University of Alberta

Security Issues in Heterogeneous Data Federations

by

Gregory Leighton

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computing Science

Fall 2011
Edmonton, Alberta

Abstract

Data federations allow the contents of multiple source databases to be accessed in a consistent manner. Since each source database is typically administered independently, heterogeneity often results, both in terms of how data is represented (i.e., the database schema), and in how controlled access to data is regulated. Typically, each data source exports data in relational format where it is combined into a semi-structured representation (e.g., XML).

In this thesis, we address two aspects of securing heterogeneous data federations. The first deals with the accurate translation of access control policies specified over source databases into a single, unified access control policy applicable to the wider data federation. Such a translation involves mapping each local identity to one or federated identities, and ensuring that the semantics of each original source policy are preserved (i.e., that no federated identity receives access to a larger region of federated data than intended by any source policy). We outline an efficient algorithm for automating policy translation. We also underscore the importance of automated translation methods by showing that in many realistic scenarios, verifying that a federated policy satisfies all source policies is intractable. Finally, we contribute an algorithm for minimizing the size of a translated policy.

The second problem we address is the prevention of information disclosures at the federated level. A disclosure risk is present when a user is able to combine the result of one or more allowable queries (i.e., queries which are permitted under the federated access control policy) with prior background knowledge in order to obtain a sufficiently high certainty of the answer to a disallowed query. We classify potential disclosure risks based on whether they can be detected at database design-time, or only when the contents of the database are known. We also describe a new measure for evaluating the magnitude of instance-based disclosure risks at query-time. Finally, we discuss the implementation of a prototype system, and conduct experiments that demonstrate the effectiveness and scalability of the proposed solution.

Acknowledgements

Graduate study is a strange business. You occasionally get to be a rock star, performing on conference stages around the globe. But, more often, you lead a pseudo-solitary existence, pouring over research papers in an ill-lit office or lab that seems miles away from civilization. It certainly helps to have good people around you – those who are willing to offer their support even if they don't always perfectly understand or agree with what you are up to.

On the academic side, I'd like to thank my supervisor, Denilson Barbosa, for giving me the freedom to explore new ideas, for encouraging me to read up on database theory, and, on a personal level, for being there through the good times and the bad. I'd also like to thank my MSc advisor, Tomasz Müldner, for continuing to provide advice and support years after my tenure as his student had officially ended. It's no exaggeration to say that I wouldn't have pursued graduate studies without his early and ongoing encouragement. I'd also like to thank the members of my doctoral and candidacy committees, Eleni Stroulia, Osmar Zaiane, Lukasz Kurgan, Gerome Miklau, Davood Rafiei, and Marek Reformat, for their constructive comments.

One of the peculiarities of my PhD was that I spent roughly equal time at two universities; this provided me with the opportunity to meet and learn from various faculty at the University of Calgary: Michael Jacobson was the first to spark my interest in data security and cryptography; John Watrous made complexity theory understandable to me, and is an amazing instructor. I'd also like to thank the various graduate students at UofA and UofC who generously shared their office/lab spaces with me, as well as the other interesting and sundry characters I met along the way.

Finally, I'd like to thank my family and friends for supporting me throughout my graduate studies.

Table of Contents

List of Tables

List of Figures

List of Symbols

Symbol	Explanation
A	Access control policy (ACP)
$\mathfrak{A}$	Set of all relational attributes
c	ID mapping constraint
$\mathcal{C}$	Set of ID mapping constraints
D	Source database
$\mathcal{D}$	Document Type Definition (DTD)
$\mathfrak{D}$	Union of all relational attribute domains
DS	Data source
δ	Set of transduction rules associated with a publishing transducer
δ'	Set of augmented transduction rules associated with a secure publishing transducer
ϵ	Threshold value for instance-based disclosure risks
f	Federated identity (role)
F	Set of federated identities
$\mathcal{F}$	Set of terminal symbols associated with a PRTG
G	Probabilistic regular tree grammar (PRTG)
I_i	Identity mapping function defined by the i-th data source
$\mathcal{I}$	Set of identity mapping functions
N	Set of non-terminal symbols associated with a PRTG
$\mathbf{P}_G$	Probability mass function associated with a PRTG G
ϕ	Relational query
Φ	Set of relational queries
Π	Publishing function
Π'	Secure publishing function
R	A relation or a set of grammar rules associated with a PRTG
$\mathcal{R}$	Disclosure risks table
$\mathfrak{R}$	Relation schema
S	Relational schema
s	Axiom (start symbol) associated with a PRTG
Σ	XML tag alphabet associated with a (secure) publishing transducer
Σ_D	Alphabet associated with a deterministic finite automaton
T	XML database (tree)
τ	Query twig
u	Local identity (user)
U	Set of local identities
$\mathfrak{U}$	Universe of discourse

Chapter 1

Introduction

With the passage of time, distributed applications have grown increasingly heterogeneous. This is due to many factors, including the introduction of new frameworks and "best practices", as well as the complexity and expense involved with porting legacy data into more modern architectures. In many cases, it has proved more palatable to focus on technologies for enabling data exchange between applications that differ in terms of how data is accessed, organized, and managed, rather than attempt to force each such application to adopt common policies.

The standard way of exchanging data across independent applications is to use the W3C's *eXtensible Markup Language (XML)* [21] as the common data representation format. Conceptually, this is done by defining a *mapping* of the source database into an agreed-upon XML representation which is then consumed by the target database. The availability of XML processing tools across several programming environments removes many of the barriers caused by differences in hardware, operating systems, and programming languages. The advantages of XML are even more pronounced when the setting above is generalized into a *federation* of independent databases, maintained by separate organizations (or departments of a single organization), and in which several pairwise mappings exist. Such federations are ever more common among government and large organizations. As an example, consider a state-wide health care governmental agency integrating several large, independently maintained healthcare systems of individual cities into a single federation. Because each city remains administratively independent, the resulting federation forms a heterogenous system.

XML has proven to be a popular data representation format for such settings, and several frameworks for exposing data as XML have been developed within the research community and by industry. For the case of relational data, the mappings exposing XML representations of the data are often referred to as *publishing functions*. Since XML is universal and ubiquitous, the users of any of the systems in a federation can access any other database in the federation using a common set of tools. In particular queries and/or update operations can be defined in terms of the exposed XML representation of the data using an XML query language (e.g., XQuery [17]), and then be translated into equivalent statements over the original databases. Because publishing functions can be defined between pairs of independent systems, this framework is very generic, capturing various scenarios in data integration (mediators, P2P, etc.).

While a mature body of research exists in related areas such as schema translation and publishing relational data as XML, there has been comparatively little attention paid to security issues in data exchange. This is troublesome, as the growing availability of data through avenues such as the Web has meant that the traditional application assumption of a relatively small, stable, and closed set of users *local* to an organization (i.e. employees of a company) has increasingly given way to settings in which a much larger set of actors – many of whom reside *outside* the organization – engage in dynamic and short-lived interactions. From a security standpoint, one must consider all possible interactions and the potential attack vectors they may introduce. In this thesis, we address two security issues relevant to heterogeneous data exchange: translation of locally-defined access control policies into an equivalent policy expressed at the federation level, and disclosure control of federated data.

The key difficulty in integrating several locally-defined access control policies into a unified federation policy is ensuring that the semantics of each original policy are obeyed (i.e., that each user has access to all federated data needed to fulfill their requirments), while ensuring that any conflicts are resolved in such a way that no federated user gains access to *more* data than they need (i.e., ensuring that the "need-to-know" principle is not violated). This problem is the focus of Chapter 3.

Even after one ensures that access control policy translation has been successfully carried out, there remains a danger that an adversary can exploit their knowledge of accessible

portions of federated data to determine, with a high probability of success, the values of one or more federated data items which are not accessible to him under the stated access control policy. Guarding against such risks produces a trade-off between security and data availability: to make a system completely secure from such disclosures may dramatically reduce the proportion of federated data each user can access. Conversely, data availability for each user can be increased by instead allowing certain disclosures, as long as the degree of risk they present falls below a specific threshold. We study disclosure control and this trade-off in Chapters 4 and 5.

1.1 Data Federations

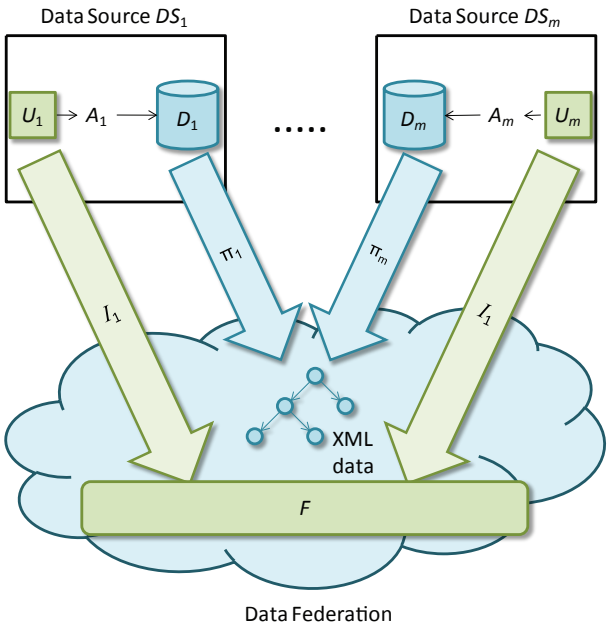


Figure 1.1: Thesis application setting.

The application setting considered in this thesis is captured in Fig. 1.1. A set of *data sources* $\{DS_1, \ldots, DS_m\}$ agree to export data into a larger *data federation*, where it can be queried by a set of users using a common interface. More specifically, each data source DS_i defines a corresponding *publishing function* Π_i that acts as a contract between it and the other entities (data sources and users) in the federation, detailing exactly what data will be exported from a *source database* D_i, and how it will be represented to users within the federation. In this thesis, we assume that each publishing function specifies a transformation

of relational data into an XML representation. This is based on XML's emergence as the dominant means for exchanging and integrating relational data originating from distinct sources [49, 61].

In addition, we assume that each data source DS_i has an existing *access control policy* A_i defining access privileges to D_i for a set of *local identities* U_i. We assume that each DS_i applies an *identity mapping function* I_i that serves to map each local identity in U_i to one or more *federated identities* within the set of all federated IDs, F [1].

Research interest in *data federations* dates back to the early 1980s. Sheth and Larson [112] identified several factors defining the trade-off between data source autonomy on the one hand, and the federation's functionality and transparency from the perspective of federated users on the other. These include *design autonomy*, indicating the degree to which individual data sources are able to define their own methods for storing data, and which query language is used to access it; *communication autonomy*, which references the freedom for individual data sources to choose which other sources they interact with; *execution autonomy*, which defines the ability for data sources to perform local operations on data without interference from the outside federation; and *association autonomy*, which allows a data source to decide whether (and how) to share its data with federated users. Jonscher and Dittrich [70] defined a specific type of association autonomy, called *authorization autonomy*, referring to the ability of a data source to specify which external accesses on local data are to be allowed, and which are to be refused. It is this last type of autonomy that is most relevant to this thesis.

Another important distinction has historically been drawn between *tightly-* and *loosely-coupled* federations [112]. The latter are characterized by greater transparency, as federated users are able to directly query the schema of a data source. Such systems are more flexible in the sense that data sources can join and leave the federation with little impact on the remaining participants, at the expense of making querying more complicated: a federated user must first determine which source(s) contain relevant data, and then formulate a sepa-

[1] Our notion of an identity mapping function is not to be confused with the concept of an *identity function* in mathematics, i.e., one in which each domain element is mapped to itself. In the present context, *identity mapping* refers to the process of converting a local user identity into one or more identities defined at the federated level.

rate query over the exported schema for each such source. Tightly-coupled systems perform a conversion from local schemas to a single federated schema, allowing federated users to retrieve data with a single query. The need to define a schema translation for each data source makes extending the federation to accommodate new sources a complex process.

The initial work on data federations presaged the proliferation of applications in the last twenty years in which negotiated data exchange takes place between multiple, autonomous actors. In some cases, such transactions are very ad hoc in nature, while in others the interactions between actors are carefully choreographed. Additionally, the relationships between actors are in some cases extremely short-lived, while in others partnerships persistent over several data exchange operations. We now review these application classes.

- *Peer-to-peer (P2P) networks* [6] feature multiple data sources (peers), connected by an abstract *overlay network*. The overlay network provides functionaility allowing peers to discover and communicate with one another without worrying about details of the underlying physical networks, such as firewalls and network address translation (NAT) schemes. P2P networks can be *structured* or *unstructured*. In structured networks, an indexing data structure such as a distributed hash table is used to allow peers to efficiently locate resources in a uniform way. While such schemes enable efficient query routing, they require extra overhead in order to ensure that the indexing structure remains consistent as peers join and leave the network (network churn). Unstructured networks lack indexing infrastructure, typically resorting to query flooding (in which each peer issues the query to known neighbours, and relies on them to pass the query on to their neighbouring peers, and so on) for resource discovery. In this sense, structured P2P networks resemble tightly-coupled data federations, while unstructured ones are more similar to loosely-coupled federations.

- *Web services applications* [19] feature ad hoc data transactions in which a client discovers a compatible service provider via means of a service registry, then uses the accompanying service description to formulate a request to the provider. XML dialects are used to encode service descriptions, invocations, and responses. Each client and provider has a high degree of association autonomy, since they ultimately

choose which provider (or clients, respectively) to contact (or respond to).

- *Data integration systems* [61] are obvious descendants of the data federation systems proposed in the 1980s. Clients express their queries over a global schema, and these queries are then transparently re-written by middleware into equivalent queries over relevant source databases.

While different in many important respects, these application classes do share some commonalities. All of them feature independent actors with a desire to share data with one another, while maintaining a degree of autonomy in terms of which parties they interact with, and in deciding which portions of their data will be shared. It is precisely this autonomy and independence that leads to heterogeneity in terms of how data is organized, and in how accesses to data are regulated.

1.2 Data Security

Having described the data federation setting, we now turn to the topic of securing data. In determining the precise issues that need to be addressed, it is beneficial to first review the requirements for a system to be considered "secure". A complete data security solution provides the following properties [16]:

- *Confidentiality.* This requires that data be protected from unauthorized disclosure and observation.
- *Integrity.* Data must be protected from unauthorized modifications.
- *Availability.* Protections from software and hardware malfunctions, as well as malicious attacks, must be incorporated into data applications in order to ensure that data remains available to users when it is needed.

Integrity is typically provided by a non-repudiation mechanism, such as requiring that each data modification be accompanied by a digital signature. This serves to both identify the party that caused the modification, and as a historical record preventing them from later denying having made the change. Properly-defined access control policies can also serve

to protect integrity by preventing unauthorized users from altering data. Availability is often achieved by a combination of techniques, including concurrency controls (ensuring multiple users can access the same data, and ensuring that updates to data are executed in the proper order) and recovery subsystems, such as those found in modern database management systems.

The main focus of the thesis is on data confidentiality issues. Confidentiality can be *partially* enforced via access controls, yet additional measures are needed to guard data against disclosures from *indirect* means such as inferences based on combining accessible data with prior background knowledge to reason about inaccessible data [16].

1.3 Thesis Statement

This thesis strives to provide a comprehensive study of two security topics relevant to heterogeneous data federations, namely access control policy translation and the prevention of unintended information disclosure. In the case of access control policy translation, we aim to demonstrate that automated methods for performing policy translation are definable in many realistic application scenarios, and further, that the same is true for the closely related problems of policy verification and minimization. Of equal importance, we seek to identify those scenarios in which the policy translation, verification, and minimization problems prove intractable.

Our treatment of information disclosure intends to show that federated databases comprised of XML data are subject to a variety of disclosure risks. In a departure from earlier approaches, we aim to demonstrate that certain risk types are detectable at database design time, and, by moving detection to this earlier phase, one improves system performance by reducing the number of calculations that must be performed per query to include only those remaining risks that cannot be measured at design time. Our chosen strategy to validate this hypothesis is to develop design-time and query-time algorithms for detecting and removing the respective disclosure risk types, and to illustrate the efficiency and scalability of these solutions via experiments carried out upon a prototype implementation using a variety of synthetic and realistic data sets.

1.4 Summary of Contributions

This thesis provides the following contributions.

- *Expressibility of translated policies:* we show that it is always possible to preserve the semantics of arbitrary relational access control policies, defined over source databases, within a federated access control policy defined over XML representations of this data. This is demonstrated by providing a polynomial-time algorithm for automating the access control policy translation process. (Chapter 3)

- *Verification of translated policies:* we discuss the complexity of testing whether a translated access control policy over federated data correctly enforces a set of relational ACPs defined over source databases, and show this problem is intractable even for many restricted scenarios. Further, we discuss some implications of these findings. (Chapter 3)

- *Representing translated policies in an XML access control policy expression language:* we provide an algorithm for encoding secure publishing functions in the eXtensible Access Control Markup Language (XACML) [91], thereby easing portability and interoperability with other systems. (Chapter 3)

- *Minimization of translated policies:* we provide an algorithm for minimizing the number of rules in the translated access control policy, and for simplifying the relational queries appearing in each rule. (Chapter 3)

- *Classification of disclosure risks at the federation level:* we provide a characterization of common types of disclosure risks applicable to XML databases, based on whether they can be detected at design-time (schema-based risks) or only at query-time (instance-based risks). (Chapter 4)

- *A design-time algorithm for removing schema-based risks:* we provide an algorithm that performs static analysis on an input schema description of an XML database and an access control policy defined over that database, outputting a customized schema

description for each federated user in which all schema-based risks are removed. (Chapter 4)

- *Efficient computation and storage of XML probability models:* we introduce a method for modeling the current contents of an XML database based on the use of probabilistic tree grammars. (Chapter 5)

- *An information-theoretic measure for monitoring instance-based risks:* we propose a measure for calculating the magnitude of instance-based disclosure risks, and illustrate a query-time algorithm for removing instance-based disclosure risks based on this measure. (Chapter 5)

- *Experimental validation:* We also present experimental results indicating the scalability of the algorithms presented in Chapter 5. (Chapter 6)

Chapter 2

Background

In this chapter, we provide a review of concepts and definitions that are relevant to future chapters of the thesis.

2.1 Computational Complexity

In this section, we briefly review various computational complexity classes that will be referred to later in the thesis.

2.1.1 Decision Problems and Languages

Complexity analysis most often deals with *decision problems*: those for which the set of all possible instances of the problem can be disjointly partitioned into "yes" and "no" instances [101]. Intuitively, "yes" instances are those that satisfy the input parameters of the decision problem, while all remaining instances are classified as "no" instances. For example, the boolean satisfiability decision problem takes as input a boolean formula ϕ, and asks whether there exists a satisfying truth assignment for the boolean variables in ϕ. In this case, the set of "yes" instances consists of all boolean formulas that are satisfiable, while "no" instances include all boolean formulas for which there exists no satisfying variable assignment.

An alternative representation of a decision problem $\mathcal{DP}$ is to consider it as a string language $L_{\mathcal{DP}}$. Under this interpretation, each problem instance $I \in \mathcal{DP}$ is encoded into a specific string $L_{\mathcal{DP}}(I)$ within this language using a consistent translation scheme. Returning to the example of boolean satisfiability, one could represent an instance consisting of

a boolean formula of n literals $x_1, \ldots, x_n$ as a binary string of length n, where a specific truth assignment is indicated by setting the i-th bit iff x_i is `true`.

2.1.2 P and NP

To classify the complexity of a decision problem, one often considers the amount of time required to determine whether an input problem instance is a "yes" or "no" instance. Time requirements are measured relative to the size of the input problem instance I. Intuitively, the size of I, denoted as $|I|$, is precisely that of its string language representation $L_{\mathcal{DP}}(I)$.

Two fundamentally important complexity classes connect time requirements for decision problems with deterministic and non-deterministic models of computing. The class **P** contains those decision problems for which all instances can be classified as a "yes" or "no" instance by a *deterministic* program in an amount of time that is upper-bounded by a polynomial function in $|I|$. The class **NP** includes those decision problems for which all "yes" instances can be *verified* in an amount of time that is upper-bounded by a polynomial function in $|I|$. This corresponds to a non-deterministic program that is capable of instantaneously "guessing" a solution, and then verifying that it yields a "yes" instance. Clearly, $\mathbf{P} \subseteq \mathbf{NP}$, since any problem that can be decided in polynomial time must also be verifiable within polynomial time. It is widely conjectured that $\mathbf{P} \subset \mathbf{NP}$, although no formal separation proof yet exists.

The decision problem of evaluating a boolean formula over a specific truth assignment is in **P**, since at each literal position, one only needs to determine which variable is referenced, substitute the corresponding truth assignment for that variable, and finally, evaluate the formula resulting once all literals have been instantiated in this manner. The overall time requirement is polynomially-bounded by the number of literals, and further, on conclusion, each problem instance has been classified as a "yes" or "no" instance. On the other hand, boolean satisfiability is in **NP**, since the determination of each "yes" instance requires one to "guess" a truth assignment for literals in the input boolean formula, and then evaluate this truth assignment over the formula to ensure that it is in fact a satisfying truth assignment.

Another important complexity class is **coNP**, composed of all decision problems for which *succinct disqualifications* exist. Each decision problem $\overline{\mathcal{DP}} \in \mathbf{coNP}$ is the comple-

ment of a decision problem $\mathcal{DP} \in \mathbf{NP}$; that is, an instance I is a "yes" instance of $\overline{\mathcal{DP}}$ iff it is a "no" instance of $\mathcal{DP}$. An example of a **coNP** decision problem is boolean unsatisfiability, in which "yes" instances correspond to an input boolean formula for which there is no satisfying truth assignment: while *any* satisfying truth assignment serves as a succinct disqualification for "no" instances, determining conclusively that the input formula of n variables is unsatisfiable requires one to examine *all* 2^n possible truth assignments and test that each does not satisfy the formula.

2.1.3 Reductions and Completeness

A common method of establishing a lower bound on the difficulty of a decision problem is to show that it is at least as hard as another decision problem, for which the decision complexity is known. This is accomplished by demonstrating an efficient reduction from the language corresponding to the former problem to that of the latter problem.

Definition 2.1.1 (Polytime Reducibility)**.** *A language L_1 is* polytime reducible *to another language L_2 if there is a deterministic function R from strings to strings computable in time upper-bounded by a polynomial in $|x|$ such that, for all inputs x, $x \in L_1$ if and only if $R(x) \in L_2$.* □

It is also beneficial to establish the group of "most difficult" decision problems within a complexity class. Such problems provide a natural starting point for a reduction.

Definition 2.1.2 (Completeness of Complexity Classes)**.** *A language L is said to be* complete with respect to complexity class C*, or* C-complete*, if (1) $L \in C$ and (2) every language in C is polytime reducible to L.* □

A language is $\mathbf{coNP}$-complete iff its complementary language is $\mathbf{NP}$-complete. Boolean unsatisfiability and boolean satisfiability, discussed earlier, provide an example of a $\mathbf{coNP}$-complete and an $\mathbf{NP}$-complete problem, respectively.

2.1.4 Polynomial Hierarchy

While **NP**-complete problems represent the most difficult ones solvable by a non-deterministic program within polynomial time, it is possible to consider even more difficult problems by

augmenting the computational model to permit a program to consult an *oracle* for another decision problem during its execution, and to incorporate the answers received by the oracle in its future computation. We denote by M^A that a program for a decision problem residing in complexity class M has access to an oracle for a decision problem residing in complexity class A. The *polynomial hierarchy* defines a set of complexity classes beyond **NP**, organized according to the complexity of the decision problem for which the calling program possesses an oracle for, and additionally the computational model (deterministic or non-deterministic) afforded to the calling program itself.

Definition 2.1.3 (Polynomial Hierarchy of Complexity Classes)**.** *The* polynomial hierarchy *consists of the following sequence of complexity classes:*

$$\begin{aligned} \Delta_0^P = \Sigma_0^P = \Pi_0^P \textit{ and for all } i \geq 0, \\ \Delta_{i+1}^P = \mathbf{P}^{\Sigma_i^P} \\ \Sigma_{i+1}^P = \mathbf{NP}^{\Sigma_i^P} \\ \Pi_{i+1}^P = \mathbf{coNP}^{\Sigma_i^P} . \end{aligned}$$

The cumulative polynomial hierarchy is the class $\mathbf{PH} = \bigcup_{i \geq 0} \Sigma_i^P$. □

2.2 Relational Model and Query Languages

In this section, we summarize the relational data model and several important query classes within this model.

2.2.1 Relational Model

In the relational model, an n-ary *relation* R is assigned a set of *attributes* $\{A_1, \ldots, A_n\}$. Each attribute A_i is associated with a finite *domain* $dom(A_i)$. An *instance* of a relation R is defined as a set of mappings $\{t_1, \ldots, t_k\}$ such that, for each *tuple* $t \in R$, $t[A_i] \in dom(A_i)$. Finally, a *relational schema* $S = \{R_1, \ldots, R_m\}$ is a set of relations, while a *database instance* D of S is a set containing exactly one instance of each relation in S.

2.2.2 First Order Queries and Codd's Relational Calculus

The *tuple relational calculus* was introduced by Codd as a language for expressing declarative queries in the relational model [32]. It is essentially equivalent in expressive power to first-order logic [115], and hence, we will often refer to relational calculus expressions as *first-order queries* throughout the thesis.

Each *tuple relational calculus expression* has the form $\{t \mid \psi(t)\}$, where t is a *tuple variable* denoting a tuple of some fixed length, and ψ is a *formula* built from *atoms* and various operators. The atoms of formulas are of the following three types:

- $s \in R$, where R is a relation and s is a tuple variable. This atom designates that tuple s is contained in R.
- $s[i]\,\theta\,u[j]$, where s and u designate tuple variables and θ is an operator from $\{<, > , \leq, \geq, =, \neq\}$. This atom indicates that the i-th attribute of tuple s is related to the j-th attribute of tuple u by the operator θ.
- $s[i]\,\theta\,a$ and $a\,\theta\,s[i]$, where θ and $s[i]$ are as above, and a denotes a constant.

It is important to distinguish tuple variables that are *bound* from those that are *free*. An occurrence of a tuple variable t in a formula is said to be *bound* if that variable is associated with an existential quantifier ($\exists$) or a universal quantifier ($\forall$). Any unbound occurrence of a tuple variable is said to be *free*.

Formulas in tuple relational calculus expressions are defined recursively as follows.

1. Every atom is a formula. All occurrences of tuple variables mentioned in the atom are free in the derived *atomic formula*.
2. If ψ_1 and ψ_2 are formulas, then $\psi_1 \wedge \psi_2$ (conjunction of formulas), $\psi_1 \vee \psi_2$ (disjunction of formulas), and $\neg\psi_1$ (negation of a formula) are all formulas. In each of the derived formulas, occurrences of tuples variables are free or bound as they were in ψ_1 or ψ_2, depending on where they originally occurred.
3. If ψ is a formula, then $(\exists s)(\psi)$ is a formula. Each free occurrence of s in ψ is bound to $(\exists s)$ in $(\exists s)(\psi)$. All other occurrences of tuple variables in ψ are free or bound in

$(\exists s)(\psi)$ as they were in ψ.

4. If ψ is a formula, then $(\forall s)(\psi)$ is a formula. Each free occurrence of s in ψ is bound to $(\forall s)$ in $(\forall s)(\psi)$. All other occurrences of tuple variables in ψ are free or bound in $(\forall s)(\psi)$ as they were in ψ.

5. If ψ is a formula, then (ψ) is a formula, allowing one to override the default ordering of operations. In evaluating a formula, the default order of precedence places comparison operators θ highest, followed by $\exists$ and $\forall$, followed by $\neg$, $\wedge$, and $\vee$, in that order.

6. Nothing else constitutes a formula.

In a well-formed tuple relational calculus expression $\{t \mid \psi(t)\}$, t must be the only free tuple variable in ψ.

2.2.3 Conjunctive Queries

Conjunctive Queries (*CQ*) comprise a subset of first order queries inductively defined as follows.

1. Any atomic formula restricting θ operations to equality $(=)$ is a conjunctive query.

2. Any formula formed from the conjunction $(\wedge)$ of two conjunctive queries is a conjunctive query.

3. If ψ is a conjunctive query, then $(\exists s)(\psi)$ is a conjunctive query. Each free occurrence of s in ψ is bound to $(\exists s)$ in $(\exists s)(\psi)$. All other occurrences of tuple variables in ψ are free or bound in $(\exists s)(\psi)$ as they were in ψ.

Conjunctive queries have two desirable properties: they capture a large portion of the common queries one issues on a relational database, and can be evaluated and optimized more efficiently than arbitrary first order queries. In particular, Chandra and Merlin [28] proved that the combined evaluation complexity of conjunctive queries is **NP**-complete, while combined evaluation for arbitrary first order queries is undecidable. The same results hold for the *query minimization* problem, in which for an input formula ψ one seeks to

discover the smallest formula ψ_{min} that is equivalent to ψ (where equivalence holds iff ψ and ψ_{min} evaluate to an identical answer on the same database instance D, for all such database instances).

as is the query minimization problem. When one allows θ operations other than =, the evaluation and optimization problems become more difficult to decide.

For certain subclasses of conjunctive queries, the associated evaluation and optimization problems become even more tractable.

Acyclic Queries

One common way of visualizing a conjunctive query is as a *hypergraph*, in which the nodes correspond to the vatiables and constants defined in the query, and hyperedges to subgoals within the query. An important subclass of conjunctive queries are those whose hypergraphs are acyclic. These *acyclic queries* have a combined polynomial time complexity for the evaluation and minimization decision problems [29].

Conjunctive Queries with No Self-Joins

Another interesting subclass of conjunctive queries includes those featuring no self-joins; that is, no relation appears in more than one clause in the query. Since each clause is independent of the others, in the sense that each operates over a disjoint set of data objects, they are already minimized.

2.2.4 Relationship to SQL

The *Structured Query Language (SQL)* has long been the most widely-used query language for relational database systems. It was inspired by Codd's relational calculus, yet differs in some important respects. One is that while the calculus assumes set semantics, SQL uses bag semantics, allowing duplicate tuples to appear in relations and in query results. Another is the adoption by SQL of three-valued logic: `true`, `false`, and `NULL`. `NULL` serves as a placeholder in cases where an attribute value is unknown. Thirdly, it is not possible to directly express universal quantification in SQL; one must instead resort to using negated existential quantification.

Each of the query classes described above corresponds to a fragment of SQL. Conjunctive queries (without inequalities) is equivalent to SQL queries using select, project, and join operations(i.e., `SELECT`, `FROM`, and `WHERE` statements containing only equality comparisons and the `AND` connective). Permitting inequalities in conjunctive queries is equivalent to `SELECT-FROM-WHERE` SQL queries with $\neq$, $<$, $\leq$, $>$, $\geq$ operators permitted in the `WHERE` clause. The full power of SQL:99 lies beyond first-order logic (for example, one cannot express reachability as a first-order query, but recursive functionality added in SQL:99 *does* permit reachability to be expressed). In particular, SQL:99 is equivalent in expressive power to first-order queries extended with aggregation functions [80].

2.2.5 Query Containment and Equivalence

One of the classic problems in database theory is *query containment*, that is, deciding whether for all possible database instances for a fixed schema S, the answer for one query forms a subset of the answer of the second query.

Definition 2.2.1 (Relational Query Containment)**.** *Query ϕ_1 is* contained *within another query ϕ_2 over a relational schema S if for every instance database D conforming to S, $\phi_1(D) \subseteq \phi_2(D)$. We denote the containment of ϕ_1 by ϕ_2 as $\phi_1 \subseteq \phi_2$.* □

A close relationship exists between query containment and another common decision problem, *query equivalence*.

Definition 2.2.2 (Relational Query Equivalence)**.** *If $\phi_1 \subseteq \phi_2$ and $\phi_2 \subseteq \phi_1$, we say that ϕ_1 and ϕ_2 are* equivalent*, written as $\phi_1 \equiv \phi_2$.* □

Complexity results for query containment have been established for each of the query classes we have described earlier in this section. Table 2.1 summarizes these results.

2.3 Access Control Models

An *access control model* provides mechanisms for specifying and enforcing limitations on accesses to system resources. Specification is most often done through an *access control policy (ACP)*, which consists of a set of *policy rules*. Although the precise syntax and semantics of policy rules differs according to the access control model under use, they

Query Class	Complexity of Query Containment
First order queries	undecidable
Conjuctive queries with inequalities	Π_2^P-complete
Conjunctive queries without inequalities	**NP**-complete
Acyclic queries	PTIME
Queries without self-joins	PTIME

Table 2.1: Complexity of deciding query containment for various query classes.

typically contain a set of *subjects* (or *users*), which are the local identifiers to which the current rule applies; a set of *objects* defining which system resources the current rule is defining access over; and *actions*, indicating the operations that designated subjects will be permitted to perform over the specified objects.

Access control policy enforcement refers to the techniques used to ensure that a designated policy is correctly applied to regulate access to objects. These can include centralized components such as reference monitors, or in the case of decentralized enforcement, cryptographic techniques. At the most general level, access control models can be grouped into three categories, *discretionary*, *mandatory*, and *role-based*, based on how they assign and delegate access permissions.

2.3.1 Discretionary Access Control Models

In discretionary models, access is regulated based on the identity of the requestor. Policy rules in which the requestor appears as a subject are used to determine whether the requested access is granted or denied. The name of the model derives from the ability to assign users the capability of propagating their permissions over an object to other users, at their discretion. An *administrative policy* dictates how permissions are granted and revoked.

Discretionary models are prone to a significant security vulnerability stemming from the lack of separation between subjects and users. While the latter represent passive entities who connect to the system, the former can additionally include the processes and programs that the user invokes after connecting to the system. Each such process will inherit the permissions assigned to the user who invoked them, leaving the system vulnerable to attacks from malicious programs. One example is the *trojan horse* attack, in which a program contains extra functionality which is unknown to the initiating user, and leverages the per-

missions assigned to the user to carry out an attack (such as installing a virus, or deleting all files belonging to the user). The delegation model used in discretionary models exacerbates the vulnerability to such attacks, since once a permission is acquired by a malicious process, it can bestow this access to another party without the knowledge of the security administrator.

2.3.2 Mandatory Access Control Models

In mandatory models, there is a clearer separation between users and subjects. Access decisions are solely based on regulations developed by a central authority (i.e., there is no delegation of permissions as in discretionary models). The most common mandatory systems employ a *multilevel security policy*, in which a *classification level* is assigned to each data object and each user [106]. The set of all classification levels assigned in the system typically forms a partial order, representable as a lattice structure. One level l_1 is then said to *dominate* another level l_2 if l_1 resides at the same or higher level in the lattice than l_2. In *secrecy-based* mandatory policies, the classification level assigned to an object reflects its sensitivity, or the relative amount of damage that would be caused if the object were to be accessed by an unauthorized user. The classification level assigned to a user indicates their *clearance*, or trustworthiness not to disclose classified objects to unauthorized parties. Such policies serve to limit indirect flows of information, as well as enforce the "need-to-know principle" by limiting user access to those data objects needed to perform their duties, by basing access decisions on a comparison of the levels assigned to the requested object and the user issuing the request. In Bell-LaPadula [10] policies, a user is allowed to read only those objects whose classification level is dominated by their own classification level, and to write only those objects whose classification level dominates their own. Such policies prevent most overt leaks (such as the Trojan horse attack mentioned above), as information cannot flow from a higher classification level to a lower one.

Mandatory access control models suffer from certain shortcomings. These include the inability to prevent covert flows of information based on observable properties of the system state, such as *timing attacks*, in which an adversary is able to infer characteristics about a classified object based on the response time of the access request [106]. Another weakness

is that mandatory policies offer less flexibility than alternative models, requiring one to assign each new user and object to a specific classification level when they join the system. Great care must be taken to ensure that the set of all classification levels is diverse enough to enable each user to access those data objects needed to carry out their responsibilities, and no more.

2.3.3 Role-Based Access Control Models

Discretionary and mandatory models often prove difficult to translate into real-world organizational structures. *Role-based access control (RBAC)* models focus not only on identifying who the user is, but also what their organizational responsibilities are. In a sense, they strike a balance between the flexibility of explicit authorizations (as in discretionary models) with the need to enforce organizational constraints on access (as in mandatory models) [55]. This is done by adding an extra layer of indirection in the form of *roles*. Instead of assigning permissions directly to individual users, they are instead assigned to roles. A user is then assigned to the set of roles relevant to their responsibilities, and inherits the union of permissions assigned to each such role. Role hierarchies can be defined to model complex organizational structures in which one role's permissions form a superset of those assigned to other roles. A significant advantage of RBAC models is that they promote a form of logical independence; when an administrator crafts a policy, they are able to clearly separate user identities from their responsibilities. In mandatory models, there is no such separation since a user is directly assigned a single classification level representing one set of responsibilities and, hence, it is not trivial to express situations where a user may take on different responsibilities at various phases of an application.

2.3.4 Access Control Models in Modern Database Management Systems

Discretionary Models

Early discretionary access control models for relational databases [47, 59] heavily influenced the access control model eventually adopted as part of the SQL standard [86]. Most modern relational database management systems employ similar models to specify which users within a set of local identities U have access to particular database objects (tables and

columns). It is up to the owner (typically, the creator) of a database table to determine appropriate access permissions for specific users. In addition, the grantor of a permission can optionally allow the grantee to bestow the permission (or a more restrictive subset thereof) to other users. The SQL standard includes `GRANT` and `REVOKE` statements which are used to construct a discretionary access control policy over a given relational database. Each such statement serves to grant (or revoke) permissions on a set of *database objects* (cells) to one or more users (or subjects). A `GRANT` statement may optionally contain the `WITH GRANT OPTION`, which designates that the subject is granted the ability to pass the granted privilege along to other users.

In Chapter 3, we use the following concise notation to represent the contents of SQL `GRANT` statements. [1]

Definition 2.3.1 (Relational Access Control Policy)**.** *An access control policy rule is a 5-tuple* $\langle s, q, p, go, g \rangle$*, where* s *is a set of users (subjects) drawn from* U*;* q *is a relational query;* p *is a set of permissions drawn from the set* $\{read, insert, delete, update\}$*;* go *is a boolean value taking the value* true *if the grant option has been granted to* s*, and* false *otherwise; and* $g \in U$ *indicates the permission grantor. An access control policy (ACP) consists of a set of access control policy rules.* □

The semantics of an access control policy rule are that each user in s is granted the permissions in p over the relation r constructed by the query q. If go is true, then each user in s may arbitrarily grant to any other user(s) a permission set $p' \subseteq p$ over r. Additionally, note that our notation allows us to define access control over virtual relations (views) as well as base relations, by expressing the view definition as a query over the corresponding base relation(s). We denote by $accessible(u, p)$ (respectively, $inaccessible(u, p)$) the set of all accessible (respectively, inaccessible) database objects in D for user u under the context of permission p.

[1] We assume that the existing access control policy defined over each relational database is well-defined, and, in particular, contains no conflicting policy rules.

Mandatory and Role-Based Models

Attempts to incorporate mandatory access control models in databases have mainly been limited to research prototypes, in the form of multilevel databases. A significant barrier to their acceptance by industry has been the so-called "polyinstantiation problem" [69, 107], where the requirement to store multiple tuples with the same primary key (each one assigned to a distinct classification level), can greatly complicate consistent view construction and index maintenance, and can also dramatically increase the storage requirements for a single table. In recent years, the introduction of label-based access controls [65, 96] has led to a resurgence in interest in mandatory models within commercial database management systems. The addition of these features has been motivated by the increasing number of applications which integrate data from different sources within the same relation, causing a desire for more fine-grained access control specification down to the cell level.

Database vendors have been slow to adopt role-based access control models. While basic features of RBAC have been added to the most recent versions of SQL, these have been incorporated to varying degrees by commercial systems. In particular, most DBMSs do not permit the definition of role hierarchies [16].

Summary

An important point to be made is that, on its own, no access control model is capable of forming a complete security solution. Access control models can assist in protecting data confidentiality by restricting direct accesses to data, but additional mechanisms are needed to prevent information leakage caused when an adversarial user is able to combine prior knowledge with the results of allowable queries to reduce their uncertainty with respect to the answers of disallowed queries.

2.4 XML Data Model

The *eXtensible Markup Language* (*XML*) is a textual encoding format consisting of *elements* that surround text segments, or *parseable character data* (*PCDATA*), with descriptive tags serving to indicate application-specific data semantics. XML allows elements to be nested, creating a hierarchical structure. We say that an XML document is *well-formed* if it

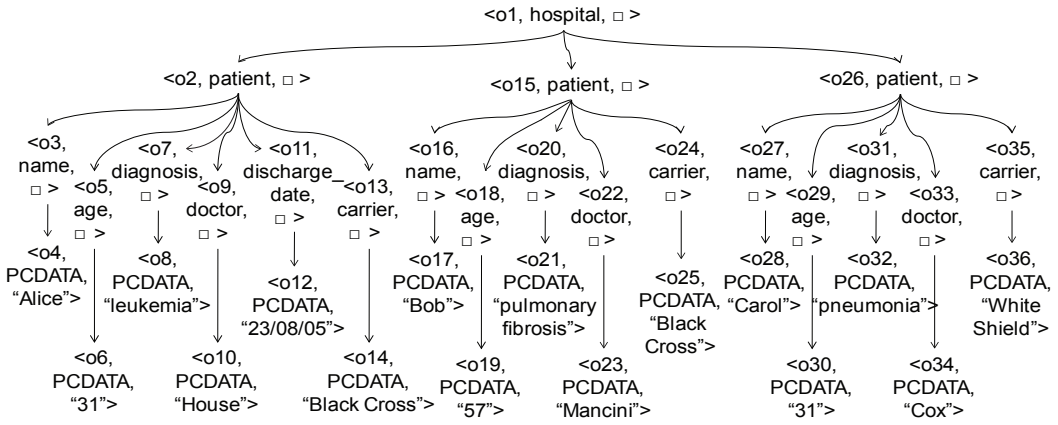


Figure 2.1: Tree representation of a hospital database.

conforms to the syntactic requirements of the XML standard [21], including proper nesting of elements and presence of a unique root element. Additional metadata about elements can be encoded using *attributes*.

2.4.1 XML Trees

XML documents are typically modeled as ordered, unranked, labeled trees in which the internal nodes correspond to elements and attributes, and leaf nodes to data values (PCDATA) and attribute values. The induced tree hierarchy preserves the nesting relationships between elements, attributes, and values within the document.

Fig. 2.1 depicts an XML tree containing hospital patient records. Following convention, we distinguish attribute nodes by prefixing their labels with the `@` symbol. We now provide more formal definitions of XML trees and associated concepts that will be referred to in Chapters 4 and 5.

