Chapter 1
Enabling Collaborative Data Authorization Between Enterprise Clouds

Meixing Le, Krishna Kant, Sushil Jajodia
Center for Secure Information Systems
George Mason University, Fairfax, VA
{*mlep, kkant, jajodia*}*@gmu.edu*

Abstract We consider a collaborative enterprise computing environment where a group of enterprises or parties maintain their own relational databases to which they allow restricted access to other parties. The access is regulated by means of a set of authorization rules that may be defined using relational calculus, including joins over relations from multiple parties. In this chapter, we provide an overview of the issues that arise in such an environment and some solutions. In particular, since individual parties are likely to formulate the rules in a somewhat piecemeal manner, the rules may be mutually inconsistent or inadequate to answer the desired queries. We address the issues of detecting inconsistencies and methods for fixing them. We also discuss the question of enforceability (or adequacy) of the rules. When rules, as given, are not enforceable, we can either augment the access rights or employ trusted third parties to perform unenforceable operations. We also address the issue of handling dynamic changes to rules. Finally, we consider the problem of generating efficient query plans in this environment.

1.1 Introduction

Enterprises increasingly need to collaborate to provide rich business services to clients and with minimal manual intervention. This requires the enterprises involved in the service path to share data in an orderly manner. For instance, an automated determination of patient coverage and costs requires that a hospital and insurance company be able to make certain queries against each others' databases. Similarly, to arrange for automated shipping of merchandise and to enable automated status checking, the e-commerce vendor and shipping company should be able to exchange relevant information, perhaps in form of database queries. To achieve collaborative computation, data owners need to provide access to their data to other parties based on the needs of the allowable queries. It is also important not to release more information than necessary. For example, an insurance company may wish to access pa-

tient data at hospital for the individuals that it insures. However, it would be highly undesirable for the hospital to release information about patients that are not the clients of the said insurance company. In relational terms, this means that the access granted to the insurance company is over the join of its client table and hospital's patient table projected over the desired columns. With multiple parties involved, each with their own data sharing and protection requirements, the picture could get rather complicated, thereby leading to the problems such as conflicts between rules or insufficient access to answer the desired queries. These are the issues of primary concern in this chapter. In the rest of the chapter, we introduce the cooperative data access model and problems in Section 1.2. We discuss the mechanisms to solve the various problems in Section 1.3. In Section 1.4, we discuss use of trusted third parties for collaboration and handling of authorization rule changes. At last, we conclude our discussion and list interesting future directions for research in Section 1.5.

1.2 Cooperative Data Access Model

Without loss of generality, we assume each collaborative party or enterprise maintains its own data in its private cloud. Such a party may have its own data center running the private cloud or possibly running the cloud on infrastructures rented from a provider. We assume here that all data is stored in relational form and structured in a standard form such as BCNF. The latter property allows for lossless joins over keys. It may be possible to extend the analysis to more general data models, but that aspect is beyond the scope of this chapter.

As the enterprises need to collaborate with one another to fulfill the desired business requirements, they will negotiate among themselves suitable access rights. For instance, an insurance company may request access to some hospital data, perhaps in exchange for providing some of its data to the hospital. We define the data access privileges using a set of authorization rules. Since we are dealing with the relational model, the authorization rules are made over the original tables belonging to enterprises or over the lossless joins ($\bowtie$) over two or more relational tables. The join operations, coupled with appropriate projection and selection operations define the access restrictions. In order to enable working with only the schemas, in this chapter, we do not consider the selection operation. We use the join operation over the relations because it can implicitly constrain the tuples being released to the authorized party and it meets the requirement of cooperative data access. For example, if the hospital thinks the insurance company should be able to obtain the patient information but only these patients who have plans with this insurance company, then the authorization given to the insurance company is defined only on the join result of hospital and insurance tables.

We assume that the authorization rules themselves are not considered sensitive and are visible to all parties. In cases where this is not desirable, all the rules could be managed by a trusted third party but this only affects where the algorithms considered in this chapter can run. In either case, we assume that all rules are available

in a central place for manipulations. The purpose of cooperative data access is for parties to run queries against one-another's databases. Thus, we first need to check if the information requested by the querier (or client) is authorized, and if so build a query execution plan to retrieve the desired data. The query execution plan must follow the given authorization rules at every step. Figure 1.1 shows a possible architecture for this environment. As a client initiates a query, it is first handled by the query planner which checks authorizations and generates a safe query plan.

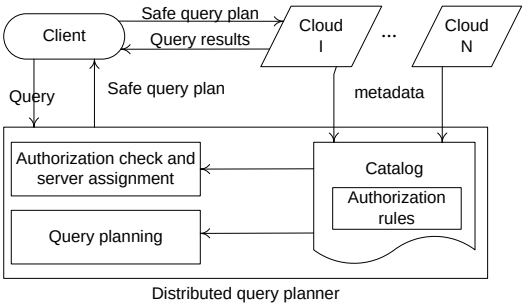

Fig. 1.1 Centralized authorization rule control

For simplicity, we assume simple select-project-join queries (e.g., no cyclic join schemas or queries). In general, the join operation cannot be done on any two arbitrary attributes, and the possible joins between different relations are usually limited. We assume that the join schema is given – i.e., all the possible join attributes between relations are known. Each join in the schema is assumed to be lossless so that a join attribute is always a key attribute of some relations. We also assume that the collaborating parties are non-malicious and strictly follow the given rules. Finally, we assume that there is only one authorization rule over each distinct join operation.

1.2.1 Notations and Definitions

We first introduce our authorization model. An **authorization rule** r_t is a triple $[A_t, J_t, P_t]$, where J_t is called the join path of the rule, A_t is the authorized attribute set, and P_t is the party authorized to access the data.

