Verification Process

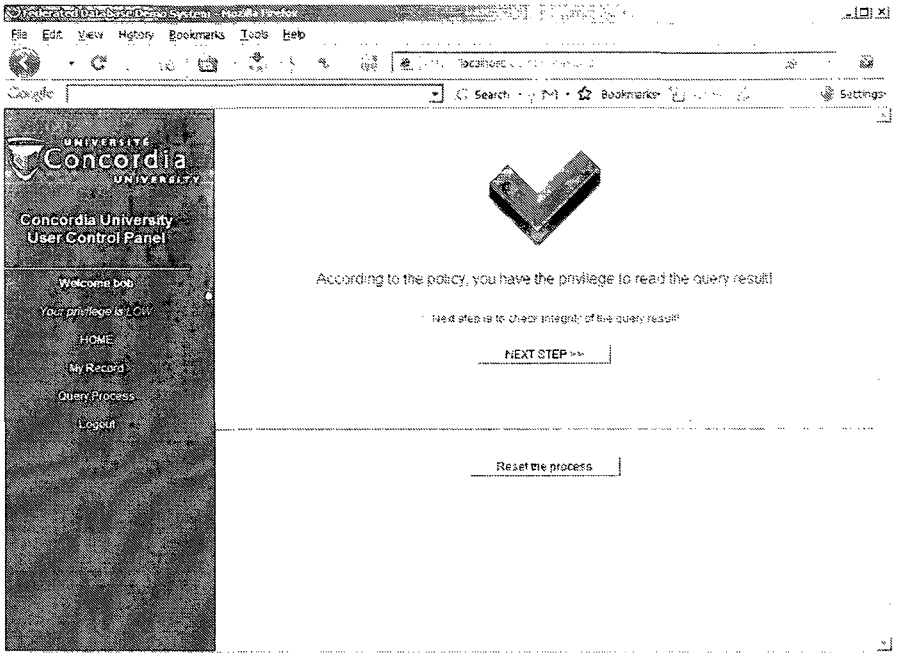



Figure 28: The Result of Policy Checking Passes

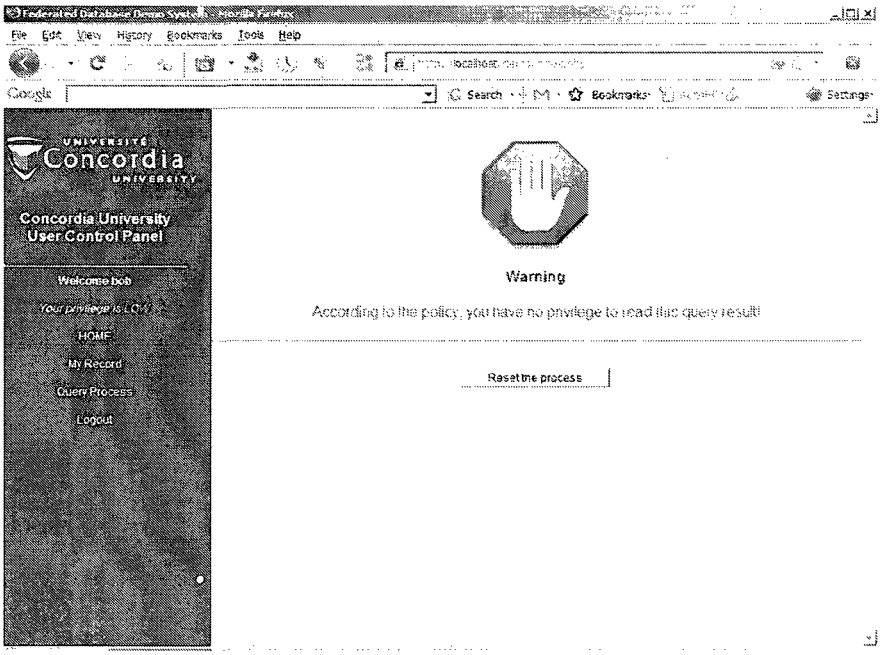



Figure 29: The Result of Policy Checking Fails

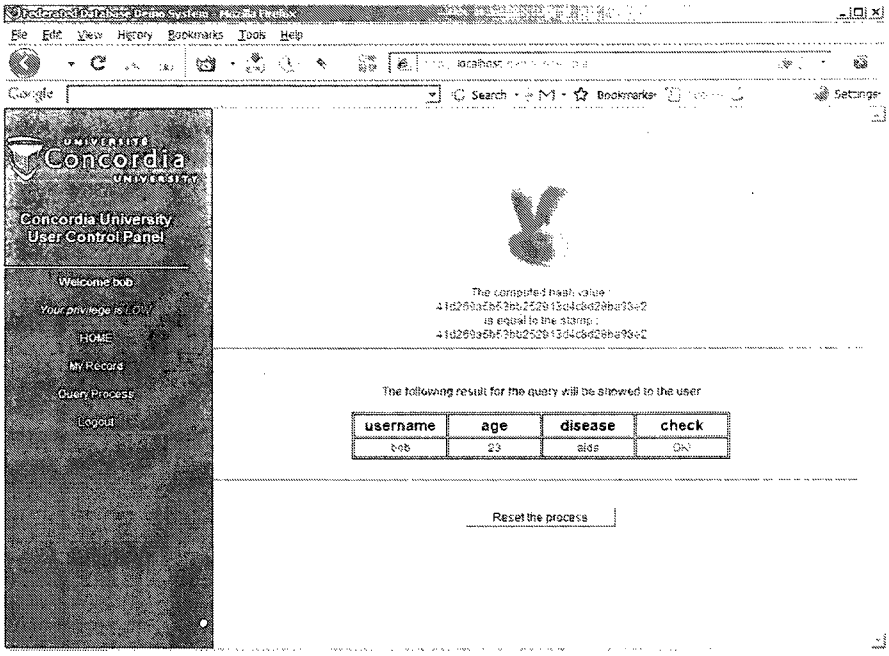



Figure 30: Stamp Verification

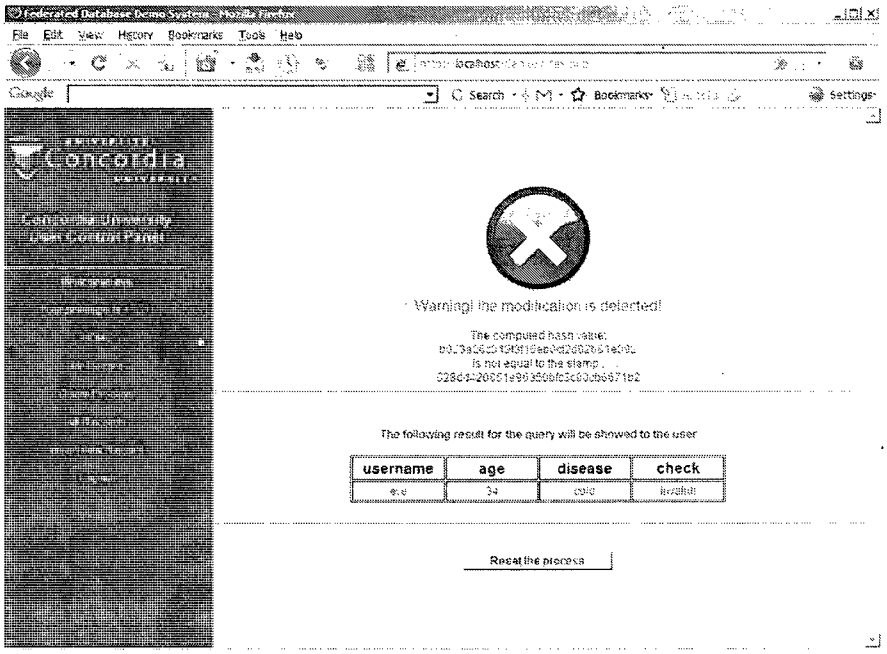



Figure 31: Stamp Verification Failed

# Chapter 7

# Conclusion

We have addressed the integrity and confidentiality issues in the context of a loosely coupled database federation. Unlike centralized databases or tightly coupled database federations, a loosely coupled database federation lacks the central authority required by traditional authorization models. How to protect the integrity and confidentiality of data stored in remote databases thus becomes a challenging issue. We provided a solution that is composed of an architecture and both integrity and confidentiality mechanisms.

First, we revisited the integrity lock architecture originally proposed for multilevel databases. We showed that the architecture provides a natural solution to the distributed authorization in loosely coupled database federations. The architecture not only allowed a local database to take full control of authorization decisions but also enabled fine-grained and data dependent authorizations.