Definition 2.4.1 (XML Tree)**.** *An* XML tree (database) *is an ordered, rooted directed tree* $T = (V, E, r)$*, where* V *is the set of nodes,* E *is the set of edges, and* $r \in V$ *is the root node of* T*. For a node* v*,* $ordinal(v)$ *denotes the position of* v *according to a preorder traversal of* T*;* $type(v)$ *denotes an element/attribute name, or* `PCDATA` *if* v *is a leaf node storing element content; and* $value(v)$*, a string representing an attribute value or element content, or* □ *if* v *is a non-PCDATA node.* □

Definition 2.4.2 (Location Path)**.** *The* location path *of node* v *contains the labels of the*

nodes in the path from the root of the tree into v. It is defined recursively as follows (below, + indicates string concatenation and $parent(v)$ denotes the parent node of v):

- *if v is the root node, then $path(v) = / + type(v)$.*
- *if $type(v) =$* `PCDATA`*, then $path(v) = path(parent(v)) +$* `/text()`*.*
- *otherwise, $path(v) = path(parent(v)) + / + type(v)$.* □

We distinguish all nodes having the same location path l. We say that the set of such nodes forms the *node class* of l:

Definition 2.4.3 (Node Class)**.** *The* node class *of path expression l is defined as*

$$class(l) = \{ordinal(v) \mid v \in V \wedge path(v) = l\}\,. \quad \square$$

We will refer to the set of PCDATA (or as applicable, attribute) values belonging to instances of the specific location path l in T as the *active domain* of l:

Definition 2.4.4 (Active Domain)**.** *The* active domain *of a location path l is defined as*

$$adom(l) = \{value(v) \mid v \in V \wedge type(v) = \texttt{PCDATA} \wedge path(parent(v)) = l\}\,. \quad \square$$

Example 2.4.5. *For the hospital database of Fig. 2.1,*

$path(o3) =$ `/hospital/patient/name`*,*

$path(o36) =$ `/hospital/patient/carrier/text()`*,*

$class($`/hospital/patient/age`$) = \{o5, o18, o29\}$*, and*

$adom($`/hospital/patient/name`$) =$ {*"Alice"*, *"Bob"*, *"Carol"*}. □

2.4.2 XML Schema Languages

In addition to well-formedness, a second notion of correctness for XML documents is *validity* with respect to a specified *schema definition*. While it is possible for a document to be well-formed yet invalid, any document which is valid with respect to any particular schema definition must additionally be well-formed. A schema definition specifies legal names for elements and attributes, along with patterns for nesting and ordering of elements. Each schema definition is an instance of a particular XML *schema language*. Such languages vary in terms of their expressiveness and syntax; below, we summarize the two most common schema languages, DTD and XML Schema.

```
<!ELEMENT course (name,instructor,students)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT instructor (#PCDATA)>
<!ELEMENT students (student)+>
<!ELEMENT student (a1,a2,midterm,(project|finalexam))>
<!ATTLIST student id ID #REQUIRED>
<!ELEMENT a1 (#PCDATA)>
<!ELEMENT a2 (#PCDATA)>
<!ELEMENT midterm (#PCDATA)>
<!ELEMENT project (#PCDATA)>
<!ELEMENT finalexam (#PCDATA)>
```

Figure 2.2: A DTD for course enrollment documents.

Document Type Definition (DTD)

A *Document Type Definition* (*DTD*) allows one to specify an extended context-free grammar containing a separate production rule for each allowable element and attribute. Hence, an individual document is valid with respect to the DTD only if that document can be derived through application of the DTD's rules, beginning with the rule defined for the root element. DTD rules may contain alternative expressions (specified by a `|` symbol), denoting that any one of the indicated possibilities are acceptable. In addition, rudimentary element occurrence constraints can be specified in rules: a `?` symbol indicates the preceding element (or sequence of elements) is optional, a + indicates the preceding element type occurs one or more times, and a $\star$ denotes zero or more occurrences. An example DTD for course enrollment documents is listed in Figure 2.2. This DTD illustrates several of the rule types previously described. The rule for `student` elements contains a conditional allowing either a `project` or `finalexam` element to appear as the fourth child. An occurrence constraint is specified for `students` elements, forcing each to have at least one child `student` element.

In addition, it is possible to define several *modifiers* for XML attributes. Some modifiers allow one to control the *data type of the attribute*, which defaults to `CDATA`, or *character data*. The `ID` modifier serves as a primitive form of key constraint, indicating that each occurrence of the associated attribute type within a single document must have a unique value. In the example DTD, the `id` attribute belonging to `student` elements has the `ID`

modifier. The `IDREF` modifier allows one to specify foreign key relationships; in particular, each occurrence of an `IDREF` attribute must take the value of an `ID` attribute. The `IDREFS` modifier generalizes this behaviour to allow the attribute value to be a sequential list of `ID` attribute values. Other modifiers allow one to indicate occurrence constraints on a modifier, within the context of an appearance of its parent element type. By default, attributes are assigned the `#IMPLIED` modifier, indicating that the attribute *optionally* appears. This can be overriden by specifying the `#REQUIRED` modifier, as is done for the `id` attribute type in the example DTD. Finally, the `#FIXED` modifier indicates that every occurrence of the attribute type takes on the specified value.

We now provide a formal definition of DTDs that will be utilized in Chapter 4.

Definition 2.4.6 (Document Type Definition (DTD))**.** *A* Document Type Definition (DTD) *is a 4-tuple $\mathcal{D} = \langle \Sigma_e, \Sigma_a, l_r, \mathcal{R} \rangle$, where Σ_e and Σ_a are finite label alphabets for elements and attributes, respectively, $l_r \in \Sigma_e$ is the distinguished root element label, and $\mathcal{R}$ is a function assigning to each label $e \in \Sigma_e$ a regular expression ρ_e defined by the following EBNF grammar:*

$$\begin{aligned} \rho_e &::= str \mid \epsilon \mid expr \mid term \\ expr &::= \text{`(' } expr \text{ `,' } expr \text{ `)'} \mid \text{`(' } expr \text{ `}\vee\text{' } expr \text{ `)'} \mid term \\ term &::= b_i * \mid b_i + \mid b_i ? \mid b_i \end{aligned}$$

where str denotes a PCDATA segment, ϵ is the empty string, each b_i is a label in $\Sigma_e \cup \Sigma_a$, ',' denotes concatenation, and '$\vee$' indicates disjunction, Cardinality constraints *definable over a label b_1 include 'b_1*', 'b_1+', and 'b_1?', representing zero or more occurrences, at least one occurrence, and zero or one occurrence, respectively. Each element label e may optionally be assigned an ordered* attribute set*, consisting of entries in Σ_a. For each label $a \in \Sigma_a$, $\mathcal{R}$ assigns a set μ of* modifiers *drawn from {ID, IDREF, IDREFS, #REQUIRED, #FIXED $\langle value \rangle$}.* □

An XML database T is *valid* with respect to a DTD $\mathcal{D}$ when the following conditions are satisfied: (1) $type(r) = l_r$; (2) for each interior edge e, $type(e) \in \{\Sigma_e \cup \Sigma_a\}$, while the label for each terminal edge is drawn from the set of all strings $\mathcal{S}$; and (3) for each interior

```
R1: <!ELEMENT hospital (patient*)>
R2: <!ELEMENT patient (name,age,diagnosis,doctor,
     discharge_date?,carrier)>
R3: <!ELEMENT name (#PCDATA)>
R4: <!ELEMENT age (#PCDATA)>
R5: <!ELEMENT diagnosis (#PCDATA)>
R6: <!ELEMENT doctor (#PCDATA)>
R7: <!ELEMENT discharge_date (#PCDATA)>
R8: <!ELEMENT carrier (#PCDATA)>
```

Figure 2.3: DTD for the hospital database.

edge $e = (v_1, v_2)$ whose label is contained in Σ_e, the ordered list of edge labels exiting from v_2 satisfies the regular expression ρ_e.

Note that for any database T conforming to $\mathcal{D}$, the location path for each node is formed by a successive application of the rules in $\mathcal{D}$. Therefore, $\mathcal{D}$ defines all valid location paths that may appear in any conformant database. By $paths(\mathcal{D})$, we denote the set of all location paths specified by $\mathcal{D}$. For a specific location path p, we indicate by $derivation(p, \mathcal{D})$ the ordered list of DTD rules $r_1, \ldots, r_m$ in $\mathcal{D}$ whose successive application forms p.

Example 2.4.7. *Fig. 2.3 depicts the DTD for the hospital database shown in Fig. 2.1. It is represented under our notation as* $\Sigma_e = \{$`hospital`, `patient`, `name`, `age`, `diagnosis`, `doctor`, `discharge_date`, `carrier`$\}$, $\Sigma_a = \emptyset$, *and root element* $r =$ `hospital`. *The production for the root* `hospital` *element is given by* $\mathcal{R}($`hospital`$) = \{$`patient`$*\}$, *while the production for* `patient` *elements is* $\mathcal{R}($`patient`$) = \{$`name`, `age`, `diagnosis`, `doctor`, `discharge_date`$?$, `carrier`$\}$. *Furthermore,* $derivation(/$`hospital`$/$`patient`$/$`age`$, \mathcal{D}) = \langle$`R1`, `R2`$\rangle$. □

A DTD $\mathcal{D}$ can be modeled as a directed graph [110] in which the nodes correspond to labels in $\Sigma_e \cup \Sigma_a \cup \#$`PCDATA`, and each edge (l_1, l_2), with $derivation(l_1, \mathcal{D}) = \langle r_{\pi_1}, \ldots, r_{\pi_m} \rangle$, indicates that $derivation(l_2, \mathcal{D}) = \langle r_{\pi_1}, \ldots, r_{\pi_m}, r_{\pi_{m+1}} \rangle$ for some additional rule $r_{\pi_{m+1}} \in \mathcal{R}$. Where applicable, edge (l_1, l_2) is annotated with cardinality constraints and/or attribute modifiers present in rule $r_{\pi_{m+1}}$.

Example 2.4.8. *Fig. 2.4 depicts the DTD graph for the DTD in Fig. 2.3.* □

While DTDs have proven sufficient for many XML applications, they suffer from some shortcomings, such as the inability to define a separate content model for each location path

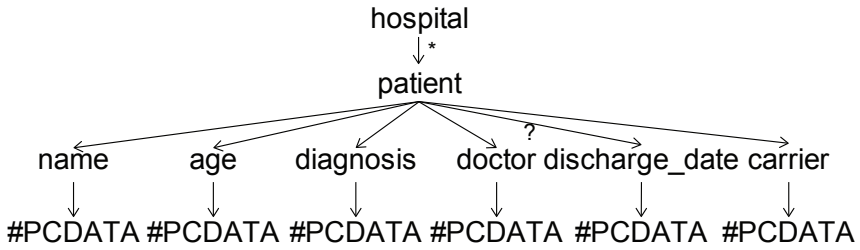


Figure 2.4: DTD graph representation for DTD of Fig. 2.3.

in a conformant XML document. Instead, one is limited to specifying content models based solely on the name of the element or attribute. This means, for example, that one could not define separate DTD rules for `/hospital/patient/name` and `/hospital/doctor/name`. In addition, content models in DTDs are coarse-grained in the sense that all character data sequences are implicitly treated as strings, preventing constraints based on data types (such as defining legal ranges for numeric data values) from being formed over element content and attribute values.

XML Schema

To address the above shortcomings of DTDs, the World Wide Web Consortium introduced the *XML Schema Definition (XSD)* language. XML Schema allows context-sensitive content models to be specified, thereby overcoming the first weakness of DTDs mentioned above. Further, it provides an extensible data type system (allowing one to indicate that, for example, the content of `/hospital/patient/name` elements are strings, while each `/hospital/patient/age` element contains an integer value). Another feature introduced with XML Schema is namespace awareness, which is of particular use for applications in which data is integrated from multiple sources; one can avoid naming collisions for elements and attributes by defining separate namespaces for each source. Another useful feature of XSD specifications is that they themselves are well-formed XML documents, allowing the same toolset to be used for processing the schema definition as for instance documents conforming to the specification. Fig. 2.5 lists an XSD equivalent for the example course DTD of Fig. 2.2.

```
<xs:schema elementFormDefault="qualified"
        xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="course">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="name" type="xs:string" />
        <xs:element name="instructor" type="xs:string" />
        <xs:element name="students">
          <xs:complexType>
            <xs:element name="student" type="StudentType"
                    minOccurs="1" maxOccurs="unbounded">
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:complexType name="StudentType">
    <xs:sequence>
      <xs:element name="a1" type="xs:integer" />
      <xs:element name="a2" type="xs:integer" />
      <xs:element name="midterm" type="xs:integer" />
      <xs:choice>
        <xs:element name="project" type="xs:integer" />
        <xs:element name="finalexam" type="xs:integer" />
      </xs:choice>
    </xs:sequence>
    <xs:attribute name="id" type="xs:integer"
            use="required" />
  </xs:complexType>
</xs:schema>
```

Figure 2.5: XML Schema Definition for the course document.

2.4.3 Query Languages for XML

Currently, the two most popular query languages for XML are XPath [12] and XQuery [17]. In XPath, *path expressions* are used to select arbitrary nodes from an XML tree; such expressions return a *set of nodes* that share a particular relationship with a specified context node. In the *unabbreviated* syntax, each path expression consists of a sequence of one or more *location steps*. Each location step takes the form `/axis::nodeTest[predicate]`, where `axis` is one of the thirteen XPath axes (`self`, `child`, `parent`, `descendant`, `ancestor`, `descendant-or-self`, `ancestor-or-self`, `preceding`,

following, preceding-sibling, following-sibling, attribute, or namespace) defining the tree relationship between selected nodes and members of the node set formed by evaluating the previous location step; nodeTest indicates a test that be carried out to filter the selected nodes, typically based on the name of the node; and predicate is an optional component allowing one to specify additional filtering conditions on the selected node set. In the case of the first location step, it is assumed that the current node set consists of a virtual document node whose only child is the document root node.

Example 2.4.9. *When evaluated on a course database conforming to the DTD of Fig. 2.2, the path expression*

$$/\texttt{child} :: \texttt{course}/\texttt{child} :: \texttt{students}/\texttt{child} :: \texttt{student}$$

first selects the document root node, course*, based on the node test specified in the first location step. Evaluation of the second location step begins at the* course *node, and begins by selecting all children of* course *as the initial node set, since the* child *axis is specified. The node test for this step is then carried out, which indicates that only nodes with name* students *are to be kept. The final step travels down the* child *axis from the* students *node, only keeping those children named* student*. The two* student *nodes form the query result.*

The path expression

$$/\texttt{child} :: \texttt{course}/\texttt{child} :: \texttt{students}/\texttt{child} :: \texttt{student}[\texttt{attribute} :: \texttt{id} = 1001]$$

contains a predicate indicating that only student *nodes with a* id *attribute value of* 1001 *should be included. Therefore, the query result only contains the left* student *subtree.* □

The unabbreviated syntax allows queries involving all XPath axes to be expressed in a consistent way. In many cases, however, queries will only refer to the child and descendant axes, in which case an *abbreviated syntax* can be used. In the abbreviated syntax, all node tests are assumed to be based on the node name, and / is used to indicate the presence of the child axis within a location step and // the descendant axis.

Example 2.4.10. *The path expressions* /course/students/student *and* /course/students/student[@id = 1001] *are abbreviated syntax versions of the queries in Ex. 2.4.9.* □

In the thesis, we will mostly use the abbreviated syntax, resorting to the unabbreviated syntax only for more complex queries.

XQuery builds on XPath, and provides additional constructs to allow more complex selection, filtering, and processing of query results. As the remainder of the thesis focuses on XPath, we do not describe XQuery further.

Core XPath

The full XPath 2.0 standard includes many sophisticated features, such as functions and data types. Gottlob et al [58] identified a subset of XPath encapsulating its basic tree navigation capabilities over node sets, removing the more complex, less frequently used features such as operations over strings and arithmetic expressions, and termed this fragent *Core XPath*. They showed that while full XPath has a worst-case polynomial running time in the combined size of queries and trees, Core XPath admits an implementation with a worst-case linear running time.

Core XPath is defined by the following EBNF grammar

expr ::=locationpath | '/' locationpath

locationpath ::=locationstep ('/' locationstep)*

locationstep ::=χ'::' t | χ'::' t '[' pred ']'

pred ::=pred 'and' pred | pred 'or' pred |

'not' '(' pred ')' | expr | '(' pred ')'

where expr is the start production, χ represents an XPath axis and t is a node test (either a specific XML tag name, or the wildcard $\star$ symbol, meaning that any node label is considered a match).

Core XPath forms the basis of the query language assumed in Chapter 5, defined as follows.

expr ::=locationpath | '/' locationpath

locationpath ::=locationstep ('/' locationstep)*

locationstep ::=χ'::' t | χ'::' t '[' pred ']'

pred ::=pred 'and' pred | pred 'or' pred |

'not' '(' pred ')' | expr | '(' pred ')' |

expr '/' χ '::' 'text()' '=' str

Above, χ is an XPath *axis* (self, parent, child, ancestor, descendant, ancestor-or-self, descendant-or-self, following, preceding, following-sibling, preceding-sibling, or attribute), t represents a *node test* (an XML tag name, the wildcard symbol '*', or `text()`), and str represents a string constant. This fragment, which we denote as $\texttt{CXP}^{[=]}$, corresponds to Core XPath augmented with string comparisons within predicates.

We also consider updates composed of one or more of the following primitive operations on individual tree nodes, referenced by ordinal values x and y: (1) **Append(**x, y**)**, which inserts a node y as the right-most child of node x; (2) **InsertBefore(**x, y**)**, which inserts node y as the left sibling of node x; and (3) **Delete(**x**)**, which results in the deletion of node x and its descendants. This small set of operations is "sound" and "complete", in the sense that any XML tree can be constructed by their combination [9].

2.4.4 Access Control Models for XML

Several XML-specific access control models have been proposed in recent years. At a high level, they can be divided into *interactive, view-based approaches*, which employ a centralized module to control access to XML documents (e.g., [14, 39, 75, 81]), and *secure publishing approaches* (e.g., [13, 36, 88]), which use cryptographic techniques to enforce a designated access control policy over a single, published document.

To serve as a demonstration of the interactive approach, we briefly describe the model of Damiani et al [39]. In this model, access control policies can be specified at either the

element or attribute level; the objects of each rule are specified using XPath. Propagation is supported, that is, element-level rules can be recursively applied to the subtree rooted at that element. Permissions can be *positive* (specifying that the rule subject(s) are allowed access to the rule object(s)) or *negative* (denying the rule subject(s) access to the specified object(s)), and can be specified at either the document or DTD level.

Based upon the specified policy rules, a *labelling algorithm* assigns security annotations to each tree node. During a second pass over the tree, those nodes which are not accessible to the current user are pruned. If pruning results in an invalid document, a "loosening" operation is performed on the DTD to reestablish validity. The fact that several passes have to be made over the document to generate several customized views of the original tree (containing only the nodes that are accessible to each user) makes this approach ill-suited to online use, and additionally does not scale well to handle very large documents.

An illustrative example of the secure publishing approach is the scheme proposed by Miklau and Suciu [88], in which access control policy rules are specified in a high-level language extending XQuery. These rules are then converted to an equivalent *tree protection* over the target XML document, in which each tree node may be guarded by a positive boolean formula defined over a set of cryptographic keys; a user may only access a node if she possesses a combination of keys satisfying the guarding formula. These boolean formulas are represented by adding additional metadata nodes to the original document tree. Recently, Miklau and Suciu's tree protection scheme was shown to be provably secure, in the sense that an adversary provided with a partially-encrypted document essentially cannot do better than a random guess as to the true values of encrypted nodes [1]. A weakness of this approach is that a complex policy will tend to require several metadata nodes to be added to the document tree, increasing the document size dramatically.

Discussion. We first note that in all these approaches, the focus is on specifying an access control policy over an *existing* XML document, and no attention has been paid to the issue of propagating a pre-existing access control policy defined over relational data to its XML representation.

When viewed in comparison with the interactive approach, secure publishing XML ac-

cess control methods can claim several advantages. Scalability is obviously improved, as users are able to obtain and query a local copy of a document in lieu of placing the burden of answering such queries on a server. A somewhat related advantage is increased document availability: in interactive models, all documents cease to be available if the server becomes unable to serve requests (e.g. due to a hardware or software failure, or because the server has suffered a denial-of-service attack). Conversely, since users of an application employing a secure publishing approach are free to redistribute documents to other users, document availability is likely to increase over time as a given document propagates across the network.

In reference to the disadvantages of secure publishing, one can quickly notice that this approach is unfriendly with respect to document changes; if even a single node within the document tree is inserted, deleted, or updated, several expensive re-encryption operations may be required. In addition, complicated access control policies containing several defined user roles will obviously require several key sets to be generated and distributed.

In Chapters 4 and 5, we assume an interactive access control model is in place over federated XML data, defined formally as follows.

Definition 2.4.11 (XML Access Control Policy)**.** *An* access control policy (ACP) $A = \{pr_1, \ldots, pr_p\}$ *consists of a set of* policy rules*. Each policy rule is a 4-tuple* $pr_i = \langle R_i, e_i, a_i, p_i \rangle$*, where* $R_i \subseteq F$ *is the set of federated IDs to whom the rule applies (*subjects*),* e_i *is a* $\texttt{CXP}^{[=]}$ *query identifying the sub-tree(s) the specified permission is applied to (*objects*),* a_i *is an* action *(e.g., "read" or "write"), and* $p_i \in \{$"allow","deny"$\}$ *is the associated* permission*. If* e_i *contains one or more conditions, we often refer to* pr_i *as a conditional policy rule.* □

This model is consistent with the majority of access control models in the literature, which similarly allow access permissions to be expressed over subtrees. Note also that this model is universal in the sense that one can express permissions ranging from individual nodes (the finest possible granularity) up to document-level permissions (expressed over the root node's subtree).

Semantics. In our access control model, we assume a "closed world" semantics in which access to nodes is denied unless there exists a policy rule specifying otherwise. Additionally, we assume that permissions are propagated from a parent node to its children. In case of rule conflict (where one policy rule grants access to a node while another denies access), we assume that the most specific rule (i.e., the rule defined for the closer ancestor, potentially the node itself) takes precedence, where ties are broken in favour of the least permissive rule. For those situations in which a user has been allocated multiple federated IDs assigning conflicting permissions to a node, we assume that the most permissive such policy takes precedence.

Node Access Levels. A useful way to analyze an ACP is to consider, for each node in the database, which subset of roles is granted permission to perform a particular action on that node. We define the *access level* of node v under action a, denoted $\mathcal{L}(v, a)$, to be the subset of federated IDs who are permitted to perform a on v. By comparing the access levels of two nodes v_1 and v_2, we can determine whether v_1 is more (or less) highly classified than v_2, or is of equal classification to v_2. More formally, for a given action a, the set of node access levels $\mathcal{L}(\cdot, a)$ forms a partial ordering in which for nodes v_1 and v_2, we say that $\mathcal{L}(v_2, a)$ *is dominated by* $\mathcal{L}(v_1, a)$, written $\mathcal{L}(v_2, a) \leq \mathcal{L}(v_1, a)$, if $\mathcal{L}(v_1, a) \subseteq \mathcal{L}(v_2, a)$. Note that while a partial order is guaranteed, a total order is not, as the access levels of two nodes v_1 and v_2 may be incomparable, i.e., $\mathcal{L}(v_1, a) \not\subseteq \mathcal{L}(v_2, a)$ and $\mathcal{L}(v_2, a) \not\subseteq \mathcal{L}(v_1, a)$.

Views. The application of an access control policy A to an XML tree T results in the generation of a set of *views* $\mathcal{V} = \{V_1, \ldots, V_{|F|}\}$ in which view V_i contains only the subset of database nodes which are accessible to federated ID $f_i \in F$ under A. For a given location path p, action a, and federated ID f_i, the application of the access control policy can result in f_i having *full access* to p (i.e., all instances of p are contained in V_i), *partial access* to p (i.e., only a proper subset of p instances are contained in V_i), or *no access* to p (i.e., no instances of p are contained in V_i) within the context of a.

One can also view the application of an access control policy as applying to a DTD $\mathcal{D}$ by materializing the set $paths(\mathcal{D})$ and applying all rules in A which are applicable to each individual location path. Consistent with this interpretation, we denote by $\oplus_a(f_i, A, \mathcal{D})$, $\ominus_a(f_i, A, \mathcal{D})$, and $\odot_a(f_i, A, \mathcal{D})$ the sets of location paths in $\mathcal{D}$ which are fully-accessible,

partially-accessible, and non-accessible to federated ID f_i within the context of action a under AC. More formally,

$$\begin{aligned}
\odot_a(f_i, A, \mathcal{D}) =& \{t | t \in paths(\mathcal{D}) \wedge (\forall n \in class(t) \\
& (f_i \notin \mathcal{L}(n, read)))\} \\
\ominus_a(f_i, A, \mathcal{D}) =& \{t | t \in paths(\mathcal{D}) \wedge (\exists n_1, n_2 \in class(t) \\
& (f_i \in \mathcal{L}(n_1, read) \wedge f_i \notin \mathcal{L}(n_2, read)))\} \\
\oplus_a(f_i, A, \mathcal{D}) =& \{t | t \in paths(\mathcal{D}) \wedge (\forall n \in class(t) \\
& (f_i \in \mathcal{L}(n, read)))\}.
\end{aligned}$$

To improve readability, we frequently omit a, A and $\mathcal{D}$ when their values are made clear by the context. These sets can be efficiently populated during the schema-level analysis phase, as we describe later in Sec. 4.7.

Example 2.4.12. *Consider the following access control policy, applicable to the hospital database of Fig. 2.1, where the set of federated IDs is* $F = \{AIns, BCross, WShield, Alice, Bob, Carol\}$.

```
PR1: <{AIns, BCross, WShield, Alice, Bob, Carol}, /hospital,
     read, allow>
PR2: <AIns, /hospital/patient[carrier neq "ACME Insurance"]/
     name, read, deny>
PR3: <BCross, /hospital/patient[carrier neq "Black Cross"]/
     name, read, deny>
PR4: <WShield, /hospital/patient[carrier neq "White Shield"]/
     name, read, deny>
PR5: <Alice, /hospital/patient[name neq "Alice"], read, deny>
PR6: <Bob, /hospital/patient[name neq "Bob"], read, deny>
PR7: <Carol, /hospital/patient[name neq "Carol"], read, deny>
```

Policy rule `PR1` *specifies that all federated IDs have default read access to the entire database. This is then refined by subsequent rules: rules* `PR2` *through* `PR4` *limit the employ-*

ees of each insurance company to only have access to the patient names of their respective clients, while rules PR5 *through* PR7 *have the effect of restricting each patient to accessing only their individual records. Applying this access control policy to the hospital database in Fig. 2.1 results in access levels* $\mathcal{L}(o2, read) = \{$`Alice, AIns, BCross, WShield`$\}$ *and* $\mathcal{L}(o3, read) = \{$`BCross, Alice`$\}$ *and hence we can say that* $\mathcal{L}(o2, read) \leq \mathcal{L}(o3, read)$*. An employee of ACME Insurance would inherit the permissions granted to the* AIns *ID, consisting of*

$\odot_{read}$`(AIns) =` $\emptyset$

$\ominus_{read}$`(AIns) =` $\{$`/hospital/patient/name`$\}$

$\oplus_{read}$`(AIns) =` $\{$`/hospital, /hospital/patient,`
`/hospital/patient/age, /hospital/patient/diagnosis,`
`/hospital/patient/doctor, /hospital/patient/carrier,`
`/hospital/patient/discharge_date`$\}$ □

Visualizing the domination relation as a *Hasse diagram* [40] is one means of interpreting the result of applying an access control policy to an XML tree, within the context of a specific action a. Each vertex in the diagram represents a distinct access level, with the vertices ordered in such a manner that vertex n_1 is located below n_2 iff for the corresponding access levels v_1 and v_2, $\mathcal{L}(v_1, a) \leq (v_2, a)$. An upward edge is drawn from n_1 to n_2 iff $\mathcal{L}(v_1, a) \leq \mathcal{L}(v_2, a)$, and further, there exists no other access level v_3 satisfying $\mathcal{L}(v_1, a) \leq \mathcal{L}(v_3, a) \leq (v_2, a)$. Fig. 2.6 depicts a Hasse diagram for the hospital database when the access control policy of Ex. 2.4.12 has been applied.

Definition 2.4.13 (Montonicity of XML Access Control Policies)**.** *An access control policy* A *is* monotonic *over an XML database* $T = (V, E, r, L)$ *with respect to action* a *iff* $\forall v \in V$*,* $\mathcal{L}(parent(v), a) \leq \mathcal{L}(v, a)$. □

By definition, an access control policy is monotonic if and only if there exists no circumstance in which the access permissions granted to a child node are less restrictive than those bestowed upon its parent, or equivalently, each derived view in $\mathcal{V}$ consists of a single, connected tree. Whenever a monotonic ACP is applied to an XML tree, the corresponding

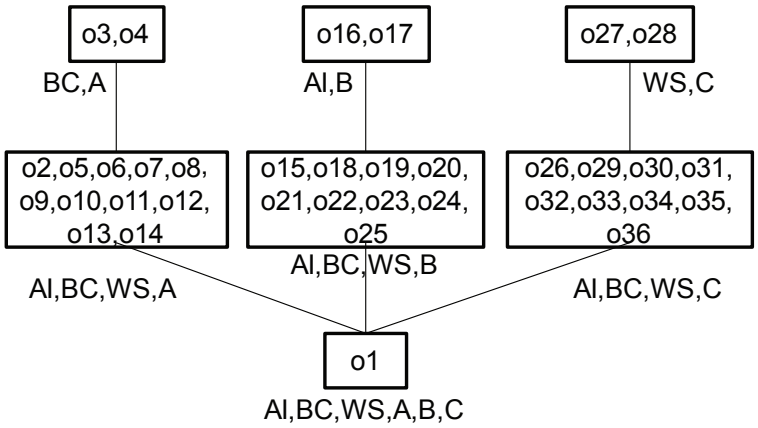


Figure 2.6: Hasse diagram for hospital database of Fig. 2.1.

Hasse diagram – for each child node c with parent node p – satisfies either of the following conditions: (1) c and p are contained within the same vertex (access level); or (2) c is contained within a vertex for which an edge connects to the vertex containing p.

One can verify that the access control policy of Ex. 2.4.12 applied to the example hospital database is monotonic by observing that Fig. 2.6 satisfies one of the above conditions for each parent-child relationship. We note that the majority of XML access control models [15, 13, 20, 31, 36, 39, 50, 81, 82, 88, 93, 117, 119] in the literature require that their policies be monotonic. In contrast, the Author-X [14] and Kudo-Hada [75] models do support the specification of non-monotonic access control policies. Henceforth, we assume that all legal access control policies under our model possess the monotonic property. In later chapters, we will utilize this assumption to derive more efficient algorithms for detecting and removing disclosure risks from XML databases.

XML access control expression languages

While XML access control models define the *semantics* of protections specified over an XML document tree, such descriptions, in their original form, tend to be somewhat abstract and lack a uniform syntax. To promote interoperation and integration with other applications, and also to facilitate easy deployment and processing of policies, many models also enable their policies to be exported to an XML access control *expression language*; such languages allow access control policies over XML documents to be specified in a standardized syntax that is understood by many software tools. Typically, these expres-

sion languages are themselves XML vocabularies (e.g., [46, 91]). For example, the secure publishing approach of Miklau and Suciu takes an input tree protection and generates a partially-encrypted XML document, where encrypted subtrees are expressed using the W3C recommendation for XML encryption syntax [46].

The *eXtensible Access Control Markup Language (XACML)* [91] is an OASIS-endorsed standard consisting of both an XML-based declarative policy specification language as well as a processing model that specifies how such policies are interpreted and enforced. Among other benefits, XACML is well-suited for applications in which multiple access control policies must be integrated, and conflicts between such policies must be resolved. In particular, through the specification of *rule combining algorithms*, an administrator may indicate conflict resolution protocols stating that rules allowing access take precedence over those denying access to the same resource, or vice-versa.

2.5 Information Theory

In this section, we provide a review of discrete probability mass functions and various information theoretic measures referred to in subsequent parts of the thesis.

First, we assume that X denotes a *discrete random variable*, and $\mathbf{P}$ denotes its *probability mass function*; i.e., the probability that X takes on value x is given by $\mathbf{P}[x]$ and $\sum_x \mathbf{P}[x] = 1$ as x runs through the set of all possible values for X.

Information theory provides several measures relating to the amount of *information* (or equivalently, the level of *unpredictability*) relayed by an event expressed as a random variable. In Chapter 5, we will refer to two such measures, *entropy* and *relative entropy*. These measures are defined as follows.

Definition 2.5.1 (Entropy [111])**.** *The* entropy *of a discrete random variable X with probability mass function $\mathbf{P}$ is defined by*

$$H(X) = -\sum_x \mathbf{P}[x] \cdot \log_2 \mathbf{P}[x]\,. \tag{2.1}$$

It always holds that $0 \leq H(X) \leq \log_2 |dom(X)|$, where $|dom(X)|$ indicates the size of the domain of X (the number of distinct values that X may take). $H(X) = 0$ if and

only if there is only one possible value for X (i.e., there is no uncertainty), while $H(X) = \log_2 |dom(X)|$ when the probability mass function for X is the uniform distribution (i.e., all possible values for X are equally likely). □

Entropy is measured in bits, and indicates the average uncertainty in the random variable X. Another way of interpreting entropy is as the number of bits needed to describe X. If one views each bit as a "yes/no" question, entropy designates the average number of answers to such questions one needs to collect in order to identify the true value of the random variable.

In Chapter 5, we will also have occasion to measure the "distance" between two probability mass functions $\mathbf{P}$ and $\mathbf{Q}$ defined over the same random variable. The *relative entropy* measure from information theory provides one method for doing so.

Definition 2.5.2 (Relative Entropy [111])**.** *The* relative entropy *of two probability mass functions $\mathbf{P}$ and $\mathbf{Q}$ defined over the same discrete random variable X is defined as*

$$KL(\mathbf{P}||\mathbf{Q}) = \sum_x \mathbf{P}[x] \cdot \log_2 \frac{\mathbf{P}[x]}{\mathbf{Q}[x]}. \tag{2.2}$$

$KL(\mathbf{P}||\mathbf{Q}) \geq 0$, with equality when $\mathbf{P}$ and $\mathbf{Q}$ are the same distribution. □

Relative entropy is also measured in bits. Intuitively, smaller values indicate that $\mathbf{P}$ and $\mathbf{Q}$ are "closer", while larger values correspond to greater distances between the two distributions. Another way of interpreting relative entropy sees it as the average number of extra bits of error introduced when the "false" distribution $\mathbf{Q}$ is assumed to describe the random variable X instead of the "true" distribution $\mathbf{P}$ (or equivalently, as the average extra number of answers to "yes/no" questions one needs in order to learn the value of X).

A closely related quantity is the *cross entropy* of a random variable X within the context of probability mass functions $\mathbf{P}$ and $\mathbf{Q}$, computed as the sum of $H(X)$ and $KL(\mathbf{P}||\mathbf{Q})$. It serves as an indication of the overall difficulty of learning X's value when the "false" probability mass function $\mathbf{Q}$ is used, by considering both the inherent uncertainty of X, measured by its entropy, and the extra effort needed to correct errors introduced by assuming $\mathbf{Q}$ is the true probability mass function describing X.

Chapter 3

Access Control Policy Translation in Heterogeneous Data Federations

In this chapter, we bridge the gap between relational and XML ACPs. We develop a framework in which existing ACPs over source relational databases (which are prevalent in critical database applications) can be automatically and faithfully re-formulated over XML representations of the corresponding data. Our goal is to support a level of expressiveness in publishing functions that ensures compatibility with all of the proposed publishing frameworks to date.

3.1 Introduction

Two major challenges related to ensuring that all access control rules and privileges are correctly enforced arise when exchanging data within a federation. First, because individual databases in the federation are maintained independently of each other, access control policies (ACPs) are defined in terms of *local identities* (or local classes of users) which are valid within the system where the data resides. To overcome this limitation, a centralized user authentication system is often employed to translate local identities into *federated identities* shared across systems, thus allowing the access privileges and restrictions to be expressed within the federation as a whole. Second, the process of *translating* the ACPs in relational systems (which are still prevalent) is poorly supported by current systems and tools, despite the existence of several proposed access control models and mechanisms for XML. Currently, security administrators must *manually* convert existing ACPs into the formulation language used by a designated XML-specific access control model and *verify* that such a

ssn	name	age
123456789	Carol	31
197453163	Doug	45

Figure 3.1: `Patient` relation.

translation is indeed correct.

The manual translation of relational ACPs is complex and error-prone, especially because (1) real-world relational ACPs, especially those pertaining to large enterprises [103], may consist of hundreds of rules defined over a similarly large number of database objects (i.e., tables, columns, and tuples); (2) the hierarchical and semi-structured nature of XML creates additional complexity, as one has to take into account factors such as the propagation of access permissions from parent nodes to child nodes; and (3) each source database may define an individual access control policy pertaining to different sets of users, creating the potential for inter-policy conflicts. An additional consideration is that the XML publishing function is typically *fixed*; that is, the goal is to translate the relational ACP for a given XML publishing function. This reflects the reality that often the publishing functions express *contracts* between sources that are not easy to change. We refer to the resulting augmentation of a publishing function with access control specifications as a *secure publishing function* Π'.

3.2 Preliminaries

To illustrate the discussion, we refer to the relation depicted in Fig. 3.1, containing information about patients of a medical clinic.

3.2.1 Publishing Relational Databases as XML

Consistent with Def. 2.4.1, we view an XML document as an ordered, labeled tree. Fig. 3.2 depicts one possible manner of representing the database of Fig. 3.1 as an XML tree. A unique identifier for each tree node is indicated as a superscript of the node's label (in this figure, each node is also associated with an *access bitstring*, depicted within a rectangle; the purpose of these bitstrings is explained in Sec. 3.3).

A *publishing function* Π specifies how an XML document is created from the contents

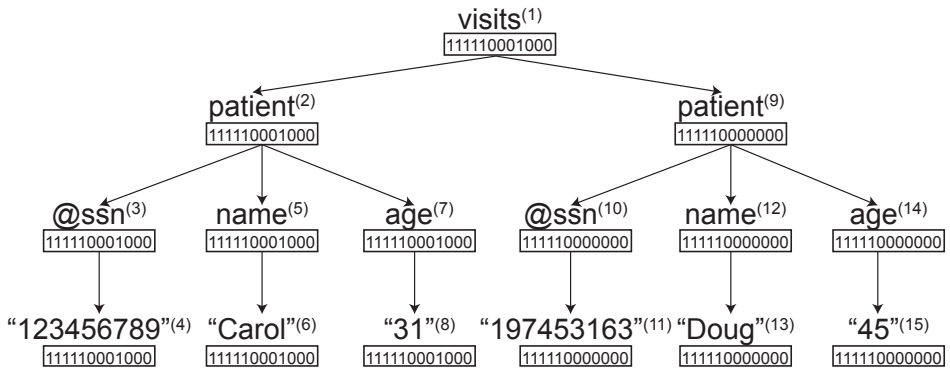


Figure 3.2: XML representation of the relation of Fig. 3.1.

of a relational database D conforming to schema S. In our model, we represent publishing functions using the formalism of publishing transducers [49]. This model provides a natural basis for our work, as it has proven to be capable of expressing all major XML publishing languages that have been introduced both by industry [97, 99, 109] and academia [11, 54, 8].

Definition 3.2.1 (Publishing Transducer)**.** *A publishing transducer is given by* $\Pi = (Q, \Sigma, q_0, \delta)$ *where* Q *is a finite set of states,* Σ *is a finite tag alphabet,* $q_0 \in Q$ *is the designated start state associated with the root tag* $r \in \Sigma$*, and* δ *is a finite set of transduction rules.* □

By convention, we distinguish attribute labels in Σ by prepending the label value with '@' (e.g., "@ssn" denotes the label value "ssn" for an attribute node).

Transduction rules have the form

$$(q, a) \rightarrow (q_1, a_1, \phi_1(\bar{x}_1; \bar{y}_1)), \ldots, (q_k, a_k, \phi_k(\bar{x}_k; \bar{y}_k))$$

where $a_1, \ldots, a_k$ are distinct tags in $\Sigma \cup text$, $(q_i, a_i) \in Q \times \Sigma$ for $i \in [1, k]$, and each ϕ_i is a relational query. We refer to each $(q_i, a_i, \phi_i(\bar{x}_i; \bar{y}_i))$ triple as a *clause*. Furthermore, the RHS of a transduction rule consists of an ordered list of clauses.

Publishing transducers are deterministic [1]: for each $(q, a) \in Q \times \Sigma$, there is a unique transduction rule. In the case of the start state q_0, only a single rule (q_0, r) is defined. Additionally, transduction rules may not contain q_0 or r in their right-hand sides, and the

[1]More precisely, for an input containing a fixed ordering of relational tuples, a given transducer produces an identical XML tree on each run. If relational tuple ordering is not consistent, then transducer determinism holds up to a consistent ordering of sibling XML subtrees.

right-hand side for rules of the form $(q, text)$ must be empty, where $text$ is a reserved symbol in Σ used to indicate a text segment (PCDATA). By a *recursive transducer*, we refer to one in which there exists a recursive reference within the RHS of a transduction rule r_2 to a rule r_1 that (either directly, or transitively) refers to r_2 within its RHS.

Essentially, a publishing transducer Π operates as a finite state machine which, for an input relational database D, generates an XML document tree in a "top-down" manner, using element tags drawn from Σ. Each node u with label $a \in \Sigma$ in the tree is associated with a *register* $Reg_a(u)$, which is used to store a relation of fixed arity. The transduction rule for the current machine state and node label is applied; the query $\phi_i(\bar{x}_i, \bar{y}_i)$ is executed on D and/or $Reg_a(u)$, and the result is used to produce the child nodes of the current node u. Here, $\bar{x}_i$ designates a set of "group-by" attributes, i.e., the result of ϕ_i is partitioned according to these attributes, with each set in the resulting partition contributing a new child node w of u whose register contains all tuples in that partition set. Conversely, all tuples with identical values for all $\bar{y}_i$ attributes create a single child node w, and are grouped together within the register associated with w.

The transduction process stops at a leaf node u with label a whenever one of the following conditions is met: (1) the right-hand sides of all (q, a) transduction rules applicable to u are empty; (2) the query $\phi_i(\bar{x}_i; \bar{y}_i)$ for each $i \in [1, \ldots, k]$ returns empty when evaluated on $Reg_a(u)$ and D; or (3) there is a node v on the path from the root to u such that expanding u will not add any new information to the tree (i.e., the state q, label a, and the content of $Reg_a(v)$ of v are all repeated by u). Condition (3) ensures that the application of recursive transduction rules will eventually terminate. We call each transduction rule that satisfies one of these conditions a *terminating transduction rule*.

We illustrate the operation of a publishing transducer with an example.

Example 3.2.2. *A publishing transducer that constructs the XML document structure depicted by Fig. 3.2 from an instance of the medical clinic relational schema of Fig. 3.1 is given by* $\Pi = (Q_1, \Sigma_1, q_0, \delta_1)$*, where* $Q_1 = \{q_0, q_1\}$*,* $\Sigma_1 = \{@ssn, age, name, patient, visits\}$*, and* δ_1 *consists of the set of transduction rules given in Fig. 3.3. Note that some queries, such as* ϕ_1*, select all tuples from a relation or register, while others carry out a selection and/or projection operation on the contents of a relation or register (for example,* ϕ_2

$$
\begin{aligned}
r_1 : (q_0, visits) &\rightarrow (q_1, patient, \phi_1(s, n, a; \emptyset)), \text{where} \\
\phi_1(s, n, a) &= \texttt{Patient}(s, n, a) \\
r_2 : (q_1, patient) &\rightarrow (q_1, @ssn, \phi_2(s; \emptyset)), (q_1, name, \phi_3(n; \emptyset)), \\
&\quad (q_1, age, \phi_4(a; \emptyset)), \text{where} \\
\phi_2(s) &= \exists n, a\, Reg_{patient}(s, n, a) \\
\phi_3(n) &= \exists s, a\, Reg_{patient}(s, n, a) \\
\phi_4(a) &= \exists s, n\, Reg_{patient}(s, n, a) \\
r_3 : (q_1, @ssn) &\rightarrow (q_1, text, \phi_5(s; \emptyset)), \text{where} \\
\phi_5(s) &= Reg_{@ssn}(s) \\
r_4 : (q_1, name) &\rightarrow (q_1, text, \phi_6(n; \emptyset)), \text{where} \\
\phi_6(n) &= Reg_{name}(n) \\
r_5 : (q_1, age) &\rightarrow (q_1, text, \phi_7(a; \emptyset)), \text{where} \\
\phi_7(a) &= Reg_{age}(a) \\
r_6 : (q_1, text) &\rightarrow . \qquad / * \texttt{empty string} * /
\end{aligned}
$$

Figure 3.3: Transduction rules for the publishing transducer of Ex. 3.2.2.

projects only the `ssn` *attribute of each tuple within the register associated with a* `patient` *node; this is indicated by binding the remaining two relational attributes –* `name` *and* `age` *– within the register to an existential quantifier).* □

We note that each query appearing in Fig. 3.3 defines $\bar{y}$ as the empty set (and $\bar{x}$ is non-empty), resulting in each node in the generated XML tree being associated with a tuple register. We now modify Ex. 3.2.2 slightly to illustrate the use of a relation register.