Definition 1 *A* **join path** *is the result of a series of join operations over a set of relations $R_1, R_2 \ldots R_n$ with the specified equi-join predicates $(A_{l1}, A_{r1}), (A_{l2}, A_{r2}) \ldots (A_{ln}, A_{rn})$ among them, where (A_{li}, A_{ri}) are the join attributes from the two relations. We use JR_t to indicate the set of relations in a join path J_t. The* **length** *of a join path is the cardinality of JR_t.*

Rule No.	Authorized attribute set	Join Path	Party
1	{pid, location}	W	P_W
2	{oid, pid}	E	P_W
3	{oid, pid, location}	$E \bowtie_{pid} W$	P_W
4	{oid, pid, total}	E	P_E
5	{oid, pid, total, issue}	$E \bowtie_{oid} C$	P_E
6	{oid, pid, total, issue, address}	$S \bowtie_{oid} E \bowtie_{oid} C$	P_E
7	{oid, pid, location, total, address}	$S \bowtie_{oid} E \bowtie_{pid} W$	P_E
8	{oid, pid, issue, assistant, total, address, delivery}	$S \bowtie_{oid} E \bowtie_{oid} C \bowtie_{pid} W$	P_E
9	{oid, address, delivery}	S	P_S
10	{oid, pid, total}	E	P_S
11	{oid, pid, total, address, delivery}	$E \bowtie_{oid} S$	P_S
12	{oid, pid, total, location}	$E \bowtie_{pid} W$	P_S
13	{oid, location, pid, total, address, delivery}	$S \bowtie_{oid} E \bowtie_{pid} W$	P_S
14	{oid, pid}	E	P_C
15	{oid, issue, assistant}	C	P_C
16	{oid, pid, issue, assistant}	$E \bowtie_{oid} C$	P_C
17	{oid, pid, issue, assistant, total, address, location}	$S \bowtie_{oid} C \bowtie_{oid} E \bowtie_{pid} W$	P_C

Table 1.1 Authorization rules for e-commerce cooperative data access

We can consider a join path as the result of join operations with all the attributes intact. Then A_t can be interpreted as set of attributes projected on the join path accessible to party P_t. Table 1.1 shows an example set of rules given to the cooperative parties. The first column is the rule number, the second column gives the attribute set of the rules, the third column is the join path, and the last column shows the authorized parties of the rule. Only one rule can be given to a party on a given join path. We assume that each authorization rule includes all of the key attributes of the relations that appear in the join path. In other words, a rule has all the join attributes on its join path. We believe that this is a reasonable assumption as in most cases when the information is released, it is released along with the key attributes.

Table 1.1 corresponds to our running example throughout this chapter. It concerns an e-commerce scenario with four parties (or Enterprises): (a) *E-commerce*, denoted as E, is a company that sells products online, (b) *Customer_Service*, denoted C, that provides customer service functions (potentially for more than one company), (c) *Shipping*, denoted S, provides shipping services (again, potentially to multiple companies), and finally (d) *Warehouse*, denoted W, is the party that provides storage services. To keep the example simple, we assume that each party has but one relation described as follows:

1. E-commerce (order_id, product_id, total) as E
2. Customer_Service (order_id, issue, assistant) as C
3. Shipping (order_id, address, delivery_type) as S
4. Warehouse (product_id, location) as W

In the following, we use *oid* to denote *order_id* for short, *pid* stands for *product_id*, and *delivery* stands for *delivery_type*. The possible join schema is also given in fig-

ure 1.2. Relations E, C, S can join over their common attribute oid; relation E can join with W over the attribute pid. The relations are in BCNF, and the only FD (Functional Dependency) in each relation is the underlined key attribute determines the non-key attributes.

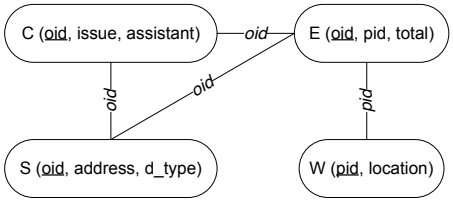

Fig. 1.2 The given join schema for the example

When a query is given, it should be answered by one of the parties that have the authorization. Since our authorization model is based on attributes, any attribute appearing in the Selection predicate in an SQL query is treated as a Projection attribute. In other words, the authorization of a PSJ query is transformed into an equivalent Projection-Join query authorization. Thus, a query q can be represented by a pair $[A_q, J_q]$, where A_q is the set of attributes appearing in the Selection and Projection predicates, and the query join path J_q is the FROM clause of an SQL query. For instance, there is an SQL query:

"Select $oid, total, address$ From E Join S On $E.oid = S.oid$ Where $delivery$ = 'ground'"

The query can be represented as the pair $[A_q, J_q]$, where A_q is the set $\{oid, total, address, delivery\}$; J_q is the join path $E \bowtie_{oid} S$. In fact, each join path defines a new relation/view, and we say two join paths J_i and J_j are **equivalent**, noted as $J_i \cong J_j$, if any tuple in J_i appears in J_j and vice versa. As information release is explicitly defined by the rules, an authorized query must have a matching rule to allow the access.

Definition 2 *A query q is* **authorized** *if there exists a rule r_t such that $J_t \cong J_q$ and $A_q \subseteq A_t$*

The rule and the authorized query must have the equivalent join paths. Otherwise, the relation/view defined by the rule will have fewer or more tuples than the query asks for. Here we don't consider the situation where the projections on two different join paths get the same result (e.g., by joining on foreign keys) since data coming from different parties usually does not have foreign key constraints. For instance, the example query Q_1 is authorized by r_{11}, but it cannot be authorized by r_{13}. Although all the required attributes are authorized by r_{13}, their join paths are not equivalent.

1.2.2 Issues in Collaborative Data Access

The data authorization rules are supposed to satisfy the requirements laid down by each enterprise, but without a careful analysis of interactions between them, we may find that the rules are either mutually inconsistent or inadequate to allow desired queries. For example, a hospital may choose to release data to an insurance company without realizing what additional information the insurance company can get from other parties such as a credit card company. If the data that the insurance receives from hospital and credit card company is joinable, it can perform the join and thereby effectively have access to data that is not authorized for it by any explicitly stated rule. In other words, we now have an authorization rule that was perhaps not intended to be granted. For example, the insurance company can now deduce credit score of the patients at the hospital in question. We say such a rule is inconsistent relative to the set of intended authorizations. Rule inconsistency is obviously undesirable since it amounts to information leakage without explicit knowledge of the parties involved.

Another problem is inadequacy of the given rules, which may cause a query to be authorized but not implementable. The simplest way to illustrate this problem is by considering the following situation: a rule specifies access to $R \bowtie S$ (where R and S are relations owned by two different parties); however, no party has access to both R and S individually and thus no party is able to do the join operation! In such case, a query requesting the data on the join result of R and S is authorized by the rule, but the query cannot be answered. We say that a rule can be enforced among the cooperative parties if there exists a series of operations among the cooperative parties that is allowed by the rule permissions and the final result is exactly the information conveyed by the rule.