Second, we proposed a three-stage procedure as the integrity mechanism of the integrity lock architecture. The procedure extended techniques in outsourced databases to remove their limitations in dealing with frequent updates. As a result, modifications of data could be detected and localized when they are involved in queries. The remote database would then provide log entries as proofs of the legitimacy of such modifications. As a result, legitimate updates were accommodated while unauthorized modifications were excluded

from query results.

Third, we proposed a new over-encryption scheme as the confidentiality mechanism. We replaced the key derivation function with our new scheme based on secret sharing to reduce the number of required public tokens. The over-encryption scheme could allow updates of access control policies without shipping data back to the local database for re-encryption. We illustrated the proposed scheme through a case study.

Finally, we evaluated several aspects of the proposed solution through implementation and experiments. We compared the performance of different caching schemes, which leads to the conclusion that ensuring data integrity using the proposed mechanisms incurs an acceptable performance overhead if appropriate caching schemes are used; different caching schemes are suitable for different types of queries. We studied the effect of partitioning tables into subtables of different sizes. The conclusion is that partitioning tables helps the most with respect to range queries while it is not as effective for point queries. Finally, we studied the performance overhead of over-encryption in terms of both selection queries and updates. The conclusion is that such overhead is reasonably low regardless of the encryption algorithms being used.

In the broad context of loosely coupled database federations, different security issues may arise when the local and remote databases interact in different ways, or when the trust placed upon remote databases is of a different degree or nature. Our future work will continue to explore security issues in such situations. Other future directions include the optimization of security mechanisms in the presence of concurrent accesses and the issue of query processing in the presence of encryption techniques.

# Bibliography

[1] Paul Ammann, Sushil Jajodia, and Peng Liu. Recovery from malicious transactions. *IEEE Transactions on Knowledge and Data Engineering*, 14(5):1167–1185, 2002.

[2] Mikhail J. Atallah, Keith B. Frikken, and Marina Blanton. Dynamic and efficient key management for access hierarchies. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*, pages 190–202, New York, NY, USA, 2005. ACM.

[3] Philippe Beguin and Antonella Cresti. General short computational secret sharing schemes. In *Advances in Cryptology-EUROCRYPT'95*, pages 194–208. Springer Berlin and Heidelberg, 1995.

[4] Amos Beimel and Benny Chor. Secret sharing with public reconstruction. In *IEEE*, pages 1887–1896. IEEE, 1998.

[5] Jonscher D. and K.R. Dittrich. Argos - a configurable access control system for interoperable environments. In *IFIP Workshop on Database Security*, pages 43–60, 1995.

[6] S. Dawson, P. Samarati, S. De Capitani di Vimercati, P. Lincoln, G. Wiederhold, M. Bilello, J. Akella, and Y. Tan. Secure access wrapper: Mediating security between heterogeneous databases. In *DARPA Information Survivability Conference and Exposition (DISCEX)*, 2000.

[7] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Over-encryption: Management of access control evolution on outsourced data. In *VLDB*, 2007.

[8] D.E. Denning. Cryptographic checksums for multilevel database security. In *Proc. of the 1984 IEEE Symposium on Security and Privacy*, pages 52–61, 1984.

[9] D.E. Denning. Commutative filters for reducing inference threats in multilevel database systems. In *Proc. of the 1985 IEEE Symposium on Security and Privacy*, pages 134–146, 1985.

[10] Premkumar T. Devanbu, Michael Gertz, Chip Martel, and Stuart G. Stubblebine. Authentic third-party data publication. In *IFIP 11.3 Working Conference on Database Security*, pages 101–112, 2000.

[11] R. Graubart. The integrity-lock approach to secure database management. In *Proc. of the 1984 IEEE Symposium on Security and Privacy*, page 62, 1984.

[12] E. Gudes and M.S. Olivier. Security policies in replicated and autonomous databases. In *Proc. of the IFIP TC11 WG 11.3 Twelfth International Conference on Database Security*, pages 93–107, 1998.

[13] H. Guo, Y. Li, A. Liu, and S. Jajodia. A fragile watermarking scheme for detecting malicious modifications of database relations. *Information Sciences*, 176(10):1350–1378, 2006.

[14] Xiangji Huang, Qingsong Yao, and Aijun An. Applying language modeling to session identification from database trace logs. *Knowl. Inf. Syst.*, 10(4):473–504, 2006.

[15] S. Jajodia and R.S. Sandhu. Toward a multilevel secure relational data model. In M.D. Abrams, S. Jajodia, and H.J. Podell, editors, *Information Security An Integrated Collection of Essays*, pages 460–492. IEEE Computer Society Press, 1995.