Example 3.2.3. *In place of the XML tree of Fig. 3.2, one could choose an alternative representation in which the root* `visits` *node has a single* `patients` *child node, making each* `patient` *node a child of the latter. Fig. 3.4 shows the transduction rule set for a publishing transducer capable of generating such a representation. In particular, the start rule* r_1 *has been updated such that a single child node labelled* `patients` *is created in response to issuing query* ϕ_1 *against the* `Patient` *relation. Since* ϕ_1 *specifies that* $\bar{x} = \emptyset$*, and* $\bar{y}$ *contains all attributes in* `Patient`*, the result is that the constructed* `patients` *node is associated with a relation register whose contents are a copy of the original* `Patient` *relation. Rule* r_2 *then issues a query on this relation register, and constructs a separate* `patient` *child node of* `patients` *for each tuple in the register, as desired. The remain-*

$$
\begin{aligned}
r_1 : (q_0, visits) &\rightarrow (q_1, patients, \phi_1(\emptyset; s, n, a)), \text{where} \\
\phi_1(s, n, a) &= \texttt{Patient}(s, n, a) \\
r_2 : (q_1, patients) &\rightarrow (q_1, patient, \phi_2(s, n, a; \emptyset)), \text{where} \\
\phi_2(s, n, a) &= Reg_{patients}(s, n, a) \\
r_3 : (q_1, patient) &\rightarrow (q_1, @ssn, \phi_3(s; \emptyset)), (q_1, name, \phi_4(n; \emptyset)), \\
&\quad (q_1, age, \phi_5(a; \emptyset)), \text{where} \\
\phi_3(s) &= \exists n, a\, Reg_{patient}(s, n, a) \\
\phi_4(n) &= \exists s, a\, Reg_{patient}(s, n, a) \\
\phi_5(a) &= \exists s, n\, Reg_{patient}(s, n, a) \\
r_4 : (q_1, @ssn) &\rightarrow (q_1, text, \phi_6(s; \emptyset)), \text{where} \\
\phi_6(s) &= Reg_{@ssn}(s) \\
r_5 : (q_1, name) &\rightarrow (q_1, text, \phi_7(n; \emptyset)), \text{where} \\
\phi_7(n) &= Reg_{name}(n) \\
r_6 : (q_1, age) &\rightarrow (q_1, text, \phi_8(a; \emptyset)), \text{where} \\
\phi_8(a) &= Reg_{age}(a) \\
r_7 : (q_1, text) &\rightarrow . \qquad / * \texttt{empty string} * /
\end{aligned}
$$

Figure 3.4: Transduction rules for the publishing transducer of Ex. 3.2.3.

ing rules correspond to those in Ex. 3.2.2. □

3.2.2 Access Control for Relational Databases

We recall the definition of discretionary access control policies for relational databases given as Def. 2.3.1, and illustrate its usage in the following example.

Example 3.2.4. *Consider the following SQL ACP over the schema of Fig. 3.1, with user set* $U = \{\texttt{UserA}, \texttt{UserB}, \texttt{UserC}, \texttt{DBA}\}$*:*

1. `GRANT select, update ON Patient TO UserA WITH GRANT OPTION`
2. `GRANT select ON Patient TO UserB`
3. `GRANT select ON CarolView TO UserC`.

`CarolView` *is a view over the* `Patient` *relation defined using the SQL command:* `CREATE VIEW CarolView AS SELECT * FROM PATIENT WHERE name='Carol'`. *Under our notation, this ACP would be expressed using the following set of 5-tuples:*

1. $\langle \texttt{UserA}, \phi(s, n, a) = \texttt{Patient}(s, n, a), \{select, update\}, true, DBA \rangle$

2. $\langle$`UserB`$, \phi(s,n,a) =$ `Patient`$(s,n,a), \{select\}, false, DBA\rangle$

3. $\langle$`UserC`$, \phi(s,n,a) =$ `Patient`$(s,n,a) \wedge n =$ '$Carol', \{select\}, false, DBA\rangle$. □

One may classify the expressiveness of an access control language according to the types of q queries that are allowed to appear within policy rules. For example, each SQL `GRANT` statement selects one or more columns within a table, and may therefore be specified using a *conjunctive query* (CQ) (recall from Sec. 2.2.3 that such queries are formed from the fragment of first-order logic consisting of conjunction and existential quantification over atomic formulae). View definition queries in SQL can be much more complex, as they may include features such as negation and union operations. They essentially correspond to the class of first-order (FO) queries (cf. Sec. 2.2.2)[2].

3.2.3 Federated Identities

Each source database defines its access control policies in terms of a closed set of *local identities* U_i; typically, such identities are individual users or roles. Since such sets often differ from one source to another, a mapping from local identities to a pool F of *federated identities* that are known across the federation is needed. For each data source DS_i, we assume an *identity mapping function* $I_i : U_i \rightarrow 2^F$ that assigns to each local identity $u \in U_i$ one or more federated identities in F. Furthermore, for each federated identity $f \in F$, $accessible(f,p) = \{o \,|\, \exists u \in U_i : f \in I_i(u) \wedge o \in accessible(u,p)\}$, and $inaccessible(f,p) = D \backslash accessible(f,p)$; that is, a database object (cell) that is accessible to one local identity that has been mapped to a specific federated identity f should be accessible to *all* members of f.

Example 3.2.5. *We provide an example of an ID mapping function based on the* `Patient` *relation of Fig. 3.1. We assume that there are four local IDs defined at the data source hosting the* `Patient` *relation, named* `Alice`*,* `Bob`*,* `Carol`*, and* `Doug`*. The latter two represent individual patients, while Alice and Bob are the doctors treating Carol and Doug, respectively. As such, Alice should maintain access to Carol's patient record after it has been published to the federation, and similarly, Bob should keep access to Doug's record.*

[2]More precisely, full SQL:1999 is equivalent to first-order queries plus aggregation [80].

Further, Alice and Bob presumably have access to additional federated data relevant to their doctor duties, and Carol and Doug must receive access to the federated version of their respective patient records. One possible ID mapping function satisfying these requirements would be $I(\texttt{Alice}) = \{\texttt{Doctor}, \texttt{PatientCarol}\}$, $I(\texttt{Bob}) = \{\texttt{Doctor}, \texttt{PatientDoug}\}$, $I(\texttt{Carol}) = \{\texttt{PatientCarol}\}$, $I(\texttt{Doug}) = \{\texttt{PatientDoug}\}$. □

It is crucial to avoid violating the principle of least privilege when designing the identity mapping functions. Specifically, while one wants to ensure that such a mapping allows each local user to keep access to the XML publication of all relational database objects made accessible to them under the applicable relational access control policy, it is also crucial that they not be given *too many* permissions by the mapping. When mapping functions are defined manually, the onus is on security administrators to provide a sufficiently large pool of federated identities and to define each I_i properly (in particular, I_i should only assign local IDs u_1 and u_2 to the same federated identity in cases where $accessible(u_1, p) = accessible(u_2, p)$ for all applicable permissions p).

Another potential source of concern is that each data administrator only defines the mapping function pertinent to their local data source, with no control over how such mappings are defined by other data sources. This presents a challenge to each data source's authorization autonomy, as a careless or malicious administrator from another data source can compromise the intended access privileges defined by the local data source through their mappings. To illustrate, assume we extend Ex. 3.2.5 to consider a second data source with an administrator holding the local ID `Admin`. By defining his mapping function as $I(\texttt{Admin}) = \{\texttt{PatientCarol}\}$, the administrator would be able to access the federated representation of Carol's patient record. One solution for handling such situations would be to allow security administrators at each data source to specify conditions on how mappings are constructed, and to correct an existing set of ID mapping functions in order to satisfy all such conditions. In essence, this mechanism would serve to extend the "publishing contract" each data source makes with the federation, through the definition of a secure publishing function with additional constraints that other local data administrators must obey as a condition for that source's participation within the federation. We leave this as future work.

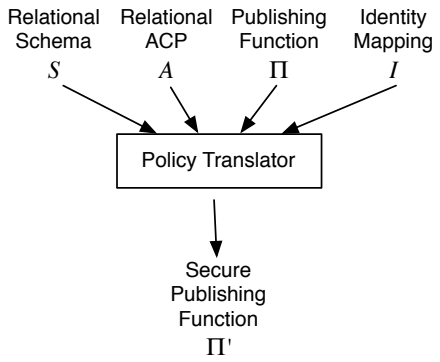


Figure 3.5: The policy translation process.

3.3 Translating Relational Policies

In this section, we discuss issues and solutions related to translating a pre-existing access control policy defined over a relational database D to an equivalent policy defined over an XML representation of D. Our goal is to have a *generic* access control policy translation procedure that works with various XML access control enforcement mechanisms. We achieve this in two steps. We start by *annotating* each node of a given XML publishing function Π[3] with *access bitstrings* specifying necessary access control restrictions of the objects that are consumed by those nodes; we call this a *secure publishing function* and denote it by Π' throughout the paper. In the second step, specific XML access control *mechanisms* can be derived from Π' to correctly enforce the original relational access control policy over the mapped data. While different mechanisms have been discussed in the literature, we focus on describing Π' in terms of the standard *eXtensible Access Control Markup Language (XACML)* [91].

Fig. 3.5 illustrates the policy translation process. The inputs consist of a relational schema S, an SQL access control policy A defined over S, a publishing function Π, and an identity mapping function I. Provided with such inputs, we wish to derive a *secure publishing function* Π' that augments Π by annotating the output XML tree nodes with additional information, allowing A to be enforced over the XML representation. These annotations are defined in terms of the federated identities, as captured by the identity mapping function

[3]Usually, the access control policy is defined *over* a specific XML representation of the data, and not the other way around.

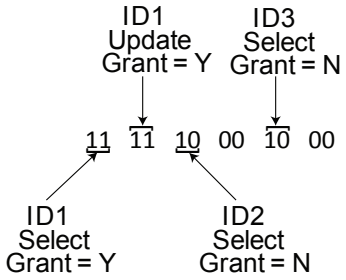


Figure 3.6: Example access bitstring.

I. We interchangeably refer to the secure publishing function as the *translated policy*.

Let o denote an individual database object (cell) returned by evaluating a ϕ query appearing within a transduction rule of Π. Each such o consists of a pair $\langle value, \phi \rangle$, where $value$ indicates the literal value retrieved from the database, and ϕ refers back to the relational query. Note that under this representation of database objects, multiple database objects (cells) sharing the same literal value remain distinguishable. This feature is crucial, since each appearance of the same literal value in the relational database may be assigned different permissions.

Definition 3.3.1 (Access Bitstring)**.** *Let $F = \langle f_1, \ldots, f_m \rangle$ be a list of federated identities and $P = \{p_1, \ldots, p_n\}$ be a set of grantable permissions. The* access bitstring *for a database object o,*

$$B_o = (b_{f_1 p_1} b_{f_1 g_1} \ldots b_{f_1 p_n} b_{f_1 g_n} \ldots b_{f_m p_n} b_{f_m g_n})$$

is a string of of $2mn$ bits which fully specifies the access control policy for over o: bit $b_{f_i p_j} = 1$ iff ID f_i holds permission p_j over o, and 0 otherwise, and bit $b_{f_i g_j} = 1$ only iff (1) bit $b_{f_i p_j} = 1$ (since a grant option for f_i is only applicable to a permission that is held by f_i) and (2) ID f_i holds the grant option for permission p_j over o. □

Example 3.3.2. *Fig. 3.6 shows an access bitstring indicating that* `ID1` *has select and update permissions, both with the grant option, while* `ID2` *and* `ID3` *each hold a select permission without grant options.* □

We now extend the definition of publishing transducer to allow access permissions over relational database objects to be preserved during the transduction process.

Definition 3.3.3 (Secure Publishing Transducer)**.** *A* secure publishing transducer $\Pi' = (Q, \Sigma, q_0, \delta')$ is a publishing transducer with Q, Σ, and q_0 defined as before, and δ' is a set of transduction rules of the form

$$(q, a, B) \rightarrow (q_1, a_1, B_1, \phi_1(\bar{x}_1; \bar{y}_1)), \ldots, (q_k, a_k, B_k, \phi_k(\bar{x}_k; \bar{y}_k))$$

where B and each B_i are access bitstrings. □

The goal is to ensure that the specified relational ACP A is preserved over the published XML document $\Pi'(D)$. It is within the access bitstrings appearing in the transduction rules of a secure publishing transducer that the semantics of A are embedded. In particular, for a clause $(q_i, a_i, \beta_i, \phi_i)$, the access bitstring β_i will be assigned to each a_i-labelled XML node produced by executing query ϕ_i [4].

Through slight abuse of the notation, we indicate by $o \in \Pi(D)$ that a representation of a given database object o is contained in the XML document $\Pi(D)$ produced by a publishing function Π on the input relational database D. By $adom(\Pi(D))$, we refer to the active domain of $\Pi(D)$, that is, the union of database objects formed by evaluating all ϕ queries appearing in the transduction rules of Π.

Definition 3.3.4 (Conditions for Relational ACP Preservation)**.** *Let* $xview : F \times P \rightarrow \mathcal{P}(\Pi(D))$ *be a function returning the subset of* $\Pi(D)$ *which is accessible to a federated ID* $f \in F$ *within the context of an individual permission* $p \in P$*. A secure publishing transducer* Π' *preserves an ACP* A *defined over a database* D *if, for all IDs* $f \in F$ *and for each permission* $p \in P$*, the following conditions are met in the published XML document* $\Pi'(D)$*:*

1. *(Sufficiency)* $\forall o \in accessible(f, p) \wedge o \in adom(\Pi(D))$*:* $o \in xview(f, p)$.
2. *(Necessity)* $\nexists o \in D : o \notin accessible(f, p) \wedge o \in xview(f, p)$. □

Example 3.3.5. *Let* I *be an identity mapping function defined as follows:* I*(*`UserA`*) =* `ID1`*,* I*(*`UserB`*) =* `ID2`*, and* I*(*`UserC`*) =* `ID3`*. Further, for convenience let us use the superscripts appearing in Fig. 3.2 to refer to the corresponding nodes in* $\Pi(D)$*. For the XML*

[4] We note that while access bitstrings are a convenient means of encoding access control policy semantics for the purposes of reasoning about the policy translation problem, alternative encoding schemes may be more space efficient within practical systems.

document listed in Fig. 3.2 and access control policy of Ex. 3.2.4, $xview(\texttt{ID3}, select) = \{1, 2, 3, 4, 5, 6, 7, 8\}$*, comprising the root node, and the leftmost* `patient` *subtree containing Carol's record. Note that for each of these nodes, the associated access bitstring (indicated within the rectangles appearing beneath each node label) has the ninth bit (corresponding to the select permission of* `ID3`*) set, while this bit remains unset in the access bitstrings of all remaining nodes.* □

The following result shows that the proposed solution is applicable to *any* relational schema, relational access control policy, identity mapping function, and publishing function provided as inputs. Additionally, it indicates that the translation need only be be performed once – at the schema level – and the obtained secure publishing function can then be applied to any instance database conforming to that schema.

Theorem 3.3.6 (Expressibility of relational ACPs)**.** *Given an arbitrary relational schema* S*, an access control policy* A *defined over a set of users* U*, an identity mapping function* I*, and a publishing transducer* Π*, a secure publishing transducer* Π' *exists that preserves* A *over* S*. Further, the derived* Π' *is applicable to any instance database* D *conforming to* S*.*

Proof. Alg. 3.3.1 demonstrates a method for converting a publishing transducer into a secure publishing transducer. We now describe how this algorithm works. A is first parsed and a hashtable is constructed from its contents (line 2). In this hashtable, each key is an attribute in S, while each key stores two linked lists containing entries of the form $(I(u), p, g, c)$ where u is a user in U, p is the applicable permission, g is *true* if the grant option has been awarded to u over the keyed attribute (and *false* otherwise), and c specifies a condition on the access granted to u over the keyed database attribute, or takes the value $\emptyset$ if no condition is present. The first linked list contains all unconditional entries, while the second contains the conditional entries (i.e. those for which c is not $\emptyset$).

Next, the transduction rules of the original publishing transducer are examined one-by-one, and transformed into augmented rules of the output secure publishing transducer (lines 3-30). Recall that each transduction rule may have multiple (q, a, ϕ) clauses on its right-hand side. Lines 5-28 iterate over each such clause; for every attribute appearing in ϕ, the hashtable H is consulted to extract all relevant entries that assign access permissions over

Input: Relational schema S, access control policy A, publishing transducer $\Pi = (Q, \Sigma, q_0, \delta)$, identity mapping function I
Output: A secure publishing transducer $\Pi' = (Q, \Sigma, q_0, \delta')$

```
1  δ' ← ∅;
2  H ← parseAndSortACP(A);
3  foreach r ∈ δ do
4      r' ← newRule();
5      foreach clause_i = (q_i, a_i, φ_i) ∈ RHS(r) do
6          condEntries ← ∅;
7          B ← 0^|F|;
8          foreach att ∈ atts(clause_i) do
9              foreach entry ∈ H.getNonConditionalEntries(att) do
10                 B.setBit(entry);
11             end
12             condEntries ← condEntries ∪ H.getConditionalEntries(att);
13         end
14         if condEntries = ∅ then
15             clause_new ← (q_i, a_i, B, φ_i);
16             r'.append(clause_new);
17         else
18             foreach entry ∈ condEntries do
19                 clause_deny ←
20                     (q_i, a_i, B, φ_i ∧ ¬entry.getConditions());
21                 B.setBit(entry);
22                 clause_grant ←
23                     (q_i, a_i, B, φ_i ∧ entry.getConditions());
24                 r'.append(clause_deny);
25                 r'.append(clause_grant);
26             end
27         end
28     end
29     δ' ← δ' ∪ r';
30 end
```

Algorithm 3.3.1: Construction of a secure publishing transducer.

that attribute. Lines 8-13 process the unconditional entries for the current attribute; for each such entry, the appropriate bit is set in the bitstring B (i.e., the values of the mapped ID, permission, and grant option fields within the entry are used to determine this bit position). If there are no conditional entries for the current attribute, the augmented clause is constructed by inserting B. Otherwise, the conditional entries are then processed one-by-one (lines 18-26). For each such entry, two clauses are constructed: one to handle the case when the condition is not satisfied, and the other to allow additional access to each ID $I(u)$ when

the condition c is satisfied.

Once all of the clauses within the original transduction rule have been processed in this way, the augmented rule is added to the transduction rule set of the secure publishing transducer (line 29).

At this point, each constructed clause (q_i, a_i, B_i, ϕ_i) serves to associate the access bit-string B_i to each database object (cell) o appearing within the relation formed by an evaluation of query ϕ_i on the instance database D. To complete the proof, we need to show that both preservation conditions are guaranteed by Alg. 3.3.1, for *any* instance database D conforming to S. For the sufficiency condition, observe that if a database object $o \in adom(\Pi(D))$ and furthermore, $o \in accessible(f, p)$ for some federated ID f and permission p, then by definition there must be a policy rule in A granting p on o to some u for which $I(u) = f$. In such a case, Alg. 3.3.1 will set the appropriate permission bit for f to 1, ensuring that f holds p over o. For the necessity condition, note that if $o \notin accessible(f, p)$, then there is no policy rule in A granting access to any u for which $I(u) = f$. In such a case, it is easy to verify that in Alg. 3.3.1 the permission bit for f will remain 0 regardless of whether $o \in adom(\Pi(D))$, meaning that f does not have the designated permission over o, and hence $o \notin xview(f, p)$. □

Let $|r|$ designate the size of transduction rule $r \in \delta$, measured as the number of clauses appearing on the right-hand side of r. Further, let $|r|_{max}$ denote the largest such $|r|$ in δ. The time requirement of Alg. 3.3.1 is $O(|\delta| \cdot |r|_{max} \cdot |S| \cdot |U|)$, while the storage requirement for the hashtable generated in the first phase is $O(|S| \cdot |U|)$. The time requirement is derived by noting that the loop in line 3 requires $|\delta|$ iterations, while the loop in line 5 requires at most $|r|_{max}$ iterations (one for every clause contained in the right-hand side of the current transduction rule). The number of iterations for the loops in lines 8 and 9 are upper bounded, respectively, by the number of relational attributes in S, and the number of local identities in U. The size of the $condEntries$ set being operated upon in lines 14-27 is again upper bounded by $|U|$, since for a fixed relational attribute, there can be at most $|U|$ entries in $condEntries$: one for each local identity in U.

Example 3.3.7. *We describe the secure publishing transducer that is constructed by Alg. 3.3.1,*

using as inputs the publishing transducer from Ex. 3.2.2, together with the ACP from Ex. 3.2.4. In the first phase, the ACP is parsed as described above, and the following hashtable is constructed:

$$\begin{aligned}
\texttt{Patient.ssn} \rightarrow\ & (\texttt{ID1}, select, f, \emptyset), (\texttt{ID1}, update, f, \emptyset), \\
& (\texttt{ID2}, select, f, \emptyset), \\
& (\texttt{ID3}, select, f, \texttt{Patient.name} = \\
& \text{`}Carol')
\end{aligned}$$

$$\begin{aligned}
\texttt{Patient.name} \rightarrow\ & (\texttt{ID1}, select, f, \emptyset), (\texttt{ID1}, update, f, \emptyset), \\
& (\texttt{ID2}, select, f, \emptyset), \\
& (\texttt{ID3}, select, f, \texttt{Patient.name} = \\
& \text{`}Carol')
\end{aligned}$$

$$\begin{aligned}
\texttt{Patient.age} \rightarrow\ & (\texttt{ID1}, select, f, \emptyset), (\texttt{ID1}, update, f, \emptyset), \\
& (\texttt{ID2}, select, f, \emptyset), \\
& (\texttt{ID3}, select, f, \texttt{Patient.name} = \\
& \text{`}Carol')
\end{aligned}$$

The resulting secure publishing transducer is $\Pi' = (Q, \Sigma, q_0, \delta')$*, with* $Q = \{q_0, q_1\}$*,* $\Sigma = \{@ssn, name, age\}$*, and* δ' *consisting of the transduction rules listed in Fig. 3.7. An example XML tree output by* Π' *is depicted in Fig. 3.2.* □

In this example, there are two distinct bitstring values, `111110000000` and `111110001000`; in particular, note that the application of the transduction rules in δ' assigns each node in `<patient>` subtrees the `111110000000` bitstring, with the exception of the nodes in the `<patient>` subtree containing Carol's record, which receive bitstring `111110001000`. This preserves the intent of the relational access policy rules, which grant `UserA` and `UserB`, mapped to respective federated IDs `ID1` and `ID2`, select permissions over all patient records, while `ID3`, to which `UserC` is mapped, only possesses select permission over the record belonging to Carol. Furthermore, `UserA`, under the federated ID `ID1`, receives update permission over all nodes.

$r_0 : (q_0, visits, 111110001000) \rightarrow$
$(q_1, patient, 111110000000, \phi_1(s,n,a;\emptyset)),$
$(q_1, patient, 111110001000, \phi_2(s,n,a;\emptyset)),$
where
$\phi_1(s,n,a) = \texttt{Patient}(s,n,a) \wedge n \neq \text{`}Carol'$
$\phi_2(s,n,a) = \texttt{Patient}(s,n,a) \wedge n = \text{`}Carol'$

$r_1 : (q_1, patient, 111110000000) \rightarrow$
$(q_1, @ssn, 111110000000, \phi_3(s;\emptyset)),$
$(q_1, name, 111110000000, \phi_4(n;\emptyset)),$
$(q_1, age, 111110000000, \phi_5(a;\emptyset)),$
where
$\phi_3(s) = \exists n, a\, Reg_{patient}(s,n,a)$
$\phi_4(n) = \exists s, a\, Reg_{patient}(s,n,a)$
$\phi_5(a) = \exists s, n\, Reg_{patient}(s,n,a)$

$r_2 : (q_1, patient, 111110001000) \rightarrow (q_1, @ssn, 111110001000, \phi_3(s;\emptyset)),$
$(q_1, name, 111110001000, \phi_4(n;\emptyset)),$
$(q_1, age, 111110001000, \phi_5(a;\emptyset))$

$r_3 : (q_1, @ssn, 111110000000) \rightarrow (q_1, text, 111110000000, \phi_6(s;\emptyset)),$
where $\phi_6(s) = Reg_{@ssn}(s)$

$r_4 : (q_1, @ssn, 111110001000) \rightarrow (q_1, text, 111110001000, \phi_6(s;\emptyset))$

$r_5 : (q_1, name, 111110000000) \rightarrow (q_1, text, 111110000000, \phi_7(n;\emptyset)),$
where $\phi_7(n) = Reg_{name}(n)$

$r_6 : (q_1, name, 111110001000) \rightarrow (q_1, text, 111110001000, \phi_7(n;\emptyset))$

$r_7 : (q_1, age, 111110000000) \rightarrow (q_1, text, 111110000000, \phi_8(a;\emptyset)),$
where $\phi_8(a) = Reg_{age}(a)$

$r_8 : (q_1, age, 111110001000) \rightarrow (q_1, text, 111110001000, \phi_8(a;\emptyset))$

$r_9 : (q_1, text, 111110000000) \rightarrow .$ /* empty string */

$r_{10} : (q_1, text, 111110001000) \rightarrow .$ /* empty string */

Figure 3.7: Transduction rules for the example secure publishing transducer.

As shown in the example, the biggest complication is caused by the final ACP rule of A, granting `UserC` access to a view containing only Carol's record. This necessitates expanding the original transduction rule for `patient` in order to output the correct permission bitstring, according to whether or not the associated patient name equals "Carol". Also, the transduction rules for each child element of patient are affected; in particular, we now need two separate rules for each child element to handle the cases where the parent `<patient>` element is – and is not – accessible to `ID3`.

3.4 Verifying Policies

A key concern for security administrators is ensuring that an ACP fulfills the needs of the corresponding application. When exchanging data in a federation, this also involves *verifying* that the XML publishing strategy exposes the original relational data in accordance with the original ACP. In this section we show that this problem is computationally very hard even for moderately expressive access control models. Given the importance of the problem, our results emphasize the need for efficient and effective ways of *automatically translating* ACPs.

We model the verification problem as follows. Given a relational schema S, the corresponding relational ACP A, an identity mapping function I, a fixed publishing transducer Π, and secure publishing transducer Π', the problem consists of determining whether 1) Π' is *equivalent* to Π (i.e., on the same instance of S it produces an XML tree equivalent to that produced by Π), and 2) the relational ACP A and the access bitstring assignments in Π' have equivalent semantics.

We address two variants of the verification problem: a *static analysis* ensures that a supplied secure publishing transducer preserves the semantics of a relational ACP A defined over any instance of relational schema S, while a *dynamic analysis* takes a specific instance database D as an additional input and only ensures that the specified secure publishing transducer satisfies A over D only (and not necessarily all databases that conform to S). These variants can be defined formally as follows.

Definition 3.4.1 (Dynamic Verification)**.** *An instance of the* dynamic verification decision problem *consists of the following inputs: an instance database D conforming to a relational schema S; a relational ACP defined over S; an identity mapping function I; a publishing transducer Π; and a secure publishing transducer Π'. A "yes" instance is constituted only if 1) $\Pi'(D)$ is* equivalent *to $\Pi(D)$ (i.e., on input D, Π' produces an XML tree equivalent to that produced by Π), and 2) the relational ACP A and the access bitstring assignments in Π' have equivalent semantics. All other instances constitute "no" instances.* □

Definition 3.4.2 (Static Verification)**.** *An instance of the* static verification decision problem *consists of the following inputs: a relational schema S; a relational ACP defined over*

S; an identity mapping function I; a publishing transducer Π*; and a secure publishing transducer* Π'*. A "yes" instance is constituted only if 1)* Π' *is* equivalent *to* Π *(i.e., on every legal instance of S it produces an XML tree equivalent to that produced by* Π*), and 2) the relational ACP A and the access bitstring assignments in* Π' *have equivalent semantics. All other instances constitute "no" instances.* □

We show the complexity of each variant for a variety of classes of secure publishing transducers (SPT) specified as SPT($\mathcal{L}, \mathcal{S}, \mathcal{A}$), where

- $\mathcal{L}$ is the language for the ϕ queries appearing in transduction rules: `CQ` (conjunctive queries) or `FO` (first-order queries);
- $\mathcal{S}$ takes the value `tp` or `rl`, indicating whether each node register in the produced XML tree stores a single tuple or a relation; and
- $\mathcal{A}$ specifies the complexity of queries appearing in the relational ACP rules: `CQ` or `FO`.

In addition, SPT$_{nr}$ denotes the more restrictive class of secure publishing transducers that lack recursive transduction rules. Table 3.1 summarizes the complexity of deciding the dynamic and static verification problems for various classes of secure publishing transducers with different expressive power. The results follow from the complexity of reasoning with the languages used to define the queries within transduction rules and/or within the queries of the relational ACP. (Recall from Sec. 2.2.4 that practical relational query languages such as SQL are essentially equivalent to `FO`, while `CQ` are the subset of `FO` consisting of selections, projections and joins.)

3.4.1 Dynamic Verification

Since it is only relevant to a particular instance database (and says nothing about other instance databases conforming to the same schema), the usefulness of dynamic verification is mainly constrained to those cases in which the contents of a database do not change frequently (e.g., archival data). The following procedure automates the process. We start by deriving a secure publishing transducer Π'' as discussed in the previous section, using

Fragment	**Dynamic Verification**	**Static Verification**
SPT($\mathcal{L}$, rl, $\mathcal{A}$)	2EXPTIME	undecidable
SPT($\mathcal{L}$, tp, $\mathcal{A}$)	EXPTIME	undecidable
SPT$_{nr}$(FO,tp,$\mathcal{A}$)	PTIME	undecidable
SPT$_{nr}$(CQ,tp,$\mathcal{A}$)	PTIME	Π_3^P-complete

Table 3.1: Complexity of static and dynamic verification of common classes of secure publishing transducers.

the supplied inputs Π, A, and I. We obtain XML trees $X_1 = \Pi''(D)$ and $X_2 = \Pi'(D)$ by supplying the instance database D as an input to both secure publishing transducers, and simply compare the two resulting annotated trees.

More precisely, we traverse both trees in depth-first order simultaneously; at each step, we check whether the corresponding nodes in X_1 and X_2:

1. have the same label;
2. are *isomorphic* [5]; and
3. have the same access bitstrings associated with them.

Two observations are relevant at this stage. First, we can compare the access bitstrings in the nodes of the two trees because we assume that there is a unique, *fixed* list of users in the federation (recall Section 3.3). Second, the complexity of the problem boils down to the complexity of materializing the XML trees given the transducers. In the presence of recursion, this ranges from EXPTIME for transducers with tuple registers to 2EXPTIME for those with relation registers [51]; for non-recursive transducers, the size of the output tree is bounded by a polynomial in the size of the instance database, allowing dynamic verification to be carried out in PTIME.

[5] In practice (assuming that the tuple ordering of the input relation supplied to both transducers is consistent), we need only check whether they have the same number of child nodes, since the first condition establishes that they share the same label, and this same condition applied recursively to each child node establishes a pairwise label agreement for X_1 and X_2 at each child position.

3.4.2 Static Verification

Static verification is, as expected, much harder as it involves reasoning about the queries in the transducers and the ACPs. We first obtain a secure publishing transducer Π'' as discussed in the previous section. The verification of $\Pi'(D)$ then consists of solving two sub-problems: (1) checking that for *every* instance D conforming to schema S, $\Pi''(D) \equiv \Pi'(D)$[6]; and (2) ensuring the relational access control policy A and the access bitstrings in Π' share the same semantics.

Checking the first condition above requires one to decide equivalence between two publishing transducers. As shown in [51], this is undecidable for all classes of transducers except for SPT$_{nr}$(CQ,tp, $\mathcal{A}$), in which case deciding equivalence is Π_3^P-complete. For the second condition, we consider only this restricted class. Now, the problem consists of checking whether the bitstrings produced by equivalent nodes in Π'' and Π' are identical. We note that all such pairwise node equivalences are found during the process of establishing overall transducer equivalence in the previous step (this is so since establishing transducer equivalence requires one to consider *all* sequential applications of transduction rules starting from the start rule which are formed by supplying every instance database conforming to S as an input). Assuming that node equivalence relationships are recorded at the time of their discovery, we can test the second condition in time polynomial in the number of transduction rules in δ' [7], the number of federated IDs, and the number of assignable permissions, by looking up each stored relationship, and performing a bitwise comparison between the respective bitstrings assigned to both nodes in the current relationship to ensure they are in fact equal. Evidently, the cost of testing the first condition dominates the cost of deciding the second condition. This approach assumes a fixed list of federated IDs, and a scenario in which a consistent encoding of federated permissions within bitstrings was not guaranteed would require more sophisticated reasoning on bitstring values.

[6] Ignoring the bitstrings in $\Pi'(D)$.

[7] Recall that this serves as a bound on the length of the longest applicable rule sequence, as we are considering a non-recursive class of transducers.

3.5 Expressing Policies in XACML

In this section, we provide an algorithm for generating an XACML policy from a non-recursive [8] secure publishing function Π', conforming to the syntax of the hierarchical resource profile of XACML 2.0 [5]. Note that since the conversion is done entirely at the schema level, this procedure only needs to be carried out once for each Π': thereafter, the generated XACML policy will be equally applicable to all generated XML trees $\Pi'(D)$ formed by applying Π' to any relational database instance D.

Definition 3.5.1 (XACML Policy)**.** *An* XACML policy *is a 3-tuple* $\mathcal{X} = \langle t_d, rca, rs \rangle$*, where* t_d *is the* target *XML document over which the policy applies,* rca *is a* rule combining algorithm *indicating how policy rule conflicts are resolved, and* rs *is a set of* policy rules*. Each rule in* rs *is a tuple* $\langle su, re, ac, ef, co \rangle$*, where* su*,* re*, and* ac *denote, respectively, the sets of* subjects*,* resources*, and* actions *for which the rule applies;* ef *is the* effect *of the rule (*permit *or* deny*); and* co *is a set of boolean* conditions *that serve to restrict the applicability of the rule to cases in which the conditions all evaluate to* true*. Such conditions may be defined over attributes relating to subjects, resources, actions, and/or environmental variables (such as the time of the access request, supporting temporal constraints on access), allowing the applicability of a policy rule to be further refined. Resources are node sets identified by XPath expressions.* □

The procedure for translating a secure publishing function into an equivalent XACML policy is given by Alg. 3.5.1. A *rule reachability graph (RRG)* is first formed from the set of transduction rules δ' for Π' (line 2), in which the root node corresponds to the transduction rule defined for the start state and root node label. Furthermore, a directed edge from the node corresponding to transduction rule δ_1 into that of δ_2 indicates that δ_1 contains a reference to δ_2 within a clause c_i in its RHS. Each edge is labeled with the query ϕ_i contained in the clause c_i that formed the edge. If Π' is recursive, then its rule reachability graph contains at least one cycle. Fig. 3.8 shows the rule reachability graph for the secure publishing transducer of Ex. 3.3.7.

[8] This assumption is made to simplify the discussion. Extending the presented algorithm to handle recursive publishing functions would be straightforward, requiring one to "mark" cycles as they are encountered during a preorder traversal of a directed graph.

Input: Secure publishing transducer Π'
Output: A set of XACML policy rules rs

1 $rs \leftarrow \emptyset$;
2 $RRG \leftarrow constructRuleReachabilityGraph(\Pi')$;
3 **foreach** $n \in nodeListInPreOrder(RRG)$ **do**
4 $\quad currentPath \leftarrow getLocation(n)$;
5 $\quad bString \leftarrow getBitString(n)$;
6 $\quad parentBString \leftarrow getBitString(getParentNode(n))$;
7 $\quad$ **if** $n.hasCondition()$ **then**
8 $\quad\quad currentPath \leftarrow resolveCondition(n)$;
9 $\quad$ **end**
10 $\quad permitsGroups \leftarrow \emptyset; denialsGroups \leftarrow \emptyset$;
11 $\quad$ **foreach** $perm \in P'$ **do**
12 $\quad\quad permit_{perm} \leftarrow$
13 $\quad\quad\quad getIDsWithNewlyPermittedAccess(bString, parentBString, perm)$;
14 $\quad\quad deny_{perm} \leftarrow$
15 $\quad\quad\quad getIDsWithNewlyDeniedAccess(bString, parentBString, perm)$;
16 $\quad\quad permitsGroups.add(permit_{perm})$;
17 $\quad\quad denialsGroups.add(deny_{perm})$;
18 $\quad$ **end**
19 $\quad mergedPermitsGroup \leftarrow$
20 $\quad\quad mergeGroupsWithSameIDSet(permitsGroup)$;
21 $\quad mergedDenialsGroup \leftarrow$
22 $\quad\quad mergeGroupsWithSameIDSet(denialsGroup)$;
23 $\quad$ **foreach** $pg \in mergedPermitsGroup$ **do**
24 $\quad\quad associatedPerms \leftarrow getPermissions(pg)$;
25 $\quad\quad associatedIDs \leftarrow getIDs(pg)$;
26 $\quad\quad rs.addPolicyRule(\langle associatedIDs, currentPath,$
27 $\quad\quad\quad associatedPerms, permit, \emptyset \rangle)$;
28 $\quad$ **end**
29 $\quad$ **foreach** $dg \in mergedDenialsGroup$ **do**
30 $\quad\quad associatedPerms \leftarrow getPermissions(dg)$;
31 $\quad\quad associatedIDs \leftarrow getIDs(dg)$;
32 $\quad\quad rs.addPolicyRule(\langle associatedIDs, currentPath,$
33 $\quad\quad\quad associatedPerms, deny, \emptyset \rangle)$;
34 $\quad$ **end**
35 **end**
36 **return** rs;

Algorithm 3.5.1: Translation of a secure publishing function into an equivalent set of XACML policy rules.

The constructed RRG is then traversed in preorder (lines 3-35). For each encountered node n, variables $currentPath$ and $bString$ are updated to refer to, respectively, the XPath path value indicating the location of n relative to the graph's root node (line 4) and the access bitstring for n (line 5). Additionally (line 6), the variable $parentBString$ is used to store the access bitstring for the parent node of n (in the case of the root node, the zero bitstring is stored).

In the next step (lines 7-9), an attempt is made to translate the condition (if any) specified within the query ϕ labelling the edge incident to n into an equivalent XPath predicate. Such a condition will be of the form $a\ \mathtt{op}\ c$, where a is a relational attribute, $\mathtt{op}$ a comparison operator, and c a constant value. The descendants of n are traversed in an attempt to discover the location at which the value of a is output; a stack s is used to keep track of the current location path, relative to n. If and when such a location is determined, the current path value is updated to $currentPath = currentPath + \text{“}[s.top()\ \mathtt{op}\ c]''$, where $s.top()$ denotes the value on top of s. We then say that this condition has been *resolved*. If the search fails, then the condition remains *unresolved*. We defer until later the discussion of how unresolved conditions are handled.

In the next phase, the access bitstrings of n and its parent are compared (lines 11-18). By P', we denote the set formed by augmenting the original permission set P with an additional element p' for each $p \in P$, used to indicate the permission p extended to include the grant option. For each permission $perm$ in P', two groups are formed: $permit_{perm}$ contains the set of federated IDs for whom the corresponding bit positions for $perm$ are set in $bString$ and unset in $parentBString$ (line 12), while $deny_{perm}$ holds those IDs for whom the corresponding bit positions are unset in $bString$ yet set in $parentBString$ (line 14). Once all permissions in P' have been processed, sets $permitsGroup$ and $denialsGroup$ serve to store these sets for all permissions in P' (lines 16-17).

Seeking to minimize the number of constructed policy rules, individual sets in $permitsGroup$ and $denialsGroup$ that contain the same sets of federated IDs are then merged; each set thus formed is then associated with the union of permissions of each original ID set. The resulting sets are designated as $mergedPermitsGroup$ and $mergedDenialsGroup$ (lines 19-22). In the final step (lines 23-34), a separate XACML policy rule is constructed and added to the policy rule set rs for each member of $mergedPermitsGroup$ and $mergedDenialsGroup$.

Example 3.5.2. *We illustrate the discussion using the secure publishing transducer of Ex. 3.3.7 as an example. The corresponding RRG is depicted in Fig. 3.8, while the generated policy rule set is shown in Table 3.2. From the root node's access bitstring* $\mathtt{111110001000}$*, the groups* $permit_{select} = \{\mathtt{ID2}, \mathtt{ID3}\}$*,* $permit_{updateGO} = \{\mathtt{ID1}\}$*, and* $permit_{selectGO} =$

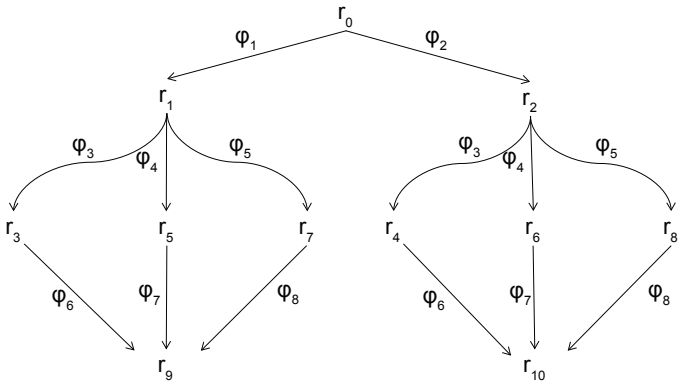


Figure 3.8: Rule reachability graph.

r_1 :	⟨ID1, "/visits", {selectWithGrantOption, updateWithGrantOption}, permit, ∅⟩
r_2 :	⟨{ID2, ID3}, "/visits", select, permit, ∅⟩
r_3 :	⟨ID3, "/visits/patient[name/text() neq 'Carol']", select, deny, ∅⟩

Table 3.2: Example XACML policy rules.