One way to enforce a rule is to introduce a trusted third party that is given enough accesses in order to compute and supply the missing information. In the above example, if there is a trusted third party trusted by owners of R and S, they can supply it with relations R and S so that the third party can generate the missing $R \bowtie S$. We shall discuss different third party models to enforce the rules and answer queries. A third party may either act as an opaque service provider that does not retain any data, or provide richer functionality such as caching of data or query results. Multiple third parties may be needed to provide data isolation, handle trust issues, or to simply improve performance. In any case, it may be desirable to minimize third party involvement due to risk of data exposure (in transit or due to hacking), data transfer costs/delays, or the money charged by third parties.

If a query is authorized and the corresponding rule(s) can be enforced, we still need a safe query execution plan to answer the query. In spite of vast literature on query planning, the problem here requires a new approach because of the access restrictions and involvement of multiple parties.

1.2.3 Related Work

The problem of controlled data release among collaborating parties has been studied in [14]. The basic model in this paper is identical to ours and provides the motivation for our work. Its main contribution is an algorithm to check if a query with a given query plan tree can be safely executed. However, it does not address the problem of rule enforceability. Without a trusted third party, the unenforceable rules are inaccurate configurations and need to be revised, and we address that in our work. In another work [13], the same authors evaluate whether the information release the query entails is allowed by all the authorization rules given to a particular user, which considers the possible combinations of rules and assumes that the rules are defined in an implicit way. In our work, we assume authorizations are explicitly given, and data release is prohibited if there is no explicit authorization. While they focus on the problem of query authorization, we emphasize the executability of the authorized queries.

Processing distributed queries under protection requirements has been studied in [6, 18]. In these works, data access is constrained by a limited access pattern called binding pattern, and the goal is to identify the classes of queries that a given set of access patterns can support. These works with access patterns only considers two subjects, the owner of the data and a single user accessing it, whereas the authorization model considered in this work involves independent parties who may cooperate in the execution of a query. There are also classical works on distributed query processing [5, 17]. Most of these techniques aim to improve performance of query processing in the distributed environments, and minimize the data transmission among the different sites. In our scenario, authorization rules made by the data owners put constraints on data access privilege. When processing the queries, we should not only optimize for performance but also make sure no security constraint is violated.

Answering queries using views [16] is close to our work also since each rule can be thought as a view over basic relations. Answering queries using views can be used for query optimization [15], maintaining physical data independence and data integration [8]. Different methods can be applied, materialized views can be treated as new options and put into the conventional query plan enumeration to find better query plan, queries can also be rewritten using given views with query rewriting techniques, and sometimes conjunctive queries are used to evaluate the query equivalence and information containment. However, these works do not consider the collaboration relationships among different parties, which make our problem different from them.

In the area of outsourced database services, some works [1, 7] discuss how to secure the data in such environments, and there are also services like Sovereign joins [2]. Such a service receives encrypted relations from the participating data providers, and sends the encrypted results to the recipients. These methods are useful to enforce our authorization rules. For instance, we can use Sovereign joins [2] as a join service in our trusted third party model. The given authorization rules is also similar to the firewall rules, which indicates what types of queries can go through.

As firewall rules are need to be enforceable and accurate [4, 12], we have the same requirements in our situation.

This chapter is mostly based on our previous works [9, 10, 20, 11]. Authorization rule consistency problem is address in [9], and [10] discusses the authorization rule enforcement checking problem. The mechanism to generate safe query plans is discussed in [20], and [11] deals with the problem of using trusted third parties in a minimal way.

1.3 Enabling Cooperative Data Access

In this section, we discuss the mechanisms to solve various problems in cooperative data access environment. These problems include achieving authorization rule consistency, checking rule enforcement and generating safe query plans for authorized queries.

1.3.1 Rule Consistency

Rules can be specified in two styles. An *implicit specification* means any valid compositions of the given rules are also considered as valid rules. In contrast, an *explicit specification* lists out all the allowed accesses and any access not included in the list is not allowed. In general, if a party obtains two joinable relations, say R and S according to two different explicit rules, it is free to join them to obtain $R \bowtie S$. With implicit specification, such a composition is, by definition, allowed and the parties involved must accept the risks of additional information leakage. However, with explicit specification, the composition is clearly not intended and we need to resolve the inconsistency. This can be done in two basic ways: (a) addition of the derived authorization such as $R \bowtie S$ to the rules, or (b) additional restrictions to disallow access to the composition.

In this work, we focus only on (a) and rely on the enterprises to expand their rules suitably so that access to compositions is explicitly included in the rules. This is a reasonable approach since it is not possible to prevent private computation by a party without restricting the component information itself. Approach (a) effectively implies that we generate a closure of the given set of rules. Formally, if rules r_i, r_j of party P can be joined losslessly according to the given join schema, and the resulting information $[A_i \bigcup A_j, J_i \bowtie J_j]$ is also authorized by another rule r_k of party P, then we say the two rules are **"upwards closed"**. For a set of rules, if any two rules that can be joined losslessly are "upwards closed", we say the set of rules is **consistent**, and the rules form a **consistent closure**. In the following, we shall consider how to systematically and efficiently generate the consistent closure of given set of rules.

Although approach (a) is straightforward, there are many instances where approach (b) is highly desirable. This happens when the association of two pieces of

information is more sensitive than the individual pieces of information. For example, a hospital may not want the insurance company to be able to correlate medical diagnosis of its patients with their insurance claim histories, even though it does want to convey diagnosis information to the insurance company. The only way to restrict composition ability is to deny unrestricted access to one of the two basic relations involved in the composition. For example, if it is problematic to allow party P_t to have access to $R \bowtie S$, we must ensure that P_t can access either R or S but not both. In particular, P_t may be given unrestricted access to R, but for any queries involving S, it needs to go through a third party that controls the amount of data transferred. Thus, P_t cannot reliably construct the full $R \bowtie S$. (As usual, it is necessary to assume that P_t cannot accumulate up to date version of the entire S over time via a series of small queries. Without such an assumption, giving any access to tuples of a relation would amount to allowing access to the entire relation.)

Returning to approach (a), it is expected that the original authorization rules specified by the participating enterprises will usually be inconsistent and we need to identify the missing compositions that would remove the inconsistency. In the following we consider the consistency problem from the perspective of an individual party, but the same procedure needs to be repeated for every party.