[16] S. Jajodia, R.S. Sandhu, and B.T. Blaustein. Solutions to the polyinstantiation problem. In M.D. Abrams, S. Jajodia, and H.J. Podell, editors, *Information Security An Integrated Collection of Essays*, pages 493–530. IEEE Computer Society Press, 1995.

[17] Markus Jakobsson, Tom Leighton, Silvio Micali, and Michael Szydlo. Fractal merkle tree representation and traversal. In *In Proc. of RSA Cryptographersąŕ Track*, pages 314–326. Springer, 2003.

[18] J.Crampton, K.Martin, and P.Samarati. On key assignment for hierarchical access control. In *19th IEEE CSFW'06*, 2006.

[19] D. Jonscher and K.R. Dittrich. An approach for building secure database federations. In *Proc. of the 20th VLDB Very Large Data Base Conference*, pages 24–35, 1994.

[20] Marek Karpinski and Yakov Nekrich. Optimal trade-off for merkle tree traversal. In *ICETE*, pages 275–282, 2005.

[21] Yongdae Kim, Adrian Perrig, and Gene Tsudik. Tree-based group key agreement. Technical Report 2002/009, 2002.

[22] Feifei Li, Marios Hadjieleftheriou, George Kollios, and Leonid Reyzin. Dynamic authenticated index structures for outsourced databases. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 121–132, New York, NY, USA, 2006. ACM Press.

[23] C. Meadows. The integrity lock architecture and its application to message systems: Reducing covert channels. In *Proc. of the 1987 IEEE Symposium on Security and Privacy*, page 212, 1987.

[24] R.C. Merkle. A certified digital signature. In *Proc. of the Advances in Cryptology (CRYPTO'89)*, pages 218–238, 1989.

[25] E. Mykletun and G. Tsudik. Aggregation queries in the database-as-a-service model. In *Proc. of the 2006 IFIP 11.3 Working Conference on Database Security*, 2006.

[26] Einar Mykletun, Maithili Narasimha, and Gene Tsudik. Authentication and integrity in outsourced databases. *ACM Transactions on Storage (TOS)*, 2(2):107–138, 2006.

[27] N.Alon, Z.Galil, and M.Yung. Efficient dynamic-resharing "verifiable secret sharing" against mobile adversary. In *Algorithms-ESA'95*, pages 523–537. Springer Berlin and Heidelberg, 1995.

[28] L. Notargiacomo. Architectures for mls database management systems. In M.D. Abrams, S. Jajodia, and H.J. Podell, editors, *Information Security An Integrated Collection of Essays*, pages 439–459. IEEE Computer Society Press, 1995.

[29] HweeHwa Pang, Arpit Jain, Krithi Ramamritham, and Kian-Lee Tan. Verifying completeness of relational query results in data publishing. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 407–418, New York, NY, USA, 2005. ACM Press.

[30] S.Akl and P.Taylor. Cryptographic solution to a problem of access control in a hierarchy. In *ACM TOCS*, pages 1(3):239–248, 1983.

[31] Adi Shamir. How to share a secret. In *Communications of the ACM*, pages 612–613, New York, NY, USA, 1979. ACM.

[32] A.P. Sheth and J.A. Larson. Federated database system for managing distributed, hetergeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, 1990.

[33] Lingyu Wang Shuai Liu, Wei Li. Towards efficient over-encryption in outsourced databases using secret sharing. In *Proc. The 2nd IFIP International Conference on New Technologies, Mobility and Security (NTMS 2008)*, pages 1–5, 2008.

[34] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *IEEE Symposium on Security and Privacy*, pages 44–55, 2000.

[35] Michael Szydlo. Merkle tree traversal in log space and time. In *In Eurocrypt 2004, LNCS*, pages 541–554. Springer-Verlag, 2004.

[36] Hai Wang and Peng Liu. Modeling and evaluating the survivability of an intrusion tolerant database system. In *ESORICS*, 2006.

[37] Xuxin Xu, Lingyu Wang, Amr M. Youssef, and Bo Zhu. Preventing collusion attacks on the one-way function tree (oft) scheme. In *ACNS*, pages 177–193, 2007.

[38] J. Yang, D. Wijesekera, and S. Jajodia. Subject switching algorithms for access control in federated databases. In *Proc. of 15th IFIP WG11.3 Working Conference on Database and Application Security*, pages 199–204, 2001.