{ID1} *are created, corresponding to the set bits in the access bitstring. The latter two groups share the same ID set (*ID1*), and are therefore merged into a single rule associated with both of these permissions. This in turn results in the creation of policy rules* r_1 *and* r_2 *in Table 3.2. The next node to be visited is* $(q_1, patient, 111110000000)$*, setting* $currentPath =$ "$/visits/patient$"*. The condition* $n \neq$ "$Carol$" *is resolved by first recording that the referenced attribute* n *is the second attribute stored in the* $Reg_{patient}$*. Next, the child edges are traversed in breadth-first order; since the query associated with the edge leading into* $(q_1, @ssn, 111110000000)$ *does not store* n *as part of its answer, this edge is disqualified. The query associated with the edge incident to* $(q_1, name, 111110000000)$ *does store* n *within* Reg_{name}*, so the temporary stack* s *is updated to store* $name$*, and the edge leading to* $(q_1, text, 111110000000)$ *is next followed; since this rule outputs the value of* n*,* s *is updated to* $name/text()$*, and the search terminates. The* $currentPath$ *is updated to* $/visits/patient[name/text()\ neq$ "$Carol$"$]$*. The current bitstring* B_c *differs with that of the parent (root) node only in the ninth bit position, which results in the creation of the set* $deny_{select} =$ {ID3} *and yields the policy rule* r_3*. Traversing the remaining nodes in the RRG does not produce any additional rules.* □

Handling unresolved conditions. The condition resolution procedure outlined above

depends on the values of each referenced relational attribute being output as part of the published XML document; in practice, this may not always happen (for instance, if the patient records in the above example are anonymized, the value of each patient name will not be output by the publishing function). In such cases, a different solution is needed. One technique is to augment the secure publishing function with additional rules that output the missing values and assign to each the zero bitstring. This serves to ensure that no federated IDs will have access to such values (thereby obeying the intentions of the original access control policy), while allowing all conditions within the RRG to be resolvable.

Example 3.5.3. *For the case alluded to above, where patient names are not visible to any federated IDs, the modified transduction rule set in Fig. 3.9 will allow all conditions to be resolved (the bolded rules indicate those that have been altered from the original version in Fig. 3.7).* □

3.6 Minimizing Translated Policies

Multiple reasons exist for desiring that translated policies are as "small as possible", both in terms of the number of generated policy rules as well with respect to the size of the relational queries appearing within policy rules. An obvious motivator is that a simpler policy is typically easier for a human security administrator to interpret. Another relates to application performance: a smaller set of policy rules means that fewer rules are checked each time an access control decision needs to be made.

Based on the earlier results on static verification, the difficulty of performing transducer minimization should be evident: indeed, an alternative way of carrying out static verification would be to apply the minimization procedure on the supplied S.P.T. Π' and the S.P.T. Π'' derived from the verification inputs S, A, and Π, and check to see whether the resulting minimized transducers are equivalent. Hence, minimization must be at least as hard as static verification. For this reason, we focus on minimizing the one class of S.P.T.s for which static verification is decidable, $\text{SPT}_{nr}(\text{CQ,tp,}\mathcal{A})$, and rely on a "weaker" notion of equivalence to guide the minimization procedure. We also leverage prior results on query containment [29, 116] to identify particular subclasses of conjunctive queries for which

$r_0 : (q_0, visits, 111110001000) \rightarrow$
$(q_1, patient, 111110000000, \phi_1(s,n,a;\emptyset))$,
$(q_1, patient, 111110001000, \phi_2(s,n,a;\emptyset))$,
where
$\phi_1(s,n,a) = \texttt{Patient}(s,n,a) \wedge n \neq \text{`}Carol'$
$\phi_2(s,n,a) = \texttt{Patient}(s,n,a) \wedge n = \text{`}Carol'$

$\mathbf{r_1} : (\mathbf{q_1}, \mathbf{patient}, \mathbf{111110000000}) \rightarrow$
$(\mathbf{q_1}, @\mathbf{ssn}, \mathbf{111110000000}, \phi_{\mathbf{3}}(\mathbf{s};\emptyset))$,
$(\mathbf{q_1}, \mathbf{name}, \mathbf{000000000000}, \phi_{\mathbf{4}}(\mathbf{n};\emptyset))$,
$(\mathbf{q_1}, \mathbf{age}, \mathbf{111110000000}, \phi_{\mathbf{5}}(\mathbf{a};\emptyset))$,
where
$\phi_{\mathbf{3}}(\mathbf{s}) = \exists \mathbf{n}, \mathbf{a}\, \mathbf{Reg_{patient}}(\mathbf{s},\mathbf{n},\mathbf{a})$
$\phi_{\mathbf{4}}(\mathbf{n}) = \exists \mathbf{s}, \mathbf{a}\, \mathbf{Reg_{patient}}(\mathbf{s},\mathbf{n},\mathbf{a})$
$\phi_{\mathbf{5}}(\mathbf{a}) = \exists \mathbf{s}, \mathbf{n}\, \mathbf{Reg_{patient}}(\mathbf{s},\mathbf{n},\mathbf{a})$

$\mathbf{r_2} : (\mathbf{q_1}, \mathbf{patient}, \mathbf{111110001000}) \rightarrow (\mathbf{q_1}, @\mathbf{ssn}, \mathbf{111110001000}, \phi_{\mathbf{3}}(\mathbf{s};\emptyset))$,
$(\mathbf{q_1}, \mathbf{name}, \mathbf{000000000000}, \phi_{\mathbf{4}}(\mathbf{n};\emptyset))$,
$(\mathbf{q_1}, \mathbf{age}, \mathbf{111110001000}, \phi_{\mathbf{5}}(\mathbf{a};\emptyset))$

$r_3 : (q_1, @ssn, 111110000000) \rightarrow (q_1, text, 111110000000, \phi_6(s;\emptyset))$,
where $\phi_6(s) = Reg_{@ssn}(s)$

$r_4 : (q_1, @ssn, 111110001000) \rightarrow (q_1, text, 111110001000, \phi_6(s;\emptyset))$

$\mathbf{r_5} : (\mathbf{q_1}, \mathbf{name}, \mathbf{000000000000}) \rightarrow (\mathbf{q_1}, \mathbf{text}, \mathbf{000000000000}, \phi_{\mathbf{7}}(\mathbf{n};\emptyset))$,
where $\phi_{\mathbf{7}}(\mathbf{n}) = \mathbf{Reg_{name}}(\mathbf{n})$

$r_7 : (q_1, age, 111110000000) \rightarrow (q_1, text, 111110000000, \phi_8(a;\emptyset))$,
where $\phi_8(a) = Reg_{age}(a)$

$r_8 : (q_1, age, 111110001000) \rightarrow (q_1, text, 111110001000, \phi_8(a;\emptyset))$

$r_9 : (q_1, text, 111110000000) \rightarrow .$ /* empty string */

$r_{10} : (q_1, text, 111110001000) \rightarrow .$ /* empty string */

$\mathbf{r_{11}} : (\mathbf{q_1}, \mathbf{text}, \mathbf{000000000000} \rightarrow$ ./* empty string */

Figure 3.9: Modified transduction rules for the example secure publishing transducer (bolded rules indicate those modified from the original versions in Fig. 3.7).

minimization proves to be especially tractable.

Our minimization algorithm is a generalization of Hopcroft's $O(n \log n)$ algorithm for minimizing a deterministic finite automaton (DFA) with n states [64]. It operates by determining which transduction rules are equivalent, and combines all equivalent rules into a single transduction rule in the minimized S.P.T. We first establish the conditions for transduction rule equivalence.

Definition 3.6.1 (Conditions for Equivalence of Transducer Rule Clauses)**.** *We say that clauses* $c_{i_p} = (q_{i_p}, a_{i_p}, \beta_{i_p}, \phi_{i_p})$ *and* $c_{j_p} = (q_{j_p}, a_{j_p}, \beta_{j_p}, \phi_{j_p})$ *are equivalent, written* $c_{i_p} \equiv c_{j_p}$*, iff (1)* $a_{i_p} = a_{j_p}$*; (2)* $\beta_{i_p} = \beta_{j_p}$*; and (3)* $\phi_{i_p} \equiv \phi_{j_p}$*.* □

Recall from Def. 2.2.2 that the third condition mandates *query equivalence* between ϕ_{i_p} and ϕ_{j_p}.

Definition 3.6.2 (Conditions for Equivalence of Transducer Rules)**.** *Two transduction rules* $r_i = (q_i, a_i, \beta_i) \rightarrow c_{i_1}, \ldots, c_{i_k}$ *and* $r_j = (q_j, a_j, \beta_j) \rightarrow c_{j_1}, \ldots, c_{j_k}$ *within the transduction rule set* δ' *of a S.P.T.* Π' *are said to be* equivalent*, denoted by* $r_i \equiv r_j$*, iff the number of clauses on the RHS of* r_i *and* r_j *is equal, and further, there exists a pairwise equivalence between the clauses at each position; i.e, for* $1 \leq p \leq k$*,* $c_{i_p} \equiv c_{j_p}$*.*

If r_i *and* r_j *are not equivalent, we say they are* distinguishable*.* □

Note that when the above condition is satisfied, rules r_i and r_j will produce equivalent subtrees given the same instance database as input. Further, both subtrees will share the same access control semantics at every corresponding tree position.

A "Weaker" Notion of Transducer Equivalence

Definition 3.6.3 (Conditions for "Weak" Equivalence of Secure Publishing Transducers)**.** *We say that two S.P.T.s* Π'_1 *and* Π'_2 *are* weakly equivalent*, denoted* $\Pi'_1 \equiv_w \Pi'_2$*, if there exists a pairwise equivalence between their respective transduction rule sets.* □

The above definition equates to an equivalence between the respective RRGs of both S.P.T.s, up to state relabelling. This notion of equivalence does not necessarily capture *all* S.P.T.s generating the same XML tree given the same input database, thereby falling short of the criteria for "strong" transducer equivalence as defined in [49]. The following example demonstrates an instance where the "weak" notion of transducer equivalence from Def. 3.6.3 fails to classify two S.P.T.s as equivalent, even though they do output identical XML trees given the same input database, for all potential input databases.

Example 3.6.4. *Consider a schema* S *consisting of a single relation* $\mathtt{R}(A, B)$ *with domain* $\{x, y\}$*, and two S.P.T.s* Π'_1 *and* Π'_2 *defined over* S*. The respective transduction rule sets are*

$$
\begin{aligned}
r_{11} : (q_0, a, 1111) \rightarrow & \\
& (q_1, \epsilon, 1111, \phi_{11}(A, B; \emptyset)), \\
& \text{where} \\
& \phi_{11}(A, B) = \mathtt{R}(A, B) \wedge A = \text{`}x' \\
r_{12} : (q_1, \epsilon, 1111) \rightarrow & \\
& (q_2, b, 11111, \phi_{12}(A, B; \emptyset)), \\
& \text{where} \\
& \phi_{12}(A, B) = Reg_{\epsilon}(A, B) \wedge B = \text{`}x'
\end{aligned}
$$

Figure 3.10: Transduction rules for secure publishing transducer Π'_1.

$$
\begin{aligned}
r_{21} : (q_0, a, 1111) \rightarrow & \\
& (q_1, \epsilon, 1111, \phi_{21}(A, B; \emptyset)), \\
& \text{where} \\
& \phi_{21}(A, B) = \mathtt{R}(A, B) \wedge B = \text{`}x' \\
r_{22} : (q_1, \epsilon, 1111) \rightarrow & \\
& (q_2, b, 11111, \phi_{22}(A, B; \emptyset)), \\
& \text{where} \\
& \phi_{22}(A, B) = Reg_{\epsilon}(A, B) \wedge A = \text{`}x'
\end{aligned}
$$

Figure 3.11: Transduction rules for secure publishing transducer Π'_2.

listed in Fig. 3.10 and Fig. 3.11. These transducers are "strongly" equivalent, as they will produce the same XML tree given the same instance of R *as input. More precisely, each will produce an XML tree featuring an* a*-labelled node as root with a single* b *child if the tuple* $(x, x) \in$ R*, and a tree consisting of a single* a *node otherwise. However,* $\Pi'_1 \not\equiv_w \Pi'_2$*, since their respective RRGs are clearly not equivalent (e.g., the root rules* r_{11} *and* r_{21} *are distinguishable since the contained queries* ϕ_{11} *and* ϕ_{21} *lack equivalence).* □

The minimization procedure is listed as Alg. 3.6.1. In the first step (line 1) the RRG for the input S.P.T. Π' is converted into a DFA representation where each transduction rule is represented as a distinct state, and edges connect those states whose corresponding transduction rules are connected by the RRG. The pseudocode for the `GenerateDFAFromRRG`

subprocedure is listed as Alg. 3.6.2. Note that since Π' is non-recursive, its RRG – as well as the resulting DFA – will be acyclic. The constructed DFA $DFA_{\Pi'}$ is then used as an input to a modified version of Hopcroft's algorithm. In Step (4) of the `GenerateDFAFromRRG` subprocedure, another subprocedure called `QueryReduction` is invoked in order to minimize the size of the relational queries appearing in transduction rules, and to eliminate any redundant queries. The steps involved in `QueryReduction` are listed as Alg. 3.6.3.

Input: S.P.T. Π'
Output: a minimized S.P.T. Π'_{min} that is equivalent to Π'

1 $DFA_{\Pi'} = (Q_D, \Sigma_D, \delta_D, s_0, F_D) \leftarrow$ `GenerateDFAFromRRG`$(RRG_{\Pi'})$;
2 $splitList \leftarrow \{F_D, Q_D - F_D\}$;
3 $currentPartition \leftarrow \{F_D, Q_D - F_D\}$;
4 **while** $|splitList| > 0$ **do**
5 $\quad splitter \leftarrow selectElementFromList(splitList)$;
6 $\quad splitList \leftarrow splitList - splitter$;
7 $\quad$ **foreach** $a \in \Sigma_D$ **do**
8 $\quad\quad previousStatesSet \leftarrow \delta_D^{-1}(splitter, a)$;
9 $\quad\quad refinableSubsets \leftarrow \{s | s \in currentPartition$
10 $\quad\quad\quad \wedge s \cap previousStatesSet \neq \emptyset \wedge s \not\subseteq previousStatesSet\}$;
11 $\quad\quad$ **foreach** $rs \in refinableSubsets$ **do**
12 $\quad\quad\quad rs_1 \leftarrow rs \cap previousStatesSet$;
13 $\quad\quad\quad rs_2 \leftarrow rs - rs_1$;
14 $\quad\quad\quad currentPartition \leftarrow currentPartition - rs$;
$\quad\quad\quad currentPartition \leftarrow currentPartition \cup rs_1 \cup rs_2$;
15 $\quad\quad\quad$ **if** $rs \in splitList$ **then**
16 $\quad\quad\quad\quad splitList \leftarrow splitList - rs$;
17 $\quad\quad\quad\quad splitList \leftarrow splitList \cup rs_1 \cup rs_2$;
18 $\quad\quad\quad$ **else**
19 $\quad\quad\quad\quad$ **if** $|rs_1| \leq |rs_2|$ **then**
20 $\quad\quad\quad\quad\quad splitList \leftarrow splitList \cup rs_1$;
21 $\quad\quad\quad\quad$ **else**
22 $\quad\quad\quad\quad\quad splitList \leftarrow splitList \cup rs_2$;
23 $\quad\quad\quad\quad$ **end**
24 $\quad\quad\quad$ **end**
25 $\quad\quad$ **end**
26 $\quad$ **end**
27 **end**
28 `MergeStates`$(DFA_{\Pi'})$;
29 **return** $\Pi'_{min} \leftarrow$ `GenerateRRGFromDFA`$(DFA_{\Pi'})$;

Algorithm 3.6.1: S.P.T. minimization procedure.

Hopcroft's algorithm is then applied to the resulting DFA (lines 2-27). The core strategy of this algorithm involves repeated refinement of a partition of states based on a "splitter"

Input: rule reachability graph $RRG_{\Pi'}$ for a S.P.T. Π'
Output: a complete and initially connected deterministic finite automaton $DFA_{\Pi'} = (Q_D, \Sigma_D, \delta_D, s_0, F_D)$

1. Construct a state s_0 in $DFA_{\Pi'}$ corresponding to the starting transduction rule (q_0, r, β_r) in $RRG_{\Pi'}$. Add a final state s_t to F_D for each terminating transduction rule r_t. Add s_0 and each final state s_t to Q_D.

2. For each remaining transduction rule r_i ($i = 1, \ldots, |\delta'|$) appearing in $RRG_{\Pi'}$, construct a corresponding state s_i and add it to Q_D.

3. For each edge $e = (r_i, r_j)$ in $RRG_{\Pi'}$, add an edge $e_d = (s_i, s_j)$ to $DFA_{\Pi'}$. Assign label $\langle o_j, a_j, \beta_j, \phi_j \rangle$ to e_d, where o_j indicates the ordinal position of e relative to other edges originating from r_i and $a_j \in \Sigma$, β_j, and ϕ_j are the tag symbol, access bitstring, and relational query, respectively, appearing within the clause on the RHS of r_i referring to r_j. Assign to Σ_D the set of all such labels. Add corresponding entries $(s_i, \langle o_j, a_j, \beta_j, \phi_j \rangle) \rightarrow s_j$ to δ_D.

4. Perform algorithm `QueryReduction` on the set Φ of queries appearing in Σ_D. For each query ϕ that has been replaced by an equivalence symbol ϕ_{min}, substitute each reference to ϕ in Σ_D with ϕ_{min}.

5. If necessary, remove any states which are not reachable from s_0. Ensure that the transduction function δ_D is total by adding a cyclic edge (s_i, s_i) labelled $\langle o_j, a_j, \beta_j, \phi_j \rangle$ for each entry (s_i, a_j) missing from δ_D, where o_j is assigned the next consecutive ordinal value.

6. Return the resulting DFA $DFA_{\Pi'}$.

Algorithm 3.6.2: Algorithm `GenerateDFAFromRRG` for converting a rule reachability graph $RRG_{\Pi'}$ into a deterministic finite automaton $DFA_{\Pi'}$.

criteria consisting of a state and an alphabet symbol. Once all such "splitters" have been considered, all member states in any partition set of size greater than one are replaced by a single state in the minimized DFA; in such a case, the partition set holding the start state becomes the start state in the reduced DFA, while those sets holding final states are each replaced by a single final state (line 28). The minimized $DFA_{\Pi'}$ is then converted back into an RRG representation of the minimized S.P.T. (line 29).

Lemma 3.6.5. *Alg. 3.6.1 always produces an equivalent S.P.T. of minimal size.*

Proof. (Sketch.) It suffices to show that the algorithm is both *sound* (i.e., it merges two transduction rules only if they are in fact equivalent according to the criteria of Def. 3.6.2) and *complete* (i.e. all pairs of rules meeting such criteria are in fact merged by the algorithm). To demonstrate soundness, we observe that if two rules r_i and r_j are equivalent, ac-

Input: initial set of relational queries Φ
Output: a reduced set of relational queries Φ_{min}

1. *Expand* each query $\phi \in \Phi$ to replace any clauses referring to local node registers with equivalent clauses expressed over the base relation(s). Let ϕ_{expand} denote the expanded form of ϕ.
2. Perform query minimization on each ϕ_{expand}, using the algorithm appropriate for the query class containing ϕ_{expand}.
3. *Contract* each ϕ_{expand} by replacing any clauses containing references to base relation(s) with equivalent clauses expressed over local node registers. Let ϕ_{min} denote the contracted version of ϕ. Add each unique ϕ_{min} to Φ_{min}.
4. Return Φ_{min}.

Algorithm 3.6.3: Algorithm `QueryReduction` for combining equivalent relational queries.

cording to Def. 3.6.2, there must be a pairwise equivalence between their respective clauses at each position $1, \ldots, k$. Then Step (3) of Alg. 3.6.2 will assign identical edge labels to each such pair of outgoing edges leading from s_i and s_j in the constructed DFA, and the ordered set of states incident to these edges will also be identical. At this stage, we rely on the established soundness of the Hopcroft algorithm [64] to ensure that states s_i and s_j cannot be distinguished by any choice of splitter, and hence will be merged into a single state.

To demonstrate *completeness*, we assume that the algorithm has "incorrectly" merged two rules r_i and r_j. If r_i and r_j are in fact distinguishable, then again by Def. 3.6.2 there must be at least one position p for which c_{i_p} is not equivalent to c_{j_p}: they differ in at least one of the tag, access bitstring, or relational query values. But according to Step (3) of Alg. 3.6.2, any of these three cases would result in the p-th outgoing edges leading from s_i and s_j receiving distinct labels, say l_{i_p} and l_{j_p}. Relying on the established completeness result for the Hopcroft algorithm, this would result in the states incident to these edges being marked as distinguishable when either l_{i_p} or l_{j_p} is chosen as part of the splitter, and hence they would not be merged either in the minimized DFA nor in the resulting minimized RRG. □

Example 3.6.6. *We provide an example of S.P.T. minimization, using the S.P.T.* Π'_b *whose*

transduction rules are given in Fig. 3.12. Fig. 3.13 shows the DFA that is initially constructed by Steps (1) to (3) of `GenerateDFAFromRRG`*. To improve readability of edge labels, a single letter is assigned to each distinct label.*

Queries ϕ_{b4}, ϕ_{b5}, and ϕ_{b6} all contain a redundant clause mandating that the value of the name attribute must not equal `Carol`*. The cause of the redundancy is that this clause was specified in the ancestor query ϕ_{b1} defining the contents of $Reg_{patient}$. We show below the expansion, minimization, and contraction steps for ϕ_{b4} as performed by Alg. 3.6.3; the steps are similar for queries ϕ_{b5} and ϕ_{b6}.*

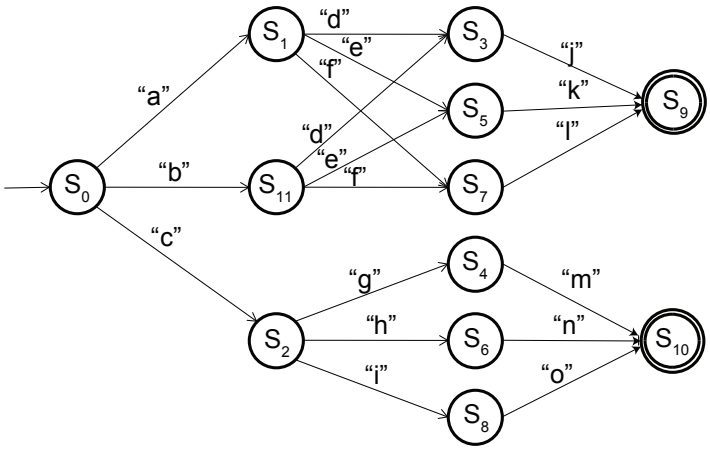
$$\begin{aligned}
\phi_{b4}(s) &= \exists n, a Reg_{patient}(s, n, a) \wedge n \neq \text{`}Carol' \\
&\Rightarrow \exists n, a(\texttt{Patient}(s, n, a) \wedge n \neq \text{`}Carol') \wedge n \neq \text{`}Carol' \quad (Expansion) \\
&\Rightarrow \exists n, a(\texttt{Patient}(s, n, a) \wedge n \neq \text{`}Carol' \quad (Minimization) \\
&\Rightarrow \exists n, a Reg_{patient}(s, n, a) \quad (Contraction)
\end{aligned}$$

A similar redundancy is found for queries ϕ_{b7}, ϕ_{b8}, and ϕ_{b9}, with the clause `n = ''Carol''` *already present within the defining query for the $Reg_{patient}$ local node register. Furthermore, the minimized form of ϕ_{b4} is equivalent to the minimization of ϕ_{b7}, and similarly $\phi_{b5} \equiv \phi_{b8}$ and $\phi_{b6} \equiv \phi_{b9}$. Lastly, $\phi_{b3} \equiv \phi_{b1}$. The updated DFA featuring the reduced label set is depicted in Fig. 3.14. Note that while Φ_{min} has four fewer queries than the original Φ, the label set has in fact only been reduced in size by one, as the tag name and access bitstring values continue to distinguish each of the remaining labels.*

Hopcroft's algorithm is then carried out on this label-reduced DFA. Assuming that the pair $\langle Q_D - F_D, \text{“}a''\rangle$ is chosen as the first splitter, $\{s_1, s_{11}\}$ is removed from the original set consisting of all non-final states and added as a separate set in the partition. This is so because the inverse transition function for both of these states leads back to s_0, while for the remaining states, there is no defined transduction rule for “a”, so each state maps to itself. This procedure is repeated for the remaining symbols in Σ_D, until the final partition of states consists of $\{\{s_0\}, \{s_1, s_{11}\}, \{s_2\}, \{s_3\}, \{s_4\}, \{s_5\}, \{s_6\}, \{s_7\}, \{s_8\}, \{s_9\}, \{s_{10}\}\}$. As a result, the only state minimization that results is that s_1 and s_{11} will be merged into a single state. Finally, the minimized DFA is converted back into an RRG, leading to the RRG depicted in Fig. 3.15. One can easily verify that this RRG is isomorphic to the one in

$$
\begin{aligned}
r_0 : (q_0, visits, &111110001000) \rightarrow \\
&(q_1, patient, 111110000000, \phi_{b1}(s,n,a;\emptyset)), \\
&(q_1, patient, 111110001000, \phi_{b2}(s,n,a;\emptyset)), \\
&(q_2, patient, 111110000000, \phi_{b3}(s,n,a;\emptyset)), \\
&\text{where} \\
&\phi_{b1}(s,n,a) = \texttt{Patient}(s,n,a) \wedge n \neq \text{`}Carol' \\
&\phi_{b2}(s,n,a) = \texttt{Patient}(s,n,a) \wedge n = \text{`}Carol' \\
&\phi_{b3}(s,n,a) = \texttt{Patient}(s,n,a) \wedge n \neq \text{`}Carol' \\
r_1 : (q_1, patient, &111110000000) \rightarrow \\
&(q_1, @ssn, 111110000000, \phi_{b4}(s;\emptyset)), \\
&(q_1, name, 111110000000, \phi_{b5}(n;\emptyset)), \\
&(q_1, age, 111110000000, \phi_{b6}(a;\emptyset)), \\
&\text{where} \\
&\phi_{b4}(s) = \exists n, a\, Reg_{patient}(s,n,a) \wedge n \neq \text{`}Carol' \\
&\phi_{b5}(n) = \exists s, a\, Reg_{patient}(s,n,a) \wedge n \neq \text{`}Carol' \\
&\phi_{b6}(a) = \exists s, n\, Reg_{patient}(s,n,a) \wedge n \neq \text{`}Carol' \\
r_2 : (q_1, patient, &111110001000) \rightarrow (q_1, @ssn, 111110001000, \phi_{b7}(s;\emptyset)), \\
&(q_1, name, 111110001000, \phi_{b8}(n;\emptyset)), \\
&(q_1, age, 111110001000, \phi_{b9}(a;\emptyset)) \\
&\text{where} \\
&\phi_{b7}(s) = \exists n, a\, Reg_{patient}(s,n,a) \wedge n = \text{`}Carol' \\
&\phi_{b8}(n) = \exists s, a\, Reg_{patient}(s,n,a) \wedge n = \text{`}Carol' \\
&\phi_{b9}(a) = \exists s, n\, Reg_{patient}(s,n,a) \wedge n = \text{`}Carol' \\
r_3 : (q_1, @ssn, &111110000000) \rightarrow (q_1, text, 111110000000, \phi_{b10}(s;\emptyset)), \\
&\text{where } \phi_{b10}(s) = Reg_{@ssn}(s) \\
r_4 : (q_1, @ssn, &111110001000) \rightarrow (q_1, text, 111110001000, \phi_{b10}(s;\emptyset)) \\
r_5 : (q_1, name, &111110000000) \rightarrow (q_1, text, 111110000000, \phi_{b11}(n;\emptyset)), \\
&\text{where } \phi_{b11}(n) = Reg_{name}(n) \\
r_6 : (q_1, name, &111110001000) \rightarrow (q_1, text, 111110001000, \phi_{b11}(n;\emptyset)) \\
r_7 : (q_1, age, &111110000000) \rightarrow (q_1, text, 111110000000, \phi_{b12}(a;\emptyset)), \\
&\text{where } \phi_{b12}(a) = Reg_{age}(a) \\
r_8 : (q_1, age, &111110001000) \rightarrow (q_1, text, 111110001000, \phi_{b12}(a;\emptyset)) \\
r_9 : (q_1, text, &111110000000) \rightarrow . \quad \text{/* empty string */} \\
r_{10} : (q_1, text, &111110001000) \rightarrow . \quad \text{/* empty string */} \\
r_{11} : (q_2, patient, &111110000000) \rightarrow \\
&(q_1, @ssn, 111110000000, \phi_{b4}(s;\emptyset)), \\
&(q_1, name, 111110000000, \phi_{b5}(n;\emptyset)), \\
&(q_1, age, 111110000000, \phi_{b6}(a;\emptyset))
\end{aligned}
$$

Figure 3.12: Transduction rules for the example secure publishing transducer Π'_b.



Legend:

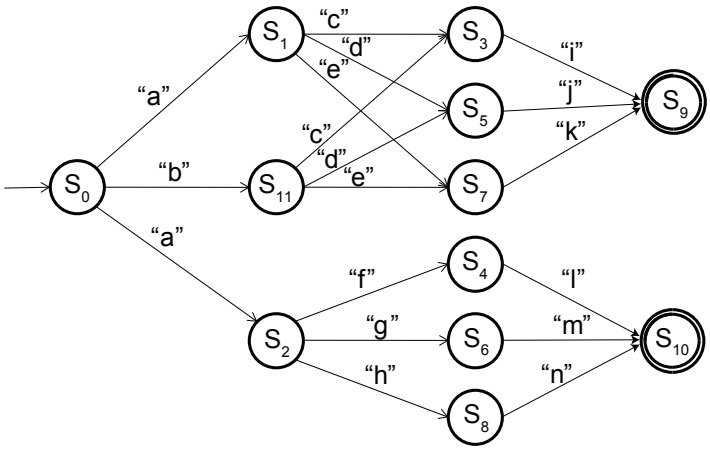
"a" : <1, patient, 111110000000, φ_{b1}>
"b" : <2, patient, 111110001000, φ_{b2}>
"c" : <3, patient, 111110000000, φ_{b3}>
"d" : <1, @ssn, 111110000000, φ_{b4}>
"e" : <2, name, 111110000000, φ_{b5}>
"f" : <3, age, 111110000000, φ_{b6}>
"g" : <1, @ssn, 111110001000, φ_{b4}>
"h" : <2, name, 111110001000, φ_{b5}>
"i" : <3, age, 111110001000, φ_{b6}>
"j" : <1, text, 111110000000, φ_{b10}>
"k" : <1, text, 111110000000, φ_{b11}>
"l" : <1, text, 111110000000, φ_{b12}>
"m" : <1, text, 111110001000, φ_{b10}>
"n" : <1, text, 111110001000, φ_{b11}>
"o" : <1, text, 111110001000, φ_{b12}>

Figure 3.13: DFA $DFA_{\Pi'_b}$ formed from S.P.T. of Fig. 3.12.

Fig. 3.8 up to state relabelling, indicating that in fact the S.P.T.s Π' *and* Π'_b *are (weakly) equivalent.* □

3.6.1 Complexity of S.P.T. Minimization

We now examine the computational complexity of the minimization problem for various classes of secure publishing transducers. Table 3.3 provides a summary of the results. In interpreting these results, note that a given S.P.T. inherits the query class of the *expanded* form of its most sophisticated query. By $|\Phi|$ we denote the number of distinct relational queries appearing within the transduction rule set, while by $|\phi_{max}|$ we indicate the number of clauses (subgoals) appearing in the largest query in Φ. As mentioned previously, this problem is closely related to static verification of S.P.T.s and in particular, proves un-



Legend:

“a” : <1, patient, 111110000000, φ_{b1}>
“b” : <2, patient, 111110001000, φ_{b2}>
“c” : <1, @ssn, 111110000000, φ_{b4}>
“d” : <2, name, 111110000000, φ_{b5}>
“e” : <3, age, 111110000000, φ_{b6}>
“f” : <1, @ssn, 111110001000, φ_{b4}>
“g” : <2, name, 111110001000, φ_{b5}>
“h” : <3, age, 111110001000, φ_{b6}>
“i” : <1, text, 111110000000, φ_{b10}>
“j” : <1, text, 111110000000, φ_{b11}>
“k” : <1, text, 111110000000, φ_{b12}>
“l” : <1, text, 111110001000, φ_{b10}>
“m” : <1, text, 111110001000, φ_{b11}>
“n” : <1, text, 111110001000, φ_{b12}>

Figure 3.14: DFA $DFA_{\Pi'_b}$ after completion of `QueryReduction` subprocedure.

decidable for all classes of recursive S.P.T.s. When non-recursive S.P.T.s are considered, the situation becomes somewhat better: for the full class of conjunctive queries including inequalities, the cost of determining query equivalence dominates the overall complexity, leading to Π_2^P-completeness [116]; when inequalities are excluded, query equivalence once again constitutes the largest cost leading to **NP**-completeness [28]. Note that for both cases, the complexity is one level lower in the polynomial hierarchy than the corresponding static verification decision problem, due to our substitution of a weaker notion of equivalence. For various subclasses of conjunctive queries, S.P.T. minimization can be performed in polynomial time.

For *acyclic conjunctive queries* (AQ) (cf. Sec. 2.2.3) (i.e., those for which a hypergraph representation – in which nodes are query variables and hyperedges correspond to subgoals – is acyclic), the running time can be dominated either by performing Hopcroft's algorithm

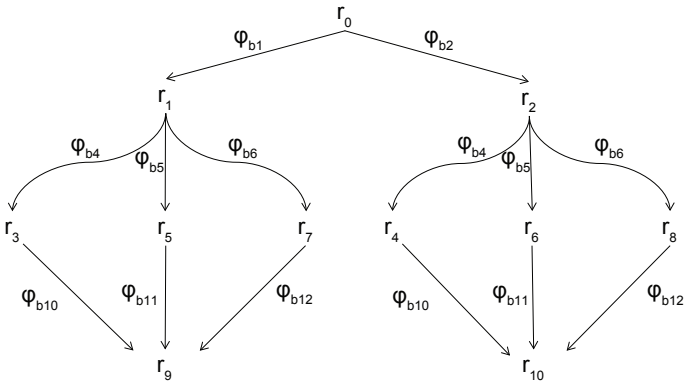


Figure 3.15: RRG for minimized S.P.T. Π'_b derived from minimized DFA $DFA_{\Pi'}$.

Fragment	**S.P.T. Minimization**
SPT$(\mathcal{L}, \mathcal{S}, \mathcal{A})$	undecidable
SPT$_{nr}$(FO$, \mathcal{S}, \mathcal{A})$	undecidable
SPT$_{nr}$(CQ$, \mathcal{S}, \mathcal{A})$	Π_2^P-complete
SPT$_{nr}$(CQE$, \mathcal{S}, \mathcal{A})$	**NP**-complete
SPT$_{nr}$(AQ$, \mathcal{S}, \mathcal{A})$	$O(\lvert\Phi\rvert \cdot \lvert\delta'\rvert^3 \log_2 \lvert\delta'\rvert + \lvert\Phi\rvert \cdot \lvert\phi_{max}\rvert^3 \log_2 \lvert\phi_{max}\rvert)$
SPT$_{nr}$(NSJ$, \mathcal{S}, \mathcal{A})$	$O(\lvert\Phi\rvert \cdot \lvert\delta'\rvert^3 \log_2 \lvert\delta'\rvert)$

Table 3.3: Complexity of minimization procedure for various classes of secure publishing transducers.

(represented by the first term), or by carrying out query minimization according to the algorithm in [29] (the second term), depending on the relative sizes of queries and the number of rules in the transduction rule set of the instance under consideration. The first term is obtained by first recalling the established $O(|\Sigma_D| \cdot n \log n)$ time complexity result for Hopcroft's algorithm – where Σ_D denotes the size of the automaton's alphabet, and n is the number of automaton states – followed by the observation that by the construction of our minimization algorithm, $\Sigma_D \subseteq \Phi \times \Sigma_T \times 2^{|F|}$. A tighter bound on the latter result is possible by observing that even if each of the $|\delta'|$ transduction rules features a distinct alphabet symbol and a distinct bitstring value at its head, then $|\Sigma_D| \leq |\Phi| \cdot |\delta'|^2$. This leads to a $O(|\Phi| \cdot |\delta'|^2 \cdot |\delta'| \log_2 |\delta'|)$ time complexity, which simplifies to the first term. Under the assumption that the set of minimized queries Φ_{min} is maintained as a hash table allowing average cost $O(1)$ insertions and lookups, the second term is a straightforward result of the requirement to perform $|\Phi|$ iterations of the `QueryReduction` procedure,

with each iteration being dominated by the $O(|\phi_{max}| \log_2 |\phi_{max}|)$ cost of performing query minimization using the algorithm in [29] during the second step of the procedure.

In the case of *conjunctive queries with no self-joins* (NSJ) (cf. Sec. 2.2.3), the situation is better still. Since by definition no two subgoals within the query operate over the same relation, the query cannot be made smaller. Hence, the running time is always dominated by the cost of applying Hopcroft's algorithm to merge equivalent states in the induced DFA.

3.7 Experimental Results

In this section, we present the results of an experiment in which the access control policy translation algorithm (Alg. 3.3.1) is implemented, and the time requirements for policy translation are measured as the sizes of the federated ID set, number of relational access control policy rules, and total number of clauses appearing in publishing transducer rules are varied.

The experiment was carried out in three phases, using synthetic inputs. In each phase, one parameter – the size of the set of federated IDs; the number of relational access control policy rules; or the total number of clauses appearing on the right-hand side of transduction rules – was varied from a minimum value of 10 up to a maximum value of 10000, while the remaining two parameters were fixed at the maximum value of 10000. The results of this experiment are summarized in Fig. 3.16. The displayed results represent an average of 5 trials conducted for each combination of the three inputs. In each case, the results are consistent with the expected linear behaviour (cf. Sec. 3.3) as the value of the parameter being varied is increased.

3.8 Related Work

As discussed in Sec. 2.4.4, several access control models and access control expression languages have been proposed for XML. While this chapter focused on expressing translated policies in XACML, it would be possible to define algorithms for transforming the transduction rule set of a S.P.T. into an alternate expression language, or into policy rules conforming to the syntax of an XML-specific access control language.

Research into access control specification and enforcement in data federations extends back into the late 1980s. Such work focused on practical protocols and implementations allowing for varying degrees of authorization autonomy. Sheth and Larson [112] distinguish between *tightly-* and *loosely-coupled* federations. The former are characterized by greater transparency, as federated users are able to directly query the schema of a data source. Such systems are more flexible in the sense that data sources can join and leave the federation with little impact on the remaining participants, at the expense of making querying more complicated: a federated user must first determine which source(s) contain relevant data, and then formulate a separate query over the exported schema for each such source. Tightly-coupled systems perform a conversion from local schemas to a single federated schema, allowing federated users to retrieve data with a single query. The need to define a schema translation for each data source makes extending the federation to accommodate new sources a complex process.

Jonscher and Dittrich [70] consider various approaches for implementing access control within tightly-coupled federations, and weigh the impacts on data source autonomy associated with each presented approach. In contrast to our approach, they focus on specification of access control policies at the federated level, and consider how to enforce such policies using access control enforcement mechanisms at the data sources. Idris et al [66] consider the problem of data overclassification in loosely-coupled federated systems, which results in federated users being denied legitimate access to data due to an ACP integration policy that classifies data based on the most restrictive policy defined locally by a data source. They suggest a scheme based on secret sharing that allows local administators to grant access dynamically for specific time periods. De Capitani di Vimercati and Samarati [44] propose an access control model for tightly-coupled federations allowing each local administrator to decide what data objects to export, and to define which federated users can access each object, while allowing the federation administrator to decide what data offered by data sources are accepted into the federation. They also investigate different enforcement mechanisms, such as performing access checks locally at the data sources versus globally at the federated level, and weigh the implications of each on data source autonomy. These works all predate the adoption of XML as a data exchange standard, and consider a narrower def-

inition of data federations. Additionally, they do not engage in a formal analysis of access control policy translation and enforcement within federations.

In recent years, there has been growing attention paid to composing unified access control policies from several input policies [18, 23, 94]. The emphasis of these works is on defining algebraic models for policy composition, rather than on examining the intrinsic difficulty of translating access control policies in a federated setting.

3.9 Conclusions

In this chapter, we studied issues related to integrating existing access control policies defined over source databases within larger data federations. We first investigated issues related to mapping the local IDs originating at each data source into a set of federated IDs, pointing out that a careless mapping strategy can easily lead to violations of the principle of least privilege.

Furthermore, we provided an algorithm that allows the translation of relational access control policies to be carried out automatically, and also examined the difficulty of verifying that an existing translation obeys the original access control policy, and of determining the smallest representation of a translated policy that maintains the original access control semantics.

While properly defined and implemented access controls provide an effective way of restricting *direct* accesses to data, additional methods are required to guard against *indirect* accesses caused by data leakage. This forms the topic of the next three chapters.

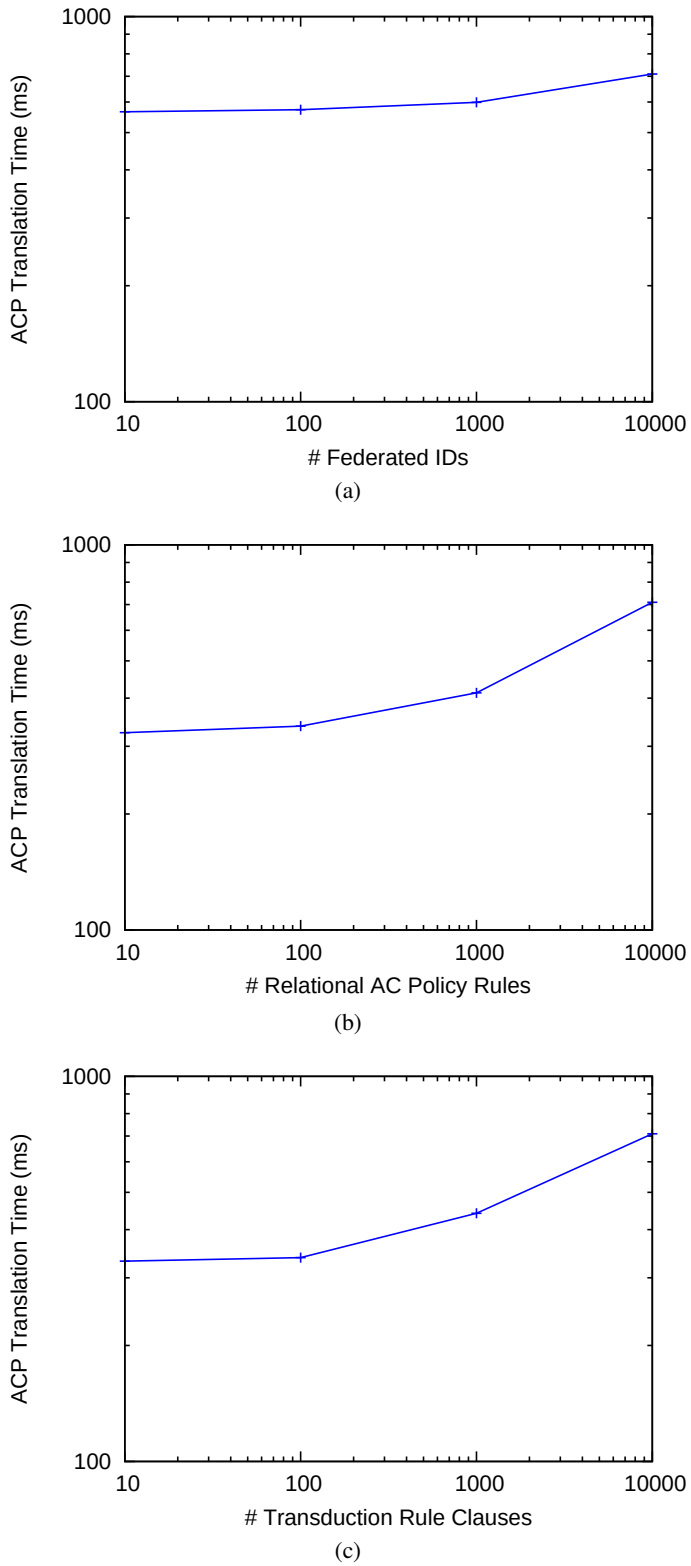


Figure 3.16: Results of varying (a) the number of federated IDs; (b) the number of relational access control policy rules; and (c) the number of clauses appearing in transduction rules on the time required to complete policy translation.