We start by introducing the notion of key attribute hierarchy, which is useful for iterative construction of the closure. Consider two relations R and S with key attributes $R.K$ and $S.K$ respectively. If these relations can join losslessly, then the joining attribute must be the key attribute in at least one of them [3]. That is, either the join is performed on $R.K$, $S.K$, or $R.K$ is the same attribute as $S.K$. In either case, one key attribute from a basic relation is also the key attribute of the join result of the two relations. If the join is performed over the attribute $S.K$ ($R.K \neq S.K$), then the attribute $R.K$ can functionally determine the relation S. In such case, we say $R.K$ is at a higher level than $S.K$, denoted $R.K \rightarrow S.K$. If $R.K = S.K$, there is no hierarchy, and such key attribute of R and S is also the key attribute of the join result. For a given valid join path, the key attribute of the join path is always a key attribute from a basic relation. We call the key attribute of the join path in an authorization rule as **key** of the rule. Also, the join attributes in the join paths are always key attributes of some basic relations and these join attributes form the hierarchal relationship. For instance, in the given example rules, the key attribute *oid* is at the top level and $oid \rightarrow pid \rightarrow sid$.

Now for each key attribute of the basic relation, we create a group for the rules, called *join group* that takes this attribute as its key attribute. Since the rules within this group share the same key attribute, any two of them can join over their key attributes. More formally, a **join group** is a group of authorization rules associated with a key (join) attribute, where all the attributes in these rules functionally depend on this attribute. If a join group is **consistent**, then it is called a **consistent join group**.

Since some rules can be the result of private computation over other rules with respect to join paths, the rules themselves have relationships. Given a rule r_t with join path J_t, we call a join path as a **sub-join path** of J_t if it is a join path that contains a proper subset of relations of JR_t. We say a rule defined on a sub-join path of J_t is

a **relevant rule** to r_t. A rule r_t can be generated only by combining the information from its relevant rules, since any other combination will contain extra information from relations not in J_t. Thus we can organize the rules into a **relevance graph** where each node is a rule marked by its join path and the nodes are connected by the relevance relationship. For instance, Figure 1.3 shows a relevance graph. Here J_2 is a sub-path of J_6, and r_2 is a relevant rule to r_6, the rules are connected in the graph.

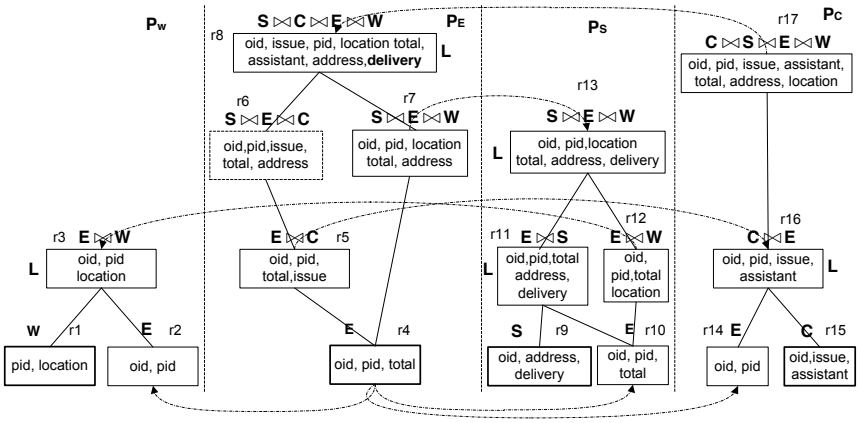

Fig. 1.3 Graph structure built for the example

It is now possible to outline the closure algorithm, although for brevity we refer the reader to [9] for details. The algorithm first divides rules into join groups and generates consistent join groups. Next, based on the join attribute hierarchy, each join attribute is considered for deriving further rules, and any such rules are added to the rule closure. When this procedure terminates, we have the entire consistent closure.

1.3.2 Rule Enforcement

In section 1.2.1, we introduced the concept of query authorization. However, "authorized" is only a necessary condition for a query to be answered but not sufficient. To perform the required join operations to answer the query, we need to find appropriate parties that have the sufficient privileges to do these joins. Therefore, at least one legitimate query execution plan is required to answer a given query. A **query execution plan** or "query plan" for short, includes several ordered steps of operations over authorized and obtainable information and provides the composed results to a party. A query plan generates relational results, which can also be represented with the triple $[A_{pl}, J_{pl}, P_{pl}]$. All the operations executed and the final results pro-

duced by a valid query plan should be authorized by some given authorization rules. A query plan *pl* answers a query q, if the final generated relational result of the plan satisfies $J_{pl} \cong J_q \cong J_t$, $A_q = A_{pl} \subseteq A_t$ and $P_{pl} = P_t$. An authorization rule defines the maximal set of attributes that a query on the specified join path can retrieve. Thus, each rule can also be treated as a query. We call the query plan to enforce a rule as an **enforcement plan** or "plan" for short in the following.

Definition 3 *A rule r_t can be* totally enforced*, if there exists a plan pl such that $J_t \cong J_{pl}$, $A_t = A_{pl}$, $P_t = P_{pl}$. r_t is* partially enforceable*, if it is not totally enforceable and there is a plan pl that $J_t \cong J_{pl}$, $A_t \supset A_{pl}$, $P_t = P_{pl}$. Otherwise, r_t is not enforceable. A join path J_t is enforceable if there is a plan pl that $J_t \cong J_{pl}$.*

If a rule can be correctly enforced, there should be at least one valid enforcement plan for it. To enforce a rule with a long join path, we need to access the information from its underlying relations. Hence, whether a long join path can be enforced depends on the enforceability of the shorter join paths relevant to it. An rule enforcement plan with a long join path also uses the results of enforcement plans with shorter join paths. To that end, the enforcement plan building process requires a systematic walk through the rules with increasing join path length. At the beginning, the rules involving the basic relations (i.e., access rights of an enterprise to its own data) are trivially known to be totally enforceable. In the next step, we consider enforcement of rules with join path length of 2, and so on. In considering enforcement of a rule involving join of data from two distinct parties, we may need transmission of attributes from an owner party to another one that has access to it but does not own it. We call a plan as **joinable plan** if it contains all the key attributes of the basic relations in its join path. In some cases, a rule does not have a total enforcement plan. However, there are plans whose results implement subsets of the rule attribute set. We say that an attribute set is a **maximal enforceable attribute set** for a rule, if it is the result of a valid plan, and there is no other plan of the same rule that can implement a superset of these attributes. If a rule is totally enforceable, its maximal enforceable attribute set is the rule attribute set. Each rule has only one maximal enforceable attribute set.