Chapter 4

Detecting and Removing Schema-Based Disclosure Risks in Federated Data

This chapter initiates our study into the identification and prevention of unintentional information disclosures resulting from answering queries against a federated database, represented as an XML database (tree). We begin with a discussion of the application setting and attack model, considering the various types of potential threats and when they can be most effectively detected and removed. This leads to a classification of disclosure risks against XML trees into *schema-based* and *instance-based* categories. Next, we focus on related approaches, and indicate which types of disclosure risks each was designed to address. The chapter ends by presenting our strategy for detecting and removing schema-based disclosure risks, deferring a similar treatment of instance-based disclosure risks until the next chapter.

4.1 Introduction

We assume an interactive setting in which users issue queries against a database (tree) T, conforming to a DTD $\mathcal{D}$. An access control policy (ACP) defines permissions over T for a set of federated IDs F; each user may be assigned to one or more IDs in F, and inherits the union of permissions associated with each such ID. An XML query q issued by user u is *allowable* if its formulation contains only nodes from T which are accessible to at least one ID held by u, and, similarly, its answer must consist of nodes that are accessible to at

least one ID held by u.

An *unintentional disclosure* occurs when an adversarial user is able to exploit the answers of allowable queries to guess with "sufficiently high" accuracy the content of one or more inaccessible nodes. Such a disclosure can be *total*, in which case the adversary becomes completely certain as to the values of one or more hidden nodes, or *partial*, in which case the adversary's uncertainty about the unknown content drops below a threshold ϵ.

Example 4.1.1. *The database in Fig. 2.1 contains information about patients discharged from a hospital. Assume that the access control policy of Ex. 2.4.12 is applied, granting various insurance companies access to anonymized records (withholding patient names), for purposes of actuarial computation. Further, suppose that Alice applies for a job with ACME Insurance but does not wish the company to learn of her medical history. User Mallory, ACME's interviewer, knows Alice's age and finds a post on her blog referring to her recent hospitalization. Observe that Mallory can now infer Alice's condition with, 50% accuracy, by issuing the query:*

```
/hospital/patient[age="31"]/diagnosis
```

The information disclosure is characterized as follows. From the original anonymized database, Mallory had full knowledge of the content of every node except for $o4, o17$ *and* $o28$*. That is, Mallory's initial beliefs with respect to the three unknown nodes were:* $P(o4 \rightarrow \textit{Alice}) = P(o17 \rightarrow \textit{Alice}) = P(o28 \rightarrow \textit{Alice}) = p$*, where* p *is some marginal probability that Alice had been treated at that hospital. Once Mallory becomes aware of Alice's age and her recent hospitalization, she can refine her model as follows:* $P(o4 \rightarrow \textit{Alice}) = P(o28 \rightarrow \textit{Alice}) = 1/2$ *and* $P(o17 \rightarrow \textit{Alice}) = 0$*, characterizing the* partial disclosure*; if Mallory had further knowledge that Alice was treated by Dr. House, a* total disclosure *of node* $o4$ *would have occurred.* □

Our solution. We assume the query execution model depicted in Fig. 4.1. Each query submitted to the DBMS is validated twice before the user sees any results. First, the query is intercepted by the *access control verification module*, which consults the access control policy defined for $\mathcal{D}$ to determine if the query issuer has access permissions for every node

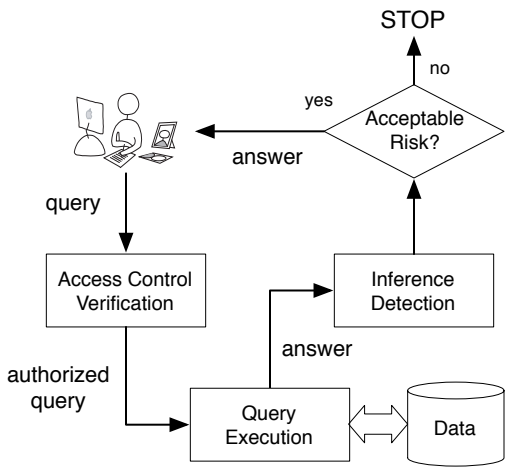


Figure 4.1: The query execution process.

specified in the formulation of the query, and in the answer to the query. If not, then the query is rejected. Otherwise, the query is executed to obtain a set of query results. Then, the results of the query are sent to the *inference detection module* which determines the query's unintentional disclosure risk. If such risk is higher than a specified threshold, the answer is withheld from the user.

Evaluating disclosure risks. We introduce a new measure for precisely evaluating the magnitude of disclosure risks, based on comparing the average amount of effort required by an adversary to guess the content of an inaccessible node after witnessing the nodes in his view of the database, versus the *a priori* effort of doing so without any knowledge of the database contents. This addresses an attack model in which the adversary *refines* their probabilistic model of the database in response to query answers. This measure is able to efficiently discover both partial and total disclosures.

4.2 Attack Model

We assume that the adversary models an XML database as a probabilistic tree, and further, has prior knowledge of the active domains of each location path in the XML database (including those location paths which are non-accessible to her). However, she has no inclination as to the probability with which each such value occurs[1]. By obtaining answers to al-

[1]Although, as will be shown in Sec. 5.2, calculations in our model can be extended to handle instances in which the adversary *does* possess a prior belief as to value probabilities.

lowable queries, the adversary successively refines their probabilistic model of the database by eliminating all database states which were previously possible, but do not agree with one or more query answers. In addition, the adversary has access to a DTD allowing her to issue valid queries over the database. We assume this DTD contains the rules relevant to location paths which are fully- and partially-accessible to the adversary, while rules pertaining to non-accessible location paths have been "pruned" from the DTD before it is supplied to the adversary (this pruning operation is performed during the design-time detection phase described later in Sec. 4.7). Without loss of generality, we assume the adversary is "working alone", that is, no collusion is taking place with additional users. However, we note that our model can be extended to such situations by considering the union of permissions assigned to each participating federated ID in the suspected collusion, rather than limiting the analysis to the IDs assigned to an individual adversary.

4.3 Design-time vs. Run-time Detection

Broadly speaking, detection of disclosure risks can take place either at *design-time* (i.e., when only the DTD is known) or at *run-time* (i.e., detection is carried out on a per-query basis upon the current database contents). Design-time detection offers multiple advantages: firstly, it is less detrimental to database performance in the sense that more of the computational effort can be done "offline", instead of requiring this work to be done during query processing; secondly, it facilitates detection of disclosure threats at the schema level, allowing these results to be applied to *any* database that conforms to a specific DTD. In applications involving collections of structurally similar XML databases, this feature is especially beneficial.

Unfortunately, it is not always possible to obtain a precise characterization of all disclosure risks posed to an individual database at design-time; instead, the results of design-time analysis often must be complemented by a run-time analysis. Ex. 4.1.1 provides an illustration of this fact, as the precise degree of the presented disclosure risk is dependent on the *current* database contents: if the database in this example contained only one patient record with an age value of 31, then it is clear that a *total* disclosure would have taken place; conversely, if all three patient records in the database possessed this age value, then *no* dis-

	Schema-based Disclosure Risks	Instance-based Disclosure Risks
Design-time Analysis	Fan et al [50] Farkas et al [52] Yang & Li [119]	N/A
Query-time Analysis	N/A	Hashimoto et al [62]

Table 4.1: Classification of existing approaches to disclosure control for XML data.

closure would have taken place since Mallory would not have been able to exclude any of the patient records from belonging to Alice based on the query answer. Clearly, in most circumstances it proves difficult to accurately anticipate the future contents of a database at design-time.

We distinguish between two types of disclosures in XML databases. The first type, *schema-based* disclosures, represent a threat to all databases which conform to a specific DTD. Such risks can be efficiently detected and removed at design-time. Members of the second type are referred to as *instance-based* disclosures, and may only exist when an individual database occupies particular states. While the *potential* for such risks to exist is detectable at design-time, a precise characterization of the degree of risk depends on the current database contents and therefore can only be determined at run-time. More specifically, we note that an instance-based disclosure can only be present when a federated ID f has partial or no access to a *sensitive* node s (i.e., $path(s) \in \ominus_{\text{read}}(f, AC, \mathcal{D}) \cup \odot_{\text{read}}(f, AC, \mathcal{D})$). Any nodes belonging to one of f's fully-accessible node classes are, by definition, already known to all holders of f. Such conditions can be detected via a static analysis of the DTD and the ACP, without requiring knowledge of the actual contents of the database (yet a precise determination of the risk magnitude *does* depend on the current database state, and therefore must be deferred until query-time).

4.4 Related Work

Having elaborated upon the nature of the problem, we are now in a position to compare and contrast related approaches.

4.4.1 Disclosure Control and Access Control Models for XML Data

To date, little attention has been paid to addressing the information disclosure risks for XML databases. The work most relevant to ours is that of Hashimoto et al [62], which presents a query-time framework for preventing disclosures in XML databases based closely on the concept of k-anonymity [105]. In their model, an adversary issues a set of allowable queries, and uses the results of each such query to refine their belief about the result of a *secret query*, which contains one or more inaccessible nodes. Their approach models each query/query result pair as a tree transducer; the contents of this transducer indicate the possible database states that match the query and query result. By computing the intersection of each of these transducers, one obtains a transducer that reflects the candidate database states matching the full set of issued allowable queries. The third step augments this transducer by incorporating the secret query, such that the resulting transducer encodes all database states that (1) satisfy each of the allowable queries, and (2) provide an answer to the secret query. By analyzing this final transducer, a determination is made as to whether there are at least k candidate database trees (for a defined threshold parameter value k), and declares that a disclosure risk is present only if there are fewer than k such trees.

The accompanying experimental results indicate that this approach exhibits poor space scalability, mainly due to the expense of materializing tree transducers for each query. This cost is greatest in the beginning, when many database states that ultimately prove to be irrelevant cannot yet be disqualified. This greatly inflates the sizes of the transducers used to model the issued allowable queries.

Other approaches [119, 52] restrict their analysis to inferences caused by an adversary's prior knowledge of constraints between XML element types. Yang and Li [119] considered the case of preventing the accidental release of sensitive information in a published XML document. In their model, the adversary is able to perform data inferences based on prior knowledge of semantic constraints between elements in the document. They consider three types of such constraints: *parent-child* (in which the constraint specifies that each node of a specified type a must have a child node of type b); *ancestor-descendant*, which indicates that each node of a specified type a must have a child node of type b at some position within its

subtree; and *functional dependencies*, which specify that each node of a specified type a that additionally possesses a child path p_1 within its subtree must also possess an additional child path p_2. Their key contribution is an algorithm utilizing a chase procedure for determining the largest document subset which is free of disclosure risks formed from the above three types of semantic constraints. Their approach requires the database designer to specify the semantic associations between elements; since such associations are often difficult to identify at database design time, and often can be formed independently by an attacker who applies real-world knowledge "outside" the database, the utility of their approach is limited. Furthermore, their approach cannot identify instances of disclosure which are not caused by semantic association, such as Ex. 4.1.1. Farkas et al [52] presented a solution for preventing inferences in semi-structured databases. They consider a broader class of semantic constraints, expressible as Horn clauses, and use a chase procedure to remove undesired inferences caused by these constraints. Similar to the Yang and Li approach, their solution requires semantic constraints to be supplied to the inference removal algorithm. For this reason, the approach suffers from the same shortcomings; in particular, both approaches are unable to detect partial disclosures, and rely heavily on the security administrator's ability to accurately model the adversary's prior knowledge of inter-element dependencies.

Table 4.1 classifies each of these approaches according to whether they perform analysis at the design- or query-time phases, and whether a given approach is capable of detecting schema- and/or instance-based disclosures. As the table shows, none of these prior approaches is able to detect both types of disclosures, and the only prior approach capable of detecting instance-based disclosures performs all processing at query-time, greatly hampering its scalability. Our approach is the first to be able to detect both risk types, and the first to separate disclosure risk detection into design- and query-time phases.

Several XML-specific access control models have been proposed (e.g., [14, 13, 20, 31, 39, 50, 75, 93, 88]), which allow one to limit direct access to portions of an XML document. However, these approaches do not address the potential security risks posed by an adversary exploiting covert channels such as inference to indirectly access restricted data. Fan et al [50] describe an XML access control model in which an access control policy defined over a DTD is used to generate a set of user-specific *security views*, each consisting

of a sanitized DTD and a query re-writing function allowing user queries to be efficiently translated into an equivalent query over the user-accessible portions of the original XML database. This run-time approach avoids the expense of materializing and maintaining user-specific views. While their DTD sanitization process does remove any DTD rules pertaining to totally inaccessible node classes, it does not address inference risks caused by partially-accessible node classes (more specifically, it will not delete a DTD rule describing a node class c as long as even one instance of c is accessible to the user). Additionally, it does not guard against collusion attacks.

Wang and Lakshmanan [117] consider secure query evaluation for XML databases in the "database-as-a-service" scenario, where an untrusted server is used to store data and service client queries over that data. In their solution, client queries are encrypted before being sent to the server; the server uses associated metadata along with the contents of the encrypted database to answer queries, and returns responses to clients, at which point they are decrypted. The encryption scheme employed, together with the introduction of spurious records, ensures that the server cannot guess with sufficiently high accuracy the true contents of the database or queries.

4.4.2 Disclosure Control for Statistical and Relational Databases

Information disclosure risks relating to relational and statistical databases have been well-studied since the early 1980s. Early works focused on detecting and removing inference channels from statistical databases maintained by the U.S. Census Bureau. In this setting, the goal was to prevent a user from discovering statistical data associated with an individual by issuing aggregation queries returning statistics over a population. Proposed inference controls for statistical databases can be classified according to whether they restrict queries and query answers which may leak individual data, or alter query answers before returning them to the user. Denning and Schlörer [43] provide a survey of these approaches.

A considerable body of work has focused on disclosure prevention in multi-level relational databases [22, 85, 90]. In this scenario, an inference channel consists of a set of associations between database objects which allows a user to infer the value of classified data. Such channels are typically formed by an adversary combining their knowledge of

integrity constraints of the database (e.g., functional dependencies) with non-sensitive data. Solutions typically involve increasing the classifications assigned to certain database objects in order to break the inference channels. A survey of such approaches is provided in [53].

4.4.3 Disclosure Control for Exchanged Data

In these early approaches, the focus has been on limiting the ability of the adversary to infer secrets by directly querying the database. Beginning in the late 1990s, growing attention has been paid to preventing information disclosures in data sets that have been published *outside* the database. This shift has largely been driven by the dramatic increase in data availability from sources such as the World Wide Web, and also by the rising prominence of applications in which data is exchanged between parties. One branch of this research is *privacy-preserving data mining*, which aims to preserve the utility of data mining techniques for extracting patterns from large data sets while preventing the disclosure of sensitive information. Aggarwal and Yu [3] provide a survey of privacy-preserving data mining techniques.

Another active area of research is *secure data publishing* [83, 105, 118], where anonymized database records are released and the goal is to prevent an adversary from being able to associate records with individuals through the application of additional prior knowledge. However, it recently has been shown that these approaches, and any others that assume that the adversary operates based on either a "random worlds" model, an assumption of an independent and identically-distributed distribution of data values, or probabilistic independence between tuples, severely underestimate the computational power of a typical adversary and are open to a range of machine learning-based attacks [71]. A different branch of anonymization known as *differential privacy* [45] focuses on interactive settings in which user information is gathered by issuing queries against a statistical database. It essentially seeks to guarantee that an adversary is unable to distinguish an instance of the database containing a target individual's data from one that does not, with sufficiently high probability. While this approach has attractive theoretical properties, it tends to require a large degree of data distortion to enforce its privacy guarantee. To our knowledge, there

Table 4.2: Types of schema-based disclosure risks.

(1) Parent-Child	
DTD Rule	**Policy Rule(s)**
<!ELEMENT e_1 (e_2)>	$\langle f_i, \text{anc-or-self}(e_1), \texttt{read}, \texttt{allow}\rangle$ and $\langle f_i, \text{anc-or-self}(e_1)[c]/e_2, \texttt{read}, \texttt{deny}\rangle$ or $\langle f_i, e_1/e_2[c], \texttt{read}, \texttt{deny}\rangle$
(2) Required Attribute	
DTD Rule	**Policy Rule(s)**
<!ATTLIST e_1 a_1 CDATA #REQUIRED>	$\langle f_i, \text{anc-or-self}(e_1), \texttt{read}, \texttt{allow}\rangle$ and $\langle f_i, \text{anc-or-self}(e_1)[c]/@a_1, \texttt{read}, \texttt{deny}\rangle$ or $\langle f_i, e_1/@a_1[c], \texttt{read}, \texttt{deny}\rangle$
(3) Unique Attribute	
DTD Rule	**Policy Rule(s)**
<!ATTLIST e_1 a_1 ID>	$\langle f_i, \text{anc-or-self}(e_1), \texttt{read}, \texttt{allow}\rangle$ and $\langle f_i, \text{anc-or-self}(e_1)[c]/@a_1, \texttt{read}, \texttt{deny}\rangle$ or $\langle f_i, e_1/@a_1[c], \texttt{read}, \texttt{deny}\rangle$

have been no attempts to extend the differential privacy model to tree-based data formats such as XML.

Miklau and Suciu [89] conducted a theoretical study of the query-view security problem for relational data exchange scenarios. They proposed a standard for ensuring perfect security: namely, that a set of views $\bar{V}$ over a database D does not leak any information about the results of a secret query Q. This work was subsequently extended to define a relaxed security standard, in which arbitrarily small amounts of leakage are permissible [37]. Machanavajjhala and Gehrke [83] provide an alternative characterization of perfect query-view security in terms of query containment. Furthermore, they use previously established results on testing containment over various query classes to determine several classes of queries for which perfect security can be efficiently determined (i.e., using an algorithm running in polynomial time in the size of the secret query Q and the view definition query V).

4.5 Schema-Based Disclosure Risks

Certain types of disclosure risks can be detected at database design time, that is, even before the first node is inserted into the XML database. We refer to these as *schema-based disclosure risks*. All such risks share the following three properties: (1) they always cause

a *total* disclosure of the sensitive node s; (2) they are detectable through static analysis of the schema specification and access control policy, without necessitating knowledge of the database contents, and hence apply to *any* database conformant to that schema specification; and (3) they can be removed during the same static analysis process. As the expressiveness of various XML schema languages differs, so too does the amount of information they furnish to the adversary. In the case of DTDs, we identify the following three types of schema-based disclosure risks that are detectable at design time.

Parent-child disclosure. In this type of disclosure, a user holding federated ID f_i is able to exploit their knowledge of an accessible parent node with label e_1 to infer the value of a hidden child node labeled e_2, based on the DTD rule defining the content model for e_1. This scenario is summarized in the top row of Table 4.2. The DTD rule for e_1 specifies that there is a mandatory child node of type e_2. One cannot simply delete this parent-child relationship from the sanitized DTD given to f_i if there are other instances of e_2 which are accessible to f_i. Yet some solution is evidently needed, as the existing DTD rule allows f_i to associate an e_2 child with *every* occurrence of e_1.

Required attribute disclosure. In this situation, the DTD rule containing the definition of the attribute contains the `#REQUIRED` modifier, indicating that every node with label e_1 *must* contain an attribute with label a_1. Left unaltered, this DTD rule would allow f_i to recover hidden instances of the a_1 attribute, provided that the parent e_1 node is accessible. This type of risk is described in the second row of Table 4.2.

Attribute uniqueness disclosure. This type of risk takes place when the DTD rule for a_1 attributes contains the `ID` modifier. Armed with this information, f_i knows that each value in $adom(a_1)$ can only appear once in the database, allowing her to eliminate from consideration those values which she has observed in accessible instances of a_1. The third row of Table 4.2 illustrates this type of risk.

Note that in our model, *enumerated attribute* definitions (in which the DTD rule for an attribute specifies a finite list of possible values that attribute may take) and *fixed attribute* definitions (in which all occurrences of the attribute type take on the same value) do not provide the adversary with any additional information under our model's assumption that the adversary possesses prior knowledge of the active domains of each attribute in the database.

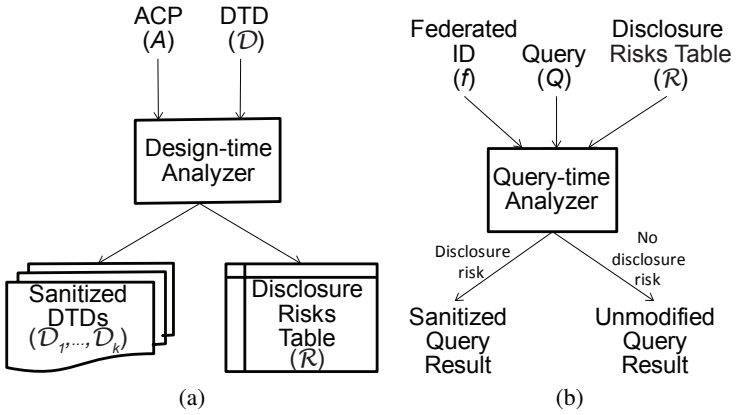


Figure 4.2: Schematic for (a) Design-time and (b) Query-time risk analysis.

Hence, neither constitutes a source of disclosure.

Example 4.5.1. *We show an example of a parent-child disclosure based on the example hospital DTD of Fig. 2.3 and ACP of Ex. 2.4.12. The adversary Mallory is an employee of ACM Insurance and therefore inherits the permissions associated with the* `AIns` *federated ID as specified by the ACP. Rule* `PR2` *results in her only having access to those* `/hospital/patient/name` *instances associated with clients of ACME Insurance, while rule* `PR1` *grants her access to* all *instances of* `/hospital/patient`*. This partial access prevents one from simply pruning the DTD rule for* `name` *elements, since Mallory requires it to properly formulate queries over those* `/hospital/patient/name` *instances she* does *have access to. However, the status quo clearly presents a parent-child disclosure risk, since Mallory can apply the DTD rule for* `patient` *elements to infer that* every *occurrence of* `/hospital/patient` *must have a child* `name` *element, allowing her to infer the existence of those* `/hospital/patient/name` *instances that are not accessible to her. Note that the first condition stated above is satisfied: the DTD rule essentially specifies a functional dependency indicating that every appearance of a* `/hospital/patient` *node implies the existence of a child* `name` *element, leading to a total disclosure. It will soon be demonstrated that the remaining two conditions also hold for this example.* □

4.6 Overview

Our approach to disclosure risk analysis is separated into *design-time* and *query-time* phases, as depicted in Fig. 4.2. As shown in Fig. 4.2a, the design-time phase takes as inputs an access control policy AC and DTD $\mathcal{D}$. The *design-time analyzer* processes these inputs and produces two outputs, a set of sanitized DTDs $\mathcal{D}_1, \ldots, \mathcal{D}_k$ and a *disclosure risks table* denoted as $\mathcal{R}$. The former contains a customized DTD for each federated ID in which all schema-based disclosure risks have been removed, while the latter stores, for each federated ID, the set of applicable partially- accessible and inaccessible location paths.

Fig. 4.2b summarizes the query-time analysis procedure. For each query q issued by a user holding federated ID f, the disclosure risk table $\mathcal{R}$ is consulted to determine whether there are any instance-based disclosure risks present within the context of q and f. If a possible risk is constituted, the *query-time analyzer* is responsible for altering the result of q in order to reduce the risk below an acceptable level; otherwise, the unchanged query result is returned to the user.

4.7 Design-time Detection via Static Analysis

Given an input DTD $\mathcal{D}$ and an ACP A, we wish to construct a *sanitized* DTD $\mathcal{D}_{f_i}$ for each federated ID f_i: a modified version of $\mathcal{D}$ in which any instances of the three types of schema-based disclosure risks described in Sec. 4.5 exploitable by f have been removed. The sanitization process consists of a combination of *pruning* and *DTD rule generalization* operations. The pruning operation deletes nodes from the original DTD corresponding to location paths which are non-accessible to f_i. Each DTD rule generalization alters a rule appearing in the original DTD to indicate that a *required* child element or attribute of type p is instead *optional*; this is done in cases where p is partially-accessible to f_i.

Definition 4.7.1 (Fully-Sanitized DTD)**.** *We say that a DTD $\mathcal{D}$ has been* fully-sanitized *with respect to federated ID f_i, denoted as $\mathcal{D}_{f_i}$, iff all schema-based disclosure risks exploitable by f_i have been removed via a combination of pruning and rule generalization operations.* □

Alg. 4.7.1 lists the procedure for design-time detection and removal of schema-based

Input: DTD graph DG; access control policy A defining access over DTD $\mathcal{D}$ to a set of active federated IDs $\{f_1, \ldots, f_k\} \subseteq F$

Output: A set of sanitized DTDs $\{\mathcal{D}_1, \ldots, \mathcal{D}_k\}$; disclosure risks table $\mathcal{R}$

```
1  R ← ∅;
2  foreach active ID f_i do
3      DG_i ← DG;
4      PR_i ← ApplicableRules(A, f_i);
5      foreach r ∈ PR_i do
6          AnnotateNode(r, DG_i);
7      end
8      foreach n ∈ DFS(DG_i) do
9          if n.isInaccessibleNode() then
10             DeleteNodeAndDescendents(n, DG_i);
11             newRiskEntry ← ⟨f_i, n, n.getCondition()⟩;
12             R.addEntry(newRiskEntry);
13         end
14         else if n.hasCondition() then
15             cond ← n.getCondition();
16             SplitNode(n, cond, DG_i);
17         end
18     end
19     D_i ← GenerateSanitizedDTD(DG_i, R);
20 end
21 return R, {D_1, ..., D_k};
```

Algorithm 4.7.1: Design-time analysis procedure.

disclosure risks, together with the identification of *potential* instance-based disclosure risks. The procedure takes as inputs a graph representation of the original DTD, DG, as well as an access control policy A specifying access permissions over DG for a set F of active federated IDs. Upon conclusion, it produces a set of fully-sanitized DTDs (one for each active federated ID) and a disclosure risks table $\mathcal{R}$ populated with entries describing all *potential* instance-based disclosure risks for each federated ID in F.

Each iteration of lines 2-20 produces a sanitized DTD for an individual active ID $f_i \in F$. Initially, the DTD graph DG_i is a copy of the original DTD graph DG (line 3); the subset of policy rules relevant to f_i are then selected (line 4), and each such rule is then examined in turn (lines 5-7) and is used to annotate each node of DG_i with a label indicating whether the corresponding location path is fully-, partially-, or non-accessible to f_i (line 6). The annotation is performed by the `AnnotateNode` subprocedure listed as Alg. 4.7.2. This subprocedure simply performs annotation of one or more nodes in the DTD graph, based

Input: Access control policy rule $pr = \langle R, e, a, p \rangle$; DTD graph DG

```
1  contextNodes ← EvaluateQueryOnDTDGraph(e, DG);
2  foreach n ∈ contextNodes do
3      if e.hasPredicate() then
4          n.setScope(PARTIAL);
5          n.setCondition(e.getPredicate());
6      else
7          n.setScope(FULL);
8          n.setCondition(∅);
9      end
10     n.setAction(a);
11     n.setPermission(p);
12 end
```

Algorithm 4.7.2: `AnnotateNode` subprocedure.

on the contents of the supplied policy rule pr. The query e belonging to pr is executed on the DTD graph DG_i in order to determine which graph node(s) are covered by e (line 1). Each such node n is then assigned an annotation based on the values of the action field, a, and permission field, p, of pr (lines 10-12). For cases where e contains a predicate, this annotation will also record it as the condition governing the partial access to node n for f_i (lines 4-5); otherwise, the annotation's scope is set to FULL (lines 7-8).

Once the annotation process has concluded, Alg. 4.7.1 continues execution and the annotated DG_i is traversed in preorder (lines 8-18); if the currently visited node n is not accessible to f_i, then n and its descendant nodes are deleted from DG_i, and an entry for n is added to the disclosure risks table $\mathcal{R}$ (lines 9-13) (our assumption of a monotonic access control model permits the entire subtree rooted at n to be removed at once, without visiting each descendant node first). If n is partially-accessible to f_i, the governing condition $cond$ is used to replace n and its subtree with two copies – one of which is annotated to allow (deny) access when $cond$ is satisfied, and the other to deny (allow) access when $cond$ is not satisfied (lines 14-17).

In the last step (line 19), the `GenerateSanitizedDTD` subprocedure is invoked to produce a sanitized DTD from the annotated DTD graph representation. This subprocedure is listed as Alg. 4.7.3, and proceeds as follows. Lines 2-22 cause the annotated DG_i to be traversed in preorder; each encountered node n is then tested to see whether it is annotated with partial (line 3) or full access. In the latter case, no sanitization is required. For nodes

<table>
<tr><th colspan="2">Parent-Child</th></tr>
<tr><th>Risk Pattern</th><th>Solution</th></tr>
<tr><td>e1 <full, read, allow, Ø>
↓
e2 <partial, read, deny, <i>cond</i>></td><td>e1 <full, read, allow, Ø>
↓ ?
e2 <partial, read, deny, <i>cond2</i>></td></tr>
<tr><td>e1 <partial, read, deny, <i>cond1</i>>
↓
e2 <partial, read, deny, <i>cond2</i>></td><td>e1 <partial, read, deny, <i>cond1</i>>
↓ ?
e2 <partial, read, deny, <i>cond2</i>></td></tr>
<tr><th colspan="2">Required Attribute</th></tr>
<tr><th>Risk Pattern</th><th>Solution</th></tr>
<tr><td>e1 <full, read, allow, Ø>
↓ #REQUIRED
a1 <partial, read, deny, <i>cond</i>></td><td>e1 <full, read, allow, Ø>
↓
a1 <partial, read, deny, <i>cond</i>></td></tr>
<tr><td>e1 <partial, read, deny, <i>cond1</i>>
↓ #REQUIRED
a1 <partial, read, deny, <i>cond2</i>></td><td>e1 <partial, read, deny, <i>cond1</i>>
↓
a1 <partial, read, deny, <i>cond2</i>></td></tr>
<tr><th colspan="2">Unique Attribute</th></tr>
<tr><th>Risk Pattern</th><th>Solution</th></tr>
<tr><td>e1 <full, read, allow, Ø>
↓ ID
a1 <partial, read, deny, <i>cond</i>></td><td>e1 <full, read, allow, Ø>
↓
a1 <partial, read, deny, <i>cond</i>></td></tr>
<tr><td>e1 <partial, read, deny, <i>cond1</i>>
↓ ID
a1 <partial, read, deny, <i>cond2</i>></td><td>e1 <partial, read, deny, <i>cond1</i>>
↓
a1 <partial, read, deny, <i>cond2</i>></td></tr>
</table>

Figure 4.3: Listing of annotated DTD graph patterns that correspond to various types of schema-based disclosure risks. The left-hand column lists a specific pattern, while the right-hand column shows the modification made to the DTD graph to remove the presented risk.

Input: Annotated DTD graph DG; federated ID f_i; disclosure risks table $\mathcal{R}$
Output: Set of sanitized DTD rules $DTDRules$

```
1  DTDRules ← ∅;
2  foreach n ∈ DFS(DG) do
3      if n.getScope() = PARTIAL then
4          p ← n.getParentNode();
5          if
           p.hasCondition() = false || p.getCondition() ≠ n.getCondition()
           then
6              if n.isAttributeNode() then
7                  if n.hasModifier(ID) then
8                      n.deleteModifier(ID);
9                  end
10                 if n.hasModifier(#REQUIRED) then
11                     n.deleteModifier(#REQUIRED);
12                 end
13             end
14             else if n.isElementNode() then
15                 n.addCardinalityConstraint(?);
16             end
17         end
18         newRiskEntry ← ⟨f_i, n, n.getCondition()⟩;
19         R.addEntry(newRiskEntry);
20     end
21     DTDRules ← DTDRules ∪ GenerateDTDRule(n);
22 end
23 return DTDRules;
```

Algorithm 4.7.3: `GenerateSanitizedDTD` subprocedure.

annotated with partial access, a reference is obtained to the parent node p of n (line 4). A test is then conducted to see whether p has no condition regulating its access (or equivalently, whether it is annotated with full access), in which case the first row corresponding to each risk type in Fig. 4.3 *may* be applicable. If this test fails, a second test is carried out to see whether the conditions regulating access to p and n fail to match, in which case the second row corresponding to each risk type may apply.

If either of these tests succeeds (line 5), then tests are done to determine whether n is an attribute node (line 6) or an element node (line 14). In the former case, the potential exists for either a required attribute risk or a unique attribute risk to be present. In the case of a unique attribute risk, the edge incident to n will be labelled with an `ID` modifier, and will be detected by the test on line 7. As indicated by the right-hand columns in Fig. 4.3,

the solution is to delete this modifier, which is done by line 8. Similarly, required attribute risks can be identified by the presence of a `#REQUIRED` modifier on the edge incident to n (tested by line 10), in which case each such modifier will be removed by line 11. Finally, if n is an element node, a parent-child risk is the only possible consideration. This test is conducted at line 14, at which point one of the two tests conducted at line 5 must have succeeded, meaning one of the scenarios presented in the left-hand column of rows one and two of Fig. 4.3 must hold, and there is indeed a parent-child risk. As indicated by the corresponding right-hand column cells of Fig. 4.3, in either case the solution is to simply add a `?` cardinality constraint to the edge incident to n. This is done by line 15.

The final consideration is to populate the disclosure risks table with a new entry for each node that is annotated with partial access. This is carried out by lines 18-19. Finally, after each node n has been processed and any relevant risks have been removed, a sanitized DTD rule describing the content model for n is created and added to the collection (line 21). In particular, nodes annotated with full access for f_i will produce a rule identical to the corresponding rule for that node type in the original DTD, while those nodes that have been linked to a schema-based disclosure risk will produce an altered rule in which the risk has been removed. Table 4.3 describes the precise format of the sanitized DTD rules corresponding to each risk type.

After all active federated IDs have been processed by Alg. 4.7.1, the complete set of sanitized DTDs $\{\mathcal{D}_1, \ldots, \mathcal{D}_k\}$ is returned, together with the populated $\mathcal{R}$ (line 19).

Complexity analysis. Alg. 4.7.1 has a worst case running time of $O(|F| \cdot (|DG|^2 + |A| \cdot |DG|))$, where $|F|$ denotes the number of distinct federated IDs, $|A|$ is the number of rules in the federated access control policy, and $|DG|$ is the number of nodes in the input DTD graph. We derive this bound as follows, first examining the time complexities of each subprocedure. The `AnnotateNode` subprocedure is dominated by the time required to determine which graph nodes are relevant to the query expressed within the input policy rule (line 1). This requires $O(|DG|)$ time as each graph node has to be considered in relation to the query. All other steps in `AnnotateNode` require $O(1)$ time, leading to an overall time complexity of $O|DG|)$. The `GenerateSanitizedDTD` subprocedure also features an

Type	Modified DTD Rule
Parent-Child	<!ELEMENT e_1 (e_2?)>
Required Attribute	<!ATTLIST e_1 a_1 CDATA #IMPLIED>
Unique Attribute	<!ATTLIST e_1 a_1 CDATA #IMPLIED>

Table 4.3: DTD rules altered to remove schema-based disclosure risks.

$O(|DG|)$ running time, since each node within the DTD graph is visited in DFS order, while the various tests, lookups, and modifications carried out on node properties during each individual visit each contribute an $O(1)$ time cost. We note that the remaining three subprocedures called by Alg. 4.7.1, `ApplicableRules`, `DeleteNodeAndDescendants`, and `SplitNode`, have respective time costs of $O(|A|)$ (since each policy rule in A must be tested to see if federated ID f_i appears within its subject list), $O(|1|)$ (under the assumption that the DTD graph implementation considers all descendents of a node n to be removed from the graph once the reference between n and its parent node has been deleted), and $O(|DG|)$ (since in the worst case, the node to be split is the DTD graph's root node, meaning a deep copy of every other node in the graph must be carried out).

Taking these results into account, the time complexity of Alg. 4.7.1 is obtained as follows. We note that $|F|$ iterations of the loop formed by lines 2-18 are carried out, and each such iteration is dominated by the respective costs contributed by the inner two loops (lines 5-7 and lines 8-16). The first loop has a time requirement of $O(|A|\cdot|DG|)$, while the second has an $O(|DG|^2)$ time cost, based on the need to perform a separate iteration for each node in the DTD graph and on the dominating $O(|DG|)$ time cost represented by the potential call to `SplitNode` within each iteration. As it is possible to have more or fewer policy rules than nodes in the DTD graph, neither $|A|$ nor $|DG|$ serves as a bound on the other value, leading to the claimed worst case time complexity.

Lemma 4.7.2. *On inputs DG and A, Alg. 4.7.1 produces a set of fully-sanitized DTDs $\{\mathcal{D}_1, \ldots, \mathcal{D}_k\}$.*

Proof. This requires one to establish that all schema-based disclosure risks actually present in the context of the input DTD and access control policy are successfully detected and removed by the algorithm. We focus on demonstrating that all parent-child disclosure risks are successfully detected and removed, as similar arguments can be used to verify the same

is true for the other two types of schema-based disclosure risks.

Demonstrating successful detection equates to establishing that, when presented with each of the scenarios listed in the corresponding row of Table 4.2 for parent-child disclosure risks, the DTD graph will be annotated to satisfy one of the two risk patterns identified in the left-hand column of Fig. 4.3. Next, we must demonstrate that the result of applying `GenerateSanitizedDTD` to either of these risk patterns leads to the corresponding solution pattern depicted within the right-hand column of Fig. 4.3. Doing so equates to demonstrating that all detected parent-child risks are in fact removed by Alg. 4.7.1.

We now consider the detection phase of parent-child risks. The characteristic structure of the DTD rules and access control policy rules constituting such a risk are listed in Table 4.2. Based on the existence of the DTD rule specifying that each occurrence of e_1 has a child element of type e_2, the constructed DTD graph DG will feature nodes n_1 and n_2 corresponding to these respective element types, and an edge from n_1 to n_2 will designate the parent-child relationship. Now consider the first access control policy rule listed in Table 4.2. It indicates that the federated ID f_i possesses full read access to e_1 or one of its ancestors. Based on this policy rule, n_1 will either be identified as a context node by line 1 of `AnnotateNode`, or n_1 will inherit the annotation of the ancestor node that serves as the context node. In either case, lines 6-11 of `AnnotateNode` serve to annotate n_1 with full read access for f_i. Now we consider the inclusion of the second listed policy rule, which overrides the first policy rule by specifying that f_i only has partial access to e_1 (and its descendents). In this case, `AnnotateNode` will identify n_1 (or one of its ancestors) as a context node (line 1) and associate node n_1 with an annotation indicating partial read access, along with the accompanying condition $cond$ governing the partial access (lines 3-5 and lines 10-11). This annotation will be propagated to n_2. Note that in this case, a disclosure risk is only present if there is another policy rule mandating conditional access to n_2 based on a different condition than $cond$ (otherwise, the only accessible instances of n_2 will correspond to the only accessible instances of n_1, namely those for which $cond$ is satisfied, leading to no disclosure). Such a case will lead to the second DTD graph risk pattern depicted in the left-hand column of Fig. 4.3, in which both n_1 and n_2 are annotated with partial access, yet with differing conditions.

Alternatively, if it is the third policy rule in Table 4.2 that is present, then only the annotation for n_2 is updated to reflect partial access, and the original annotation assigned to n_1 by the first policy rule, designating full access, remains in effect. This situation matches the first risk pattern depicted in the left-hand column of Fig. 4.3. Hence, all parent-child risks presented by an arbitrary DTD graph and access control policy will be translated by the `AnnotateNode` subprocedure into one of the two risk patterns in Fig. 4.3. Those risks translated into the first risk pattern will be detected by the `GenerateSanitizedDTD` procedure at the time n_2 is visited, as the full access for the parent node n_1 and the partial access for n_2 will cause the first condition within the test on line 5 to succeed. Similarly, for those risks conforming to the second risk pattern, the test on line 5 also succeeds since the conditions governing access to n_1 and n_2 are different, meaning that all such risks are in fact detected.

We now consider the removal of detected parent-child risks. As indicated above, the test on line 5 of `GenerateSanitizedDTD` succeeds when either risk pattern in Fig. 4.3 is encountered. Since n_2 is an element node type, the inner test on line 14 also succeeds, and the result is that the DTD graph edge incident to n_2 is annotated with the `?` cardinality constraint. This translates each risk pattern into the corresponding solution pattern depicted in the right-hand column of Fig. 4.3, indicating that all parent-child risks are successfully removed.

□

Example 4.7.3. *We illustrate the design-time analysis procedure using the sanitized DTD D_{AIns} generated for the adversary from the ongoing examples, Mallory. The initial DTD graph will be as depicted in Fig. 2.3. There are two rules in the ACP of Ex. 2.4.12 which are applicable to the ACME Insurance role held by Mallory: rule* `PR1` *(granting default access to all nodes in the database), and rule* `PR2` *(denying access to the names of patients who are not clients of ACME Insurance). After both rules are processed in Alg. 4.7.1, the DTD graph is annotated as depicted in Fig. 4.4a. As rule* `PR2` *prescribes conditional access to the* `hospital/patient/name` *node class, a further alteration is made to replace ("split") the corresponding* `name` *node in the DTD graph into two cases corresponding to*

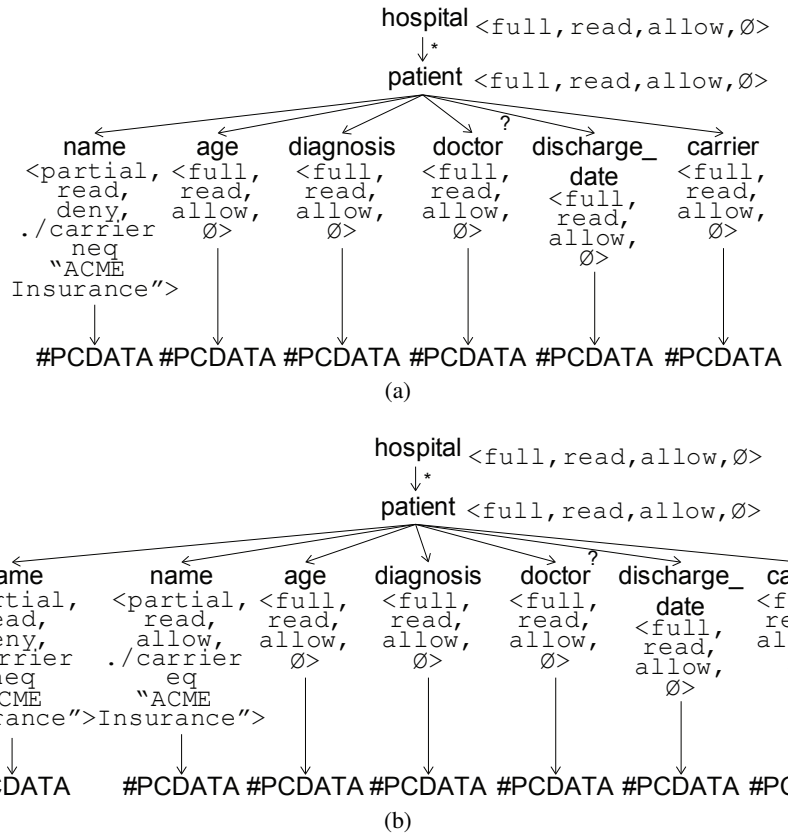


Figure 4.4: Example DTD graph after (a) annotation and (b) node splitting procedures.

when the condition is and is not satisfied, as shown in Fig. 4.4b.