It is obvious that not all the rules are enforceable. Whether an enforcement plan exists depends on whether pieces of enforceable information on shorter join paths are available and whether they can be joined losslessly at some place. In a cooperative environment, the enforceable information on remote cooperative parties may also be helpful to construct an enforcement plan. We do need a mechanism to check rule enforcement so as to tell which rules can be enforced and what are their maximal enforceable attribute sets. We address the rule enforcement checking problem in two steps. First, we examine the enforceability of each authorization rule in a constructive bottom-up manner, and build a relevance graph that captures the relationships and the enforceability among the rules. Second, we deal with the unenforceable information in the examined rules.

Unenforceable rules can be handled in two ways. The first choice is that we keep only the found enforceable rules with their maximal enforceable attribute sets, and rules that are not enforceable as well as the unenforceable attributes are removed

from the rule definitions. In other words, the algorithm finds all the information that can be safely retrieved according to the given set of rules, and all inaccurate and unenforceable definitions are removed. This solution can be thought as a conservative one since it prohibits some authorized information to be released because of the enforceability. The second choice is to modify the rules as needed. For this, we take the view that all the information regulated by the rules is authorized, and authorized information should be retrievable. Whenever any information in the defined rules cannot be enforced, we change the rule configurations by granting more privileges so as to make this information enforceable. Since there are different ways to modify the rules, we prefer to find the way that has minimum impact on the existing rules. That is, we try to find the minimum amount of additional information to release. We have developed algorithms of both flavors and the details can be found in [10].

1.3.3 Query Planning

Once the enforceability of a query – or rather a rule satisfying the query – is known, we need a mechanism to generate the detailed query plan. A query plan is generated top-down by considering operations over sub-plans until the sub-plans refer to basic relations. The possible operations on plans are projection, join and data transmission. For instance, there is an enforcement plan for r_3 in Table 1.1, and such a plan contains a join over two sub-plans on the data authorized by r_1 and r_2 respectively. P_W owns the information authorized by r_1, and the sub-plan for it is an access plan reading the table W. The sub plan for r_2 includes an access plan reading table S at P_S, and another operation transmitting the data from P_S to P_w. The example plan authorized by r_3 has the $J_{pl} = E \bowtie_{pid} W$, and $A_{pl} = \{oid, pid, location\}$. We say a rule r_t **authorizes** ($\succeq$) a plan pl, if $J_{pl} \cong J_t$, $P_{pl} = P_t$, and $A_{pl} \subseteq A_t$.

Definition 4 *An operation in a query plan is* **consistent** *with the given rules R, if for the operation, there exist rules that authorize access to the input tuples of the operation and to the resulting output tuples.*

For the three types of operations in our scenario, we give the corresponding conditions for consistent operation.

1. For a projection (π) to be consistent with the rules, there must be a rule r_p authorizes ($\succeq$) the input information.
2. Join ($\bowtie$) involves two input subplans pl_{i1} and pl_{i2} to generate the resulting plan $pl_o = pl_{i1} \bowtie pl_{i2}$. For a join operation to be consistent with R, all the three plans need to be authorized by rules. Since join is performed at a single party, and rules are upwards closed, if the input plans are authorized by rules, the join operation is consistent.
3. Data transmission ($\rightarrow$) is an operation that involves two parties. The input is a plan pl_i on a party P_i, and the output is a plan pl_o for a party P_o, where $pl_o = pl_i \rightarrow P_o$. As each join path defines a different relation, the receiving party must have

a rule that is defined on the equivalent join path as the information being sent. Otherwise, the transmission is not safe. Therefore, a data transmission operation to be consistent with R, if $\exists r_i, r_o \in R, J_i \cong J_o, P_i \neq P_o$ and $r_i \succeq pl_i, r_o \succeq pl_o$. If P_i is sending information with attributes not in A_o, P_i should do a projection operation $\pi_{A_o}(pl_i)$ first.

In the example, r_8 authorizes P_E to get information on $(S \bowtie E \bowtie C \bowtie W)$. If P_S sends the information of r_{11} to P_E, it will not be allowed. Although the attribute set of r_{11} is contained by r_8, there is no rule for P_E to get data on the join path of $(E \bowtie S)$, and the data transmission is disallowed.

Definition 5 *A query execution plan pl is* **consistent** *with the given rules R, if for each step of operation in the plan is consistent with the given rule set R.*

Let us now consider the basic query planning problem: given a set of authorization rules R and an incoming query q against enforceable information, generate a consistent query plan pl that is optimal and satisfies all the rules.

Due to the difficulties in enumerating all possible ways of answering a query, we consider a greedy algorithm based on the relevance graph [20]. To generate a consistent query plan, we need to make sure all the join operations in a join path can be safely implemented. In other words, we need to find a way of enforcing the query join path and retrieve all the attributes for the query. To find an efficient consistent query plan, we always choose the optimal join path enforcement plan first, and then apply the greedy mechanism to obtain required relevant rules. The problem of covering all the required attributes is similar to the classical weighted set covering problem, and hence the greedy algorithm also follows a similar approach.

The optimal enforcement plan for a join path on a specified party can be precomputed by extending the rule enforcement checking algorithm using a dynamic programming approach. Such a plan only enforces the join operations in query and usually results in missing attributes. To retrieve missing attributes, we traverse the graph structure again to decompose the target rule into a set of relevant rules that can provide these attributes. We record the required operations among these rules, and then recursively find ways to enforce these relevant rules to generate a query plan. With the greedy heuristic, we always try to decompose a rule into a minimal number of relevant rules. As we recursively look for the plans to enforce the relevant rules, we try to use the intermediate results as much as possible to improve the performance. The details of the algorithm and proofs of correctness can be found in [20].

The time complexity of the proposed greedy algorithm is $O(N^3)$, where N is the total number of rules. In addition, we evaluated the generated query plans of the algorithm. Since the optimal plan cannot be found in general, we cannot compare the resulting query plans with the optimal ones. Thus, we use simple case studies, where manually finding the optimal plans becomes possible, and we compare the results on these cases. The results show that the greedy query planning algorithm is effective in finding a good query plan for an authorized query. In most of the cases, it generated the optimal plans, and it gave close to optimal plans in the remaining cases.

1.4 Other Authorization Issues

We discuss several other issues in this section. The first problem is using a trusted third party in a minimal way to enforce rules that are not enforceable among existing collaborating parties. The second problem is maintaining the rule consistency property in the case of rule changes.

1.4.1 Rule Enforcement with Trusted Third Parties

As discussed earlier, the enforcement checking may reveal that no party is capable of performing certain operations. One way to handle such a case is by introducing trusted third parties that provided required data accesses in order to enforce unenforceable rules. However, third parties may be expensive to use and the data given to them could be at greater risk of exposure than the data maintained by original parties. Therefore, we focus on the problem of using third parties minimally in order to deliver the information regulated by the given authorization rules. We model the cost of using third party by communication and computing costs. It is not surprising that finding the minimal cost with third party to implement a given rule is *NP*-hard, and thus a greedy algorithm becomes essential.

We assume that a trusted third party (TP) is not among the existing cooperative parties and can receive information from any cooperative party. We assume that the TP always performs required operations honestly, and does not leak information to any other party. The simplest third party model is one of memoryless service provider. That is, each time we want to enforce a rule, we need to send all relevant information to the third party. The third party does its job, returns the results and then completely cleans up its storage space (i.e., no retention of data between successive requests). With the existence of a third party, we can always enforce a rule by sending relevant information from cooperative parties to TP.

Since each rule defines a relational table, we can quantify the amount of information represented by a rule. This can be exploited in minimizing the amount of information used by third parties. All the selected rules must be relevant to the target rule r_t that is to be enforced. If a relevant rule of r_t is not relevant to any other relevant rules of r_t with longer join paths on the same party, we call it a *Candidate Rule*. We only choose from candidate rules to decide the data that needs to be sent to the TP. Sending minimal information to the third party can minimize not only communication cost but also computation costs. However, estimating computation costs precisely can be challenging.

Suppose that we have a set of cooperative parties $\{P_1, P_2 ... P_m\}$ together with a set of rules $R = \{r_1, r_2 ... r_n\}$ and a *target rule* r_t to be enforced at the third party TP. The amount of the information is quantified by sum of the number of attributes picked from each rule multiplied by the number of tuples in that selected rule. Thus, we want to minimize the communication cost $Cost = \sum_{i=0}^{k} \pi(r_i) * card(r_i)$, where r_i is a selected rule, k is the number of selected rules, and $\pi(r_i)$ is the number of attributes

selected to be sent, and $card(r_i)$ is the number of tuples in r_i. More specifically, the communication cost can be defined as follows:

$$cost(C) = \sum_{i=1}^{k} w(S_i)\pi(S_i), \pi(S_i) = \begin{cases} |S_j \bigcap (U \setminus \bigcup_{j=1}^{i-1} S_j)|, & \text{if } (key(S_i) \notin \bigcup_{j=1}^{i-1} S_j) \\ |S_j \bigcap (U \setminus \bigcup_{j=1}^{i-1} S_j)| + 1, & \text{if } (key(S_i) \in \bigcup_{j=1}^{i-1} S_j) \end{cases} \tag{1.1}$$

In equation (1.1), the function $key(S_i)$ gives the key attribute of candidate rule r_i. In general, this can depend on the number of attributes selected by the rule r_i. To illustrate, suppose that we have a rule $\{oid, total, pid, location\}, (E,W) \rightarrow$ Party E. Even though *oid* is the key of the entire rule, if we only need *location* in this rule, *pid* can be the key of the selected rule. In such a case, if *oid* is covered by previous selected rules but *pid* is not, then using *pid* as the key can reduce the overall cost. However, due to the complexity of these situations, we assume function $key(S_i)$ always gives the key of r_i, which is *oid*. We can think $w(S_i)$ is the per attribute cost for the rule r_i which is mostly determined by $card(r_i)$. In fact, the number of tuples in a relation/join path depends on the length of the join paths and the join selectivity among the different relations. Join selectivity [17] is the ratio of tuples that agree on the join attributes between different relations, and it can be well estimated using the historical and statistical data of these relations in many cases.

The "computing cost" is defined as the cost of CPU usage and disk I/O. These costs are incurred as the third party fetches data from storage devices, performs join operations, and writes out the join results. (The I/O cost of receiving the incoming data from cooperative parties and relaying results to them is counted as part of communication cost and not included in the computation cost.) The computing cost is difficult to estimate because of the different access methods for relations (e.g., index scan or sequential scan), and different join methods (e.g., nested loop, sort-merge, hash-join, etc.) Moreover, the order of joins and the size of the input data and join results also influence the computing cost. We assume the sizes (in terms of number of tuples) of the basic relations and results of joins are known. We denote the cost of a resulting relation on join path J_i as $w(J_i)$, which can be estimated as discussed above. We also assume all the joins are done with nested loop method, and given n rules, the third party always does $n-1$ sequential join operations. We assume the relations have indices on the join attributes. For a nested loop join with two input relations, the cost can be estimated as: $Access(Outer) + (Card(Outer) * Access(Inner))$, where $Access(R)$ is the cost of access the relation R, and $Card(R)$ is the number of tuples in R. Obviously, we always prefer using the smaller input relation as the outer relation. In addition, as we need to perform $n-1$ joins, we keep the intermediate join results of the previous joins. The result of a join can be estimated as $Access(Result) = Access(Out * Inner * SelectivityFactor)$, where $SelectivityFactor$ is the estimate of what fraction of input tuples will be in the result. Therefore, the total cost of $n-1$ join operations is:

$$CompCost = Access(R_1) + \sum_{i=1}^{n-1}(((Card(JR_i) * Card(R_{i+1}) * Attr(R_{i+1}) * SF_{JRi,R_{i+1}})$$

In the above equation, R_1 is the selected rule with least cost $w(J_i)$, and $Access(R) = Card(R) * Attr(R)$, where$Attr(R)$ is the number of attributes of R. JR_i is the join results of the rules from R_1 to R_i, and $SF_{JRi,R_{i+1}})$ is the selectivity factor for each join operations. To minimize *CompCost*, it is preferred to have fewer operations, and for each operation, one with smaller cardinality should be used as the *Outer* relation. Given a set of selected relevant rules generated by the previous algorithms, we calculate the computing cost using the above model. Our experiments indicate that the communication cost and computing cost are generally closely related in practice (even though counterexamples are easy to construct). For the detailed algorithms and the comparisons with brute force algorithms, please refer to [11].