The sanitized DTD D_{AIns} produced from the annotated DTD graph is listed in Fig. 4.5. Note that the rule for `patient` *elements has been generalized so that* `name` *child elements are now optional. In addition, an entry* $\langle R_{AIns}$, `/hospital/patient/name`, `./carrier neq "ACME Insurance"`$\rangle$ *is added to* $\mathcal{R}$. □

4.7.1 Discussion

In relation to the approaches discussed in Sec. 4.4.1, we note that our definition of parent-child disclosure incorporates the parent-child constraints of Yang and Li [119], and the two remaining types of schema-based disclosures are essentially specializations of parent-child disclosure that pertain to attribute node types. The remaining constraint types of Yang and Li's model (ancestor-descendant constraints and functional dependencies), as well as the constraints expressible in the model of Farkas et al [52], cannot be detected by our design time algorithm. Our query time algorithm (presented in the following chapter) is capable

```
R1: <!ELEMENT hospital (patient*)>
R2: <!ELEMENT patient (name?,age,diagnosis,doctor,
     discharge_date?,carrier)>
R3: <!ELEMENT name (#PCDATA)>
R4: <!ELEMENT age (#PCDATA)>
R5: <!ELEMENT diagnosis (#PCDATA)>
R6: <!ELEMENT doctor (#PCDATA)>
R7: <!ELEMENT discharge_date (#PCDATA)>
R8: <!ELEMENT carrier (#PCDATA)>
```

Figure 4.5: Sanitized DTD $\mathcal{D}_{\text{AIns}}$ for the hospital database.

of detecting these remaining forms of disclosure risk. The important distinction is that in our solution, parent-child disclosures are detectable at design time, while in the other approaches, detection and removal of such disclosures must be deferred until the precise database contents are known.

4.8 Conclusions

This chapter began by distinguishing between schema-based and instance-based disclosure risks for federated data, and presented a design-time solution for removing the former risk type. In the next chapter, we shall present a solution for instance-based disclosure risks.

Chapter 5

Detecting and Removing Instance-based Disclosure Risks in Federated Data

In the previous chapter, we outlined a design-time algorithm for detecting and removing schema-based disclosure risks. In this chapter, we propose a measure for evaluating instance-based disclosure risks, together with an algorithm for calculating this measure at query-time.

5.1 Instance-Based Disclosure Risks

Unfortunately, it is not possible to detect all disclosure risks at database design time. In particular, the actual threat posed by an identified potential risk can greatly vary as the database contents change. In other words, such risks may not be present in every database that conforms to the same DTD, requiring a case-by-case analysis at the time a potentially risky query is issued. We illustrate this with the following example.

Example 5.1.1. *(Total disclosure.) Recall Ex. 4.1.1, where the adversary Mallory was able to achieve a reduction in uncertainty of Alice's diagnosis based solely on the knowledge of the latter's age. This partial disclosure was due to the fact that only two out of three patient records in the database matched this age value. Now suppose that the* `patient` *subtree for Carol is deleted from the database. Issuing the same query from Ex. 4.1.1 will now return only one value,* `leukemia`*, allowing Mallory to learn Alice's diagnosis with complete certainty, thereby constituting a total disclosure.* □

It becomes clear that an effective measure for evaluating such *instance-based disclosure risks* must have access to – and be able to reason over – an accurate probabilistic model of the current database contents. We will elaborate on this further in Sec. 5.2.

5.2 Query-time Measurement of Instance-Based Risks

Recall that an instance-based disclosure risk is present when an adversary is able to apply a query answer to refine her probabilistic model of a database to an unacceptably-high degree of accuracy. Hence, a measure for such risks must take into account the current database contents (more precisely, the probabilistic relationships between node types that are defined by the database contents). We introduce such a measure based on concepts from information theory [111].

At a high level, our measure calculates the *magnitude* $\mathcal{M}$ of an instance-based disclosure risk as

$$\mathcal{M} = 1 - \frac{\text{computational effort for adversary to learn } s}{\text{avg. computational effort required by a random guess of } s}. \tag{5.1}$$

where s designates a *sensitive* location path that is inaccessible or partially-accessible to the adversary.

Information theoretic measures can be used to quantify computational effort in terms of bits. In particular, the *cross entropy* (cf. Sec. 2.5) of two probability distributions $\mathbf{P}$ and $\mathbf{Q}$ measures the average number of bits required to identify an event from $\mathbf{P}$, the true probability distribution, using $\mathbf{Q}$ as an approximation of $\mathbf{P}$. This corresponds to the numerator in Eq. 5.1. In our setting, $\mathbf{P}$ corresponds to the probability distribution defined over the entire database for instances of s appearing in the context of the query q, while $\mathbf{Q}$ is the corresponding probability distribution defined over the adversary's view of the database. Cross entropy in turn is the summation of two values, the *entropy* of $\mathbf{P}$, and the *relative entropy* of $\mathbf{P}$ and $\mathbf{Q}$. We shall describe the cross entropy calculation in greater detail shortly.

The denominator in Eq. 5.1 represents the worst-case scenario for the adversary in which the entropy for s is maximized: every value in $adom(path(s))$ occurs with equal probability, and therefore the adversary must resort to a random guess as to the value of an

s instance [1]. After dividing the cross entropy score by this value, we obtain a normalized value in $[0, \infty)$ indicating the relative amount of computational effort actually needed by the adversary to learn the hidden s value (i.e., 0 implies the adversary knows the value of hidden s with complete certainty after witnessing the answer to q, while any value greater than or equal to 1 indicates that no reduction in effort has occurred). By subtracting this quantity from 1, we obtain a disclosure risk magnitude score in $(-\infty, 1]$ in which higher values indicate greater risks, and any value less than or equal to 0 indicates that no disclosure has taken place.

5.2.1 Probabilistic Regular Tree Grammars (PRTGs)

The information theoretic measures discussed above all operate over probability distributions. To efficiently model probabilistic information for an XML database, we utilize *probabilistic regular tree grammars (PRTGs)*, an extension of the established notion of deterministic regular tree grammar [34] that assigns a probability to each grammar rule.

Definition 5.2.1 (Probabilistic Regular Tree Grammar (PRTG))**.** *A probabilistic regular tree grammar (PRTG) $G = \langle N, s, \mathcal{F}, R, \mathbf{P}_G \rangle$ is composed of a set N of non-terminal symbols; an axiom (start symbol) $s \in N$; a set $\mathcal{F}$ of terminal symbols; a set R of rules of the form $A \to term(\beta)$, where A is a non-terminal, $term \in \mathcal{F}$ is a terminal symbol, and β is either the empty tree or a tree containing non-terminal and/or terminal symbols; and $\mathbf{P}_G : R \to [0, 1]$ is a function assigning a probability to each rule. For each $n \in N$, the probabilities of all n-rules must sum to 1.* □

Occasionally, we abuse the notation slightly by stating that a rule $r \in G$ as a shorthand for indicating that $r \in R$, where R is the set of rules associated with PRTG G.

PRTGs are built by traversing the XML database tree bottom-up according to the algorithm listed as Alg. 5.2.1. This procedure invokes the recursive `BuildGrammarRule` subprocedure on the root node, and generates the PRTG over the entire XML tree based on the result.

[1] This assumes that the adversary has zero prior knowledge as to the true probability mass function for $adom(path(s))$. Cases in which an adversary does possess *partial* prior knowledge of the true function can be modelled by substituting the entropy of $\mathbf{Q}$, the probability mass function representing this prior knowledge, as the denominator.

Input: Root node $root$ of an XML tree
Output: Generated PRTG $G = \langle N, s, \mathcal{F}, R, \mathbf{P} \rangle$
1 $N \leftarrow \emptyset; \mathcal{F} \leftarrow \emptyset; R \leftarrow \emptyset;$
2 **return** $\langle headSymbol, rhsList \rangle \leftarrow$ `BuildGrammarRule`$(root, N, \mathcal{F}, R)$*;*

Algorithm 5.2.1: Procedure `BuildGrammar` for generating a probabilistic regular tree grammar (PRTG) from an input XML tree T.

Most of the grammar building functionality is performed by the `BuildGrammarRule` subprocedure. Each iteration builds the grammar rule describing a specific subtree of T, identified by the input node. Other inputs include the working sets of terminal symbols ($\mathcal{F}$), non-terminal symbols (N), and grammar rules (R). During execution, the subprocedure adds additional elements to these sets, corresponding to the newly constructed grammar rules. Line 3 adds the node type of the subtree root node to the new rule's RHS. If the root node is an attribute type, then the RHS will also contain the attribute value (lines 4-5). Otherwise, the root node must be an element type and its child nodes will be processed by lines 7-15. The `BuildGrammarRule` subprocedure is recursively invoked for element and attribute children (line 9), with the head symbol of the rule constructed by the recursive call being added to the current rule's right-hand side (line 10). In the case of PCDATA children, the node value is simply added to the current rule's right-hand side (line 12) and to the set of terminal symbols (line 13). Once all children nodes have been processed, the right-hand side for the current rule is closed off (line 17), and a lookup is performed in the set of existing rules to see if a matching rule has previously been constructed (lines 18-26). If the rule does not appear, then it is added to R with an initial occurrence count of 1 (line 20) its head symbol is added to N, the set of non-terminals (line 21), and the signature of its right-hand side (formed by replacing the head symbols of child rules in the right-hand side by a placeholder) is added to the set of terminals, $\mathcal{F}$ (line 22). If the rule has previously been created, the new occurrence is assigned the head symbol of the existing rule (line 24), and the occurrence count for the existing rule is incremented (line 25). At the conclusion of the procedure, the head and right-hand side of the constructed rule are returned (line 27).

As a notational convenience, we utilize subscripts to distinguish between different rules with the same non-terminal in the rule head. Each application of a rule p derives a tree t, which can be denoted as $p \Rightarrow t$. In general, t may consist of a mixture of terminal and

non-terminal symbols; by recursively applying the rules corresponding to the non-terminal symbols appearing in each derived tree t, one eventually ends up with a tree containing only terminal symbols, referred to as a *ground tree*.

Property 5.2.2. *The ground tree derivable starting from a specific rule p is unique; by $p \stackrel{*}{\Rightarrow} t$, we denote that t is the unique ground tree derivable (in one or more steps) from p. Further, each occurrence of a specific ground tree t is produced by the same grammar rule p.* □

As will be seen shortly, this one-to-one correspondence between a grammar rule and a ground tree plays an integral role within our algorithm for detecting instance-based disclosure risks. We now provide an example illustrating the generation of an XML tree from a PRTG.

Example 5.2.3. *The PRTG generating the example hospital database of Fig. 2.1 is given by $G = \langle N, \texttt{HOSPITAL}_1, \mathcal{F}, R, \mathbf{P}_G \rangle$, with N, $\mathcal{F}$, R, and $\mathbf{P}_G$ as shown in Fig. 5.1.* □

For databases containing a high degree of structural repetition, the corresponding PRTG will be much smaller than the original database size; as the following result shows, in cases where there is little redundancy, the PRTG size approaches that of the database.

Property 5.2.4. *The overall size of the database grammar, measured as the sum of symbols (terminal and non-terminal) appearing in all rules, increases linearly with the size of the database T.*

Proof. The number of rules in the grammar is evidently $O(|V|)$, where $|V|$ is the number of nodes in the database tree, since each node is produced from the application of a single grammar rule. In the worst case, we have a database tree in which each node has a single child, and hence, a new grammar rule is applied to produce each of the $|V|$ nodes in the tree. Further, the size of each rule's right-hand side is also bounded by $|V|$, since each node is produced by a single rule (and therefore, can appear in the right-hand side of only one rule). □

The following result proves beneficial for efficiently calculating the magnitude of instance-based disclosure risks. It builds upon Property 5.2.2, and indicates that one can reason over

Grammar Rules (R)			**Rule Prob. ($\mathbf{P}_G$)**
$\texttt{HOSPITAL}_1$	$\rightarrow$	$\texttt{hospital}(\texttt{PATIENT}_1, \texttt{PATIENT}_2, \texttt{PATIENT}_3)$	1
$\texttt{PATIENT}_1$	$\rightarrow$	$\texttt{patient}(\texttt{NAME}_1, \texttt{AGE}_1, \texttt{DIAGNOSIS}_1, \texttt{DOCTOR}_1, \texttt{DISCHARGE\_DATE}_1, \texttt{CARRIER}_1)$	1/3
$\texttt{PATIENT}_2$	$\rightarrow$	$\texttt{patient}(\texttt{NAME}_2, \texttt{AGE}_2, \texttt{DIAGNOSIS}_2, \texttt{DOCTOR}_2, \texttt{CARRIER}_1)$	1/3
$\texttt{PATIENT}_3$	$\rightarrow$	$\texttt{patient}(\texttt{NAME}_3, \texttt{AGE}_1, \texttt{DIAGNOSIS}_3, \texttt{DOCTOR}_3, \texttt{CARRIER}_2)$	1/3
$\texttt{NAME}_1$	$\rightarrow$	$\texttt{name}$(“Alice”)	1/3
$\texttt{NAME}_2$	$\rightarrow$	$\texttt{name}$(“Bob”)	1/3
$\texttt{NAME}_3$	$\rightarrow$	$\texttt{name}$(“Carol”)	1/3
$\texttt{AGE}_1$	$\rightarrow$	$\texttt{age}$(“31”)	2/3
$\texttt{AGE}_2$	$\rightarrow$	$\texttt{age}$(“57”)	1/3
$\texttt{DIAGNOSIS}_1$	$\rightarrow$	$\texttt{diagnosis}$(“leukemia”)	1/3
$\texttt{DIAGNOSIS}_2$	$\rightarrow$	$\texttt{diagnosis}$(“pulmonary fibrosis”)	1/3
$\texttt{DIAGNOSIS}_3$	$\rightarrow$	$\texttt{diagnosis}$(“pneumonia”)	1/3
$\texttt{DOCTOR}_1$	$\rightarrow$	$\texttt{doctor}$(“House”)	1/3
$\texttt{DOCTOR}_2$	$\rightarrow$	$\texttt{doctor}$(“Mancini”)	1/3
$\texttt{DOCTOR}_3$	$\rightarrow$	$\texttt{doctor}$(“Cox”)	1/3
$\texttt{DISCHARGE\_DATE}_1$	$\rightarrow$	$\texttt{discharge\_date}$(“23/08/05”)	1
$\texttt{CARRIER}_1$	$\rightarrow$	$\texttt{carrier}$(“Black Cross”)	2/3
$\texttt{CARRIER}_2$	$\rightarrow$	$\texttt{carrier}$(“White Shield”)	1/3

$N = \{\texttt{HOSPITAL}_1, \texttt{PATIENT}_1, \texttt{PATIENT}_2, \texttt{PATIENT}_3, \texttt{NAME}_1, \texttt{NAME}_2, \texttt{NAME}_3, \texttt{AGE}_1, \texttt{AGE}_2, \texttt{DIAGNOSIS}_1, \texttt{DIAGNOSIS}_2, \texttt{DIAGNOSIS}_3, \texttt{DOCTOR}_1, \texttt{DOCTOR}_2, \texttt{DOCTOR}_3, \texttt{DISCHARGE\_DATE}_1, \texttt{CARRIER}_1, \texttt{CARRIER}_2\}$

$\mathcal{F} = \{\texttt{hospital}(\cdot,\cdot,\cdot), \texttt{patient}(\cdot,\cdot,\cdot,\cdot,\cdot), \texttt{patient}(\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot), \texttt{name}(\cdot), \texttt{age}(\cdot), \texttt{diagnosis}(\cdot), \texttt{doctor}(\cdot), \texttt{discharge\_date}(\cdot), \texttt{carrier}(\cdot),$ “Alice”, “Bob”, “Carol”, “31”, “57”, “leukemia”, “pulmonary fibrosis”, “pneumonia”, “House”, “Mancini”, “Cox”, “23/08/05″, “Black Cross”, “White Shield”$\}$

Figure 5.1: Probabilistic regular tree grammar describing the hospital tree of Fig. 2.1.

the probability distribution formed by p-rules within a PRTG G, in place of reasoning over the probability distribution formed by p-trees within the XML tree generated by G.

Lemma 5.2.5. *For any location path p (cf. 2.4.2), the set of p-rules derives the set of all p-trees, and no other trees. Further, there exists for each unique p-tree t a single rule r_{p_i} satisfying $r_{p_i} \stackrel{*}{\Rightarrow} t$, and the occurrence count assigned to p-rule r_{p_i} equals the number of occurrences of t in T.*

Proof. We illustrate using an inductive proof.

Basis: Let p denote the parent node type of a leaf node type c. The latter can be either a PCDATA node, an attribute node, or an element node type instance corresponding to an empty element. In the former case, then by line 12 of `BuildGrammarRule` a rule of the form $p \rightarrow p(text)$ will be constructed, where $text$ is the value of the c node. If this rule does not already appear in the grammar, it will be added to the rule set R with an initial occurrence count of 1 (line 20); if it is present, its occurrence count will be incremented (line 25).

If c is non-PCDATA, a c-rule $c \rightarrow c()$ will be constructed for the instance (line 9), and a reference to this rule will be added to the RHS of the rule for the parent p instance (line 10). As above, the occurrence count for the newly created c-rule will be either incremented (if this rule previously exists in R) (line 25), or will be added to R with an occurrence count of 1 (line 20).

If c is an attribute node type, then a rule of the form $c \rightarrow @c(val)$ will be formed, where val designates the attribute value (line 5). If this rule does not occur in R, it is added with an occurrence count of 1 (line 20). Otherwise, its current count is incremented (line 25). A reference to the newly constructed c-rule is added to the RHS of the rule belonging to the parent p instance (line 10).

Each c instance is therefore associated with exactly one p-rule r_{p_i} in G, and the count associated with r_{p_i} equals the number of times the rule has been applied. Additionally, a p-rule will only be constructed when a p instance is encountered in T.

Inductive step: Assume that $height(T) - k + 1$ levels of T have been processed by

Alg. 5.2.1, and that for each node type n at levels $k, k+1, \ldots, height(T)$, the generated set of n-rules derives exactly the set of n-trees. Further, assume that the occurrence count for each n-rule equals the number of times its generated ground tree appears in T. We first consider the case in which an attribute node type instance appearing at level $k-1$ is next to be processed. By line 5, a rule of the form $p \rightarrow @p(val)$ is formed, where val indicates the attribute value. If this rule does not previously appear in R, it is added with an initial occurrence count of 1 (line 20); otherwise, its existing count is incremented (line 25). Under this construction, one p-rule is applied per occurrence of a p-tree in T, and further, the occurrence count of a p-rule will match the number of times its ground p-tree appears in T.

Now assume that an instance of an element node type p appearing at level $k-1$ is next to be processed, with an ordered set of child node types $c_1, \ldots, c_m$. Each child c_i will be processed in order (line 7 of `BuildGrammarRule`), according to the following strategy:

- if c_i is a PCDATA node instance, then $text$ will be appended to the RHS of the p-rule corresponding to the parent p instance (line 12);
- if c_i is an instance of an attribute node type, then it will have an existing rule $r_{c_i} \rightarrow @c_i(val)$, where val is the attribute value (line 5). A reference to r_{c_i} is then added to the RHS of the rule associated with the parent p instance (line 10).
- if c_i is an instance of an element node type, then it will have an existing rule $r_{c_i} \rightarrow c_i(gc_1, \ldots, gc_x)$, where $gc_1, \ldots, gc_x$ is the ordered list of child nodes for c_i (lines 5-17), and a reference to r_{c_i} will be added to the RHS of the rule associated with the parent p instance (line 10).

Once all children nodes have been processed, the RHS for the p-rule will be closed off (line 13). The final form of the rule will be $p \rightarrow p(r_{c_1}, \ldots, r_{c_m})$, where r_{c_i} is a reference to the corresponding c_i-rule belonging to child c_i (if c_i is an instance of an element or attribute node type), or the value of c_i (if it is an instance of the PCDATA node type). This rule will be added to R with an initial count of 1, if not already present in R (line 16); otherwise, the count for the existing rule will be incremented (line 21). In either case, there will exist

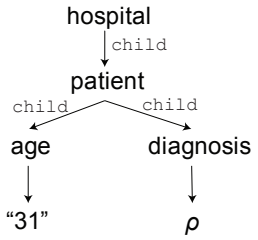


Figure 5.2: Twig representation of the query from Ex. 4.1.1.

a separate p-rule for each distinct p-tree in T, and the occurrence count of each p-rule will equal the number of times its derived p-tree appears in T. □

5.2.2 Twigs

Within the context of query analysis, we will frequently need to select only those subtrees within an XML database that possess specific structural properties. For this purpose, we use the notation of *twigs* [73]. A twig mandates a specific label from $\Sigma_e \cup \Sigma_a \cup \mathcal{S} \cup$ '*' $\cup \rho$ for each node (where $\mathcal{S}$ denotes the set of all strings and '*' is a wildcard character indicating *any* string constitutes a match) at each position. Twig nodes labeled with ρ denote those contained in the query answer. A twig is built from the top down based on a left-to-right reading of an XPath query, such that the twig root corresponds to the first location step, its children to the second location step, and so on. An edge connecting two twig nodes is labelled with the XPath axis (`child`, `attribute`, `descendant`, or `descendant-or-self`) [2] connecting the corresponding location steps in the query. The absence of such a label indicates that the `child` axis is in use.

Example 5.2.6. *The twig representation of the query from Ex. 4.1.1 is given in Fig. 5.2.* □

Definition 5.2.7 (Twig Satisfiability)**.** *A twig τ is* satisfied *by a ground tree t iff there exists a mapping M from the nodes of τ to the nodes of t such that (1) for each node n of τ with $type(n) \in \Sigma_e \cup \Sigma_a$, $label(M(n)) = type(n)$; (2) for each node n of τ with $value(n) \in \mathcal{S}$, $label(M(n)) = value(n)$; (3) for each* `child` *and* `attribute` *labelled edge (n_1, n_2) in τ, there is a corresponding edge $(M(n_1), M(n_2))$ in t; and (4) for each* `descendant` *and*

[2] Note that the remaining XPath axes (cf. Sec. 2.4.3) are either captured by the twig structure (`preceding`, `following`, `preceding-sibling`, and `following-sibling` axes), or in the case of `ancestor` and `ancestor-or-self` axes, are expressible as `descendant` or `descendant-or-self` labeled twig edges, respectively, in which the node labels connected by the edge are swapped.

`descendant-or-self` *labelled edge* (n_1, n_2) *in* τ*, there is a corresponding directed path from* $M(n_1)$*) to* $M(n_2)$ *in* t*.* □

By $forest(\tau, T)$, we denote the set of all ground subtrees in database T that satisfy twig τ. Twigs can also be applied to a PRTG G describing an XML database T in a straightforward manner: we say that a rule r in G *satisfies* a twig τ if there is a matching between τ and the right-hand side of r. Then $prod(\tau, G)$ denotes the set of rules in G that satisfy τ, i.e., those rules $p \in G$ for which $p \stackrel{*}{\Rightarrow} t$, for some $t \in forest(\tau, T)$. By $\mathbf{P}_{\{\tau,G\}}$ we indicate the probability distribution function defined over $prod(\tau, G)$, i.e., for each rule $r_i \in prod(\tau, G)$,

$$\mathbf{P}_{\{\tau,G\}}[r_i] = \frac{\mathbf{P}_G[r_i]}{\sum_{r_j \in prod(\tau,G)} \mathbf{P}_G[r_j]}\,.$$

5.2.3 Cross Entropy

From the perspective of the adversary, one can view instance-based disclosure as a learning problem. For a specific hidden node s of type $path(s)$, the adversary is limited to applying a partially-learned probabilistic model of $path(s)$, derived from his view of the database, to attempt to infer the correct value of s. Denoting by G and G_A, respectively, the PRTGs defined over the entire database and that restricted to the adversary's view of the database, one intuitively observes that the disclosure risk will be smallest when the following two conditions are true:

1. *The value of the hidden node* s *is difficult to guess.* More precisely, each value within $adom(path(s))$ occurs with roughly equal probability in G.

2. *The adversary's view provides a poor approximation of the probability distribution for* $adom(path(s))$ *defined over the entire database.* In other words, for each value $d \in adom(path(s))$, $\mathbf{P}_G(d)$ and $\mathbf{P}_{G_A}(d)$ differ substantially.

Information theoretic measures can be used to model both conditions. In particular, minimizing the risk according to the above two conditions equates to maximizing the cross entropy between $\mathbf{P}_G$ and $\mathbf{P}_{G_A}$. The first condition equates to the entropy of the probability distribution function for $prod(\tau, G)$ where τ is the twig representation of the issued query

q, and G is the PRTG defined over the entire database. Adapting the entropy formula [35] to this setting gives us the following.

Definition 5.2.8 (Twig Entropy)**.** *The entropy of a twig τ over a PRTG G, denoted as $H_{\{\tau,G\}}$ and expressed in bits, is computed as*

$$H_{\{\tau,G\}} = -\sum_{r_i \in prod(\tau,G)} \mathbf{P}_{\{\tau,G\}}[r_i] \cdot \log_2 \mathbf{P}_{\{\tau,G\}}[r_i]\,. \tag{5.2}$$

□

To see why twig entropy satisfies the first condition above, it is helpful to employ the interpretation of entropy that sees it as the average required number of "yes/no" questions one needs to ask in order to identify an element from a set of possible alternatives; in the present case, this corresponds to how many questions the adversary must ask in order to determine the correct value of s from $adom(path(s))$, the set of all candidate values. We observe the following property of twig entropy values.

Property 5.2.9. *Twig entropy ranges from $0 \leq H_{\{\tau,G\}} \leq \log_2 |prod(\tau, G)|$, with $H_{\{\tau,G\}} = 0$ when $prod(\tau, G)$ contains a single rule (i.e., the identity of each rule satisfying τ can be predicted with complete certainty). When the identity of the satisfying rule is completely unpredictable (i.e., it is equally likely that the identity of each rule satisfying τ could be any of the rules in $prod(\tau, G)$), $H_{\{\tau,G\}} = \log_2 |prod(\tau, G)|$.* □

The second condition above requires one to determine the "distance" between two probability distribution functions defined over $adom(path(s))$: one generated by answering the query q over the entire database, and another formed by answering q over the adversary's view of the database. This type of calculation corresponds to computing the *relative entropy* (cf. Sec. 2.5) of the two distributions, as defined below.

Definition 5.2.10 (Relative Entropy of PRTGs)**.** *For a fixed twig τ, the* relative entropy *of two PRTGs G_1 and G_2, for which $prod(\tau, G_1) = prod(\tau, G_2)$, is denoted by $KL(\tau, G_1||\tau, G_2)$ and expressed in bits. It is computed as*

$$KL(\tau, G_1||\tau, G_2) = \sum_{r_i \in prod(\tau,G_1)} \mathbf{P}_{\{\tau,G_1\}}[r_i] \cdot \log_2 \left(\frac{\mathbf{P}_{\{\tau,G_1\}}[r_i]}{\mathbf{P}_{\{\tau,G_2\}}[r_i]} \right) \tag{5.3}$$

with conventions $0 \log_2 \frac{0}{0} = 0$, $0 \log_2 \frac{0}{q} = 0$, *and* $p \log_2 \frac{p}{0} = \infty$. □

It always holds that $KL(\tau, G_1 || \tau, G_2) \geq 0$, with equality if and only if $\forall r_i \in prod(\tau, G_1)$, $\mathbf{P}_{\{\tau,G_1\}}[r_i] = \mathbf{P}_{\{\tau,G_2\}}[r_i]$. We utilize twig entropy and relative entropy of PRTGs within our measure for evaluating the magnitude of instance-based disclosure risks, as follows.

Definition 5.2.11 (Measuring Disclosure Risk)**.** *The* magnitude *of the disclosure risk presented by a particular twig* τ *to a secret node* s*, within the context of a database PRTG* G *and a second PRTG* G_A *describing the database view accessible to the adversary, is denoted by* $\mathcal{M}(\tau \rightsquigarrow s, G, G_A)$ *and computed as*

$$\mathcal{M}(\tau \rightsquigarrow s, G, G_A) = 1 - \left(\frac{H_{\{\tau,G\}} + KL(\tau, G || \tau, G_A)}{\log_2 |adom(path(s))|} \right). \tag{5.4}$$

By convention, $\mathcal{M}(\tau \rightsquigarrow s, G, G_A) = 1$ *when the denominator* $\log_2 |adom(path(s))| = 0$. □

In cases of *total* disclosure, $\mathcal{M}(\tau \rightsquigarrow s, G, G_A) = 1$, while $0 < \mathcal{M}(\tau \rightsquigarrow s, G, G_A) < 1$ indicates the existence of a *partial* disclosure risk. If $\mathcal{M}(\tau \rightsquigarrow s, G, G_A) \leq 0$, there is no disclosure risk present.

One can simplify the numerator to obtain a more concise version of the risk magnitude equation:

$$\mathcal{M}(\tau \rightsquigarrow s, G, G_A) = 1 - \left(\frac{- \sum\limits_{r_i \in prod(\tau,G)} \mathbf{P}_{\{\tau,G\}}[r_i] \cdot \log_2 \mathbf{P}_{\{\tau,G_A\}}[r_i]}{\log_2 |adom(path(s))|} \right). \tag{5.5}$$

Property 5.2.12. $\mathcal{M}(\tau \rightsquigarrow s, G, G_A)$ *is computable in* $O(|V|)$ *time, where* $|V|$ *denotes the number of nodes in the database* T *derived by* G*.*

Proof. (Sketch.) Since each node in T is produced via application of a single rule in G, there can be at most $|V|$ rules in G, and hence, $|prod(\tau, G)| \leq |V|$. Computing twig entropy over the database grammar requires $|prod(\tau, G)|$ logarithm calculations, $|prod(\tau, G)|$ multiplications, and $|prod(\tau, G)|$ additions, resulting in an overall time cost of $O(|prod(\tau, G)|) = O(|V|)$. Computing the relative entropy over the database grammar and the adversary's

view grammar requires a similar number of operations, so it also requires $O(|V|)$ time. The denominator of Eq. 5.4 can be computed in constant time, leading to an overall time cost of $O(|V|)$. □

In the interests of promoting data utility and availability, administrators at the federation and data source levels may agree to view certain disclosure risks as acceptable, as long as the magnitude of the risk is "small enough". In our model, this trade-off is specified using a parameter $\epsilon \in [0, 1]$. In such cases, we say that a disclosure $\tau \rightsquigarrow s$ exists only if $\mathcal{M}(\tau \rightsquigarrow s, G, G_V) \geq \epsilon$.

One can also make the following observations about the $\mathcal{M}$ function: (1) if $H_{\{\tau,G\}} = \log_2 |adom(path(s))|$, then $path(s)$ is *already random* and therefore the adversary cannot do better than a random guess, regardless of how accurately G_A represents the true probabilistic model of $adom(path(s))$; (2) if $H_{\{\tau,G\}}$ is near 0, then $adom(path(s))$ is already *highly predictable* and therefore may lead to a demonstrable risk even in cases where G_A provides the adversary with an inaccurate approximation of the true probabilistic model of $adom(path(s))$.

5.2.4 Risk Analysis Procedure

We begin our description of the query-time risk analysis procedure by proving a useful property that allows us to reduce the time requirement for performing the analysis.

Definition 5.2.13 (Twig Generalization)**.** *Let τ_2 be a twig rooted by node type* `b`*. We say another twig τ_1* generalizes *τ_2 iff (1) τ_2 forms a proper subtree of τ_1, and (2) τ_1 is rooted by a node type* `a` *that is the parent of* `b`*.* □

The generalization relationship models a bottom-up traversal of a twig. Specifically, τ_2 in the definition corresponds to the twig formed at node type `b`, while τ_1 reflects the twig formed one level higher, at the parent node type `a`.

Property 5.2.14. *(Twig generalization does not decrease cross-entropy.) For any twig τ_1 that generalizes twig τ_2, and for any fixed PRTGs G and G_A,*

$$H_{\{\tau_2,G\}} + KL(\tau_2, G \,||\, \tau_2, G_A) \leq H_{\{\tau_1,G\}} + KL(\tau_1, G \,||\, \tau_1, G_A)\,.$$

Proof. We assume that τ_2 is rooted by node type `b`, and τ_1 by parent node type `a`, as in Def. 5.2.13 above. Let $b_1, \ldots, b_m$ designate the set of b-trees comprising $forest(\tau_2, G)$, and let $o_i \geq 1$ and $o_i^{'} \geq 0$ indicate the occurrence counts of $b_i \in forest(\tau_2, G)$ within the entire database and within the adversary's view, respectively. For simplicity, we assume that grammar rule r_i identifies the rule that generates b_i. Then we arrive at the following cross entropy calculation for τ_2, using the concise form shown in the numerator of Eq. 5.5:

$$\begin{aligned} H_{\{\tau_2,G\}} + KL(\tau_2, G \,||\, \tau_2, G_A) &= \sum_{r_i \in prod(\tau_2,G)} \mathbf{P}_G[r_i] \cdot \log_2 \left(\frac{1}{\mathbf{P}_{G_A}[r_i]} \right) \\ &= \sum_{i=1}^{m} \mathbf{P}_G[b_i] \cdot \log_2 \left(\frac{1}{\mathbf{P}_{G_A}[b_i]} \right) \\ &= \sum_{i=1}^{m} \left[\frac{o_i}{\sum_{j=1}^{m} o_j} \cdot \log_2 \left(\frac{\sum_{j=1}^{m} o_j^{'}}{o_i^{'}} \right) \right] . \end{aligned} \tag{5.6}$$

Now, let $\rho(b_i)$ be the set of parent rules for b_i, namely those a-rules containing a reference to rule r_i on their RHS. By the definition of twig generalization, we have $prod(\tau_1, G) = \rho(b_1) \cup \ldots \rho(b_m)$. Additionally, the sum of occurrence counts for each a-rule in $\rho(b_i)$ equates to o_i, since each occurrence of b_i must have a parent a_i instance, and by Lemma 5.2.5 each such parent-child relationship is captured by exactly one rule. Allowing $o_{\rho(b_i)[j]}$ and $o_{\rho(b_i)[j]}^{'}$ to denote the occurrence count of the j-th rule in $\rho(b_i)$ in the entire database and in the adversary's view, respectively, we arrive at the following cross entropy calculation for τ_1:

$$\begin{aligned} H_{\{\tau_1,G\}} + KL(\tau_1, G \,||\, \tau_1, G_A) &= \sum_{r_{a_i} \in prod(\tau_1,G)} \mathbf{P}_G[r_{a_i}] \cdot \log_2 \left(\frac{1}{\mathbf{P}_{G_A}[r_{a_i}]} \right) \\ &= \sum_{i=1}^{m} \sum_{j=1}^{|\rho(b_i)|} \left[\mathbf{P}_G[\rho(b_i)[j]] \cdot \log_2 \left(\frac{1}{\mathbf{P}_{G_A}[\rho(b_i)[j]]} \right) \right] \\ &= \sum_{i=1}^{m} \left[\sum_{j=1}^{|\rho(b_i)|} \left[\frac{o_{\rho(b_i)[j]}}{o_i} \cdot \log_2 \left(\frac{o_i^{'}}{o_{\rho(b_i)[j]}^{'}} \right) \right] \right] \\ &= \sum_{i=1}^{m} \left[\sum_{j=1}^{|\rho(b_i)|} \left[\frac{o_{\rho(b_i)[j]}}{\sum_{k=1}^{|\rho(b_i)|} o_{\rho(b_i)[k]}} \cdot \log_2 \left(\frac{\sum_{k=1}^{|\rho(b_i)|} o_{\rho(b_i)[k]}^{'}}{o_{\rho(b_i)[j]}^{'}} \right) \right] \right] \end{aligned} \tag{5.7}$$

Next, note that the inner sum in Eq. 5.7 computes the cross-entropy over the probability mass functions induced over the database and the adversary's view for a fixed $\rho(b_i)$. By the properties of entropy and relative entropy, each inner cross-entropy reaches a minimum value of 0 only when the following conditions all hold: (1) $|\rho(b_i)| = 1$ and therefore $o_{\rho(b_i)[1]} = o_i$ (i.e., there is only one parent rule for b_i, so there is no uncertainty and therefore entropy over $\rho(b_i)$ is minimized); and (2) $o_i = o_i^{'}$ and $o_{\rho(b_i)[1]} = o^{'}_{\rho(b_i)[1]}$ (i.e., the respective occurrence counts for each b-tree and a-tree match in the database and in the adversary's view, leading to a relative entropy of 0). Rewriting Eq. 5.7 using these equalities, we obtain

$$\begin{aligned} H_{\{\tau_1,G\}} + KL(\tau_1, G \,||\, \tau_1, G_A) &= \sum_{i=1}^{m} \left[\sum_{j=1}^{|\rho(b_i)|} \left[\frac{o_{\rho(b_i)[j]}}{\sum_{k=1}^{|\rho(b_i)|} o_{\rho(b_i)[k]}} \cdot \log_2 \left(\frac{\sum_{k=1}^{|\rho(b_i)|} o^{'}_{\rho(b_i)[k]}}{o^{'}_{\rho(b_i)[j]}} \right) \right] \right] \\ &= \sum_{i=1}^{m} \left[\frac{o_i}{\sum_{j=1}^{m} o_j} \cdot \log_2 \left(\frac{\sum_{j=1}^{m} o_j^{'}}{o_i^{'}} \right) \right] \\ &= H_{\{\tau_2,G\}} + KL(\tau_2, G \,||\, \tau_2, G_A)\,. \end{aligned}$$

For cases in which one or more of the above conditions fail to hold, at least one of the inner cross-entropy values will be greater than zero, leading to

$$H_{\{\tau_2,G\}} + KL(\tau_2, G \,||\, \tau_2, G_A) < H_{\{\tau_1,G\}} + KL(\tau_1, G \,||\, \tau_1, G_A)$$

and completing the proof. □

Note that this result holds for any parent-child relationship within an XML tree. In particular, any grammar rule describing the subtree rooted at a parent node type p generalizes any rule describing the subtree rooted at its child node type c. One intuitively sees this by visualizing walking the XML tree from top-to-bottom: at the time that an instance of p is visited, one has no knowledge of the structure of that instance's subtree (only that it must conform to *one* of the p-rules defined within the database grammar). It is only after travelling into the next level of the tree that one knows whether that p instance has an instance of c as its child. More generally, descending into deeper levels of the subtree of the p instance serves to disqualify a greater number of potential rules that *could* apply to that tree level, thereby reducing uncertainty as to the contents of future levels. Hence, the gen-

eralization relationship for grammar rules further extends to cover all ancestor-descendant relationships within an XML tree.

We now move on to describe the query-time risk analysis procedure. Recall from Sec. 4.7 that during the DTD sanitization procedure (cf. Alg. 4.7.3) all potential instance-based disclosure risks applicable to the active federated ID f – in the form of partially- and non-accessible location paths – are recorded in the disclosure risks table $\mathcal{R}$. When a query q is issued against database T by an adversary holding f, $\mathcal{R}$ is consulted to retrieve all sensitive (i.e., partially- and non-accessible) location paths s for f. For each such s, the evaluation procedure of Alg. 5.2.3 is carried out.

First, a twig τ is formed from q (line 1). Next, it is determined whether τ represents a potential risk for the current sensitive location path s (line 2). This entails establishing if either (1) s is contained in τ, or (2) a path exists between s and a node contained in τ. If neither condition is met, the algorithm terminates and answers that no risk is present (line 3). Otherwise, τ is *extended* if necessary in order to form a path between the closest ancestor node of s in the original τ and s itself (line 5). We refer to this closest ancestor node as c and the extended twig as τ'. In the next step, a bottom-up *matching* is performed between τ' and the PRTG G corresponding to the entire database T, in which each subtree of τ' appearing in the RHS of a rule r in G is replaced by a single node labeled with the non-terminal symbol appearing in the LHS of r (in cases where multiple rules match a subtree, the replacement consists of a list containing the head symbol of each such rule) (line 6). During the matching process, each twig node in τ' is also annotated with the set of nodes of the corresponding node class which are inaccessible to f. The matching process terminates once c has been matched, leveraging Property 5.2.14. In particular, this property tells us that if no risk is constituted at c, no risk can be present when evaluated at any ancestor of c; similarly, the property also tells us that the amount of information the adversary gains from the query answer is greatest at c (since entropy will be lower at this node than in any ancestor node appearing in the query), so it is equally important not to terminate the matching process any earlier. In the fourth step, the twig entropy (cf. Eq. 5.2) is computed over the probability distribution formed by the rule(s) matching c (line 7).

The bottom-up matching procedure is then repeated, this time using the PRTG G_f, the

PRTG describing the database view of f, in place of G (line 8). Note that it is not necessary to explicitly store G_f in the database as we did with G. Instead, G_f can be computed "on-the-fly" as the bottom-up matching is conducted between τ' and G_f, by simply decrementing the existing counts for the grammar rules associated with each inaccessible node class instance stored in the corresponding twig node. At the next twig level, the parent node will then store grammar rule(s) capable of matching against the updated set of matching rules for each child, thereby propagating changes to the original grammar further up the twig. As before, matching concludes at c. To facilitate the accurate computation of relative entropy of c-rules in G and G_f, a further step is needed in which any c-rule c_f in G_f not appearing in G is replaced with the shortest c-rule in G containing all of the symbols found on the RHS of c_f (note that by the construction of G and G_f, such a rule in G is guaranteed to exist). The occurrence count assigned to c_f is then allocated to the c-rule in G replacing it (line 9). This is a manifestation of *Occam's razor*: in the absence of a complete match the adversary chooses the shortest rule (simplest hypothesis) in G whose RHS contains the RHS of c_f. The relative entropy calculation between c-rules in G and G_f is then carried out (cf. Eq. 2.2) (line 10), and the risk magnitude calculation itself is performed (cf. Eq. 5.4) (line 11). Finally, if the computed risk magnitude meets or exceeds the threshold value ϵ, a "yes" answer is returned to indicate that a disclosure risk is indeed present (line 13); otherwise, "no" is returned (line 15).

Theorem 5.2.15. *Alg. 5.2.3 correctly identifies disclosure risks.*

Proof. We demonstrate that the algorithm is both *sound* (i.e., it identifies as a disclosure risk any instance in which the computed magnitude, according to Eq. 5.4, equals or exceeds the threshold value ϵ) and *complete* (i.e., the algorithm reports that a disclosure risk occurs only if the same instance results in a computed magnitude equal to or exceeding ϵ, according to Eq. 5.4). Comparing Eq. (2.1) with Eq. (5.2), and Eq. (2.2) with Eq. (5.3), one sees that twig entropy and relative entropy of PRTGs are extensions of their classic counterparts in information theory: each performs the same fixed calculation over an input probability mass function as does the corresponding classic measure. Using the established soundness and completeness results for these classic measures as a starting point, what remains to be

shown is that the twig entropy and relative entropy of PRTG measures operate over the correct probability mass function. This equates to establishing that at each level during the rule matching process, the set of applicable rules $prod(\tau, G)$ is *exactly* the same as the set of matching rules reported by Alg. 5.2.3 at that level. Lemma 5.2.5 states that the set of p-rules in a grammar produces all p-trees, and further, that the occurrence count of each p-rule equals the occurrence count of its unique ground tree. Since the rule matching procedure only selects those p-rules that match the structure of the subtwig rooted at node type p, and only those p-trees within the XML tree T matching the twig structure form part of the query answer, the matched set of rules at p generates exactly the set of p-trees that match the query. And, further, the probability mass function defined over the matched p-rules is equivalent to that defined over p-trees contained in the query result. From this point, an inductive argument similar to that of Lemma 5.2.5 is used to verify that the correct probability mass function is utilized at each twig node position during the bottom-up twig matching process.