1.4.2 Handling Rule Changes

Until now, we have assumed the rules to be static. In practice, the rules may change with varying frequencies. One potential reason is simply a change in business policies, which is expected to occur only occasionally. The other case is where the interaction might involve multiple phases or stages with different (or somewhat different) rules in each phase. Other intermediate situations are also possible – such as when access rights are couple with some kind of reputation system. In the following we briefly consider the issues that may arise due to changes in access rules.

In general, a change of authorization rule that meets the new requirement and also has minimal impact on the remaining authorization rules is the optimal solution. In the algorithm considered here, we simply minimize the number of rules that need to be modified; however, in general, several other considerations may apply. For instance, some authorization rules may be more important than the others, and this aspect may need to be considered in minimizing the change. Similarly, some parties may collaborate more intimately or be more trustworthy than the others, and the changes should consider this gradation as well.

As rules are being changed, usually we need to modify a set of rules to maintain the rule consistency property. There are basically two types of rule changes. The first type of rule change is granting or revoking non-key attributes (non-join attributes) to an existing rule. In such scenarios, we can take advantage of the relevance graph to maintain the rule consistency. In case of rule grant, we search upwards in the relevance graph starting from the rule being modified, and this can be done with a depth first search. If the rule being inspected does not have the newly granted attributes, then the algorithm adds these attributes to the rule. If the rule being inspected already has these attributes, the search along this path will stop and another path will be picked. Consequently, the added attributes will be propagated to all the related rules that are at a higher level from the rule being changed. In the case of revocation, we search the relevance graph downwards, and the process is similar.

There is another type of rule change, where a rule with a new join path is granted to a party or an existing rule is completely revoked. We first discuss the new rule grant. In such a case, we need to check if this rule can join with existing rules to generate legitimate new rules. The mechanism is similar to the previous approach for generating the consistent closure. As the newly added rule has a new join path, we first obtain the key attribute of it, and then the rule is put into its corresponding join group. Within this group, as a new rule is added, the algorithm recomputes the consistent join group. This can be done efficiently since these rules all can join over their key attributes. The new rule is inserted into the graph of the join group. The algorithm will not check all its relevant rules in the graph since their composition will not create new join paths. All the other rules are checked and the new rule can join with each of them to form another new rule and be put into the consistent join group. In the next step, each of the added rules is iterated to see what are the other rules that can be generated based on it. By iterating the key attributes and the consistent join groups associated with them, the algorithm adds all the generated rules into the rule set so as to complete it as a consistent closure.

If an existing rule is completely revoked, we need to make sure that such a join path can no longer be generated from the remaining relevant rules. Therefore, each possible ways to enforce the join path need to be obtained and the possible pairs should be taken out. To achieve that, we use an algorithm taking advantage of the relevance graph as well. In the graph, only the *direct relevant rules* of the revoked rule denoted as r_v are examined. The direct relevant rules of r_v are the relevant ones in the graph that directly connect with r_v with one edge. For each of the directly connected rule r_d, the algorithm computes its matching rule r_m if it exists. Given the join schema and relevance graph, r_m can be efficiently determined, and (r_d, r_m) forms a pair which means a join over them can enforce the join path of rule r_v.

For each found pair of rules, the algorithm needs to remove one rule from it so as to make the join path no longer enforceable. If a rule in the pair is not locally enforceable, we prefer to remove it since it does not cause cascade revocations. In contrast, if a rule in the pair is locally enforceable, by removing this rule, we need to make sure all the rules that can compose this one are taken out. Thus, a cascade of revocations will occur. As this is a recursive process, we want to revoke minimal number of rules so that the impact is minimal. Hence, when iterating each pair of rules, the algorithm also records the number of appearances of the rules. The rule with most appearances is preferred to be removed since removing one such rule can break several pairs. In the worst case, half of the existing rules need to be removed from the rule set. The detailed algorithms for this are available in [9].

1.5 Conclusions and Future Work

In this chapter, we considered scenarios that require different parties and enterprises to cooperate with one another to perform computations to satisfy business requirements. Each of these enterprises owns and manages its data independently using a

private cloud, and these parties need to selectively share some information with one another. We considered an authorization model where authorization rules are used to constrain the access privileges based on the results of join operations over relational data.

In such an environment, we identified the problems of rule consistency and rule enforcement. For a query requesting enforceable information, consistent query plans are required so as to answer the query. We introduced the notion of consistent query plan and a mechanism to generate such plans. For the authorization rules that cannot be enforced among the cooperative parties, we proposed to use a trusted third party to perform the required join operations. We defined cost models to minimize the interactions between cooperative parties and third parties. Finally, we discussed how to maintain the rule consistency when some rules are modified.

We assumed that the collaborating parties first make the rules via negotiations, and then check whether a query is authorized and the safe ways to answer the query. It is possible to consider reversing the process. That is, we may want to figure out the complete set of queries that should be answered to meet business requirements, and after that we design authorization rules for cooperative parties so that only these wanted queries can be answered. However, due to the local computation, we may authorize extra information when granting privileges for this set of queries. Thereby, the problem becomes to find the best way of making rules so that minimal amount of extra information will be released together with the rules. To achieve that, we may also need a limited number of third parties are given, and there is a problem of finding the optimal solution under such a scenario.

We studied the rule consistency problem with infrequent rule changes. In a military or workflow scenario, the permissions as well as the data may change on a per mission basis so that an authorization rule given to a party applies only for a short period of time. Since the relevant data also changes frequently in this case, it will become useless after some time. In such environments, the authorization rule can be granted dynamically based on the demands. For instance, for each step in a query, we can grant permissions to authorize the operation on the fly. Once such a step is executed, the authorizations are revoked. This is similar to the workflow scenario. By granting privileges for a short time period, the extra information that is obtainable via local computation can be limited. The challenging problem becomes finding a way to schedule the queries as well as the time points to grant the authorizations so that minimal amount of extra information is released.

In our current model, access privileges are specified at the attribute level. Once a party can access an attribute, it can get all the tuples projected on that attribute. Since certain tuples can be more sensitive than others, restrictions on the tuple level are also necessary to prevent undesired data release. Thus if would interesting to consider simplified forms of selection operations that can be handled by the same framework. In addition, it is also interesting to consider the write permissions. Our current models assume that only the data owners may change their data and other parties just read the data from these owners. In some situations, it is desirable that a collaborating party can also modify the data owned by others. In addition to the

synchronization problems, there is also the challenging problem of organizing the privileges and correctly granting and revoking write privileges to certain parties.