The justification for stopping the rule matching process at the closest ancestor node to the sensitive node appearing in the original (unextended) twig comes from Property 5.2.14. This property says that the computed risk magnitude can only be lower if one moves beyond the closest ancestor node, and hence, there is no need to do so if the magnitude at the closest ancestor node already falls below the defined threshold. And further, it says that it is equally important to continue matching until the closest ancestor node is reached, since it represents the full extent of the knowledge gained by the adversary from seeing the query answer (any nodes in the extended twig that do not appear in the original twig are *not* part of the answer).

Taken together, these two observations establish the correctness of Alg. 5.2.3. □

5.2.5 Examples

We provide three examples of computing disclosure risk magnitude, illustrating cases where there is *total*, *partial*, and *no* disclosure. All three examples are based on Ex. 4.1.1, with the access control policy of Ex. 2.4.12 applied in each case.

Example 5.2.16 (Partial Disclosure)**.** *Ex. 4.1.1 demonstrated that knowledge of Alice's age alone was not sufficient for the adversary to uniquely identify Alice's patient record, but that, intuitively, a partial disclosure did take place since Bob's patient record could be dis-*

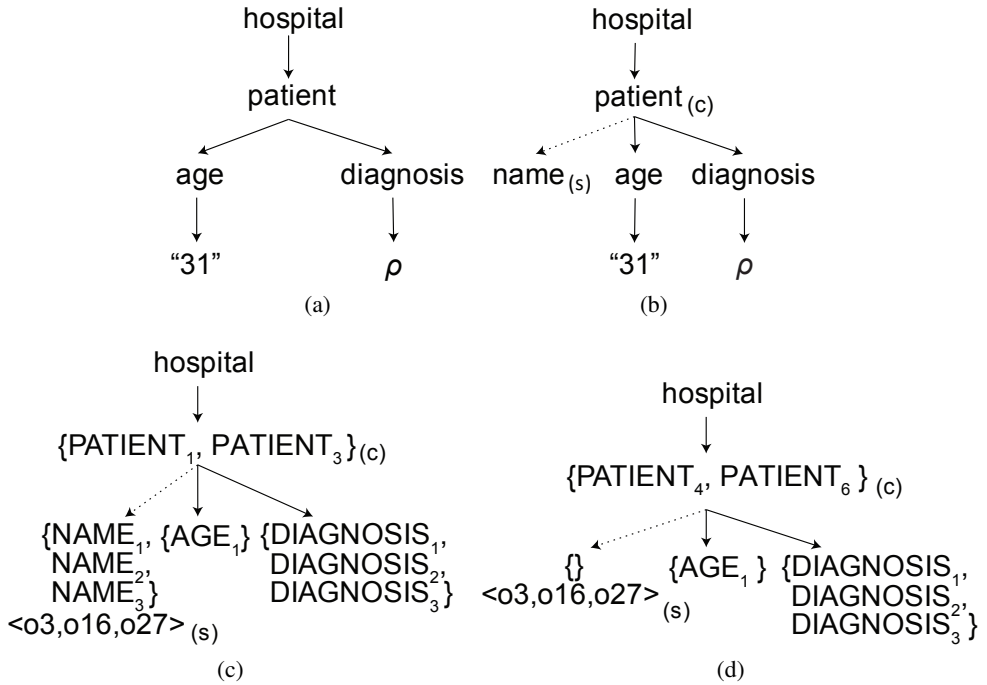


Figure 5.3: Example of a partial disclosure. (a) Twig τ formed from query; (b) Extended twig τ' formed by connected the sensitive twig node with its closest ancestor in the original twig τ; (c) Result of bottom-up matching of τ' with the database grammar G; (d) Result of bottom-up matching of τ' with the adversary's view grammar G_{AIns}.

qualified based on prior knowledge of Alice's age. We now reinvestigate this example using Alg. 5.2.3. In the first step, the query twig τ shown in Fig. 5.3a is formed from the query `/hospital/patient[age="31"]/diagnosis`. *The closest ancestor of the sensitive location path,* `/hospital/patient/name`, *in τ is* `/hospital/patient`, *so in the second step the extended twig τ' is formed by adding an edge connecting these two location paths, as shown in Fig. 5.3b. The sensitive location path,* `/hospital/patient/name`, *is marked with* `(s)`, *while* `/hospital/patient`, *its closest ancestor from the original twig, is marked with* `(c)`.

In the third step, shown in Fig. 5.3c, a bottom-up matching is conducted between τ' and the PRTG G (depicted in Fig. 5.1) defined over the entire database. In addition, the `name` *node in τ' is annotated with* `o3,o16,027`, *the inaccessible instances of* `/hospital/patient/name`. *Once the matching has been concluded at the closest ancestor twig node, the matching rules are* $\texttt{PATIENT}_1$ *and* $\texttt{PATIENT}_3$. *The fourth step*

Grammar Rules (R)			**Rule Prob. ($\mathbf{P}_G$)**
$\texttt{HOSPITAL}_2$	$\rightarrow$	hospital($\texttt{PATIENT}_4$, $\texttt{PATIENT}_5$, $\texttt{PATIENT}_6$)	1
$\texttt{PATIENT}_4$	$\rightarrow$	patient($\texttt{AGE}_1$, $\texttt{DIAGNOSIS}_1$, $\texttt{DOCTOR}_1$, $\texttt{DISCHARGE\_DATE}_1$, $\texttt{CARRIER}_1$)	1/3
$\texttt{PATIENT}_5$	$\rightarrow$	patient($\texttt{AGE}_2$, $\texttt{DIAGNOSIS}_2$, $\texttt{DOCTOR}_2$, $\texttt{CARRIER}_1$)	1/3
$\texttt{PATIENT}_6$	$\rightarrow$	patient($\texttt{AGE}_1$, $\texttt{DIAGNOSIS}_3$, $\texttt{DOCTOR}_3$, $\texttt{CARRIER}_2$)	1/3
$\texttt{AGE}_1$	$\rightarrow$	age("31")	2/3
$\texttt{AGE}_2$	$\rightarrow$	age("57")	1/3
$\texttt{DIAGNOSIS}_1$	$\rightarrow$	diagnosis("leukemia")	1/3
$\texttt{DIAGNOSIS}_2$	$\rightarrow$	diagnosis("pulmonary fibrosis")	1/3
$\texttt{DIAGNOSIS}_3$	$\rightarrow$	diagnosis("pneumonia")	1/3
$\texttt{DOCTOR}_1$	$\rightarrow$	doctor("House")	1/3
$\texttt{DOCTOR}_2$	$\rightarrow$	doctor("Mancini")	1/3
$\texttt{DOCTOR}_3$	$\rightarrow$	doctor("Cox")	1/3
$\texttt{DISCHARGE\_DATE}_1$	$\rightarrow$	discharge_date("23/08/05")	1
$\texttt{CARRIER}_1$	$\rightarrow$	carrier("Black Cross")	2/3
$\texttt{CARRIER}_2$	$\rightarrow$	carrier("White Shield")	1/3

Figure 5.4: Set of rules in G_{AIns} and their associated probabilities.

computes the entropy for the probability distribution defined over these rules in G as

$$H_{\{\tau,G\}} = \frac{1}{2}\log_2 2 + \frac{1}{2}\log_2 2 = 1\,.$$

The fifth step repeats the grammar rule matching of τ', this time against G_{AIns}, the PRTG defining the adversary's view of the database. The rules and associated probabilities for G_{AIns} are given in Fig. 5.4. As Fig. 5.3d illustrates, once the matching is completed for the closest ancestor (i.e., `/hospital/patient`) the candidate rules are $\texttt{PATIENT}_4$ and $\texttt{PATIENT}_6$. Since neither rule appears in G, each must be mapped to the shortest rule in G that contains its RHS. This leads to the mappings $\texttt{PATIENT}_4 \Rightarrow \texttt{PATIENT}_1$ and $\texttt{PATIENT}_6 \Rightarrow \texttt{PATIENT}_3$.

In the sixth step, the relative entropy calculation is carried out between the probability distributions for `PATIENT` rules induced by matching τ' against G (step 3) and G_{AIns} (step 5). The result is

$$KL(\tau', G||\tau', G_{AIns}) = \frac{1}{2}\log_2 1 + \frac{1}{2}\log_2 1 = 0\,.$$

Finally, the risk magnitude calculation is performed, resulting in

$$\mathcal{M}(\tau \rightsquigarrow /hospital/patient/name, G, G_{AIns}) = 1 - [(1+0)/\log_2 3] \approx 0.369\,.$$

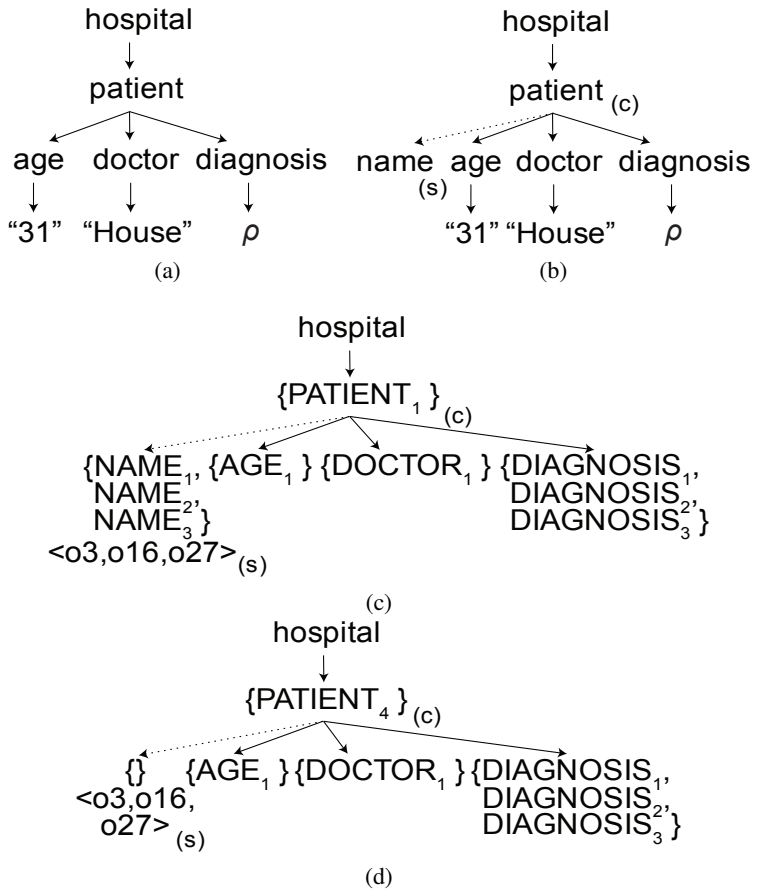


Figure 5.5: Example of a total disclosure. (a) Twig τ formed from query; (b) Extended twig τ' formed by connected the sensitive twig node with its closest ancestor in the original twig τ; (c) Result of bottom-up matching of τ' with the database grammar G; (d) Result of bottom-up matching of τ' with the adversary's view grammar G_{AIns}.

This indicates that there has indeed been a partial disclosure. □

Example 5.2.17 (Total Disclosure)**.** *Ex. 4.1.1 pointed out that intuitively, the adversary's knowledge of Alice's age and doctor is enough to uniquely identify her patient record. We carry out this computation using our measure. The twig τ formed from query* `/hospital/patient[age="31" and doctor="House"]/diagnosis` *is shown in Fig. 5.5a. Next, it is extended to form a path between the sensitive location path,* `/hospital/patient/name`*, and its closest ancestor within τ,* `/hospital/patient`*. The resulting twig τ' is shown in Fig. 5.5b. In the third step, a bottom-up matching between τ' and G, the database PRTG, is carried out, with the results as shown in Fig. 5.5c.* $\texttt{PATIENT}_1$ *is the only matching rule at the closest ancestor (*`patient`*) node, and the* `name`

Grammar Rules (R)			**Rule Prob. ($\mathbf{P}_G$)**
$\texttt{HOSPITAL}_3$	$\rightarrow$	hospital($\texttt{PATIENT}_1$)	1
$\texttt{PATIENT}_1$	$\rightarrow$	patient($\texttt{NAME}_1$, $\texttt{AGE}_1$, $\texttt{DIAGNOSIS}_1$, $\texttt{DOCTOR}_1$, $\texttt{DISCHARGE\_DATE}_1$, $\texttt{CARRIER}_1$)	1
$\texttt{AGE}_1$	$\rightarrow$	age("31")	1
$\texttt{DIAGNOSIS}_1$	$\rightarrow$	diagnosis("leukemia")	1
$\texttt{DOCTOR}_1$	$\rightarrow$	doctor("House")	1
$\texttt{DISCHARGE\_DATE}_1$	$\rightarrow$	discharge_date("23/08/05")	1
$\texttt{CARRIER}_1$	$\rightarrow$	carrier("Black Cross")	1

Figure 5.6: Set of rules in G_{Alice} and their associated probabilities.

node is annotated with the three instances of `/hospital/patient/name`*, since all are inaccessible to* `AIns` *under the ACP. In the fourth step, twig entropy is carried out for the matching* `PATIENT` *rules; since there is only a single matching rule,* $\texttt{PATIENT}_1$*, the result is*

$$H_{\{\tau',G\}} = 1 \log_2 1 = 0\,.$$

Bottom-up matching is then conducted between τ' and G_{AIns}, the PRTG defined over the database view for the `AIns` *role (cf. Fig. 5.4). As shown in Fig. 5.5d, only* $\texttt{Patient}_4$ *emerges as a matching rule at the closest ancestor node in τ'. Since this rule does not exist in the original PRTG G, it is replaced with the shortest rule in G containing its RHS, which is* $\texttt{PATIENT}_1$*. The rule matching procedures have yielded the same set of matching* `PATIENT` *rules for G and G_{AIns}, so the relative entropy calculation is*

$$KL(\tau', G||\tau', G_{AIns}) = 1 \log_2 1 = 0\,.$$

In the final step, the disclosure risk magnitude is computed as

$$\mathcal{M}(\tau \rightsquigarrow /hospital/patient/name, G, G_{AIns}) = 1 - [(0+0)/\log_2 3] = 1\,,$$

indicating that a total disclosure has taken place. □

Example 5.2.18 (No Disclosure)**.** *To demonstrate an example where a query presents no disclosure risk, we consider the query* `/hospital/patient` *issued by federated ID* `Alice`*. We recall that under the designated ACP,* `Alice` *has access to the entirety of her*

own patient record, and no parts of other patient records. The PRTG G_{Alice} defining her view of the database consists of the rules shown in Fig. 5.6.

Notice that under this scenario, `/hospital/patient` *functions as the query answer, the sensitive location path, and as its own closest ancestor in the original query twig τ, as shown in Fig. 5.7a. For this reason, there is no need to extend τ. Bottom-up matching of τ with the database PRTG G starts and ends at the* `patient` *node, and the result is shown in Fig. 5.7b. While all three patient rules are included (since this node represents the query answer over the entire database), the node is also annotated with* `o15,o26` *to indicate that these two instances of* `/hospital/patient` *are inaccessible to* `Alice`*. Computing the twig entropy for* `PATIENT` *rules results in*

$$H_{\{\tau,G\}} = \frac{1}{3}\log_2 3 + \frac{1}{3}\log_2 3 + \frac{1}{3}\log_2 3 = \log_2 3 \approx 1.585\,.$$

When τ is matched against G_{Alice} (shown in Fig. 5.7c), the result consists of the only patient rule in this PRTG, $\texttt{PATIENT}_1$*. There is no need to perform rule replacement, since this rule also appears in G. Computation of the relative entropy of* `PATIENT` *rules returned by each matching operation yields*

$$KL(\tau, G||\tau, G_{Alice}) = \frac{1}{3}\log_2 \frac{1}{3} + \frac{1}{3}\log_2 \frac{1/3}{0} + \frac{1}{3}\log_2 \frac{1/3}{0} = \infty\,.$$

The risk magnitude is

$$\mathcal{M}(\tau \rightsquigarrow /hospital/patient, G, G_{Alice}) \approx 1 - [(1.585 + \infty)/\log_2 3] = -\infty\,,$$

indicating that there is no disclosure risk represented by the query. This result makes sense on an intuitive level, as Alice gains no extra knowledge – not even an indication of the cardinality of patient records in the database – from the query answer. □

5.2.6 Optimizing the Disclosure Risk-Data Utility Tradeoff

A coarse solution for removing instance-based risks is to reject outright any query result presenting a disclosure risk above the ϵ threshold. Doing so can greatly hamper the utility of the database, as it prevents a user from seeing even the "risk-free" portions of the query result. A more amenable solution might seek to maximize data utility by only pruning the

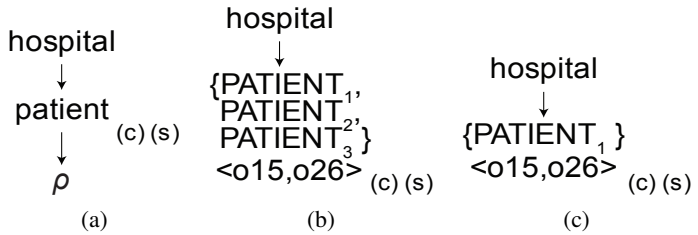


Figure 5.7: Example of no disclosure. (a) Twig τ formed from query; (b) Result of bottom-up matching of τ with the database grammar G; (c) Result of bottom-up matching of τ with the adversary's view grammar G_{Alice}.

minimal number of nodes from the query result required to reduce the disclosure risk below ϵ, starting with those nodes that are deemed to be of least importance to the user.

In formalizing this solution, we assume that there is a total function $utility_q : q \times U \rightarrow \mathbb{Q}^+$ assigning a user-specific importance score to each node in the query q. Additionally, a function for evaluating the degree of additional disclosure risk posed by including an individual node $n \in q$ in the adversary's view is given by

$$risk(n) = \mathcal{M}(\tau \rightsquigarrow s, G, G_A) - \mathcal{M}(\tau \rightsquigarrow s, G, \{G_A - n\})$$

where τ is the twig capturing q, s is the sensitive node, and G and G_A represent the PRTGs defined over the entire database and the adversary's view, respectively. As a notational convenience, $\{G_A - n\}$ is used to indicate the PRTG that generates the adversary's view excluding the subtree rooted by node n. The goal is then to determine the *maximally-useful secure query result* from the space of all trees formed by deleting one or more nodes from the original query result, namely the twig $\hat{\tau}$ that achieves the highest cumulative utility score (calculated by summing the utility score for each node contained in $\hat{\tau}$) while falling below the specified disclosure risk threshold (i.e., $\mathcal{M}(\hat{\tau} \rightsquigarrow s, G, G_A) \leq \epsilon$).

Unfortunately, obtaining such a *maximally-useful secure query result* proves to be intractable.

Theorem 5.2.19. *Computing the maximally-useful secure query result for an arbitrary query q is* **NP**-hard.

Proof. We state the decision version of the problem as determining whether there exists any solution twig that falls below the specified disclosure risk threshold, while achieving a

cumulative utility score of at least S, where $S \in \mathbb{Q}^+$ is an additional parameter. Henceforth, we will abbreviate this problem as MUSQR-DP.

To show that MUSQR-DP is in **NP**, we observe that a candidate twig $\hat{\tau}$ consisting of m nodes can be verified in $O(m)$ time by first summing the m node utility scores and ensuring that the result is at least S, and then calculate $\mathcal{M}(\hat{\tau} \rightsquigarrow s, G, G_A)$ (which also takes $O(m)$ time, cf. Property 5.2.12) and compare the result to ϵ.

NP-completeness is demonstrated by a polytime reduction from 0-1 KNAPSACK in which each of the k knapsack items $i_1, \ldots, i_k$ are mapped to nodes $n_1, \ldots, n_k$ in $\hat{\tau}$, $risk(n_j) = weight(i_j)$ and $utility_Q(n_j, u) = value(i_j)$, $j = 1 \ldots, k$, ϵ is set to the weight threshold value W, and S is set to the value goal G. Under this reduction, the only MUSQR-DP instances yielding a "yes" answer correspond to instances of 0-1 KNAPSACK that would yield a "yes" answer from a 0-1 KNAPSACK solver.

Since MUSQR-DP is **NP**-complete, it follows that the corresponding optimization problem is **NP**-hard. □

While this result is negative, it is likely that polynomial-time approximation algorithms for finding a secure query result that comes close to maximizing usefulness exist. One obvious possibility would be to design such an algorithm based on an approximation algorithm for 0-1 KNAPSACK. We leave this as future work.

5.2.7 Discussion

We now relate our query time disclosure detection algorithm to the various disclosure risk types identified within the related works of Sec. 4.4.1. As shown in the previous chapter, our design time detection algorithm is capable of identifying all disclosures caused by the parent-child constraints of the Yang and Li model [119]. Our query time detection algorithm is capable of identifying the ancestor-descendant constraints of Yang and Li: given such a relationship between node types a and b, an adversary with full access to a and partial access to b issuing the query $a//b$ would yield a twig matching in which *none* of the a-rules within the database grammar have been disqualified, which in turn would cause Alg. 5.2.3 to decide that answering the query would cause a total disclosure.

Disclosures caused by Yang and Li's notion of functional dependencies between node

types would also be identified as a total disclosures by Alg. 5.2.3. Recalling that such a functional dependency mandates that all nodes of a specific type a that possess a child path p_1 must also possess a second child path p_2, we observe that in each such circumstance the database grammar must possess a grammar rule $a_i \Rightarrow a(r_1, \ldots, r_2)$, where r_1 and r_2 designate the grammar rules deriving paths p_1 and p_2, respectively, and the probability assigned to this rule would be 1. Issuing a query that includes a/p_1 and a/p_2 would yield a twig matching in which only rule a_i would constitute a matching for the twig position associated with a. This in turn would cause the twig entropy to be 0, once again resulting in the classification of a total disclosure.

Finally, we recall that the disclosure risks in the model of Farkas et al [52] are formed by constraints between node types, expressed as Horn clauses. Such constraints are essentially equivalent to the functional dependency constraints of Yang and Li, and would be handled in the same manner by Alg. 5.2.3. In comparison to the approach of Hashimoto et al, we note that our approach to query time risk analysis is more fine-grained in the sense that we move beyond considering the number of candidate answers to a secret query to also incorporate the differing probabilities with which each such candidate appears in the database. Further, we are able to improve scalability by reasoning over a compressed representation of the database contents (i.e., probabilistic regular tree grammars) in place of using tree transducers.

5.3 Conclusions

This chapter presented a query-time solution to detection and removal of instance-based disclosure risks on federated data. In the next chapter, we describe an efficient implementation of this approach, along with experimental results indicating the scalability of the implementation.

Input: Root node $subtreeRootNode$ of an XML subtree; N, a set of non-terminal symbols; $\mathcal{F}$, a set of terminal symbols; R, a set of grammar rules

Output: Generated grammar rule $\langle headSymbol, rhsList \rangle$

```
1  headSymbol ← "";
2  rhsList ← ∅;
3  rhsList.add(subtreeRootNode.getType() + "(");
4  if subtreeRootNode.isAttribute() then
5  |  rhsList.add(subTreeRootNode.getValue());
6  else
7  |  foreach childNode ∈ subtreeRootNode.children() do
8  |  |  if childNode.getType() ≠ PCDATA then
9  |  |  |  childRule ← BuildGrammarRule(childNode);
10 |  |  |  rhsList.add(childRule.getHeadSymbol());
11 |  |  else
12 |  |  |  rhsList.add(childNode.getValue());
13 |  |  |  F.add(childNode.getValue());
14 |  |  end
15 |  end
16 end
17 rhsList.add(")");
18 existingRule ← R.get(subtreeRootNode.getType(), rhsList);
19 if existingRule ∉ R then
20 |  headSymbol ← R.add(subtreeRootNode.getType(), rhsList, 1);
21 |  N.add(headSymbol);
22 |  F.add(rhsList.getSignature());
23 else
24 |  headSymbol ← existingRule.getHeadSymbol());
25 |  R.incrementRuleOccurrenceCount(existingRule);
26 end
27 return ⟨headSymbol, rhsList⟩;
```

Algorithm 5.2.2: Subprocedure `BuildGrammarRule` for generating a PRTG rule from an input XML subtree T.

Input: $\texttt{CXP}^{[=]}$ query q; federated ID f; threshold value ϵ; sensitive node s; database PRTG G; disclosure risks table $\mathcal{R}$

Output: "no" if the computed risk magnitude is less than ϵ; "yes" otherwise

```
1  τ ← FormTwigFromQuery(q);
2  if IsPotentialRisk(τ, s) = false then
3  |  return "no"
4  else
5  |  τ′ ← ExtendTwigToSensitiveNode(s);
6  |  MP_G ← PerformBottomUpRuleMatching(τ′, G);
7  |  H ← ComputeTwigEntropy(MP_G, G);
8  |  MP_f ← PerformBottomUpRuleMatching(τ′, G_f);
9  |  MP_f ← MP_f/MP_G;
10 |  KL ← ComputeRelativeEntropy(MP_G, MP_f);
11 |  M ← ComputeRiskMagnitude(H, KL, s);
12 |  if M ≥ ε then
13 |  |  return "yes"
14 |  else
15 |  |  return "no"
16 |  end
17 end
```

Algorithm 5.2.3: Query-time risk analysis procedure.

Chapter 6

Implementation and Experimental Validation of Instance-Based Disclosure Risk Detection and Removal

6.1 Implementation

Our implementation was built using Java 1.6. Persistent storage of XML database trees and grammars was carried out using Berkeley DB Java Edition 4.0. Fig. 6.1 illustrates the architecture of the implementation. The end user interacts with the *database manager* component, which is responsible for invoking the *access control enforcement* module in order to determine which nodes the user has access to, and for using this information to determine whether the issued query is allowable according to the ACP. For allowable queries, the database manager also interacts with the *disclosure risks monitor* to evaluate the level of risk posed by the query, and the *Core XPath query processor* to obtain the query result.

In order to evaluate instance-based disclosure risks, the disclosure risk monitor performs Alg. 5.2.3. It interacts with the *PRTG storage* module, which manages persistent grammar storage in Berkeley DB. Table 6.1 specifies the data structures used to maintain the PRTGs defined over a database. The `Grammar Rules` table stores the right-hand side (RHS) of each grammar rule, keyed by the identifier of that rule. Each RHS expression is stored as a sequence of tokens, where each token can be a numeric value (referring to the identifier of another rule) or a sequence of textual characters, representing a node label or PCDATA. `Rule Counts` stores the number of times each rule is applied within the current database.

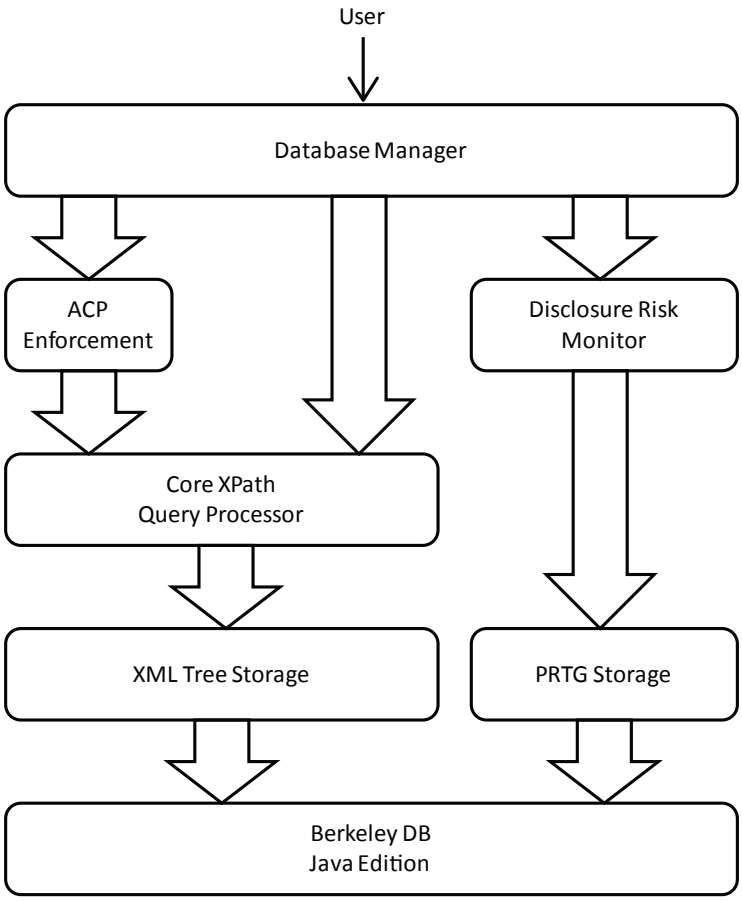


Figure 6.1: System architecture of the prototype.

The `Rule-By-Root-Node` index allows for a fast lookup of the grammar rule used to derive the subtree rooted at the key node. `Rule References` provides a method for quickly determining which rules contain a reference to the keyed rule within their respective right-hand sides. Additionally, a secondary index is maintained on `GrammarRules`, allowing a rule to be quickly located based on the contents of its RHS.

The query processor builds an execution plan, then retrieves needed information from the Berkeley DB structures associated with XML database storage via calls to the *XML tree storage* component. Table 6.2 lists the various data structures used to store the structure and content of an XML database: in particular, `First Child` stores a reference to the node identifier of each internal node's first child; `Next Sibling` maintains a pointer to the identifier of the key node's next sibling; `Content` stores the textual content belonging to each node; `Node Types` indicates the type of each node (i.e., element, attribute, text, or

Table 6.1: Data Structures Used to Store Database Grammars.

Name	Key	Value	Access Method
Grammar Rules	rule_id	RHS	BTree
Rule Counts	rule_id	occurrence_count	BTree
Rule-By-Root-Node	node_id	rule_id	BTree
Rule References	referred_rule_id	referring_rule_id	BTree

Table 6.2: Data Structures Used to Store XML Trees.

Name	Key	Value	Access Method
First Child	node_id	child_id	BTree
Next Sibling	node_id	sibling_id	BTree
Content	parent_id	value	BTree
Node Types	node_type	node_id	BTree
Node Names	node_name	node_id	BTree

document root); and Node Names records the label assigned to each element and attribute node.

6.2 Incremental Grammar Maintenance

A key challenge in implementing our approach within a dynamic database lies in keeping the generated grammars up-to-date as database contents are modified. We consider the complexity of performing the various primitive update operations first discussed in Sec. 2.4 and repeated below.

Append(x, y)**:** This operation inserts a node y as the right-most child of node x.

InsertBefore(x, y)**:** In this operation, node y is inserted as the left sibling of node x.

Delete(x)**:** This operation results in the deletion of the subtree rooted at node x.

We first investigate the worst-case time complexity of updating stored PRTGs in terms of these primitive operations, using the data structures in Table 6.2 and Table 6.1.

Claim 6.2.1. *Assume that the data structures in Table. 6.2 and Fig. 6.1 permit insertions, deletions, and lookups in amortized $O(\log r)$ time, where r is the number of records contained in the corresponding data structure. Let h denote the maximum height of the XML tree T, and n denote the number of nodes in T. Incremental grammar modification in response to each* **Append**(x, y) *or* **InsertBefore**(x, y) *operation takes $O(h \log n)$ time. Mod-*

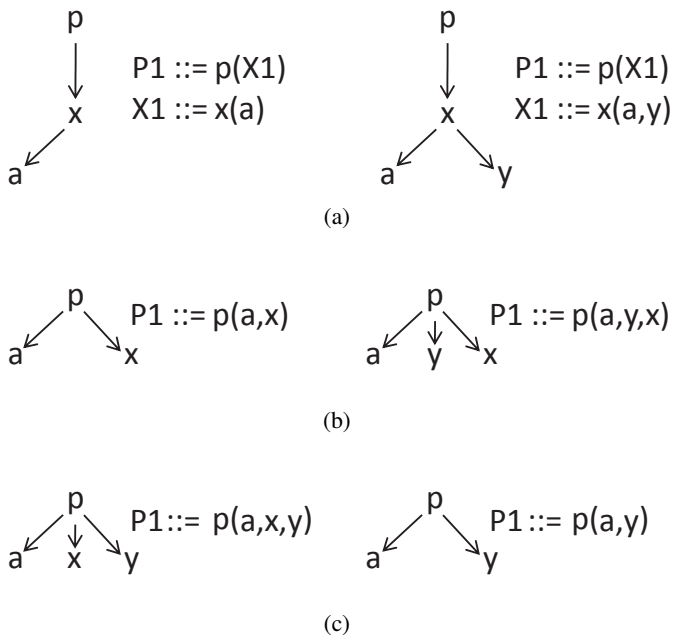


Figure 6.2: Incremental grammar maintenance in response to (a) **Append**(x, y), (b) **InsertBefore**(x, y), and (c) **Delete**(x) operations.

ifications for each **Delete**(x) *operation require* $O(n \log n)$ *time.*

Proof. Fig. 6.2 depicts each operation. In the case of **Append**(x, y) operations (Fig. 6.2a), the following steps must be performed: (1) the count for the assigned grammar rule must be decremented, requiring a lookup in the `RulesByRootNode` table to find the rule pertaining to x's subtree. Then, a lookup and insertion are performed in the `RuleCounts` table to decrement the occurrence count for this rule; (2) a lookup is performed on the secondary index for `GrammarRules` to see if an existing rule with the modified RHS exists; if so, a lookup and insertion are performed on the `RuleCounts` table, and otherwise, a new record is inserted into `GrammarRules` describing the new rule, and an associated entry is added to `RuleCounts` to initialize the occurrence count for the new rule to 1. An insertion is then performed on the `Rules-By-Root-Node` table to add/update the entry keyed by x to store the rule id pertaining to the modified RHS. The cumulative cost of these updates is $O(\log n)$. Next, the rules for each ancestor of x must be updated so that their RHS's contain a reference to the new rule for x in place of the obsolete rule. At each ancestor, this consists of performing the sequence of steps listed previously, and also updating the `Rule References` table to ensure that each reference to an outdated rule

File Name	Size (B)	Scale Factor
xmark512K.xml	570761	0.005
xmark4M.xml	4129094	0.035
xmark32M.xml	33379489	0.285
xmark256M.xml	268422667	2.291

Table 6.3: XMark documents used in grammar generation experiment.

for the child node is replaced with a reference to the new rule. Therefore, processing at each ancestor also costs $O(\log n)$ time. In the worst case, x is a leaf node in T, and therefore h ancestors must be processed, leading to the worst-case cost of $O(h \log n)$. The rationale for **InsertBefore(**x, y**)** operations is similar.

In the case of a **Delete(**x**)** operation, intuitively the worst case is when x is the root node. This requires deleting all grammar rules and associated entries. From Property 5.2.4, this necessitates the deletion of $O(n)$ rules, with each deletion coming at a cost of $O(\log n)$. □

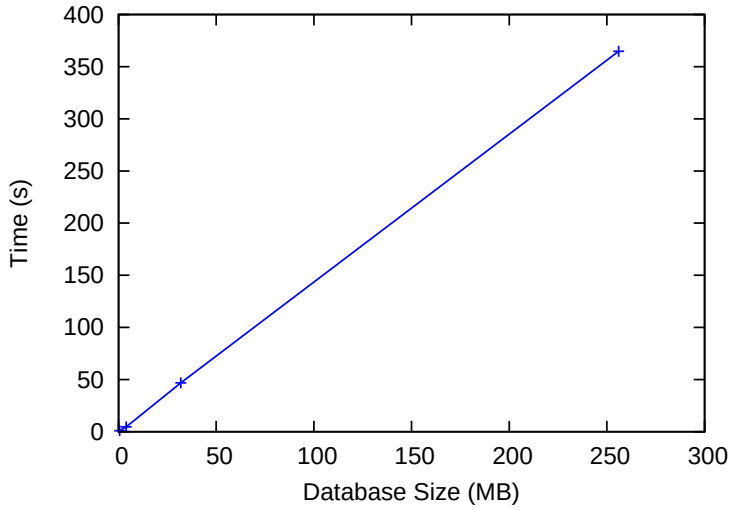
6.3 Experiments

In this section, we provide the results of experiments designed to evaluate the efficency and scalability of the proposed solutions in Chapter 5. All experiments were performed on an entry-level server with a 2.2 GHz QuadCore AMD Opteron processor. For each experiment, the BerkeleyDB cache size was set to 1 GB, while the maximum Java heap size was capped at 4 GB.

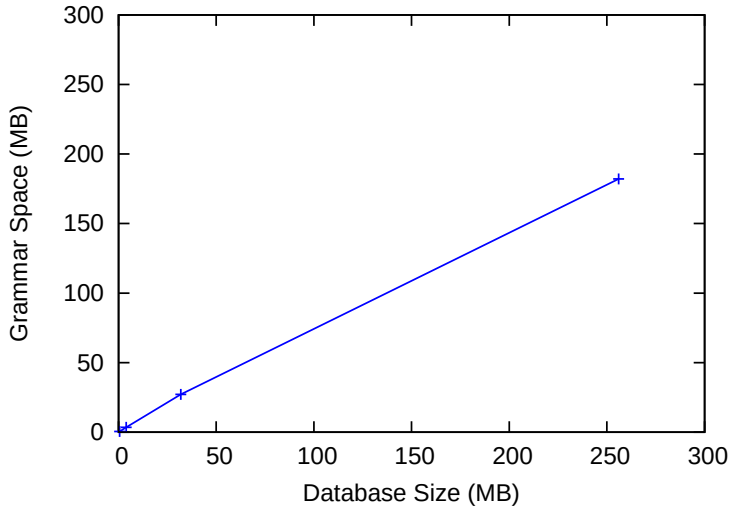
6.3.1 Grammar Generation

XMark Documents

In the first experiment, we sought to test the scalability – in terms of both time and space – of grammar generation. To do so, we performed a series of batch operations of increasing size, each consisting of several node insertion operations which are valid against the XMark [108] DTD. Each of the batch operations is executed on an initially empty database. Fig. 6.3a plots the size of each batch operation (x-axis) against the corresponding time requirements (y-axis) for generating the grammar; for each batch operation, the average time requirement from five trials is reported. As expected, the time costs increase linearly with respect to the number of nodes inserted. In a similar manner, Fig. 6.3b reports the space re-



(a)



(b)

Figure 6.3: Scalability of grammar construction in terms of (a) time and (b) space.

quirements for storing the grammar induced by each of the batch operations, and indicates that the grammar storage costs increase linearly with the number of insertions performed.

Other Document Types

Since XMark databases are randomly generated, it is not too surprising that the resulting lack of regularity in structure and content causes the grammar size to approach that of the database itself. To get a more complete sense of the typical space requirements for grammars, we also conducted a second experiment that compared the sizes of generated grammars against database size for several XML documents obtained from the University

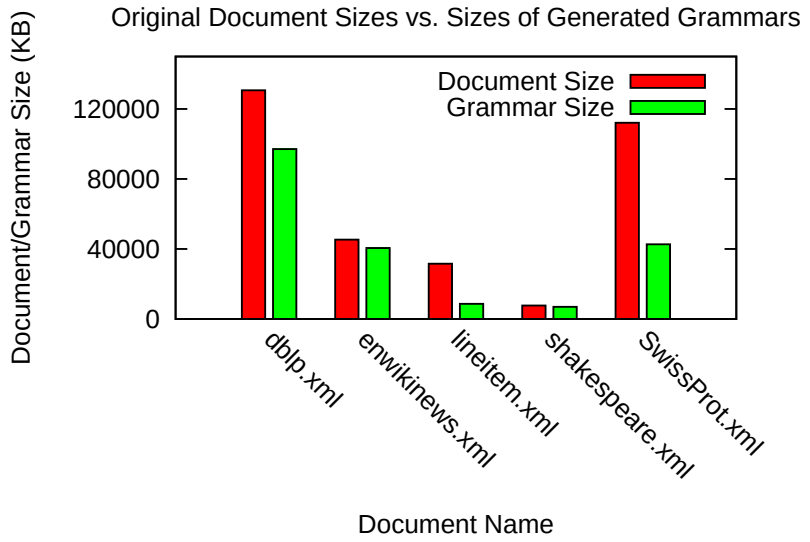


Figure 6.4: Comparison of original XML document sizes with corresponding PRTG sizes.

of Washington XML Repository[1] and the Wratislavia XML Corpus[2]. Fig. 6.4 plots the original size of each document together with the corresponding grammar size. As expected, the size difference between the original document and the corresponding PRTG is smaller for "document-centric" XML files such as Shakespeare plays and Wikipedia entries than it is for the "data-centric" files in the experiment (DBLP entries, the SwissProt genetic database, and an XML encoding of the lineitem table from the TPC-H benchmark).

Grammar Maintenance

While the previous two experiments measured the cumulative costs of generating a database grammar from an initially empty database, the next set of experiments measured the time requirements for performing incremental grammar maintenance operations on a pre-existing grammar. A separate experiment was carried out for $\mathbf{Append}(x, y)$, $\mathbf{Insert}(x, y)$, and $\mathbf{Delete}(x)$ operations, in which the worst case scenario for each operation – as described in the proof of Claim 6.2.1 – was modeled. That is, each appended node was attached as a child of a leaf node in the existing XML tree, each inserted node was similarly added as a left-child of a leaf node, and each delete operation was carried out on the root node of an XML tree containing the specified number of nodes. Under these conditions, the height of

[1] http://www.cs.washington.edu/research/xmldatasets/
[2] http://www.ii.uni.wroc.pl/~inikep/research/Wratislavia/

Database	Size (MB)
dblp2000	144.698
dblp1990	185.652
dblp1980	197.540
dblp1970	201.726

Table 6.4: Sizes of the databases in the large DBLP files set.

each tree, h, is set to equal n, the number of nodes in the XML tree. Fig. 6.5 displays the results for each experiment. The results are consistent with the expected $O(n \log n)$ cost for each operation.

6.3.2 Query-Time Risk Analysis

DBLP Files – Large Set

The third experiment aimed to capture the impact upon query performance posed by query-time risk analysis. Throughout this experiment, we used a set of DBLP [3] databases formed by extracting articles based on publication year (i.e. `dblp2000` includes all articles published in 2000 or later, `dblp1990` all articles since 1990, and so on). The characteristics of these databases are shown in Table 6.4. In addition, we defined a federated ID `Editor` with a single ACP rule allowing access only to those articles not authored by `E. F. Codd`, and employed the fixed query `/child::*` to retrieve the entire view available to `Editor`.

The results of this experiment, indicating the total query response times using a "cold" and "warm" cache, are shown in Fig. 6.6a. Fig. 6.6b provides an alternative analysis of the results by depicting the contributing time costs of each phase of query-time analysis using a warm cache (the results prove similar for a cold cache). In each case, rule matching dominates the overall cost, consuming between 81-83% of the overall query response time and 94-97% of the time requirement for query-time disclosure analysis. The twig formation and extension phases had a non-significant cost (less than 0.01 seconds each per trial), and are excluded from the figure for clarity. Finally, it is interesting to note that the total time cost of performing query-time risk analysis ranges from 5.59 to 6.74 times the comparative cost of answering the query without performing any risk analysis.

[3] `http://www.informatik.uni-trier.de/~ley/db/`

Database	Size (MB)
dblp2010	13.225
dblp2009	31.931
dblp2008	50.560
dblp2007	68.165
dblp2006	84.169
dblp2005	98.463
dblp2004	110.096
dblp2003	120.693
dblp2002	129.707
dblp2001	137.621

Table 6.5: Sizes of the databases in the small DBLP files set.

Q1: /child::site
Q2: /child::site/child::regions
Q3: /child::site/child::regions/child::europe
Q4: /child::site/child::regions/child::europe/child::item
Q5: /child::site/child::regions/child::europe/child::item/ child::mailbox
Q6: /child::site/child::regions/child::europe/child::item/ child::mailbox/child::mail

Table 6.6: Queries issued against xmark256.xml in the experiment testing the impact of query size on query-time analysis performance.

DBLP Files – Small Set

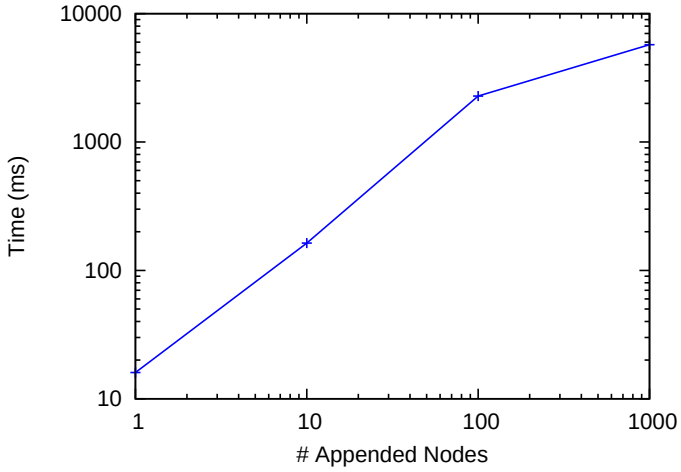
We also repeated the previous experiment using a smaller set of 11 DBLP files; the smallest contains articles published since 2010, the second smallest articles published since 2009, and so on down to the largest file, containing articles published since 2001. The sizes of these files are shown in Table. 6.5. The results of this experiment complement those of the first phase of the previous experiment by supplying timing results against a larger set of data points. The results are depicted in Fig. 6.7a and Fig. 6.7b. These results are consistent with those of the previous experiment, showing that query response times for both "cold" and "warm" cache queries scale linearly with increasing database size, and additionally, that the various phases of query-time risk analysis each scale linearly with increasing database size.

XMark Documents

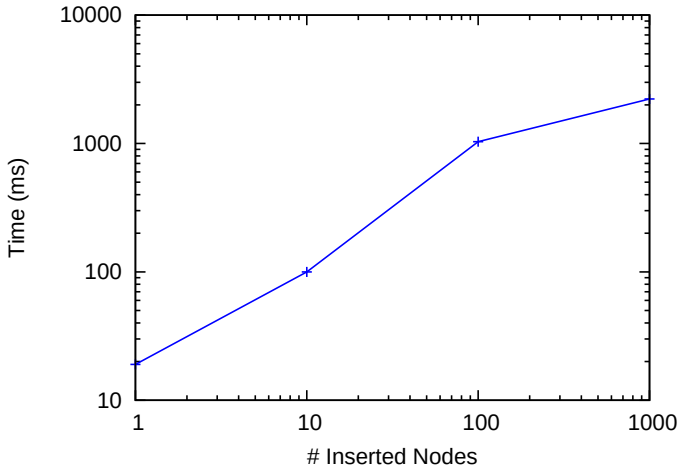
Finally, we carried out the above experiments on the set of XMark documents described in Table 6.3. This time, we utilized an access control policy based on a user named Buyer,

with access to all parts of the document except for `type` elements having a content value of `Featured`. The results are depicted in Fig. 6.8. As with the previous data sets, increasing database size results in a scalable increase in query response time, as shown in Fig. 6.8a. To better cope with the wide range in document sizes (and resulting variety in query response times), we utilize a log-log scale. For a similar reason, we represent the breakdown of time requirements for query response time without risk analysis, rule matching, and risk magnitude calculation using separate plots within a log-log scale in Fig. 6.8b, instead of employing a stacked bar chart as was done in the previous experiments. We note from this figure that the time requirements for performing rule matching and magnitude calculation are outweighed by the query response time without analysis for the three smaller documents, yet the cost of rule matching does become dominant for the largest XMark document.

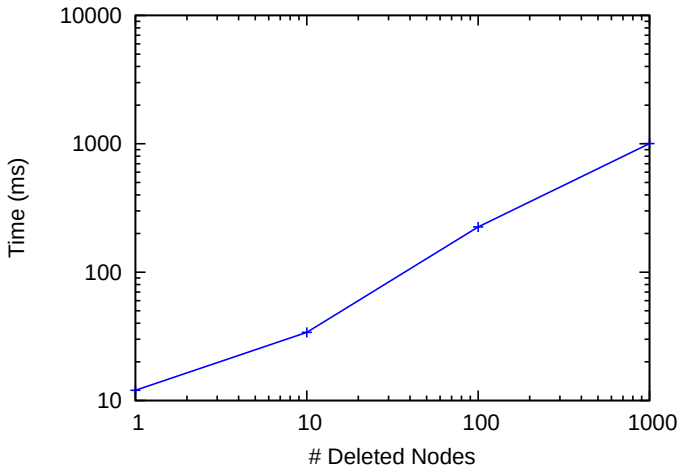
XMark documents are also typically deeper than DBLP documents. Taking advantage of this, we also conducted an analysis of the impact of query size (measured as the number of location steps within the query) on the time requirements for risk analysis. Fig. 6.8c depicts the results for test queries ranging from 1 to 6 location steps; these queries are listed in Table 6.6.



(a)

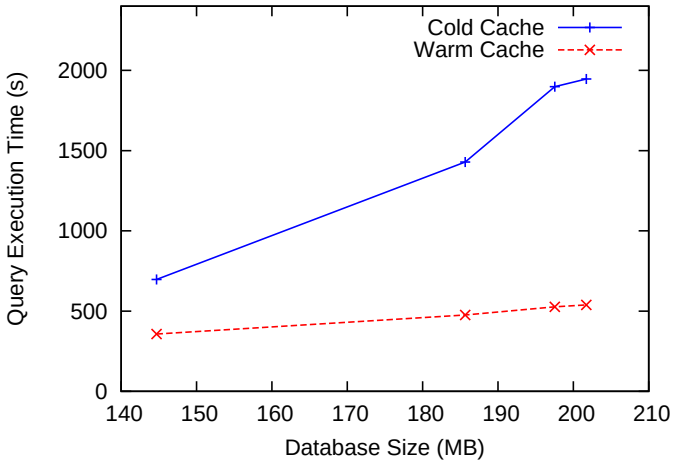


(b)

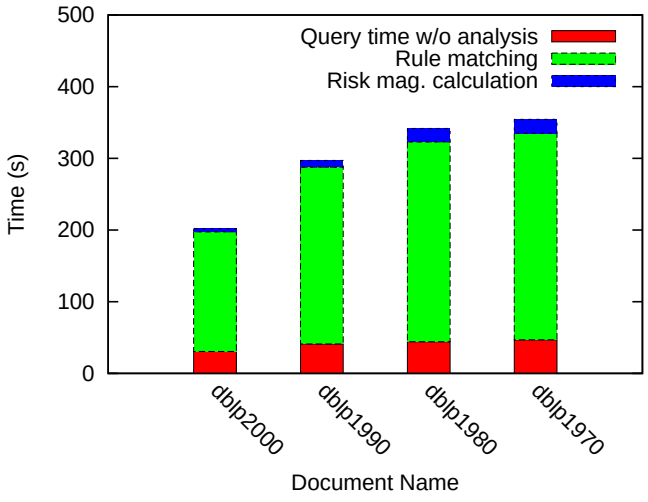


(c)

Figure 6.5: Worst case grammar maintenance costs for (a) $\mathbf{Append}(x, y)$, (b) $\mathbf{Insert}(x, y)$, and (c) $\mathbf{Delete}(x)$ operations.

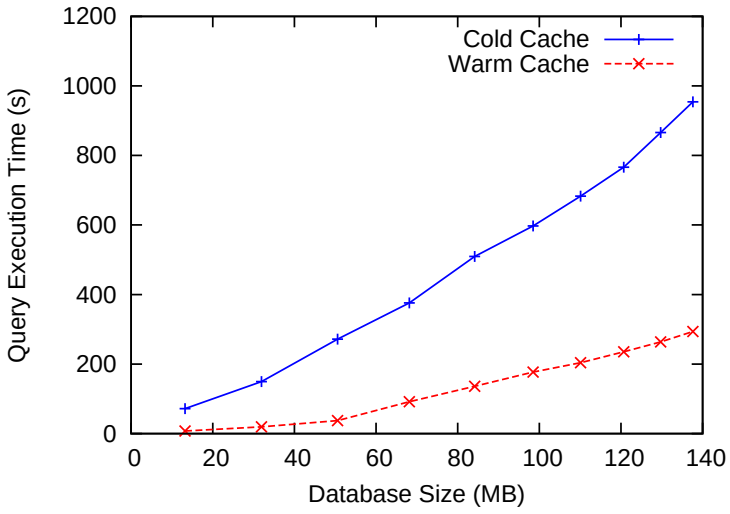


(a)

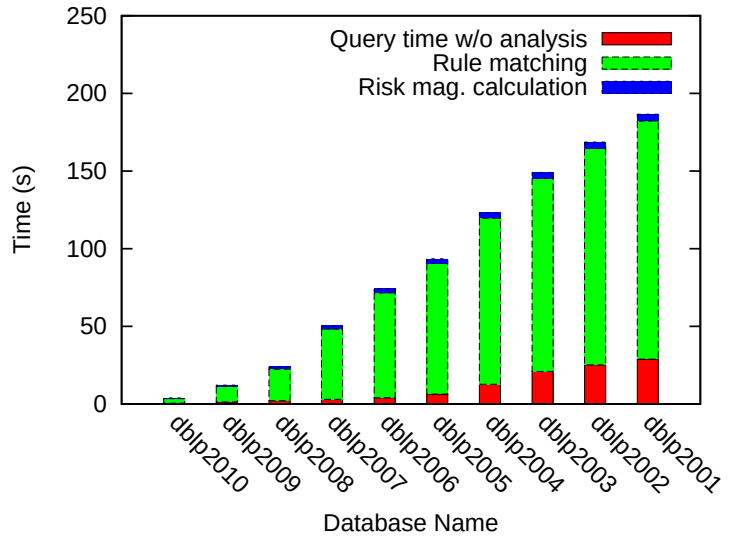


(b)

Figure 6.6: Time requirements for query-time risk analysis as database size is varied for the large DBLP files.

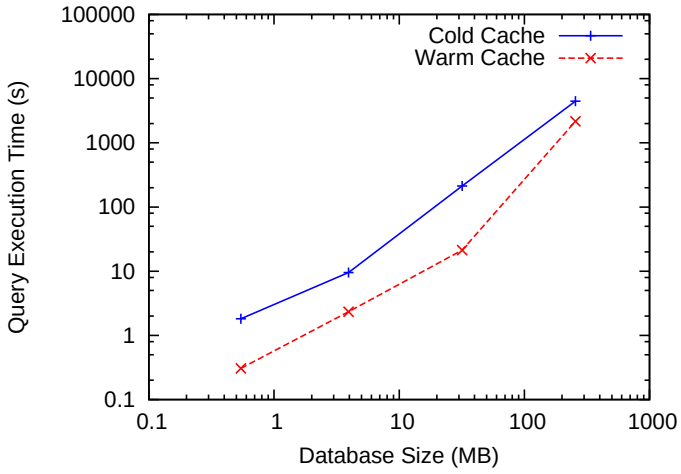


(a)

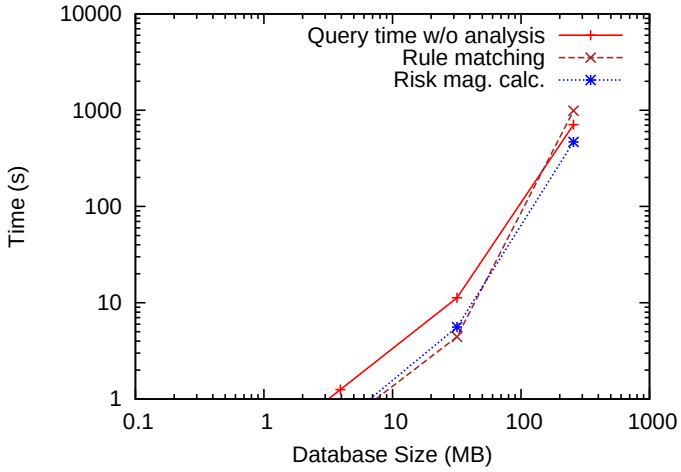


(b)

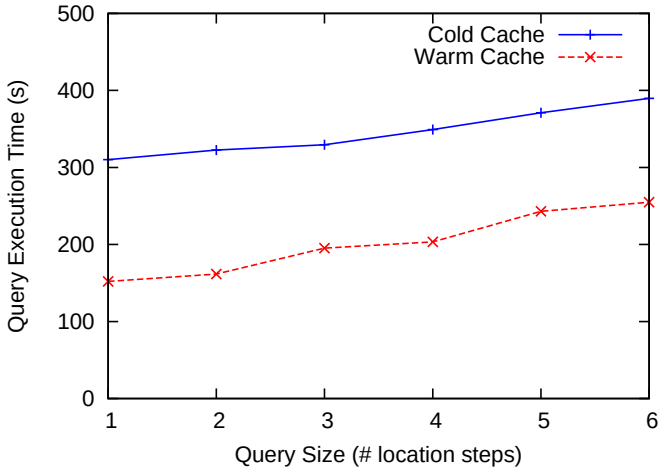
Figure 6.7: Time requirements for query-time risk analysis as database size is varied for the smaller DBLP files.



(a)



(b)



(c)

Figure 6.8: Time requirements for query-time risk analysis as database and query sizes are varied for the XMark documents.

Chapter 7

Conclusions

This chapter concludes the thesis by first summarizing the most important contributions made within preceding chapters. This is followed by a discussion of potential directions for future research.

7.1 Review of Contributions

In Chapter 3, we provided several complexity results related to the enforcement of multiple access control policies defined over individual source databases at the data federation level.

- We showed that for arbitrary inputs consisting of a defined XML publishing function, an access control policy, a relational schema, and an identity mapping function, it is always possible to derive a *secure publishing function* capable of preserving the access control policy semantics of the local data source over XML data exported to the federation. Further, this function can be efficiently derived using a polynomial-time function we provide.

- We also considered the problem of verifying whether a federated policy satisfies the semantics of the local access control policies defined at each data source. We considered two problem variants: static verification examines the secure publishing function against the original relational schema, access control policy, XML publishing function, and identity mapping function and produces a result that holds for any instance database conformant to the supplied schema. For all but the simplest cases, static verification proves to be undecidable, while for the restricted case in which the

XML publishing function contains only conjunctive queries, the problem remains intractable. For dynamic verification, in which the output of the secure publishing function on a specific database instance is compared against that of the original inputs, the operation requires at least exponential time (with respect to the database size) for recursive secure publishing functions, while in the case of non-recursive functions, it can be carried out in polynomial time.

- We examined the minimization problem for secure publishing functions: given an input function, return the smallest function with equivalent semantics. After establishing the equivalence of this problem with the static verification problem (and therefore establishing its hardness), we considered a variant of the problem that used a weaker notion of equivalence, and showed that the resulting complexities were one level lower in the polynomial hierarchy than the corresponding static verification complexities.

Finally, we contributed an algorithm for expressing a secure publishing function in the eXtensible Access Control Markup Language (XACML), allowing our framework to more easily be incorporated within existing architectures.

Motivated by the realization that even a well-crafted access control policy cannot adequately guard against indirect accesses to data caused by information leakage, Chapters 4- 6 outlined a scalable framework for detecting and removing disclosure risks on XML databases. In particular, these chapters provided the following contributions:

- We provided a classification of disclosure risks relevant to XML databases based on whether they can be detected at database design time (*schema-based risks*) or are specific to the current database contents and therefore, can only be accurately measured during the evaluation of a user query (*instance-based risks*)

- We then presented a two-phase solution for detecting and removing both types of disclosure risks. During the first phase, design-time risks are identified and removed via a DTD analysis procedure that also serves to identify potential instance-based disclosure risks. In the second phase, a query-specific analysis is performed to evaluate the

current disclosure risks relevant to the current query, using a novel measure based on information theory.

- We demonstrated that probabilistic regular tree grammars provide a means for maintaining an accurate probabilistic model of an XML database's content. Such a model is essential for computing the magnitude of instance-based disclosure risks using our proposed measure.

- We described a prototype implementation of the two-phase approach to disclosure risk detection and removal, and performed an experimental evaluation that demonstrated the scalability of this implementation, as database and query sizes are increased.

7.2 Future Directions

We now discuss potential avenues for future research that would extend the results of the thesis.

7.2.1 Access Control Policy Translation

Although Chapter 3 provided theoretical analysis and algorithmic solutions to several issues related to access control translation in data federations, there remain several areas for future work. One can examine other formulations of the verification problem, including a variation covering scenarios in which the existing translated ACP is specified not as a secure publishing transducer, but rather in a declarative language such as XACML. Another area for investigation is the use of alternative access control models on the relational side, such as role-based and mandatory models. The substitution of one of these models could affect our existing results on translation, verification, and minimization of ACPs. For example, it is established that role-based models are strictly more powerful than either mandatory or discretionary models, being capable of simulating any policy originally expressed in either of these models [98].

A different articulation of the policy verification problem relaxes the assumption that the list of federated users is *fixed* (e.g., to account for users being added to and removed

from the federation over time). In such cases, the bitstrings of two different secure publishing transducers are incompatible. Thus, both the static and dynamic verification solutions we provide must be changed: instead of directly comparing bitstrings, we must resort to reasoning about the ACPs encoded in each transducer. While this assumption is not likely to change our existing complexity results (as verification is dominated by the cost of materializing XML trees, not by the expense of access bitstring comparisons), efficient algorithms for handling such cases for the simplest S.P.T. classes are still needed.

Finally, while Chapter 3 focused on cases where access control enforcement is centralized, other possibilities exist, including secure publishing solutions in which access control is implemented over a single copy of an XML document via encryption. Integrating our framework into such applications requires an algorithm for producing partially-encrypted XML documents by applying a secure publishing transducer to an instance database. A large-scale case study, involving the practical implementation of these ideas in realistic scenarios, would constitute an interesting area for future work. Such a study would also prove valuable in identifying theoretical constructs in our framework which do not necessarily admit an efficient implementation in practice (such as access bitstrings, which may incur excessive storage costs in applications where the set of federated IDs is very large), and in discovering more suitable alternatives.

7.2.2 XML Disclosure Control

The design-time disclosure risks considered in Chapter 4 were limited to those posed by a specific XML schema language (DTDs). Other schema languages such as XML Schema [48], Schematron [67], and Relax NG [68] are more expressive than DTDs [92], and hence, often furnish the adversary with additional prior knowledge about the database such as key and occurrence constraints. An area of future work involves classifying the additional potential risks posed by each of these alternative schema languages and proposing efficient measures for their detection and removal.

One extension of this work would involve using the approach presented in this paper to suggest methods for "fine-tuning" an existing ACP in order to minimize the degree of potential leakage. Another would be to investigate whether it would be possible to devise

a grammar-based representation of the database that allows an expressive range of queries to be anwered directly on it, thereby removing the need to store the original database along with its PRTG. Existing work on grammar-based XML compression suggests that there is some promise in this direction [24, 25, 56, 79]. We intend to investigate efficient approximation algorithms for improving the utility of query results without violating the specified leakage threshold. Finally, the prototype implementation presented in Chapter 6 provides a starting point for more sophisticated solutions supporting multi-user scenarios.

7.2.3 Unexplored Topics

In Chapters 4 and 5, we presented a technique for preventing information disclosure at the federated level. While it is true that some risks can only be detected at the federated level (such as those caused by an adversary performing additional inference based on linking the federated data exported by multiple data sources, when data from any one of these sources would not constitute a risk), it is obviously more desirable from the standpoint of data source autonomy to perform risk analysis at the *local* level. This would reduce each data source's reliance on federation security administrators to accurately detect and report back all possible risks. Within the context of our framework, this would correspond to determining whether the locally defined publishing function serves to leak any information about local data that is not exported to the federation. This problem would be made more interesting by the fact that it is possible to represent a set of relational data in many different ways as XML; therefore, some publishing functions will likely leak more information than others. An investigation into what features of publishing functions make them more vulnerable (or conversely, more secure) would be beneficial to the designers of such functions. Measures for quantifying the amount of leakage and algorithms for automatically suggesting repairs to insecure functions form two possible contributions resulting from such a study.

Bibliography

[1] Martin Abadi and Bogdan Warinschi. Security analysis of cryptographically controlled access to XML documents. *Journal of the ACM*, 55(2):108–117, May 2008.

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, Reading, MA, USA, 1995.

[3] Charu C. Aggarwal and Philip S. Yu. Privacy-preserving data mining: A survey. In *Handbook of Database Security*, chapter 18, pages 431–460. Springer, New York, NY, USA, 2007.

[4] A. V. Aho, Y. Sagiv, and J. D. Ullman. Equivalences among relational expressions. *SIAM Journal of Computing*, 8(2):218–246, 1979.

[5] Anne Anderson. Hierarchical resource profile of XACML v2.0. OASIS Standard, February 2005. `http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-hier-profile-spec-os.pdf`.

[6] Stephanos Androutsellis-Theotokis and Diomidis Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Computing Surveys*, 36(4):335–371, December 2004.

[7] S. Banerjee, V. Krishnamurthy, M. Krishnaprasad, and R. Murthy. Oracle8i - the XML enabled data management system. In *Proceedings of the 16th International Conference on Data Engineering*, pages 561–568, San Diego, CA, 2000. IEEE Computer Society.

[8] Denilson Barbosa, Juliana Freire, and Alberto O. Mendelzon. Designing information-preserving mapping schemes for XML. In *Proceedings of the 31st International Conference on Very Large Data Bases*, pages 109–120, New York, NY, USA, 2005. VLDB Endowment.

[9] Denilson Barbosa, Gregory Leighton, and Andrew Smith. Efficient incremental validation of XML documents after composite updates. In *Proceedings of the Fourth International XML Database Symposium*, pages 107–121, Seoul, Korea, 2006. Springer.

[10] D. E. Bell and L. J. LaPadula. Secure computer systems: mathematical foundations. Technical Report Technical Report ESD-TR-278, vol. 1, The MITRE Corporation, 1973.

[11] Michael Benedikt, Chee Yong Chan, Wenfei Fan, Rajeev Rastogi, Shihui Zheng, and Aoying Zhou. DTD-directed publishing with attribute translation grammars. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 838–849, New York, NY, USA, 2002. VLDB Endowment.

[12] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, and Jérôme Siméon. *XML Path Language (XPath) 2.0*. World Wide Web Consortium, January 23 2007. `http://www.w3.org/TR/2007/REC-xpath20-20070123/`.

[13] Elisa Bertino, Barbara Carminati, and Elena Ferrari. A temporal key management scheme for secure broadcasting of XML documents. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 31–40, New York, NY, USA, 2002. ACM.

[14] Elisa Bertino, Silvana Castano, and Elena Ferreri. Securing XML documents with Author-X. *IEEE Internet Computing*, 5(3):21–31, May/June 2001.

[15] Elisa Bertino and Elena Ferrari. Secure and selective dissemination of XML documents. *ACM Transactions on Information and System Security*, 5(3):290–331, 2002.

[16] Elisa Bertino and Ravi S. Sandhu. Database security-concepts, approaches, and challenges. *IEEE Transactions on Dependable and Secure Computing*, 2(1):2–19, 2005.

[17] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML query language. W3C Recommendation, January 2007. `http://www.w3.org/TR/xquery/`.

[18] Piero Bonatti, Sabrina De Capitani di Vimercati, and Pierangela Samarati. An algebra for composing access control policies. *ACM Transactions on Information and System Security*, 5(1):1–35, 2002.

[19] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, and David Orchard. Web services architecture. W3C Recommendation, February 2004. `http://www.w3.org/TR/ws-arch/`.

[20] Luc Bouganim, Francois Dang Ngoc, and Philippe Pucheral. Dynamic access-control policies on XML encrypted data. *ACM Transactions on Information and System Security*, 10(4):1–37, 2008.

[21] Tim Bray, Jean Paoli, C.M. Sperberg-McQueen, Eve Maler, and Francois Yergeau. *Extensible Markup Language (XML) 1.0 (Fourth Edition)*. World Wide Web Consortium, September 29 2006. `http://www.w3.org/TR/2006/REC-xml-20060816/`.

[22] Alexander Brodsky, Csilla Farkas, and Sushil Jajodia. Secure databases: Constraints, inference channels, and monitoring disclosures. *IEEE Transactions on Knowledge and Data Engineering*, 12(6):900–919, 2000.

[23] Glenn Bruns and Michael Huth. Access-control policies via belnap logic: effective and efficient composition and analysis. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium*, pages 163–176, Washington, DC, USA, 2008. IEEE Computer Society.

[24] Peter Buneman, Martin Grohe, and Christoph Koch. Path queries on compressed XML. In *Proceedings of the 29th International Conference on Very Large Data Bases*, pages 141–152, New York, NY, USA, 2003. VLDB Endowment.

[25] Giorgio Busatto, Markus Lohrey, and Sebastian Maneth. Efficient memory representation of XML documents. In *Proceedings of the 10th International Symposium on Database Programming Languages*, pages 199–216, New York, NY, USA, 2005. Springer.

[26] Canadian Department of Justice. Personal Information Protection and Electronic Documents Act. `http://laws.justice.gc.ca/en/P-8.6/text.html`, April 2000.

[27] Alberto Caprara, Matteo Fischetti, and Dario Maio. Exact and approximate algorithms for the index selection problem in physical database design. *IEEE Transactions on Knowledge and Data Engineering*, 7(6):955–967, 1995.

[28] A.K. Chandra and P.M. Merlin. Optimal implementation of conjunctive queries in relational databases. In *Proceedings of the 9th ACM Symposium on the Theory of Computing*, pages 77–90, New York, USA, 1977. ACM.

[29] C. Chekuri and A. Rajaraman. Conjunctive query containment revisited. In *Proceedings of the 6th International Conference on Database Theory*, pages 56–70, London, UK, 1997. Springer-Verlag.

[30] Rada Chirkova, Alon Y. Halevy, and Dan Suciu. A formal perspective on the view selection problem. *The VLDB Journal*, 11:216–237, November 2002.

[31] SungRan Cho, Sihem Amer-Yahia, Laks V. S. Lakshmanan, and Divesh Srivastava. Optimizing the secure evaluation of twig queries. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 490–501, New York, NY, USA, 2002. VLDB Endowment.

[32] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[33] Douglas Comer. The difficulty of optimum index selection. *ACM Transactions on Database Systems*, 3(4):440–445, 1978.

[34] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available at: `http://www.grappa.univ-lille3.fr/tata`, 2007.

[35] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. Wiley-Interscience, Hoboken, NJ, USA, 2nd edition, 2006.

[36] Jason Crampton. Applying hierarchical and role-based access control to XML documents. In *Proceedings of the 2004 Workshop on Secure Web Services*, pages 37–46, New York, NY, USA, 2004. ACM.

[37] Nilesh Dalvi, Gerome Miklau, and Dan Suciu. Asymptotic conditional probabiliities for conjunctive queries. In *Proceedings of the 10th International Conference on Database Theory*, pages 289–305, New York, NY, USA, 2005. Springer.

[38] Ernesto Damiani, Sabrina De Capitani di Vimercati, Stefano Paraboschi, and Pierangela Samarati. Securing XML documents. In *Proceedings of the 7th International Conference on Extending Database Technology*, pages 121–135, London, UK, 2000. Springer-Verlag.

[39] Ernesto Damiani, Sabrina De Capitani di Vimercati, Stefano Paraboschi, and Pierangela Samarati. A fine-grained access control system for XML documents. *ACM Transactions on Information and System Security*, 5(2):169–202, May 2002.

[40] B.A. Davey and Hilary A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, UK, 2002.

[41] Steven Dawson, Sabrina De Capitani di Vimercati, and Pierangela Samarati. Specification and enforcement of classification and inference constraints. In *IEEE Symposium on Security and Privacy*, pages 181–195, Los Alamitos, CA, USA, 1999. IEEE Computer Society.

[42] Sabrina de Capitani di Vimercati and Pierangela Samarati. Access control in federated systems. In *Proceedings of New Security Paradigms Workshop*, pages 87–99, New York, NY, USA, 1996. ACM.

[43] Dorothy E. Denning and Jan Schlörer. Inference controls for statistical databases. *IEEE Computer*, 16(7):69–82, July 1983.

[44] Sabrina De Capitani di Vimercati and Pierangela Samarati. An authorization model for federated systems. In *4th European Symposium on Research in Computer Security*, pages 99–117, New York, NY, USA, 1996. Springer.

[45] Cynthia Dwork. Ask a better question, get a better answer: A new approach to private data analysis. In *11th International Conference on Database Theory*, pages 18–27, New York, NY, USA, 2007. Springer.

[46] Donald Eastlake and Joseph Reagle. XML encryption syntax and processing. W3C Recommendation, December 2002. `http://www.w3.org/TR/xmlenc-core/`.

[47] Ronald Fagin. On an authorization mechanism. *ACM Transactions on Database Systems*, 3(3):310–319, September 1978.

[48] David C. Fallside and Priscilla Walmsley. *XML Schema Part 0: Primer Second Edition*. World Wide Web Consortium, October 28 2004. `http://www.w3.org/TR/xmlschema-0/`.

[49] Wenfei Fan. XML publishing: Bridging theory and practice. In *Proceedings of the 11th International Conference on Database Programming Languages*, pages 1–16, London, UK, 2007. Springer-Verlag.

[50] Wenfei Fan, Chee-Yong Chan, and Minos Garofalakis. Secure XML querying with security views. In *Proceedings of the SIGMOD International Conference on Management of Data*, pages 587–598, New York, NY, USA, 2004. ACM.

[51] Wenfei Fan, Floris Geerts, and Frank Neven. Expressiveness and complexity of XML publishing transducers. In *Proceedings of the 26th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 83–92, New York, NY, USA, 2007. ACM.

[52] Csilla Farkas, Alexander Brodsky, and Sushil Jajodia. Unauthorized inferences in semistructured databases. *Information Sciences*, 176(22):3269–3299, 2006.

[53] Csilla Farkas and Sushil Jajodia. The inference problem: A survey. *SIGKDD Explorations*, 4(2):6–10, 2002.

[54] Mary F. Fernandez, Yana Kadiyska, Dan Suciu, Atsuyuki Morishima, and Wang Chiew Tan. SilkRoute: A framework for publishing relational data in XML. *ACM Transactions on Database Systems*, 27(4):438–493, December 2002.

[55] David F. Ferraiolo, D. Richard Kuhn, and Ramaswamy Chandramouli. *Role-Based Access Control*. Computer Security Series. Artech House, Norwood, MA, 2003.

[56] Barbara Fila and Siva Anantharaman. Automata for positive Core XPath queries on compressed documents. In *13th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, pages 467–481, New York, NY, USA, 2006. Springer.

[57] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York, NY, USA, 1979.

[58] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient algorithms for processing XPath queries. *ACM Transactions on Database Systems*, 30(2):444–491, June 2005.

[59] Patricia P. Griffiths and Bradford W. Wade. An authorization mechanism for a relational database system. *ACM Transactions on Database Systems*, 1(3):242–255, September 1976.

[60] John Hale and Sujeet Shenoi. Catalytic inference analysis: Detecting inference threats due to knowledge discovery. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 188–199, Los Alamitos, CA, USA, 1997. IEEE Computer Society.

[61] Alon Halevy, Anand Rajaraman, and Joann Ordille. Data integration: the teenage years. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 9–16, New York, NY, USA, 2006. VLDB Endowment.

[62] Kenji Hashimoto, Fumikazu Takasuka, Kimihide Sakano, Yasunori Ishihara, and Toru Fujiwara. Verification of the security against inference attacks on XML databases. In *Proceedings of the 10th Asia-Pacific Web Conference on Progress in WWW Research and Development*, pages 359–370, Berlin, Germany, 2008. Springer-Verlag.

[63] Thomas H. Hinke, Harry S. Delugach, and Randall P. Wolf. Protecting databases from inference attacks. *Computers and Security*, 16(8):687–708, 1997.

[64] J. E. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In Z. Kohavi and A. Paz, editors, *Theory of Machines and Computations*, pages 189–196. Academic Press, London, England, 1971.

[65] IBM. *Label-based Access Control (LBAC) Overview*, 2008. `http://publib.boulder.ibm.com/infocenter/db2luw/v9/topic/com.ibm.db2.udb.admin.doc/doc/c0021114.htm`.

[66] Norbik Bashah Idris, W. A. Gray, and R. F. Churchhouse. Providing dynamic security control in a federated database. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 13–23, New York, NY, USA, 1994. VLDB Endowment.

[67] International Standards Organization. *ISO/IEC 19757-3:2006 Information technology – Document Schema Definition Language (DSDL) – Part 3: Rule-based validation – Schematron*, June 1 2006. `http://www.schematron.com/`.

[68] International Standards Organization. *ISO/IEC 19757-2:2008 Document Schema Definition Language (DSDL) – Part 2: Regular-grammar-based validation – RELAX NG*, December 15 2008. `http://www.relaxng.org/`.

[69] S. Jajodia, R. Sandhu, and B. Blaustein. Solutions to the polyinstantiation problem. In M.A. Abrams et al., editor, *Information Security: An Integrated Collection of Essays*, volume 1, pages 493–529. IEEE Computer Society, Los Alamitos, California, USA, 1994.

[70] Dirk Jonscher and Klaus R. Dittrich. An approach for building secure database federations. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 24–35, New York, NY, USA, 1994. VLDB Endowment.

[71] Daniel Kifer. Attacks on privacy and deFinetti's theorem. In *Proceedings of the 35th SIGMOD International Conference on Management of Data*, pages 127–138, New York, NY, USA, 2009. ACM.

[72] Ralph Kimball and Margy Ross. *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling*. Wiley, Hoboken, NJ, USA, 2nd edition, 2002.

[73] Benny Kimelfeld and Yehoshua Sagiv. Twig patterns: From XML trees to graphs. In *Proceedings of the 9th International Workshop on the Web and Databases*, New York, NY, USA, 2006. ACM. Online proceedings.

[74] Anthony Klug. On conjunctive queries containing inequalities. *Journal of the ACM*, 35(1):146–160, January 1988.

[75] Michiharu Kudo and Satoshi Hada. XML document security based on provisional authorization. In *Proceedings of the 7th ACM Conference on Computer and Communications Security*, pages 87–96, New York, NY, USA, 2000. ACM.

[76] Gregory Leighton. Preserving SQL access control policies over published XML data. In *Proceedings of the 2009 EDBT/ICDT Workshops*, pages 185–192, New York, NY, USA, 2009. ACM.

[77] Gregory Leighton and Denilson Barbosa. Access control policy translation and verification within heterogeneous data federations. In *Proceedings of the 15th ACM Symposium on Access Control Models and Technologies*, pages 173–182, New York, NY, USA, 2010. ACM.

[78] Gregory Leighton and Denilson Barbosa. Access control policy translation, verification, and minimization within heterogeneous data federations. In press, 2012.

[79] Gregory Leighton, Jim Diamond, and Tomasz Müldner. AXECHOP: A grammar-based compressor for XML. In *Proceedings of the IEEE Data Compression Conference*, page 467, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[80] Leonid Libkin. Expressive power of SQL. *Theoretical Computer Science*, 296(3):379–404, 2003.

[81] Bo Luo, Dongwon Lee, Wang-Chien Lee, and Peng Liu. QFilter: Fine-grained runtime XML access control via NFA-based query rewriting. In *Proceedings of the 13th ACM International Conference on Information and Knowledge Management*, pages 543–552, New York, NY, USA, 2004. ACM.

[82] Bo Luo, Dongwon Lee, and Peng Liu. Pragmatic XML access control using off-the-shelf RDBMS. In *Proceedings of the 12th European Symposium On Research In Computer Security*, pages 55–71, New York, NY, USA, 2007. Springer.

[83] Ashwin Machanavajjhala, Daniel Kifer, Johannes Gehrke, and Muthuramakrishnan Venkitasubramaniam. *l*-diversity: Privacy beyond *k*-anonymity. *ACM Transactions on Knowledge Discovery from Data*, 1(1):Online publication, 2007.

[84] Christopher D. Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing*. The MIT Press, Cambridge, MA, USA, 1999.

[85] Donald G. Marks. Inference in MLS database systems. *IEEE Transactions on Knowledge and Data Engineering*, 8(1):46–55, February 1996.

[86] Jim Melton and Alan R. Simon. *Understanding the New SQL: A Complete Guide*. Morgan-Kaufmann, San Francisco, 1993.

[87] Gerome Miklau. *Confidentiality and Integrity in Data Exchange*. PhD thesis, University of Washington, 2005.

[88] Gerome Miklau and Dan Suciu. Controlling access to published data using cryptography. In *Proceedings of the 29th International Conference on Very Large Data Bases*, pages 898–909, New York, NY, USA, 2003. VLDB Endowment.

[89] Gerome Miklau and Dan Suciu. A formal analysis of information disclosure in data exchange. *Journal of Computer and System Sciences*, 73(3):507–534, 2007.

[90] Matthew Morgenstern. Controlling logical inference in multilevel database systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 245–255, Los Alamitos, CA, USA, 1988. IEEE Computer Society.

[91] Tim Moses. Extensible access control markup language (XACML) version 2.0. OASIS Standard, February 2005. `http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf`.

[92] Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM Transactions on Internet Technology*, 5(4):1–45, 2005.

[93] Makoto Murata, Akihiko Tozawa, Michiharu Kudo, and Satoshi Hada. XML access control using static analysis. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, pages 73–84, New York, NY, USA, 2003. ACM.

[94] Qun Ni, Elisa Bertino, and Jorge Lobo. D-algebra for composing access control policy decisions. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, pages 298–309, New York, NY, USA, 2009. ACM.

[95] Matthias Nicola and Bert Van Der Linden. Native XML Support in DB2 Universal Database. In *Proceedings of the 31st International Conference on Very Large Data Bases*, pages 1164–1174, New York, NY, USA, 2005. ACM.

[96] Oracle Corporation. *Oracle 10g Release 2 Security*, 2005. `http://www.oracle.com/technology/deploy/security/database-security/pdf/twp_security_db_database_10gr2.pdf`.

[97] Oracle Database 11g XML DB technical overview, 2007. `http://www.oracle.com/technology/tech/xml/xmldb/Current/xmldb_11g_twp.pdf`.

[98] Sylvia Osborn, Ravi Sandhu, and Qamar Munawer. Configuring role-based access control to enforce mandatory and discretionary access control policies. *ACM Transactions on Information and System Security*, 3(2):85–106, 2000.

[99] Shankar Pal, Mark Fussell, and Irwin Dolobowsky. XML Support in Microsoft SQL Server 2005. `http://msdn2.microsoft.com/en-us/library/ms345117.aspx`, 2005.

[100] Christos H. Papadimitriou. On the complexity of unique solutions. *Journal of the ACM*, 31(2):392–400, 1984.

[101] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, Reading, Massachusetts, 1994.

[102] Arnon Rosenthal and Edward Sciore. Abstracting and refining authorization in SQL. In *Proceedings of the Secure Data Management Workshop*, pages 148–162, New York, NY, USA, 2004. Springer.

[103] Arnon Rosenthal and Marianne Winslett. Security of shared data in large systems: state of the art and research directions. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 962–964, New York, NY, USA, 2004. ACM.

[104] Yehoshua Sagiv and Mihalis Yannakakis. Equivalences among relational expressions with the union and difference operators. *Journal of the ACM*, 27(4):633–655, October 1980.

[105] Pierangela Samarati. Protecting respondents' identities in microdata release. *IEEE Transactions on Knowledge and Data Engineering*, 13(6):1010–1027, 2001.

[106] Pierangela Samarati and Sabrina De Capitani di Vimercati. Access control: policies, models, and mechanisms. In *IFIP WG 1.7 International School on Foundations of Security Analysis and Design: Tutorial Lectures*, pages 137–196, London, UK, 2000. Springer-Verlag.

[107] R. Sandhu and F. Chen. The multilevel relational data model. *ACM Transactions on Information and System Security*, 1(1):93–132, 1998.

[108] Albrecht Schmidt, Florian Waas, Martin Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. XMark: a benchmark for XML data management. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 974–985, New York, NY, USA, 2002. VLDB Endowment.

[109] Jayavel Shanmugasundaram, Eugene J. Shekita, Rimon Barr, Michael J. Carey, Bruce G. Lindsay, Hamid Pirahesh, and Berthold Reinwald. Efficiently publishing relational data as XML documents. *VLDB Journal*, 10(2-3):133–154, September 2001.

[110] Jayavel Shanmugasundaram, Kristen Tufte, Gang He, Chun Zhang, David DeWitt, and Jeffrey Naughton. Relational databases for querying XML documents: limitations and opportunities. In *Proceedings of the 25th International Conference on Very Large Data Bases*, pages 302–314, New York, NY, USA, 1999. VLDB Endowment.

[111] Claude E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423 and 623–656, July and October 1948.

[112] A.P. Sheth and J.A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):180–236, 1990.

[113] Latanya Sweeney. k-anonymity: A model for protecting privacy. *International Journal on Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(5):557–570, 2002.

[114] Jeroen Terstegge. Privacy in the law. In Milan Petković and Willem Jonker, editors, *Security, Privacy, and Trust in Modern Data Management*, chapter 2, pages 11–20. Springer, New York, NY, USA, 2007.

[115] Jeffrey D. Ullman. *Principles of Database Systems*. W. H. Freeman & Co., New York, NY, USA, 1983.

[116] Ron van der Meyden. The complexity of querying indefinite data about linearly ordered domains. *Journal of Computer and System Sciences*, 54(1):113–135, 1997.

[117] Hui Wang and Laks V.S. Lakshmanan. Efficient secure query evaluation over encrypted XML databases. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 127–138, New York, NY, USA, 2006. VLDB Endowment.

[118] Xiaokui Xiao and Yufei Tao. m-invariance: towards privacy preserving republication of dynamic datasets. In *Proceedings of the 33rd ACM SIGMOD International Conference on Management of Data*, pages 689–700, New York, NY, USA, 2007. ACM.

[119] Xiaochun Yang and Chen Li. Secure XML publishing without information leakage in the presence of data inference. In *Proceedings of the 30th International Conference on Very Large Data Bases*, pages 96–107, New York, NY, USA, 2004. VLDB Endowment.

[120] Shohei Yokoyama, Manabu Ohta, Kaoru Katayama, and Hiroshi Ishikawa. An access control method based on the prefix labeling scheme for XML repositories. In *Proceedings of the 16th Australasian Database Conference*, pages 105–113, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.