Our current model does not assume any malicious insiders and all the parties are expected to strictly follow the given authorization rules. In practice, a party may not behave honestly during the collaboration. For instance, a party may obtain some authorized information from a data owner, and then leak it to some unauthorized parties. As another example, a party that receives data from the data owner and sends it to another party according to the generated query plan may change some of the data. Thus, it is required to have a mechanism that can verify the integrity of the received data. One possibility is to use the existing mechanisms such as hash values, Merkle trees, and signatures to ensure the data integrity [22, 23]. Considering the properties in the collaboration environment, it may be possible to check the data integrity through collaboration. In cooperative data access, there may exist more than one legitimate data transmission path beginning from the data owner to the authorized party. Therefore, parties can exchange the information they have. By doing that, if the number of misbehaving parties is limited, it is possible to detect them. It is also possible to define rules in such a way that each query be answered in at least k ways (for some k), and misbehaviors can be detected if only fewer than $k/2$ ways behave irregularly. Furthermore, existing mechanisms such as reputation systems [19] and trust management [21] can be considered to ensure the data integrity in the cooperative data access environments.

Our third party model has been rather simplistic. It is possible to consider more sophisticated models where the third party can store data and even the intermediate results for more efficient enforcement. Because of the limited availability of storage, and the varying potential of reuse of stored data, one needs to design caching policies carefully. The optimal cache policy of the third party can be different from the file cache and process cache because the relational data is structured and we can cache the data on a per tuple or column basis. Since the business data of the cooperative parties may be changing dynamically, caching also introduces the tricky problem of maintaining synchronization between original data and its copies.

To build a private cloud, different parties may rent the cloud infrastructure from the same service provider. It is also possible for an enterprise to build a hybrid cloud where the data owner manages the sensitive data locally, but the data for sharing is put in a public cloud. These emerging trends create new challenges and opportunities for secure cooperative data access. If cooperative parties use the same cloud provider, then the cloud provider could be used as a partially trusted third party to help enforce the security policies. In addition, it may be possible to perform privacy preserving join operations in such an environment. The expected mechanism can be a hybrid of using a trusted third party and the secure multiparty computation. Also, the cost model should also be revised under such situations.

References

1. G. Aggarwal, M. Bawa, P. Ganesan, and etc. Two can keep A secret: A distributed architecture for secure database services. In CIDR 2005, pages 186-199.
2. R. Agrawal, D. Asonov, M. Kantarcioglu, and Y. Li. Sovereign joins. In , ICDE 2006, 3-8 April 2006, Atlanta, GA, USA, page 26, 2006.
3. A. V. Aho, C. Beeri, and J. D. Ullman. The theory of joins in relational databases. ACM Transactions on Database Systems, 4(3):297-314, 1979.
4. E. Al-Shaer, A. El-Atawy, and T. Samak. Automated pseudo-live testing of firewall configuration enforcement. IEEE Journal on Selected Areas in Communications, 27(3):302-314, 2009.
5. P. A. Bernstein, N. Goodman, E. Wong, C. L. Reeve, and J. B. Rothnie, Jr. Query processing in a system for distributed databases (SDD-1). ACM Transactions on Database Systems, 6(4):602-625, Dec. 1981.
6. A. Cali and D. Martinenghi. Querying data under access limitations. In ICDE 2008, April 7-12, 2008, Cancun, Mexico, pages 50-59, 2008.
7. S. De Capitani di Vimercati, S.Foresti, S.Jajodia, S.Paraboschi, and P.Samarati. Keep a few: Outsourcing data while maintaining confidentiality. In ESORICS 2009, pages 440-455.
8. R. Pottinger and A. Y. Halevy. Minicon: A scalable algorithm for answering queries using views. VLDB Journal, 10(2-3):182-198, 2001.
9. M. Le, K. Kant, and S. Jajodia. Access rule consistency in cooperative data access environment. In 8th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing, 2012.
10. M. Le, K. Kant, and S. Jajodia. Rule configuration checking in secure cooperative data access. In 5th Symposium on Configuration Analytics and Automation (SafeConfig), 2012.
11. M. Le, K. Kant, and S. Jajodia. Rule enforcement with third parties in secure cooperative data access. In 27th IFIP WG 11.3 Working Conference on Data and Applications Security and Privacy (DBSec), 2013.
12. A. Wool. A quantitative study of firewall configuration errors. IEEE Computer, 37(6):62-67, 2004.
13. S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Assessing query privileges via safe and efficient permission composition. In CCS 2008, Virginia, USA, October 27-31, 2008.
14. S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Controlled information sharing in collaborative distributed query processing. In ICDCS 2008, Beijing, China, June 2008.
15. J. Goldstein and P. Larson. Optimizing queries using materialized views: A practical, scalable solution. In SIGMOD 2001, pages 331-342.
16. A. Y. Halevy. Answering queries using views: A survey. VLDB Journal,10(4):270-294,2001.
17. D. Kossmann. The state of the art in distributed query processing. ACM Comput. Survey, 32(4):422-469, 2000.
18. C. Li. Computing complete answers to queries in the presence of limited access patterns. VLDB Journal, 12(3):211-227, 2003.
19. K. Hoffman, D. Zage, and C. Nita-Rotaru, A survey of attack and defense techniques for reputation systems, ACM Computing Surveys (CSUR), vol. 42, no. 1, p. 1, 2009.
20. M. Le, K. Kant, and S. Jajodia. Consistent query plan generation in secure cooperative data access. Under submission. http://mason.gmu.edu/ mlep/submission.pdf
21. R. K. Ko, P. Jagadpramana, M. Mowbray, S. Pearson, M. Kirchberg, Q. Liang, and B. S. Lee, Trustcloud: A framework for accountability and trust in cloud computing, in Services (SERVICES), 2011 IEEE World Congress on, 2011, pp. 584588.
22. J. Buchmann, E. Dahmen, E. Klintsevich, K. Okeya, and C. Vuillaume, Merkle signa- tures with virtually unlimited signature capacity, in Applied Cryptography and Net- work Security, 2007, pp. 3145.
23. C. Wang, Q. Wang, K. Ren, and W. Lou, Privacy-preserving public auditing for data storage security in cloud computing, in INFOCOM, 2010 Proceedings IEEE, 2010, pp. 19.